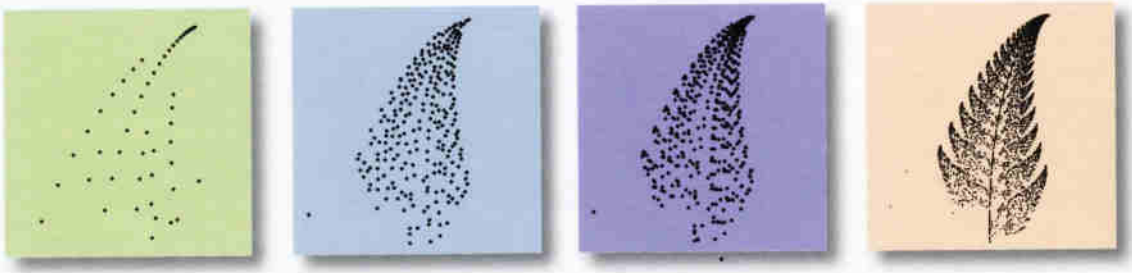
Программирование на языке Python



учебный курс

Роберт Седжвик • Кевин Уэйн • Роберт Дондеро

INTRODUCTION TO PROGRAMMING IN PYTHON

An Interdisciplinary Approach

Robert Sedgewick
Kevin Wayne
Robert Dondero

Princeton University

Addison-Wesley

New York • Boston • Indianapolis • San Francisco
Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ PYTHON:

учебный курс

Роберт Седжвик
Кевин Уэйн
Роберт Дондеро

Принстонский университет

ДИАЛЕКТИКА

Москва • Санкт-Петербург • Киев
2017

ББК 32.973.26-018.2.75
С28
УДК 681.3.07

Компьютерное издательство "Диалектика"

Зав. редакцией *С.Н. Тригуб*

Перевод с английского и редакция *В.А. Коваленко*

По общим вопросам обращайтесь в издательство "Диалектика" по адресу:
info@dialektika.com, http://www.dialektika.com

Седжвик, Роберт, **Уэйн**, Кевин, **Дондеро**, Роберт.

С28 Программирование на языке Python: учебный курс. : Пер. с англ. — СПб. : ООО "Альфа-книга", 2017. — 736 с. : ил. — Парал. тит. англ.

ISBN 978-5-9908462-1-0 (рус.)

ББК 32.973.26-018.2.75

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения издательства Addison-Wesley Publishing Company, Inc.

Authorized translation from the English language edition published by Addison-Wesley Publishing Company, Inc, Copyright © 2015.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the publisher.

Russian language edition is published by Dialektika Computer Books Publishing according to the Agreement with R&I Enterprises International, Copyright © 2017.

Книга напечатана согласно договору с ООО "ПРИМСНАБ".

Научно-популярное издание

Роберт Седжвик, Кевин Уэйн, Роберт Дондеро

Программирование на языке Python: учебный курс

Литературный редактор *И.А. Попова*
Верстка *О.В. Мишутина*
Художественный редактор *Е. П. Дынник*
Корректор *Л.А. Гордиенко*

Подписано в печать 21.09.2016. Формат 70x100/16
Гарнитура Times
Усл. печ. л. 59,34. Уч.-изд. л. 41,9
Тираж 400 экз. Заказ № 6235

Отпечатано в АО «Первая Образцовая типография»
Филиал «Чеховский Печатный Двор»
142300, Московская область, г. Чехов, ул. Полиграфистов, д.1
Сайт: www.chpd.ru, E-mail: sales@chpd.ru, тел. 8(499)270-73-59

ООО "Альфа-книга", 195027, Санкт-Петербург, Магнитогорская ул., д. 30

ISBN 978-5-9908462-1-0 (рус.)

ISBN 978-0-13-407643-0 (англ.)

© Компьютерное изд-во "Диалектика", 2017, перевод, оформление, макетирование

© Pearson Education, Inc., 2015

Содержание

Введение 11

Глава 1. Элементы программирования 19

1.1. Первая программа 20
1.2. Встроенные типы данных 31
1.3. Условные выражения и циклы 70
1.4. Массивы 110
1.5. Ввод и вывод 147
1.6. Случай из практики: случайная навигация по сайтам 193

Глава 2. Функции и модули 211

2.1. Определение функций 212
2.2. Модули и клиенты 247
2.3. Рекурсия 285
2.4. Случай из практики: просачивание 316

Глава 3. Объектно-ориентированное программирование 345

3.1. Использование типов данных 346
3.2. Создание типов данных 393
3.3. Разработка типов данных 440
3.4. Случай из практики: моделирование N тел 486

Глава 4. Алгоритмы и структура данных 499

4.1. Эффективность 500
4.2. Сортировка и поиск 542
4.3. Стеки и очереди 576
4.4. Таблицы идентификаторов 619
4.5. Случай из практики: феномен “тесного мира” 667

Контекст 709

Глоссарий 713

Функции API 719

Предметный указатель 729

Элементы программирования

Первая программа

Программа 1.1.1. Hello, World (helloworld.py) 21
Программа 1.1.2. Использование аргумента командной строки (useargument.py) 24

Встроенные типы данных

Программа 1.2.1. Пример конкатенации строк (ruler.py) 39
Программа 1.2.2. Целочисленные операторы (intops.py) 42
Программа 1.2.3. Операторы чисел с плавающей точкой (floatops.py) 45
Программа 1.2.4. Квадратичная формула (quadratic.py) 46
Программа 1.2.5. Високосный год (leapyear.py) 50

Условные выражения и циклы

Программа 1.3.1. Орел или решка (flip.py) 73
Программа 1.3.2. Ваш первый цикл (tenhellos.py) 76
Программа 1.3.3. Вычислительные степеней числа 2 (powersoftwo.py) 77
Программа 1.3.4. Ваш первый вложенный цикл (divisorpattern.py) 83
Программа 1.3.5. Гармонические числа (harmonic.py) 86
Программа 1.3.6. Метод Ньютона (sqrt.py) 87
Программа 1.3.7. Преобразование в двоичный формат (binary.py) 89
Программа 1.3.8. Модель разорения игрока (gambler.py) 91
Программа 1.3.9. Разложение на множители целых чисел (factors.py) 93

Массивы

Программа 1.4.1. Выборка без замены (sample.py) 122
Программа 1.4.2. Модель коллекции купонов (couponcollector.py) 126
Программа 1.4.3. Решето Эратосфена (primesieve.py) 128
Программа 1.4.4. Случайные блуждания без самопересечений (selfavoid.py) 136

Ввод и вывод

Программа 1.5.1. Создание случайной последовательности (`randomseq.py`) 149
Программа 1.5.2. Интерактивный пользовательский ввод (`twentyquestions.py`) 157
Программа 1.5.3. Среднее потока чисел (`average.py`) 159
Программа 1.5.4. Простой фильтр (`rangefilter.py`) 163
Программа 1.5.5. Стандартный ввод и фильтрация рисунка (`plotfilter.py`) 169
Программа 1.5.6. Вывод графика функции (`functiongraph.py`) 171
Программа 1.5.7. Прыгающий мяч (`bouncingball.py`) 176
Программа 1.5.8. Обработка цифрового сигнала (`playthattune.py`) 181

Случай из практики: случайная навигация по сайтам

Программа 1.6.1. Вычисление матрицы переходов (`transition.py`) 196
Программа 1.6.2. Моделирование случайной навигации (`randomsurfer.py`) 199
Программа 1.6.3. Смешение цепи Маркова (`markov.py`) 205

Функции и модули

Определение функций

Программа 2.1.1. Гармонические числа (повторно) (`harmonicf.py`) 215
Программа 2.1.2. Гауссовы функции (`gauss.py`) 225
Программа 2.1.3. Коллекционер купонов (повторно) (`coupon.py`) 226
Программа 2.1.4. Проигрывание мелодии (повторно) (`playthattunedeluxe.py`) 235

Модули и клиенты

Программа 2.2.1. Модуль Гауссовых функций (`gaussian.py`) 249
Программа 2.2.2. Пример клиента модуля Гауссовых функций (`gaussiantable.py`) 250
Программа 2.2.3. Модуль случайных чисел (`stdrandom.py`) 259
Программа 2.2.4. Система итерационных функций (`ifs.py`) 266
Программа 2.2.5. Модуль анализа данных (`stdstats.py`) 269
Программа 2.2.6. Рисование значений данных (`stdstats.py`, продолжение) 272
Программа 2.2.7. Исследование Бернулли (`bernoulli.py`) 274

Рекурсия

Программа 2.3.1. Алгоритм Евклида (`euclid.py`) 290
Программа 2.3.2. Ханойская башня (`towersofhanoi.py`) 293
Программа 2.3.3. Код Грея (`beckett.py`) 299
Программа 2.3.4. Рекурсивная графика (`htree.py`) 301
Программа 2.3.5. Броуновский мост (`brownian.py`) 303

Случай из практики: просачивание

Программа 2.4.1. Скаффолдинг просачивания (`percolation0.py`) 320
Программа 2.4.2. Обнаружение вертикального просачивания (`percolationv.py`) 322
Программа 2.4.3. Ввод-вывод просачивания (`percolationio.py`) 324
Программа 2.4.4. Клиент визуализации (`visualizev.py`) 325
Программа 2.4.5. Оценка вероятности просачивания (`estimatev.py`) 327
Программа 2.4.6. Обнаружение просачивания (`percolation.py`) 329
Программа 2.4.7. Адаптивный графический клиент (`percplot.py`) 332

Объектно-ориентированное программирование

Использование типов данных

Программа 3.1.1. Идентификация потенциального гена (`potentialgene.py`) 353
Программа 3.1.2. Клиент заряженной частицы (`chargeclient.py`) 357
Программа 3.1.3. Квадраты Альберса (`alberssquares.py`) 362
Программа 3.1.4. Модуль яркости (`luminance.py`) 364
Программа 3.1.5. Преобразование цвета в полутон (`grayscale.py`) 368
Программа 3.1.6. Масштабирование изображений (`scale.py`) 369
Программа 3.1.7. Эффект постепенного изменения (`fade.py`) 370
Программа 3.1.8. Визуализация электрического потенциала (`potential.py`) 373
Программа 3.1.9. Конкатенация файлов (`cat.py`) 376
Программа 3.1.10. Анализ экранных данных для котировки акций (`stockquote.py`) 378
Программа 3.1.11. Разделение файла (`split.py`) 379

Создание типов данных

Программа 3.2.1. Заряженная частица (`charge.py`) 400
Программа 3.2.2. Секундомер (`stopwatch.py`) 404
Программа 3.2.3. Гистограмма (`histogram.py`) 406

Программа 3.2.4. Черепашья графика (`turtle.py`) 409
Программа 3.2.5. Удивительная спираль (`spiral.py`) 412
Программа 3.2.6. Комплексные числа (`complex.py`) 417
Программа 3.2.7. Множество Мандельброта (`mandelbrot.py`) 421
Программа 3.2.8. Биржевая учетная запись (`stockaccount.py`) 425

Разработка типов данных

Программа 3.3.1. Комплексные числа (`complexpolar.py`) 446
Программа 3.3.2. Счетчик (`counter.py`) 449
Программа 3.3.3. Пространственные векторы (`vector.py`) 456
Программа 3.3.4. Эскиз документа (`sketch.py`) 473
Программа 3.3.5. Обнаружение подобия (`comparedocuments.py`) 475

Случай из практики: моделирование N тел

Программа 3.4.1. Гравитационное тело (`body.py`) 490
Программа 3.4.2. Моделирование N тел (`universe.py`) 493

Алгоритмы и структура данных

Эффективность

Программа 4.1.1. Задача суммирования триплетов (`threesum.py`) 503
Программа 4.1.2. Проверка гипотезы удвоения (`doublingtest.py`) 505

Сортировка и поиск

Программа 4.2.1. Бинарный поиск (20 вопросов) (`questions.py`) 544
Программа 4.2.2. Дихотомический поиск (`bisection.py`) 548
Программа 4.2.3. Бинарный поиск (в отсортированном массиве) (`binarysearch.py`) 551
Программа 4.2.4. Сортировка вставкой (`insertion.py`) 555
Программа 4.2.5. Проверка сортировки удвоением (`timesort.py`) 557
Программа 4.2.6. Сортировка с объединением (`merge.py`) 560
Программа 4.2.7. Подсчет частот (`frequencycount.py`) 565

Стеки и очереди

Программа 4.3.1. Стек (массив переменного размера) (`arraystack.py`) 580
Программа 4.3.2 Стек (связанный список) (`linkedstack.py`) 584
Программа 4.3.3. Вычисление выражения (`evaluate.py`) 591
Программа 4.3.4. Очередь FIFO (связанный список) (`linkedqueue.py`) 595

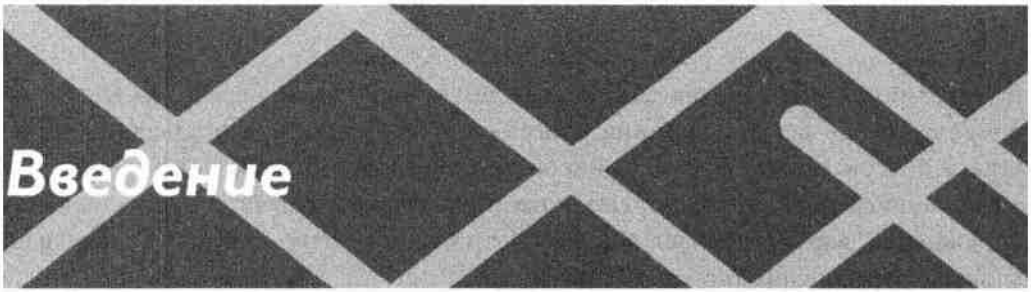
Программа 4.3.5. Модель очереди M / M / 1 (mm1queue.py) 601
Программа 4.3.6. Моделирование балансировки нагрузки (loadbalance.py) 604

Таблицы идентификаторов

Программа 4.4.1. Поиск в словаре (lookup.py) 626
Программа 4.4.2. Индексация (index.py) 628
Программа 4.4.3. Хеш-таблица (hashst.py) 634
Программа 4.4.4. Бинарное дерево поиска (bst.py) 641

Случай из практики: феномен "тесного мира"

Программа 4.5.1. Тип данных графа (graph.py) 674
Программа 4.5.2. Использование графа для инверсии индекса (invert.py) 678
Программа 4.5.3. Клиент кратчайших путей (separation.py) 681
Программа 4.5.4. Реализация кратчайших путей (pathfinder.py) 687
Программа 4.5.5. Проверка "тесного мира" (smallworld.py) 692
Программа 4.5.6. Граф "исполнитель–исполнитель" (performer.py) 694



Введение

В прошлом тысячелетии основами образования были чтение, письмо и арифметика; теперь это чтение, письмо и *компьютеры*. Умение программировать является основой образования любого студента технических специальностей. Кроме самих приложений, это первый этап в понимании природы бесспорного воздействия информатики на современный мир. Эта книга предназначена для тех, кто хочет изучить программирование или нуждается в нем для научных целей.

Задача книги — предоставить студентам практические навыки и знания основных инструментальных средств, необходимых для эффективного использования вычислительной техники. Наш подход — убедить студентов, что создание программ — это вполне естественная, занимательная и творческая практика. Для иллюстрации основных концепций и демонстрации возможностей написания программ в практических целях здесь представлены классические приложения в области прикладной математики и научных задач.

Для всех программ в этой книге используется язык программирования Python. Хотя язык Python упоминается в названии книги, по существу эта книга о *фундаментальных концепциях программирования*, а не о самом языке Python. Книга позволяет приобрести практические навыки решения задач средствами, применимыми во многих современных вычислительных системах, а исчерпывающее изложение материала предназначено для тех, кто не обладает опытом программирования.

В книге использован *междисциплинарный* подход к традиционной программе обучения CS1, при этом подчеркивается роль вычислений в других дисциплинах: от материаловедения и генетики до астрофизики и сетевых систем. Этот подход демонстрирует студентам основную идею: математика, наука, техника и компьютерные вычисления тесно переплетаются в современном мире. Хоть этот учебник по программе CS1 предназначен для всех студентов-первокурсников, интересующихся математикой, наукой и техникой, книга применима также и для самостоятельного изучения или как дополнительный курс при интеграции программирования с другой областью науки.

Изложение. Книга организована по четырем уровням обучения программированию: базовые элементы, функции, объектно-ориентированное

программирование и алгоритмы. На каждом уровне, перед переходом к следующему, необходимая для создания программ информация предоставляется читателю в доверительной форме. Основой применяемого подхода является использование примеров программ, решающих интригующие задачи, и предоставление упражнений, варьирующихся по сложности от достаточно простых до довольно сложных, требующих творческих решений.

К базовым элементам относятся переменные, оператор присвоения, встроенные типы данных, управление потоком, массивы, ввод-вывод, графика и звук.

Функции и модули — это первый шаг студента к модульному программированию. Предполагается, что читатель уже знаком с математическими функциями перед переходом к функциям Python и последующим рассмотрением значения программных функций, включая библиотеки функций и рекурсии. Здесь подчеркивается фундаментальная идея разделения программы на компоненты, допускающие независимую отладку, поддержку и многократное использование.

Объектно-ориентированное программирование — это введение в абстракцию данных. Здесь излагаются концепции типов данных и их реализации с использованием механизма классов Python. Студенты обучаются тому, как *использовать*, *создавать* и *проектировать* типы данных. Модульность, инкапсуляция и другие современные парадигмы программирования — это центральные концепции на данном этапе.

Алгоритмы и структуры данных объединяют современные парадигмы программирования с классическими методами организации и обработки данных, которые остаются эффективными и для современных приложений. Этап содержит введение в классические алгоритмы сортировки и поиска, а также фундаментальные структуры данных. Здесь подчеркивается использование научного метода для изучения характеристик эффективности реализаций.

Приложения в науке и технике — главная тема книги. Каждая рассматриваемая концепция программирования мотивируется исследованием ее воздействия на определенные приложения. Приводятся примеры из прикладной математики, физики, биологии и самой информатики. Они включают моделирование физических систем, числовые методы, визуализацию данных, синтез звука, обработку изображений, финансовые модели и информационные технологии. К конкретным примерам относится применение в первой главе цепей Маркова для рангов веб-страницы, а также анализа и решения задачи просачивания, моделирования N тела и феномена "тесного мира". Эти приложения — неотъемлемая часть книги. Они привлекают внимание студентов к материалу, иллюстрируют важность концепций программирования и предоставляют убедительное доказательство важности роли компьютерных вычислений в современной науке и технике.

Главная задача книги — научить определенным техникам и привить навыки, необходимые для поиска эффективных решений любых задач программирования.

Поэтому мы сосредоточимся на программировании частностей, а не на общем программировании.

Использование в учебном процессе. Книга предназначена для студентов первого курса, для обучения новичков программированию в контексте научных приложений. Материал книги подходит для студентов, обучающихся программированию в знакомом контексте. Студенты, закончившие курс с использованием этой книги, будут хорошо подготовлены к применению своих навыков в последующих курсах, если понадобится дальнейшее изучение информатики.

Студенты информационных специальностей также могут извлечь пользу из примеров программирования в контексте научных приложений. Программист нуждается в той же базовой подготовке и научных методах, а также в том же знании роли компьютерных вычислений в науке, как и биолог, инженер или физик.

Действительно, междисциплинарный подход позволяет институтам и университетам обучать будущих специалистов в области информатики и других научно-технических специальностей на *том же* курсе. Здесь изложен материал согласно учебному плану CS1, но перенос основного внимания на приложения вносит жизнь в абстрактные концепции и поощряет студентов изучать их. Междисциплинарный подход предлагает студентам задачи по многим дисциплинам, помогая им выбрать профилирующий предмет более осмысленно.

Безотносительно конкретного механизма, эта книга наилучшим образом подходит для упомянутого ранее учебного плана. Во-первых, такая позиция позволяет манипулировать знакомым по средней школе математическим материалом. Во-вторых, студенты, изучавшие программирование на предыдущих курсах, научатся эффективнее использовать компьютеры при переходе к курсам по своим специальностям. Подобно письму и чтению, программирование бесспорно является основополагающим навыком для любого ученого или инженера. Студенты, овладевшие концепциями из этой книги, будут непрерывно совершенствовать полученные навыки в практической деятельности, пользуясь преимуществами использования компьютеров для решения или лучшего понимания задач, возникающих в выбранной ими области.

Предпосылки. Книга предназначена для обычных студентов первого курса научно-технических вузов. Таким образом, здесь не предусматривается предварительная подготовка свыше обычной обязательной для других базовых научных и математических курсов.

Знание математики важно. Хотя на математический материал здесь акцент и не делается, в книге учтен школьный курс математики, включая алгебру, геометрию и тригонометрию. Большинство студентов из нашей целевой аудитории изначально отвечают этим требованиям. На самом деле авторы полагаются на знакомство читателя с базовым курсом математики, чтобы ознакомить их с основными концепциями программирования.

Интерес к науке — также необходимый компонент. Студенты научных и технических специальностей априори полагают, что научный поиск способен помочь объяснить происходящее в природе. Это мнение подкрепляется примерами простых программ, говорящих красноречивее всяких слов о природном мире. Здесь не подразумевается наличие специальных знаний, кроме предоставляемых обычными уроками математики, физики, биологии или химии в средней школе.

Опыт программирования необязателен, но и не повредит. Обучение программированию является главной задачей книги, поэтому наличие предварительного опыта программирования не подразумевается. Однако создание программы для решения новой проблемы является сложной интеллектуальной задачей, поэтому студенты, писавшие программы в средней школе, могут извлечь пользу из имеющихся знаний. Книга подходит для обучения студентов с различной предварительной подготовкой, поскольку приложения ориентированы и на новичков, и на экспертов.

Таким образом, практически все студенты научных и технических специальностей готовы пройти курс по этой книге в составе учебного плана первого семестра.

Задачи. Что *преподаватели* старших курсов научных и технических вузов могут ожидать от студентов, закончивших курс на основании этой книги?

Книга учитывает учебный план CS1, но любой, кто вел вводный курс программирования, знает, что ожидания преподавателей старших курсов обычно весьма высоки: каждый из них полагает, что все студенты знакомы с компьютерной средой и подходом, который они хотели бы использовать. Профессор физики мог бы ожидать, что студенты за выходные разработают моделирующую программу; технический профессор мог бы ожидать, что студенты будут использовать некий пакет для решения дифференциальных уравнений в цифровой форме; а профессор информатики мог бы ожидать знания подробностей определенной среды программирования. Реально ли удовлетворить столь разнообразные ожидания? Должен ли быть отдельный вводный курс для каждого набора студентов?

Институты и университеты решают эти вопросы с тех пор, как компьютеры получили широкое распространение во второй половине XX столетия. Наш ответ на них вы найдете в этом общем вводном курсе по программированию, похожем на обычные вводные курсы по математике, физике, биологии или химии. Эта книга призвана обеспечить базовую подготовку, необходимую всем студентам научных и технических специальностей, а также помочь уяснить, что в информатике есть много больше, чем только программирование. Преподаватели, обучающие студентов, учившихся по этой книге, могут ожидать, что они будут иметь знания и опыт, необходимые для адаптации к новым вычислительным системам, и эффективно использовать компьютеры для разных целей.

Что могут ожидать *студенты* старших курсов, закончившие обучение на основании этой книги?

Мы заявляем, что изучить программирование вовсе не трудно и использовать компьютерные средства весьма полезно. Овладевшие материалом этой книги студенты готовы решать вычислительные задачи везде, где они могут возникнуть в последующей практической деятельности. Они узнают, что современные среды программирования, такие как язык Python, помогают решить любую вычислительную задачу, с которой они могли бы встретиться позже, а также приобрести опыт по изучению, освоению и использованию других вычислительных средств. Студенты, интересующиеся информатикой, смогут удовлетворить свой интерес, а студенты научных и технических специальностей будут готовы интегрировать компьютерные вычисления в свои исследования.

Сайт книги. Дополнительная информация по этой книге приведена на веб-сайте по адресу `http://introcs.cs.princeton.edu/python`.

Для экономии места мы называем этот сайт в книге просто *сайтом книги.* Он содержит материал для преподавателей, студентов и всех читателей книги. Хотя материал сайта кратко описывается здесь, все пользователи Интернета знают, что лучше посмотреть самому, чем прочитать описание. За несколькими исключениями, весь материал общедоступен.

Важнейшее преимущество сайта книги в том, что он позволяет преподавателям и студентам использовать собственные компьютеры для преподавания и изучения материала. Любой читатель, имеющий компьютер и браузер, может начать обучение программированию, следуя инструкциям на сайте книги. Процесс не труднее загрузки медиапроигрывателя и музыки. Подобно любым веб-сайтам, сайт книги непрерывно развивается. Это основной ресурс для всех владельцев данной книги. В частности, размещенные на нем дополнительные материалы критически важны для нашей задачи интеграции информатики в образовательный процесс.

Преподавателям сайт книги предоставляет информацию об обучении. В первую очередь эта информация сосредоточивается на стиле обучения, который за прошедшее десятилетие существенно развился. Мы предоставляем две лекции в неделю для большой аудитории и дополняем их двумя классными занятиями в неделю, когда студенты небольшими группами встречаются с преподавателями или ассистентами. На сайте книги есть также слайды к лекциям.

Ассистентам сайт книги предоставляет подробные наборы задач и программных проектов на основании упражнений из книги. Каждая программа предназначена для демонстрации соответствующей концепции в контексте занимательного приложения, достаточно сложного для каждого студента. Набор заданий воплощает наш подход к обучению программированию. Сайт книги полностью определяет все задания и предоставляет подробную, структурированную

информацию, призванную помочь студентам выполнить их за выделенное время. Здесь также содержатся описания предложенных подходов и уточнение того, что именно следует преподавать на классных занятиях.

Студентам сайт предоставляет быстрый доступ к большей части материала книги, включая исходный код и дополнительные материалы для самоподготовки. Для большинства упражнений "книге" предоставляются решения, включая полный код программ и проверочные данные. Здесь есть много информации, связанной с заданиями на программирование, включая рекомендованные подходы, контрольные списки, часто задаваемые вопросы и проверочные данные.

Другим читателям сайт служит ресурсом дополнительной информации, связанной с материалом книги. Все содержимое сайта представляет собой ссылки и другие указания на подробную информацию по рассматриваемой теме. Детальной информации доступно больше, чем сможет полностью освоить любой человек, но наша задача заключается в том, чтобы предоставить достаточно дополнительной информации о содержимом книги и удовлетворить потребности любого читателя.

Благодарности. Этот проект находится в разработке с 1992 года. На протяжении этих лет слишком много людей поспособствовали его успеху, чтобы упомянуть их всех здесь. Особая благодарность Энн Роджерс (Anne Rogers) за то, что помогла начать дело; Дейву Хэнсону (Dave Hanson), Эндрю Аппель (Andrew Appel) и Крису ван Вик (Chris van Wyk) за их терпение в объяснении абстракций данных; Лайзе Вортингтон (Lisa Worthington), которая стала первой студенткой, преодолевшей трудности изучения этого материала на первом курсе. Мы также с благодарностью признаем усилия `/dev/126`. Мы признательны аспирантам, преподавателям и тысячам студентов Принстонского университета, обучавшихся этому материалу прошедшие 25 лет.

Роберт Седжвик
Кевин Уэйн
Роберт Дондеро
Апрель 2015

Соглашения, принятые в книге

Здесь используются соглашения, общепринятые в компьютерной литературе.

- Новые термины в тексте выделяются *курсивом*. Чтобы обратить внимание читателя на отдельные фрагменты текста, также применяется *курсив*.
- Текст программ, функций, переменных, URL веб-страниц и другой код представлен моноширинным шрифтом.
- Все, что придется вводить с клавиатуры, выделено **полужирным моноширинным** шрифтом.
- Знакоместо в описаниях синтаксиса выделено курсивом. Это указывает на необходимость заменить знакоместо фактическим именем переменной, параметром или другим элементом, который должен находиться на этом месте: `BINDSIZE=(`*`максимальная ширина колонки`*`)*(`*`номер колонки`*`)`.
- Пункты меню и названия диалоговых окон представлены следующим образом: Menu Option (Пункт меню).
- В листингах каждая строка имеет номер. Это сделано исключительно для удобства описания. В реальном коде нумерация отсутствует. Текст некоторых строк кода в листингах иногда бывает слишком длинным и не помещается на странице книги. Таким образом, отсутствие номера в начале строки свидетельствует о том, что строка является продолжением предыдущей и в реальном коде их разрывать не следует.

Текст некоторых абзацев выделен специальным шрифтом. Это примечания, советы и предостережения, которые помогут обратить внимание на наиболее важные моменты в изложении материала и избежать ошибок в работе.

От издательства

Вы, читатель этой книги, и есть главный ее критик. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересно услышать и любые другие замечания, которые вам хотелось бы высказать авторам.

Мы ждем ваших комментариев. Вы можете прислать письмо по электронной почте или просто посетить наш веб-сайт, оставив на нем свои замечания. Одним словом, любым удобным для вас способом дайте нам знать, нравится ли вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более подходящими для вас.

Посылая письмо или сообщение, не забудьте указать название книги и ее авторов, а также ваш электронный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию следующих книг. Наши электронные адреса:

E-mail: info@dialektika.com
WWW: http://www.dialektika.com

Наши почтовые адреса:

в России: 195027, г. Санкт-Петербург, ул. Магнитогорская, д. 30, ящик 116
в Украине: 03150, Киев, а/я 152

Глава 1 Элементы программирования

1.1. Первая программа **20**
1.2. Встроенные типы данных **31**
1.3. Условные выражения и циклы **70**
1.4. Массивы **110**
1.5. Ввод и вывод **147**
1.6. Случай из практики: случайная навигация по сайтам **193**

Задача этой главы — убедить читателя, что создание программы проще написания такой части текста, как этот абзац, или сочинения. Писать прозу трудно, этому придется учиться несколько лет. В отличие от этого, чтобы составить программу, способную решить различные задачи, не решаемые в противном случае, достаточно знать лишь несколько стандартных блоков. Данная глава посвящена этим стандартным блокам; она позволит начать программировать на языке Python и изучить множество интересных программ. Всего через нескольких недель вы уже будете в состоянии самовыражаться (создавая программы). Как и способность писать прозу, способность программировать — это навык, приобретаемый со временем и непрерывно совершенствуемый.

Данная книга позволяет изучить *язык программирования Python*. Эта задача намного проще, чем, например, изучение иностранного языка. Действительно, языки программирования насчитывают лишь несколько десятков слов и правил. Большая часть рассматриваемого здесь материала могла быть также выражена на языке Java, C++ или любом другом современном языке программирования. Мы выбрали язык Python специально, чтобы вы сразу могли начать писать и запускать программы. С одной стороны, мы сосредоточимся на изучении программирования, а не на деталях языка Python. Но, с другой стороны, для программирования важно знать, какие детали языка уместны в данной конкретной ситуации. Поскольку язык Python получил широкое распространение, его изучение

позволит составлять программы на многих компьютерах (собственном, например). Кроме того, изучение программирования на языке Python облегчит самостоятельное изучение других языков, включая такие низкоуровневые языки, как C, и такие специализированные языки, как Matlab.

1.1. Первая программа

Программы этого раздела...

Программа 1.1.1. Hello, World (helloworld.py) 21

Программа 1.1.2. Использование аргумента командной строки (useargument.py) 24

Этот раздел задуман как введение в мир программирования Python, в нем излагаются фундаментальные этапы, необходимые для получения простой программы и ее выполнения. *Система Python* (далее просто *Python*) представляет собой набор приложений, мало чем отличающихся от большинства других привычных приложений (таких, как текстовой процессор, программа электронной почты и веб-браузер). Подобно любым приложениям, Python необходимо правильно установить на компьютере. На многие компьютеры он установлен изначально, но его можно легко загрузить и установить самостоятельно. Понадобятся также текстовый редактор и терминальное приложение. В первую очередь следует найти инструкции по установке среды программирования Python на компьютере по адресу `http://introcs.cs.princeton.edu/python`.

Как уже упоминалось, мы называем этот сайт *сайтом книги*. Он содержит обширный объем дополнительной информации о материале этой книги для справки и использования при программировании.

Программирование на языке Python. Для ознакомления с разработкой программ на языке Python разделим процесс на два этапа. Для программирования на языке Python необходимо следующее.

- *Создать* программу, набрав и сохранив ее в файле, скажем, по имени `myprogram.py`.
- *Запустить* (или *выполнить*) программу, введя `python myprogram.py` в окне терминального приложения.

На первом этапе все начинается с пустого экрана, а завершается набором введенных символов, как и при написании сообщения электронной почты или текста. Для описания текста программ программисты используют термин *код*, а термином *кодирование* (coding) называют процесс создания и редактирования кода. На втором этапе вы передаете управление компьютером своей программе (которая возвращает управление операционной системе по завершении). Многие системы поддерживают несколько способов создания и запуска программ. Мы выбрали представленный здесь порядок потому, что его проще описать и использовать для маленьких программ.

Создание программы. Программа Python не более чем последовательность символов, как абзац текста или стихотворение, сохраненное в файле с именем и расширением `.py`. Для их создания достаточно просто определить эту последовательность символов таким же образом, как и при написании сообщения электронной почты или в любом другом компьютерном приложении. Для этого можно использовать любой текстовый редактор или одну из более сложных сред разработки, описанных на сайте книги. Для рассматриваемых в этой книге программ такие среды разработки необязательны, но они имеют много полезных средств, и использовать их вовсе не трудно. Профессионалы широко используют их.

Выполнение программы. Как только программа будет составлена, ее можно запустить на выполнение. Это самая захватывающая часть, когда ваша программа берет под свой контроль компьютер (в пределах того, что позволяет язык Python). Точнее, компьютер следует вашим инструкциям. А если еще точнее, то *компилятор* Python переводит программу Python на язык, более подходящий для исполнения на компьютере. Затем *интерпретатор* Python переходит к управлению компьютером согласно этим инструкциям. Термины *выполнение* (executing) и *запуск* (running) в этой книге подразумевают комбинацию компиляции и интерпретации программы. Чтобы использовать компилятор и интерпретатор Python для запуска программы, достаточно ввести в окне терминала команду `python`, сопровождаемую именем файла, содержащего программу Python.

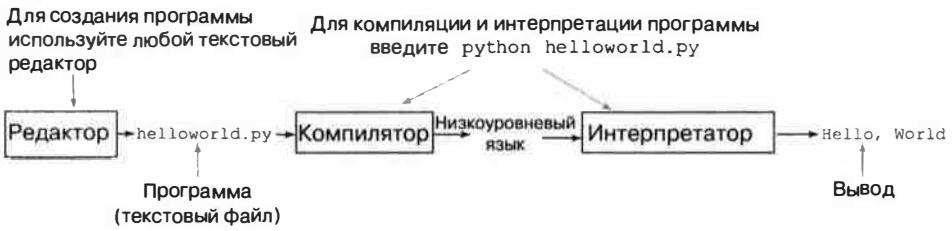
Программа 1.1.1 — это пример вполне законченной программы Python. Ее код располагается в файле `helloworld.py`. Единственное действие программы — вывод сообщения в окне терминала. Программа Python состоит из *операторов*. Каждый оператор обычно помещают в отдельную строку.

Программа 1.1.1. Hello, World (`helloworld.py`)

```
import stdio

# Выводит 'Hello, World' на стандартное устройство вывода.
stdio.writeln('Hello, World')
```

Этот код — программа Python, выполняющая простую задачу. Традиционно это первая программа новичка. Ниже показано, что происходит при выполнении программы. Терминальное приложение предоставляет приглашение к вводу команд (в этой книге символ `%`) и выполняет вводимые команды (выделены полужирным шрифтом). Результат выполнения этого кода — вывод в окне терминала текста `Hello, World` (четвертая строка кода).

```
% python helloworld.py
Hello, World--
```



Разработка программы Python

Первая строка файла `helloworld.py` содержит оператор `import`, указывающий на намерение использовать средства, определенные в *модуле* `stdio`, т.е. в файле `stdio.py`. Файл `stdio.py` мы специально разработали для этой книги. В нем определены функции чтения ввода и записи в вывод. Импортировав модуль `stdio`, можно впоследствии *вызвать функцию*, определенную в этом модуле.

- Вторая строка — это пустая строка. Python игнорирует пустые строки; программисты используют их для отделения логических блоков кода.

Третья строка содержит *комментарий*, предназначенный для документирования программы. В языке Python комментарий начинается с символа `'#'` и продолжается до конца строки. Python игнорирует комментарии — они предназначены только для людей, читающих программы.

- Четвертая строка — основа программы. Этот оператор вызывает функцию `stdio.writeln()`, предназначенную для вывода одной строки с заданным текстом. Обратите внимание, что происходит вызов функции из другого модуля: сначала указано имя модуля, затем точка и имя функции.

Python 2. Лингва франка в этой книге — Python 3, поскольку это будущее программирования на языке Python. Тем не менее авторы сделали все возможное, чтобы код в книге работал и с Python 2, и с Python 3. Например, в Python 2 программа 1.1.1 могла бы использовать просто строку `print 'Hello, World'`, но это недопустимо в Python 3. Чтобы получить код, осуществляющий вывод в любой версии языка Python, мы используем собственный модуль stdio. Каждый раз, когда будет существенное различие между этими двумя языками, мы обратим внимание пользователей Python 2 на это.

По традиции, начиная с 1970-х годов, первая программа начинающего программиста выводит слова `'Hello, World'`. Таким образом, наберите код программы 1.1.1, сохраните его в файле `helloworld.py`, а затем запустите его. Сделав это, вы начнете свой путь, как и множество других, изучавших программирование. Кроме того, это позволит проверить наличие пригодного для использования текстового редактора и терминального приложения. На первый взгляд, процесс вывода чего-то в окно терминала может показаться не очень интересным;

но впоследствии вы увидите, что одной из наиболее необходимых программам функций является способность сказать, что они делают.

Поскольку пока код всех наших программ будет точно таким же, как код `helloworld.py`, кроме иных имен файлов, комментариев и последовательностей операторов после него, не стоит начинать программу каждый раз с чистого листа. Вместо этого

- скопируйте файл `helloworld.py` в файл с новым именем, но не забудьте про расширение `.py`;
- замените комментарий и оператор `stdio.writeln()` новыми.

Программа характеризуется последовательностью ее операторов и именем. Согласно соглашению, каждая программа Python находится в файле с расширением `.py`.

Ошибки. Довольно просто размыть различие между редактированием, компиляцией и интерпретацией программы. Но обучаясь программированию, следует помнить о них, чтобы лучше понять происхождение ошибок, которые неизбежно возникнут. Несколько примеров ошибок можно найти в разделе "Вопросы и ответы" в конце этого раздела.

Внимательность при создании и просмотре программы позволяет устранить или избежать большинства ошибок, точно так же, как проверка правописания позволяет избежать грамматических и синтаксических ошибок в сообщении электронной почты. Некоторые ошибки, известные как *ошибки времени компиляции* (compile-time error), обнаруживаются при компиляции программы, поскольку они препятствуют трансляции программы. Python оповещает об ошибке времени компиляции сообщением `SyntaxError`. Другие ошибки, известные как *ошибки времени выполнения* (run-time error), не проявляются, пока Python не интерпретирует программу. Например, если в программе `helloworld.py` забыть оператор `import stdio`, то во время ее выполнения Python передаст сообщение `NameError`.

Обычно *ошибки в программах* (bug) отравляют жизнь программиста: сообщения об ошибках могут быть сомнительны или вводить в заблуждение, а найти источник ошибки может быть очень трудно. Один из первых приобретаемых навыков — это поиск ошибок; вы научитесь также быть достаточно внимательным при написании кода, чтобы не допускать большинства ошибок.

Ввод и вывод. Как правило, программе необходимо обеспечить ввод данных, чтобы они могли обработать их и получить результат. Самый простой способ обеспечения ввода данных иллюстрирует программа 1.1.2 (`useargument.py`). При каждом запуске программы `useargument.py` ей передают аргумент командной строки, вводимый после имени программы, и она выводит его в составе сообщения. Результат выполнения этой программы зависит от того, что введено после имени программы. Программу можно запустить с различными аргументами командной строки и получить различные результаты.

Оператор `import sys` в программе `useargument.py` указывает на необходимость использовать средства, определенные в модуле `sys`. Одно из этих средств, `argv`, является списком аргументов командной строки (вводимых в командной строке после "`python useargument.py`" и разделяемых пробелами). Подробней этот механизм обсуждается позже, в разделе 2.1. Пока этого достаточно, чтобы понять, что `sys.argv[1]` — это первый аргумент командной строки, располагающийся во вводе сразу после имени программы; `sys.argv[2]` — второй аргумент командной строки, вводимый после первого, и т.д. Аргумент `sys.argv[1]` можно использовать в теле программы для представления строки, введенной в командной строке при запуске такой программы, как `useargument.py`.

Программа 1.1.2. Использование аргумента командной строки (`useargument.py`)

```
import sys
import stdio

stdio.write('Hi, ')
stdio.write(sys.argv[1])
stdio.writeln('. How are you?')
```

Этот пример демонстрирует возможность контроля действий программ за счет ввода аргумента в командной строке. Это позволяет изменить поведение программ. Программа получает аргумент командной строки и выводит использующее его сообщение.

```
% python useargument.py Alice
Hi, Alice. How are you?

% python useargument.py Bob
Hi, Bob. How are you?

% python useargument.py Carol
Hi, Carol. How are you?
```

Кроме функции `writeln()`, программа 1.1.2 использует функцию `write()`. Эта функция точно такая же, как и функция `writeln()`, но она выводит строку, не завершая ее символом новой строки.

Как и прежде, сама по себе задача вывода того, что было введено в программу, возможно, не покажется интересной, но впоследствии вы поймете, что следующей фундаментальной функцией программы является способность реагировать на введенную пользователем информацию, что позволяет контролировать деятельность программы. Представленной в программе `useargument.py` простой модели вполне

достаточно, чтобы представить себе базовый механизм программирования Python и применять его для решения разнообразных вычислительных задач.

Какможнозаметить,программа `useargument.py` всего лишь вставляет одну строку символов (аргумент) в другую (получая выводимое сообщение). Таким образом, программу Python можно считать "черным ящиком", преобразующим входящую строку в выходящую.

Общая панорама программы Python

Эта модель привлекательна потому, что она не только проста, но и достаточно общая, чтобы позволить объяснить принцип любой вычислительной задачи. Например, сам компилятор Python — не более чем программа, получающая одну строку как ввод (файл `.py`) и создающая другую строку как вывод (соответствующая программа на более низкоуровневом языке). Впоследствии вы будете создавать программы, решающие множество интересных задач (хотя мы не дойдем до таких сложных программ, как компилятор). На настоящий момент мы имеем ограничения на размеры и типы ввода и вывода программ; более сложные механизмы ввода и вывода данных программы рассматриваются в разделе 1.5. В частности, рассматривается возможность работы со строками произвольной длины и другими типами данных, такими как звук и изображения.

Вопросы и ответы

Почему мы используем именно Python?

Изучаемые здесь программы очень похожи на их аналоги на других языках, поэтому наш выбор языка не крайне важен. Мы используем язык Python потому, что он широко распространен, доступен, охватывает весь набор современных абстракций и обладает множеством средств автоматической проверки ошибок в программах. Таким образом, он вполне подходит для обучения программированию. Язык Python постоянно развивается и выходит во многих версиях.

Какую версию языка Python использовать?

Мы рекомендуем использовать Python 3, но мы сделали все возможное, чтобы код в книге работал и на Python 2, и на Python 3. Весь код был проверен на языке Python версии 2.7 и 3.4 (последние выпуски Python 2 и Python 3 на момент публикации). Мы используем общепринятый термин Python 2 для описания Python 2.7 и Python 3 для Python 3.4.

Как установить среду программирования Python?

На сайте книги есть поэтапная инструкция по установке среды программирования Python на операционной системе Mac OS X, Windows или Linux. Там же есть инструкция по установке книжных модулей, таких как `stdio.py`.

Должен ли я набирать все программы, приведенные в книге, чтобы испытать их?

Я уверен, что вы уже запускали их и что они создают приведенный вывод. Имеет смысл набрать и запустить программу `helloworld.py`. Вы приобретете больше навыков, если запустите также и программу `useargument.py`, опробовав различные входные данные, чтобы проверить собственные идеи. Чтобы сэкономить на наборе, весь код (и многое другое) можно найти на сайте книги. Там же есть информация по установке среды разработки Python на компьютере, ответы на упражнения, веб-ссылки и другая полезная или интересная дополнительная информация.

Когда я запускаю программу `helloworld.py`, Python выдает сообщение

```
ImportError: No module named stdio.
```

Что это означает?

Это означает, что книжный модуль `stdio` недоступен для Python.

Как сделать книжный модуль `stdio` доступным для Python?

Если вы следовали поэтапным инструкциям по установке среды программирования Python с сайта книги, то модуль `stdio` должен уже быть доступен для Python. Но можно также загрузить файл `stdio.py` с сайта книги и поместить его в тот же каталог, где находится и использующая его программа.

Каковы правила Python для символов отступа, таких как табуляция, пробелы и новые строки?

Для языка Python большинство отступов в тексте программы эквивалентно, кроме двух важных исключений: *строковых литералов* (string literal) и *выравнивания* (indentation). Строковый литерал — это последовательность символов в одинарных кавычках, такая как `'Hello, World'`. Если поместить несколько пробелов в кавычки, то в строковом литерале вы получаете то же количество пробелов. *Выравнивание* — это отступ в начале строки. Количество пробелов в начале строки играет важную роль в структурировании программ Python, как будет продемонстрировано в разделе 1.3. Пока выравнивать код не нужно.

Зачем используются комментарии?

Комментарии помогают другим программистам понять ваш код или вам понять собственный код через некоторое время. Ограничения на объем книги требуют экономно использовать комментарии в представленных здесь программах (вместо этого мы полностью описываем каждую программу в тексте и рисунках). Программы на сайте книги прокомментированы более подробно.

Могу ли я поместить несколько операторов в одну строку?

Да, отделяя операторы точками с запятой. Например, эта единственная строка кода создает тот же вывод, что и программа `helloworld.py`

```
import stdio; stdio.writeln('Hello, World')
```

Однако многие программисты не рекомендуют использовать этот стиль.

Что будет, если пропустить круглую скобку или написать такое слово, как `stdio`, `write` или `writeln`, с ошибкой?

Это зависит от того, что именно сделано. Эти так называемые *синтаксические ошибки* (syntax error) обычно обнаруживает компилятор. Например, если вы создадите программу `bad.py`, точно такую же, как и `helloworld.py`, но без первой (левой) круглой скобки, то получите следующее сообщение об ошибке:

```
% python bad.py
  File 'bad.py', line 4
    stdio.write'Hello, World')
                              ^
SyntaxError: invalid syntax
```

Исходя из этого сообщения, можно вполне резонно предположить, что необходимо добавить левую круглую скобку. Компилятор не сможет указать ошибку точно, поэтому сообщение об ошибке может быть трудно понять. Например, если пропустить вторую (правую) круглую скобку вместо первой (левой), вы получите следующее, куда менее полезное сообщение, в котором к тому же указана строка, следующая после ошибочной:

```
% python bad.py
  File 'bad.py', line 5
                      ^
SyntaxError: unexpected EOF while parsing
```

Чтобы быстрей привыкнуть к таким сообщениям, можно преднамеренно вводить ошибки в простую программу и наблюдать результат. Несмотря на сообщения об ошибках, считайте компилятор своим другом, поскольку он просто пытается предупредить вас, что с программой что-то неправильно.

При запуске программы `useargument.py` я получаю странное сообщение об ошибке. Пожалуйста, объясните.

Вероятней всего, вы забыли аргумент командной строки:

```
% python useargument.py
Hi, Traceback (most recent call last):
File 'useargument.py', line 5, in <module>
stdio.write(sys.argv[1])
IndexError: list index out of range
```

Интерпретатор Python жалуется, что вы запустили программу, но не ввели аргумент командной строки, как ожидалось. Более подробная информация об индексах списка приведена в разделе 1.4. Запомните это сообщение об ошибке: вы, вероятно, увидите его снова. Даже опытные программисты забывают иногда вводить аргументы командной строки.

Какие модули Python и функции доступны для использования?

В любой версии Python есть много стандартных модулей. Еще больше доступно дополнительных *модулей расширения* (extension module), которые можно загрузить и установить впоследствии. Мы создали еще и другие модули (такие, как `stdio`) специально для этой книги и ее сайта; мы назвали их *книжными модулями* (booksite module). Короче говоря, для использования доступны сотни модулей Python с разными функциями. В этой книге рассматриваются только самые основные модули и функции.

Упражнения

1.1.1. Напишите программу, выводящую слова `Hello, World` десять раз.

1.1.2. Что будет, если в программе `helloworld.py` пропустить следующее:

a. `import`

b. `stdio`

c. `import stdio`

1.1.3. Что будет, если в программе `helloworld.py` допустить орфографическую ошибку (скажем, пропустить второй символ) в следующем:

a. `import`

b. `stdio`

c. `write`

d. `writeln`

1.1.4. Что будет, если в программе `helloworld.py` пропустить следующее:

a. Первый `'`

b. Второй `'`

c. Оператор `stdio.writeln()`

1.1.5. Что будет, если попытаться выполнить программу `useargument.py` с каждой из следующих командных строк:

a. `python useargument.py python`

b. `python useargument.py @!&^%`

c. `python useargument.py 1234`

d. `python useargument Bob`

e. `useargument.py Bob`

f. `python useargument.py Alice Bob`

1.1.6. Переделайте программу `useargument.py` в программу `usethree.py`, получающую три имени и выводящую целое предложение с именами в порядке, обратном их вводу. Например, вызов `python usethree.py Alice Bob Carol` создает вывод `'Hi Carol, Bob, and Alice'`.

1.2. Встроенные типы данных

Программы этого раздела...

Программа 1.2.1. Пример конкатенации строк (ruler.py) 39
Программа 1.2.2. Целочисленные операторы (intops.py) 42
Программа 1.2.3. Операторы чисел с плавающей точкой (floatops.py) 45
Программа 1.2.4. Квадратичная формула (quadratic.py) 46
Программа 1.2.5. Високосный год (leapyear.py) 50

При программировании на языке Python вы всегда должны знать тип данных, используемых вашей программой. Программы раздела 1.1 обрабатывали строки, большинство программ данного раздела обрабатывают числа, а многие другие типы данных рассматриваются позже. Понимание различий между ними настолько важно, что мы формально определяем идею: *тип данных* (data type) — это *набор значений* и *набор операций*, определенных для этих значений.

В язык Python встроено несколько типов данных. В этом разделе рассматриваются такие встроенные типы данных Python, как `int` (для целых чисел), `float` (для чисел с плавающей точкой), `str` (для последовательностей символов) и `bool` (для значений `true` и `false`). Они представлены в таблице ниже.

Основные встроенные типы данных

Тип	Набор значений	Общие операторы	Пример литералов
`int`	Целые числа	`+ - * // % **`	`99 12 2147483647`
`float`	Числа с плавающей точкой	`+ - * / **`	`3.14 2.5 6.022e23`
`bool`	Значения `true` и `false`	`and` или `not`	`True False`
`str`	Последовательности символов	`+`	`'AB' 'Hello' '2.5'`

Пока сосредоточимся на программах, использующих эти четыре базовых встроенных типа данных. Позже вы узнаете о дополнительных типах данных, доступных для использования, а также о том, как создавать собственные типы данных. На самом деле программирование на языке Python зачастую сосредоточивается именно на создании типов данных, но об этом речь пойдет в главе 3.

После определения основных терминов рассмотрим несколько примеров программ и фрагментов кода, иллюстрирующих использование различных типов данных. Эти фрагменты кода выполняют не много реальных вычислений, но скоро будет представлен подобный код в более длинных программах. Понимание типов данных (значений и операций с ними) является необходимым этапом в начале программирования. Это подготовит почву для начала самостоятельной работы с большим количеством более сложных программ в следующем разделе. Каждая создаваемая далее программа будет использовать код, подобный небольшим фрагментам, представленным в данном разделе.

Определения. Прежде чем обсуждать типы данных, необходимо ввести некую терминологию. Для этого начнем со следующего фрагмента кода:

```
a = 1234
b = 99
c = a + b
```

Этот код создает три *объекта* (object), каждый типа `int`, используя *литералы* `1234` и `99`, а также выражение `a + b` и *привязывает* (bind) (технический термин, означающий создание ассоциации) *переменные* `a`, `b` и `c` к этим объектам, используя *операторы присвоения* (assignment statement). В результате получается, что переменная `c` связана с объектом типа `int`, хранящим значение `1333`. Теперь рассмотрим термины, выделенные курсивом, подробней.

Литерал (literal). Литерал Python — это представление значения типа данных в коде. Для представления значения типа данных `int` здесь были использованы такие последовательности цифр, как `1234` и `99`, для представления значения типа `float` была бы добавлена десятичная точка, как в `3.14159` или `2.71828`, для представления двух значений типа `bool` использовались бы литералы `True` и `False`; а последовательность заключенных в кавычки символов, таких как `'Hello, World'`, представляет значение типа `str`.

Оператор (operator). Оператор Python — это представление операции типа данных в коде. Язык Python использует знаки `+` и `*` для представления сложения и умножения целых чисел и чисел с плавающей точкой. Для представления логических операций язык Python использует ключевые слова `and`, `or` и `not` и т.д.. Далее в этом разделе будут подробно описаны наиболее популярные операции для каждого из четырех базовых встроенных типов.

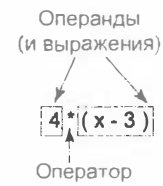
Идентификатор (identifier). Идентификатор Python — это представление имени в коде. Каждый идентификатор — это последовательность символов, цифр и знаков подчеркивания, первым из которых не может быть цифра. Последовательности символов `abc`, `Ab_`, `abc123` и `a_b` являются вполне допустимыми идентификаторами Python, а `Ab*`, `1abc` и `a+b` — нет. Идентификаторы различают регистр символов, таким образом, `Ab`, `ab` и `AB` — это разные имена. Такие *ключевые слова* (keyword), как `and`, `import`, `in`, `def`, `while`, `from` и `lambda`, зарезервированы и не могут быть использованы как идентификаторы. Другие имена, такие как `int`, `sum`, `min`, `max`, `len`, `id`, `file` и `input`, имеют в Python специальное значение, поэтому их также лучше не использовать.

Переменная (variable). Переменная — это имя, связанное со значением типа данных. Переменные используются для отслеживания изменений значений по ходу вычислений. Например, в нескольких программах этой книги переменная `total` используется для хранения результатов вычисления суммы ряда чисел. Для имен переменных программисты обычно используют стилистическое соглашение. В этой книге соглашение заключается в том, что имя каждой переменной

состоит из строчной буквы, сопровождаемой строчными и прописными буквами, а также цифрами. Мы используем прописные буквы в именах из нескольких слов. Так, например, мы используем следующие имена переменных: `i`, `x`, `y`, `total`, `isLeapYear` и `outDegrees` среди многих других.

Константная переменная (constant variable). Этот термин-оксиморон используется для описания переменной, ассоциированное значение типа данных которой не изменяется во время выполнения программы (или между запусками программы). Согласно принятому в этой книге соглашению, имя каждой константной переменной состоит из прописной буквы, сопровождаемой строчными и прописными буквами, цифрами и знаками подчеркивания. Например, для константной переменной можно было бы использовать имена `SPEED_OF_LIGHT` и `DARK_RED`.

Выражение (expression). Выражение — это комбинация литералов, переменных и операторов, *вычисляемое* (evaluate) Python для получения значения. Большинство выражений выглядит так же, как математические формулы, используя операторы для определения операций типа данных, выполняемых с одним или несколькими *операндами* (operand). В основном здесь используются *парные операторы* (binary operator), обрабатывающие два операнда, такие как `x - 3` или `5 * x`. Каждый операнд может быть любым выражением (возможно применение круглых скобок). Например, можно составить такие выражения `4 * (x - 3)` или `5 * x - 6`, и Python их поймет. Выражение — это директива выполнить последовательность операций; выражение — это также представление результирующего значения.



Анатомия выражения

Приоритет операторов (operator precedence). Выражение — это по сути запись последовательности операций, но в каком порядке должны быть применены операторы? У языка Python есть естественные и четкие правила *приоритета*, полностью определяющие этот порядок. Арифметические операции умножения и деления выполняются перед сложением и вычитанием; так, выражения `a - b * c` и `a - (b * c)` представляют ту же последовательность операций. При одинаковом приоритете арифметические операторы имеют *левосторонний порядок*, а это означает, что выражения `a - b - c` и `(a - b) - c` представляют ту же последовательность операций. Оператор возведения в степень `**` является исключением: он имеет *правосторонний порядок*. Это значит, что выражения `a ** b ** c` и `a ** (b ** c)` представляют ту же последовательность операций. Для переопределения правил можно использовать круглые скобки; вы вполне можете написать `a - (b - c)`, если это именно то, что нужно. В будущем вы можете встретить код Python, полностью полагающийся на правила приоритета, но в этой книге мы избегаем такого кода, используя круглые скобки. Если вам это интересно, то все подробности правил приоритета можно найти на сайте книги.

Оператор присвоения (assignment statement). Как мы определяем идентификатор, который будет переменной в коде Python? Как мы ассоциируем переменную со значением типа данных? В языке Python в обеих целях используется оператор присвоения. Когда в коде Python пишут `a = 1234`, то это не выражение математического равенства, это требование следующих *действий*.

- Определить идентификатор `a` как имя новой переменной (если таковой еще нет).
- Ассоциировать переменную `a` со значением `1234` целочисленного типа.

Правая сторона оператора присвоения может быть любым выражением. В таких случаях Python вычисляет выражение и ассоциирует переменную с левой стороны со значением, созданным выражением. Например, когда пишут `c = a + b`, имеют в виду такое действие: "ассоциировать переменную `c` с суммой значений, связанных с переменными `a` и `b`". Левая сторона оператора присвоения должна быть одиночной переменной. Таким образом, операторы `1234 = a` и `a + b = b + a` недопустимы в языке Python. *Короче говоря, значение оператора = в программе радикально отличается от такового в математическом уравнении.*

	a	b	c
a = 1234	1234		
b = 99	1234	99	
c = a + b	1234	99	1333

Ваша первая свободная трассировка

Свободная трассировка (informal trace). Эффективный способ отслеживания связанных с переменными значений подразумевает использование таблицы (как на рисунке), в каждой строке которой представлены значения после выполнения каждой операции. Такая называемая *трассировкой* (trace) таблица — проверенная временем методика понимания поведения программы. В этой книге мы используем трассировки повсюду.

Хотя подобные описания — вполне допустимый способ изучения кода Python в этом разделе, вначале следует более подробно рассмотреть то, как Python представляет значения типа данных, используя объекты, а затем вернуться к определениям в этом контексте. Несмотря на то что эти определения сложней приведенных выше, понять базовый механизм очень важно, поскольку он используется в Python практически всюду и готовит вас к объектно-ориентированному программированию, описанному в главе 3.

Объект (object). Все значения данных в программе Python представляются объектами и отношениями между объектами. Объект — это представление в машинной памяти значения определенного типа данных. Каждый объект характеризуется его *идентификатором*, *типом* и *значением*.

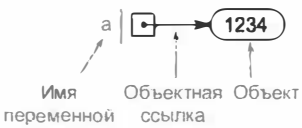
Идентификатор (identity) однозначно идентифицирует объект. Его можно считать адресом области в памяти компьютера, где хранится объект.

Тип (type) объекта полностью определяет его поведение — это набор значений, которые он может представлять, и набор операций, которые могут быть выполнены с ним.

Значение (value) объекта — это значение типа данных, которое оно представляет. Каждый объект хранит одно значение; например, объект типа `int` может хранить значение `1234`, или `99`, или `1333`. Различные объекты могут хранить то же значение. Например, один объект типа `str` мог бы хранить значение `'hello'`, а другой объект типа `str` также мог бы хранить то же значение `'hello'`. К объекту можно применить любые из определенных для его типа операций (и только этих операций). Например, можно умножить два объекта типа `int`, но не два объекта типа `str`.

Объектная ссылка (object reference) — это не более чем конкретное представление идентификатора объекта (адрес области памяти, где хранится объект). Программы Python используют объектные ссылки как для доступа к значению объекта, так и для манипулирования самими объектными ссылками.

- *Формальное определение объекта.* Теперь приведем более формальные определения для используемой терминологии.
- *Литерал* — директива создать объект, имеющий определенное значение.
- *Переменная* — имя объектной ссылки. На схеме показана *привязка* (binding) переменной к объекту.
- *Выражение* — директива выполнить указанные операции, создав объект, содержащий результирующее значение выражения.
- *Оператор присвоения* — директива привязать переменную с левой стороны оператора = к объекту, полученному в результате вычисления выражения с правой стороны (независимо от объекта, с которым переменная была связана ранее, если таковой вообще имелся).



Привязка переменной к объекту

Трассировка объектного уровня (object-level trace). Иногда для более полного понимания отслеживают объекты и ссылки. Это представленная на рисунке *трассировка объектного уровня*. В данном примере трассировка объектного уровня иллюстрирует действие трех операторов присвоения.

- Оператор `a = 1234` создает объект типа `int` со значением `1234`, а затем привязывает к этому объекту переменную `a`.
- Оператор `b = 99` создает объект типа `int` со значением `99`, а затем привязывает к этому объекту переменную `b`.
- Оператор `c = a + b` создает объект типа `int` со значением `1333` как сумму значений объектов типа `int`, связанных с переменными `a` и `b`, а затем привязывает к этому новому объекту переменную `c`.

Почти повсюду в этой книге используются более краткие и интуитивно понятные свободные трассировки, обсуждавшиеся только что, трассировки объектного

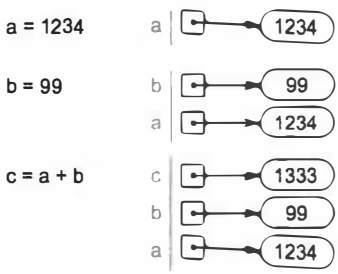
уровня используются только в тех ситуациях, когда они действительно обеспечивают более понятное представление описываемого вычисления.

Пример: инкремент переменной. Для первой проверки этих концепций рассмотрим следующий код, привязывающий переменную `i` к объекту типа `int` со значением `17`, а затем осуществляющий его инкремент (приращение):

```
i = 17
i = i + 1
```

Как можно заметить, второй оператор не имеет смысла как математическое равенство, но это вполне обычная операция в программировании. А именно: эти два оператора — директива предпринять следующие действия.

- Создать объект типа `int` со значением `17` и привязать переменную `i` к этому объекту.
- Вычислить результат выражения `i + 1` и создать новый объект типа `int` со значением `18`.
- Связать переменную `i` с этим новым объектом.



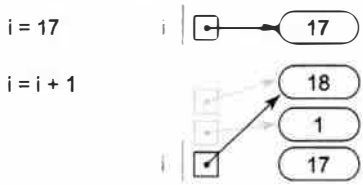
Ваша первая трассировка объектного уровня

Результаты выполнения каждого оператора можно проследить по трассировке.

Свободная трассировка

	i
i = 17	17
i = i + 1	18

Трассировка объектного уровня с новым объектом



Инкремент переменной

Этот простой фрагмент кода позволяет рассмотреть во всех подробностях два аспекта процесса создания объекта. Во-первых, оператор присвоения `i = i + 1` не изменяет значения никакого объекта. Он создает новый объект (с необходимым значением) и привязывает переменную `i` к этому новому объекту. Во-вторых, после выполнения оператора присвоения `i = i + 1` с объектом, содержащим значение `17` (а также с объектом, содержащим значение `1`), никакой переменной не связано. Ответственность за управление ресурсами памяти берет на себя Python. Когда

он обнаруживает, что программа больше неспособна обращаться к объекту, он автоматически освобождает занимаемую им область памяти.

Пример: обмен двух переменных. Для второй проверки этих концепций предположим, что следующий код *обменивает* переменные `a` и `b` (точнее, связанные с ними объекты):

```
t = a
a = b
b = t
```

С учетом, что переменные `a` и `b` связаны с объектами, имеющими два разных значения, `1234` и `99` соответственно, проследим этапы выполнения каждого оператора по трассировке.

- `t = a` присвоение переменной `t` значения переменной `a`. Таким образом, переменная `a` (ее объектная ссылка) присваивается переменной `t`. В результате переменные `a` и `t` будут привязаны к тому же объекту типа `int` со значением `1234`.
- `a = b` присвоение переменной `a` значения переменной `b`. Таким образом, переменная `b` (ее объектная ссылка) присваивается переменной `a`. В результате переменные `a` и `b` будут привязаны к тому же объекту типа `int` со значением `99`.
- `b = t` присвоение переменной `b` значения переменной `t`. Таким образом, переменная `t` (ее объектная ссылка) присваивается переменной `b`. В результате переменные `t` и `b` будут привязаны к тому же объекту типа `int` со значением `1234`.

Обмен двух переменных

Таким образом, происходит обмен ссылками: переменная `a` теперь привязана к объекту со значением `99`, а переменная `b` — к объекту со значением `1234`.

Предупреждение о сокращении. С этого момента мы зачастую будем сокращать описания операций Python, использующих переменные, объекты и объектные ссылки. Например, мы будем частично или полностью опускать фразы в квадратных скобках, как в примерах ниже. При описании оператора `a = 1234` мы могли бы написать так.

- "Привязка [переменной] a к [объекту типа int со значением] 1234."
- "Присвоение [переменной] a [ссылке на] [объект типа int со значением] 1234."

Точно так же после выполнения оператора a = 1234 мы могли бы написать так.

- "[Переменная] a привязана к [объекту типа int со значением] 1234."
- "[Переменная] a — ссылка на [объект типа int со значением] 1234."
- "Значением [объекта типа int, привязанного к] [переменной] a является 1234."

Так, при описании оператора c = a + b мы могли бы написать "c — это сумма a и b", вместо более точной, но длинной формулировки "переменная c связана с объектом, значением которого является сумма значений объектов, связанных с переменными a и b." Мы будем по возможности кратки, а точны не более, чем абсолютно необходимо.

Мы называем первую привязку переменной к объекту в программе *определением* и *инициализацией* переменной. Так, при описании оператора a = 1234 в первой трассировке мы могли бы сказать, что это "определяет переменную a и инициализирует ее значением 1234." В языке Python нельзя определить переменную, не инициализировав ее одновременно.

Мог бы возникнуть вопрос: различие между объектом и ссылкой на объект — это не просто педантизм? Фактически понимание различия между объектами и ссылками на них является ключом к овладению многими важными средствами программирования, включая *функции* (см. главу 2), *объектно-ориентированное программирование* (см. главу 3) и *структуры данных* (см. главу 4).

Далее мы рассмотрим подробности наиболее популярных типов данных (строки, целые числа, числа с плавающей точкой и логические значения), а также примеры кода, иллюстрирующие их использование. Для использования типа данных необходимо знать не только набор его значений, но и какие операции с ним можно выполнять, а также каковы соглашения для определения его литералов.

Строки. Строки представляет тип данных str. Он же используется для обработки текста. Значение объекта типа str — это последовательность *символов* (character). Для определения строкового литерала следует заключить последовательность символов в одинарные кавычки; например, объект 'ab' типа str представляет два символа — 'a' и 'b'. Существует много символов, но мы обычно ограничиваемся буквами, цифрами, знаками и символами отступа, такими как табуляция и новая строка. Для определения символов, которые в противном случае имели бы специальное значение, используется наклонная черта влево. Например, для определения символов табуляции, новой строки, наклонной черты влево и одинарной кавычки можно использовать *управляющие последовательности* (escape sequence) '\t', '\n', '\\' и '\' соответственно.

Тип данных `str`

Значение	Последовательность символов
Типичные литералы	`'Hello, World'` `'Python\'s'`
Операции	Конкатенация
Операторы	`+`

Используя оператор +, можно осуществить *конкатенацию* двух строк. Таким образом, оператор + получает в качестве операндов два объекта типа `str` и создает новый строковый объект, значением которого является последовательность символов из первой, а затем и из второй строки. Например, выражение `'123' + '456'` даст результат `'123456'`. Этот пример иллюстрирует существенно иное применения оператора + к двум объектам типа `str` (конкатенация строк) и к двум объектам типа `int` (сумма).

Типичные выражения с типом `str`

Выражение	Значение	Комментарий
`'Hello, ' + 'World'`	`'Hello, World'`	Конкатенация
`'123' + '456'`	`'123456'`	Не сумма
`'1234' + ' + ' + '99'`	`'1234 + 99'`	Две конкатенации
`'123' + 456`	Ошибка времени выполнения	Второй операнд не строка

Предупреждение о сокращении. С этого момента для обозначения *объекта типа* `str` мы используем термин *строка*, если полная форма фразы не является абсолютно необходимой. Мы также используем термин `'abc'` вместо *объект типа* `str` *со значением* `'abc'`.

Программа 1.2.1. Пример конкатенации строк (`ruler.py`)

```
import stdio

ruler1 = '1'
ruler2 = ruler1 + ' 2 ' + ruler1
ruler3 = ruler2 + ' 3 ' + ruler2
ruler4 = ruler3 + ' 4 ' + ruler3
stdio.writeln(ruler1)
stdio.writeln(ruler2)
stdio.writeln(ruler3)
stdio.writeln(ruler4)
```

```
% python ruler.py
1
1 2 1
1 2 1 3 1 2 1
1 2 1 3 1 2 1 4 1 2 1 3 1 2 1
```

Эта программа выводит линейные отрезки относительной длины. Энная строка вывода — это относительная длина меток на линейке с делениями по $1/2^n$ дюйма.

Конкатенация строк обладает достаточными возможностями, чтобы позволить решать некоторые нетривиальные вычислительные задачи. Так, например, программа 1.2.1 (`ruler.py`) вычисляет таблицу значений линейчатой функции, описывающей относительные длины меток на линейке. Одна из примечательных особенностей этого вычисления в том, что оно иллюстрирует, как простые короткие программы позволяют выработать огромные объемы вывода. Если дополнить эту программу очевидным способом, написав пять, шесть, семь строк и т.д., то можно заметить, что каждый раз в программу добавляется только два оператора, а размер ее вывода увеличивается вдвое. А именно, если программа выводит n строк, то n-ная строка содержит $2^n - 1$ чисел. Например, если, добавляя операторы таким образом в программу, написать 30 строк, то она попытается вывести больше 1 миллиарда чисел.

Линейчатая функция для n = 4

Теперь рассмотрим два вспомогательных механизма преобразования числа в строку и строки в число.

Преобразование чисел в строки при выводе. Язык Python предоставляет встроенную функцию `str()`, преобразующую числа в строки. Например, вызов функции `str(123)` вернет строку `'123'`, а `str(123.45)` — строку `'123.45'`. Если аргумент функции `stdio.write()` или `stdio.writeln()` имеет тип, отличный от `str`, то эти две функции автоматически вызывают функцию `str()` для преобразования аргумента в строку. Например, все вызовы функций `stdio.write(123)`, `stdio.write(str(123))` и `stdio.write('123')` выведут на экран `123`.

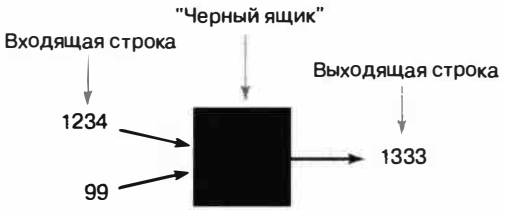
Чаще всего оператор конкатенации строк используют для объединения результатов вычисления при выводе функциями `stdio.write()` или `stdio.writeln()`, зачастую совместно с функцией `str()`, как в этом примере:

```
stdio.writeln(str(a) + ' + ' + str(b) + ' = ' + str(a+b))
```

Если `a` и `b` — объекты типа `int` со значениями `1234` и `99` соответственно, то этот оператор выведет строку `1234 + 99 = 1333`. Мы рассматриваем тип данных `str` первым только потому, что будем использовать подобный вывод в программах, обрабатывающих данные других типов.

Преобразование строк в числа при вводе. Язык Python предоставляет также встроенные функции для преобразования строк (таких, как вводимые аргументы командной строки) в числовые объекты. Для этого используются встроенные функции `int()` и `float()`. Например, код `int('1234')` в тексте программы эквивалентен использованию литерала `int 1234`. Если в качестве первого аргумента командной строки пользователь ввел `1234`, то код `int(sys.argv[1])` создаст объект типа `int` со значением `1234`. В этом разделе будет приведено несколько примеров применения этого преобразования.

При наличии этих средств представленная как "черный ящик" программа Python может не только получать строковые аргументы и выводить строковые результаты, но и интерпретировать эти строки как числа и использовать их для числовых вычислений.



Общий вид программы Python (пересмотренный)

Целые числа. Целые (или натуральные) числа представляет тип данных `int`. Для определения литерала типа `int` можно использовать последовательность цифр от `0` до `9`. Когда Python встречает целочисленный литерал, он создает объект типа `int`, хранящий указанное значение. Объекты типа `int` используют не только потому, что они часто встречаются в реальном мире, но и потому, что они естественным образом возникают при создании программы.

Язык Python предоставляет операторы для арифметических операций с целыми числами, включая `+` для сложения, `-` для вычитания, `*` для умножения, `//` для целочисленного деления, `%` для остатка и `**` для возведения в степень. Эти парные операторы получают как операнды два объекта типа `int` и создают в результате объект типа `int`. Язык Python предоставляет также унарные операторы `+` и `-` для определения знака целого числа. Все эти операторы работают точно так, как вы учили в средней школе (только оператор целочисленного деления дает в результате только целые числа): если объекты `a` и `b` имеют тип `int`, то результатом выражения `a // b` будет количество значений переменной `b` в значении переменной `a` без дробной части (деление без остатка), а результатом выражения `a % b` как раз и будет остаток от деления `a` на `b`. Например, `17 // 3` дает `5`, а `17 % 3` дает `2`. Попытка целочисленного деления или вычисления остатка с нулевым делителем приводит к ошибке `ZeroDivisionError`.

Тип данных `int`

Значение					Целые числа		
Типичные литералы		1234	99	0	1000000		
Операции	Знак	Сумма	Разница	Умножение	Целочисленное деление	Остаток	Степень
Операторы	`+ -`	`+`	`-`	`*`	`//`	`%`	`**`

Программа 1.2.2 (`intops.py`) иллюстрирует основные операции по манипулированию объектами типа `int`, такие как использование выражений с применением арифметических операторов. Здесь демонстрируется также использование встроенной функции `int()` для преобразования ввода из командной строки в объект типа `int`, а также использование встроенной функции `str()` для преобразования объекта типа `int` в строку для вывода.

Предупреждение о сокращении. С этого момента мы используем термин *целое число* для обозначения *объекта типа* `int`, если полная форма фразы не является необходимой. Мы также используем термин *значение* `123` вместо *объект типа* `int` *со значением* `123`.

Программа 1.2.2. Целочисленные операторы (`intops.py`)

```
import sys
import stdio

a = int(sys.argv[1])
b = int(sys.argv[2])

total = a + b
diff = a - b
prod = a * b
quot = a // b
rem = a % b
exp = a ** b

stdio.writeln(str(a) + ' + ' + str(b) + ' = ' + str(total))
stdio.writeln(str(a) + ' - ' + str(b) + ' = ' + str(diff))
stdio.writeln(str(a) + ' * ' + str(b) + ' = ' + str(prod))
stdio.writeln(str(a) + ' // ' + str(b) + ' = ' + str(quot))
stdio.writeln(str(a) + ' % ' + str(b) + ' = ' + str(rem))
stdio.writeln(str(a) + ' ** ' + str(b) + ' = ' + str(exp))
```

```
% python intops.py
1234 5
1234 +  5 = 1239
1234 -  5 = 1229
1234 *  5 = 6170
1234 // 5 = 246
1234 %  5 = 4
1234 ** 5 =
2861381721051424
```

Эта программа получает целочисленные аргументы командной строки `a` и `b`, использует их для иллюстрации целочисленных операторов и вывода результатов. Арифметика для целых чисел встроена в Python. Большая часть этого кода посвящена чтению ввода и записи вывода; фактическая арифметика находится в простых операторах в середине программы, присваивающих значения переменным `total`, `diff`, `prod`, `quot`, `rem` и `exp`.

В языке Python тип `int` имеет произвольно большой диапазон значений, ограниченный только объемом памяти, доступным в компьютере. Большинство других языков программирования ограничивают диапазон целых чисел. Например, язык программирования Java ограничивает целые числа диапазоном от -2^{31} (-2147483648) до $2^{31} - 1$ (2147483647). С одной стороны, программисты Python могут не волноваться, что целое число окажется слишком большим для диапазона допустимых; с другой стороны, им придется позаботиться о дефектной программе, заполняющей всю доступную память на их компьютере одним или несколькими чрезвычайно большими целыми числами.

Типичные выражения с типом `int`

Выражение	Значение	Комментарий
`99`	`99`	Целочисленный литерал
`+99`	`99`	Знак "плюс"
`-99`	`-99`	Знак "минус"
`5 + 3`	`8`	Сложение
`5 - 3`	`2`	Вычитание
`5 * 3`	`15`	Умножение
`5 // 3`	`1`	Без дробной части
`5 % 3`	`2`	Остаток
`5 ** 3`	`125`	Возведение в степень
`5 // 0`	Ошибка времени выполнения	Деление на нуль
`3 * 5 - 2`	`13`	`*` имеет приоритет
`3 + 5 // 2`	`5`	`//` имеет приоритет
`3 - 5 - 2`	`-4`	Левосторонний порядок
`(3 - 5) - 2`	`-4`	Лучший стиль
`3 - (5 - 2)`	`0`	Однозначно
`2 ** 2 ** 3`	`256`	Правосторонний порядок
`2 ** 1000`	`107150...376`	Произвольно большое

Предупреждение пользователям Python 2. В языке Python 3 у оператора `/` то же действие, что и у оператора деления с плавающей точкой, когда оба его операнда — целые числа. В языке Python 2 у оператора `/` то же действие, что и у оператора целочисленного деления `//`, когда оба его операнда — целые числа. Например, `17 / 2` дает `8.5` в Python 3 и `8` в Python 2. Для совместимости версий Python мы не используем в этой книге оператор `/` с двумя операндами типа `int`.

Числа с плавающей точкой. Для представления чисел с плавающей точкой, использующихся в научных и коммерческих приложениях, применяется тип данных `float`. Числа с плавающей точкой используются для представления вещественных чисел, но это не то же самое, что и вещественные числа! Есть бесконечно много вещественных чисел, но на цифровом компьютере можно представить число с плавающей точкой только с конечным количеством знаков. Числа с плавающей точкой действительно очень близки к вещественным числам, мы можем использовать их в приложениях, но зачастую придется мириться с тем фактом, что не всегда получится осуществить абсолютно точные вычисления.

Тип данных `float`

Значение				Вещественные числа	
Типичные литералы	`3.14159`	`6.022e23`	`2.0`	`1.4142135623730951`	
Операции	Сумма	Разница	Умножение	Деление	Степень
Операторы	`+`	`-`	`*`	`/`	`**`

Для определения литерала числа с плавающей точкой используется последовательность цифр с десятичной точкой. Например, литерал `3.14159` приблизительно представляет число p. В качестве альтернативы можно использовать литерал в экспоненциальной форме `6.022e23`; он представляет число 6.022×10^{23}. Подобно целым числам, в программах можно использовать эти соглашения для выражения литералов чисел с плавающей точкой, а также предоставлять их как строковые аргументы в командной строке.

Типичные выражения с типом `float`

Выражение	**Значение**	**Комментарий**
`3.14159`	`3.14159`	Литерал числа с плавающей точкой
`6.02e23`	`6.02e23`	Литерал числа с плавающей точкой
`3.141 + 2.0`	`5.141`	Сложение
`3.141 - 2.0`	`1.141`	Вычитание
`3.141 * 2.0`	`6.282`	Умножение
`3.141 / 2.0`	`1.5705`	Деление
`5.0 / 3.0`	`1.6666666666666667`	17 цифр точности
`3.141 ** 2.0`	`9.865881`	Возведение в степень
`1.0 / 0.0`	Ошибка времени выполнения	Деление на нуль
`2.0 ** 1000.0`	Ошибка времени выполнения	Слишком большое для представления
`math.sqrt(2.0)`	`1.4142135623730951`	Математическая функция модуля
`math.sqrt(-1.0)`	Ошибка времени выполнения	Квадратный корень отрицательного числа

Программа 1.2.3 (`floatops.py`) иллюстрирует основные операции с объектами типа `float`. Язык Python предоставляет операторы для чисел с плавающей точкой, включая `+` для сложения, `-` для вычитания, `*` для умножения, `/` для деления и `**` для возведения в степень. Эти операторы получают как операнды два объекта типа `float` и, как правило, создают в результате объект типа `float`. Программа 1.2.3 иллюстрирует также использование функции `float()` для преобразования строки в объект типа `float`, а также использование функции `str()` для преобразования объекта `float` в объект `str`.

Одной из первых проблем чисел с плавающей точкой является точность: `5.0/2.0` дает `2.5`, а `5.0/3.0` — `1.6666666666666667`. Как правило, у чисел с плавающей запятой 15–17 десятичных цифр точности. В разделе 1.5 рассматривается механизм

Python для контроля количества цифр в выводе, а до тех пор мы будем работать со стандартным форматом вывода Python. Хотя при использовании объектов типа `float` придется учитывать множество деталей, они позволяют создавать программы Python для всех видов вычислений, а не использовать калькулятор. Например, программа 1.2.4 (`quadratic.py`) демонстрирует применение объектов типа `float` в вычислении двух корней квадратного уравнения по квадратичной формуле.

Программа 1.2.3. Операторы чисел с плавающей точкой (`floatops.py`)

```
import sys
import stdio

a = float(sys.argv[1])
b = float(sys.argv[2])

total = a + b
diff = a - b
prod = a * b
quot = a / b
exp = a ** b

stdio.writeln(str(a) + ' + ' + str(b) + ' = ' + str(total))
stdio.writeln(str(a) + ' - ' + str(b) + ' = ' + str(diff))
stdio.writeln(str(a) + ' * ' + str(b) + ' = ' + str(prod))
stdio.writeln(str(a) + ' / ' + str(b) + ' = ' + str(quot))
stdio.writeln(str(a) + ' ** ' + str(b) + ' = ' + str(exp))
```

Эта программа получает аргументы командной строки `a` и `b` типа `float`, использует их для иллюстрации операций с числами с плавающей точкой и вывода результатов. Арифметика чисел с плавающей точкой встроена в Python. Подобно программе 1.2.2, большая часть этого кода осуществляет чтение ввода и запись вывода; фактическая арифметика находится в простых операторах, присваивающих значения переменным `total`, `diff`, `prod`, `quot` и `exp`.

```
% python floatops.py 123.456 78.9
123.456 + 78.9 = 202.356
123.456 - 78.9 = 44.556
123.456 * 78.9 = 9740.6784
123.456 / 78.9 = 1.5647148288973383
123.456 ** 78.9 = 1.0478827916671325e+165
```

Программа 1.2.4. Квадратичная формула (quadratic.py)

```
import math
import sys
import stdio

b = float(sys.argv[1])
c = float(sys.argv[2])

discriminant = b*b - 4.0*c
d = math.sqrt(discriminant)
stdio.writeln((-b + d) / 2.0)
stdio.writeln((-b - d) / 2.0)
```

Эта программа выводит корни уравнения $x^2 + bx + c$, используя квадратичную формулу. Например, корнями выражения $x^2 - 3x + 2$ будут 1 и 2, поскольку мы можем разложить уравнение как $(x - 1)(x - 2)$; корнями выражения $x^2 - x - 1$ будут ϕ и $\phi - 1$, где ϕ — это коэффициент золотого сечения, а корни выражения $x^2 + x + 1$ не будут вещественными числами.

```
% python quadratic.py -3.0 2.0
2.0
1.0

% python quadratic.py -1.0 -1.0
1.618033988749895
-0.6180339887498949

% python quadratic.py 1.0 1.0
Traceback (most recent call last):
  File "quadratic.py", line 9, in
<module>
    d = math.sqrt(discriminant)
ValueError: math domain error
```

Обратите внимание на использование функции `math.sqrt()`. Стандартный модуль `math` определяет тригонометрические, логарифмические и другие общие математические функции. Когда язык Python вызывает функцию, он создает значение, которое вычисляет функция. Модуль `math` можно использовать таким же образом, как и модуль `stdio` в программе `helloworld.py`: поместить оператор `import math` в начале программы, а затем вызывать функции, используя такой синтаксис, как `math.sqrt(x)`. Более подробная информация о лежащем в основе этого соглашения механизме приведена в разделе 2.1, а о модуле `math` — в конце этого раздела.

Как демонстрирует один из случаев запуска программы 1.2.4, проверки на ошибки она не осуществляет. В частности, она предполагает, что корни — вещественные числа. В противном случае она вызовет функцию `math.sqrt()` с отрицательным числом в качестве аргумента, что приведет к ошибке `ValueError` во время выполнения. Вообще-то, хороший стиль программирования подразумевает проверку условий, способных вызвать такие ошибки, и сообщение о них пользователю. Мы опишем, как это сделать, сразу после того, как рассмотрим еще несколько механизмов языка Python.

Предупреждение о сокращении. С этого момента мы используем термины *float* и *число с плавающей точкой* для обозначения *объекта типа* `float`, если полная форма фразы не является необходимой. Мы также используем термин *значение* `123.456` вместо *объект типа* `float` *со значением* `123.456`.

Логические значения. Логические значения `True` (истина) и `False` (ложь) представляет тип данных `bool`. У этого типа только два возможных значения и два соответствующих литерала: `True` и `False`. Логические операции используют операнды со значениями `True` и `False` и возвращают результат также с логическими значениями. Но эта простота обманчива: логический тип данных — основа информатики. Определенные для объектов типа `bool` операторы (`and`, `or` и `not`) известны как *логические операторы* (logical operator) и имеют общеизвестные определения:

- `a and b` дает `True`, если оба операнда `True`, и `False`, если хотя бы один из них `False`.
- `a or b` дает `False`, если оба операнда `False`, и `True`, если хотя бы один из них `True`.
- `not a` дает `True`, если `a` имеет значение `False`, и `False`, если `a` имеет значение `True`.

Тип данных `bool`

Значение		`true` или `false`	
Литералы		`True`	`False`
Операции	**и**	**Или**	**не**
Операторы	`and`	`or`	`not`

Несмотря на интуитивно понятный характер этих определений, имеет смысл полностью определить каждую возможность каждой операции в *таблице истинности* (truth-table), как показано ниже. У оператора `not` только один операнд: его результат для каждого из двух возможных значений операнда приведен во втором столбце. У каждого из операторов `and` и `or` по два операнда: результаты для каждой из четырех возможных комбинаций значений операндов этих операторов приведены в двух столбцах справа.

Таблица истинности логических операций

a	not a	A	b	a and b	a or b
False	True	False	False	False	False
True	False	False	True	False	True
		True	False	False	True
		True	True	True	True

Для создания произвольно сложных выражений, определяющих логическую функцию, эти операторы можно использовать с круглыми скобками (и согласно правилам приоритета). Приоритет оператора `not` выше, чем у оператора `and`, приоритет которого, в свою очередь, выше, чем у оператора `or`.

Зачастую та же функция предстает в разных "обличьях". Например, выражения `(a and b)` и `not (not a or not b)` эквивалентны. Для доказательства этого можно использовать таблицу истинности.

Таблица истинности доказывает идентичность выражений `a and b` и `not (not a or not b)`

a	b	a and b	not a	not b	not a or not b	not (not a or not b)
False	False	False	True	True	True	False
False	True	False	True	False	True	False
True	False	False	False	True	True	False
True	True	True	False	False	False	True

Исследование выражений подобного вида известно как *Булева логика*. Эта область математики фундаментальна для компьютеров: она играет ключевую роль при проектировании и работе компьютерных аппаратных средств, а также является отправной точкой для теоретических основ вычисления. В нынешнем контексте логические выражения интересны тем, что они позволяют контролировать поведение программ. Как правило, в условии определяется логическое выражение, а код программы проектируется так, что если результат выражения истинен, то выполняется один набор его операторов, в противном случае — другой набор операторов. (Этой теме посвящен раздел 1.3.)

Предупреждение о сокращении. С этого момента мы используем термин *логический* для обозначения *объекта типа* `bool`, если полная форма фразы не является необходимой. Мы также используем термин `True` или *истина* вместо *объект типа* `bool` *со значением* `True` и `False` или ложь вместо *объект типа* `bool` *со значением* `False`.

Сравнения. Некоторые операторы *смешанного типа* (mixed-type) получают операнды одного типа, а возвращают результат другого типа. Важнейшими операторами этого вида являются *операторы сравнения* ==, !=, <, <=, > и >=, определенные

и для целых чисел, и для чисел с плавающей точкой, а возвращают они логический результат. Поскольку операторы определяются только для типов данных, каждый из этих операторов существует в нескольких версиях. Оба операнда должны иметь совместимый тип. Результат всегда логический.

Результатом сравнения операндов типа `int` является тип `bool`

Оператор	Значение	`True`	`False`
`==`	Равно	`2 == 2`	`2 == 3`
`!=`	Не равно	`3 != 2`	`2 != 2`
`<`	Меньше	`2 < 13`	`2 < 2`
`<=`	Меньше или равно	`2 <= 2`	`3 <= 2`
`>`	Больше	`13 > 2`	`2 > 13`
`>=`	Больше или равно	`3 >= 2`	`2 >= 3`

Типичные выражения сравнения

Неотрицательный дискриминант?	`(b*b - 4.0*a*c) >= 0.0`
Начало столетия?	`(year % 100) == 0`
Правильный месяц?	`(month >= 1) and (month <= 12)`

Даже не вникая в подробности представления чисел, вполне понятно, что операции с различными типами действительно весьма отличаются. Например, одно дело сравнение двух целых чисел (`2 <= 2`) на неравенство, и совсем другое — сравнение двух чисел с плавающей точкой (`2.0 <= 0.002e3`). Тем не менее эти операции широко используются при создании кода проверки таких условий, как `(b*b - 4.0*a*c) >= 0.0`, что нередко бывает необходимо.

Приоритет операторов сравнения ниже, чем у арифметических операторов, но выше, чем у логических операторов, следовательно, в таком выражении, как `(b*b - 4.0*a*c) >= 0.0`, круглые скобки не нужны, и вполне можно написать выражение `month >= 1 and month <= 12` без круглых скобок, чтобы проверить принадлежность значения переменной `month` к диапазону от 1 до 12. (Тем не менее лучше использовать скобки.)

Совместно с булевой логикой операции сравнения позволяют программам Python принимать решения. Программа 1.2.5 (`leapyear.py`) демонстрирует применение логических выражений и операций сравнения для вычисления, является ли заданный год високосным. В конце этого раздела можно найти другие примеры в упражнениях. Роль логических выражений в более сложных программах будет продемонстрирована в разделе 1.3.

Программа 1.2.5 иллюстрирует также такое особое свойство логических операторов, как *вычисление по сокращенной схеме*: оператор `and` вычисляет второй операнд, *только если* первый операнд `True`; оператор `or` вычисляет второй операнд, *только если* первый операнд `False`. Например, в программе `leapyear.py`

Python вычисляет выражение `(year % 100) != 0`, только если год делится на 4, и выражение `(year % 400) != 0`, только если год делится на 100.

Программа 1.2.5. Високосный год (leapyear.py)

```
import sys
import stdio

year = int(sys.argv[1])

isLeapYear = (year % 4 == 0)
isLeapYear = isLeapYear and ((year % 100) != 0)
isLeapYear = isLeapYear or ((year % 400) == 0)

stdio.writeln(isLeapYear)
```

Эта программа проверяет, соответствует ли целое число високосному году в григорианском календаре. Год считается високосным, если он делим на 4 (2004), если он не делим на 100 (т.е. это не 1900) и если он не делим на 400 (как 2000).

```
% python leapyear.py 2016
True
% python leapyear.py 1900
False
% python leapyear.py 2000
True
```

Функции и API. Как можно заметить, во многих задачах программирования используются не только встроенные операторы, но и *функции* (function), также выполняющие полезные операции. Различают три вида функций: *встроенные функции* (такие, как `int()`, `float()` и `str()`), которые можно использовать в любой программе Python непосредственно, *стандартные функции* (такие, как `math.sqrt()`), они определены в стандартном модуле Python и доступны в любой программе, импортировавшей модуль, и *функции сайта книги* (такие, как `stdio.write()` и `stdio.writeln()`), они определены в модулях нашей книги и доступны для использования после того, как модули станут доступны для Python и будут импортированы. Общее количество доступных встроенных функций, стандартных функций и функций сайта книги очень велико. По мере изучения программирования вы научитесь использовать все больше этих функций, но вначале лучше ограничиться небольшим набором. В данной главе уже использовались функции для вывода на экран, для преобразования данных одного типа в другой и для вычисления математических функций. В этом разделе рассматриваются некоторые

более полезные функции. В следующих главах рассматривается не только использование других функций, но и определение собственных.

Для удобства информацию о функциях и их использовании можно объединить в таблицы. Могут быть объединены, например, встроенные функции, функции сайта книги из модуля `stdio` и стандартные функции из модулей Python `math` и `random`. Такая таблица известна как *интерфейс прикладных программ* (Application Programming Interface — API). В первом столбце определена информация, необходимая для использования функции, включая ее имя и количество аргументов; во втором столбце описывается ее цель.

Для вызова функции в коде следует ввести ее имя и отделяемые запятыми *аргументы* (argument) в круглых скобках. Когда Python выполняет программу, мы говорим, что он *вызывает* (call) функцию с переданными ей аргументами и что функция *возвращает* (return) значение. Точнее, функция возвращает ссылку на содержащий значение объект. Вызов функции — это выражение, поэтому его можно использовать таким же образом, как переменные и литералы для создания более сложных выражений. Например, вполне можно создавать такие выражения, как `math.sin(x) * math.cos(y)` и т.д. Для аргумента также можно использовать выражение — Python вычислит его и передаст результат как аргумент функции. Так, вполне можно написать такой код, как `math.sqrt(b*b - 4.0*a*c)`, и Python поймет, что вы имеете в виду.

У функции могут быть *стандартные значения* (default value) для необязательных аргументов. Примером является функция `math.log()` — она получает основание логарифма как необязательный второй аргумент. Если не определить второй аргумент, функция вычислит натуральный логарифм (с основанием `e`).

Типичные вызовы функции

Вызов функции	Возвращаемое значение	Комментарий
`abs(-2.0)`	`2.0`	Встроенная функция
`max(3, 1)`	`3`	Встроенная функция с двумя аргументами
`stdio.write('Hello')`		Функция сайта книги (с побочным эффектом)
`math.log(1000, math.e)`	`6.907755278982137`	Функция модуля `math`
`math.log(1000)`	`6.907755278982137`	Второй аргумент имеет стандартное значение
`math.sqrt(-1.0)`	Ошибка времени выполнения	Квадратный корень отрицательного числа
`random.random()`	`0.3151503393010261`	Функция модуля `random`

API для некоторых общепринятых функций Python

Вызов функции	Описание
	Встроенные функции
`abs(x)`	Абсолютное значение `x`
`max(a, b)`	Максимальное из значений `a` и `b`
`min(a, b)`	Минимальное из значений `a` и `b`
	Функции из модуля `stdio` с сайта книги для стандартного устройства вывода
`stdio.write(x)`	Выводит `x` на стандартное устройство вывода
`stdio.writeln(x)`	Выводит на стандартное устройство вывода `x`, а затем символ новой строки *Примечание 1*: применим любой тип данных (он будет автоматически преобразован в тип `str`) *Примечание 2*: если никакой аргумент не передан, стандартным значением `x` будет пустая строка
	Стандартные функции модуля Python `math`
`math.sin(x)`	Синус `x` (в радианах)
`math.cos(x)`	Косинус `x` (в радианах)
`math.tan(x)`	Тангенс `x` (в радианах)
`math.atan2(y, x)`	Полярный угол точки (`x`, `y`)
`math.hypot(x, y)`	Евклидово расстояние между исходной точкой и (`x`, `y`)
`math.radians(x)`	Преобразование `x` из градусов в радианы
`math.degrees(x)`	Преобразование `x` из радиан в градусы
`math.exp(x)`	Экспоненциальная функция $x(e^x)$
`math.log(x, b)`	Логарифм `x` по основанию `b` ($\log_b x$) (стандартное значение аргумента `b` — e, т.е. натуральный логарифм)
`math.sqrt(x)`	Квадратный корень `x`
`math.erf(x)`	Функция ошибок `x`
`math.gamma(x)`	Гамма-функция `x`
	Примечание: модуль `math` предоставляет также обратные функции `asin()`, `acos()` и `atan()`, а также константы `e` (2,718281828459045) и `pi` (3,141592653589793)
	Стандартные функции модуля Python `random`
`random.random()`	Случайное число типа `float` в интервале [0, 1)
`random.randrange(x, y)`	Случайное число типа `int` в интервале [`x`, `y`), где `x` и `y` — целые числа

За тремя исключениями все это *чистые функции* (pure function); при передаче тех же аргументов они всегда возвращают то же значение, не создавая заметного *побочного эффекта* (side effect). Функция `random.random()` таковой не является, поскольку при каждом вызове она потенциально возвращает иное значение; функции `stdio.write()` и `stdio.writeln()` также не являются чистыми, поскольку они имеют побочный эффект — вывод строк на стандартное устройство вывода.

Модуль `math` определяет также константные переменные `math.pi` (π) и `math.e` (e), чтобы их можно было использовать в программах по именам. Например, вызов

функции `math.sin(math.pi/2)` возвращает `1.0` (поскольку функция `math.sin()` получает аргумент в радианах), а вызов функции `math.log(math.e)` возвращает `1.0` (поскольку стандартным аргументом для основания функции `math.log()` является e).

Эти API типичны для сетевой документации и являются стандартом в современном программировании. В сети есть обширная документация по API Python, используемая профессиональными программистами, и она доступна (если интересно) на сайте нашей книги. Вам не обязательно обращаться к сетевой документации, чтобы понять код этой книги или писать подобный код, поскольку мы демонстрируем и объясняем в тексте все используемые в API функции, а также приводим их описание в приложении.

В главах 2 и 3 описано, как разработать собственные API и реализовать их функции для использования.

Предупреждение о сокращении. С этого момента мы сокращаем свои описания операторов Python, применяющих функции и вызовы функции. Например, мы часто будем опускать заключенные в скобки фразы в таких описаниях вызова функций, например `math.sqrt(4.0)`, как следующее: "вызов функции `math.sqrt(4.0)` возвращает [ссылку на] [объект типа `float` со значением] `2.0`." Таким образом, мы могли бы написать "вызов функции `sqrt(16.0)` возвращает `4.0`" вместо более точного, но более подробного "после передачи функции `math.sqrt()` ссылки на объект типа `float` со значением `16.0` она вернула ссылку на объект типа `float` со значением `4.0`." Мы также используем термин *возвращаемое значение* (return value) для описания объекта, ссылку на который возвращает функция.

Преобразование типов Типичные задачи программирования подразумевают обработку данных нескольких типов. Всегда следует знать тип данных, обрабатываемых вашей программой, поскольку только при знании типа можно точно знать, какие значения устанавливать и какие операции можно выполнять с каждым объектом. В частности, нередко приходится преобразовывать данные из одного типа в другой. Предположим, например, что необходимо вычислить среднее значение четырех целых чисел: `1`, `2`, `3` и `4`. Естественно, на ум приходит выражение `(1+2+3+4) / 4`, но оно не дает желаемого результата во многих языках программирования из-за соглашений преобразования типов. Действительно, как уже упоминалось, языки Python 3 и Python 2 дают разные результаты при вычислении этого выражения. Таким образом, это достойный пример для рассмотрения данной темы.

Причина проблемы в том факте, что операнды — целые числа, но вполне естественно ожидать результат типа `float`, поэтому в некоторый момент понадобится преобразование целого числа в число с плавающей точкой. Для этого в языке Python существуют два способа.

Явное преобразование типов (explicit type conversion). Этот подход подразумевает применение функции, получающей аргумент одного типа (преобразуемый объект) и возвращающей объект другого типа. Для преобразования строк в целые числа или числа с плавающей точкой (и наоборот) имеются встроенные функции, например `int()`, `float()` и `str()`. Безусловно, это наиболее популярные функции, но для преобразования между целыми числами и числами с плавающей точкой можно также использовать дополнительную функцию `round()`, как показано в API далее. Для преобразования типа `float` в целое число можно использовать функцию `int(x)`, а можно `int(round(x))`. Для преобразования целого числа в тип `float` можно использовать функцию `float(x)`. Таким образом, выражение `float(1+2+3+4) / float(4)` дает результат `2.5` и в Python 3, и в Python 2, как и ожидалось.

Типичные преобразования типов

Выражение	Значение	Тип
Явное		
`str(2.718)`	`'2.718'`	`str`
`str(2)`	`'2'`	`str`
`int(2.718)`	`2`	`int`
`int(3.14159)`	`3`	`int`
`Float(3)`	`3.0`	`float`
`int(round(2.718))`	`3`	`int`
Неявное		
`3.0 * 2`	`6.0`	`float`
`10 / 4.0`	`2.5`	`float`
`math.sqrt(4)`	`2.0`	`float`

Предупреждение пользователям Python 2. В языке Python 3 функция `round(x)` возвращает целое число, а в Python 2 — число с плавающей точкой. В этой книге мы всегда используем выражение `int(round(x))` для округления числа с плавающей точкой `x` до ближайшего целого числа, чтобы код работал и в Python 3, и в Python 2.

API для некоторых встроенных функций преобразования типов

Вызов функции	Описание
`str(x)`	Преобразование объекта `x` в строку
`int(x)`	Преобразование строки `x` в целое число или числа с плавающей точкой `x` в усеченное целое число
`float(x)`	Преобразование строки или целого числа `x` в число с плавающей точкой
`round(x)`	Ближайшее целое к числу `x`

Неявное преобразование типов (из `int` *во* `float`*).* Целое число можно использовать там, где ожидается тип `float`, поскольку при необходимости Python автоматически преобразует целые числа в числа с плавающей точкой. Например, выражение `10/4.0` даст `2.5`, поскольку число `4.0` имеет тип `float`, и оба операнда должны иметь одинаковый тип, число `10` преобразуется в тип `float`, а результатом деления двух чисел типа `float` будет тип `float`. Другой пример: вызов `math.sqrt(4)` дает `2.0`, поскольку `4` преобразуется в тип `float`, как и ожидает функция `math.sqrt()`, которая затем и возвращает значение типа `float`. Это *автоматическое преобразование* (automatic promotion) или *приведение типа данных* (coercion). Python реализует автоматическое преобразование потому, что это может быть сделано без потери информации. Но, как можно заметить, есть проблемы. Как уже упоминалось, Python 3 автоматически преобразует каждый целочисленный операнд оператора `/` в тип `float`, но Python 2 так не поступает. Таким образом, выражение `(1+2+3+4)/4` дает в Python 3 значение `2.5` типа `float`, а в Python 2 — значение `2` типа `int`.

Концепция автоматического преобразования не нужна, если всегда использовать такие функции, как `int()` и `float()`, для явного преобразования типов; некоторые программисты вообще стараются избегать автоматического преобразования когда бы то ни было. В этой книге мы полагаемся на автоматическое преобразование, поскольку так код получается короче и понятнее. *Но при делении чисел оператором `/` мы всегда принимаем меры, чтобы по крайней мере один из двух операндов имел тип* `float`. В наших примерах, когда Python вычисляет выражение `(1+2+3+4)/4.0`, он вызывает автоматическое преобразование в тип `float` для первого операнда и дает значение `2.5` типа `float`, как и ожидается. Эта практика гарантирует, что наш код будет правильно работать и в Python 3, и в Python 2 (и во многих других языках). Напомним, что в этой книге мы не используем оператор `/`, когда оба операнда — целые числа.

Начинающих программистов необходимость преобразования типов зачастую раздражает, но опытные программисты знают, что внимание к типам данных — залог успеха в программировании и способ избежать ошибок. В известном инциденте 1985 года французская ракета взорвалась в воздухе из-за проблемы преобразования типов. Хотя ошибка в вашей программе может и не привести к взрыву, все равно имеет смысл уделить время изучению преобразования типов. Написав несколько программ, вы убедитесь, что понимание типов данных помогает не только писать компактный код, но и делать свои намерения явными, избегая ошибок в нюансах.

Фото: ESA

Взрыв ракеты Ariane 5

Резюме. *Тип данных* — это набор значений и операций, определенных для этих значений. Python предоставляет встроенные типы данных `bool`, `str`, `int` и `float`; другие типы используются в книге далее. В коде Python операторы и выражения используются как в знакомых математических выражениях. Тип `bool` предназначен для работы с логическими значениями; тип `str` — для последовательностей символов; а типы `int` и `float` — для чисел.

Тип `bool` (включающий логические операторы `and`, `or` и `not`) является основой логики принятия решений в программах Python, когда он используется вместе с операторами сравнения ==, `!=`, <, <=, > и >=. Логические выражения используются в условиях таких управляющих выражений Python, как `if` и `while`, рассматриваемых в следующем разделе.

Числовые типы, встроенные функции, а также функции, определенные в стандартных модулях Python, в модулях расширения и в наших книжных модулях, позволяют использовать Python как мощный математический калькулятор. Мы составляем арифметические выражения, используя, встроенные операторы +, -, *, /, //, % и **, а также вызовы функций Python.

Хотя программы в этом разделе являются элементарными по сравнению с таковыми в следующем разделе, этот вид программ весьма полезен в своем роде. Вы будете регулярно использовать типы данных и простые математические функции в программировании на языке Python, поэтому потраченные на их понимание усилия не пройдут даром.

Интерактивный Python. Действительно, язык Python можно использовать как калькулятор. Для этого введите в окне терминала команду `python` (только слово `python` без последующего имени файла). Python идентифицирует себя и выведет приглашение к вводу >>>. Теперь можно вводить операторы, и Python выполнит их. Либо можно ввести выражение, и Python вычислит и выведет результирующее значение. Либо можно ввести `help()` и получить доступ к обширной интерактивной документации Python. Некоторые из примеров этого приведены ниже (полужирным выделено то, что вы вводите). Обращение к документации — это удобный способ проверки новых конструкций и изучения интересующих вас модулей и функций. В некоторых из пунктов раздела "Вопросы и ответы" предлагается сделать именно это.

```
% python
...
>>> 1 + 2
3
>>> a = 1
>>> b = 2
>>> a + b
3
```

```
>>> import math
>>> math.sqrt(2.0)
1.4142135623730951
>>> math.e
2.718281828459045
>>>

% python
...
>>> import math
>>> help(math)

Help on module math:

NAME
  Math

DESCRIPTION
  This module is always available. It provides
  access to the mathematical functions defined
  by the C standard.

FUNCTIONS
  acos(...)
    acos(x)
    Return the arc cosine (in radians) of x.
...
  sqrt(...)
    sqrt(x)
    Return the square root of x
...
DATA
  e = 2.718281828459045
  pi = 3.141592653589793
```

Вопросы и ответы

Строки

Строки в Python 2. Язык Python 2 использует кодировку символов ASCII вместо Unicode. ASCII — устаревший стандарт, поддерживающий 128 символов[1], включая английский алфавит, цифры и знаки. Python 2 предоставляет отдельный тип данных `unicode` для строк, состоящих из символов Unicode, но многие библиотеки Python 2 не поддерживают его.

Как Python хранит строки внутренне?

Строки — это последовательности символов в кодировке *Unicode* — современном стандарте для символов текста. Unicode поддерживает более 100 000 символов для более чем 100 языков плюс математические и музыкальные символы.

Какой тип данных Python представляет символы?

В языке Python нет никакого специального типа данных для символов. Символ — это просто строка, состоящая из одного символа, например `'A'`.

Можно ли использовать такие операторы сравнения, как == и <, или такие встроенные функции, как `max()` и `min()`, для сравнения строк?

Да. Неофициально для сравнения двух строк Python использует *лексикографический порядок* (lexicographic order), как у слов в предметном указателе книги или в словаре. Например, слова `'hello'` и `'hello'` равны, `'hello'` и `'goodbye'` неравны и `'goodbye'` меньше, чем `'hello'`. См. раздел "Вопросы и ответы" в конце раздела 4.2.

Можно ли использовать для строковых литералов двойные кавычки вместо одинарных?

Да. Литералы `'hello'` и `"hello"`, например, идентичны. Двойные кавычки полезны при определении строк, содержащих одинарные кавычки, без применения для них управляющих последовательностей. Например, строковые литералы `'Python\'s'` и `"Python's"` идентичны. Для многострочного текста можно также использовать тройные кавычки. Например, следующий код создает строку из двух строк и присваивает ее переменной `s`:

```
s = """Python's "triple" quotes are useful to
specify string literals that span multiple lines
"""
```

[1] Вообще-то 256. — *Примеч. ред.*

В этой книге мы не используем парные или тройные кавычки для разграничения строковых литералов.

Целые числа

Как Python хранит целые числа внутренне?

Самое простое представление небольших положительных целых чисел в *двоичной системе счисления* — это представление каждого целого числа в фиксированном объеме машинной памяти.

Что такое двоичная система счисления?

Вы, вероятно, учили это в средней школе. В двоичной системе счисления целое число представляется последовательностью *битов*. Бит — это одна двоичная цифра (0 или 1) или основа представления информации на компьютере. В данном случае биты — показатели степеней числа 2. А именно: последовательность битов $b_n b_{n-1} \ldots b_2 b_1 b_0$ представляет целое число (тип integer)

$$b_n 2^n + b_{n-1} 2^{n-1} + \ldots b_2 2^2 + b_1 2^1 + b_0 2^0$$

Например, двоичное число 1100011 представляет целое десятичное число

$$99 = 1 \cdot 64 + 1 \cdot 32 + 0 \cdot 16 + 0 \cdot 8 + 0 \cdot 4 + 1 \cdot 2 + 1 \cdot 1.$$

Куда более знакома *десятичная система счисления* — то же самое, но используются цифры от 0 до 9 и степени числа 10. Преобразование чисел из двоичной системы в десятичную (и наоборот) является интересной вычислительной задачей, рассматриваемой в следующем разделе. Для небольших целых чисел Python использует фиксированное количество битов, как правило, продиктованное конструкцией используемого компьютера (обычно 32 или 64). Например, целое число 99 может быть представлено 32 битами 00000000000000000000000001100011.

Как насчет отрицательных чисел?

Небольшие отрицательные числа хранятся в соответствии с соглашением *двоичного кода* (two's complement), которое нет смысла рассматривать подробно. Размер "небольших" чисел зависит от конкретной компьютерной системы. На 32-разрядных машинах "небольшими" обычно считаются числа в диапазоне от –2147483648 (-2^{31}) до 2147483647 ($2^{31} - 1$). На 64-битовых машинах к "небольшим" относят диапазон от -2^{63} до $2^{63} - 1$, так что "небольшие" не так уж и малы! Если целое число не является "небольшим", то Python автоматически использует более сложное представление, диапазон которого ограничивается только объемом памяти, доступной на компьютере. Обратите внимание,

что подробности этих внутренних представлений скрываются от ваших программ, поэтому вы можете использовать их с различными представлениями, не заботясь о смене представления.

К чему приведет вычисление выражения `1 / 0` в Python?

К ошибке `ZeroDivisionError` во время выполнения. *Примечание*: проще всего находить ответы на такие вопросы в интерактивном режиме Python. Опробуйте его!

Как операторы целочисленного деления `//` и остатка `%` работают с отрицательными операндами?

Опробуйте и посмотрите! Выражение `-47 // 5` дает `-10`, а `-47%5` — `3`. В общем, оператор целочисленного деления `//` возвращает целочисленное частное; т.е. частное округляется к минус бесконечности. Поведение оператора остатка `%` сложнее. В языке Python, если `a` и `b` — целые числа, то результатом выражения `a % b` будет целое число с тем же знаком, что и у `b`. Это подразумевает, что `b * (a // b) + a % b == a` для любых целых чисел `a` и `b`. В некоторых других языках (таких, как Java) выражение `a % b` дает целое число с таким же знаком, как у `a`.

Как с отрицательными операндами работает оператор возведения в степень `**`?

Опробуйте и убедитесь лично. Обратите внимание, что приоритет оператора `**` выше, чем у унарных операторов `+` и `-` слева, но ниже, чем у унарных операторов `+` и `-` справа. Например, выражение `-3 ** 4` дает `-81` (а не `81`). Кроме того, может получиться объект другого типа. Например, результатом выражения `10 ** -2` будет значение `0.01` типа `float`, а выражение `(-10) ** (10 ** -2)` даст в Python 3 комплексное число (а в Python 2 ошибку времени выполнения).

Почему выражение `10 ^ 6` дает `12`, а не `1000000`?

Оператор `^` — это не оператор возведения в степень, как вы, возможно, подумали. В этой книге данный оператор не используется. Если необходимо литеральное `1000000`, то можно использовать выражение `10 ** 6`, но применение выражения, требующего вычисления, весьма расточительно, когда достаточно литерала.

Целые числа в Python 2. Python 2 предоставляет для целых чисел два отдельных типа: `int` (для небольших целых чисел) и `long` (для больших). Язык Python 2 автоматически преобразует тип `int` в тип `long` при каждой необходимости.

Числа с плавающей точкой

Почему тип для вещественных чисел называется `float`?

Десятичная точка может "плавать" (float) между цифрами, составляющими вещественное число. У целых чисел десятичная точка (неявная), напротив, жестко зафиксирована после наименьшей значащей цифры.

Как Python хранит числа с плавающей точкой внутренне?

Вообще, Python использует вполне естественное для компьютерных систем представление. Большинство современных компьютерных систем хранит числа с плавающей точкой согласно стандарту IEEE 754. Этот стандарт определяет, что число с плавающей точкой хранится в трех областях памяти: знак, мантисса и экспонента. Если вам интересны детали, обратитесь к сайту книги. Стандарт IEEE 754 определяет также специальные значения с плавающей точкой: положительный нуль, отрицательный нуль, положительная бесконечность, отрицательная бесконечность и `NaN` (не число). Например, результатом выражения `-0.0 / 3.0` должно быть значение `-0.0`, выражения `1.0 / 0.0` — положительная бесконечность, а выражения `0.0 / 0.0` — значение `NaN`. Для положительной и отрицательной бесконечностей в некоторых простых вычислениях можно использовать (довольно необычные) выражения `float('inf')` и `float('-inf')`, но эту часть стандартов IEEE 754 язык Python не поддерживает. Например, в языке Python выражение `-0.0 / 3.0` дает `-0.0`, но выражения `1.0 / 0.0` и `0.0 / 0.0` приводят к ошибке `ZeroDivisionError` во время выполнения.

Мне кажется, что пятнадцать цифр в числе с плавающей точкой — это достаточно много. Должен ли я волноваться о точности?

Да, поскольку вы привыкли к математике на основании вещественных чисел с бесконечной точностью, тогда как компьютер всегда имеет дело с приближениями. Например, согласно стандарту IEEE 754, для чисел с плавающей точкой выражение `(0.1 + 0.1 == 0.2)` дает `True`, но выражение `(0.1 + 0.1 + 0.1 == 0.3)` дает `False`! В научных вычислениях эта проблема вполне обычна, а начинающим программистам стоит избегать сравнения чисел с плавающей точкой на равенство.

При выводе чисел типа `float` так раздражают все эти лишние цифры. Можно ли заставить функции `stdio.write()` и `stdio.writeln()` выводить только две или три цифры после десятичной точки?

Эту задачу решает книжная функция `stdio.writef()`. Она подобна простой функции форматированного вывода в языке C и во многих других языках. Она

обсуждается в разделе 1.5, а до тех пор придется мириться с дополнительными цифрами (что вовсе не так плохо, поскольку это помогает привыкнуть к различным типам чисел).

Можно ли применить оператор целочисленного деления `//` к двум операндам типа `float`?

Да, произойдет целочисленное деление его операндов. Таким образом, в результате получится частное без цифр после позиции десятичной точки. В этой книге мы не используем оператор целочисленного деления для операндов типа `float`.

Что возвратит функция `round()`, если аргумент имеет дробную часть `0.5`?

В Python 3 она возвращает ближайшее целое число; таким образом, вызов `round(2.5)` даст 2, `round(3.5)` — 4, а `round(-2.5)` — `-2`. Но в Python 2 функция `round()` округляет от нуля (и возвращает тип `float`), поэтому вызов `round(2.5)` дает `3.0`, `round(3.5)` — `4.0` и `round(-2.5)` — `-3.0`.

Могу ли я сравнить значение типа `float` со значением типа `int`?

Без преобразования типов — нет, но Python осуществляет необходимое преобразование типов автоматически. Например, если `x` — целое число 3, то выражение `(x < 3.1)` дает в результате значение `True`, поскольку Python преобразует целое число 3 в значение `3.0` типа `float`, а затем сравнит значения `3.0` и `3.1`.

Есть ли в модуле `math` языка Python функции для других тригонометрических функций, таких как арксинус, гиперболический синус и секанс?

Да, модуль Python `math` включает обратные тригонометрические и гиперболические функции. Но функций для секанса, косеканса и котангенса нет, поскольку для их вычисления вполне можно использовать функции `math.sin()`, `math.cos()` и `math.tan()`. Выбор включаемых в API функций — это компромисс между удобством наличия каждой функции и раздражающей необходимостью поиска одной из немногих нужных в длинном списке имеющихся. Никакой выбор не удовлетворит всех пользователей, а их множество. Обратите внимание на то, что даже в имеющихся API много избыточности. Например, вместо вызова `math.tan(x)` вполне можно использовать `math.sin(x) / math.cos(x)`.

Что случится при обращении к переменной, не привязанной к объекту?

Ошибка `NameError` во время выполнения.

Как определить тип переменной?

Это каверзный вопрос. В отличие от переменных многих других языков программирования (таких, как Java), у переменной Python нет типа. В языке Python есть *объект*, к которому привязана переменная, а у объекта есть тип. Ту же переменную можно связать с объектами других типов, как в следующем фрагменте кода:

```
x = 'Hello, World'
x = 17
x = True
```

Но это, как правило, плохая идея.

Как можно определить тип, идентификатор и значение объекта?

Для этого Python предоставляет встроенные функции. Функция `type()` возвращает тип объекта; функция `id()` — идентификатор объекта; функция `repr()` — однозначное строковое представление объекта.

```
>>> import math
>>> a = math.pi
>>> id(a)
140424102622928
>>> type(a)
<class 'float'>
>>> repr(a)
'3.141592653589793'
```

В обычном программировании вы редко будете использовать эти функции, но при отладке они могут оказаться полезными.

Есть ли различие между операторами = и ==?

Да, это совсем разные операторы! Первый осуществляет присвоение переменной, а второй — оператор сравнения, дающий логический результат. Ваша способность ответить на этот вопрос — хорошая проверка того, поняли ли вы материал данного раздела. Представьте, что объясняете это различие другу.

Будет ли выражение `a < b < c` сравнивать три числа в порядке `a`, `b` и `c`?

Да, Python поддерживает произвольное *сцепление* сравнений, такое как `a < b < c`, работающее согласно стандартным математическим соглашениям. Но во многих языках программирования (таких, как Java) выражение `a < b < c` недопустимо, поскольку результат первой части, `a < b`, имеет логический тип,

а сравнение логического значения с числом бессмысленно. В этой книге мы не используем сцепленное сравнение; мы предпочитаем такие выражения, как `(a < b) and (b < c)`.

Присвоит ли выражение `a = b = c = 17` значение `17` всем трем переменным?

Да. Даже при том, что операторы присвоения Python — не выражения, он поддерживает произвольное *сцепление* операторов присвоения. Мы не используем в книге сцепленные присвоения, поскольку большинство программистов Python считают это плохой практикой.

Можно ли использовать логические операторы `and`, `or` и `not` с операндами, тип которых не является логическим?

Да, но это обычно плохая идея. В этом контексте Python полагает, что значения `0`, `0.0` и пустая строка `''` означают `False`, а любое другое число или строка означают `True`.

Можно ли использовать арифметические операторы с логическими операндами?

Да, но делать этого также не стоит. Когда вы используете логические операнды с арифметическими операторами, они преобразуются в целые числа: `False` в `0` и `True` в `1`. Например, результатом выражения `(False - True - True) * True` будет значение `-2` типа `int`.

Я могу назвать переменную `max`?

Да, но потом вы не сможете использовать встроенную функцию `max()`. То же относится к функциям `min()`, `sum()`, `float()`, `eval()`, `open()`, `id()`, `type()`, `file()` и другим встроенным функциям.

Упражнения

1.2.1. Предположим, что `a` и `b` — целые числа. Что делает следующая последовательность операторов? Составьте трассировку объектного уровня для этого вычисления.

```
t = a
b = t
a = b
```

1.2.2. Составьте программу, использующую функции `math.sin()` и `math.cos()`, для проверки того, что результатом выражения $\cos^2\theta + \sin^2\theta$ является значение `1.0` для любого θ, переданного как аргумент командной строки. Просто выведите результат. Почему значения не всегда точно равны `1.0`?

1.2.3. Предположим, что переменные `a` и `b` имеют тип `bool`. Объясните, почему результатом выражения

```
(not (a and b) and (a or b)) or ((a and b) or not (a or b))
```

является `True`?

1.2.4. Предположим, что `a` и `b` — целые числа. Упростите следующее выражение:

```
(not (a < b) and not (a > b))
```

1.2.5. Что делает каждый из операторов?

a. `stdio.writeln(2 + 3)`
b. `stdio.writeln(2.2 + 3.3)`
c. `stdio.writeln('2' + '3')`
d. `stdio.writeln('2.2' + '3.3')`
e. `stdio.writeln(str(2) + str(3))`
f. `stdio.writeln(str(2.2) + str(3.3))`
g. `stdio.writeln(int('2') + int('3'))`
h. `stdio.writeln(int('2' + '3'))`
i. `stdio.writeln(float('2') + float('3'))`
j. `stdio.writeln(float('2' + '3'))`
k. `stdio.writeln(int(2.6 + 2.6))`
l. `stdio.writeln(int(2.6) + int(2.6))`

Объясните каждый результат.

1.2.6. Объясните, как использовать программу 1.2.4 (`quadratic.py`) для поиска квадратного корня числа.

1.2.7. Что делает выражение `stdio.writeln((1.0+2+3+4) / 4)`?

1.2.8. Предположим, что `a` имеет значение `3.14159`. Что делает каждый из следующих операторов?

a. `stdio.writeln(a)`
b. `stdio.writeln(a + 1.0)`

c. `stdio.writeln(8 // int(a))`
d. `stdio.writeln(8.0 / a)`
e. `stdio.writeln(int(8.0 / a))`
Объясните каждый результат.

1.2.9. Опишите результат использования записи `sqrt` вместо `math.sqrt` в программе 1.2.4.

1.2.10. Каков результат выражения `(math.sqrt(2) * math.sqrt(2) == 2)`? `True` или `False`?

1.2.11. Составьте программу, получающую два положительных целых числа в аргументах командной строки и выводящую `True`, если они делятся без остатка.

1.2.12. Составьте программу, получающую два положительных целых числа в аргументах командной строки и выводящую `False`, если любой из них больше или равен сумме двух других, и `True` в противном случае. (*Примечание*: этот код проверяет, могут ли эти три числа быть длинами сторон некоего треугольника.)

1.2.13. Каково значение `a` после выполнения каждой из следующих последовательностей:

```
a = 1          a = True       a = 2
a = a + a      a = not a      a = a * a
a = a + a      a = not a      a = a * a
a = a + a      a = not a      a = a * a
```

1.2.14. Студент-физик получил неожиданный результат при использовании кода

```
force = G * mass1 * mass2 / radius * radius
```

для вычисления значения по формуле $F = Gm_1m_2/r^2$. Объясните проблему и исправьте код.

1.2.15. Предположим, что `x` и `y` имеют тип `float` и представляют координаты (x, y) точки на Декартовой плоскости. Составьте выражение для вычисления расстояния от этой точки до исходной.

1.2.16. Составьте программу, получающую в аргументах командной строки два целых числа, `a` и `b`, и выводящую случайное целое число в диапазоне от `a` до `b`.

1.2.17. Составьте программу, выводящую сумму двух случайных целых чисел от `1` до `6` (как при бросании кубика).

1.2.18. Составьте программу, получающую в аргументах командной строки переменную `t` типа `float` и выводящую результат вычисления выражения $\sin(2t) + \sin(3t)$.

1.2.19. Составьте программу, получающую в аргументах командной строки три значения x_0, v_0 и `t` типа `float` и выводящую результат вычисления

выражения $x_0 + v_0 t - Gt^2 / 2$. (*Примечание*: G — это константа 9,80665. Это значение перемещения объекта в метрах после t-й секунды свободного падения от начальной позиции x_0 при скорости v_0 в метрах в секунду).

1.2.20. Составьте программу, получающую в аргументах командной строки два целых числа, `m` и `d`, и выводящую `True`, если день `d` месяца `m` приходится на 20 марта и 20 июня, и `False` в противном случае. (Значение `1` переменной `m` соответствует январю, `2` — февралю и т.д.)

Практические упражнения

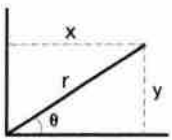
1.2.21. *Непрерывно начисляемый сложный процент.* Составьте программу, вычисляющую и выводящую сумму денег, выручаемую в результате инвестиции при заданной непрерывно начисляемой процентной ставке. Количество лет t, основная сумма вклада P и ежегодная процентная ставка r передаются как аргументы командной строки. Результирующее значение вычисляется по формуле Pe^{rt}.

1.2.22. *Температура воздуха с учетом влияния ветра.* T — температура в градусах Фаренгейта; v — скорость ветра в милях в час. Национальная служба погоды определяет эффективную температуру воздуха (с учетом влияния ветра) так:

$$w = 35.74 + 0.6215\ T + (0.4275\ T - 35.75)\ v^{0.16}$$

Составьте программу, получающую в аргументах командной строки два значения, `t` и `v` типа `float`, и выводящую температуру воздуха с учетом влияния ветра. *Примечание*: эта формула недопустима, если t больше 50 в абсолютном значении или если v больше 120 либо меньше 3 (подразумевается, что вводимые значения находятся в этом диапазоне).

1.2.23. *Полярные координаты.* Составьте программу, осуществляющую преобразование координат из Декартовых в полярные. Программа должна получать из командной строки значения координат `x` и `y` типа `float`, а выводить полярные координаты `r` и θ. Используйте функцию Python `math.atan2(y, x)`, вычисляющую значение арктангенса `y/x`, находящееся в диапазоне от $-\pi$ до π.



Полярные координаты

1.2.24. *Гауссовы случайные числа.* Один способ создания случайного числа в соответствии с распределением Гаусса подразумевает использование формулы *Бокса-Мюллера*

$$w = \sin(2\ \pi\ v)\ (-2\ \ln\ u)^{1/2},$$

где u и v — вещественные числа от 0 и 1, созданные функцией `math.random()`. Составьте программу, выводящую значение согласно стандартному Гауссову распределению.

1.2.25. *Проверка порядка.* Составьте программу, получающую в аргументах командной строки три значения, `x`, `y` и `z` типа `float`, а выводящую `True`, если значения расположены в порядке возрастания или убывания ($x < y < z$ или $x > y > z$), и `False` в противном случае.

1.2.26. *День недели.* Составьте программу, получающую дату и выводящую день недели, выпадающий на эту дату. Программа должна получать три аргумента командной строки: `m` (месяц), `d` (день) и `y` (год). Значение `1` переменной `m` соответствует январю, `2` — февралю и т.д. В выводе `0` соответствует воскресенью, `1` — понедельнику, `2` — вторнику и т.д. Для григорианского календаря используйте следующие формулы:

$$y_0 = y - (14 - m) / 12$$
$$x = y_0 + y_0 / 4 - y_0 / 100 + y_0 / 400$$
$$m_0 = m + 12 \times ((14 - m) / 12) - 2$$
$$d_0 = (d + x + (31 \times m_0) / 12) \% 7$$

Пример. На какой день недели выпадает 14 февраля 2000 года?

$$y_0 = 2000 - 1 = 1999$$
$$x = 1999 + 1999 / 4 - 1999 / 100 + 1999 / 400 = 2483$$
$$m_0 = 2 + 12 \times 1 - 2 = 12$$
$$d_0 = (14 + 2483 + (31 \times 12) / 12) \% 7 = 2500 \% 7 = 1 \text{ (понедельник)}$$

1.2.27. *Равномерные случайные числа.* Составьте программу, выводящую пять равномерных случайных чисел типа `float` от 0.0 до 1.0, их среднее, минимальное и максимальное значения. Используйте встроенные функции `min()` и `max()`.

1.2.28. *Проекция Меркатора.* Проекция Меркатора — это конформная (сохраняющая углы) проекция, сопоставляющая широту φ и долготу λ с прямоугольными координатами x и y. Это широко используется в навигационных схемах и картах, закачиваемых из веб. Проекция определяется уравнениями $x = \lambda - \lambda_0$ и $y = 1/2 \ln((1 + \sin\varphi) / (1 - \sin\varphi))$, где λ_0 — долгота точки в центре карты. Составьте программу, получающую из командной строки значения $\lambda 0$, широты и долготы точки и выводящая ее проекцию.

1.2.29. *Преобразование цветов.* Для представления цвета используется несколько форматов. Например, для экранов LCD, цифровых камер и веб-страниц используется *формат RGB*, определяющий уровни красного (R), зеленого (G), и синего (B) цветов целочисленным значением в диапазоне от 0 до 255. Для печати книг и журналов используется *формат CMYK*, определяющий

уровни синего (C), красного (M), желтого (Y) и черного (K) цветов вещественным значением в диапазоне от 0.0 до 1.0. Составьте программу, преобразующую значения RGB в CMYK. Передавайте в командной строке три целых числа, `r`, `g` и `b`, а выводите эквивалентные значения CMYK. Если все значения RGB равны 0, то и все значения CMY равны 0, а значение K — 1; в противном случае используйте следующие формулы:

$$w = \max(r/255, g/255, b/255)$$
$$c = (w - (r/255))/w$$
$$m = (w - (g/255))/w$$
$$y = (w - (b/255))/w$$
$$k = 1 - w$$

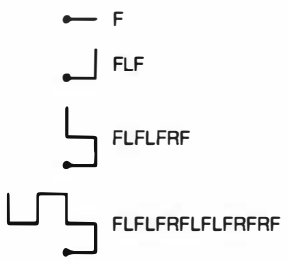
1.2.30. *Большая окружность.* Составьте программу, получающую в командной строке четыре аргумента, `x1`, `y1`, `x2` и `y2` типа `float` (широта и долгота в градусах двух точек на глобусе), и выводящую длину большой окружности между ними. Длина большой окружности d (в навигационных милях) определяется следующим уравнением:

$$d = 60 \arccos(\sin(x_1)\sin(x_2) + \cos(x_1)\cos(x_2)\cos(y_1 - y_2)).$$

Обратите внимание, что это уравнение использует градусы, а тригонометрические функции Python — радианы. Для их преобразования используйте функции `math.radians()` и `math.degrees()`. Используйте свою программу для вычисления длины большой окружности между Парижем (48.87° N и −2.33° W) и Сан-Франциско (37.8° N и 122.4° W).

1.2.31. *Сортировка.* Составьте программу, получающую в аргументах командной строки три целых числа и выводящую их в порядке возрастания. Используйте встроенные функции `min()` и `max()`.

1.2.32. *Кривые дракона.* Составьте программу, выводящую инструкции по рисованию кривых дракона в порядке от 0 до 5. Инструкции — это строки `F`, `L` и `R`, где символ `F` означает "нарисовать линию на одну единицу вперед"; `L` — "повернуть налево"; `R` — "повернуть направо". Для создания кривой дракона n-го порядка сложите бумажную полоску пополам n раз, а затем разверните ее так, чтобы все углы были прямыми. Для решения этой задачи обратите внимание на то, что кривая порядка n — это кривая порядка $n - 1$, затем `L`, затем кривая порядка $n - 1$, развернутая в обратном порядке, а после представьте подобное описание для обратной кривой.



Кривые дракона 0-, 1-, 2- и 3-го порядков

1.3. Условные выражения и циклы

В программах, рассмотренных на настоящий момент, каждый из операторов выполняется только однажды, причем по порядку. Большинство программ куда сложнее, поскольку последовательность операторов и количество их выполнений могут изменяться. Для описания изменения порядка операторов в программе мы используем термин *контроль потока* (control flow). В этом разделе мы ознакомимся с операторами, позволяющими изменять последовательность выполнения на основании логики и значения переменных программы. Это основные компоненты программирования.

Программы этого раздела...

Программа 1.3.1. Орел или решка (`flip.py`) 73
Программа 1.3.2. Ваш первый цикл (`tenhellos.py`) 76
Программа 1.3.3. Вычислительные степеней числа 2 (`powersoftwo.py`) 77
Программа 1.3.4. Ваш первый вложенный цикл (`divisorpattern.py`) 83
Программа 1.3.5. Гармонические числа (`harmonic.py`) 86
Программа 1.3.6. Метод Ньютона (`sqrt.py`) 87
Программа 1.3.7. Преобразование в двоичный формат (`binary.py`) 89
Программа 1.3.8. Модель разорения игрока (`gambler.py`) 91
Программа 1.3.9. Разложение на множители целых чисел (`factors.py`) 93

Конкретно мы рассмотрим операторы языка Python, реализующие *условные выражения* (conditional), где некие другие операторы могут быть выполнены или не выполнены в зависимости от определенных условий, и *циклы* (loop), где некие другие операторы могут быть выполнены многократно, опять же в зависимости от определенных условий. Как будет продемонстрировано в многочисленных примерах этого раздела, условные выражения и циклы позволяют использовать всю мощь компьютера для решения программным способом широкого разнообразия задач, которые даже не стоило пытаться решать без компьютера.

Оператор `if`. В большинстве вычислений для разных исходных данных требуются разные действия. Одним из способов выражения этих различий в языке Python является оператор `if`:

```
if <логическое выражение>:
    <оператор>
    <оператор>
    ...
```

Эта форма записи известна как *шаблон* (template) и используется для описания формата конструкций языка Python. В угловых скобках (<>) приводят уже определенную конструкцию, означая, что в этом месте можно использовать

любой ее экземпляр. В данном случае <*логическое выражение*> представляет выражение, возвращающее результат логического типа, такое, например, как оператор сравнения, а <*оператор*> представляет выполняемый оператор (это могут быть самые разные операторы). Вполне можно было бы сделать и формальные определения для элементов <*логическое выражение*> и <*оператор*>, но мы воздерживаемся от перехода на такой уровень детализации. Смысл оператора `if` вполне очевиден: Python выполняет операторы с отступом, если и только если логическое выражение истинно. Набор операторов с отступом называется *блоком* (block). Первая строка без отступа отмечает конец блока. Большинство программистов Python используют для отступа четыре пробела.

Предположим, например, что необходимо вычислить абсолютное значение целого числа `x`. Эту задачу решает следующий код:

```
if x < 0:
    x = -x
```

(А именно: если объект, на который ссылается переменная `x`, содержит отрицательное значение, переменная `x` привязывается к новому объекту с абсолютным значением прежнего.)

В качестве второго примера рассмотрим следующий код:

```
if x > y:
    temp = x
    x = y
    y = temp
```

Анатомия оператора `if`

Этот код располагает значения переменных `x` и `y` в порядке возрастания, обменивая их ссылками на объекты в случае необходимости.

Большинство других современных языков программирования использует некий механизм разграничения блоков операторов (такой, как заключение операторов в фигурные скобки). В языке Python *важен размер отступа каждой строки*, поэтому необходимо обращать на них внимание. Сравните, например, два следующих фрагмента кода, немного отличающихся только отступом:

```
if x >= 0:
    stdout.write('not ')
stdout.writeln('negative')
```

```
if x >= 0:
    stdout.write('not ')
    stdout.writeln('negative')
```

Код оператора `if` слева имеет блок с одним оператором, сопровождаемый другим оператором; код оператора `if` справа имеет блок с двумя операторами. Если значение `x` больше или равно `0`, то выводятся оба фрагмента, `not` и `negative`. Напротив, если значение `x` меньше 0, код слева выводит только `negative`, а код справа не выводит вообще ничего.

Директива `else`. К оператору `if` можно добавить директиву `else`, позволяющую выполнить один оператор (или блок операторов) в одном случае, а другой оператор (или блок операторов) — в другом, в зависимости от истинности или ложности результата логического выражения, как в следующем шаблоне:

```
if <логическое выражение>:
    <блок операторов>
else:
    <блок операторов>
```

В качестве примера необходимости в директиве `else` рассмотрим следующий код, присваивающий максимальное из значений двух переменных типа `int` переменной `maximum`. (Альтернативно тот же результат можно получить при вызове встроенной функции `max()`.)

```
if x > y:
    maximum = x
else:
    maximum = y
```

Когда блок `if` или `else` содержит только один оператор, для краткости его можно поместить в той же строке, что и ключевое слово `if` или `else`, как в третьей и четвертой строках таблицы, приведенной ниже.

Типичные примеры использования операторов `if`

Абсолютное значение	`if x < 0:` `  x = -x`

Сортировка *x* и *y*	`if x > y:`
	`  temp = x`
	`  x = y`
	`  y = temp`

Максимальное среди *x* и *y*	`if x > y: maximum = x` `else:  maximum = y`
Проверка допустимости операции остатка	`if den == 0: stdio.writeln('Division by zero')` `else:  stdio.writeln('Remainder = ' + num % den)`

Проверка допустимости значений квадратичной формулы	`discriminant = b*b - 4.0*a*c`
	`if discriminant < 0.0:`
	`  stdio.writeln('No real roots')`
	`else:`
	`  d = math.sqrt(discriminant)`
	`  stdio.writeln((-b + d)/2.0)`
	`  stdio.writeln((-b - d)/2.0)`

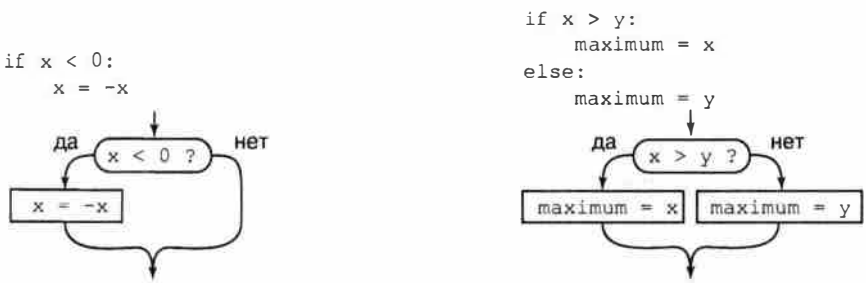
Программа 1.3.1. Орел или решка (`flip.py`)

```
import random
import stdio

if random.randrange(0, 2) == 0:
    stdio.writeln('Heads')
else:
    stdio.writeln('Tails')
```

Эта программа моделирует бросок монеты, падающей орлом (`Heads`) или решкой (`Tails`), в зависимости от результата вызова функции `random.randrange()`. Результаты будут напоминать таковые при бросках реальной монеты, но в действительности эта последовательность результатов не абсолютно случайна.

```
% python flip.py
Heads
% python flip.py
Tails
% python flip.py
Tails
```

Вот два других примера использования конструкций `if` и `if-else`. Эти примеры типичны для простых вычислений, которые вы, вероятно, примените в своих программах. Условные операторы — основа программирования. Поскольку *семантика* (смысл) операторов подобна их значению в обычном языке, вы быстро к ним привыкнете.



Примеры блок-схем с использованием операторов `if`

Программа 1.3.1 (`flip.py`) приводит другой пример использования конструкции `if-else`, в данном случае для моделирования броска монеты. Тело программы — один оператор, подобный другим, рассмотренным на настоящий момент, но этот достоин особого внимания, поскольку он затрагивает интересную

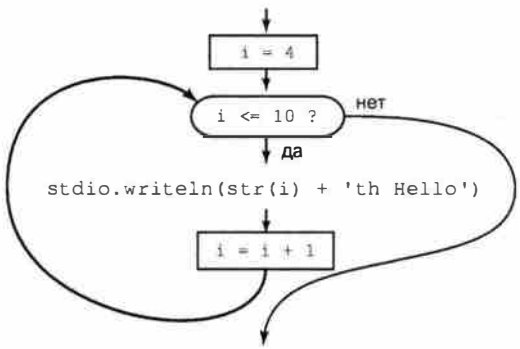
философскую проблему: может ли компьютерная программа создавать на самом деле *случайные* значения? Конечно нет, но программа *может* создавать значения, обладающие большинством свойств на самом деле случайных значений.

Чтобы лучше понять управление потоком, можно визуализировать его на *блок-схеме*. Пути следования на блок-схеме соответствуют путям управления потоком в программе. Довольно давно, когда программисты использовали низкоуровневые языки и трудные для понимания пути выполнения, блок-схемы были основой программирования. В новых языках блок-схемы используются только для демонстрации фундаментальных стандартных блоков, таких как оператор `if`.

Оператор `while`. Вычисления нередко приходится повторять. Предназначенная для этого базовая конструкция Python имеет следующий формат:

```
while <логическое выражение>:
    <оператор 1>
    <оператор 2>
    ...
```

Оператор `while` имеет ту же форму, что и оператор `if` (единственное различие — ключевое слово `while` вместо `if`), но назначение у него совсем иное. Он заставляет компьютер вести себя так: если результат логического выражения `False`, не делать ничего; если результат `True` — выполнить блок операторов ниже (так и в конструкции `if`), а затем снова проверить выражение и выполнить последовательность операторов, если выражение возвращает `True`, и продолжать так до тех пор, пока выражение остается истинным. Таким образом, поток "идет по кругу", пока логическое выражение истинно. Этот цикл представлен на блок-схеме. (Цикл `while` на этой схеме взят из программы 1.3.2, которую мы вскоре рассмотрим.) Оператор `while` реализует *цикл* (loop). Блок операторов с отступом в пределах цикла `while` — это *тело цикла* (body of the loop), а логическое выражение — это *условие продолжения цикла* (loop-continuation condition). Условие продолжения цикла — это, как правило, проверка значения некой переменной (переменных), поэтому каждому циклу `while` обычно предшествует код *инициализации* (initialization), устанавливающий исходное значение (значения).



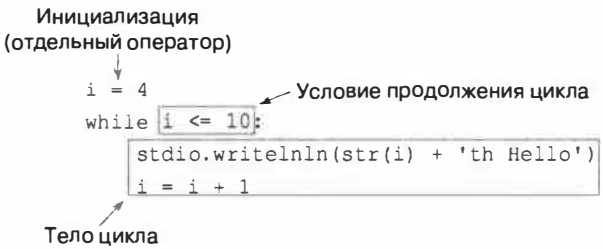
Пример блок-схемы (оператор `while`*)*

```
i = 4
while i <= 10:
    stdio.writelnln(str(i) + 'th Hello')
    i = i + 1
```

Оператор `while` эквивалентен последовательности одинаковых операторов `if`:

```
if <логическое выражение>:
    <оператор 1>
    <оператор 2>
    ...
if <логическое выражение>:
    <оператор 1>
    <оператор 2>
    ...
if <логическое выражение>:
    <оператор 1>
    <оператор 2>
    ...
...
```

Чтобы условие продолжения цикла стало в конечном счете ложным (`False`) и цикл закончился, в теле цикла должно изменяться значение одной или нескольких переменных.



Анатомия цикла `while`

Общепринятая практика программирования подразумевает использование целочисленной переменной, значение которой отслеживает количество выполняемых итераций цикла. Вначале переменной присваивается некое исходное значение, а затем на каждом цикле оно увеличивается на 1 и проверяется, не превышает ли оно заранее заданный максимум прежде, чем решить продолжать цикл. Простым примером этой парадигмы является программа 1.3.2 (`tenhellos.py`), использующая оператор `while`. Основой вычисления является оператор `i = i + 1`.

Программа 1.3.2. Ваш первый цикл (tenhellos.py)

```
import stdio

stdio.writeln('1st Hello')
stdio.writeln('2nd Hello')
stdio.writeln('3rd Hello')

i = 4
while i <= 10:
    stdio.writeln(str(i) + 'th Hello')
    i = i + 1
```

Эта программа выводит 10 сообщений `Hello`. Для этого используется цикл `while`. После третьей выведенной строки остальные отличаются только индексом, поэтому мы определяем переменную `i` для его хранения. После инициализации переменной `i` значением `4` начинается цикл `while`, где переменная `i` используется в вызове функции `stdio.writeln()` и в операторе ее приращения на каждой итерации цикла. После вывода сообщения `10th Hello` переменная `i` получит значение `11` и цикл завершится.

```
% python tenhellos.py
1st Hello
2nd Hello
3rd Hello
4th Hello
5th Hello
6th Hello
7th Hello
8th Hello
9th Hello
10th Hello
```

`i` **Счетчик управления циклом**

Трассировка цикла `while`

`i`	`i <= 10`	**Вывод**
4	true	4th Hello
5	true	5th Hello
6	true	6th Hello
7	true	7th Hello
8	true	8th Hello
9	true	9th Hello
10	true	10th Hello
11	false	

Как математическое уравнение данный оператор не имеет смысла, поскольку это оператор присвоения Python, вычисляющий значение `i + 1`, а затем присваивающий результат переменной `i`. Если `i` имеет значение `4`, оператор выполнялся, если `5` — тоже; если `6`, оператор выполнится снова и т.д. Изначально в программе `tenhellos.py` переменная `i` содержит значение `4`, поэтому блок операторов выполняется семь раз, пока `i` наконец не превысит значение `10` и цикл не закончится.

Для столь простой задачи использовать оператор `while` едва ли стоит, но скоро придется решать такие задачи, где операторы будут повторяться слишком много раз,

чтобы обходиться без циклов. Есть серьезное различие между программами с операторами while и без них, поскольку потенциально операторы while позволяют задать неограниченное количество раз выполнения операторов в программе. В частности, оператор while позволяет осуществлять длительные вычисления в коротких программах. Эта возможность открывает дверь к созданию программ для решения таких задач, которые без компьютера решать было бы затруднительно. Но за все надо платить: по мере усложнения программ их все труднее понимать.

Программа 1.3.3 (powersoftwo.py) использует оператор while для вывода таблицы степеней числа 2. Кроме управляющего циклом счетчика i, здесь есть переменная power, содержащая степени числа 2 при их вычислении. Тело цикла содержит три оператора: первый выводит текущую степень 2, второй вычисляет следующую степень числа 2 (умножив текущую на 2), а третий увеличивает значение счетчика управления циклом.

Программа 1.3.3. Вычисление степеней числа 2 (powersoftwo.py)

```
import sys
import stdio

n = int(sys.argv[1])
power = 1
i = 0
while i <= n:
    # Вывести i-ю степень числа 2.
    stdio.writeln(str(i) + ' ' + str(power))
    power = 2 * power
    i = i + 1
```

n	Значение выхода из цикла
i	Счетчик управления циклом
power	Текущая степень числа 2

Эта программа получает как аргумент командной строки целое число n и выводит таблицу, содержащую первые n степеней числа 2. На каждом цикле значение переменной i увеличивается на 1, а значение переменной power удваивается. Мы представили только три первых и три последних строки таблицы; всего программа выводит $n + 1$ строк.

```
% python powersoftwo.py 5
0 1
1 2
2 4
3 8
4 16
5 32
```

```
% python powersoftwo.py 29
0 1
1 2
2 4
...
27 134217728
28 268435456
29 536870912
```

Кстати, в информатике много ситуаций, когда полезно знать степени числа 2. По крайней мере, первые 10 значений этой таблицы следует знать на память, и обратите внимание, что 2^{10} — это примерно 1 тысяча, 2^{20} — примерно 1 миллион, а 2^{30} — примерно 1 миллиард.

Программа 1.3.3 — это прототип многих полезных вычислений. Смена вычисляемого выражения, изменение накапливаемого значения и способа приращения управляющей переменной цикла позволяют выводить таблицы для множества функций (см. упр. 1.3.10).

Поведение использующих циклы программ имеет смысл тщательно исследовать, возможно, даже с использованием трассировки. Например, трассировка программы `powersoftwo.py` позволит проследить значение каждой переменной и состояние контролирующего цикл условного выражения перед каждой итерацией. Трассировка цикла может быть очень утомительной, но ее почти всегда следует выполнять, поскольку это ясно показывает, что делает программа.

Программа 1.3.3 — практически программа самотрассировки, поскольку она выводит значения своих переменных на каждом цикле. Для трассировки любой программы достаточно добавить соответствующие операторы `stdio.writeln()`. Современные среды программирования предоставляют для трассировки сложные инструментальные средства, но этот проверенный метод прост и эффективен. Вам, конечно, стоит добавить операторы `stdio.writeln()` в несколько первых своих циклов, чтобы убедиться в том, что они делают именно то, что вы ожидаете.

Для более сложного примера предположим, что необходимо вычислить наибольшую степень числа 2, меньшую или равную заданному положительному целому числу `n`. Если `n` равно `13`, то необходим результат `8`; если `n` — `1000`, необходим результат `512`; если `n` — `64`, результат `64`; и т.д. Реализовать это вычисление при помощи цикла `while` довольно просто:

```
power = 1
while 2 * power <= n:
    power = 2 * power
```

Довольно сложно поверить, что этот простой фрагмент кода дает желаемый результат. Убедиться в этом можно, сделав следующие наблюдения:

- `power` — это всегда степень числа 2;
- `power` — никогда не больше `n`;
- `power` увеличивается при каждой итерации, поэтому цикл должен закончиться;
- по завершении цикла `2 * power` больше `n`.

Рассуждение такого типа зачастую очень важно для понимания работы цикла `while`. Даже при том, что большинство создаваемых вами циклов будет намного проще этого, вы должны убедиться, что каждый создаваемый цикл будет вести себя так, как ожидалось.

Логика остается той же, осуществляет ли цикл итерации лишь несколько раз, как в программе `tenhellos.py`; множество раз, как в программе `powersoftwo.py`; или миллионы раз, как в нескольких примерах, приведенных далее. Этот переход от нескольких скромных случаев к огромным вычислениям довольно велик. Для понимания циклов, изменения значений переменных на каждой итерации (и их проверки) добавление операторов отслеживания значений переменных во время выполнения программы при небольшом количестве итераций вполне обосновано. По завершении отработки цикла эти подпорки вполне можно удалить и действительно использовать всю мощь компьютера.

Трассировка программы `powersoftwo.py` при `n`, равном 29

I	power	i <= n
0	1	true
1	2	true
2	4	true
3	8	true
4	16	true
5	32	true
6	64	true
7	128	true
8	256	true
9	512	true
10	1024	true
11	2048	true
12	4096	true
13	8192	true
14	16384	true
15	32768	true
16	65536	true
17	131072	true
18	262144	true
19	524288	true
20	1048576	true
21	2097152	true
22	4194304	true
23	8388608	true
24	16777216	true
25	33554432	true
26	67108864	true
27	134217728	true
28	268435456	true
29	536870912	true
30	1073741824	false

Сокращенная запись присвоения. Изменение значения переменной столь популярно в программировании, что такие современные языки, как Python, предоставляют для этого сокращение записи. Наиболее распространена практика сокращения формы `i = i + 1` оператора присвоения на `i += 1`. Та же форма записи предоставляется и для других парных операторов, включая `-`, `*` и `/`. Например, в программе 1.3.3 большинство программистов использовали бы запись `power *= 2` вместо `power = 2 * power`. Такие сокращения получили широкое распространение благодаря языку C, а с 1970-х годов стали стандартом. Они выдержали испытание временем, поскольку они компактней, изящней и легче для понимания. С этого момента мы будем использовать такие сокращения в наших программах.

Оператор `for`. Оператор `while` вполне позволяет создавать в программах практически любые циклы, но прежде чем переходить к дальнейшим примерам, рассмотрим альтернативную конструкцию — оператор `for`, обеспечивающий больше гибкости при создании программ с циклами. Эта альтернативная конструкция существенно не отличается от базового оператора `while`, но зачастую она позволяет получать более компактные и более читаемые программы, чем при использовании только операторов `while`.

Как уже упоминалось, весьма распространены циклы, использующие для отслеживания количества выполняемых итераций целое число. Сначала переменной присваивается некое целое число, затем, на каждой итерации цикла, к ее значению добавляется другое целое число, увеличивая текущее значение переменной. Перед началом следующей итерации содержимое переменной сравнивается с неким заранее заданным целочисленным значением и принимается решение о продолжении цикла. Это цикл со *счетчиком* (counter).

В языке Python цикл со счетчиком можно реализовать, используя оператор `while` и следующий шаблон:

```
<переменная> = <старт>
while <переменная> < <стоп>:
    <блок операторов>
    <переменная> += 1
```

Оператор `for` обеспечивает более компактный способ реализации цикла со счетчиком. В языке Python оператор `for` имеет несколько форматов. Пока рассмотрим следующий шаблон:

```
for <переменная> in range(<старт>, <стоп>):
    <блок операторов>
```

Аргументы <*старт*> и <*стоп*> встроенной функции `range()` должны быть целыми числами. Когда оператор дан в этой форме, Python многократно выполняет блок операторов с отступом. На первой итерации цикла <*переменная*> имеет значение <*старт*>, на второй — <*старт*> `+ 1` и так далее. На последней итерации <*переменная*> имеет значение <*стоп*> `- 1`. Короче говоря, оператор `for` многократно выполняет свой оператор с отступом, причем значение <*переменная*> меняется от <*старт*> до <*стоп*> `- 1` включительно. Рассмотрим фрагмент программы 1.3.2 (`tenhellos.py`):

```
i = 4
while i <= 10:
    stdio.writeln(str(i) + 'th Hello')
    i = i + 1
```

С использованием оператора `for` его можно было бы выразить короче:

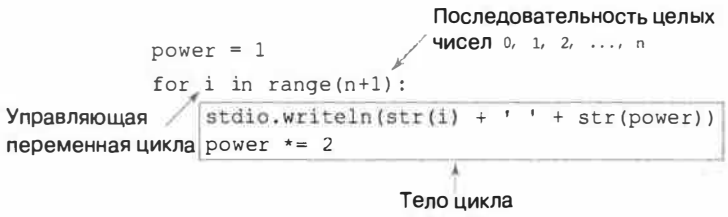
```
for i in range(4, 11):
    stdio.writeln(str(i) + 'th Hello')
```

Если у функции `range()` есть только один аргумент, то это значение <*стоп*>, стандартным значением аргумента <*старт*> является `0`. Рассмотрим следующий цикл `for`:

```
power = 1
for i in range(n+1):
    stdio.writeln(str(i) + ' ' + str(power))
    power *= 2
```

Это модификация цикла `while` из программы 1.3.3 (`powersoftwo.py`).

Выбор формы того же вычисления — это вопрос вкуса каждого программиста, как он выбирает возможный из синонимов, или между использованием активного или пассивного залога при составлении предложения. Правила составления программ не жестче, чем при составлении предложений. Ваша задача — выработать стиль, позволяющий осуществлять вычисления и вполне понятный другим. В этой книге мы используем операторы `for` для циклов со счетчиком и операторы `while` для всех других видов циклов.



Анатомия цикла `for` (со счетчиком)

Типичные примеры использования операторов `for` и `while`

Вывести первые n+1 степени числа 2

```
power = 1
for i in range(n+1):
    stdio.writeln(str(i) + ' ' + str(power))
    power *= 2
```

Вывести наибольшую степень числа 2, меньшую или равную n

```
power = 1
while 2*power <= n:
    power *= 2
stdio.writeln(power)
```

Вывести сумму (1 + 2 + ... + n)

```
total = 0
for i in range(1, n+1):
    total += i
stdio.writeln(total)
```

Вывести произведение (n! = 1 × 2 × ... × n)

```
product = 1
for i in range(1, n+1):
    product *= i
stdio.writeln(product)
```

Вывести таблицу значений функции n+1

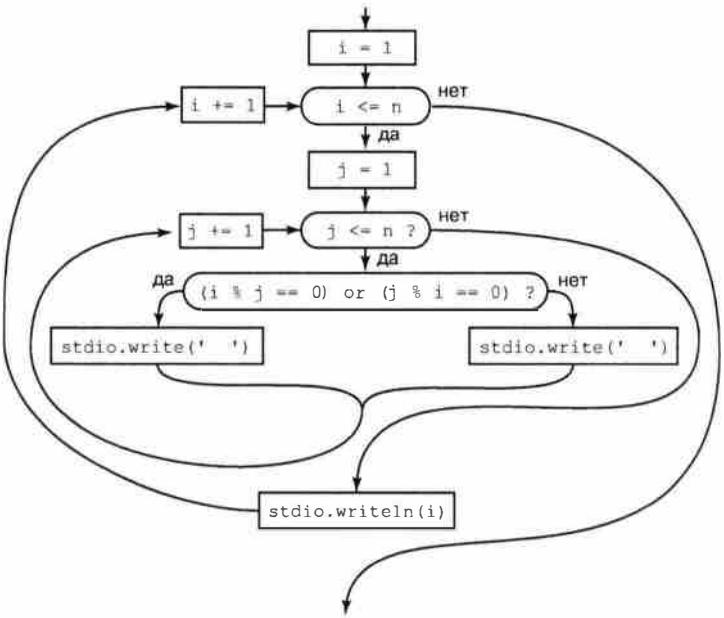
```
for i in range(n+1):
    stdio.write(str(i) + ' ')
    stdio.writeln(2.0 * math.pi * i / n)
```

Вывести линейчатую функцию (см. программу 1.2.1)

```
ruler = '1'
stdio.writeln(ruler)
for i in range(2, n+1):
    ruler = ruler + ' ' + str(i) + ' ' + ruler
    stdio.writeln(ruler)
```

Приведенная выше таблица включает несколько фрагментов кода с типичными примерами циклов, используемых в программах Python. Некоторые из фрагментов вы уже видели; другие — новый код для простых вычислений. Чтобы лучше понимать циклы в Python, поместите каждый из этих фрагментов кода в программу, получающую в аргументе командной строки целое число `n` (как в программе `powersoftwo.py`), *и запустите ее.* Затем составьте собственные циклы для подобных вычислений по своему усмотрению или выполните несколько первых упражнений в конце этого раздела. Нет ничего лучше опыта, полученного при запуске кода, созданного самостоятельно, и это именно то, что нужно, чтобы научиться использовать циклы.

Как можно заметить на представленной ниже блок-схеме, поток выполнения программы 1.3.4 куда сложнее. Чтобы получить подобную блок-схему, лучше работать с циклами на базе оператора `while` (см. упр. 1.3.15), поскольку цикл версии `for` скрывает детали. Эта схема иллюстрирует важность использования в программировании ограниченного количества простых управляющих структур. Используя вложение, можно составлять циклы и условные выражения так, чтобы получать простые и понятные программы, несмотря на сложность их потока выполнения. Для очень многих полезных вычислений достаточно лишь одного или двух уровней вложенности. Например, у многих программ в этой книге та же общая структура, что и у программы `divisorpattern.py`.



Блок-схема программы `divisorpattern.py`

Программа 1.3.4. Ваш первый вложенный цикл (divisorpattern.py)

```
import sys
import stdio

n = int(sys.argv[1])

for i in range(1, n+1):
    # Вывести i-тую строку.
    for j in range(1, n+1):
        # Вывести j-тый элемент i-той строки.
        if (i % j == 0) or (j % i == 0):
            stdio.write('* ')
        else:
            stdio.write('  ')
stdio.writeln(i)
```

n	Количество строк и столбцов
i	Индекс строки
j	Индекс столбца

Эта программа получает из командной строки целочисленный аргумент n и выводит таблицу из n строк с последовательностью звездочек и номером столбца. Если i делится на j или j делится на i, выводится звездочка, в противном случае — пробел. В программе используются вложенные циклы for. Вычисление контролируют управляющие переменные цикла i и j.

```
% python divisorpattern.py 3
* * * 1
* *   2
*   * 3

% python divisorpattern.py 16
* * * * * * * * * * * * * * * * 1
* *   *   *   *   *   *   *   * 2
*   *     *     *     *     *   3
* *   *       *       *       * 4
*       *           *           5
* * *     *           *         6
*               *           *   7
* *   *       *               * 8
*   *             *             9
* *     *           *           10
*                   *           11
* * * *   *             *       12
*                       *       13
* *           *               *   14
*   *   *                       * 15
* *   *       *               *   16
```

Трассировка при n, равном 3

i	j	i % j	j % i	Вывод
1	1	0	0	*
1	2	1	0	*
1	3	1	0	*
				1
2	1	0	1	*
2	2	0	0	*
2	3	2	1	
				2
3	1	0	1	*
3	2	1	2	
3	3	0	0	*
				3

Для обозначения вложения в коде программы используется отступ. Отступ в языке Python существен (во многих других языках программирования для вложений требуются фигурные скобки или другая форма записи). В программе 1.3.4 цикл `i` является *внешним* (inner), а цикл `j` — *внутренним* (outer). Внутренний цикл осуществляет полный цикл итераций для каждой итерации внешнего цикла. Как обычно, для изучения новой конструкцию программирования лучше всего применять трассировку.

Для второго примера вложенности рассмотрим программу, вычисляющую подоходный налог. Люди без дохода (или с небольшим доходом) не платят подоходный налог; люди с доходом от 0 и больше, но меньше 8 925, платят 10 процентов; люди с доходом от 8 925 и больше, но меньше, чем 36 250, платят 15 процентов; и т.д. Для этого можно было бы использовать вложенные один в другой операторы `if` с директивами `else`:

```
if income < 0.0:
    rate = 0.00
else:
    if income < 8925:
        rate = 0.10
    else:
        if income < 36250:
            rate = 0.15
        else:
            if income < 87850:
                rate = 0.25
            ...
```

В этом приложении уровень вложения настолько глубок, что код становится трудно понять. Выбор одной из нескольких альтернатив — это обычное дело, здесь имеет смысл избежать глубокого вложения. Для этого пригодится обобщенная конструкция Python `if`, допускающая любое количество директив "else if" перед директивой "else" в формате

```
elif <логическое выражение>:
    <блок операторов>
```

Когда блок `elif` содержит только один оператор, для краткости и ясности его можно поместить в той же строке, что и ключевое слово `elif`. В этой конструкции код вычисления предельной налоговой ставки довольно прост:

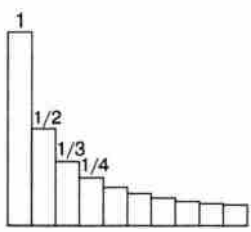
```
if   income <      0: rate = 0.00
elif income <   8925: rate = 0.10
elif income <  36250: rate = 0.15
elif income <  87850: rate = 0.23
elif income < 183250: rate = 0.28
elif income < 398350: rate = 0.33
elif income < 400000: rate = 0.35
else:                 rate = 0.396
```

Python вычисляет последовательность логических выражений, пока не найдет возвращающее `True`, и выполняет соответствующий ему блок операторов. Обратите внимание, что заключительная директива `else` предназначена для случая, когда ни одно из условий не удовлетворяется (в данном случае налоговая ставка для самых богатых). Эта конструкция используется очень часто.

Приложения. Возможность использования в программах условных выражений и циклов открывает весь мир вычислений. Чтобы подчеркнуть этот факт, мы рассмотрим множество примеров. Все эти примеры задействуют работу с типами данных, рассмотренными в разделе 1.2, а остальные подразумевают, что те же механизмы хорошо сработают в любом вычислительном приложении. Программы тщательно проработаны как для простоты изучения и понимания, так и для подготовки к созданию собственных программ, содержащих циклы.

В рассматриваемых здесь примерах задействованы вычисления с числами. Некоторые из наших примеров ориентированы на проблемы, решавшиеся математиками и учеными на протяжении нескольких прошлых столетий. Хотя компьютеры существуют всего пятьдесят лет, в основе большинства используемых нами вычислительных методов лежит богатейшая математическая традиция, уходящая корнями в античность.

Конечная сумма. Используемая в программе `powersoftwo.py` вычислительная парадигма используется весьма часто. Она использует две переменные: одну как контролирующий цикл индекс, другую для суммирования результатов вычисления. Программа 1.3.5 (`harmonic.py`) использует ту же парадигму для вычисления конечной суммы $H_n = 1 + 1/2 + 1/3 + \ldots + 1/n$. Это *гармонические числа* (harmonic numbers), нередко возникающие в дискретной математике. Гармонические числа — это дискретный аналог логарифма. Они также приблизительно описывают площадь ниже кривой $y = 1 / x$. Программу 1.3.5 можно также использовать как модель для вычисления значения других сумм (см. упр. 1.3.16).



Вычисление квадратного корня. Как реализованы функции в модуле Python `math`, такие как `math.sqrt()`? Одну из методик иллюстрирует программа 1.3.6 (`sqrt.py`). Для реализации функции квадратного корня она использует итерационное вычисление, известное вавилонянам еще 4 000 лет назад. Это также частный случай общей вычислительной методики, разработанной в XVII веке Исааком Ньютоном и Джозефом Рафсоном и широко известной как *метод Ньютона* (Newton's method). При общих условиях для данной функции $f(x)$ метод Ньютона — эффективный способ нахождения корней (значений x, при котором функция равна 0). Начнем с начальной оценки t_0. При данной оценке t_i вычислим новую оценку, проведя касательную линию к кривой $y = f(x)$ в точке $(t_i, f(t_i))$, и установим для t_{i+1} координату x точки, где эта линия пересекает ось X. Осуществляя итерации этого процесса, мы все ближе подходим к корню.

Программа 1.3.5. Гармонические числа (harmonic.py)

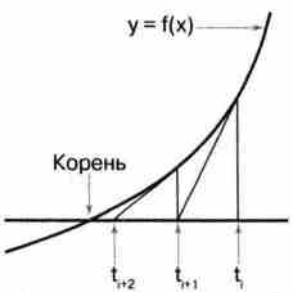
```
import sys
import stdio

n = int(sys.argv[1])

total = 0.0
for i in range(1, n+1):
    # Добавить i-тое слагаемое в сумму.
    total += 1.0 / i

stdio.writeln(total)
```

N	Количество слагаемых в сумме
I	Управляющая переменная цикла
Total	Накопленная сумма

Эта программа получает как аргумент командной строки целое число n и выводит n-ное гармоническое число. В математическом анализе это значение известно как $\ln() + 0.57721$ для больших n. Обратите внимание: $\ln(10\,000) \approx$ ≈ 9.21034.

```
% python harmonic.py 2
1.5
% python harmonic.py 10
2.9289682539682538
% python harmonic.py 10000
9.787606036044348
```

Вычисление квадратного корня положительного числа эквивалентно поиску положительного корня функции $f(x) = x^2 - c$. Для этого частного случая метод Ньютона дает процесс, реализованный в программе 1.3.6 (sqrt.py) и упражнении 1.3.17. Начинаем с оценки $t = c$. Если t равно c/t, то t равно квадратному корню c, а следовательно, вычисление закончено. В противном случае уточните оценку, заменив t средним от t и c/t. С использованием метода Ньютона мы получаем значение квадратного корня 2 с точностью 15 знаков только при 5 итерациях цикла.



Метод Ньютона

Метод Ньютона очень важен в научных вычислениях, поскольку этот итерационный подход эффективен для поиска корней широкого класса функций, включая те, для которых не известны аналитические решения (так что модуль Python math для них не в помощь). В настоящее время мы считаем само собой разумеющимся, что вполне можем найти любые значения необходимых математических функций; до появления компьютеров ученые и инженеры вынуждены были использовать таблицы или вычислять значения вручную.

Вычислительные методы, разработанные для вычислений вручную, должны были быть очень эффективными, поэтому нет ничего удивительного в том, что большинство из них эффективно и для компьютеров. Метод Ньютона — классический пример этого явления.

Программа 1.3.6. Метод Ньютона (sqrt.py)

```
import sys
import stdio

EPSILON = 1e-15

c = float(sys.argv[1])
t = c
while abs(t - c/t) > (EPSILON * t):
    # Заменить t средним значением от t и c/t.
    t = (c/t + t) / 2.0
stdio.writeln(t)
```

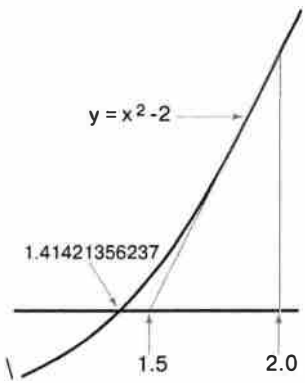
c	Аргумент
EPSILON	Защита от ошибки
t	Оценка c

Эта программа получает как аргумент командной строки положительное число c типа `float` и выводит его квадратный корень с точностью до 15 знаков после десятичной точки. Для вычисления квадратного корня она использует метод Ньютона.

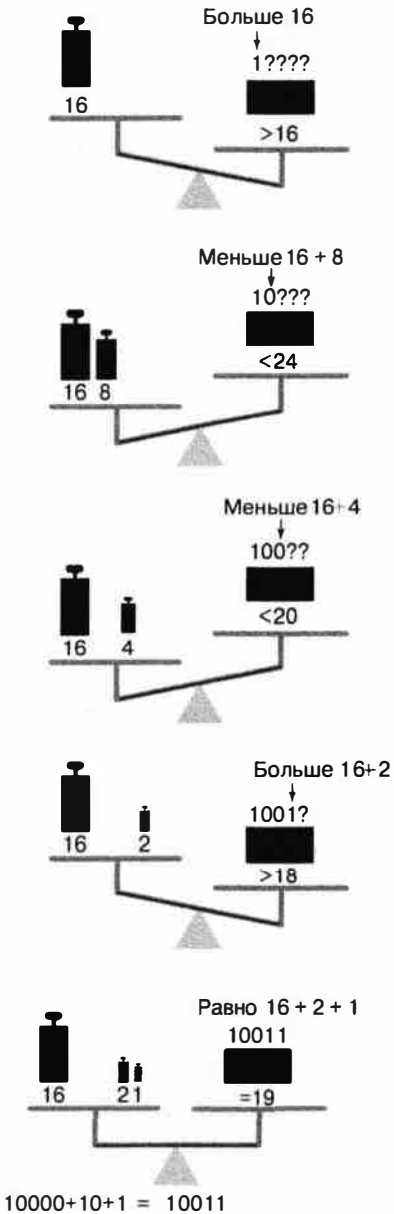
```
% python sqrt.py 2.0
1.414213562373095
% python sqrt.py 2544545
1595.1630010754388
```

Трассировка при c, равном 2.0

Итерация	t	c/t
1	2.00000000000	1.0
2	1.50000000000	1.33333333333
3	1.41666666667	1.41176470588
4	1.41421568627	1.41421143847
5	1.41421356237	1.41421356237



Еще один полезный подход для вычисления математических функций подразумевает использование ряда Тейлора (см. упр. 1.3.37 и 1.3.38). Его типичным применением является вычисление тригонометрических функций.



Метафорическое представление двоичного преобразования

Преобразование чисел. Программа 1.3.7 (`binary.py`) выводит двоичное (с основанием 2) представление десятичного числа, введенного как аргумент командной строки. Ее принцип основан на разложении числа на сумму степеней числа 2. Например, двоичным представлением числа 19 является 10011, что то же самое, что и 19 = 16 + 2 + 1. Для вычисления двоичного представления числа `n` мы находим степени числа 2, меньшие или равные `n` в порядке убывания, чтобы установить их принадлежность к двоичной декомпозиции (а следовательно, соответствию 1 биту в двоичном представлении). Этот процесс точно соответствует взвешиванию объекта на весах с гирями, вес которых кратен степеням числа 2. Сначала находим наибольшую гирю, которая не тяжелее объекта. Затем применим гири в порядке убывания, добавляя их по одной и проверяя, не тяжелее ли гири объекта. Если это так, то последнюю гирю снимаем; в противном случае оставляем гирю и пробуем следующую. Каждая гиря соответствует биту в двоичном представлении: оставшаяся гиря означает 1, а снятая — 0.

В программе `binary.py` переменная `v` соответствует проверяемой гире, переменная `n` — лишней (неизвестной) части веса объекта (для моделирования остатка веса на весах мы вычитаем этот вес из `n`). Значение `v` уменьшается согласно степени числа 2. Когда `v` больше `n`, программа `binary.py` выводит `0`; в противном случае она выводит `1` и вычитает `v` из `n`. Как обычно, трассировка (значений `n`, `v`, `n < v` и выводимого бита на каждой итерации цикла) может быть очень полезна для понимания работы программы. Читая сверху вниз правый столбец таблицы трассировки, получим вывод `10011` — двоичное представление числа 19.

Программа 1.3.7. Преобразование в двоичный формат (binary.py)

```
import sys
import stdio

n = int(sys.argv[1])

# Вычислить v как наибольшую степень 2 <= n.
v = 1
while v <= n // 2:
    v *= 2

# Приведение степени 2 в порядке убывания.
while v > 0:
    if n < v:
        stdio.write(0)
    else:
        stdio.write(1)
        n -= v
    v //= 2
stdio.writeln()
```

v	Текущая степень числа 2
n	Текущий избыток

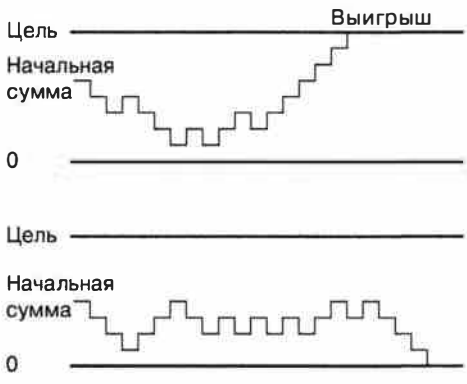
Эта программа выводит двоичное представление положительного целого числа, переданного как аргумент командной строки, в результате приведения степеней числа 2 в порядке убывания.

```
% python binary.py 19
10011
% python binary.py 255
11111111
% python binary.py 512
100000000
% python binary.py 100000000
101111101011110000100000000
```

Трассировка преобразования типов для случая `python binary.py 19`

n	Двоичное представление n	v	v > 0	Двоичное представление v	n < v	Вывод
19	10011	16	True	10000	False	1
3	0011	8	True	1000	True	0
3	011	4	True	100	True	0
3	01	2	True	10	False	1
1	1	1	True	1	False	1
0	0	False				

Преобразование данных от одного представления до другого является популярной темой компьютерных программ. Размышление о преобразовании выявляет различие между абстракцией (целое число, подобное количество часов в дне) и представлением этой абстракции (24 или 11000). Ирония здесь в том, что компьютерное представление целого числа фактически основано на двоичном формате.



Последовательность модели игры

Модель Монте-Карло. Наш следующий пример отличается по характеру от рассмотренных ранее, но он демонстрирует обычную ситуацию, когда компьютеры используются для моделирования того, что могло бы произойти в реальном мире, чтобы принимать более обоснованные решения. Данный конкретный пример относится к изучению класса задач, известных как *задача о разорении игрока* (gambler's ruin). Предположим, что игрок последовательно делает неограниченную серию ставок, начиная с некоторой заданной суммы. В конечном счете игрок всегда проиграет, но если установить некие пределы для игры, то возникают различные вопросы. Предположим, например, что игрок загодя решает уйти после получения определенного выигрыша. Какова вероятность, что игрок выиграет? Сколько ставок может понадобиться, чтобы выиграть или проиграть? Какова максимальная сумма, которую игрок будет иметь в течение игры?

Модель в программе 1.3.8 (`gambler.py`) может помочь ответить на эти вопросы. Для моделирования последовательности ставок, пока игрок не разорится или не достигнет цели, используется функция `random.randrange()`, а количество выигрышей и проигрышей отслеживается. После запуска эксперимента для определенного количества ставок программа вычисляет среднее и выводит результаты. Данную программу можно запускать с разными аргументами командной строки (и не обязательно планировать свой следующий поход в казино), это поможет обдумать следующие вопросы: действительно ли модель точно отражает происходящее в реальной жизни? Сколько попыток необходимо для получения точного ответа? Каковы вычислительные пределы при такой имитации? Моделирование широко используются в экономике, науке и технике, а вопросы такого рода очень важны для любой модели.

Программа 1.3.8. Модель разорения игрока (gambler.py)

```
import random
import sys
import stdio

stake = int(sys.argv[1])
goal = int(sys.argv[2])
trials = int(sys.argv[3])

bets = 0
wins = 0
for t in range(trials):
    # Запуск одной попытки.
    cash = stake
    while (cash > 0) and (cash < goal):
        # Модель одной ставки.
        bets += 1
        if random.randrange(0, 2) == 0:
            cash += 1
        else:
            cash -= 1
    if cash == goal:
        wins += 1
stdio.writeln(str(100 * wins // trials) + '% wins')
stdio.writeln('Avg # bets: ' + str(bets // trials))
```

stake	Начальная сумма
goal	Цель
trials	Количество попыток
bets	Счетчик ставок
wins	Счетчик выигрышей
cash	Наличность

В аргументах командной строки эта программа получает целочисленные значения stake (начальная сумма), goal (цель) и trials (попытки). Она осуществляет trials попыток, каждая из которых начинается с наличия stake фишек, а завершается 0 фишек или goal фишек. Затем она выводит процент выигрышей и среднее количество ставок в попытке. Внутренний цикл while моделирует игрока, имеющего stake фишек, делающего серию ставок по одной фишке до разорения или выигрыша goal фишек. Продолжительность работы этой программы пропорциональна полному количеству ставок (trials раз по среднему количеству ставок). Например, в последнем представленном случае было создано почти 100 миллионов случайных чисел.

```
% python gambler.py 10 20 1000
50% wins
Avg # bets: 100
% python gambler.py 50 250 100
19% wins
Avg # bets: 11050
% python gambler.py 500 2500 100
21% wins
Avg # bets: 998071
```

В случае gambler.py мы проверяем классические следствия теории вероятности, гласящие, что *вероятность успеха — это соотношение ставки к цели* и что ожидаемое *количество ставок — это произведение ставки и желаемого выигрыша* (различие между целью и ставкой). Например, если вы захотите посетить казино в Монте-Карло и попытаться превратить 500 евро в 2 500, то у вас есть разумный (20-процентный) шанс успеха, но вам придется сделать миллион ставок по 1 евро[2]! Если вы попытаетесь превратить 1 евро в 1 000, то при шансе 0,1 процента может потребоваться (если не проиграть раньше) примерно 999 ставок (в среднем).

Моделирование и анализ идут рука об руку, поддерживая друг друга. Практическое значение моделирования в том, что оно может предложить ответы на вопросы, слишком трудные для аналитического решения. Предположим, например, что наш игрок понял, что у него никогда не будет достаточно времени, чтобы сделать миллион ставок, и решает загодя установить верхний предел для количества ставок. Сколько денег игрок может ожидать унести домой в этом случае? Вы можете ответить на этот вопрос, внеся простые изменения в программу 1.3.8 (см. упр. 1.3.24), а вот решение той же задачи методами математического анализа не такое простое.

Разложение на множители. Простой множитель (prime) — это целое число, большее или равное 1, делящееся только на 1 и на себя. Разложение на простые множители целого числа — это набор множителей, произведение которых дает целое число. Например, 3757208 = 2*2*2*7*13*13*397. Программа 1.3.9 (factors.py) осуществляет разложение на простые множители любого заданного положительного целого числа. В отличие от большинства других рассмотренных программ (результат работы которых можно вычислить за несколько минут с помощью калькулятора или даже карандаша и бумаги), это вычисление нельзя выполнить без компьютера. Как бы вы попытались найти множители такого числа, как 287994837222311? Множитель 17 можно найти довольно быстро, но даже с калькулятором поиск множителя 1739347 займет довольно много времени.

Хотя программа factors.py коротка и понятна, потребуется некоторое воображение, чтобы убедить себя в том, что она дает желаемый результат для любого заданного целого числа. Как обычно, трассировка, демонстрирующая значения переменных в начале каждой итерации внешнего цикла for, облегчит понимание вычислений. Для случая, где n первоначально равно 3757208, внутренний цикл while выполняется три раза, когда factor равно 2, чтобы удалить три множителя 2; затем ни разу для factor 3, 4, 5 и 6, так как ни одно из тех чисел не делит 469651; и т.д..

[2] В реальности минимальная ставка 5 евро. — *Примеч. ред.*

Программа 1.3.9. Разложение на множители целых чисел (`factors.py`)

```
import sys
import stdio

n = int(sys.argv[1])

factor = 2
while factor*factor <= n:
    while (n % factor) == 0:
        # Вычислить и вывести множитель.
        n //= factor
        stdio.write(str(factor) + ' ')
    factor += 1
    # Все множители n больше или равны factor.

if n > 1:
    stdio.write(n)
stdio.writeln()
```

`n`	Не разложенная на множители часть
`factor`	Потенциальный множитель

Эта программа выводит разложение на простые множители любого положительного целого числа. Код прост, но трудно убедить себя в том, что он правильный (см. текст).

```
% python factors.py 3757208
2 2 2 7 13 13 397
% python factors.py
287994837222311
17 1739347 9739789
```

Трассировка программы для нескольких примеров входных данных ясно показывает простоту действия. Чтобы убедиться в правильности поведения программы для всех входных данных, давайте обсудим, что мы ожидаем на каждом из циклов. Внутренний цикл `while` просто выводит и удаляет из `n` все множители `factor`. Для понимания программы следует заметить, что следующий инвариант содержится в начале каждой итерации внешнего цикла `while`: у `n` нет никаких множителей между `2` и `factor-1`. Таким образом, если `factor` не будет простым множителем, то он не будет делиться на `n`; если `factor` будет простым множителем, то цикл `while` сделал свою работу и инвариант останется таковым. Поиск множителей прекратится, когда значение `factor*factor` превысит значение `n`, поскольку, если у целого числа `n` есть множитель, он меньше или равен квадратному корню `n`.

В более наивной реализации мы, возможно, использовали бы для завершения внешнего цикла просто условие (`factor < n`). Даже при скорости вычислений современных компьютеров такое решение окажет драматическое воздействие на размер чисел, которые мы могли бы разложить. Упражнение 1.3.26 предлагает поэкспериментировать с программой и исследовать влияние этого простого изменения. На компьютере, способном осуществлять миллиарды операций в секунду, мы смогли разложить числа порядка 10^9 за несколько секунд; при условии (`factor*factor <= n`) за сопоставимый период времени мы смогли разложить числа порядка 10^{18}. Циклы позволяют решать трудные задачи, но они позволяют также писать простые программы, способные выполняться довольно долго, поэтому всегда следует помнить о проблемах эффективности.

В современных приложениях, подобных криптографическим, есть ситуации, когда приходится разлагать действительно огромные числа (скажем, сотни или тысячи цифр). Даже для экспертов такие вычисления (во всяком случае, пока) оказались предельно трудными, несмотря на использование компьютера.

Трассировка при `n`, равном 3757208

factor	n	Вывод
2	3757208	2 2 2
3	469651	
4	469651	
5	469651	
6	469651	
7	469651	7
8	67093	
9	67093	
10	67093	
11	67093	
12	67093	
13	67093	13 13
14	397	
15	397	
16	397	
17	397	
18	397	
19	397	
20	397	
		397

Неполный цикл. Иногда структура управления потоком `for` или `while` не очень точно соответствует необходимому циклу. Предположим, например, что необходим цикл, многократно выполняющий некую последовательность операторов и завершающийся, если удовлетворяется некое условие выхода, а затем продолжающий выполнять некую другую последовательность операторов. Таким образом, условие выхода следует расположить в середине цикла, а не в начале. Это *неполный цикл* (loop and a half), поскольку перед проверкой условия выхода часть цикла придется пройти. Для этого Python предоставляет оператор `break`. Встретив оператор `break`, Python немедленно выходит из самого внутреннего цикла.

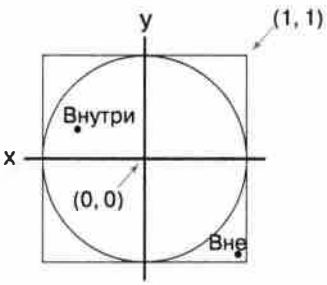
Рассмотрим, например, задачу генерации точек, случайно распределенных в единичном круге. Для получения координат x и y точек, случайно распределенных в квадрате 2×2 с центром в начале координат, можно использовать функцию `random.random()`. Большинство этих точек окажется в пределах единичного круга, поэтому достаточно отбросить те, которые в него не попадают.

Поскольку необходима по крайней мере одна точка, создаем бесконечный цикл `while`, условие продолжения которого всегда удовлетворяется. Он создает случайную точку (x, y) в четырехугольнике 2×2, а оператор `break` используется для завершения цикла, если координаты (x, y) находятся в круге.

```
while True:
    x = 1.0 + 2.0*random.random()
    y = 1.0 + 2.0*random.random()
    if x*x + y*y <= 1.0:
        break
```

Поскольку площадь единичного круга равна π, а площадь квадрата — 4, ожидаемое количество итераций цикла составляет $4/\pi$ (примерно 1,27).

Эксперты осуждают такой выход из внутренних циклов. При неправильном использовании операторы `break` способны существенно усложнить управление потоком в цикле. Но в некоторых ситуациях (как в упражнении 1.3.30) альтернативы также могут быть сложными. Некоторые языки предоставляют для таких ситуаций конструкцию `do-while`. В языке Python мы рекомендуем разумное использование оператора `break` (только в случае крайней необходимости).



Бесконечные циклы. Прежде чем создавать программы, использующие циклы, необходимо обдумать следующую проблему: что, если условие продолжения цикла `while` всегда удовлетворяется? С изученными на настоящий момент операторами могла случиться одна из двух неприятностей, с обеими из которых необходимо уметь справляться.

Первая. Предположим, что такой цикл вызывает функцию `stdout.writeln()`. Если бы условие продолжения цикла в программе `tenhellos.py` было `I > 3` вместо `i <= 10`, то оно всегда было бы истинно. Что бы было? В настоящее время мы используем термин *вывод* как абстракцию, означающую *отображение в окне терминала*, а результат попытки отображения бесконечного количества строк в окне терминала зависит от соглашений операционной системы. Если в вашей системе *вывод* означает *распечатку на бумаге*, то бумага в принтере закончится и он остановится. В окне терминала необходима команда остановки вывода. Прежде чем самому запускать программу с циклами, следует удостовериться, что вы знаете, как прервать бесконечный цикл с оператором `stdout.writeln()`, а затем проверить эту стратегию, внеся в программу `tenhellos.py` описанные ранее изменения и попытавшись остановить ее. На большинстве операционных систем для остановки текущей программы используется комбинация клавиш <Ctrl+c>.

Вторая. *Ничего* не происходит. По крайней мере, ничего видимого. Если бесконечный цикл вашей программы не создает вывод, то она будет выполнять цикл постоянно, но никаких результатов не будет видно вообще. Программа

```
import stdio
i = 4
while i > 3:
    stdio.write(i)
    stdio.writeln('th Hello')
    i += 1
```

Бесконечный цикл (с выводом)

```
% python infiniteloop1.py
1st Hello
2nd Hello
3rd Hello
5th Hello
6th Hello
7th Hello
8th Hello
9th Hello
10th Hello
11th Hello
12th Hello
13th Hello
14th Hello
...
```

```
while True:
    x = random.random()
    y = random.random()
    if x*x + y*y >= 2.0:
        break
```

Бесконечный цикл (без вывода)

```
% python infiniteloop2.py
...
```

кажется зависшей. Оказавшись в такой ситуации, вы можете исследовать циклы и удостовериться, что условие выхода всегда срабатывает, но эта задача не так проста. Один из способов поиска такой ошибки подразумевает добавление вызовов функции `stdout.writeln()` для трассировки. Если эти вызовы находятся в пределах бесконечного цикла, они переводят проблему к обсуждаемой в предыдущем абзаце, а вывод может оказаться полезен для выяснения ее причин.

Вы можете не знать, бесконечен ли цикл или только очень долог (или это может не иметь значения). Если запустить программу `gambler.py` с такими аргументами, как `python gambler.py 100000 200000 100`, то можно и не дождаться ответа. Впоследствии вы научитесь узнавать и оценивать продолжительность работы программ. Этой теме посвящен раздел 4.1.

Почему Python не обнаруживает бесконечные циклы и не предупреждает нас о них? Как ни удивительно, но этого невозможно сделать вообще. Этот загадочный факт — один из фундаментальных результатов теоретической информатики.

Резюме. Для справки ниже прилагается таблица программ, рассматривавшихся в этом разделе. Это примеры различных видов задач, которые могут решать короткие программы, использующие операторы `if`, `while` и `for` для обработки данных встроенных типов. Эти виды вычислений — правильный способ знакомства с базовыми конструкциями управления потоком Python. Время, потраченное на работу с такими программами, с лихвой окупится в будущем.

Чтобы научиться использовать условные выражения и циклы, следует попрактиковаться в создании и отладке программ с операторами `if`, `while` и `for`. Упражнения в конце этого раздела предоставляют много возможностей, чтобы самостоятельно начать этот процесс. Для каждого упражнения вы составите программу Python, а затем запустите и проверите ее. Все программисты знают, что программы никогда не работают сразу, как планировалось вначале, поэтому

в программе придется разобраться и выяснить, что нужно делать. Сначала используйте явные трассировки для проверки своего понимания и ожиданий. По мере приобретения опыта в создании собственных циклов вы начнете думать в терминах того, что могла бы дать трассировка. Задайте себе следующие виды вопросов: каковы будут значения переменных после первой итерации цикла? После второй? После последней? Есть ли у этой программы способ попасть в бесконечный цикл?

Список программ этого раздела

Программа	Описание
`flip.py`	Модель броска монеты
`tenhellos.py`	Ваш первый цикл
`powersoftwo.py`	Вычислить и вывести таблицу значений
`divisorpattern.py`	Ваш первый вложенный цикл
`harmonic.py`	Вычислить конечную сумму
`sqrt.py`	Классический итерационный алгоритм
`binary.py`	Простое преобразование чисел
`gambler.py`	Модель с вложенными циклами
`factors.py`	Цикл `while` внутри цикла `while`

Циклы и условные выражения — это гигантский шаг в возможности вычислений: операторы `if`, `while` и `for` переводят нас от простых прямолинейных программ к произвольно сложному управлению потоком. В нескольких следующих главах мы сделаем еще более гигантские шаги, позволяющие обрабатывать большие объемы исходных данных, а также определять и обрабатывать типы данных, отличные от простых числовых типов. Операторы `if`, `while` и `for` этого раздела играют основную роль в программах, рассматриваемых на этих этапах.

Вопросы и ответы

В чем разница между = и ==?

Мы повторяем этот вопрос здесь, чтобы напомнить, что недопустимо использовать оператор = в условном выражении, где действительно подразумевается ==. Оператор `x = y` присваивает переменной `y` значение переменной `x`, тогда как выражение `x == y` проверяет равенство этих двух значений. В некоторых языках программирования такую ошибку может быть трудно обнаружить в программе. В языке Python операторы присвоения не являются выражениями. Например, если в программе (скажем, 1.3.8) ошибочно ввести `cash = goal` вместо `cash == goal`, то компилятор сам найдет ошибку:

```
% python gambler.py 10 20 1000
    File "gambler.py", line 21
        if cash = goal:
                ^
SyntaxError: invalid syntax
```

Что будет, если пропустить двоеточие в операторах `if`, `while` или `for`?

Ошибка `SyntaxError` во времени компиляции.

Каковы правила отступа для блоков операторов?

У каждого оператора в блоке должен быть одинаковый отступ; если это не так, то во время компиляции произойдет ошибка `IndentationError`. Обычно программисты Python используют для отступа четыре пробела, как в этой книге.

Можно ли для отступов использовать символы табуляции?

Нет, символов табуляции в файлах `.py` следует избегать. Многие текстовые редакторы позволяют автоматически вставлять набор пробелов при нажатии клавиши <Tab>. При создании программ Python эта возможность очень удобна.

Могу ли я распространить длинный оператор на несколько строк?

Да, но необходимо соблюдать осторожность из-за способа, которым Python учитывает отступ. Если охватывающее несколько строк выражение заключается в круглые скобки (или квадратные, или фигурные), то нет никакой необходимости в чем-то особенном. Например, следующий единый оператор распространен на три строки:

```
stdio.write(a0 + a1 + a2 + a3 +
    a4 + a5 + a6 + a7 +
    a8 + a9)
```

Но если однозначного продолжения строки нет, то придется использовать символ наклонной черты влево в конце каждой строки с продолжением.

```
total = a0 + a1 + a2 + a3 + \
    a4 + a5 + a6 + a7 + \
    a8 + a9
```

Предположим, что я хочу перескочить через часть кода в цикле или чтобы тело условного оператора было пусто и никаких операторов не выполнялось. Есть ли у языка Python поддержка для таких вещей?

Да, для этих целей язык Python предоставляет операторы `continue` и `pass` соответственно. Однако ситуации, в которых они действительно необходимы, редки, и мы не используем их в этой книге. Кроме того, в языке Python нет никакого аналога оператора *goto* (для взаимоисключающих альтернатив), хотя другие языки его еще иногда поддерживают (для неструктурного контроля потока).

Могу ли я использовать не логическое выражение в операторе `if` или `while`?

Да, но это не лучшая идея. Если результат выражения нуль или пустая строка, он считается как `False`; все другие числовые и строковые выражения считаются `True`.

Есть ли случаи, когда я должен использовать оператор `for`, а не `while`, и наоборот?

Оператор `while` позволяет реализовать любой вид цикла, но, как уже упоминалось, оператор `for` имеет смысл использовать для цикла, перебирающего конечную последовательность целых чисел. Позже (см. разделы 1.4, 3.3 и 4.4) мы рассмотрим и другие способы использования оператора `for`.

Могу ли я использовать встроенную функцию `range()` для создания последовательности целых чисел с шагом, отличным от 1?

Да, функция `range()` поддерживает необязательный третий аргумент `step` со стандартным значением `1`. Таким образом, вызов `range(start, stop, step)` дает последовательность целых чисел `start`, `start + step`, `start + 2 * step` и т.д. Если `step` — положительное целое число, то последовательность продолжается, пока значение `start + i * step` меньше, чем `stop`; если `step` — отрицательное целое число, то последовательность продолжается, пока `start + i * step` больше, чем `stop`. Например, вызов `range(0, -100, -1)` возвращает целочисленную последовательность 0, –1, –2, ..., –99.

Могу ли я использовать как аргументы функции `range()` числа типа `float`?

Нет, все аргументы должны быть целыми числами.

Могу ли я изменить индексную переменную в пределах цикла `for`?

Да, но это не повлияет на последовательность целых чисел, создаваемую функцией `range()`. Например, следующий цикл выведет 100 целых чисел от 0 до 99:

```
for i in range(100):
    stdio.writeln(i)
    i += 10
```

Каково значение управляющей переменной цикла `for` после его завершения?

Это последнее значение управляющей переменной во время цикла. После завершения приведенного выше цикла `for` переменная `i` ссылается на целое число `109`. Использование управляющей переменной цикла после завершения цикла `for` считается плохим стилем программирования, поэтому в своих программах мы так не поступаем.

Упражнения

1.3.1. Составьте программу, получающую в командной строке три целочисленных аргумента и выводящую слово `'equal'`, если все три равны, и `'not equal'` в противном случае.

1.3.2. Составьте более общую и надежную версию программы 1.2.4 (`quadratic.py`), выводящую корни уравнения $ax^2 + bx + c$, выводящую соответствующее сообщение, если дискриминант отрицателен, и принимающую соответствующие меры, избегая деления на нуль.

1.3.3. Составьте фрагмент кода, получающий два аргумента командной строки типа `float` и выводящую `True`, если оба находятся в диапазоне от `0.0` до `1.0`, и `False` в противном случае.

1.3.4. Улучшите решение упражнения 1.2.22, добавив код проверки аргументов командной строки на принадлежность к диапазону допустимых для формулы температуры воздуха с учетом влияния ветра и код вывода сообщения об ошибке, если они не в диапазоне.

1.3.5. Каково значение `j` после выполнения каждого из следующих фрагментов кода?

```
a. j = 0
   for i in range(j, 10):
       j += i
b. j = 0
   for i in range(10):
     j += j
c. for j in range(10):
     j += j
```

1.3.6. Переделайте программу `tenhellos.py`, объединив ее с программой `hellos.py` так, чтобы она получала в аргументе командной строки количество выводимых строк. Можно считать, что аргумент меньше `1000`. *Подсказка*: чтобы решить, когда применять `st`, `nd`, `rd` или `th` при выводе `i`-го сообщения `Hello`, используйте выражения `i % 10` и `i % 100`.

1.3.7. Составьте программу `fiveperline.py`, использующую один цикл `for` и один оператор `if` для вывода, по пять в строку, целые числа от 1 000 (включительно) до 2 000 (исключительно). *Подсказка*: используйте оператор `%`.

1.3.8. Обобщая упражнение о *равномерных случайных числах* из раздела 1.2 (см. упр. 1.2.27), составьте программу `stats.py`, получающую в аргументе командной строки целое число `n` и использующую функцию `random.random()` для вывода `n` равномерно случайных чисел от 0 до 1, а затем выводящую их среднее, минимальное и максимальное значения.

1.3.9. В этом разделе был представлен следующий код, реализующий линейчатую функцию:

```
ruler = '1'
stdio.writeln(ruler)
for i in range(2, n+1):
    ruler = ruler + ' ' + str(i) + ' ' + ruler
    stdio.writeln(ruler)
```

Опишите происходящее при запуске этого кода, если значение `n` слишком велико (например, 100).

1.3.10. Составьте программу `functiongrowth.py`, выводящую таблицу значений $\log_2 n$, n, $n \log_e n$, n^2, n^3 и 2^n для $n = 2, 4, 8, 16, 32, 64, \ldots, 2048$. Для выравнивания столбцов таблицы используйте табуляцию (символы `\t`).

1.3.11. Каковы значения `m` и `n` после выполнения следующего кода?

```
n = 123456789
m = 0
while n != 0:
    m = (10 * m) + (n % 10)
    n /= 10
```

1.3.12. Что выводит этот код?

```
f = 0
g = 1
for i in range(16):
    stdio.writeln(f)
    f = f + g
    g = f - g
```

Решение: даже опытный программист скажет, что единственный способ понять работу программы — трассировка. Сделав это, вы обнаружите, что она выводит значения 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 134, 233, 377 и 610. Эти числа — первые 16 из *последовательности Фибоначчи*, определяемой следующими формулами: $F_0 = 0$, $F_1 = 1$ и $F_n = F_{n-1} + F_{n-2}$ для $n > 1$. Последовательность Фибоначчи на удивление распространена; она известна на протяжении многих столетий, и большинство ее свойств широко используется. Например, когда n стремится к бесконечности, соотношение последовательных чисел приближается к *золотому сечению* (golden ratio) (примерно 1,618).

1.3.13. Составьте программу, получающую в аргументах командной строки `n` и выводящую все положительные степени числа 2, меньшие или равные `n`.

Удостоверьтесь, что программа работает правильно для всех значений `n`. (Для отрицательных `n` программа не должна выводить ничего.)

1.3.14. Модифицируйте свое решение упражнения о *непрерывно начисляемом сложном проценте* из раздела 1.2 (см. упр. 1.2.21) так, чтобы оно выводило таблицу с общей суммой выплат и остающейся основной суммой вклада после каждой ежемесячной выплаты.

1.3.15. Составьте версию программы 1.3.4 (`divisorpattern.py`), использующую циклы `while` вместо циклов `for`.

1.3.16. В отличие от гармонических чисел, сумма последовательности $1/1^2 + 1/2^2 + \ldots + 1/n^2$ *действительно* сходится к константе при n, стремящемся к бесконечности. (Поскольку эта константа — $\pi^2/6$, данная формула используется для вычисления значения числа π.) Какой из следующих циклов `for` вычисляет эту сумму? Подразумевается, что `n` — это целое число `1000000`, а переменная `total` типа `float` инициализирована значением `0.0`.

a.
```
for i in range(1, n+1):
    total += 1 / (i*i)
```

b.
```
for i in range(1, n+1):
    total += 1.0 / i*i
```

c.
```
for i in range(1, n+1):
    total += 1.0 / (i*i)
```

d.
```
for i in range(1, n+1):
    total += 1.0 / (1.0*i*i)
```

1.3.17. Расскажите, как программа 1.3.6 (`sqrt.py`) реализует метод Ньютона при поиске квадратного корня. *Подсказка*: используйте факт, что наклон касательной к (дифференцируемой) функции $f(x)$ при $x = t$ составляет $f'(t)$. Это позволяет найти уравнение касательной линии, а затем использовать его для поиска точки, где касательная линия пересекает ось X. Это позволяет использовать метод Ньютона для поиска корня любой функции следующим образом: при каждой итерации заменяйте оценку t на $t - f(t)/f'(t)$.

1.3.18. Используя метод Ньютона, разработайте программу, получающую в аргументах командной строки целые числа `n` и `k` и выводящую корень `k`th числа `n` (*Подсказка*: см. упр. 1.3.17.)

1.3.19. Измените программу `binary.py` так, чтобы создать программу `kary.py`, получающую в аргументах командной строки значения `i` и `k`, а затем преобразующую `i` в основание `k`. Подразумевается, что `k` — целое число от 2 до 16. Для оснований, больших, чем 10, используются символы от `A` до `F`, соответствующие цифрам от 11 до 16.

1.3.20. Составьте фрагмент кода, преобразующий двоичное представление положительного целого числа n в строку s.

Решение: эту задачу решает программа 1.3.7

```
s = ''
v = 1
while v <= n//2:
    v *= 2
while v > 0:
    if n < v:
        s += '0'
    else:
        s += '1'
        n -= v
    v //= 2
```

Упрощенный вариант работает справа налево:

```
s = ''
while n > 0:
    s = str(n % 2) + s
    n /= 2
```

Оба этих метода заслуживают внимания.

1.3.21. Составьте версию gambler.py, который использует два вложенных цикла while или два вложенных цикла while вместо цикла while в цикле for.

1.3.22. Составьте программу gamblerplot.py, отслеживающую модель разорения игрока, выводя строки после каждой ставки, где одна звездочка соответствует каждой фишке, принадлежащей игроку.

1.3.23. Измените программу gambler.py так, чтобы она получала дополнительный аргумент командной строки, задающий (фиксированную) вероятность выигрыша при каждой ставке. Используйте эту программу для проверки того, как эта вероятность влияет на шанс достижения цели и количество ожидаемых ставок. Опробуйте значения, близкие к 0,5 (скажем, 0,48).

1.3.24. Измените программу gambler.py так, чтобы она получала дополнительный аргумент командной строки, задающий количество ставок, которые игрок желает сделать до завершения игры тремя возможными способами: победа, проигрыш или закончилось время. Добавьте в вывод ожидаемую сумму денег, которую будет иметь игрок по завершении игры. *Дополнительное задание*: используйте свою программу при планировании своей следующей поездки в Монако.

1.3.25. Измените программу `factors.py` так, чтобы выводить только одну копию каждого из простых множителей.

1.3.26. Запустите программу 1.3.9 (`factors.py`) на время, чтобы выяснить воздействие использования условия завершения цикла `(i < n)` вместо `(i *i <= n)`. Для каждого случая найдите наибольшее `n`, чтобы при вводе n-разрядного числа программа закончилась в течение 10 секунд.

1.3.27. Составьте программу `checkerboard.py`, получающую один аргумент командной строки `n` и использующую вложенный цикл для вывода двумерного узора `n` × `n`, наподобие шахматной доски с чередующимися пробелами и звездочками, как в следующем узоре 5 × 5:

```
* * * * *
 * * * *
* * * * *
 * * * *
* * * * *
```

1.3.28. Составьте программу `gcd.py`, находящую *наибольший общий делитель* (greatest common divisor — gcd) для двух целых чисел. Используйте *алгоритм Евклида*, подразумевающий итерационные вычисления на основании следующего наблюдения: если `x` больше, чем `y`, и если `y` делит `x`, наибольший общий делитель `x` и `y` — это `y`; в противном случае наибольший общий делитель `x` и `y` — это `x % y` и `y`.

1.3.29. Составьте программу `relativelyprime.py`, получающую один аргумент командной строки `n` и выводящую таблицу `n` × `n`, где `*` устанавливается в строке `i` и столбце `j`, если наибольший общий делитель `i` и `j` составляет 1 (`i` и `j` являются относительно простыми множителями), и пробел в противном случае.

1.3.30. Составьте программу, создающую точку, случайно расположенную на единичном круге, но без использования оператора `break`. Сравните свое решение с таковым в конце этого раздела.

1.3.31. Составьте программу, выводящую координаты случайной точки (a, b, c) на поверхности сферы. Чтобы создать такую точку, используйте *метод Марсальи*: начните с выбора случайной точки (x, y) на единичном круге, используя метод, описанный в конце этого раздела. Затем присвойте a значение $2x\sqrt{1-x^2-y^2}$, b — значение $2\sqrt{1-x^2-y^2}$ и c — значение $1-2(x^2-y^2)$.

Практические упражнения

1.3.32. *Такси Рамануджана.* С. Рамануджан — индийский математик, славившийся своей интуицией в области чисел. Когда английский математик Г. Х. Харди навестил его однажды в больнице, он обмолвился, что номером такси, на котором он приехал, было 1729, такое скучное и заурядное число. На что Рамануджан ответил: "Нет, нет! Это очень интересное число. Это наименьшее число, выражаемое как сумма двух кубов двумя разными способами". Проверьте это заявление, создав программу, получающую `n` в аргументе командной строки и выводящую все целые числа, меньшие или равные `n`, которые могут быть выражены как сумма двух кубов двумя разными способами. Другими словами, найдите различные положительные целые числа a, b, c и d, удовлетворяющие условию $a^3 + b^3 = c^3 + d^3$. Используйте четыре вложенных цикла `for`.

1.3.33. *Контрольная сумма.* Международный стандартный номер книги (International Standard Book Number — ISBN) — это 10-значный цифровой код, уникально идентифицирующий книгу. Крайняя справа цифра — это *контрольная сумма* (checksum), которая может быть вычислена из 9 других цифр согласно условию, что $d_1 + 2d_2 + 3d_3 + \ldots + 10d_{10}$ кратно 11 (здесь d_i означает i-ю цифру справа). Цифра контрольной суммы d_i может быть любым значением от 0 до 10: по соглашению ISBN, для обозначения цифры 10 используется символ `'X'`. *Пример*: контрольной суммой номера 020131452 является 5, поскольку 5 — единственное значение `X` от 0 до 10, для которого результат выражения $10 \cdot 0 + 9 \cdot 2 + 8 \cdot 0 + 7 \cdot 1 + 6 \cdot 3 + 5 \cdot 1 + 4 \cdot 4 + 3 \cdot 5 + 2 \cdot 2 + 1 \cdot X$ кратен 11. Составьте программу, получающую в аргументах командной строки целое число из 9 цифр, вычислите контрольную сумму и выведите ISBN.

1.3.34. *Вычисление множителей.* Составьте программу `primecounter.py`, получающую в аргументах командной строки число `n` и выводящую количество простых множителей, меньших или равных `n`. Используйте ее для вывода количества множителей, меньших или равных 10 миллионам. *Примечание*: если вы не сделаете все возможное для повышения эффективности программы, она не сможет завершить работу за разумный период времени. Далее, в разделе 1.4, вы узнаете о более эффективном способе этого вычисления с использованием *решета Эратосфена* (программа 1.4.3).

1.3.35. *Двумерный метод случайного блуждания.* Двумерный метод случайного блуждания моделирует поведение частицы, двигающейся на координатной плоскости. На каждом этапе частица случайно двигается на север, на юг,

на восток или на запад с вероятностью 1/4, независимо от предыдущих ходов. Составьте программу `randomwalker.py`, получающую в аргументе командной строки число `n` и оценивающую количество случайных ходов до достижения границы четырехугольника 2n × 2n с центром в отправной точке.

1.3.36. *Срединное из 5.* Составьте программу, получающую в аргументах командной строки пять разных целых чисел и выводящую срединное из них (два числа больше него, а два других меньше). *Дополнительное задание*: решите задачу, использовав меньше семи сравнений для любого ввода.

1.3.37. *Экспоненциальная функция.* Подразумевается, что `x` имеет тип `float`. Составьте фрагмент кода, использующий разложение в ряд Тейлора для присвоения переменной `total` результата выражения $e^x = 1 + x + x^2/2! + x^3/3! + \ldots$.

Решение: задача этого упражнения — обдумать реализацию библиотечной функции `math.exp()` с точки зрения элементарных операторов. Попробуйте решить задачу самостоятельно, а затем сравните свое решение с приведенным здесь.

Начнем с рассмотрения проблемы вычисления одной сроки. Предположите, что `x` имеет тип `float`, а `n` — тип `int`. Следующий код присваивает переменной `term` значение $x^n/n!$, используя прямой метод с одним циклом для числителя и другим циклом для знаменателя, а затем разделив результаты:

```
num = 1.0
den = 1.0
for i in range(1, n+1):
    num *= x
for i in range(1, n+1):
    den *= i
term = num / den
```

Но лучше использовать только один цикл `for`:

```
term = 1.0
for i in range(1, n+1):
    term *= x / i
```

Кроме того, что последнее решение компактней и изящней, оно предпочтительней, поскольку избегает погрешностей, связанных с вычислением больших чисел. Например, подход с двумя циклами разделен для таких значений, как $x = 10$ и $n = 100$, поскольку значение 100! слишком велико для точного представления как `float`.

Чтобы вычислить значение e^x, мы вкладываем этот цикл `for` в другой цикл `for`:

```
term = 1.0
total = 0.0
n = 1
while total != total + term:
    total += term
    term = 1.0
    for i in range(1, n+1):
        term *= x / i
    n += 1
```

Количество итераций цикла `while` зависит от относительных значений следующего элемента и накопленной суммы. Как только `total` перестает изменяться, цикл останавливается. (Эта стратегия эффективней использования условия завершения `(term > 0)`, так как позволяет избежать существенного количества итераций, не изменяющих значения суммы.) Этот код эффективней, но не до конца, поскольку внутренний цикл `for` повторно вычисляет все значения, которые он вычислил на предыдущей итерации внешнего цикла `for`. Вместо этого можно использовать элемент, добавленный на предыдущей итерации цикла, и решать задачу одиночным циклом `while`:

```
term = 1.0
total = 0.0
n = 1
while total != total + term:
    total += term
    term *= x/n
    n += 1
```

1.3.38. *Тригонометрические функции.* Составьте программы `sine.py` и `cosine.py`, вычисляющие функции синуса и косинуса с использованием разложения в ряд Тейлора $\sin x = x - x^3/3! + x^5/5! - \ldots$ и $\cos x = 1 - x^2/2! + x^4/4! - \ldots$

1.3.39. *Экспериментальный анализ.* Проведите эксперимент по сравнению библиотечной функции `math.exp()` и тремя подходами из упражнения 1.3.37 по вычислению e^x: прямой метод с вложенным циклом `for`, усовершенствованный с одиночным циклом `while`, и последний, с условием завершения `(term > 0)`. Используйте метод проб и ошибок, а также аргументы командной строки, чтобы определить, сколько раз компьютер может выполнить каждое вычисление за 10 секунд.

1.3.40. *Задача Пипса.* В 1693 году Сэмюэл Пипс спросил Исаака Ньютона, что вероятней: получить единицу за шесть бросков игральной кости или

дважды по единице за 12. Составьте программу, которая позволила бы Ньютону дать ответ.

1.3.41. *Модель игры.* В 1970-х годах была телевикторина *Давайте заключим сделку* (Let's Make a Deal), где игроку предлагалось три двери. За одной из них был ценный приз. После того как игрок выбирал дверь, ведущий открывал одну из других двух дверей (никогда не попадая на приз, конечно). Затем игроку предоставлялась возможность сменить дверь на другую неоткрытую. Должен ли был игрок это делать? Интуитивно могло бы показаться, что первая выбранная дверь и вторая неоткрытая дверь с одинаковой вероятность будут содержать приз, таким образом, в смене не было бы никакого смысла. Составьте программу `montehall.py` для проверки этой ситуации. Программа должна получать в аргументах командной строки значение `n` — количество попыток использования каждой из этих двух стратегий (менять или не менять), а затем вывести шанс успеха для каждой из стратегий.

1.3.42. *Хаос.* Составьте программу для простой модели прироста населения, применимую для изучения популяции рыб в водоеме, бактерий в пробирке и любых подобных ситуаций. Предполагается, что популяция колеблется от 0 (вымирание) до 1 (максимально возможная). Если популяция в момент времени t составляет x, то, как можно предположить, в момент $t + 1$ она составит $rx(1 - x)$, где r — *коэффициент плодовитости* (fecundity parameter), контролирующий прирост населения. Начнем с небольшой популяции (скажем, $x = 0.01$) и исследуем результат итераций модели для разных значений r. При каких значениях r популяция стабилизируется на уровне $x = 1 - 1/r$? Что можно сказать о популяции при r, равном 3.5, 3.8 и 5?

1.3.43. *Гипотеза Эйлера о сумме степеней.* В 1769 году Леонард Эйлер сформулировал обобщенную версию Великой теоремы Ферма, предполагая, что по крайней мере n энных степеней необходимо для получения суммы, которая сама является энной степенью для $n>2$. Составьте программу для опровержения гипотезы Эйлера (продержавшейся до 1967 года), используя пятикратно вложенный цикл для поиска четырех положительных целых чисел, сумма 5-х степеней которых равна 5-й степени другого положительного целого числа. Таким образом, найдите пять целых чисел a, b, c, d и e, удовлетворяющих условию таким образом, что $a^5 + b^5 + c^5 + d^5 = e^5$.

1.4. Массивы

В этом разделе рассматривается концепция *структуры данных* и первой структуры данных, *массива*. Основная задача массива — облегчить хранение больших объемов данных и манипулирование ими. Массивы играют существенную роль во многих задачах обработки данных. Они соответствуют также векторам и матрицам, широко используемым в науке и научном программировании. Итак, рассмотрим фундаментальные свойства массива в языке Python и проиллюстрируем их примерами применения.

Программы этого раздела...

Программа 1.4.1. Выборка без замены (sample.py) 122

Программа 1.4.2. Модель коллекции купонов (couponcollector.py) 126

Программа 1.4.3. Решето Эратосфена (primesieve.py) 128

Программа 1.4.4. Случайные блуждания без самопересечений (selfavoid.py) 136

Структура данных (data structure) — это способ организации данных, необходимый для их обработки компьютерной программой. Структуры данных играют важную роль в программировании (глава 4 этой книги посвящена исследованию классических структур данных всех типов).

Одномерный массив (one-dimensional array), или просто *массив* (array), — это структура данных, хранящая *последовательность* (sequence) ссылок на объекты. Хранимые в массивах объекты являются его *элементами* (element). Для обращения к элементам массива их *нумеруют*, а затем *индексируют*. Если в последовательности имеется n элементов, то их считают пронумерованными от 0 до $n-1$. Любой из них можно однозначно идентифицировать по номеру (индексу `i`) в этом диапазоне.

Двумерный массив (two-dimensional array) — массив ссылок на одномерные массивы. Принимая во внимание, что элементы одномерного массива индексируются одиночным целым числом, элементы двумерного массива индексируются двумя целыми числами: первое — определение ряда, второе — столбца.

Зачастую, когда объем данных велик, перед обработкой их помещают в один или несколько массивов, а затем, используя индексацию, обращаются к их индивидуальным элементам и обрабатывают. Это могут быть таблицы результатов экзамена, курсы акций, нуклеотиды в цепочке ДНК или буквы в книге. Каждый из этих примеров подразумевает большое количество объектов одинакового типа. Мы рассмотрим такие приложения при обсуждении ввода данных в разделе 1.5 и при анализе проблем в разделе 1.6. Данный раздел посвящен фундаментальным свойствам массивов; в рассматриваемых здесь примерах программы сначала заполняют массивы объектами подлежащих обработке экспериментальных данных, а затем обрабатывают их.

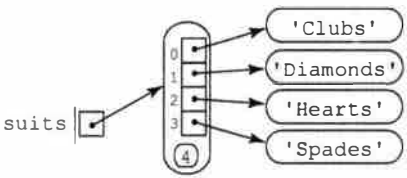
Массивы Python. Самый простой способ создания массива в языке Python — это поместить разделяемый запятыми список литералов в квадратные скобки. Например, код

```
suits = ['Clubs', 'Diamonds', 'Hearts', 'Spades']
```

создает массив `suits[]` из четырех строк, а код

```
x = [0.30, 0.60, 0.10]
y = [0.40, 0.10, 0.50]
```

создает два массива, `x[]` и `y[]`, каждый из которых содержит по три значения типа `float`. Каждый массив — это объект, содержащий данные (точнее, ссылки на объекты), структурированные для эффективного доступа. Хотя истина немного сложней (и объясняется подробно в разделе 4.1), ссылки на элементы в массиве можно считать хранящимися рядом, последовательно одна за другой, в памяти компьютера, как показано на схеме для определенного ранее массива `suits[]`.



Структура данных массива

После создания массива любой его объект можно использовать в программе везде, где использовалось бы имя переменной. Для этого указывают имя массива, сопровождаемое целочисленным индексом в квадратных скобках. В приведенных примерах элемент `suits[1]` ссылается на значение `'Diamonds'`, `x[0]` — на значение `0.30`, `y[2]` — на `0.50` и т.д. Обратите внимание, что `x` — это ссылка на весь массив, а `x[i]` — ссылка на `i`-й элемент. В тексте мы используем форму `x[]` для указания, что переменная `x` является массивом (но мы не используем такую форму в коде Python).

Очевидное преимущество использования массива — это возможность избежать индивидуального обращения к каждой переменной. Использование индекса массива — это практически то же, что и добавление индекса к имени массива. Например, если необходимо обработать восемь значений типа `float`, то к каждому из них индивидуально можно обратиться по именами переменной `a0`, `a1`, `a2`, `a3`, `a4`, `a5`, `a6` и `a7`. Обозначение множества индивидуальных переменных таким образом было бы слишком громоздким, а если их миллионы, то и крайне затруднительно.

В качестве примера рассмотрим использование массива для представления *вектора* (vector). Более подробная информация о векторе приведена в разделе 3.3; а пока достаточно знать, что вектор — это последовательность вещественных чисел. *Скалярное произведение* (dot product) двух векторов (одинаковой длины) — это сумма произведений их соответствующих компонентов. Например, если два массива, `x[]` и `y[]`, из примеров представляют векторы, их скалярное

произведение дает выражение `x[0]*y[0] + x[1]*y[1] + x[2]*y[2]`. В общем случае, если имеются два одномерных массива, `x[]` и `y[]` типа `float` длиной `n`, то для их скалярного произведения можно использовать цикл `for`:

```
total = 0.0
for i in range(n)
    total += x[i]*y[i]
```

Простота кода таких вычислений делает использование массивов естественным выбором для всех видов приложений. Прежде чем перейти к дальнейшему изучению примеров, рассмотрим важнейшие характеристики программирования с массивами.

Трассировка вычисления скалярного произведения

i	X[i]	y[i]	x[i]*y[i]	total
				0.00
0	0.30	0.50	0.15	0.15
1	0.60	0.10	0.06	0.21
2	0.10	0.40	0.04	0.25

Индексация от нуля. Первый элемент массива `a[]` всегда `a[0]`, второй `a[1]` и т.д. Казалось бы, естественней именовать первый элемент как `a[1]`, второй как `a[2]` и так далее, однако начало индексации с 0 имеет немало преимуществ и в большинстве современных языков программирования является стандартным соглашением. Несоблюдение этого соглашения зачастую приводит к обнаруживаемым и устраняемым *ошибкам*, так что будьте внимательны!

Длина массива. Длину массива позволяет узнать встроенная функция Python `len()`: `len(a)` — это количество элементов массива `a[]`. Обратите внимание, что последний элемент массива `a[]` — это всегда `a[len(a)-1]`.

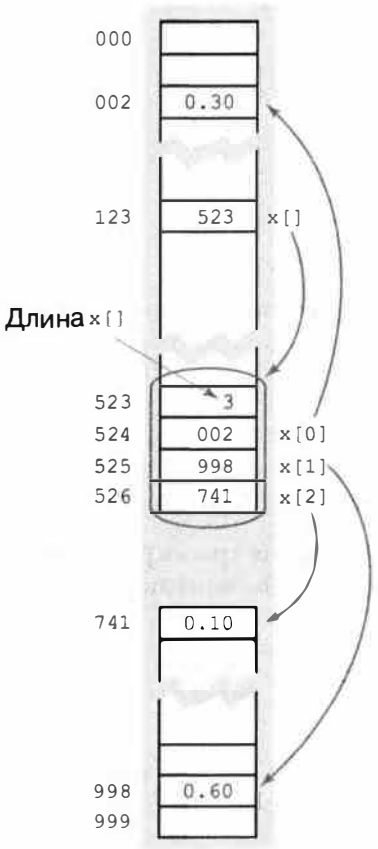
Увеличение длины массива во время выполнения. В языке Python для добавления элемента к массиву можно использовать оператор `+=`. Например, если массив `a[]` содержит `[1, 2, 3]`, то оператор `a += [4]` дополнит его до `[1, 2, 3, 4]`. Таким образом, можно создать массив из `n` значений типа `float`, каждый элемент которого инициализирован значением `0.0`.

```
a = []
for i in range(n)
    a += [0.0]
```

Оператор `a = []` создает пустой массив (длиной 0, без элементов), а оператор `a += [0.0]` добавляет один элемент в его конец. Обратите внимание, что для создания массива этим способом необходимо время, пропорциональное его длине (детали см. в разделе 4.1).

Представление в памяти. Массивы — это фундаментальные структуры данных, с прямым соответствием системам памяти практически всех компьютеров.

Ссылки на элементы массива хранятся в памяти последовательно, чтобы доступ к любому элементу массива был прост и эффективен. Действительно, саму память можно рассматривать как гигантский массив. Аппаратно память современных компьютеров реализована как последовательность индексированных ячеек памяти, к каждой из которых можно быстро обратиться по соответствующему индексу. Индекс ячейки машинной памяти обычно называют ее *адресом*. Имя массива (скажем, `x`) можно считать адресом непрерывного блока ячеек памяти, содержащих массив определенной длины, точнее, ссылок на его элементы. В иллюстративных целях предположим, что память компьютера ограничена 1 000 значений, с адресами от `000` до `999`. Теперь предположим, что массив `x[]` из трех элементов хранится в ячейках памяти с `523` по `526`, при длине, хранимой в ячейке `523`, и ссылках на элементы массива, хранящихся в ячейках `524 – 526`. При определении массива `x[i]` Python создает код, добавляющий индекс `i` к адресу в памяти первого элемента массива. В приведенном здесь примере код Python `x[2]` создал бы машинный код, находящий ссылку в ячейке памяти `524 + 2 = 526`. Тот же простой метод эффективен, даже когда память и массив (а также `i`) огромны. Доступ к ссылке на элемент `i` массива весьма эффективен, поскольку он требует суммирования двух целых чисел и обращения к памяти — всего две простые операции.

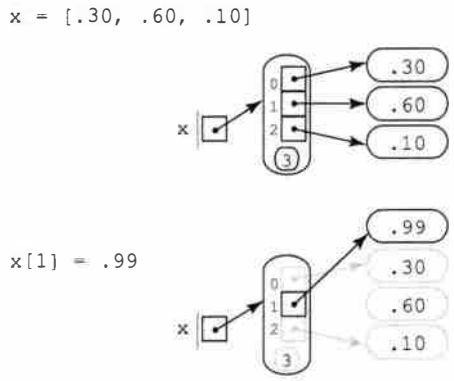


Идеализированное представление памяти `x = [0.30, 0.60, 0.10]`

Проверка границ (bounds checking). Как уже упоминалось, при программировании с массивами следует соблюдать осторожность. Ответственность за использование допустимых индексов при доступе к элементу массива лежит на программисте. Если вы создали массив размера `n` и используете индекс, значение которого превышает `n-1`, во время выполнения произойдет ошибка `IndexError`. (Во многих языках программирования система не проверяет условие *переполнения буфера*.) Подобные виды непроверяемых ошибок могут и приводят к кошмару при отладке, нередко они остаются незамеченными и в законченной программе. Как ни удивительно, но такой ошибкой может воспользоваться хакер, чтобы взять под свой контроль систему, даже ваш персональный компьютер, чтобы распространять вирусы,

похищать приватную информацию или наносить иной вред. Поначалу предоставляемые Python сообщения об ошибках раздражают, но это небольшая цена за шанс получить более надежную программу.

Изменяемость (mutability). Объект *изменяем* (mutable), если его значение может измениться. Массивы — изменяемые объекты, поскольку их элементы можно изменить. Например, если массив создает код `x = [.30, .60, .10]`, то оператор присвоения `x[1] = .99` изменит его на `[.30, .99, .10]`. Вот трассировка объектного уровня этой операции.



Переназначение элемента массива

Код перестраивающий элементы в массиве используется довольно часто . Например, следующий код меняет порядок элементов в массиве `a[]` на обратный:

```
n = len(a)
for i in range(n // 2):
    temp = a[i]
    a[i] = a[n-1-i]
    a[n-1-i] = temp
```

Три оператора в теле цикла `for` реализуют операцию обмена, описанную при первом знакомстве с операторами присвоения в разделе 1.2. Вы могли бы проверить свое понимание массивов по свободной трассировке этого кода для массива с семью элементами, `[3, 1, 4, 1, 5, 9, 2]`, представленной ниже. Эта таблица отслеживает значение переменной `i` и все семь элементов массива в конце каждой итерации цикла `for`.

Свободная трассировка смены порядка элементов массива

i	a[]						
	0	1	2	3	4	5	6
	3	1	4	1	5	9	2
0	2	1	4	1	5	9	3
1	2	9	4	1	5	1	3
2	2	9	5	3	4	1	3
	2	9	5	1	4	1	3

От массива вполне естественно ожидать изменяемости, но, как будет продемонстрировано, изменяемость — это ключевой вопрос в проекте типа данных и имеет много интересных последствий. Некоторые из них обсуждаются далее в этом разделе, а большая часть — в разделе 3.3.

Итерация (iteration), или перебор. Одна из множества фундаментальных операций со всеми элементами массива. Например, следующий код вычисляет среднее значение элементов массива типа `float`:

```
total = 0.0
for i in range(len(a))
    total += a[i]
average = total / len(a)
```

Язык Python позволяет также перебрать элементы массива, не обращаясь к индексам явно. Для этого поместите имя массива после ключевого слова `in` в операторе `for` следующим образом:

```
total = 0.0
for v in a:
    total += v
average = total / len(a)
```

Код последовательно присваивает значение каждого элемента массива управляющей переменной цикла `v`, поэтому данный код полностью эквивалентен присвоенному ранее. В этой книге мы выполняем итерации по индексам, когда необходимо обратиться к элементам массива (как в примерах скалярного произведения и смены порядка элементов массива). Мы выполняем перебор по элементам, когда имеет смысл только последовательность элементов, а не индексы (как в примере вычисления среднего значения).

Встроенные функции. В языке Python есть несколько встроенных функций, способных получать массивы как аргументы. Функцию `len()` мы уже рассмотрели. Если элементы массива `a[]` имеют числовой тип, то функция `sum(a)` вычисляет их сумму, что позволит вычислить их среднее значение как `float(sum(a))/len(a)` без всяких циклов. К другим подобным встроенным функциям относятся `min()`, вычисляющая минимум, и `max()`, вычисляющая максимум.

Вывод массива. Для вывода достаточно передать массив как аргумент функции `stdio.write()` или `stdio.writeln()`. Массив выводится в одну строку с предваряющей открывающей квадратной скобкой, разделяемыми запятыми и пробелами объектами массива, а затем закрывающей квадратной скобкой. Каждый объект в массиве преобразовывается в строку. Если этот формат не подходит для ваших потребностей, используйте оператор `for` для вывода каждого элемента массива индивидуально.

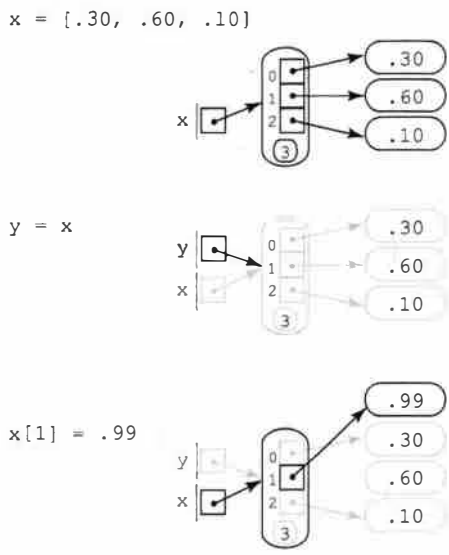
Псевдонимы и копии массива. Прежде чем переходить к использующим массивы программам, стоит подробнее исследовать две фундаментальные операции с массивами. Рассматриваемые моменты крайне важны — куда важней, чем может показаться на первый взгляд. Если рассматривать технические подробности сейчас затруднительно, вы могли бы вернуться к этому позже, после изучения примеров, использующих массивы приложений, далее в этом разделе. Повторение этого материала закрепит ваше понимание этих концепций.

Псевдоним (alias). В коде можно использовать переменные, ссылающиеся на массивы, точно так же, как и на объекты других типов. Важно уделить время пониманию этого. Если поразмыслить над ситуацией, то, возможно, первым возникнет такой вопрос: что будет при использовании имени переменной типа массива в левой части оператора присвоения? Другими словами, если `x[]` и `y[]` являются массивами, то каков результат оператора `x = y`? Ответ прост

и совместим со способом применения в языке Python других типов данных: x *и* y *ссылаются на тот же массив*. Возможно, поначалу этот результат покажется неожиданным, поскольку вполне естественно считать x и y ссылками на два независимых массива, что вовсе не так. Например, *после операторов присвоения*

```
x = [.30, .60, .10]
y = x
x[1] = .99
```

элемент y[1] *также содержит значение* .99, *хотя в коде элемент* y[1] *даже не упоминался*. Эта ситуация (когда две переменные ссылаются на тот же объект) известна как *применение псевдонимов* (aliasing) и иллюстрируется трассировкой объектного уровня ниже. Мы избегаем применения псевдонимов массивов (и других изменяемых объектов), поскольку это затрудняет поиск ошибок в программах. Но, как будет продемонстрировано в главе 2, в некоторых ситуациях применение двух псевдонимов на массив весьма полезно, поэтому имеет смысл ознакомиться с этой концепцией заранее.



Псевдонимы массивов

Копирование и разделение. Следующий вполне естественный вопрос: как сделать копию y[] массива x[]? Один из ответов на этот вопрос — в переборе массива x[] для создания массива y[], как в следующем коде:

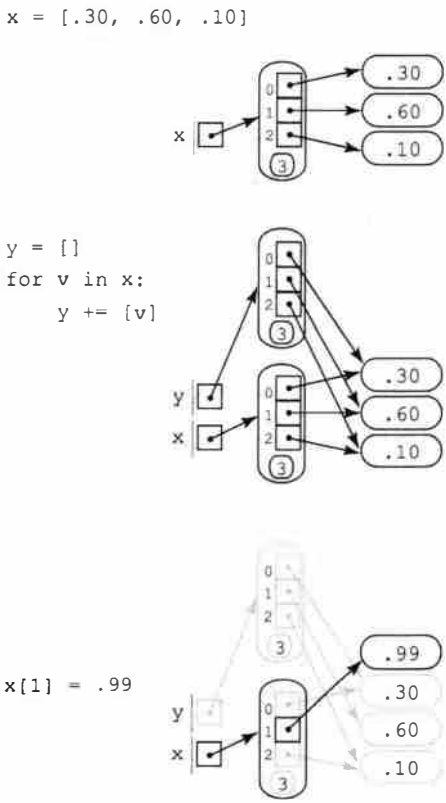
```
y = []
for v in x:
    y += [v]
```

После копирования переменные x и y ссылаются на два разных массива. Если изменить объект, на который ссылается элемент x[1], то изменения никак не затронут объект, на который ссылается элемент y[1]. Эту ситуацию иллюстрирует следующая трассировка объектного уровня.

Фактически копирование массива является настолько популярной операцией, что Python предоставляет поддержку для более общей операции *разделения* (slicing), позволяющей скопировать любую непрерывную последовательность элементов в пределах массива в другой массив. Выражение a[i:j] создает новый массив с элементами a[i], ..., a[j-1]. Кроме того, стандартным значением i является 0, а стандартным значением j — len(a). Таким образом, выражение y = x[:] эквивалентно коду, приведенному ранее. Эта компактная запись

удобна, но следует помнить, что она маскирует потенциально длительную операцию, время выполнения которой пропорционально длине массива `x[]`.

Системная поддержка массивов. Обрабатывающий массивы код Python может принять множество форм. Для полноты описания мы кратко рассмотрим каждую форму, но акцент этой книги, по большей части, смещен на несколько операций, необходимых для написания эффективного кода. После приобретения некоторого опыта в чтении и создании кода, использующего эти операции, вы сможете лучше понять различия между подходами обработки массивов, доступными в языке Python, поэтому мы вернемся к этой теме в главе 4.



Копирование массива

Встроенный тип данных Python `list`. В большинстве канонических форм массивы поддерживают четыре базовых операции: создание, доступ по индексу, присвоение по индексу и перебор. В этой книге мы используем встроенный тип данных Python для массивов `list`, поскольку он поддерживает эти базовые операции. Мы рассмотрим более сложные операции, поддерживаемые типом данных `list`, в главе 4, включая операции изменения размера массива. Массивы, как будет описано здесь, — это весьма мощное средство, позволяющее узнать много интересного о подходах программирования. Обычно программисты Python не делают различий между списками Python и массивами. Здесь мы подчеркиваем это различие, поскольку во многих других языках программирования (таких, как Java и C) есть встроенная поддержка для *массивов фиксированной длины* (fixed-length array) и четырех базовых операций с ними (но не для более сложных массивов, изменяющих размеры и операции с ними).

Модуль Python `numpy`. Суть проекта на любом языке программирования — это компромисс между простотой и эффективностью. Даже при том, что компьютеры, казалось бы, могут выполнять огромное количество элементарных операций в секунду, все знают, что они иногда "тормозят". В случае языка Python у его встроенного типа данных `list` могут быть серьезные проблемы производительности, даже в простых программах, использующих массивы для решения реальных проблем.

Поэтому ученые и инженеры зачастую используют модуль расширения Python numpy для обработки огромных массивов чисел, ведь этот модуль использует низкоуровневые представления, позволяющие избежать большинства неэффективных операций в стандартном представлении Python. Но оставим вопросы эффективности до главы 4, когда вы будете лучше понимать такие ситуации. В этом контексте мы описываем использование модуля numpy на сайте книги.

Наш собственный модуль stdarray. Ранее в этой главе упоминался наш модуль stdio, определяющий функции, связанные с выводом и вводом данных. Функции stdio.write() и stdio.writeln() уже использовались не раз. Модуль stdio — это *книжный модуль*, т.е. нестандартный модуль, который мы создали специально для этой книги и ее сайта. Теперь пришло время ознакомиться с другим книжным модулем: stdarray. В нем определены функции для обработки массивов, используемые нами повсюду в книге.

Практически в каждой обрабатывающей массивы программе должна быть такая операция, как *создание массива из n элементов, каждый из которых инициализирован предоставленным значением.* Как уже упоминалось, для этого в Python можно использовать следующий код:

```
a = []
for i in range(n):
    a += [0.0]
```

Создание массива заданной длины и инициализация всех его элементов заданным значением настолько распространены, что в языке Python для этого есть специальная сокращенная форма записи: a = [0.0]*n, эквивалентная коду, приведенному выше. Вместо нее в коде книги мы будем использовать такую форму:

```
a = stdarray.create1D(n, 0.0)
```

Модуль stdarray включает также функцию create2D(), рассматриваемую далее в этом разделе. Мы используем эти функции потому, что они "автоматически документируют" наш код, в том смысле, что код сам точно выражает наши намерения и не зависит от идиом, специфичных для языка Python. Более подробная информация о проблемах, связанных с библиотечным проектом (а также о создании собственных модулей), приведена в разделе 2.2, поскольку к тому времени вы будете лучше разбираться в таких ситуациях.

API для функций книжного модуля stdarray, связанных с созданием массивов

Вызов функции	Описание
stdarray.create1D(n, val)	Создает массив длиной n, каждый элемент которого инициализирован значением val
stdarray.create2D(m, n, val)	Создает массив размером m × n, каждый элемент которого инициализирован значением val

Теперь вы знаете, как создавать массивы в Python и обращаться к его отдельным элементам. Вам также известно о встроенном типе данных Python `list`. Операции над массивом приведены в таблице ниже. Овладев основами концепций, можно переходить к приложениям. Как будет продемонстрировано ниже, этих основ достаточно для создания кода, который не только прост и понятен, но и обеспечивает эффективное использование вычислительных ресурсов.

Предупреждение о сокращении. Повсюду в этой книге мы называем тип данных Python `list` *массивом,* поскольку списки Python поддерживают характерные для массивов фундаментальные операции создания, доступа, присвоения и перебора по индексу.

Операции над массивом и встроенные функции

Операция	Оператор	Описание
Доступ по индексу	`a[i]`	`i`-й элемент в массиве `a[]`
Присвоение по индексу	`a[i] = x`	Присвоение `i`-му элементу массива `a[]` значения `x`
Перебор (итерация)	`for v in a:`	Присвоение переменной `v` значений каждого из элементов в массиве `a[]`
Разделение	`a[i:j]`	Новый массив `[a[i], a[i+1],..., a[j-1]]` (стандартно `i` имеет значение 0, а `j` — `len(a)`)
Операция	Вызов функции	Описание
Длина	`len(a)`	Количество элементов в массиве `a[]`
Сумма	`sum(a)`	Сумма элементов в массиве `a[]`
Минимум	`min(a)`	Минимальный элемент в массиве `a[]`
Максимум	`max(a)`	Максимальный элемент в массиве `a[]`

Примечание. Для функции `sum()` элементы массива должны быть числовыми, а для функций `min()` и `max()` сопоставимыми.

Типичные приложения с использованием массивов. Далее мы рассмотрим множество приложений, иллюстрирующих возможности массивов. Кроме того, они интересны сами по себе.

Игра в карты. Предположим, мы хотим составить программу карточной игры. Можно начать со следующего кода:

```
SUITS = ['Clubs', 'Diamonds', 'Hearts', 'Spades']
RANKS = ['2', '3', '4', '5', '6', '7', '8', '9', '10',
    'Jack', 'Queen', 'King', 'Ace']
```

Например, эти два массива можно было бы использовать для вывода названия случайной карты, например `Queen of Clubs`, следующим образом:

```
rank = random.randrange(0, len(RANKS))
suit = random.randrange(0, len(SUITS))
stdio.writeln(RANKS[rank] + ' of ' + SUITS[suit])
```

Более типична ситуация, когда приходится вычислять хранимые в массиве значения. Например, можно было бы использовать следующий код для инициализации массива длиной 52, представляющего колоду (deck) карт, используя два только что определенных массива:

```
deck = []
for rank in RANKS:
    for suit in SUITS:
        card = rank + ' of ' + suit
        deck += [card]
```

После выполнения этого кода вывод элементов массива `deck[]`, от `deck[0]` до `deck[51]` по одному элементу в строку даст следующую последовательность:

```
2 of Clubs
2 of Diamonds
2 of Hearts
2 of Spades
3 of Clubs
3 of Diamonds
...
Ace of Hearts
Ace of Spades
```

Обмен. Зачастую два элемента в массиве необходимо обменять местами. Продолжая пример с колодой карт, следующий код меняет местами карты с индексами `i` и `j`, используя те же идиомы, что и ранее в этом разделе при изменении порядка элементов массива на обратный:

```
temp = deck[i]
deck[i] = deck[j]
deck[j] = temp
```

При использовании этого кода изменяется *порядок* элементов в массиве, но не сам *набор* элементов в массиве. Когда `i` и `j` равны, массив не изменяется. Когда `i` и `j` не равны, значения элементов `a[i]` и `a[j]` находятся в разных местах массива. Например, если использовать этот код с `i`, равным `1`, и `j`, равным `4`, в массиве `deck[]` предыдущего примера, получится строка `'3 of Clubs'` в элементе `deck[1]` и строка `'2 of Diamonds'` в элементе `deck[4]`.

Перестановка. Следующий код перетасовывает колоду карт:

```
n = len(deck)
for i in range(n):
    r = random.randrange(i, n)
    temp = deck[r]
    deck[r] = deck[i]
    deck[i] = temp
```

Код последовательно выбирает случайную карту из диапазона от `deck[i]` до `deck[n-1]` (каждая карта одинаково вероятна) и обменивает ее с `deck[i]`. Этот код сложней, чем может показаться. Во-первых, используя идиому обмена, мы гарантируем, что после перетасовки карты в колоде будут теми же, что и до перетасовки. Во-вторых, мы гарантируем случайность перестановок за счет равномерного выбора карт из еще не выбранных. Для обеспечения случайности Python предоставляет в модуле `random` стандартную функцию `shuffle()`, равномерно перетасовывающую заданный массив. Таким образом, вызов функции `random.shuffle(deck)` выполняет ту же задачу, что и этот код.

Выборка без замены. Не так уж и редко приходится осуществлять случайную выборку из набора таким образом, чтобы каждый выбранный элемент в наборе присутствовал максимум только однажды. Извлечение нумерованных шариков из корзины во время лотереи — это пример подобного рода выборки, как и извлечение одной карты из колоды. Программа 1.4.1 (`sample.py`) иллюстрирует подобную выборку, используя простую операцию на базе перетасовки. Она получает аргументы командной строки `m` и `n`, а затем осуществляет *перестановку* (permutation) размера `n` (перестановка целых чисел от `0` до `n-1`), чьи первые `m` элементов составляют случайную выборку.

Этот процесс иллюстрирует последующая трассировка содержимого массива `perm[]` в конце каждой итерации главного цикла (для случая со значениями `m` и `n` 6 и 16 соответственно). Если значения `r` выбираются таким образом, что каждое значение в данном диапазоне одинаково вероятно, то выборка от `perm[0]` до `perm[m-1]` в конце процесса будет случайной (даже при том, что некоторые элементы могли бы перемещаться многократно), поскольку каждый элемент в выборке выбирается из еще не выбранных элементов, при равной вероятности быть выбранной.

Программа 1.4.1. Выборка без замены (`sample.py`)

```
import random
import sys
import stdarray
import stdio

m = int(sys.argv[1]) # Выбрать столько элементов
n = int(sys.argv[2]) # из 0, 1, ..., n-1.

# Инициализация массива perm = [0, 1, ..., n-1].
perm = stdarray.create1D(n, 0)
for i in range(n):
    perm[i] = i

# Случайная выборка размером m в perm[0..m).
for i in range(m):
    r = random.randrange(i, n)

    # Обмен perm[i] и perm[r].
    temp    = perm[r]
    perm[r] = perm[i]
    perm[i] = temp

# Вывод результатов.
for i in range(m):
    stdio.write(str(perm[i]) + ' ')
stdio.writeln()
```

`m`	Размер выборки
`n`	Диапазон
`perm[ ]`	Перестановка от 0 до n-1

Эта программа принимает в аргументах командной строки целые числа `m` и `n`, а выводит случайную выборку целых чисел размером `m` в диапазоне от `0` до `n-1` (без повторений). Этот процесс полезен не только в лотереях, но и в различных научных приложениях. Если первый аргумент меньше или равен второму, результат — случайная последовательность целых чисел от `0` до `n-1`. Если первый аргумент больше второго, то во время выполнения произойдет ошибка `ValueError`.

```
% python sample.py 6 16
9 5 13 1 11 8

% python sample.py 10 1000
656 488 298 534 811 97 813 156 424 109

% python sample.py 20 20
6 12 9 8 13 19 0 2 4 5 18 1 14 16 17 3 7 11 10 15
```

Трассировка `python sample.py 6 16`

i	r	perm[]															
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15		
0	9	9	1	2	3	4	5	6	7	8	0	10	11	12	13	14	15
1	5	9	5	2	3	4	1	6	7	8	0	10	11	12	13	14	15
2	13	9	5	13	3	4	1	6	7	8	0	10	11	12	2	14	15
3	5	9	5	13	1	4	3	6	7	8	0	10	11	12	2	14	15
4	11	9	5	13	1	11	3	6	7	8	0	10	4	12	2	14	15
5	8	9	5	13	1	11	8	6	7	3	0	10	4	12	2	14	15
		9	5	13	1	11	8	6	7	3	0	10	4	12	2	14	15

Важнейшая причина явного вычисления перестановки в том, что ее можно использовать для создания случайной выборки *любого* массива при использовании элементов перестановки как индексов массива. Зачастую это весьма привлекательная альтернатива фактической перестройке массива, поскольку при этом он может оставаться упорядоченным по некой другой причине (например, компания могла бы получить случайную выборку из списка клиентов, который должен храниться в алфавитном порядке). Для демонстрации этого трюка в действии предположим, что необходима случайная сдача в покер из массива `deck[]`, создание которого описано выше. Используем код программы `sample.py` при `m = 5` и `n = 52`, а массив `perm[i]` заменим на `deck[perm[i]]` в операторе `stdio.write()` (а его заменим на `writeln()`). В результате получим следующий вывод:

```
3 of Clubs
Jack of Hearts
6 of Spades
Ace of Clubs
10 of Diamonds
```

Выборка широко используется в статистических исследованиях опросов, научных исследованиях и во многих других случаях, когда необходимо сделать выводы о большой совокупности при анализе малой случайной выборки. Для осуществления выборки язык Python предоставляет в стандартном модуле `random` функцию `random.sample(a, k)`, получающую массив `a[]` и целое число `k`, а возвращающую новый массив, содержащий однородно случайную выборку размера `k` из элементов массива `a[]`.

Предварительное вычисление значений. Еще одно применение массивов — хранение ранее вычисленных значений для более позднего использования. Предположим, например, что нужна программа, осуществляющая вычисления с использованием нескольких первых значений гармонических чисел (см. программу 1.3.5). Эти значения имеет смысл хранить в массиве следующим образом:

```
harmonic = stdarray.create1D(n+1, 0.0)
for i in range(1, n+1):
    harmonic[i] = harmonic[i-1] + 1.0/i
```

Обратите внимание, что один слот в массиве (элемент 0) мы тратим впустую для обеспечения соответствия элемента `harmonic[1]` первому гармоническому числу `1.0`, а элемента `harmonic[i]` — `i`-му гармоническому числу. Таким образом, предварительное вычисление значений является примером *обмена пространства на время*: потратив пространство (на хранение значений), мы экономим время (поскольку их не нужно вычислять повторно). Для огромных `n` данный метод не эффективен, но если необходимо относительно немного значений, то это совершенно другое дело.

Упрощение повторяющегося кода. В качестве другого простого примера применения массивов рассмотрим следующий фрагмент кода, выводящий названия месяцев по их номеру (январь для 1, февраль для 2 и т.д.):

```
if   m == 1: stdio.writeln('Jan')
elif m == 2: stdio.writeln('Feb')
elif m == 3: stdio.writeln('Mar')
elif m == 4: stdio.writeln('Apr')
...
elif m == 11: stdio.writeln('Nov')
elif m == 12: stdio.writeln('Dec')
```

Более компактная альтернатива подразумевает использование массива строк, содержащих названия месяцев:

```
MONTHS = ['', 'Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun',
              'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec']
...
stdio.writeln(MONTHS[m])
```

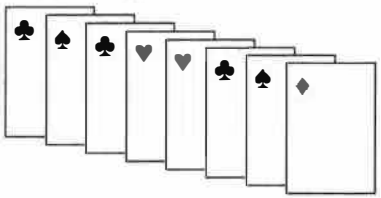
Эта методика была бы особенно полезна, если бы доступ к названию месяца по номеру нужно было получать в нескольких разных местах программы. Обратите внимание: здесь мы снова преднамеренно тратим впустую один слот массива (элемент 0) для обеспечения соответствия элемента `MONTHS[1]` январю, как и нужно.

После этих определений и простых примеров можно перейти к рассмотрению двух приложений и решить интересные классические задачи, проиллюстрировав фундаментальную важность массивов для эффективности вычислений. В обоих случаях идея использования данных для эффективной индексации массивов играет центральную роль и обеспечивает возможность вычислений, невыполнимых в противном случае.

Коллекционер купонов. Предположим, есть колода карт и вы выбираете карты наугад, одну за другой. Сколько карт необходимо открыть, прежде чем вы увидите по одной карте каждой масти? Сколько карт необходимо открыть,

прежде чем вы увидите по одной каждого ранга? Это примеры известной задачи *коллекционера купонов* (coupon collector problem). В общем случае, это задача о торговой компании, выдающей n различных торговых купонов: сколько их следует собрать, прежде чем у вас будут все n возможных купонов, с учетом одинаковой вероятности получить каждую карточку?

Программа 1.4.2 (couponcollector.py) моделирует этот процесс и иллюстрирует удобство массивов. Она получает из командной строки целое число n и создает последовательность случайных целочисленных значений от 0 до n-1 с использованием функции random.randrange(0, n) (см. программу 1.3.8). Каждое значение представляет карточку; для каждой карточки мы хотим знать, не встречалась ли она прежде. Для обеспечения этого знания мы используем массив isCollected[], индексом которого является значение карточки: элемент isCollected[i] содержит значение True, если встретилась карточка со значением i, и значение False в противном случае. Когда мы получаем новую карточку, представленную целым числом value, мы проверяем ее на предмет повтора по соответствующему значению isCollected[value]. Расчет подразумевает сохранение количества отдельных встретившихся значений, количества полученных карточек и вывод последней по достижении n.



Коллекция карточек

Трассировка типичного запуска `python couponcollector.py 6`

val	isCollected[]						count	collectedCount
	0	1	2	3	4	5		
	F	F	F	F	F	F	0	0
2	F	F	T	F	F	F	1	1
0	T	F	T	F	F	F	2	2
4	T	F	T	F	T	F	3	3
0	T	F	T	F	T	F	3	4
1	T	T	T	F	T	F	4	5
2	T	T	T	F	T	F	4	6
5	T	T	T	F	T	T	5	7
0	T	T	T	F	T	T	5	8
1	T	T	T	F	T	T	5	9
3	T	T	T	T	T	T	6	10

Как обычно, понять работу программы лучше всего по трассировке значений ее переменных для типичного запуска. Совсем не сложно добавить в программу couponcollector.py код трассировки значений в конце цикла while. В таблице для краткости мы используем сокращения: F — для False и T — для True.

Трассировка программ, использующих большие массивы, может быть затруднительна: когда в программе есть массив длиной *n*, он представляет *n* переменных и их придется выводить все. Трассировка программ, использующих функцию `random.randrange()`, также может быть затруднительна, поскольку при каждом запуске программы трассировка получается разной. Таким образом, отношения между переменными следует тщательно проверять. В данном случае обратите внимание, что значение `collectedCount` всегда равно количеству значений `True` в массиве `isCollected[]`.

Программа 1.4.2. Модель коллекции купонов (`couponcollector.py`)

```
import random
import sys
import stdarray
import stdio

n = int(sys.argv[1])

count = 0
collectedCount = 0
isCollected = stdarray.create1D(n, False)

while collectedCount < n:
    # Создать следующий купон.
    value = random.randrange(0, n)
    count += 1
    if not isCollected[value]:
        collectedCount += 1
        isCollected[value] = True

stdio.writeln(count)
```

`n`	Количество типов купонов (от `0` до `n-1`)
`count`	Количество собранных купонов
`collectedCount`	Количество различных собранных купонов
`isCollected[i]`	Есть ли уже купон `i`?
`value`	Значение текущего купона

Эта программа получает в аргументах командной строки целое число `n` и выводит количество купонов, собранных до получения всех купонов каждого из `n` типов. Таким образом, она моделирует коллекцию купонов.

```
% python couponcollector.py 1000
6583
% python couponcollector.py 1000
6477
% python couponcollector.py 1000000
12782673
```

Сбор купонов — задача не тривиальная. Например, ученым зачастую необходимо знать, не имеет ли некая последовательность те же характеристики, что

и случайная последовательность. Если это так, то данный факт может представлять интерес; в противном случае гарантировано дальнейшее исследование по поиску схем и закономерностей. Подобные проверки ученые используют для решения, какие части генома стоит учитывать. Одной из эффективных проверок последовательности на случайность является *проверка коллекционера купонов*: соотношение количества элементов, которые должны быть исследованы прежде, чем будут найдены все возможные, к соответствующему значению однородно случайной последовательности.

Модель процесса коллекционирования купонов для больших значений n без массивов практически невозможна, а с массивами довольно проста. В этой книге встречается довольно много подобных примеров.

Решето Эратосфена. Простые числа играют важную роль в математике и многих вычислениях, включая криптографию. *Простое число* — это целое число (больше 1), делящееся только на 1 и на себя. Функция расчета простых чисел, $\pi(n)$, дает количество множителей, меньших или равных n. Например, $\pi(25) = 9$, потому что первые девять множителей 2, 3, 5, 7, 11, 13, 17, 19 и 23. Эта функция играет ключевую роль в теории чисел.

Такую программу, как `factors.py` (программа 1.3.9), можно было бы использовать для подсчета множителей. А именно: код программы `factors.py` можно модифицировать так, чтобы устанавливать логическую переменную в состояние `True`, если данное число является простым, и в состояние `False` в противном случае (вместо вывода множителей). Этот код следует заключить в цикл, увеличивающий счетчик для каждого простого числа. Данный подход эффективен для малых n, но при росте n он становится слишком медленным.

Альтернативой является программа 1.4.3 (`primesieve.py`), вычисляющая функцию $\pi(n)$, используя алгоритм *решета Эратосфена* (sieve of Eratosthenes). Для записи простых чисел, меньших или равных `n`, программа использует массив `isPrime[]` из `n` логических переменных. Задача в том, чтобы установить элемент `isPrime[i]` в состояние `True`, если `i` является простым числом, и в состояние `False` в противном случае. Решето работает следующим образом: первоначально всем элементам массива присвоено значение `True`, означающее, что никаких множителей у всех целых чисел еще не найдено. Затем повторяется приведенная ниже последовательность действий, пока `i < n`.

- Поиск следующего наименьшего `i`, для которого не было найдено никаких множителей.
- Оставить в `isPrime[i]` значение `True`, поскольку никаких меньших множителей у `i` нет.
- Присвоить значение `False` всем элементам массива `isPrime[]`, индексы которых кратны `i`.

Программа 1.4.3. Решето Эратосфена (primesieve.py)

```
import sys
import stdarray
import stdio

n = int(sys.argv[1])

isPrime = stdarray.create1D(n+1, True)

for i in range(2, n):
    if (isPrime[i]):
        # Отметить i как не простое.
        for j in range(2, n//i + 1):
            isPrime[i*j] = False

# Подсчет простых чисел.
count = 0
for i in range(2, n+1):
    if (isPrime[i]):
        count += 1
stdio.writeln(count)
```

N	Аргумент
isPrime[i]	Является ли i простым?
Count	Счетчик простых чисел

Эта программа получает n в аргументе командной строки и вычисляет количество множителей, меньших или равных n. Для этого она рассчитывает массив логических переменных isPrime[i] и устанавливает значения True, если i является простым, и False в противном случае.

```
% python primesieve.py 25
9
% python primesieve.py 100
25
% python primesieve.py 10000
1229
% python primesieve.py 1000000
78498
% python primesieve.py 100000000
5761455
```

Трассировка `python primesieve.py 25`

i	isPrime[]																							
	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
2	**T**	**T**	F	**T**	F	**T**	F	**T**	F	**T**	F	**T**	F	**T**	F	**T**	F	**T**	F	**T**	F	**T**	F	**T**
3	**T**	**T**	**F**	**T**	**F**	**T**	**F**	F	**F**	**T**	F	**T**	**F**	F	**F**	**T**	F	**T**	**F**	F	**F**	**T**	F	**T**
5	**T**	**T**	**F**	**T**	**F**	**T**	**F**	**F**	**F**	**T**	**F**	**T**	**F**	**F**	**F**	**T**	**F**	**T**	**F**	**F**	**F**	**T**	**F**	F
	T	T	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	F

Когда вложенный цикл `for` завершается, все элементы массива `isPrime[]` для всех не простых чисел установлены в состояние `False`, а для всех простых останутся в состоянии `True`. Обратите внимание: мы могли бы остановиться, когда `i*i> = n`, как в программе `factors.py`, но экономия в данном случае будет в лучшем случае бесполезна, поскольку внутренний цикл `for` не выполняет итераций вообще для большого `i`. Еще один перебор массива позволяет посчитать количество простых чисел, меньшее или равное `n`.

Как обычно, добавим код трассировки. С такими программами, как `primesieve.py`, содержащими вложенную конструкцию `for-if-for`, следует соблюдать осторожность и обращать внимание на отступ, чтобы поместить код трассировки в правильное место.

Программа `primesieve.py` позволяет вычислять $\pi(n)$ при больших n, ограничение накладывает прежде всего максимально возможный размер массива в языке Python. Это еще один пример обмена пространства на время. Такие программы, как `primesieve.py`, играют важную роль, помогая математикам в разработке теории чисел, имеющей множество важных применений.

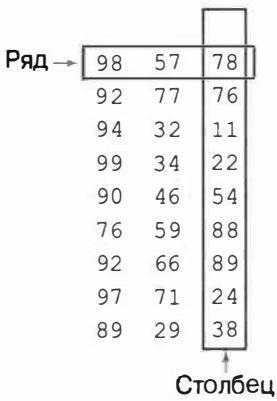


Таблица чисел

Двумерные массивы. Во многих приложениях информацию удобней хранить в прямоугольной *таблице* и обращаться к ней по *рядам* и *столбцам*. Например, учителю может понадобиться таблица, ряды которой соответствуют каждому ученику, а столбцы — каждой контрольной; ученому таблица может понадобиться для хранения экспериментальных данных, где ряды соответствуют экспериментам, а столбцы — различным результатам; программист мог бы использовать таблицу при подготовке изображения к выводу на экран, заполняя ее ячейки значениями различных цветов пикселей.

Таблицам соответствует такая математическая абстракция, как *матрица* (matrix), и такая структура данных, как *двумерный массив* (two-dimensional array). Вы, вероятно, уже встречались со многими случаями применения матриц и двумерных массивов в науке, технике и коммерческой деятельности, и, уж конечно, встретитесь с ними в многочисленных примерах этой книги. Подобно векторам и одномерным массивам, большинство важнейших случаев применения двумерных массивов подразумевает обработку больших объемов данных, поэтому отложим их рассмотрение до изучения ввода и вывода в разделе 1.5.

Расширение уже обсуждавшейся одномерной структуры данных, массива, в двумерный массив является довольно простым: поскольку вы понимаете, что массив может содержать любой тип данных, его элементами вполне могут быть и массивы! Таким образом, двумерный массив реализуется как массив одномерных массивов.

Инициализация (initialization). Проще всего создать двумерный массив в Python — это заключить разделяемые запятыми одномерные массивы в квадратные скобки. Например, следующая матрица целых чисел, имеющая два ряда и три столбца

```
18 19 20
21 22 23
```

может быть представлена в Python как следующий массив массивов:

```
a = [[18, 19, 20], [21, 22, 23]]
```

Это называется массивом 2×3. В соответствии с соглашением, первая размерность — это количество рядов, вторая — количество столбцов. В языке Python массив 2×3 представляется как массив, содержащий два объекта, каждый из которых в свою очередь является массивом, содержащим три объекта.

```
a = [[99, 85, 98],
     [98, 57, 78],
     [92, 77, 76],
     [94, 32, 11],
     [99, 34, 22],
     [90, 46, 54],
     [76, 59, 88],
     [92, 66, 89],
     [97, 71, 24],
     [89, 29, 38]]
```

Создание массива 10 на 3

Более формально, язык Python представляет массив m на n элементов как массив, содержащий m объектов, каждый из которых является массивом, содержащим n объектов. Например, следующий код Python создает массив `a[][]` размером `m` на `n` для чисел с плавающей точкой, все элементы которого инициализированы значением `0.0`:

```
a = []
for i in range(m):
    row = [0.0] * n
    a += [row]
```

Как и для одномерных массивов, можно использовать и самодокументированную альтернативу

```
stdarray.create2D(m, n, 0.0)
```

из нашего книжного модуля `stdarray`.

Индексация. Когда массив `a[][]` является двумерным, синтаксическая конструкция `a[i]` означает ссылку на его `i`-й ряд. Например, если `a[][]` является массивом `[[18, 19, 20], [21, 22, 23]]`, то `a[1]` — это массив `[21, 22, 23]`, хотя обычно обращаются к конкретному элементу в двумерном массиве. Для этого используется синтаксическая конструкция `a[i][j]`, обеспечивающая доступ к объекту в ряде `i` и столбце `j` двумерного массива `a[][]`. В данном примере элемент `a[1][0]` содержит значение `21`. Для доступа к каждому элементу в двумерном массиве используются два вложенных цикла `for`. Например, следующий код выводит все объекты массива `a[][]` размером `m` на `n`, выстроив каждый ряд в одну строку:

```
for i in range(m):
    for j in range(n):
```

```
        stdio.write(a[i][j])
        stdio.write(' ')
    stdio.writeln()
```

Этот код достигает того же эффекта, но без использования индексов:

```
for row in a:
    for v in row:
        stdio.write(v)
        stdio.write(' ')
    stdio.writeln()
```

Таблицы. Один из общеизвестных случаев использования массивов — это *таблица* (spreadsheet) чисел. Например, имеющий m учеников учитель для результатов n контрольных мог бы составить массив размером $(m + 1)$ на $(n + 1)$, резервируя последний столбец для средней оценки каждого ученика и последний ряд для средней оценки каждой контрольной. Хотя обычно такие вычисления делают в специализированных приложениях, лежащий в их основе код имеет смысл изучить как введение в обработку массивов. Для вычисления средней оценки по каждому ученику (среднее значение каждого ряда) суммируем элементы каждого ряда и делим на n. Порядок обработки матрицы ряд за рядом элементов известен как *порядный* (row-major). Точно так же при вычислении средней оценки контрольной (среднее значение для каждого столбца) суммируются элементы для каждого столбца и делятся на n. Порядок обработки столбец за столбцом известен как *постолбцовый* (column-major). Эти операции представлены на рисунке. Для учета половины баллов учитель решил использовать для оценок тип `float`.

```
for i in range(m):
    # Среднее для ряда i
    total = 0.0
    for j in range(n):
        total += a[i][j]
    a[i][n] = total / m
```

Вычисление среднего в ряду (порядный порядок)

```
for j in range(n):
    # Среднее для столбца j
    total = 0.0
    for i in range(m):
        total += a[i][j]
    a[m][j] = total / n
```

Вычисление среднего в столбце (постолбцовый порядок)

Типичные вычисления в таблицах

Операции с матрицами. Применяемые в науке и технике матрицы представляют как двумерные массивы, а затем реализуют различные математические операции с матричными операндами. Хотя такая обработка также обычно осуществляется в специализированных приложениях и библиотеках, лежащие в их основе вычисления имеет смысл изучить.

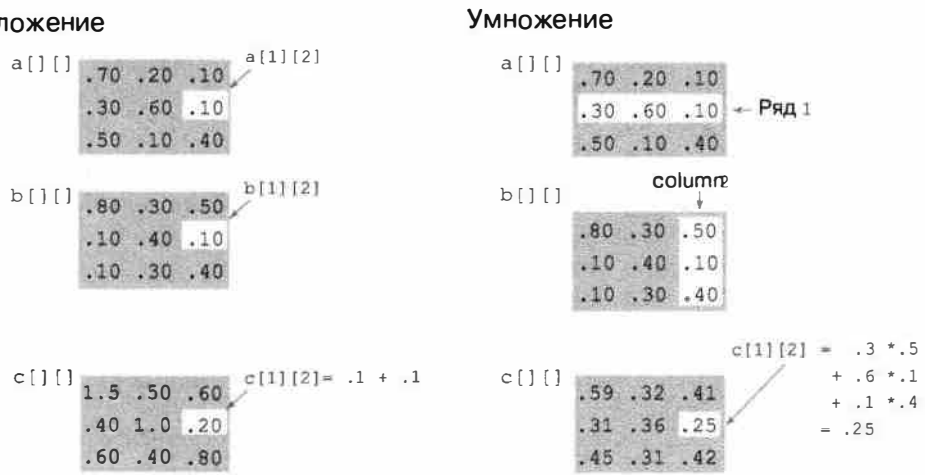
Например, можно *сложить* две матрицы, `a[][]` и `b[][]`, размером `n` на `n` следующим образом:

```
c = stdarray.create2D(n, n, 0.0)
for i in range(n):
    for j in range(n):
        c[i][j] = a[i][j] + b[i][j]
```

Точно так же две матрицы можно *умножить*. Вам, возможно, известно матричное умножение, но если нет, то следующий код Python для квадратных матриц, по существу, является его математическим определением. Каждый элемент массива `c[i][j]` является произведением массивов `a[][]` и `b[][]`, вычисляемым в результате скалярного произведения ряда `i` массива `a[][]` со столбцом `j` массива `b[][]`:

```
c = stdarray.create2D(n, n, 0.0)
for i in range(n):
    for j in range(n):
        # Вычисление скалярного произведения ряда i и столбца j
        for k in range(n):
            c[i][j] += a[i][k] * b[k][j]
```

Определение распространяется на матрицы, которые могут и не быть квадратными (см. упр. 1.4.17).



Типичные операции с матрицами

Частные случаи матричного умножения. Важны два частных случая матричного умножения. Когда одна из размерностей одной из матриц равна 1, ее можно рассматривать как вектор. При *матрично-векторном умножении* (matrix–vector multiplication) матрица m на n умножается на *вектор столбцов* (матрица n на 1) и в результате получается вектор столбцов m на 1 (каждый элемент — результат скалярного произведения соответствующего ряда матрицы с векторным операндом). При *векторно-матричном умножении* умножается *вектор рядов* (матрица 1 на m) на матрицу m на n, а в результате получается вектор рядов 1 на n (каждый элемент — скалярное произведение векторного операнда на соответствующий столбец в матрице).

Частные случаи матричных операций (когда один из аргументов — вектор)

Эти операции обеспечивают сокращенный способ выражения многочисленных матричных вычислений. Например, вычисление среднего рядов в такой таблице из `m` рядов и `n` столбцов эквивалентно матрично-векторному умножению, где у вектора есть `n` рядов элементов со всеми значениями, равными `1.0 / n`. Точно так же вычисление среднего столбцов в такой таблице эквивалентно векторно-матричному умножению, где у вектора столбцов есть `m` элементов со всеми значениями, равными `1.0 / m`. Мы вернемся к векторно-матричному умножению в контексте важного приложения в конце этой главы.

Массивы с переменной длиной рядов. На самом деле нет никакого требования, чтобы у всех рядов двумерного массива была одинаковая длина. Массив с рядами неодинаковой длины — это *массив с переменной длиной рядов* (ragged array), пример его применения есть в упражнении 1.4.32. Код обработки массивов

с переменной длиной рядов требует дополнительной заботы. Например, следующий код выводит содержимое массива с переменной длиной рядов:

```
for i in range(len(a)):
    for j in range(len(a[i])):
        stdio.write(a[i][j])
        stdio.write(' ')
    stdio.writeln()
```

Этот код проверяет ваше понимание массивов Python, поэтому имеет смысл уделить время его изучению. В этой книге мы обычно используем квадратные или прямоугольные массивы, размерности которых заданы переменными `m` и `n`. Таким образом, наличие в коде функции `len(a[i])` является четким сигналом использования массива с рядами неодинаковой длины.

Обратите внимание, что эквивалентный код без использования индексов одинаково хорош и с обычными массивами, и с массивами переменной длины рядов:

```
for row in a:
    for v in row:
        stdio.write(v)
        stdio.write(' ')
    stdio.writeln()
```

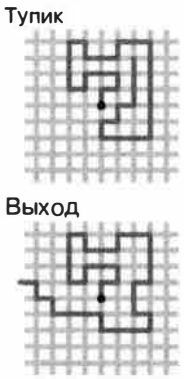
Многомерные массивы. Та же форма записи позволяет создавать код, использующий массивы с любым количеством размерностей. Используя массивы массивов массивов ..., можно создать трехмерные, четырехмерные и так далее массивы, а затем обращаться в коде к их индивидуальным элементам, как к `a[i][j][k]`, например.

Пример: случайное блуждание без самопересечений. Предположим, вы потеряли свою собаку в большом городе, улицы которого образуют знакомый узор таблицы. Предполагается, что с севера на юг и с востока на запад есть `n` улиц, и все они распределены равномерно и пересекаются под прямым углом, образуя узор в виде *решетки.* Пытаясь удрать из города, собака выбирает направление случайно, но на каждом перекрестке проверяет по запаху, не была ли она уже здесь раньше. Но собака вполне может застрять в тупике, когда у нее нет никакого выбора, кроме как попасть на собственные следы. Каков шанс, что это случится? Это занимательная задача, и ее простой пример известен как модель *случайного блуждания без самопересечений* (self-avoiding random walk). Она имеет важные применения в науке, включая исследования полимеров, в статистической механике и во многих других областях. Например, как можно заметить, этот процесс моделирует цепочку поэтапного материального роста, пока никакой рост не станет невозможен. Чтобы лучше понять такие процессы, ученые изучают свойства блуждания без самопересечений.

Вероятность успешного побега собаки зависит, конечно, от размера города. В небольшом городе, 5×5 улиц, довольно просто убедить себя в том, что собака легко сбежит. Но какова вероятность побега, когда город большой? Другие

параметры также интересны. Например, как долог средний путь собаки? Как часто в среднем собака попадает в безвыходный тупик предыдущих позиций? Как часто до выхода собаке остается только один квартал? Подобные свойства очень важны в различных случаях применения, включая упомянутые выше.

Программа 1.4.4 (`selfavoid.py`) моделирует эту ситуацию, используя двумерный массив логических значений, где каждый элемент представляет перекресток. Значение `True` означает, что собака уже была на перекрестке, а значение `False` — что еще нет. Путь начинается в центре и продолжается в случайных направлениях к еще не посещенным местам, пока собака не застрянет или найдет выход за границы. Для простоты написания кода сделаем так, что если случайный выбор сделан в направлении уже пройденного перекрестка, то не нужно делать ничего, а последующие случайные выборы найдут новое подходящее место (в этом можно быть уверенным, поскольку код явно проверяет наличие тупика и выходит в этом случае из цикла).



Блуждания без самопересечений

Обратите внимание: этот код демонстрирует важную методику программирования, согласно которой код проверки условия продолжения цикла `while` дублируется оператором *защиты* от недопустимого действия в теле цикла. В данном случае проверка условия продолжения цикла защищает от выхода за пределы массива внутри цикла. Это соответствует проверке, вышла ли собака из города. В пределах цикла успех проверки на наличие тупика приводит к оператору `break` и выходу из цикла.

Как можно заметить из пробных попыток, неутешительная правда в том, что в большом городе ваша собака почти наверняка окажется в тупике. Если задача о блуждании без самопересечений вас заинтересовала, то вы можете узнать больше и найти несколько рекомендаций в упражнениях. Например, в трехмерной версии задачи собака сбежит почти наверняка. Хотя это интуитивно понятный результат, подтвержденный нашей моделью, разработка математической модели, полностью объясняющей поведение при блуждании без самопересечений, остается известной нерешенной задачей: несмотря на обширные исследования, никто не нашел компактного математического выражения для вычисления вероятности побега, средней длины пути или любого другого важного параметра.

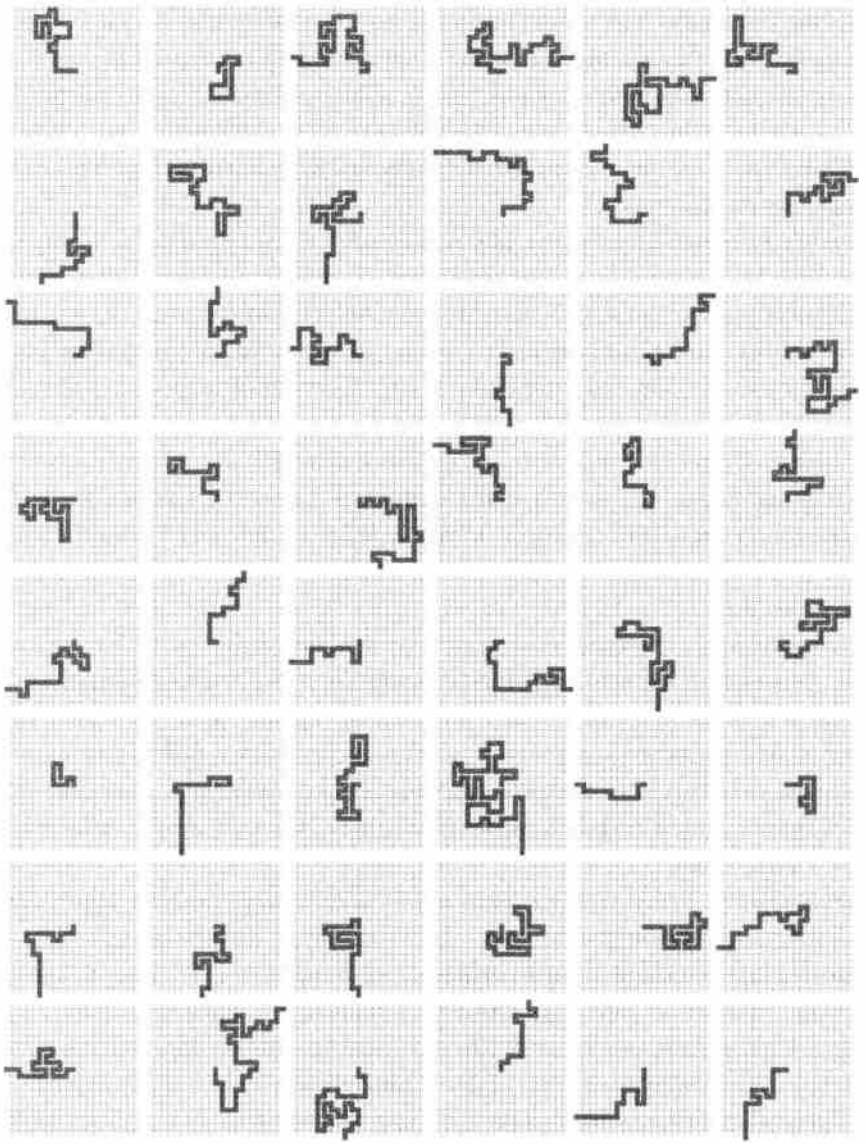
Программа 1.4.4. Случайные блуждания без самопересечений (selfavoid.py)

```
import random
import sys
import stdarray
import stdio

n      = int(sys.argv[1])
trials = int(sys.argv[2])
deadEnds = 0
for t in range(trials):
    a = stdarray.create2D(n, n, False)
    x = n // 2
    y = n // 2
    while (x > 0) and (x < n-1) and (y > 0) and (y < n-1):
        # Проверить тупик и сделать случайный ход.
        a[x][y] = True
        if a[x-1][y] and a[x+1][y] and a[x][y-1] and a[x][y+1]:
            deadEnds += 1
            break
        r = random.randrange(1, 5)
        if   (r == 1) and (not a[x+1][y]): x += 1
        elif (r == 2) and (not a[x-1][y]): x -= 1
        elif (r == 3) and (not a[x][y+1]): y += 1
        elif (r == 4) and (not a[x][y-1]): y -= 1

stdio.writeln(str(100*deadEnds//trials) + '% dead ends')
```

`n`	Размер решетки
`trials`	Количество испытаний
`deadEnds`	Количество испытаний, закончившихся тупиком
`a[][]`	Перекрестки
`x, y`	Текущая позиция
`r`	Случайное число в диапазоне [1, 5)

В аргументах командной строки эта программа получает целые числа `n` и `trials`. Она проводит `trials` экспериментов по случайному блужданию без самопересечений в решетке `n` на `n`, а затем выводит процент попаданий в тупик. Для каждого эксперимента она создает массив логических переменных, путь начинается в центре и продолжается до границы или тупика.

```
% python selfavoid.py 5 100
0% dead ends
% python selfavoid.py 20 100
35% dead ends
% python selfavoid.py 40 100
80% dead ends
% python selfavoid.py 80 100
98% dead ends
% python selfavoid.py 5 1000
0% dead ends
% python selfavoid.py 20 1000
32% dead ends
% python selfavoid.py 40 1000
76% dead ends
% python selfavoid.py 80 1000
98% dead ends
```



Случайные блуждания без самопересечений в таблице 21 × 21

Резюме. Массивы — четвертая фундаментальная конструкция (первые три — присвоение, условные выражения и циклы), присущая почти каждому языку программирования. Как было показано на нескольких примерах программ, приведенных в этом разделе, вполне можно составить программы, способные решить различного рода задачи, использующие только эти конструкции. В следующем разделе мы рассмотрим механизмы общения с нашими программами и завершим тему фундаментальных механизмов программирования.

Массивы применяются в большинстве рассматриваемых нами программ наравне с другими, обсуждавшимися здесь фундаментальными операциями, которые сослужат вам хорошую службу при решении многих задач программирования. Если вы не используете массивы явно (что весьма нередко), то вероятнее всего используете их неявно, поскольку память всех компьютеров концептуально эквивалентна массиву.

Фундаментальный компонент, добавляемый массивами в наши программы, — это потенциально очень большое количество *состояний* (state) программы. Состояние программы может быть определено как информация, необходимая для того, чтобы узнать и понять, что делает программа. В программе без массивов, если вы знаете значения переменных и следующий выполняемый оператор, то обычно можете предсказать то, что программа сделает далее. Трассируя программу, мы по существу отслеживаем ее состояние. Но когда программы используют массивы, значений может быть слишком много (и каждое из них может быть изменено каждым оператором), чтобы эффективно отслеживать их все. Это различие делает создание программ с массивами более сложным, чем создание программ без них.

Массивы непосредственно представляют векторы и матрицы, поэтому они напрямую применяются в вычислениях, связанных со многими фундаментальными проблемами в науке и технике. Массивы обеспечивают также компактную форму записи для манипулирования потенциально огромными объемами единообразных данных, поэтому они играют критически важную роль в любом приложении, обрабатывающем большие объемы данных, как будет продемонстрировано повсюду в этой книге.

Однако важней всего то, что массивы — только вершина айсберга. Это лишь первый пример *структуры данных* (data structure), позволяющий организовывать данные для удобной и эффективной их обработки. В главе 4 мы рассмотрим больше примеров структур данных, включая *массивы переменного размера* (resizing array), *связанные списки* (linked list), *бинарные деревья поиска* (binary search tree) и *хеш-таблицы* (hash table). Массивы — это также первый пример *коллекции* (collection), группирующей множество элементов в единый блок. В главе 4 приведено множество примеров других коллекций Python, включая *списки* (list), *кортежи* (tuple), *словари* (dictionary) и *наборы* (set), наряду с двумя их классическими примерами: *стеком* (stack) и *очередью* (queue).

Вопросы и ответы

Почему индексы массивов Python начинаются с 0 а не с 1?

Это соглашение произошло от машинных языков программирования, где адрес элемента массива вычислялся в результате добавления индекса к адресу начала массива. Начало индексации с 1 повлекло бы либо потерю пространства в начале массива или строки, либо потерю времени на вычитание 1.

Что будет при использовании отрицательного целого числа в качестве индекса массива?

Ответ может удивить вас. Если в массиве `a[]` использовать индекс `-i`, то получится сокращение от `len(a) - i`. Например, можно обратиться к последнему элементу в массиве как к `a[-1]` или `a[len(a) - 1]` либо к первому элементу как к `a[-len(a)]` или `a[0]`. Если использовать индекс вне диапазона допустимых элементов от `-len(a)` до `len(a) - 1`, во время выполнения произойдет ошибка `IndexError`.

Что означает запись `a[i:j]`, она включает `a[i]`, но исключает `a[j]`?

Эта форма записи соответствует диапазонам, определяемым функцией `range()`, которые включают левую конечную точку, но исключают правую. Это имеет несколько последствий: `j - i` — это длина части массива (не подразумевающая усечения); `a[0:len(a)]` — это весь массив; `a[i:i]` — пустой массив; `a[i:j] + a[j:k]` — часть массива `a[i:k]`.

Что происходит при сравнении двух массивов `a[]` и `b[]` в операторе `(a == b)`?

Это зависит от обстоятельств. Для массивов (и многомерных массивов) чисел это работает, как и следовало ожидать: массивы равны, если у каждого та же длина и соответствующие элементы равны.

Что будет, если случайное блуждание не избегает самопересечений?

Данный случай хорошо известен. Это двумерная версия задачи разорения игрока, описанной в разделе 1.3.

Какие проблемы не следует упускать при использовании массивов?

Помните, что *создание массива требует времени, пропорционального длине массива.* Особую осторожность следует соблюдать при создании массивов в пределах циклов.

Упражнения

1.4.1. Составьте программу, которая создает одномерный массив `a`, содержащий ровно 1 000 целых чисел, а затем пытается получить доступ к элементу `a[1000]`. Что произойдет при запуске программы?

1.4.2. Есть два вектора длиной `n`, представленные одномерными массивами. Составьте фрагмент кода, вычисляющий *Евклидово расстояние* между ними (квадратный корень сумм квадратов разниц соответствующих элементов).

1.4.3. Составьте фрагмент кода, меняющий на обратный порядок элементов в одномерном массиве типа `float`. Другой массив для содержания результата не создавайте. *Подсказка*: для обмена двух элементов используйте код, приведенный ранее.

1.4.4. Что не так со следующим фрагментом кода?

```
a = []
for i in range(10):
    a[i] = i * i
```

Решение. Первоначально массив `a` пуст. Никаких элементов в него впоследствии не добавляется. Таким образом, элементы `a[0]`, `a[1]` и так далее не существуют. Во время попытки их использования в операторе присвоения произойдет ошибка `IndexError`.

1.4.5. Составьте фрагмент кода, выводящий содержимое двумерного массива логических переменных, используя знак `*` для представления значения `True` и пробел для представления значения `False`. Выводите также номера рядов и столбцов.

1.4.6. Что выводит следующий фрагмент кода?

```
a = stdarray.create1D(10, 0)
for i in range(10):
    a[i] = 9 - i
for i in range(10):
    a[i] = a[a[i]]
for v in a:
    stdio.writeln(v)
```

1.4.7. Что содержит массив `a[]` после выполнения следующего фрагмента кода?

```
n = 10
a = [0, 1]
for i in range(2, n):
    a += [a[i-1] + a[i-2]]
```

1.4.8. Составьте программу `deal.py`, получающую аргумент командной строки `n` и выводящую `n` разделенных пустыми строками комбинаций карт (по пять в каждой) из перетасованной колоды.

1.4.9. Составьте фрагмент кода, создающий двумерный массив `b[][]`, являющийся копией существующего двумерного массива `a[][]`, согласно каждому из следующих предположений:

a. Массив `a` квадратный.

b. Массив `a` прямоугольный.

c. Ряды массива `a` могут быть неодинаковой длины.

Ваше решение *b* должно работать и для *a*, а решение *c* должно работать и для *b*, и для *a*.

1.4.10. Составьте фрагмент кода, выводящий *транспозицию* (замена рядов и столбцов) двумерного массива. Для примера массива таблицы в тексте ваш код должен вывести следующее:

```
99 98 92 94 99 90 76 92 97 89
85 57 77 32 34 46 59 66 71 29
98 78 76 11 22 54 88 89 24 38
```

1.4.11. Составьте фрагмент кода для транспозиции квадратного двумерного массива *на месте*, без создания второго массива.

1.4.12. Составьте фрагмент кода для создания двумерного массива `b[][]`, являющегося транспозицией существующего массива `a[][]` размером $m \times n$.

1.4.13. Составьте программу, вычисляющую произведение двух квадратных матриц логических переменных, используя оператор `or` вместо `+` и оператор `and` вместо `*`.

1.4.14. Составьте программу, получающую из командной строки целое число `n` и создающую массив `a` размером $n \times n$ элементов логического типа таким образом, чтобы элемент `a[i][j]` содержал значение `True`, если значения `i` и `j` являются *взаимно простыми* (relatively prime), т.е. не имеющими общих делителей, и `False` в противном случае. Затем выведите массив (см. упр. 1.4.5), используя `*` для представления значений `True` и пробел для `False`. Выводите также номера рядов и столбцов. *Подсказка*: используйте решето Эратосфена.

1.4.15. Измените фрагмент кода таблицы в тексте так, чтобы вычислить *средневзвешенное* значение рядов, где весовые коэффициенты каждой экзаменационной отметки находятся в одномерном массиве `weights[]`.

Например, чтобы получить последнюю из трех оценок в нашем примере, используются два других весовых коэффициента:

```
weights = [.25, .25, .50]
```

Обратите внимание: сумма весов должна составлять `1.0`.

1.4.16. Составьте фрагмент кода для умножения двух прямоугольных матриц, которые не обязательно являются квадратными. *Примечание*: для скалярного произведения количество столбцов в первой матрице должно быть равно количеству рядов во второй матрице. Если размерности не удовлетворяют этому условию, выведите сообщение об ошибке.

1.4.17. Измените программу 1.4.4 (`selfavoid.py`) так, чтобы вычислять и выводить среднюю длину путей и вероятность тупиков. Среднюю длину успешных и тупиковых путей выводите в отдельности.

1.4.18. Измените программу `selfavoid.py` так, чтобы вычислить и вывести среднюю площадь наименьшего координатного прямоугольника, заключающего путь. Статистику успешных и тупиковых путей выводите в отдельности.

1.4.19. Составьте фрагмент кода, создающий трехмерный массив $n \times n$ логических переменных, каждый элемент которого инициализирован значением `False`.

Практические упражнения

1.4.20. *Модель игральной кости.* Следующий код вычисляет точную вероятность выпадения и сумму значений двух игральных костей:

```
probabilities = stdarray.create1D(13, 0.0)
for i in range(1, 7):
    for j in range(1, 7):
        probabilities[i+j] += 1.0
for k in range(2, 13):
    probabilities[k] /= 36.0
```

По завершении этого кода элемент `probabilities[k]` содержит вероятность суммарного значения `k` игральных костей. Поэкспериментируйте и проверьте эту модель для n бросков игральных костей, следя за частотами выпадения каждого значения, когда вы вычисляете сумму двух случайных целых чисел от 1 до 6. Насколько большим должно быть n прежде, чем ваши эмпирические результаты совпадут с точными результатами при трех позициях десятичной точки?

1.4.21. *Самое длинное плато.* Дан массив целых чисел. Составьте программу, находящую длину и положение самой длинной непрерывной последовательности равных значений, где значения элементов непосредственно перед и сразу после этой последовательности меньше.

1.4.22. *Эмпирическая проверка перетасовки.* Проведите компьютерные эксперименты по проверке того, что наш код перетасовки карт работает так, как обещано. Составьте программу `shuffletest.py`, получающую в аргументах командной строки значения m и n, осуществляющую n перестановок массива размера m, инициализируемого как `a[i] = i` перед каждой перестановкой, и выводящую таблицу $m \times m$ таким образом, что ряд `i` окажется в позиции `j` для всех `j`. Все элементы в массиве должны быть близки к n/m.

1.4.23. *Плохая перетасовка.* Предположим, в нашем коде перетасовки выбирается случайное целое число от `0` до `n-1`, а не от `i` до `n-1`. Покажите, почему получающийся порядок *не даст* одинаковой вероятности и не обеспечит одну из $n!$ возможностей. Проверьте предыдущее упражнение для этой версии.

1.4.24. *Случайный порядок проигрывания.* Вы переводите свой аудиоплеер в режим случайного проигрывания. Это запускает каждую из m песен прежде, чем начать повтор. Составьте программу оценки вероятности, что вы не услышите последовательной пары песен (т.е. песня 3 не следует за песней 2, а песня 10 не следует за песней 9 и т.д.).

1.4.25. *Минимумы в перестановках.* Составьте программу, которая получает в аргументах командной строки целое число `n`, осуществляет случайную перестановку, а затем выводит перестановку и количество (слева направо) минимумов в перестановке (количество вхождений наименьшего элемента, замеченное до сих пор). Затем составьте программу, которая получает в аргументах командной строки целые числа m и n, создает m случайных перестановок размера n и выводит среднее количество (слева направо) минимумов в созданных перестановках. *Дополнительное задание:* сформулируйте гипотезу о количестве минимумов слева направо в перестановке размера n, как функцию n.

1.4.26. *Обратная перестановка.* Составьте программу, которая читает перестановку целых чисел от `0` до `n-1` из `n` аргументов командной строки и выводит ее инверсию. (Если перестановка — это массив `a[]`, то его инверсия,

массив `b[]`, получается как `a[b[i]] = b[a[i]] = i`.) Убедитесь, что ввод является допустимой перестановкой.

1.4.27. *Матрица Адамара.* Матрица Адамара $n \times n$, или H_n, — это матрица логических элементов с замечательным свойством: любые два ряда отличаются точно на $n/2$ элементов. (Это свойство делает ее весьма полезной для разработки кода исправления ошибок.) H_1 — это матрица 1×1 с одним элементом `True`, а для $n > 1$ матрица H_{2n} получается из расположения четырех копий H_n в большем четырехугольнике и последующей инверсии всех элементов в правой нижней копии $n \times n$, как показано в следующих примерах (где `True` представлено как `T`, а `False` как `F`):

H_1	H_2	H_4
T	T T	T T T T
	T F	T F T F
		T T F F
		T F F T

Составьте программу, получающую один аргумент командной строки n и выводящую H_n. Подразумевается, что n — степень числа 2.

1.4.28. *Слухи.* Алиса устраивает вечеринку на n гостей, включая Боба. Боб сообщает о вечеринке у Алисы одному из других гостей. Услышавший это приглашенный немедленно сообщает о вечеринке следующему выбранному наугад гостю, кроме Алисы и человека, от которого они услышал новость. Услышав о вечеринке во второй раз, каждый гость (включая Боба) уже не передает его далее. Составьте программу оценки вероятности того, что *все* участники (кроме Алисы) узнают о вечеринке, прежде чем слух прекратит распространяться. Вычислите также оценку ожидаемого количества людей, способных услышать слух.

1.4.29. *Поиск дубликатов.* Дан массив из n элементов (от 1 до n). Составьте фрагмент кода для поиска возможных дубликатов. Сохранять содержимое данного массива не нужно, но и нельзя использовать дополнительный массив.

1.4.30. *Подсчет множителей.* Сравните программу `primesieve.py` с подходом, альтернативным описанному в тексте. Это пример обмена пространства на время: программа `primesieve.py` быстра, но требует массива логических значений длиной n; альтернативный подход использует только две целочисленные переменные, но он существенно медленнее. Оцените величину этого различия при поиске значения n, для которого второй

подход может закончить вычисление приблизительно за то же время, что и `python primesieve.py 1000000`.

1.4.31. *Сапер.* Составьте программу, которая получает три аргумента командной строки, m, n и p, а затем создает массив логических элементов размером $m \times n$, каждый элемент которой занят с вероятностью p. В этой игре занятые ячейки соответствуют минам, а пустые безопасны. Выведите массив, используя звездочку для мин и точку для безопасных мест. Затем замените каждый безопасный квадрат количеством соседних мин (выше, ниже, слева, справа и по диагонали) и выведите результат, как в этом примере:

```
* * . . .     * * 1 0 0
. . . . .     3 3 2 0 0
. * . . .     1 * 1 0 0
```

Попытайтесь выразить свой код так, чтобы он имел по возможности меньше частных случаев, для этого используйте логический массив $(m + 2)$ на $(n + 2)$.

1.4.32. *Длина пути при блуждании без самопересечений.* Предположим, размер таблицы не имеет предела. Начните эксперименты с оценки средней длины пути.

1.4.33. *Трехмерные блуждания без самопересечений.* Проведите эксперименты по проверке нулевой вероятности тупиков при блуждании без самопересечений в трехмерном пространстве и вычислите среднюю длину пути для разных значений n.

1.4.34. *Случайные ходы.* Предположим, из центра таблицы $n \times n$ совершается n случайных ходов, по одному за раз. Вероятность выбора хода вверх, вниз, вправо и влево одинакова. Составьте программу, помогающую сформулировать и проверить гипотезу о количестве шагов, сделанных прежде, чем будут затронуты все ячейки.

1.4.35. *Сдача в бридж.* При четырех игроках в бридж сдается по 13 карт. Важнейшей статистической величиной является распределение количества карт при каждой сдаче. Что наиболее вероятно, 5–3–3–2, 4–4–3–2 или 4–3–3–3? Составьте программу, помогающую ответить на этот вопрос.

1.4.36. *Задача дня рождения.* Предположим, в пустую комнату входят люди, пока у двух из них не совпадут дни рождения. Сколько в среднем людей должно войти в комнату до совпадения? Проведите эксперименты для оценки значения этого количества. Подразумевается, что дни рождения — равномерно распределенные случайные целые числа от 0 до 364.

1.4.37. *Коллекционер купонов.* Проведите эксперименты по проверке классического математического результата для ожидаемого количества n собранных купонов из nH_n. Например, если внимательно следить за картами при сдаче в блэк-джек (и у дилера есть достаточно много тщательно перетасованных колод), то в среднем придется выждать примерно 235 карт, прежде чем они начнут повторяться.

1.4.38. *Перестановка riffle shuffle.* Составьте программу по перераспределению n карт в колоде, используя модель Гилберта–Шаннона–Ридса riffle shuffle. Сначала создайте случайное целое число r согласно *биномиальному распределению* (binomial distribution): бросьте монету n раз и присвойте r количество орлов. Затем разделите колоду надвое: первые r карт и следующие $n-r$ карт. Для завершения перетасовки последовательно берите верхнюю карту из каждой половины колоды и помещайте ее вниз другой. Если в первой половине колоды остается n_1 карт, а во второй — n_2 карт, выберите следующую карту из первой половины колоды с вероятностью $n_1 / (n_1 + n_2)$ и из второй с вероятностью $n_2 / (n_1 + n_2)$. Выясните экспериментально, сколько таких перетасовок необходимо для колоды из 52 карт, чтобы получить однородно перетасованную колоду.

1.4.39. *Биномиальные коэффициенты.* Составьте программу, создающую и выводящую двумерный массив с переменной длиной строк `a` таким образом, чтобы элемент `a[n][k]` содержал вероятность получить `k` орлов при `n` бросках монеты. Для определения максимального значения `n` используйте аргумент командной строки. Эти числа известны как *биномиальное распределение*: если умножить каждый элемент ряда k на 2^n, то получатся *биномиальные коэффициенты* (коэффициенты x^k в $(x+1)^n$), упорядоченные в *треугольник Паскаля*. Для их вычисления начните с `a[n][0] = 0.0` для всех `n` и `a[1][1] = 1.0`, затем вычислите значения в последующих рядах, слева направо, как `a[n][k] = (a[n-1][k] + a[n-1][k-1])/2.0`.

Треугольник Паскаля	Биномиальное распределение
1	1
1 1	1/2 1/2
1 2 1	1/4 1/2 1/4
1 3 3 1	1/8 3/8 3/8 1/8
1 4 6 4 1	1/16 1/4 3/8 1/4 1/16

1.5. Ввод и вывод

В этом разделе расширяется набор простых абстракций (аргументы командной строки и стандартное устройство вывода), используемых как интерфейс между нашими программами Python и внешним миром, включая *стандартное устройство ввода*, *стандартное графическое устройство* и *стандартное аудиоустройство*. Стандартное устройство ввода позволяет создавать программы для обработки произвольных объемов ввода, а также взаимодействовать с нашими программами; стандартное графическое устройство позволяет работать с изображениями, избавляя от необходимости кодировать все как только текст; стандартное аудиоустройство добавляет звук. Эти расширения очень удобны и открывают новые возможности в программировании.

Программы этого раздела...

Программа 1.5.1. Создание случайной последовательности (randomseq.py) 149
Программа 1.5.2. Интерактивный пользовательский ввод (twentyquestions.py) 157
Программа 1.5.3. Среднее потока чисел (average.py) 159
Программа 1.5.4. Простой фильтр (rangefilter.py) 163
Программа 1.5.5. Стандартный ввод и фильтрация рисунка (plotfilter.py) 169
Программа 1.5.6. Вывод графика функции (functiongraph.py) 171
Программа 1.5.7. Прыгающий мяч (bouncingball.py) 176
Программа 1.5.8. Обработка цифрового сигнала (playthattune.py) 181

I/O — это сокращение от *input/output* (*ввод и вывод*); собирательный термин, относящийся к механизмам общения программ с внешним миром. Подключенные к компьютеру физические устройства контролирует операционная система. Для реализации абстракции "стандартный ввод и вывод" используются модули, содержащие функции, взаимодействующие с операционной системой.

Вы уже получали аргументы из командной строки и выводили строки в окно терминала. Цель данного раздела — снабдить вас намного более богатым набором инструментальных средств для обработки и представления данных. Подобно использовавшимся ранее функциям `stdio.write()` и `stdio.writeln()`, эти функции не реализуют чистые математические функции — их цель в некоем *побочном эффекте* на устройстве ввода или вывода данных. Наша главная задача — использовать такие устройства для передачи информации в наши программы и ее вывода из них.

Основная особенность стандартных устройств ввода и вывода в том, что для них нет никакого предела на объем входных или выходных данных с точки зрения программы. Программы могут использовать ввод или формировать вывод неопределенно большого размера.

Одна из областей применения стандартных механизмов ввода и вывода — это использование в программах *файлов* на запоминающем устройстве компьютера. Связать с файлом стандартные устройства ввода, вывода, графики и аудио довольно просто. Такие связи облегчают программам сохранение результатов в файлы и их последующую загрузку данной или другой программой.

Общая панорама. Обычная модель, используемая в программировании Python, исправно служит нам начиная с раздела 1.1. Для создания контекста начнем с краткого рассмотрения модели.

Программа Python получает вводимые значения из командной строки, а в вывод записывает строку символов. Стандартно и *аргументы командной строки*, и *стандартное устройство вывода* связаны с получающим команды приложением (указанным после команды `python`). Для взаимодействия с этим приложением используется *окно терминала* (terminal window). Эта модель выглядит простым и удобным способом взаимодействия с нашими программами и данными.

Аргументы командной строки. Этот механизм, используемый нами для ввода данных в программы, является стандартной частью программирования Python. Операционная система представляет вводимые в программы Python аргументы командной строки как массив `sys.argv[]`. В соответствии с соглашением и Python, и операционная система обрабатывают аргументы как строки, поэтому, если аргумент должен быть числом, для преобразования строки в соответствующий тип используется функция преобразования `int()` или `float()`.

Стандартное устройство вывода. Для вывода полученных в программах значений используются функции `stdio.write()` и `stdio.writeln()` с сайта книги. Язык Python передает результаты обращения к этим функциям в программе в виде абстрактного потока символов — *стандартного вывода*. Стандартно операционная система ассоциирует стандартное устройство вывода с окном терминала. До сих пор весь вывод в наших программах осуществлялся в окне терминала.

Для справки и для начала программа 1.5.1 (`randomseq.py`) использует именно эту модель. Она получает аргумент командной строки `n` и выводит последовательность из `n` случайных чисел в диапазоне от 0 до 1.

Теперь мы дополним аргументы командной строки и стандартное устройство вывода тремя другими механизмами, снимающими их ограничения и предоставляющими намного более полезную модель программирования. Эти механизмы обеспечивают новую общую панораму программирования на языке Python, в которой программа преобразует поток стандартного ввода и последовательность аргументов командной строки в поток стандартного устройства вывода, стандартного графического устройства или поток стандартного аудиоустройства.

Программа 1.5.1. Создание случайной последовательности (randomseq.py)

```
import random
import sys
import stdio

n = int(sys.argv[1])
for i in range(n):
    stdio.writeln(random.random())
```

Эта программа получает целочисленный аргумент командной строки `n` и выводит на стандартное устройство вывода последовательность из `n` случайных чисел типа `float` в диапазоне [0, 1). Программа иллюстрирует обычную модель программирования Python, использовавшуюся до сих пор. С точки зрения программы нет никакого предела для длины выводимой последовательности.

```
% python randomseq.py 1000000
0.879948024484513
0.8698170909139995
0.6358055797752076
0.9546013485661425
...
```

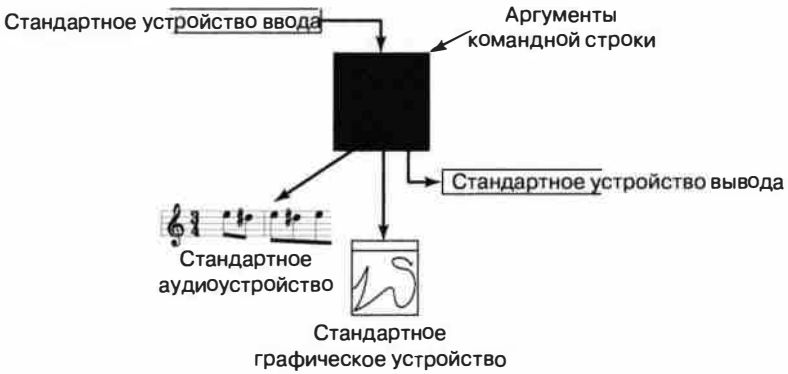
Стандартное устройство ввода. Книжный модуль `stdio` определяет еще несколько функций, кроме `stdio.write()` и `stdio.writeln()`. Эти функции реализуют абстракцию стандартного устройства ввода, чтобы дополнить абстракцию стандартного устройства вывода. Таким образом, модуль `stdio` содержит функции, позволяющие вашим программам *читать* со *стандартного устройства ввода*. Подобно тому, как на стандартное устройство вывода можно писать в любое время выполнения программы, читать из потока стандартного устройства ввода также можно в любое время.

Стандартное графическое устройство. Книжный модуль `stddraw` позволяет вашим программам создавать рисунки. Он использует простую графическую модель, позволяющую создавать рисунки, состоящие из точек, линий и геометрических фигур в окне компьютера. Модуль `stddraw` включает также средства для текста, цвета и анимации.

Стандартное аудиоустройство. Книжный модуль `stdaudio` позволяет создавать звук и манипулировать им в ваших программах. Он использует стандартный формат преобразования массива типа `float` в звук.

Для использования этих модулей файлы `stdio.py`, `stddraw.py` и `stdaudio.py` следует сделать доступными для Python (см. вопросы и ответы в конце этого раздела).

Абстракции стандартных устройств ввода и вывода появились в период разработки операционной системы Unix в 1970-х годах и в той или иной форме реализованы всеми современными системами. Хотя они и примитивны по сравнению с различными механизмами, появившимися с тех пор, современные программисты все еще зависят от них как от наиболее надежного способа объединения данных с программами. Для этой книги модули `stddraw` и `stdaudio` мы разработали в том же духе, что и прежние абстракции, чтобы снабдить вас простым способом осуществления визуального и звукового вывода.



Общая панорама программы Python (повтор)

Стандартное устройство вывода. Как уже упоминалось в разделе 1.2, *интерфейс прикладных программ* (API) представляет собой описание средств, предоставляемых модулем своим клиентам. Ниже приведена часть API модуля `stdio`, относящаяся к стандартному устройству вывода. Функции `stdio.write()` и `stdio.writeln()` уже использовались. Функция `stdio.writef()` — это основная тема данного раздела, она и будет представлять теперь интерес для вас, поскольку обеспечивает больше контроля над выводом. Она появилась в языке C в начале 1970-х годов и продолжает существовать в новых языках, поскольку она весьма полезна.

API для функций с сайта книги, связанных со стандартным устройством вывода

Вызов функции	Описание
`stdio.write(x)`	Выводит `x` на стандартное устройство вывода
`stdio.writeln(x)`	Выводит `x` и новую строку на стандартное устройство вывода (если аргумента нет, то выводит только новую строку)
`stdio.writef(fmt, arg1, ...)`	Выводит аргументы `arg1...` на стандартное устройство вывода, как определено форматом `fmt`

С самого начала мы выводили объекты типа `float` с чрезмерной раздражающей точностью. Например, при вызове функции `stdio.write(math.pi)` мы

получаем вывод `3.141592653589793`, хотя вполне могли бы (и даже предпочли бы) обойтись значением `3.14` или `3.14159`. Функции `stdio.write()` и `stdio.writeln()` представляют каждое число с точностью до 16 десятичных цифр, даже когда вполне хватило бы лишь нескольких цифр. Функция `stdio.writef()` гибче: она позволяет задать количество цифр и точность при преобразовании числовых объектов в строки для вывода. Чтобы получить в выводе `3.14159`, вызов функции `stdio.writef()` можно составить так: `stdio.writef('%7.5f', math.pi)`.

Рассмотрим смысл и работу этих операторов наряду с модификациями для обработки других встроенных типов данных.

Основы форматированного вывода. В самом простом случае функции `stdio.writef()` при вызове передается только один аргумент — строка. В этом случае она просто выводит строку на стандартное устройство вывода, что эквивалентно вызову функции `stdio.write()`. В более популярном случае функции `stdio.writef()` передаются два аргумента: первый — это *строка формата* (format string), содержащая *спецификацию преобразования* (conversion specification). Она описывает, как второй аргумент должен быть преобразован в строку для вывода. Спецификация преобразования имеет формат `%w.pc`, где `w` и `p` — небольшие целые числа, а `c` — символ, интерпретирующийся следующим образом:

- `w` — *ширина поля* (field width) — количество символов, подлежащих выводу. Если количество выводимых символов превышает (или равняется) ширину поля, то ширина поля игнорируется; в противном случае вывод дополняется с пробелами слева. Отрицательная ширина поля означает дополнение вывода пробелами справа.
- `p` — *точность* (precision). Для чисел типа `float` — это количество цифр, выводимых после десятичной точки; для строк — это количество выводимых символов строки. С целыми числами точность не используется.
- `c` — *код преобразования* (conversion code). При выводе целого числа это должно быть `d`, при выводе числа типа `float` — в обычной форме `f`, а в экспоненциальной форме — `e`. При выводе строки используется символ `s`.

Анатомия оператора форматированного вывода

Ширину поля и точность можно пропустить, но код преобразования должен быть в каждой спецификации.

Python должен быть в состоянии преобразовать второй аргумент в тип, заданный спецификацией. Для кода `s` никаких ограничений нет, поскольку любой тип данных может быть преобразован в строку (функцией `str()`). Такой оператор, как `stdio.writef('%12d', 'Hello')`, напротив, требует преобразования строки в целое число, а при невозможности сообщает об ошибке `TypeError` во время выполнения. В таблице ниже приведены строки формата, содержащие некоторые наиболее популярные спецификации преобразования.

Соглашения по формату для функции `stdio.writef()` (множество других параметров приведено на сайте книги)

Тип	Код	Типичный литерал	Пример строки формата	Вывод
`int`	`d`	`512`	`'%14d'`	`'  512'`
			`'%-14d'`	`'512  '`
`float`	`f`	`1595.1680010754388`	`'%14.2f'`	`'  1595.17'`
	`e`		`'%.7f'`	`'1595.1680011'`
			`'%14.4e'`	`'  1.5952e+03'`
`String`	`s`	`'Hello, World'`	`'%14s'`	`'  Hello, World'`
			`'%-14s'`	`'Hello, World  '`
			`'%-14.5s'`	`'Hello  '`

Любая часть строки формата, не являющаяся спецификацией преобразования, просто передается на стандартное устройство вывода. Например, оператор

```
stdio.writef('pi is approximately %.2f\n', math.pi)
```

выводит строку

```
pi is approximately 3.14
```

Обратите внимание, что символ новой строки `\n` необходимо явно включить в аргумент, чтобы с помощью функции `stdio.writef()` писать с новой строки.

Несколько аргументов. Функция `stdio.writef()` может получить больше двух аргументов. В этом случае у строки формата должен быть спецификатор формата для каждого дополнительного аргумента, возможно, отделенный другими символами для передачи в вывод. Например, оператор `stdio.write(t)` в программе 1.3.6 (`sqrt.py`) можно заменить следующим:

```
stdio.writef('The square root of %.1f is %.6f', c, t)
```

и получить такой вывод, как

```
The square root of 2.0 is 1.414214
```

В качестве более детального примера, если вы вносили платежи по ссуде, то могли бы использовать код, внутренний цикл которого содержит операторы

```
format = '%3s  $%6.2f   $%7.2f   $%5.2f\n'
stdio.writef(format, month[i], pay, balance, interest)
```

для вывода второй и последующих строк в таблице, как здесь (см. упражнение 1.5.14):

```
    payment   balance   interest
Jan $299.00  $9742.67   $41.67
Feb $299.00  $9484.26   $40.59
Mar $299.00  $9224.78   $39.52
...
```

Отформатированная запись удобна, поскольку ее код намного компактней, чем код на базе конкатенации, использовавшийся ранее для создания строк вывода. Это описание лишь базовых возможностей, подробности приведены на сайте книги.

Стандартное устройство ввода. Несколько функций книжного модуля `stdio` получают данные из *потока* стандартного устройства ввода, который может быть пуст или может содержать последовательность значений, разделенных символами отступа (пробелами, табуляцией, символами новой строки и т.д.). Каждое значение представляет целое число, число типа `float`, логическое значение или строку. Одной из главных особенностей потока стандартного ввода является то, что программа использует значения по мере их чтения. Как только программа прочитала значение, она не может вернуться и прочитать его снова. Это отражает физические особенности некоторых устройств ввода данных и упрощает реализацию абстракции. Модуль `stdio` предоставляет 13 функций, связанных с чтением со стандартного устройства ввода (см. таблицу API ниже). Эти функции относятся к одной из трех категорий: для чтения отдельных лексем по одной за раз и преобразования каждой в целое число, число типа `float`, логический или строковый тип; для чтения со стандартного устройства ввода строк по одной и для чтения последовательности значений одинакового типа (возвращая значения в массиве). Вообще-то, в той же программе лучше не смешивать функции разных категорий. Согласно потоковой модели, эти функции в значительной степени самодокументированы (имена описывают их действие), однако их конкретная реализация достойна подробного изучения, поэтому рассмотрим детально несколько примеров.

API для функций с сайта книги, связанных со стандартным устройством ввода

Вызов функции	Описание
Функции, читающие отдельные лексемы со стандартного устройства ввода	
`stdio.isEmpty()`	Действительно ли стандартное устройство ввода пусто (или это только отступ)?
`stdio.readInt()`	Читает лексему, преобразует ее в целое число и возвращает
`stdio.readFloat()`	Читает лексему, преобразует ее в тип `float` и возвращает

Окончание

Вызов функции	Описание
Функции, читающие отдельные лексемы со стандартного устройства ввода	
`stdio.readBool()`	Читает лексему, преобразует ее в тип `bool` и возвращает
`stdio.readString()`	Читает лексему и возвращает ее как строку
Функции, читающие строки со стандартного устройства ввода	
`stdio.hasNextLine()`	Есть ли у стандартного устройства ввода следующая строка?
`stdio.readLine()`	Читает следующую строку и возвращает ее
Функции, читающие последовательность значений того же типа, пока стандартное устройство ввода не окажется пусто	
`stdio.readAll()`	Читает весь оставшийся ввод и возвращает его как строку
`stdio.readAllInts()`	Читает все оставшиеся лексемы и возвращает их как массив целых чисел
`stdio.readAllFloats()`	Читает все оставшиеся лексемы и возвращает их как массив чисел типа `float`
`stdio.readAllBools()`	Читает все оставшиеся лексемы и возвращает их как массив типа `bool`
`stdio.readAllStrings()`	Читает все оставшиеся лексемы и возвращает их как массив строк
`stdio.readAllLines()`	Читает все оставшиеся строки и возвращает их как массив строк

Примечание 1. Лексема — это максимальная последовательность символов, не являющихся отступами.

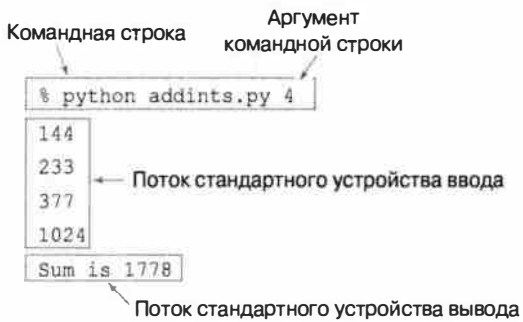
Примечание 2. Перед чтением лексемы все предваряющие отступы отбрасываются.

Примечание 3. Каждая читающая ввод функция передает ошибку времени выполнения, если не может прочитать следующее значение, будь то отсутствие ввода или несоответствие ввода ожидаемому типу.

Типизация ввода. При использовании команды `python` для запуска программы Python из командной строки фактически происходят три вещи: (1) команда на запуск выполнения программы; (2) определение значений параметров командной строки и (3) запуск чтения потока стандартного устройства ввода. Строка символов, выводимая в окно терминала после командной строки, и *является* потоком стандартного устройства ввода. Когда вы вводите символы, вы взаимодействуете со своей программой. Программа *ждет*, что именно вы создадите поток стандартного устройства ввода. Рассмотрим, например, программу `addints.py`, получающую из командной строки целое число `n`, а затем читающую `n` целых чисел со стандартного устройства ввода, суммирующую их и выводящую сумму на стандартное устройство вывода:

```
import sys
import stdio
n = int(sys.argv[1])
total = 0
for i in range(n):
    total += stdio.readInt()
stdio.writeln('Sum is ' + str(total))
```

Когда вы вводите `python addints.py 4`, начинается выполнение программы. Она получает аргумент командной строки, инициализирует переменную `total` значением `0`, входит в цикл `for` и в конечном счете, вызвав функцию `stdio.readInt()`, *ждет* от вас ввода целого числа. Предположим, что первым вы хотите ввести значение `144`. Вы нажимаете клавиши <1>, затем <4> и наконец <4>, но ничего не происходит, поскольку модуль `stdio` не знает, что вы закончили ввод числа. Однако, когда вы нажимаете клавишу <return>, чтобы подтвердить завершение ввода целого числа, функция `stdio.readInt()` немедленно возвращает значение `144`, которое ваша программа добавляет к значению переменной `total`, а затем снова вызывает функцию `stdio.readInt()`. И снова ничто не происходит, пока вы не введете второе значение: если вы нажмете клавиши <2>, <3>, <3> и <return>, чтобы завершить ввод, функция `stdio.readInt()` возвратит значение `233`, которое программа снова добавит к значению переменной `total`. После ввода четырех чисел программа перестанет ожидать последующего ввода и выведет сумму, как и ожидалось. В трассировках командной строки мы используем полужирный шрифт для выделения текста, вводимого вами, чтобы отличить его от вывода программы.



Анатомия команды

Формат ввода. Функция `stdio.readInt()` ожидает целое число. Если во время выполнения ввести `abc`, или `12.2`, или `True`, произойдет ошибка `ValueError`. Формат для каждого типа тот же, что и при определении литералов в программах Python. Для удобства модуль `stdio` рассматривает строки из нескольких символов отступа подряд как один пробел и позволяет разграничивать вводимые числа такими строками. Количество пробелов между числами не имеет значения,

равно как и введены ли числа в одну строку, разделены ли они символами табуляции или распространяются на несколько строк (за исключением случая, когда приложение читает ввод построчно, а следовательно, ожидает нажатия клавиши <return>, чтобы передать все числа в строке стандартному устройству ввода). Во входном потоке значения различных типов могут быть смешаны, но каждый раз, когда программа ожидает значение определенного типа, во входном потоке должно быть значение именно этого типа.

Интерактивный пользовательский ввод. Программа 1.5.2 (`twentyquestions.py`) является простым примером взаимодействия программы с ее пользователем. Программа создает случайное целое число, а затем предлагает пользователю угадать его. (*Примечание*: используя *бинарный поиск* (binary search), ответ можно найти максимум за 20 попыток. См. раздел 4.2.) Фундаментальное различие между этой программой и другими, описанными ранее, в том, что у пользователя есть возможность контролировать поток *во время* выполнения программы. В прошлом для первых компьютерных программ эта возможность была очень важна, но в настоящее время такие программы редки, поскольку современные приложения, как правило, осуществляют такой ввод через графический интерфейс пользователя, как обсуждается в главе 3. Даже такая простая программа, как `twentyquestions.py`, иллюстрирует потенциально весьма высокую сложность создания таких программ, поскольку придется запланировать все возможные действия пользователя.

Обработка входного потока произвольного размера. Как правило, входные потоки конечны: программа читает значения из входного потока, пока он не опустеет. Однако ограничений на размер входного потока нет, и некоторые программы просто обрабатывают весь представленный им ввод. Наш следующий пример, программа 1.5.3 (`average.py`), читает последовательности вещественных чисел со стандартного устройства ввода и выводит их среднее значение. Это иллюстрирует ключевое свойство входного потока: его длина не известна программе. Мы вводим все имеющиеся числа, а затем программа вычисляет их среднее. Перед чтением каждого числа программа вызывает функцию `stdio.isEmpty()`, чтобы удостовериться в наличии чисел во входном потоке.

Как сообщить, что данные больше вводиться не будут? В соответствии с соглашением необходимо ввести специальную последовательность символов, известную как *конец файла*. К сожалению, терминальные приложения, предоставляемые современными операционными системами, используют разные соглашения для этой критически важной последовательности. В этой книге мы используем комбинацию клавиш <Ctrl-d> (многие системы требуют ввода комбинации <Ctrl-d> в отдельной строке); согласно другому популярному соглашению это комбинация <Ctrl-z>.

Программа 1.5.2. Интерактивный пользовательский ввод (twentyquestions.py)

```
import random
import stdio

RANGE = 1000000

secret = random.randrange(1, RANGE+1)
stdio.write('I am thinking of a secret number between 1 and ')
stdio.writeln(RANGE)

guess = 0
while guess != secret:
    # Сделайте одно предположение и представьте один ответ.
    stdio.write('What is your guess? ')
    guess = stdio.readInt()

    if   (guess < secret): stdio.writeln('Too low')
    elif (guess > secret): stdio.writeln('Too high')
    else:                  stdio.writeln('You win!')
```

secret	Задуманное значение
guess	Предположение пользователя

Эта программа создает случайное целое число от 1 до 1 000 000, а затем многократно читает пользовательские предположения со стандартного устройства ввода. Она выводит на стандартное устройство вывода сообщения `Too low` (Мало) или `Too high` (Много) соответственно, в ответ на каждое предположение. Она выводит `You win!` (Вы выиграли!), когда предположение пользователя правильно. Вы вполне можете получить ответ `You win!` менее чем за 20 попыток.

```
% python twentyquestions.py
I am thinking of a secret number between 1 and 1000000
What is your guess? 500000
Too high
What is your guess? 250000
Too low
What is your guess? 375000
Too high
What is your guess? 312500
Too high
What is your guess? 300500
Too low
...
```

Фактически числа редко вводят одно за другим на стандартном устройстве ввода. Обычно входные данные сохраняют в файл, как показано в программе 1.5.3.

Конечно, программа `average.py` проста, но она демонстрирует новые возможности в программировании: используя стандартное устройство ввода, обрабатывать неограниченные объемы данных. Как будет продемонстрировано, это эффективный подход для создания многочисленных приложений обработки данных.

Стандартное устройство ввода — это существенный шаг вперед от использовавшейся ранее модели аргумента командной строки по двум причинам, проиллюстрированным в программах `twentyquestions.py` и `average.py`. Во-первых, мы можем взаимодействовать со своей программой, а с аргументами командной строки можем предоставить программе данные только *перед* началом выполнения. Во-вторых, можно читать большое количество данных, а с аргументами командной строки можно ввести только те значения, которые помещаются в командной строке. Действительно, как иллюстрирует программа `average.py`, объем обрабатываемых программой данных потенциально неограничен и многие программы используют эту возможность. Третье преимущество стандартного устройства ввода в том, что операционная система позволяет изменять источник стандартного ввода, избавляя вас от необходимости осуществить весь ввод самостоятельно. Далее мы рассмотрим механизмы, позволяющие использовать эту возможность.

Перенаправление и пересылка. Для многих приложений ввод исходных данных в поток стандартного ввода из окна терминала ненадежен, поскольку он ограничивается объемом данных, которые мы можем ввести вручную (и нашей скоростью ввода с клавиатуры). Точно так же выводимою потоком стандартного устройства информацию нередко желательно сохранить для последующего использования. Для решения проблемы этих ограничений сосредоточимся на том, что стандартное устройство ввода — это только *абстракция*, программа просто ждет свой ввод и никак не зависит от источника входного потока. Стандартное устройство вывода — это подобная абстракция. Мощь этих абстракций проистекает из способности (через операционную систему) определять для стандартных потоков ввода и вывода *различные* источники, такие как файл, сеть или другая программа. Все современные операционные системы реализуют эти механизмы.

Перенаправление стандартного вывода в файл. Добавление простой директивы в команду запуска программы позволяет *перенаправить* (redirect) ее стандартный вывод в файл для постоянного хранения или последующего ввода другой программы. Например, команда

```
% python randomseq.py 1000 > data.txt
```

Программа 1.5.3. Среднее потока чисел (average.py)

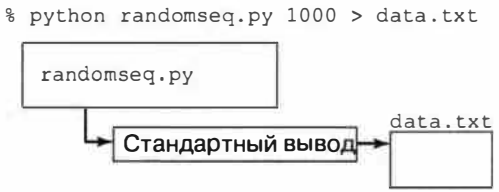
```
import stdio

total = 0.0
count = 0
while not stdio.isEmpty():
    value = stdio.readFloat()
    total += value
    count += 1
avg = total / count

stdio.writeln('Average is ' + str(avg))
```

Эта программа читает числа типа `float` из потока стандартного устройства ввода, пока не встретится конец файла. Затем она выводит на стандартное устройство вывода их среднее значение. С точки зрения программы нет никакого ограничения на размер входного потока. Представленные ниже команды, после первой, используют перенаправление и пересылку (обсуждается в следующем подразделе), чтобы предоставить 100 000 чисел программе `average.py`.

```
% python average.py
10.0 5.0 6.0
3.0
7.0 32.0
<Ctrl-d>
Average is 10.5

% python ramdomseq.py 1000 > data.txt
% python average.py < data.txt
Average is 0.510473676174824

% python randomseq.py 1000 | python average.py
Average is 0.50499417963857
```

определяет, что поток стандартного вывода должен выводиться не в окно терминала, а записываться в текстовый файл `data.txt`. Каждый вызов функции `stdio.write()`, `stdio.writeln()` или `stdio.writef()` добавляет текст в конец этого файла. В данном случае получается файл, содержащий 1 000 случайных значений. В окне терминала никакого вывода не будет: он идет непосредственно в файл, указанный после символа >. Так можно сохранить информацию для последующего использования. Обратите внимание: нам даже никак не пришлось изменять

программу 1.5.1 (randomseq.py) для применения этого механизма, он полностью полагается на использование абстракции стандартного вывода и не затрагивает нашу реализацию этой абстракции. Вы можете использовать этот механизм для сохранения вывода любой своей программы. Потратив существенные усилия на получение результата, мы зачастую хотим сохранить его для последующего использования. В современных системах вы можете сохранять небольшие фрагменты информации, используя копирование и вставку или некий подобный механизм, предоставляемый операционной системой, но для больших объемов данных копирование и вставка неудобны. Перенаправление, напротив, специально предназначено для облегчения сохранения больших объемов данных.

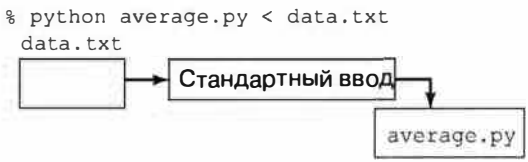


Перенаправление стандартного вывода в файл

Перенаправление из файла в поток стандартного ввода. Стандартный ввод можно перенаправить так, чтобы программа читала данные из файла, а не из терминального приложения:

```
% python average.py < data.txt
```

Эта команда читает последовательность чисел из файла data.txt и вычисляет их среднее значение. А именно: символ < — это директива, указывающая операционной системе реализовать поток стандартного ввода при чтении из текстового файла data.txt вместо ожидания ввода чего-либо от пользователя в окне терминала. Когда программа вызывает функцию stdio.readFloat(), операционная система читает значение из файла. Файл data.txt может был создан любым приложением, а не только программой Python. Текстовые файлы могут создавать многие приложения на вашем компьютере. Средства перенаправления из файла в поток стандартного ввода позволяют создать *код управляемый данными* (data-driven code), в котором обрабатываемые программой данные можно изменять без необходимости изменять саму программу вообще. Достаточно сохранять данные в файлах и создавать программы, читающие их из потока стандартного ввода.



Перенаправление из файла в поток стандартного ввода

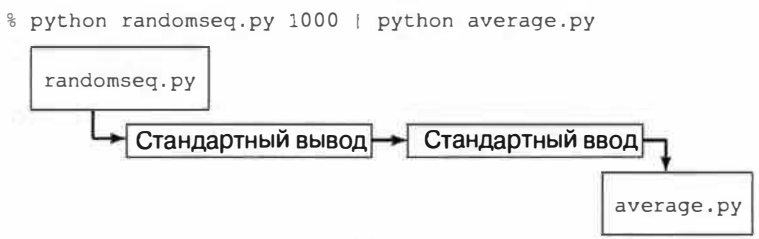
Соединение двух программ. Самый гибкий способ реализации абстракций стандартного ввода и вывода заключается в определении их реализации нашими собственными программами! Этот механизм называется *пересылкой* (piping). Например, команда

```
% python randomseq.py 1000 | python average.py
```

определяет, что поток стандартного вывода программы `randomseq.py` и поток стандартного ввода `average.py` — это *тот же* поток. В результате программа `randomseq.py` будет как бы вводить создаваемые числа в окне терминала во время выполнения программы `average.py`. Этот пример имеет тот же эффект, что и следующая последовательность команд:

```
% python randomseq.py 1000 > data.txt
% python average.py < data.txt
```

Но при пересылке файл `data.txt` не создается. Это различие весьма существенно, поскольку оно устраняет другое ограничение на размер потоков ввода и вывода, которые мы можем обработать. Например, мы могли заменить значение `1000` в нашем примере на `1000000000` даже при том, что на нашем компьютере могло бы не быть достаточно места для сохранения миллиарда чисел (однако на их обработку действительно понадобится много *времени*). Когда программа `randomseq.py` вызывает функцию `stdio.writeln()`, строка добавляется в конец потока, а когда программа `average.py` вызывает функцию `stdio.readFloat()`, строка удаляется из начала потока. Точность синхронизации обеспечивает операционная система: она выполняет программу `randomseq.py`, пока она не создаст некий вывод, а затем запускает программу `average.py`, чтобы использовать этот вывод, либо она могла бы выполнять программу `average.py`, пока она не нуждается в некоем выводе, а затем запустить программу `randomseq.py`, пока она не создаст необходимый вывод. В результате получается то же самое, но наши программы освобождаются от заботы об этих деталях, поскольку они работают исключительно с абстракциями стандартного ввода и вывода.



Пересылка вывода одной программы на ввод другой

Фильтры. Пересылка — это основное средство первоначальной системы Unix начала 1970-х годов, но оно все еще выживает на современных системах, поскольку это простая абстракция связи между несовместимыми программами. Доказательство

мощи этой абстракции — тот факт, что многие программы Unix все еще используются сегодня и обрабатывают файлы в тысячи и миллионы раз больше, чем предполагали их авторы. Мы можем общаться с другими программами Python, обращаясь к их функциям, но стандартные ввод и вывод позволяют общаться с программами, написанными в другое время и, возможно, на других языках. Используя стандартный ввод и вывод, мы получаем простой интерфейс к внешнему миру.

Во многих случаях программы можно рассматривать как *фильтр* (filter), в некотором роде преобразующий поток стандартного ввода в поток стандартного вывода, с пересылкой в качестве механизма связи программ. Например, программа 1.5.4 (`rangefilter.py`) получает два аргумента командной строки и выводит их в поток стандартного вывода со стандартного ввода, находящегося в определенном диапазоне. Можно предположить, что стандартный ввод — это данные измерений с некоего прибора, а фильтр используется для отсечения данных вне диапазона, представляющего интерес для эксперимента.

Несколько стандартных фильтров, разработанных для Unix, все еще используются (иногда под другими именами) как команды в современных операционных системах. Например, фильтр сортировки читает строки из стандартного ввода и выводит их в поток стандартного вывода в сортированном порядке:

```
% python randomseq.py 9 | sort
0.0472650078535
0.0681950168757
0.0967410236589
0.0974385525393
0.118855769243
0.46604926859
0.522853708616
0.599692836211
0.685576779833
```

Сортировка обсуждается в разделе 4.2. Второй весьма полезный фильтр, `grep`, выводит строки со стандартного ввода в соответствии с заданным шаблоном. Например, если ввести

```
% grep lo < rangefilter.py
```

то вы получаете все строки в файле `rangefilter.py`, содержащие слово `'lo'`:

```
lo = int(sys.argv[1])
    if (value >= lo) and (value <= hi):
```

Программисты нередко используют такие инструментальные средства, как `grep`, для быстрого напоминания имен переменных или подробностей используемого языка. Третий полезный фильтр, `more`, читает данные стандартного ввода (или файла, указанного в аргументе командной строки) и отображает его в окне терминала по одному экрану за раз. Например, если ввести

```
% python randomseq.py 1000 | more
```

ваше окно терминала заполнится множеством чисел, но фильтр `more` остановит их вывод и будет ждать, пока вы не нажмете клавишу <пробел>, чтобы продолжить вывод следующего экрана. Термин *фильтр* (filter), возможно, вводит в заблуждение: он предназначался для описания таких программ, как `rangefilter.py`, выводящих лишь некую часть потока стандартного ввода в поток стандартного вывода, но теперь он обычно используется для описания любых программ, читающих стандартный ввод и пишущих в стандартный вывод.

Программа 1.5.4. Простой фильтр (`rangefilter.py`)

```
import sys
import stdio

lo = int(sys.argv[1])
hi = int(sys.argv[2])

while not stdio.isEmpty():
    # Обработать одно целое число.
    value = stdio.readInt()
    if (value >= lo) and (value <= hi):
        stdio.write(str(value) + ' ')
stdio.writeln()
```

`lo`	Нижняя граница диапазона
`hi`	Верхняя граница диапазона
`value`	Текущее число

Эта программа получает из командной строки целочисленные аргументы `lo` и `hi`, а затем читает со стандартного ввода целые числа, пока не достигнет конца файла. Программа выводит в стандартный вывод каждое из тех целых чисел, которые находятся в диапазоне от `lo` до `hi` включительно. Таким образом, эта программа — фильтр (см. текст). Для длины потоков нет никакого предела.

```
% more rangedata.txt
3 1 4 1 5 9 2 6 5 3 5 8 9 7 9 3 2 3 8 4 6 2 6 4 3 3 8 3 2 7 9

% python rangefilter.py 5 9 < rangedata.txt
5 9 6 5 5 8 9 7 9 8 6 6 8 7 9

% python rangefilter.py 100 400
358 1330 55 165 689 1014 3066 387 575 843 203 48 292 877 65 998
358 165 387 203 292
<Ctrl-d>
```

Несколько потоков. Для решения многих задач необходимы программы, способные получать ввод из нескольких источников и/или осуществлять вывод

для нескольких получателей. В разделе 3.1 мы обсудим собственные модули `instream.py` и `outstream.py` (обобщенные в модуле `stdio.py`), учитывающие возможность нескольких потоков ввода и вывода. Эти модули включают средства перенаправления потоков не только в и из файлов, но также и из произвольных веб-страниц.

Обработка больших объемов информации играет основную роль во многих приложениях. Ученому, возможно, понадобится проанализировать данные, собранные в процессе экспериментов, брокеру может понадобиться проанализировать информацию о недавних финансовых транзакциях, а студенту может понадобиться поддержка для его коллекции музыки и фильмов. В этих и множестве других подобных приложениях управляемые данными программы — общепринятая норма. Стандартный вывод, стандартный ввод, перенаправление и пересылка позволяют решать такие задачи нашим программам Python. Мы можем собрать данные в файлы на компьютере через веб или любое из стандартных устройств, а затем использовать перенаправление и пересылку для передачи этих данных в наши программы.

Стандартное графическое устройство. До этого момента наши абстракции ввода-вывода ограничивались исключительно текстом. Теперь мы вводим абстракцию для создания рисунков. Этот модуль удобен и позволяет использовать визуальные средства для такого отображения информации, которое невозможно с использованием только текста.

Стандартное графическое устройство очень простое: вообразите абстрактное устройство, способное рисовать линии и точки на двумерном холсте, а затем отображающее этот холст на экране в окне стандартного графического устройства. Устройство способно реагировать на команды, отдаваемые программой в виде вызовов функций из модуля `stddraw`.

API модуля состоят из двух видов функций: *графические функции* (drawing function), заставляющие устройство предпринять некое действие (такое, как вывод линии или точки), и *функции управления* (control function), контролирующие представление и устанавливающие параметры, такие как размер пера или масштаб координат.

Создание рисунка. Базовые графические функции описаны в API ниже. Как и функции стандартного ввода и вывода, графические функции почти самодокументированы: функция `stddraw.line()` рисует сегмент прямой линии, соединяющей две точки, координаты которых передаются в виде аргументов, а функция `stddraw.point()` рисует точку с центром в заданных координатах. Стандартный масштаб координат — это единичный квадрат (все координаты от 0 до 1). Точка (0.0, 0.0) — в нижнем левом углу, а точка (1.0, 1.0) — в верхнем правом, что соответствует первому квадранту Декартовой системы координат. Стандартно задано рисование черных линий и точек на белом фоне.

Функция управления `stddraw.show()` нуждается в дополнительном объяснении. Когда программа вызывает любую графическую функцию, такую как `stddraw.line()` или `stddraw.point()`, модуль `stddraw` использует абстракцию *фонового холста* (background canvas). Фоновый холст не отображается; он существует только в машинной памяти. Все точки, линии и т.д. рисуются на фоновом холсте, а не непосредственно в окне стандартного графического устройства. Только вызов функции `stddraw.show()` копирует рисунок из фонового холста в окно стандартного графического устройства, где он и отображается, пока пользователь не закроет окно обычного стандартного графического устройства, щелкнув на кнопке <Close> (Закрыть) в заголовке окна.

API для базовых функций с сайта книги, связанных со стандартным графическим устройством

Вызов функции	Описание
`stddraw.line(x0, y0, x1, y1)`	Рисует линию от точки (x0, y0) к точке (x1, y1)
`stddraw.point(x, y)`	Рисует точку (x, y)
`stddraw.show()`	Выводит рисунок в окно стандартного графического устройства (и ожидает, пока оно не будет закрыто пользователем)

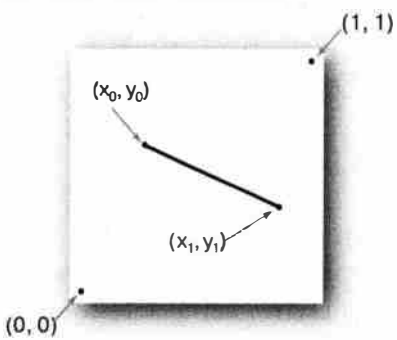
Так почему модуль `stddraw` использует фоновый холст? Основная причина в том, что использование двух холстов вместо одного повышает эффективность модуля `stddraw`. Поэтапное отображение сложного рисунка по мере его создания может быть чрезвычайно неэффективным на многих компьютерных системах. В компьютерной графике эта методика известна как *двойная буферизация* (double buffering).

Таким образом, чтобы использовать модуль `stddraw` в типичной программе, необходимо следующее:

- импортировать модуль `stddraw`;
- чтобы создать рисунок на фоновом холсте, используйте такие графические функции, как `stddraw.line()` и `stddraw.point()`;
- для отображения фонового холста в окне стандартного графического устройства до его закрытия используйте функцию `stddraw.show()`.

Следует помнить, что *весь рисунок создается на фоновом холсте*. Как правило, создающий рисунок код завершается вызовом функции `stddraw.show()`, поскольку только это позволяет увидеть созданное изображение.

```
import stddraw
stddraw.line(x0, y0, x1, y1)
stddraw.show()
```

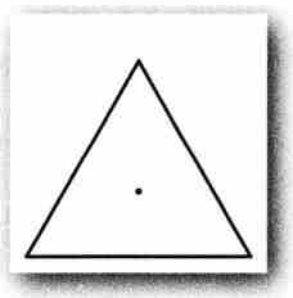


Теперь рассмотрим несколько примеров, открывающих новые возможности программирования и освобождающие от ограничений, связанных с использованием только текста.

Ваш первый рисунок. Эквивалентом программы "Hello, World" для графики является рисование треугольника с точкой внутри. Для формирования треугольника мы нарисуем три линии: от точки (0, 0) в левом нижнем углу к точке (1, 0), от нее к третьей точке (1/2,) и от этой точки назад, к точке (0, 0). И наконец, рисуем пятно в середине треугольника. Успешно загрузив и запустив программу `triangle.py` a_nd, вы готовы составлять собственные программы, рисующие фигуры, линии и точки. Эта способность буквально добавляет новую размерность к выводу, который вы можете создавать.

Когда вы используете компьютер для создания графики, результат получаете немедленно (рисунок) и можете совершенствовать и быстро улучшить свою программу. Компьютерные программы позволяют создавать рисунки, которые вы не смогли бы сделать вручную. В частности, вместо просмотра своих данных в виде чисел можно использовать изображения, которые намного выразительней. Обсудив несколько команд рисования, мы рассмотрим другие примеры.

```
import math
import stddraw

t = math.sqrt(3.0) / 2.0
stddraw.line(0.0, 0.0, 1.0, 0.0)
stddraw.line(1.0, 0.0, 0.5, t)
stddraw.line(0.5, t, 0.0, 0.0)
stddraw.point(0.5, t/3.0)
stddraw.show()
```



Ваш первый рисунок

Сохранение рисунка. Холст окна стандартного графического устройства можно сохранить в файл, что позволяет распечатать рисунок или поделиться им с другими людьми. Для этого щелкните правой кнопкой мыши где-нибудь на холсте окна (обычно во время бесконечного ожидания, поскольку ваша программа вызвала функцию `stddraw.show()`). После этого модуль `stddraw` отобразит файловое диалоговое окно, позволяющее задать имя файла. Затем, после ввода имени файла, щелкните в диалоговом окне на кнопке Save (Сохранить), и модуль `stddraw` сохранит холст окна в файл с указанным именем. Имя файла должно завершаться либо расширением `.jpg` (чтобы сохранить холст окна в формате JPEG), либо `.png` (чтобы сохранить холст окна в формате "Portable Network Graphics"). Созданные графическими программами

рисунки этой главы были сохранены в файлы с использованием именно этого механизма.

Команды управления. Стандартная система координат стандартного графического устройства — это единичный квадрат, но нам зачастую необходимо рисовать в иных масштабах. Например, типичная ситуация: нужно использовать координаты в некоем диапазоне по координате x, или y, или по обеим. Кроме того, нередко необходимо рисовать линии различной толщины и точки разного размера, отличного от стандартного. Для решения этих задач модуль `stddraw` предоставляет следующие функции.

API для функций с сайта книги, контролирующих и устанавливающих параметры рисунка

Вызов функции	Описание
`stddraw.setCanvasSize(w, h)`	Установить размер холста `w` × `h` пикселей (стандартно значениями `w` и `h` является `512`)
`stddraw.setXscale(x0, x1)`	Установить диапазон `(x0, x1)` холста по координате x (стандартные значения `x0` и `x1` — `0` и `1` соответственно)
`stddraw.setYscale(y0, y1)`	Установить диапазон `(y0, y1)` холста по координате y (стандартные значения `y0` и `y1` — `0` и `1` соответственно)
`stddraw.setPenRadius(r)`	Установить радиус пера `r` (стандартное значение `0.005`)

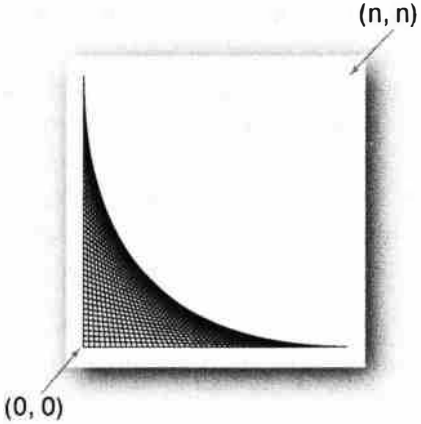
Примечание: при значении `0` радиуса пера точки и линии будут иметь минимально возможный размер.

Например, при вызове функции `stddraw.setXscale(0, n)` вы указываете графическому устройству использовать координаты `x` от `0` до `n`. Обратите внимание: последовательность из двух вызовов

```
stddraw.setXscale(x0, x1)
stddraw.setYscale(y0, y1)
```

устанавливает координаты рисунка в пределах *рамки* с левым нижним углом (x_0, y_0) и правым верхним углом (x_1, y_1). Если вы используете целочисленные координаты, Python преобразует их в тип `float`, как и ожидается. Рисунок ниже демонстрирует удобство масштабирования. Масштабирование является самым простым из преобразований, общепринятых в графике. Некоторые из рассматриваемых в этой главе приложений типичны — мы используем масштабирование прямым способом для обеспечения соответствия рисунков нашим данным.

```
import stddraw
n = 50
stddraw.setXscale(0, n)
stddraw.setYscale(0, n)
for i in range(n+1):
    stddraw.line(0, n-i, i, 0)
stddraw.show()
```



Масштабирование целочисленных координат

Перо круглое, поэтому, установив радиус пера r и нарисовав точку, вы фактически получаете круг радиусом *r*. Кроме того, линия имеет толщину *2r* и округлые концы. Стандартный радиус пера — 0,005, и масштабирование координат на него не влияет. Этот стандартный радиус пера составляет 1/200 ширины стандартного окна, так что, нарисовав 100 точек на равном расстоянии вдоль горизонтальной или вертикальной линии, можно рассмотреть отдельные круги, но если нарисовать 200 таких точек, то результат будет выглядеть, как линия. Вызов функции `stddraw.setPenRadius(.025)` устанавливает толщину линий и размер точек в пять раз больше стандарта 0,005. Для рисования точек минимально возможным радиусом (один пиксель на дисплее) установите радиус пера 0,0.

Фильтрация данных, передаваемых на стандартное графическое устройство. Один из самых простых случаев применения стандартного графического устройства — это вывод в виде рисунка отфильтрованных данных, полученных со стандартного устройства ввода. Программа 1.5.5 (`plotfilter.py`) представляет пример такого фильтра: он читает последовательность точек, определенных координатами (`x`, `y`), и рисует пятна в каждой точке. Примем также соглашение, что первые четыре прочитанных со стандартного ввода числа определяют рамку, чтобы можно было масштабировать рисунок, без необходимости предпринимать дополнительные действия по выяснению его текущего размера по всем имеющимся точкам (этот вид соглашения типичен для таких файлов данных).

Графическое представление точек намного выразительнее (и намного компактнее), чем просто числа сами по себе или любое их представление, которое мы могли бы создать, ограничиваясь возможностями стандартного вывода, как в наших программах до сих пор. Вывод изображения программой, подобной `plotfilter.py`, намного упрощает вывод свойств городов (такой, например, как кластеризация центров населения) и повышает их наглядность (по сравнению со списком координат). Всякий раз, когда приходится обрабатывать данные, представляющие физический мир, визуальное изображение, вероятно, будет одним из лучших способов отображения вывода. Программа 1.5.5 иллюстрирует, насколько легко можно создать такое изображение.

Программа 1.5.5. Стандартный ввод и фильтрация рисунка (`plotfilter.py`)

```
import stddraw
import stdio

# Прочитать и установить масштабы x и y.
x0 = stdio.readFloat()
y0 = stdio.readFloat()
x1 = stdio.readFloat()
y1 = stdio.readFloat()
stddraw.setXscale(x0, x1)
stddraw.setYscale(y0, y1)

# Читать и рисовать точки.
stddraw.setPenRadius(0.0)
while not stdio.isEmpty():
    x = stdio.readFloat()
    y = stdio.readFloat()
    stddraw.point(x, y)

stddraw.show()
```

`x0`	Левая граница
`y0`	Нижняя граница
`x1`	Правая граница
`y1`	Верхняя граница
`x, y`	Текущая точка

Эта программа читает масштабы x и y из стандартного ввода и настраивает холст соответственно. Затем она читает из стандартного ввода точки, пока не будет достигнут конец файла, и выводит их на стандартное графическое устройство. В файле `usa.txt` на сайте книги есть координаты американских городов с населением более 500 человек. Некоторые данные, такие как в файле `usa.txt`, куда понятнее визуально.

```
% python plotfilter.py < usa.txt
```

Вывод графика функции. Еще один важный случай применения модуля `stddraw` — это вывод экспериментальных данных или значений математической функции. Предположим, например, что необходимо нарисовать значения функции $y = \sin(4x) + \sin(20x)$ в интервале $[0, \pi]$. Для решения этой задачи формируется прототип *выборки*: поскольку интервал содержит бесконечное количество точек, необходимо обойтись оценкой функции за конечное количество точек в пределах интервала. Для построения графика функции выбирается набор значений по координате x, а затем вычисляются значения функции по координате y для каждого значения x. Вывод графика функции в результате соединения линиями последовательности точек называется *кусочно-линейной аппроксимацией* (piecewise linear approximation). Простейший способ получения регулярных значений по оси x — выбрать загодя размер выборки, а затем разделить пространство по координате x на размер выборки. Чтобы гарантировать попадание выводимых значений в видимую часть холста, мы масштабируем ось x в соответствии с интервалом, а ось y — в соответствии с максимальным и минимальным значениями функции в пределах интервала. Программа 1.5.6 (`functiongraph.py`) демонстрирует код Python для этого процесса.

Гладкость кривой зависит от свойств функции и размера выборки. Если размер выборки слишком мал, визуализация функции может быть очень не точной (она может быть не очень гладкой и даже пропустить некоторые существенные детали, как показано в примере далее); если выборка слишком велика, создание графика может занять много времени, так как некоторые функции вычисляются довольно долго. (В разделе 2.4 будет представлен эффективный метод точного рисования гладкой кривой.) Ту же методику можно использовать для рисования изображения графика любой функции: выбираете интервал по оси x, в котором необходимо отобразить функцию, вычислить значения функции, равномерно расположенные в этом интервале, и сохранить их в массиве, определить и установить масштаб по оси y и нарисовать сегменты линии.

Контурные и заполненные фигуры. Книжный модуль `stddraw` включает также функции рисования кругов, прямоугольников и произвольных многоугольников. Каждая форма определяет контур. Когда в имени функции указано только название фигуры, она отображает лишь контур формы. Когда имя начинается с `filled`, указанная далее по имени форма выводится заполненной. Как обычно, доступные функции собраны в API и представлены ниже.

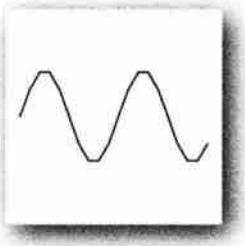
Программа 1.5.6. Вывод графика функции (`functiongraph.py`)

```
import math
import sys
import stdarray
import stddraw

n = int(sys.argv[1])

x = stdarray.create1D(n+1, 0.0)
y = stdarray.create1D(n+1, 0.0)
for i in range(n+1):
    x[i] = math.pi * i / n
    y[i] = math.sin(4.0*x[i]) + math.sin(20.0*x[i])
stddraw.setXscale(0, math.pi)
stddraw.setYscale(-2.0, +2.0)
for i in range(n):
    stddraw.line(x[i], y[i], x[i+1], y[i+1])

stddraw.show()
```

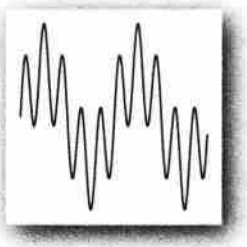
`n`	Количество выборок
`x[]`	Координаты по оси x
`y[]`	Координаты по оси y

Эта программа получает целочисленный аргумент командной строки n, а затем выводит кусочно-линейную аппроксимацию функции $y = \sin(4x) + \sin(20x)$ при выборке функции в $n+1$ точках между $x = 0$ и $x = \pi$ при рисовании n линейных сегментов. Этот пример иллюстрирует потребность в тщательном выборе количества выборок; только при 20 выборках мы пропускаем большинство колебаний кривой.

```
% python functiongraph.py 20
```



```
% python functiongraph.py 200
```



API для функций с сайта книги, связанных с рисованием геометрических фигур

Вызов функции	Описание
`stddraw.circle(x, y, r)`	Рисует круг радиусом r с центром в точке (x, y)
`stddraw.square(x, y, r)`	Рисует квадрат $2r \times 2r$ с центром в точке (x, y)
`stddraw.rectangle(x, y, w, h)`	Рисует прямоугольник $w \times h$ с левой нижней конечной точкой (x, y)
`stddraw.polygon(x, y)`	Рисует многоугольник по точкам $(x[i], y[i])$

Примечание: функции `filledCircle()`, `filledSquare()`, `filledRectangle()` и `filledPolygon()` рисуют соответствующие заполненные формы (а не только контуры).

Аргументы функций `stddraw.circle()` и `stddraw.filledCircle()` определяют радиус круга r и его центр (x, y); аргументы функций `stddraw.square()` и `stddraw.filledSquare()` определяют длину стороны квадрата $2r$ и его центр (x, y); а аргументы функций `stddraw.polygon()` и `stddraw.filledPolygon()` определяют последовательность точек, которые, будучи соединены линиями, образуют фигуру (последняя точка соединяется с первой). Если необходима фигура, отличная от квадрата или круга, используйте одну из этих функций. Чтобы проверить свое понимание, попробуйте выяснить, что рисует следующий код, прежде чем прочитать ответ:

```
xd = [x-r, x, x+r, x]
yd = [y, y+r, y, y-r]
stddraw.polygon(xd, yd)
```

Ответ вы никогда не узнали бы, поскольку рисунок останется на фоновом холсте, ведь вызова функции `stddraw.show()` здесь нет. Если бы такой вызов был, то получился бы ромб (повернутый четырехугольник) с центром в точке (x, y). Несколько других примеров кода, рисующего контурные и заполненные фигуры, представлены далее.

Текст и цвет. Иногда различные элементы рисунка следует выделить или подписать. В модуле `stddraw` есть функция для вывода текста, две функции для установки параметров текста и еще одна функция для изменения его цвета. В этой книге мы используем данные средства не очень часто, но они могут быть весьма полезны, особенно для рисунков на экране компьютера. На сайте книги вы найдете много примеров их использования.

API для функций с сайта книги для текста и цвета в рисунках

Вызов функции	Описание
`stddraw.text(x, y, s)`	Рисует строку s с центром в точке (x, y)
`stddraw.setPenColor(color)`	Устанавливает цвет пера `color` (стандартное значение `stddraw.BLACK`)
`stddraw.setFontFamily(font)`	Устанавливает семейство шрифтов `font` (стандартное значение `'Helvetica'`)
`stddraw.setFontSize(size)`	Устанавливает размер шрифта `size` (стандартное значение 12)

Используемые в этом коде типы цветов и шрифтов подробно описаны в разделе 3.1, а пока достаточно знать, что в модуле `stddraw` для цветов определены константы `BLACK`, `BLUE`, `CYAN`, `DARK_GRAY`, `GRAY`, `GREEN`, `LIGHT_GRAY`, `MAGENTA`, `ORANGE`, `PINK`, `RED`, `WHITE` и `YELLOW`. Например, вызов функции `stddraw.setPenColor(stddraw.GRAY)` изменит цвет на серый. Стандартный цвет — `stddraw.BLACK`. Стандартный шрифт вполне подходит для большинства рисунков (информацию об использовании других шрифтов см. на сайте книги). Например, их можно использовать для аннотирования графиков функций и выделения неких значений. Может оказаться весьма полезным составить подобные функции для аннотирования других частей рисунков.

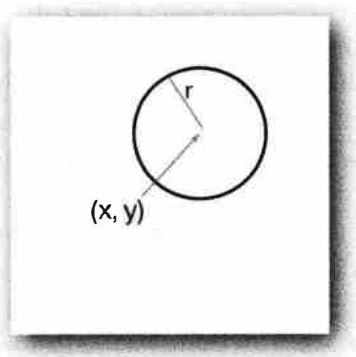
Фигуры, цвет и текст — это базовые инструментальные средства, позволяющие создавать безмерное разнообразие изображений, однако использовать их следует рассудительно. Использование графических элементов обычно усложняет проект, и команды из нашего модуля `stddraw` слабоваты по стандартам современных графических библиотек, поэтому для создания красивых изображений, вероятно, понадобится больше кода, чем можно было бы ожидать сначала.

Анимация. Если предоставить аргумент для вызова функции `stddraw.show()`, то этот вызов не должен быть последним действием программы: он скопирует фоновый холст в окно стандартного графического устройства, а затем будет ждать указанное количество миллисекунд. Как будет скоро продемонстрировано, эта возможность (вместе с возможностью стирать или *очищать* (clear) фоновый холст) обеспечивает безграничные возможности для создания интересных эффектов на базе динамического изменения изображения в окне стандартного графического устройства. Такие эффекты способны существенно улучшить визуализацию. Мы приведем простой пример, а множество других примеров, которые, вероятно, захватят ваше воображение, приведено на сайте книги.

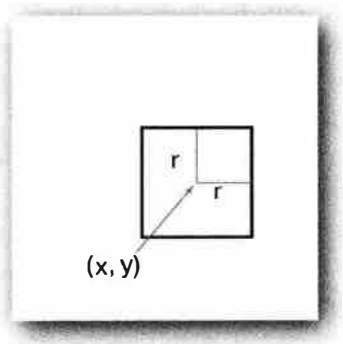
API для функций с сайта книги, связанных с анимацией

Вызов функции	Описание
`stddraw.clear(color)`	Очищает фоновый холст и окрашивает цветом `color`
`stddraw.show(t)`	Отображает рисунок в окне стандартного графического устройства и ожидает `t` миллисекунд

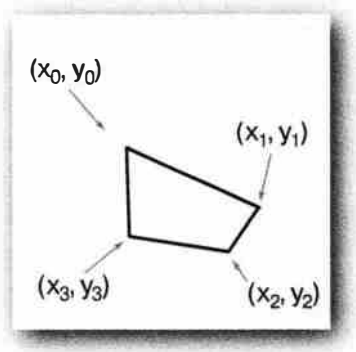
```
import stddraw
stddraw.circle(x, y, r)
stddraw.show()
```



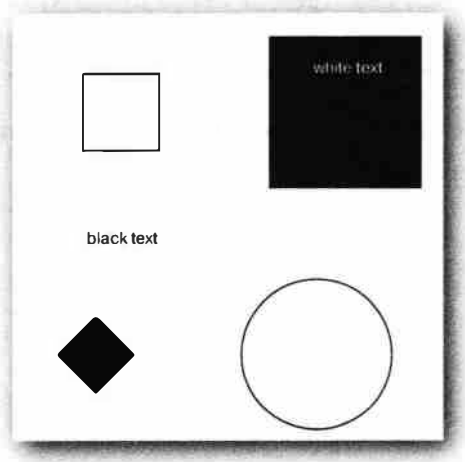

```
import stddraw
stddraw.square(x, y, r)
stddraw.show()
```




```
import stddraw
x = [x0, x1, x2, x3]
y = [y0, y1, y2, y3]
stddraw.polygon(x, y)
stddraw.show()
```



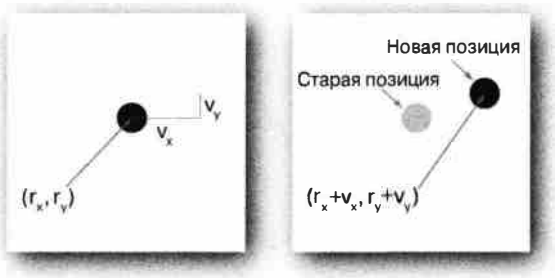

```
import stddraw
stddraw.square(.2, .8, .1)
stddraw.filledSquare(.8, .8, .2)
stddraw.circle(.8, .2, .2)
xd = [.1, .2, .3, .2]
yd = [.2, .3, .2, .1]
stddraw.filledPolygon(xd, yd)
stddraw.text(.2, .5, 'black text')
stddraw.setPenColor(stddraw.WHITE)
stddraw.text(.8, .8, 'white text')
stddraw.show()
```



Примеры стандартных рисунков с использованием форм и цветов

Прыгающий мяч. Программа уровня "Hello, World" для анимации — это черный шар, выглядящий перемещающимся на холсте. Предположим, что шар (мяч) находится в позиции (r_x, r_y) и мы хотим создать впечатление его перемещения в новую позиции неподалеку, такую, например, как $(r_x + 0.01, r_y + 0.02)$. Для этого нужно выполнить три этапа.

- Очистить фоновый холст.
- Нарисовать черный шар в новой позиции.
- Отобразить рисунок и ожидать в течение короткого времени.



Модель двигающегося шара

Чтобы создать иллюзию движения, повторим эти этапы для целой последовательности позиций (формирующих в данном случае прямую линию). Аргумент функции `stddraw.show()` определяет короткое время отображения и контролирует наблюдаемую скорость.

Программа 1.5.7 (`bouncingball.py`) реализует эти этапы, создавая иллюзию мяча, прыгающего в коробке 2×2, с центром в исходной точке. Текущая позиция мяча (r_x, r_y), а новая позиция вычисляется на каждом этапе при добавлении v_x к r_x и v_y к r_y. Поскольку расстояние (v_x, v_y) фиксировано, мяч каждый раз перемещается на единицу, представляющую скорость. Чтобы мяч остался в области рисунка, мы моделируем отскок мяча от стен согласно законам упругого соударения. Реализовать этот эффект просто: когда мяч встречает вертикальную стену, достаточно изменить скорость в направлении X с v_x на $-v_x$, а когда мяч встречает горизонтальную стену, достаточно изменить скорость в направлении Y с v_y на $-v_y$. Конечно, чтобы увидеть движение, можно загрузить код с сайта книги и запустить его на вашем компьютере.

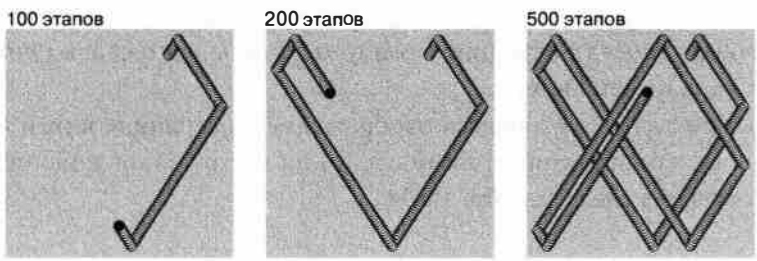
Поскольку показать движущееся изображение на странице книги невозможно, мы немного изменили программу `bouncingball.py` так, чтобы перемещающийся мяч оставлял след (см. упр. 1.5.34).

Программа 1.5.7. Прыгающий мяч (bouncingball.py)

```
import stddraw

stddraw.setXscale(-1.0, 1.0)
stddraw.setYscale(-1.0, 1.0)

DT = 20.0
RADIUS = 0.05
rx = 0.480
ry = 0.860
vx = 0.015
vy = 0.023

while True:
    # Обновить позицию мяча и нарисовать его.
    if abs(rx + vx) + RADIUS > 1.0: vx = -vx
    if abs(ry + vy) + RADIUS > 1.0: vy = -vy
    rx = rx + vx
    ry = ry + vy

    stddraw.clear(stddraw.GRAY)
    stddraw.filledCircle(rx, ry, RADIUS)
    stddraw.show(DT)
```

DT	Время ожидания
RADIUS	Радиус мяча
rx, ry	Позиция
vx, vy	Скорость

Эта программа рисует перемещающийся шар на стандартном графическом устройстве. Таким образом, она моделирует движение упругого мяча в коробке. Мяч отскакивает от стенки согласно законам упругого соударения. Период ожидания в 20 миллисекунд обеспечивает постоянное изображение черного шара на экране, даже при том, что большинство пикселей шара должно сменить цвет с черного на белый. Если изменить этот код так, чтобы задавать время ожидания `dt` как аргумент командной строки, то можно контролировать скорость мяча. Ниже представлены изображения следов перемещения мяча, созданные модифицированной версией этого кода, где вызов функции `stddraw.clear()` осуществляется вне цикла (см. упр. 1.5.34).



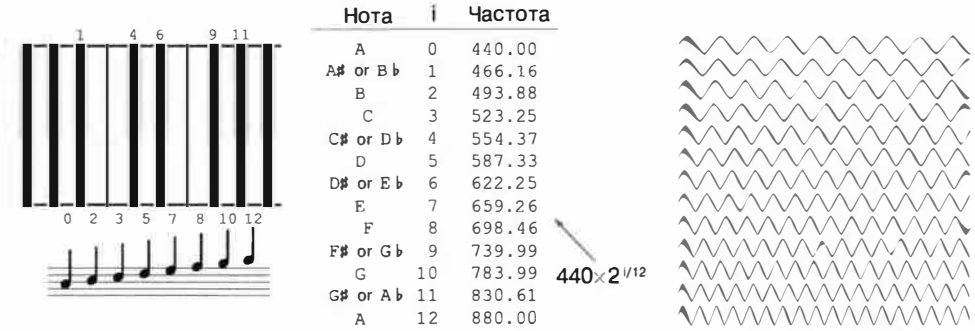
Для ознакомления с компьютерной анимацией попробуйте изменять различные параметры в программе `bouncingball.py`: нарисуйте больший мяч, заставьте его двигаться быстрее или медленнее, а также оцените различие между скоростью в моделировании и наблюдаемой скоростью на вашем экране. Для максимальной гибкости вы могли бы изменить программу `bouncingball.py` так, чтобы получать все эти параметры как аргументы командной строки.

Стандартное графическое устройство существенно улучшает нашу модель программирования, добавляя в нее компонент "картина стоит тысячи слов". Это естественная абстракция, которую можно использовать для лучшего представления программ внешнему миру. Она позволяет легко создавать графики функций или визуально представлять научные и технические данные. Данный подход нередко используется в этой книге. Все время, потраченное на изучение примеров программ на нескольких следующих страницах, будет выгодной инвестиций. На сайте книги можно найти много полезных примеров и упражнений, позволяющих реализовать ваш творческий потенциал при использовании модуля `stddraw`, удовлетворяющего различным требованиям. Можно ли нарисовать многолучевую звезду? Может ли прыгающий мяч прыгать фактически (добавить гравитацию)? Вы будете удивлены, но решить эти и другие задачи довольно легко.

Стандартное аудиоустройство. И наконец, рассмотрим последний пример фундаментальной абстракции вывода — модуль `stdaudio`, позволяющий проигрывать, манипулировать и синтезировать звук. Звуком на компьютере вас, конечно, не удивить, но теперь вы можете сами создавать программы, использующие его. В то же время предстоит изучить несколько концепций, лежащих в основе важнейшей области информатики и теории вычислений, — *цифровой обработки сигнала* (digital signal processing). Мы лишь поверхностно затронем эту тему, но вас может удивить простота основных концепций.

Концертное ля. Звук — восприятие вибрации молекул, в частности, вибрации наших барабанных перепонок. Поэтому колебание — это ключ к пониманию звука. Возможно, проще всего начать с рассмотрения музыкальной ноты *ля* (A) после *ноты до средней октавы* (middle C), известной как *концертное ля* (concert A). Эта нота не более чем синусоидальная волна частотой 440 колебаний в секунду. Функция sin(t) повторяет свои значения каждые 2π единиц, если t измеряется в секундах, то, построив график функции $\sin(2\pi t \times 440)$, мы получим кривую, совершающую 440 колебаний в секунду. Когда вы играете ноту *ля*, тронув гитарную струну, или продув воздух через трубу, или заставив вибрировать конус динамика, создается синусоидальная звуковая волна, которую ваше ухо распознает как концертное *ля*. Частота измеряется в *Герцах* (циклы в секунду). При удвоении или делении частоты на два вы перемещаетесь на одну октаву вверх или вниз. Например, звук частотой 880 Герц на одну октаву выше концертного *ля*, а частотой 110 Герц — на две октавы ниже. Для справки: человек слышит звук в частотном диапазоне примерно 20–20 000 Герц. Амплитуда звука (значение по оси y) соответствует громкости. Мы

выводим свои кривые в диапазоне от –1 до +1 и подразумеваем, что уровни записи и воспроизведения звука всех устройств настроены как следует, а в дальнейшем контролируются вами при помощи регулятора громкости.



Ноты, числа и волны

Другие ноты. Простая математическая формула характеризует другие ноты хроматической гаммы. В хроматической гамме 12 нот, равно отстоящих на логарифмической (основание 2) шкале. Мы получаем i-ю ноту выше данной ноты умножением ее частоты на $i/12$ в степени числа 2. Другими словами, частота каждой ноты хроматической гаммы — это частота предыдущей ноты гаммы, умноженная на 2 в степени одной двенадцатой (примерно 1,06). Этой информации достаточно для создания музыки! Например, чтобы проиграть мелодию *Frère Jacques*, достаточно сыграть каждую из нот *ля*, *си*, *до#*, *ля*, создавая синусоидальные волны соответствующих частот в течение приблизительно половины секунды, а затем повторять ту же схему.

Выборка. Для цифрового звука мы представляем кривую с использованием равномерной выборки, точно так же, как и при рисовании графиков функций. Чтобы получить точное представление кривой, выборку следует производить достаточно часто, для цифрового звука весьма популярна частота дискретизации 44 100 выборок в секунду. Для концертного *ля* эта частота соответствует рисованию каждого цикла синусоидальной волны при выборке приблизительно в 100 точках. Поскольку выборка производится равномерно, для выбираемых точек необходимо вычислять только значения по оси y. Все очень просто: *звук представляется как массив чисел* (значений типа `float` в диапазоне от –1 до +1). Функция `stdaudio.playSamples()` из нашего звукового модуля получает массив значений типа `float` и проигрывает представляемый этим массивом звук на вашем компьютере.

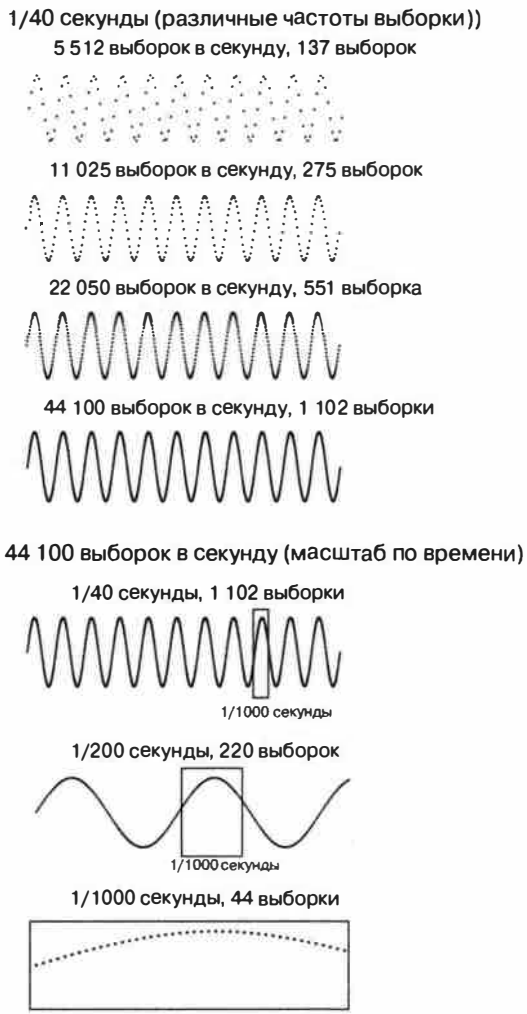
Предположим, например, что необходимо проиграть концертное *ля* в течение 10 секунд. При 44 100 выборках в секунду необходим массив на 441 001 значение типа `float`. Для заполнения такого массива используется цикл `for`, осуществляющий выборку функции $\sin(2\pi t \times 440)$ при $t = 0/44100, 1/44100, 2/44100, 3/44100, \ldots, 441000/44100$. Как только массив будет заполнен этими значениями, все готово для использования функции `stdaudio.playSamples()` в следующем коде:

```
SPS = 44100                 # выборок в секунду
hz = 440.0                  # концертное ля
duration = 10.0             # десять секунд
n = int(SPS * duration)

a = stdarray.create1D(n+1)
for i in range(n+1):
    a[i] = math.sin(2.0 * math.pi * i * hz / SPS)
stdaudio.playSamples(a)
stdaudio.wait()
```

Этот код — аналог программы "Hello, World" для цифрового звука. Как только вы используете ее для проигрывания нот на компьютере, вы сможете создавать код для проигрывания других нот и музыки!

Различие между выводом звука и рисованием колебательной кривой — не более чем в устройстве вывода. Действительно, весьма поучительно и интересно проследить вывод тех же чисел и на стандартное графическое устройство, и на стандартное аудиоустройство (см. упр. 1.5.27).

Сохранение в файл. Музыка может занять много места на компьютере. При 44 100 выборках в секунду четырехминутная песня займет $4 \times 60 \times \times 44100 = 10\,584\,000$ чисел. Поэтому общепринято представлять соответствующие звуку числа в двоичном формате, использующем меньше места, чем строки цифр, используемых для стандартного ввода и вывода. За последние годы было разработано много таких форматов; модуль `stdaudio` использует формат `.wav`. На сайте книги можно найти информацию о формате `.wav`, однако детали несущественны, поскольку модуль `stdaudio` сам позаботится о преобразованиях. Модуль `stdaudio` позволяет проигрывать файлы `.wav`, а также составлять программы, создающие и манипулирующие массивами типа `float`, а также читать и записывать их в файлы `.wav`.



Выборка синусоидальной волны

Для демонстрации простоты создания музыки при помощи функций модуля `stdaudio` (подробности см. в таблице API) рассмотрим программу 1.5.8 (`playthattune.py`). Она получает ноты со стандартного ввода, индексированные по хроматической гамме от концертного *ля*, и проигрывает их на стандартном аудиоустройстве. Для этой базовой схемы возможно множество разных модификаций, некоторые из которых рассматриваются в упражнениях.

Мы включаем модуль `stdaudio` в свой базовый арсенал инструментальных средств программирования, поскольку обработка звука — одно из важнейших применений информатики, которое, конечно, знакомо вам. Обработка цифрового сигнала имеет не только коммерческое применение, она оказала феноменальное влияние на современное общество, науку и технику, объединяя физику и информатику. Подробнее компоненты цифровой обработки сигнала рассматриваются далее. (В разделе 2.1 приведен пример создания звуков, куда более музыкальных, чем отдельные ноты в приложении `playthattune.py`.)

API для функций сайта книги, связанных с выводом звука

Вызов функции	Описание
`stdaudio.playFile(filename)`	Проигрывает все звуковые выборки из файла `filename.wav`
`stdaudio.playSamples(a)`	Проигрывает все звуковые выборки из массива `a[]` типа `float`
`stdaudio.playSample(x)`	Проигрывает звуковую выборку из переменной `x` типа `float`
`stdaudio.save(filename, a)`	Сохраняет все звуковые выборки из массива `a[]` типа `float` в файл `filename.wav`
`stdaudio.read(filename)`	Читает все звуковые выборки из файла `filename.wav` и возвращает их как массив типа `float`
`stdaudio.wait()`	Ожидает завершения проигрывания текущего звука (должен быть завершающим вызовом `stdaudio` в каждой программе)

Резюме. Ввод и вывод — особенно убедительный пример мощи абстракций, поскольку устройства стандартного ввода и вывода, графики и аудио позволяют в разное время обращаться к разным физическим устройствам, не внося изменений в программы. Хотя устройства могут существенно отличаться, мы можем составлять программы, способные осуществлять ввод и вывод вне зависимости от свойств специфических устройств. С этого момента функции из модулей `stdio`, `stddraw` и `stdaudio` мы будем использовать почти во всех программах этой книги. Одно из важнейших преимуществ использования таких модулей в том, что вы можете подключать новые устройства, которые быстрее, дешевле или способны содержать больше данных без необходимости изменять саму программу. В подобной ситуации подробности соединения — это вопрос, решаемый операционной системой и реализацией книжных модулей. На современных системах новые устройства обычно

снабжаются специальным программным обеспечением, автоматически решающем такие вопросы и для операционной системы, и для языка Python.

Программа 1.5.8. Обработка цифрового сигнала (`playthattune.py`)

```
import math
import stdarray
import stdaudio
import stdio

SPS = 44100
CONCERT_A = 440.0

while not stdio.isEmpty():
    pitch = stdio.readInt()
    duration = stdio.readFloat()
    hz = CONCERT_A * (2 ** (pitch / 12.0))
    n = int(SPS * duration)
    samples = stdarray.create1D(n+1, 0.0)
    for i in range(n+1):
        samples[i] = math.sin(2.0 * math.pi * i * hz / SPS)
    stdaudio.playSamples(samples)
stdaudio.wait()
```

`pitch`	Расстояние от ноты *ля*
`duration`	Время проигрывания ноты
`hz`	Частота
`n`	Количество выборок
`samples[]`	Выборки синусоидальной волны

Эта программа читает выборки со стандартного устройства ввода и проигрывает звук на стандартном аудиоустройстве. Эта управляемая данными программа проигрывает чистые тона нот хроматической гаммы, поступающие со стандартного ввода в виде расстояния от концертного *ля* и продолжительности (в секундах). Проверочный клиент читает ноты со стандартного ввода, создает массив выборок синусоидальной волны определенной частоты и продолжительности при 44 100 выборках в секунду, а затем проигрывает каждую ноту, вызвав функцию `stdaudio.playSamples()`.

```
% more elise.txt
7 .25
6 .25
7 .25
6 .25
7 .25
2 .25
5 .25
3 .25
0 .50
```

```
% python playthattune.py < elise.txt
```

Концептуально одним из наиболее значительных свойств потоков данных устройства стандартного ввода, устройства стандартного вывода, стандартного графического устройства и стандартного аудиоустройства является то, что они *бесконечны*: с точки зрения программы, нет никакого предела их длине. Это позволяет создавать весьма долговременные программы поскольку они менее чувствительны к изменениям в технологиях, чем программы со встроенными пределами. Это соотносится также с *машиной Тьюринга*, абстрактным устройством, используемым в теории программирования для помощи в понимании фундаментальных ограничений на возможности реальных компьютеров. Одно из основных свойств модели — идея конечного дискретного устройства, работающего с неограниченным объемом ввода и вывода.

Вопросы и ответы

Как сделать книжные модули `stdio`, `stddraw` и `stdaudio` доступными для Python?

Если вы следовали поэтапным инструкциям на сайте книги по установке Python, то эти модули уже должны быть доступны. Обратите внимание, что копирования файлов `stddraw.py` и `stdaudio.py` с сайта книги и помещения их в тот же каталог, что и использующие их программы, недостаточно, поскольку для графики и аудио они полагаются на библиотеку (набор модулей) `pygame`.

Есть ли стандартные модули Python для обработки стандартного вывода?

Фактически такие средства встроены в Python. В Python 2 для вывода данных можно использовать оператор `print`. В Python 3 оператора `print` нет; но есть подобная ему функция `print()`.

Почему мы используем для записи в поток стандартного вывода модуль `stdio` с сайта книги вместо средств, уже предоставляемых языком Python?

Нам нужно было составить код, работающий (в максимально возможной степени) со всеми версиями языка Python. Например, использование оператора `print` во всех наших программах означало бы, что они будут работать с Python 2, но не с Python 3, а для использования функций модуля `stdio` достаточно наличия надлежащей библиотеки.

А как насчет стандартного ввода?

Эти возможности (разные) есть и в Python 2, и в Python 3, они соответствуют функции `stdio.readLine()`, но ничего соответствующего функции `stdio.readInt()` и подобным ей нет. Использование модуля `stdio` снова позволяет составлять программы, которые не только используют в своих интересах эти дополнительные возможности, но и работают в обеих версиях Python.

А как насчет стандартных устройств графики и звука?

Звуковой библиотеки в языке Python нет. Для графики в языке Python имеется библиотека `tkinter`, но она слишком медленна для некоторых из графических программ, описанных в этой книге. Наши модули `stddraw` и `stdaudio` предоставляют удобные в работе API на базе библиотеки `pygame`.

Если я использую формат `%2.4f` в функции `stdio.writef()` для вывода значения типа `float`, то получу две цифры перед десятичной точкой и четыре после, не так ли?

Нет, это задаст только четыре цифры после десятичной точки. Число, предшествующее десятичной точке, задает ширину целочисленного поля. Чтобы

задать всего семь символов, четыре перед десятичной точкой и два после, используйте формат `%7.2f`.

Какие еще коды преобразования доступны для функции `stdio.writef()`?

Для целых чисел есть восьмеричный формат (`o` — octal) и шестнадцатеричный (`x` — hexadecimal). Есть также множество форматов для дат и времени. Дополнительную информацию см. на сайте книги.

Может ли моя программа перечитать данные со стандартного устройства ввода?

Нет. У вас есть только одна попытка, точно так же нельзя отменить вызов функции `stdio.writeln()`.

Что будет, если моя программа попытается читать данные со стандартного устройства ввода, после того как они закончились?

Ошибка `EOFError` во время выполнения. Функции `stdio.isEmpty()` и `stdio.hasNextLine()` позволяют избежать такой ошибки, они проверяют, доступны ли еще данные во вводе.

Почему функция `stddraw.square(x, y, r)` рисует квадрат шириной `2r`, а не `r`?

Это делает ее совместимой с функцией `stddraw.circle(x, y, r)`, где третий аргумент радиус круга, а не диаметр. В этом контексте `r` — радиус самого большого круга, который может быть вписан в квадрат.

Что будет, если моя программа вызовет функцию `stddraw.show(0)`?

Этот вызов указывает функции `stddraw` скопировать фоновый холст в окно стандартного графического устройства, а затем ждать 0 миллисекунд (т.е. не ждать вообще) перед продолжением. Такой вызов функции оправдан, если, например, необходимо запустить анимацию на самой высокой скорости, обеспечиваемой компьютером.

Позволяет ли модуль `stddraw` рисовать кривые, отличные от кругов?

Мы должны были где-то провести черту (фигурально), поэтому обеспечили вывод только самых простых форм, обсуждаемых в тексте. Вы можете рисовать и другие формы, точка за точкой, как в нескольких упражнениях, приведенных далее, но их заполнение непосредственно не поддерживается.

Таким образом, для нот ниже концертного *ля* при создании исходных файлов для программы `playthattune.py` следует использовать отрицательные целые числа?

Да. Фактически наш выбор нулевой позиции в концертном *ля* произволен. Популярный стандарт MIDI начинает отсчет с ноты *до*, на пять октав ниже концертного *ля*. В соответствии с этим соглашением концертному *ля* соответствует 69, и вам не нужно использовать отрицательные числа.

Почему я слышу странный звук на стандартном аудиоустройстве, когда пытаюсь воспроизвести синусоидальную волну с частотой 30 000 Герц (или более)?

Частота Найквиста определена как половина частоты дискретизации (выборки) и представляет самую высокую частоту, которая может быть воспроизведена. Для стандартного аудиоустройства частота выборки составляет 44 100, таким образом, его частота Найквиста — 22 050.

Упражнения

1.5.1. Составьте программу, читающую со стандартного устройства ввода целые числах (сколько пользователь введет) и выводящую на стандартное устройство вывода максимальное и минимальное значения.

1.5.2. Измените свою программу из предыдущего упражнения так, чтобы вводимые целые числа были только положительными (просить пользователя ввести положительное целое число каждый раз, когда он вводит иное значение).

1.5.3. Составьте программу, которая получает в аргументе командной строки целое число `n`, читает со стандартного устройства ввода *n* чисел типа `float`, а выводит на стандартное устройство вывода их среднее значение и *среднеквадратичное отклонение* (квадратный корень суммы квадратов их разниц со средним значением, деленный на *n*).

1.5.4. Дополните свою программу из предыдущего упражнения фильтром, выводящем на стандартное устройство ввода все числа типа `float`, среднеквадратичное отклонение от среднего которых превышает 1.5.

1.5.5. Составьте программу, читающую последовательности целых чисел и выводящую длину самой длинной последовательности одинаковых чисел и образующее ее число. Например, если введены числа `1 2 2 1 5 1 1 7 7 7 7 1 1`, то ваша программа должна вывести `Longest run: 4 consecutive 7s.`

1.5.6. Составьте фильтр, читающий последовательности целых чисел и выводящий целые числа, удаляя последовательно повторяющиеся значения. Например, если введено `1 2 2 1 5 1 1 7 7 7 7 1 1 1 1 1 1 1 1 1`, то программа должна вывести `1 2 1 5 1 7 1`.

1.5.7. Составьте программу, получающую аргумент командной строки `n`, читающую со стандартного устройства ввода `n-1` разных целых чисел от `1` до `n` и определяющую недостающее целое число.

1.5.8. Составьте программу, читающую со стандартного устройства ввода положительные вещественные числа и выводящую их средние геометрические и гармонические значения. *Среднее геометрическое* (geometric mean) из *n* положительных чисел x_1, x_2, ..., x_n составляет $(x_1 \times x_2 \times \ldots \times x_n)^{1/n}$, а *среднее гармоническое* (harmonic mean) — $(1/x_1 + 1/x_2 + \ldots + 1/x_n) / (1/n)$. *Подсказка*: для среднего геометрического берите логарифмы, чтобы избежать переполнения.

1.5.9. Предположим, файл `in.txt` содержит две строки: `F` и `F`. Рассмотрим следующую программу (`dragon.py`):

```
import stdio
dragon = stdio.readString()
```

```
nogard = stdio.readString()
stdio.write(dragon + 'L' + nogard)
stdio.write(' ')
stdio.write(dragon + 'R' + nogard)
stdio.writeln()
```

Что делает следующая команда (см. упр. 1.2.35)?

```
python dragon.py < in.txt | python dragon.py | python dragon.py
```

1.5.10. Составьте фильтр `tenperline.py`, читающий последовательность целых чисел от 0 до 99 и выводящий по 10 целых чисел в строку с выровненными столбцами. Затем составьте программу `randomintseq.py`, получающую два аргумента командной строки `m` и `n`, а выводящую `n` случайных целых чисел от `0` до `m-1`. Проверьте свои программы командой `python randomintseq.py 200 100 | python tenperline.py`.

1.5.11. Составьте программу, читающую текст со стандартного устройства ввода и выводящую количество слов в тексте. В данном упражнении слово — это последовательность символов, окруженная символами отступа.

1.5.12. Составьте программу, которая читает со стандартного устройства ввода строки, содержащие имя и два целых числа, а затем использует функцию `writef()` для вывода таблицы со столбцами, содержащими имена, целые числа и результаты деления первого числа на второе с точностью в три позиции после десятичной точки. Эту программу можно использовать для сведения в таблицу средних результатов бейсбольных игроков или оценок учеников.

1.5.13. Какие из следующих задач *требуют* сохранения всех значений, поступающих со стандартного ввода (скажем, в массив), и что может быть реализовано как фильтр, использующий только фиксированное количество переменных? В каждом случае ввод поступает со стандартного устройства ввода и представляет собой n чисел типа `float` от 0 до 1.

- Выведите максимальное и минимальное числа.
- Выведите k-е наименьшее число.
- Выведите сумму квадратов чисел.
- Выведите среднее значение для n чисел.
- Выведите процент чисел, значение которых больше среднего.
- Выведите n чисел в порядке возрастания.
- Выведите n чисел в случайном порядке.

1.5.14. Составьте программу, которая получает в аргументах командной строки три числа (количество лет, основную сумму и процентную ставку)

и выводит таблицу ежемесячных платежей, остатка основной суммы и выплаченного по ссуде процента (см. упр. 1.2.21).

1.5.15. Составьте программу, которая получает три аргумента командной строки `x`, `y` и `z`, читает со стандартного ввода последовательность координат точек (x_i, y_i, z_i) и выводит расстояния точки, ближайшей к точке (x, y, z). Напомним, что квадрат дистанции между точками (x, y, z) и (x_i, y_i, z_i) составляет $(x - x_i)^2 + (y - y_i)^2 + (z - z_i)^2$. Для эффективности не используйте ни функцию `math.sqrt()`, ни оператор `**`.

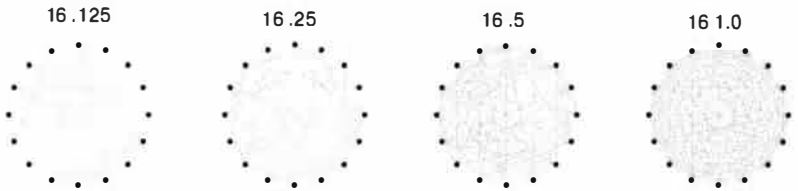
1.5.16. Составьте программу, получающую позиции и массы ряда объектов и вычисляющую их центр массы, или *среднюю точку* (average *mag*nitude). Средняя точка — это позиция центра массы всех n объектов. Если даны позиции и массы (x_i, y_i, m_i), то средняя точка (x, y, m) вычисляется как

$$m = m_1 + m_2 + \ldots + m_n$$
$$x = (m_1 x_1 + \ldots + m_n x_n) / m$$
$$y = (m_1 y_1 + \ldots + m_n y_n) / m$$

1.5.17. Составьте программу, которая читает последовательности вещественных чисел от –1 до +1 и выводит их среднюю точку, среднюю степень и количество переходов через нуль. *Средняя точка* — это среднее от абсолютных значений. *Среднее квадратов значений* (average power) — это среднее от квадратов значений. Количество переходов через нуль (zero crossing) — это количество переходов значений от строго отрицательного к строго положительному, или наоборот. Эти три статистических величины широко используются в анализе цифровых сигналов.

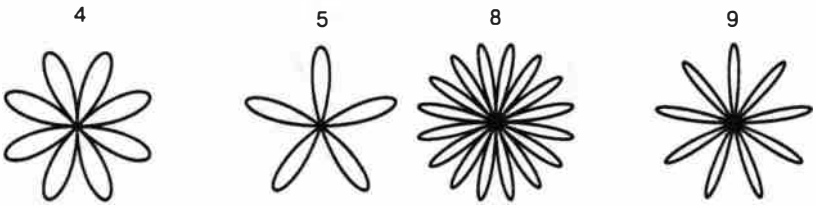
1.5.18. Составьте программу, которая получает аргумент командной строки `n` и рисует шахматную доску из n × n красных и черных квадратов. Красные квадраты начинаются с левого нижнего угла.

1.5.19. Составьте программу, которая получает в аргументах командной строки целое число `n` и число `p` типа `float` (от 0 до 1) и рисует `n` равноудаленных черных точек на окружности, а затем с вероятностью `p` для каждой пары точек соединяющую их серой линией.



1.5.20. Составьте код рисования червы, пики, трефы и бубны. Чтобы нарисовать черву, нарисуйте бубну, а затем пририсуйте два полукруга слева и справа верху.

1.5.21. Составьте программу, которая получает аргумент командной строки `n` и рисует "цветок" из `n` лепестков (если `n` нечетное), или `2n` лепестков (если `n` четное), при рисовании полярных координат (r, θ) в функции $r = \sin(n\,\theta)$ для θ в диапазоне от 0 до 2π радианов.

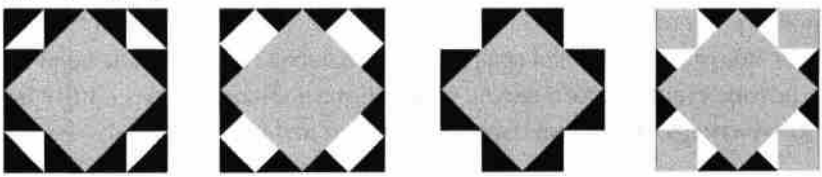


1.5.22. Составьте программу, которая получает из командной строки строку `s`, отображает ее на экране в стиле баннера, движущуюся слева направо, и возвращается назад к началу по достижении конца. Добавьте второй аргумент командной строки для контроля скорости.

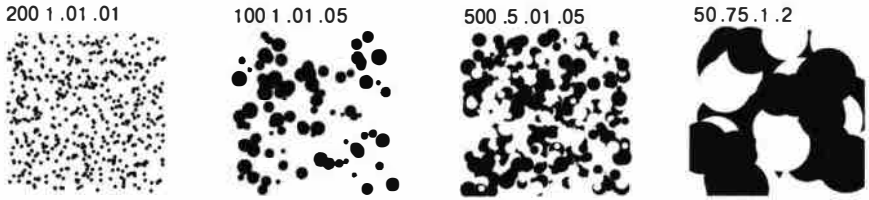
1.5.23. Измените программу `playthattune.py` так, чтобы она получала дополнительные аргументы командной строки, контролирующие громкость (умножьте каждое значение выборки на громкость) и темп (умножьте продолжительность каждой ноты на темп).

1.5.24. Составьте программу, получающую имя файла `.wav` и скорость воспроизведения `r` в аргументах командной строки и проигрывающую файл с заданной скоростью. Сначала используйте функцию `stdaudio.read()` для чтения файла в массив `a[]`. Если `r = 1`, просто проиграйте массив `a[]`; в противном случае создайте новый массив `b[]` размером примерно `r` раз по `a.length`. Если `r<1`, заполните массив `b[]` *выборкой* из исходного массива; если `r>1`, заполните массив `b[]` интерполяцией исходного массива. Затем проиграйте массив `b[]`.

1.5.25. Составьте программы, использующие модуль `stddraw` для создания следующих узоров.



1.5.26. Составьте программу `circles.py`, рисующую заполненные круги случайного размера, расположенные в случайном порядке в единичном квадрате. Получите изображения, подобные приведенным ниже. Программа должна получать четыре аргумента командной строки: количество кругов, вероятность, что круг окажется черным, минимальный и максимальный радиусы круга.



Практические упражнения

1.5.27. *Визуализация аудио.* Измените программу 1.5.8 (`playthattune.py`) так, чтобы передать проигрываемые значения на стандартное графическое устройство для просмотра звуковых волн по мере их проигрывания. Поэкспериментируйте с выводом кривых на холсте, чтобы синхронизировать звук и изображение.

1.5.28. *Статистический опрос.* При сборе статистических данных для политических опросов очень важно получить непредвзятую выборку по зарегистрированным избирателям. Предположим, есть файл с записями об `n` зарегистрированных избирателей, по одной строке на каждого. Составьте фильтр, осуществляющий случайную выборку размером `m` (см. программу 1.4.1 (`sample.py`)).

1.5.29. *Анализ ландшафта.* Предположим, ландшафт представлен двумерной таблицей значений высот (в метрах). *Пик* — это ячейка таблицы, значения четырех соседних ячеек которой (слева, справа, выше и ниже) строго ниже пика. Составьте программу `peaks.py`, которая читает ландшафт со стандартного ввода, а затем вычисляет и выводит количество пиков.

1.5.30. *Гистограмма.* Предположим, поток стандартного ввода — это последовательность чисел типа `float`. Составьте программу, получающую из командной строки целое число `n`, а также два числа типа `float`, `lo` и `hi`. Она использует модуль `stddraw` для графического вывода гистограммы количеств чисел в потоке стандартного ввода, относящихся к каждому из `n` интервалов равного размера, определяемых при делении интервала `(lo, hi)` на `n`.

1.5.31. *Спирограф.* Составьте программу, получающую из командной строки три параметра, R, r, и a, и действующую как *спирограф*. Спирограф рисует эпициклоиды — кривые, образующиеся вращением круга радиуса r вокруг большего неподвижного круга радиуса R. Если смещение пера от центра вращающегося круга составляет $(r + a)$, то уравнения результирующей кривой в момент времени t таковы:

$$x(t) = (R + r)\cos(t) - (r + \mathrm{a})\cos((R + r)t/r)$$
$$y(t) = (R + r)\sin(t) - (r + \mathrm{a})\sin((R + r)t/r)$$

Эти кривые рисует популярная игрушка спирограф, представляющая собой зубчатые диски с маленькими отверстиями, куда можно вставить карандаш и рисовать эпициклоиды.

1.5.32. *Часы.* Составьте программу, отображающую анимацию секундной, минутной и часовой стрелок аналоговых часов. Для обновления экрана примерно раз в секунду используйте вызов функции `stddraw.show(1000)`.

1.5.33. *Осциллограф.* Составьте программу, моделирующую осциллограф и выводящую фигуры Лиссажу. Эти фигуры названы в честь французского физика Жюля Антуана Лиссажу, изучавшего фигуры, возникающие при наложении двух взаимно перпендикулярных периодических колебаний. Подразумевая, что входные данные синусоидальны, кривую опишут следующие параметрические уравнения:

$$x(t) = A_x \sin(w_x t + \theta_x)$$
$$y(t) = A_y \sin(w_y t + \theta_y)$$

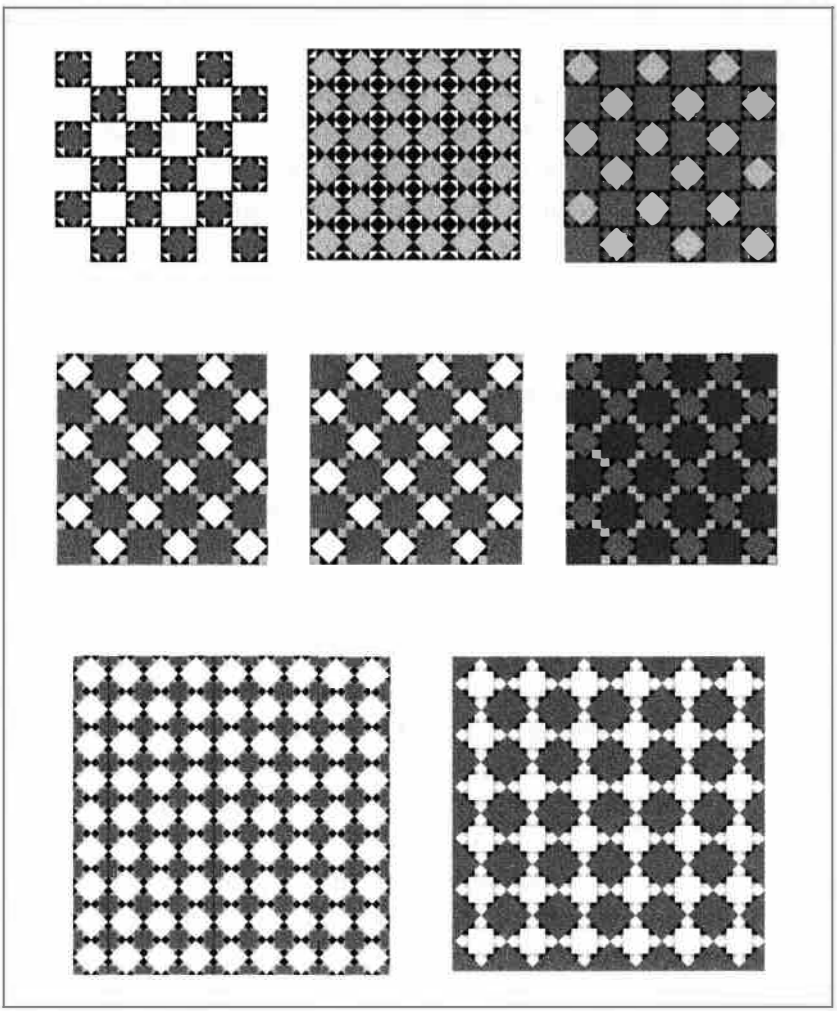
Получите из командной строки шесть параметров: A_x и A_y (амплитуды); w_x и w_y (угловые скорости); θ_x и θ_y (коэффициенты фазы).

1.5.34. *Прыгающий мяч с перемещениями.* Измените программу `bouncingball.py` так, чтобы получить изображения (см. выше), где перемещающийся мяч оставлял следы на сером фоне.

1.5.35. *Прыгающий мяч с гравитацией.* Измените программу `bouncingball.py`, добавив гравитацию в вертикальном направлении. Добавьте также вызовы функции `stdaudio.playFile()`, чтобы обеспечить один звуковой эффект, когда мяч ударяется о стену, и другой, когда он ударяется об пол.

1.5.36. *Случайные мелодии.* Составьте программу, использующую модуль `stdaudio` для проигрывания мелодий в случайном порядке. Поэкспериментируйте с различиями в вероятности, повторениями и другими параметрами при проигрывании мелодий.

1.5.37. *Плиточные узоры.* Используя свое решение упражнения 1.5.25, составьте программу `tilepattern.py`, которая получает в аргументе командной строки число `n` и рисует узор `n` × `n`, используя плитку по вашему выбору. Добавьте второй аргумент командной строки, позволяющий раскладывать плитку, как на шахматной доске. Добавьте третий аргумент командной строки для выбора цвета. Используйте образцы узоров, представленные ниже, как отправную точку для разработки эскиза плиточного пола. Проявите творческий подход! *Примечание*: все примеры, приведенные ниже, можно найти во многих древних (и современных) зданиях.



1.6. Случай из практики: случайная навигация по сайтам

Общение через веб стало неотъемлемой частью нашей повседневной жизни. Частично этому поспособствовали активные научные исследования структуры веб, начавшиеся с момента его появления. Далее мы рассмотрим упрощенную модель веб, которая упростит понимание некоторых из его свойств. Варианты этой модели широко используются и являются ключевым фактором взрывного роста приложений поиска в веб.

Программы этого раздела...

Программа 1.6.1. Вычисление матрицы переходов (`transition.py`) 196

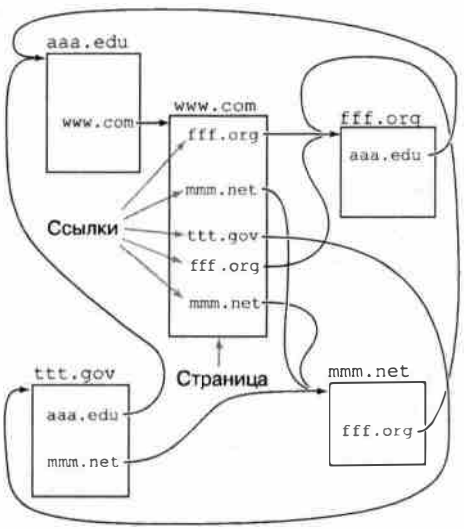
Программа 1.6.2. Моделирование случайной навигации (`randomsurfer.py`) 199

Программа 1.6.3. Смешение цепи Маркова (`markov.py`) 205

Модель *случайной навигации* (random surfer) проста. Предположим, что веб представляет собой фиксированный набор *страниц* (page), что каждая страница содержит фиксированный набор *ссылок* (link) и каждая ссылка указывает на некую другую страницу. Задача сводится к изучению человека, бесцельно переходящего со страницы на страницу по ссылкам или вводящего их адреса в строке адресов (случайная навигация).

Базовая математическая модель веб, известная как *граф* (graph), подробно рассматривается в конце книги (раздел 4.5). Отложим обсуждение подробностей обработки графов до тех пор, а пока сконцентрируемся на вычислениях, связанных с обычной и хорошо изученной вероятностной моделью, точно описывающей поведение при случайной навигации.

Первый этап изучения модели случайной навигации — это ее более конкретная формулировка. Сложность заключается в определении того, что считать случайным переходом со страницы на страницу. Интуитивно понятное *правило 90 на 10* (90-10 rule) учитывает оба метода перехода на новую страницу: *считается, что в 90 процентах случаев случайной навигации пользователь выбирает случайную ссылку на текущей странице (каждая ссылка равновероятна) и только в 10 процентах случаев переходит на случайную страницу непосредственно (все страницы в веб равновероятны).*



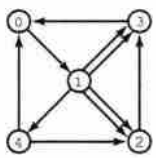
Страницы и ссылки

По собственному опыту вы знаете, что у этой модели есть существенные недочеты, поскольку поведение реального пользователя веб не совсем так просто.

- Никто не выбирает ссылки или страницы с равной вероятностью.
- В реальности перейти случайно на каждую страницу в веб непосредственно нет никакой возможности.
- Соотношение 90 к 10 (или любое другое фиксированное) — это только предположение.
- Не принимаются во внимание ни кнопка Back (Назад), ни закладки.
- Мы можем позволить себе работать только с небольшой частью веб.

Несмотря на недостатки, этой модели вполне достаточно для изучения многих свойств веб. Чтобы оценить модель, рассмотрим небольшой пример. Какая, по-вашему, страница посещается с наибольшей вероятностью при случайной навигации?

Каждый человек, использующий веб, ведет себя несколько беспорядочно, поэтому понимание поведения при случайной навигации представляет большой интерес для тех, кто создает инфраструктуру веб и веб-приложения. Модель — это инструмент для понимания действий каждого из миллиардов пользователей веб. Для изучения модели и ее значения в данном разделе будут использованы базовые средства программирования, описанные в этой главе.




```
% more tiny.txt
5
0 1
1 2  1 2
1 3  1 3  1 4
2 3
3 0
4 0  4 2
```

Ссылки

Входной формат случайной навигации

Входной формат. Поскольку необходима возможность изучить поведение при случайной навигации на разных моделях, а не только на одном примере, необходим *код, управляемый данными* (data-driven code), когда данные хранятся в файлах, а программы читают их со стандартного ввода. Первый этап этого подхода подразумевает определение входного формата, который можно использовать для структурирования информации во входных файлах. Мы вполне можем определить любой удобный для нас входной формат.

Далее вы узнаете, как в программах Python читать веб-страницы (раздел 3.1), преобразовывать имена в числа (раздел 4.4), а также ознакомитесь с другими методиками эффективной обработки графа. А пока будем считать, что есть n веб-страниц, пронумерованных от 0 до $n-1$, и ссылки, представленные парами таких номеров, причем первый определяет страницу, содержащую ссылку, а второй — страницу, на которую указывает эта ссылка. С учетом этих соглашений простейший формат входного потока задачи случайной навигации состоит из целого числа (значение n), сопровождаемого последовательностью пар целых чисел (представляющих ссылки). Благодаря способу обработки символов отступа функциями чтения модуля `stdio` каждую ссылку можно поместить в отдельную строку или упорядочить их по несколько на строку.

Матрица переходов. Для определения поведения при случайной навигации воспользуемся двумерной матрицей — *матрицей переходов* (transition matrix). При наличии n веб-страниц мы создаем матрицу $n \times n$ так, что элемент в ряду `i` и столбце `j` определяет вероятность случайного перехода на страницу `j` со страницы `i`. Наша первая задача — составить код, способный создать такую матрицу для любого заданного ввода. При применении правила 90×10 это вычисление не вызывает затруднений. Осуществим его в три этапа.

- Прочитать число `n`, а затем создать массивы `linkCounts[][]` и `outDegrees[]`.
- Прочитать ссылки, подсчитать значения так, чтобы элемент `linkCounts[i][j]` содержал количество ссылок со страницы `i` на `j`, а элемент `outDegrees[i]` — количество ссылок со страницы `i` на любую другую.
- Использовать правило 90×10 для вычисления вероятностей.

Два первых этапа элементарны, а третий немного сложнее: умножьте значение `linkCounts[i][j]` на `0.90/outDegrees[i]`, если есть ссылка от `i` на `j` (случайный выбор ссылки с вероятностью 0.9), а затем добавьте `0.10/n` к каждому элементу (переход на случайную страницу с вероятностью 0.1). Это вычисление выполняет программа 1.6.1 (`transition.py`): это просто фильтр, преобразующий список ссылок модели веб в матричное представление переходов.

Матрица переходов важна, поскольку каждый ее ряд представляет *дискретное распределение вероятности* (discrete probability distribution) — ее элементы полностью определяют следующий переход при случайной навигации, предоставляя его вероятность для каждой страницы. В частности, обратите внимание, что сумма элементов составляет 1 (пользователь всегда переходит куда-либо).

Вычисление матрицы переходов

Вывод программы `transition.py` определяет формат другого файла, для матриц значений типа `float`: количество рядов и столбцов, сопровождаемое значениями элементов матрицы. Теперь можно составить код, читающий и обрабатывающий матрицу переходов.

Программа 1.6.1. Вычисление матрицы переходов (transition.py)

```
import stdarray
import stdio

n = stdio.readInt()

linkCounts = stdarray.create2D(n, n, 0)
outDegrees = stdarray.create1D(n, 0)

while not stdio.isEmpty():
    # Подсчет ссылок.
    i = stdio.readInt()
    j = stdio.readInt()
    outDegrees[i] += 1
    linkCounts[i][j] += 1

stdio.writeln(str(n) + ' ' + str(n))

for i in range(n):
    # Вывод распределения вероятности для ряда i.
    for j in range(n):
        # Вывод распределения вероятности для столбца j.
        p = (0.90 * linkCounts[i][j] / outDegrees[i]) + (0.10 / n)
        stdio.writef('%8.5f', p)
    stdio.writeln()
```

`n`	Количество страниц
`linkCounts[i][j]`	Количество ссылок со страницы `i` на страницу `j`
`outDegrees[i]`	Количество ссылок со страницы `i`
`p`	Вероятность перехода

Эта программа — фильтр, читающий ссылки из стандартного ввода и выводящая соответствующую матрицу переходов в стандартный вывод. Сначала она обрабатывает ввод, подсчитывая ссылки с каждой страницы, затем применяет правило 90×10 для расчета матрицы переходов. Подразумевается, что во вводе нет страниц с нулевыми исходящими степенями (см. упражнение 1.6.3).

```
% more tiny.txt
5
0 1
1 2  1 2
1 3  1 3  1 4
2 3
3 0
4 0  4 2
```

```
% python transition.py < tiny.txt
5 5
 0.02000 0.92000 0.02000 0.02000 0.02000
 0.02000 0.02000 0.38000 0.38000 0.20000
 0.02000 0.02000 0.02000 0.92000 0.02000
 0.92000 0.02000 0.02000 0.02000 0.02000
 0.47000 0.02000 0.47000 0.02000 0.02000
```

Модель. Имея матрицу перехода, смоделируем поведение при случайной навигации, задействуя на удивление мало кода, как можно заметить в программе 1.6.2 (`randomsurfer.py`). Эта программа читает матрицу переходов и осуществляет перемещение по сети согласно правилам, начиная со страницы `0` и осуществляя заданное в аргументе командной строки количество переходов. Она подсчитывает количество посещений каждой страницы. Деление этого количества на количество переходов дает оценку вероятности посещения данной страницы при случайной навигации. Эта вероятность известна как *ранг* (rank) страницы. Другими словами, программа `randomsurfer.py` вычисляет ранги всех страниц.

Один случайный переход. Ключом к вычислению является случайный переход в матрице переходов. Текущее положение хранится в переменной `page`. Ряд `page` матрицы `p[][]` дает для каждого `j` вероятность перехода на страницу `j`. Другими словами, когда пользователь находится на странице `page`, наша задача заключается в создании случайного целого числа в диапазоне от `0` до `n-1` согласно распределению, заданному рядом `page` в матрице переходов (одномерный массив `p[page]`). Как решить эту задачу? Можно использовать функцию `random.random()` для создания случайного числа `r` типа `float` в диапазоне от `0` до `1`, но как это поможет добраться до случайной страницы? Один из способов — рассматривать вероятности в ряду `page` как определения набора из `n` интервалов (0, 1), длина каждого из которых соответствует вероятности. Наша случайная переменная `r` попадает в один из интервалов с вероятностью, определенной длиной интервала. Это рассуждение приводит к следующему коду:

Создание случайного целого числа из дискретного распределения

```
total = 0.0
for j in range(0, n)
    total += p[page][j]
    if r < total:
        page = j
        break
```

Переменная `total` отслеживает конечные точки интервалов, определенных в ряде `p[page]`, а цикл `for` находит интервал, содержащий случайное значение `r`. Предположим, например, что пользователь находится на странице `4`. Вероятности перехода составляют 0,47, 0,02, 0,47, 0,02 и 0,02, а переменная `total` получает одно из значений: `0.0`, `0.47`, `0.49`, `0.96`, `0.98` и `1.0`. Эти значения указывают, что вероятности определяют пять интервалов (0, 0,47), (0,47, 0,49), (0,49, 0,96), (0,96, 0,98) и (0,98, 1), по одному для каждой страницы. Теперь предположим, что вызов функции `random.random()` возвращает значение `0.71`. Увеличиваем значение `j` с 0 до 1, до 2 и останавливаемся, поскольку значение `0.71` находится в интервале (0,49, 0,96), таким образом мы посылаем пользователя на третью страницу (страница `2`). Затем выполняем то же вычисление для `p[2]`, и случайная навигация начинается. Для больших n можно использовать *бинарный поиск*, чтобы существенно ускорить вычисление (см. упр. 4.2.35). Как правило, в этой ситуации требуется ускорение поиска, поскольку, как вы увидите, нам понадобится огромное количество случайных переходов.

Цепь Маркова, вероятностный процесс, описывающий поведение при случайной навигации, назван в честь российского математика Андрея Маркова, разработавшего эту концепцию в начале XX столетия. Цепи Маркова широко применяются, хорошо изучены и имеют много замечательных и полезных свойств. Например, вы, возможно, задались вопросом, почему программа `randomsurfer.py` осуществляет случайную навигацию со страницы 0, хотя вы, возможно, ожидали случайного выбора. Фундаментальная теорема предела для цепей Маркова гласит, что пользователь может начать *откуда угодно*, поскольку вероятность того, что случайная навигация в конечном счете закончится на любой конкретной странице, одинакова для всех стартовых страниц! Независимо от того, где начат процесс, он в конечном счете стабилизируется в точке, где последующие переходы не предоставляют дальнейшей информации. Это явление известно как *смешение* (mixing). Хотя на первый взгляд это явление могло бы показаться интуитивно противоестественным, оно объясняет взаимосвязь событий в ситуации, где они кажутся хаотическими. В данном контексте ее смысл сводится к идее, что при достаточно продолжительном времени веб выглядит довольно похоже на совершенно хаотические перемещения.

Не у всех цепей Маркова есть свойство смешения. Например, если устранить из модели случайный переход, некоторые конфигурации веб-страниц могут создать проблемы для случайной навигации. Действительно, существуют веб-страницы, известные как *паучья ловушка* (spider trap); они разработаны так, чтобы привлекать входящие ссылки, но не имеют исходящих ссылок. Без случайного перехода по адресу пользователь застрянет в паучьей ловушке. Основная задача правила 90×10 в том, что оно гарантированно устраняет смешение и подобные аномалии.

Программа 1.6.2. Моделирование случайной навигации (randomsurfer.py)

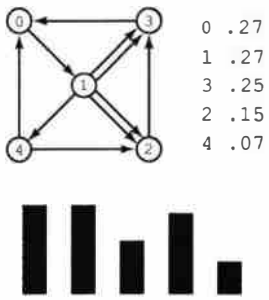
```
import random
import sys
import stdarray
import stdio

moves = int(sys.argv[1])

n = stdio.readInt()
stdio.readInt()               # Не нужно (другое n).
p = stdarray.create2D(n, n, 0.0)
for i in range(n):
    for j in range(n):
        p[i][j] = stdio.readFloat()

hits = stdarray.create1D(n, 0)
page = 0 # Start at page 0.
for i in range(moves):
    r = random.random()       # Вычислить случайную
    total = 0.0               # страницу в ряду p[page]
    for j in range(0, n):     # (см. текст) согласно
        total += p[page][j]   # распределению.
        if r < total:         #
            page = j          #
            break             #
    hits[page] += 1

for v in hits:
    stdio.writef('%8.5f', 1.0 * v / moves)
stdio.writeln()
```

`moves`	Количество переходов
`N`	Количество страниц
`page`	Текущая страница
`p[i][j]`	Вероятность перехода со страницы `i` на страницу `j`
`hits[i]`	Количество посещений страницы `i`

Эта программа использует матрицу переходов для моделирования поведения при случайной навигации. Она получает как аргумент командной строки количество переходов, читает матрицу переходов, осуществляет указанное количество переходов, как предписано матрицей, и выводит относительную частоту посещения каждой страницы. Ключом к вычислению является случайный переход к следующей странице.

```
% python transition.py < tiny.txt | python randomsurfer.py 100
0.24000 0.23000 0.16000 0.25000 0.12000
% python transition.py < tiny.txt | python randomsurfer.py 10000
0.27280 0.26530 0.14820 0.24830 0.06540
% python transition.py < tiny.txt | python randomsurfer.py 1000000
0.27324 0.26568 0.14581 0.24737 0.06790
```

Ранжирование страниц. Модель в программе `randomsurfer.py` довольно проста: она осуществляет заданное количество случайных переходов в графе. Из-за явления смешения увеличение количества итераций дает все более и более точные оценки вероятности того, что пользователь посетит каждую страницу (ранги страниц). Что подсказывает ваша интуиция при первом знакомстве с вопросом? Возможно, вы предположили, что страница `4` имеет самый низкий ранг, а страницы `0` и `1` займут место выше, чем страница `3`? Если хочется узнать, какая страница имеет самый высокий ранг, необходимо все больше и больше точности. Чтобы дать ответы с точностью d позиций после десятичной точки, программе `randomsurfer.py` потребуется 10^d переходов и еще больше переходов, чтобы ответы стабилизировались на точном значении. Для нашего примера это составит десятки тысяч итераций, чтобы получить ответы с точностью до двух позиций после десятичной точки и миллионы итераций для точности в три позиции (см. упр. 1.6.5). В результате получается, что страница `0` превосходит страницу `1` — 27,3 процента против 26,6 процента. Такое крошечное различие в столь небольшой задаче весьма удивительно: если вы предположили, что страница `0` — наиболее вероятное место завершения случайных переходов, то вы весьма удачливы!



Ранги страниц с гистограммой

На практике точные оценки ранга веб-страниц ценны по многим причинам. В первую очередь, они используются для упорядочивания страниц, соответствующих критерию поиска, в поисковых системах. Ранги страниц — это также мера уверенности в надежности инвестиций огромных сумм денег в веб-рекламу. Даже в нашем крошечном примере ранги страниц могли бы использоваться, чтобы убедить рекламодателей платить в четыре раза больше за размещение рекламы на странице 0, чем на странице 4. Вычисление рангов страниц — это задача интересная и для математиков, и для программистов, и для крупных бизнесменов.

Визуализация гистограммы. Модуль `stddraw` существенно упрощает создание визуального представления, позволяющего наглядно продемонстрировать, как частота посещений при случайной навигации соотносится с рангами страниц. Если соответственно масштабировать координаты x и y окна стандартного графического устройства, добавить следующий код в цикл случайных переходов программы `randomsurfer.py` и запустить ее, скажем, для миллиона переходов, то вы увидите рисунок гистограммы частоты, которая в конечном счете стабилизируется в соответствии с рангами страниц. (Константы `1000` и `10` в этом коде несколько произвольны; при желании вы можете изменить их в своем коде.) Применив этот инструмент однажды, вы вероятно захотите использовать его *каждый*

раз, когда изучаете новую модель (возможно, при некоторых незначительных корректировках для больших моделей).

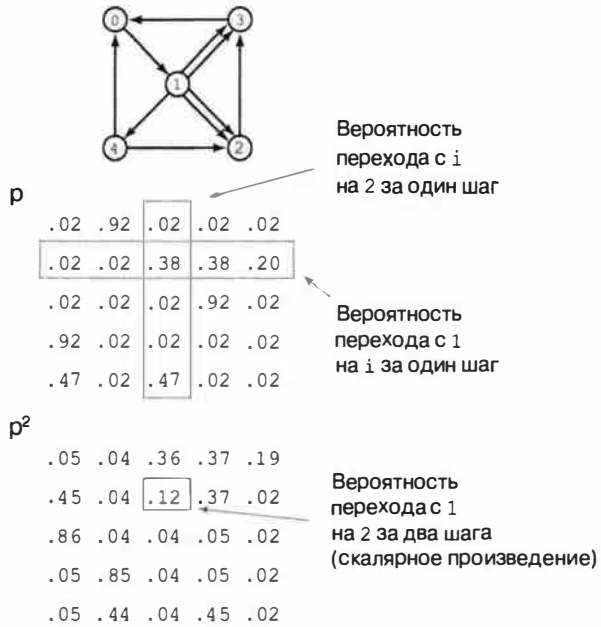
```
if i % 1000 == 0:
    stddraw.clear()
    for k in range(n):
        stddraw.filledRectangle(k - 0.25, 0.0, 0.5, hits[k])
    stddraw.show(10)
```

Изучение других моделей. Наши программы `randomsurfer.py` и `transition.py` — превосходные примеры программ, управляемых данными. Вы легко можете создать новую модель данных, всего лишь создав новый файл, подобный `tiny.txt`, содержимое которого начинается с целого числа `n` и продолжается парами целых чисел от `0` до `n-1`, представляющих ссылки между страницами. Попробуйте использовать это для разных моделей данных, как предложено в упражнениях, или составьте несколько собственных моделей для практики. Если у вас когда-либо возникал вопрос, как осуществляется ранжирование веб-страниц, то это вычисление — ваш шанс проверить свою интуицию и выяснить, почему ранг одной страницы выше, чем другой. Какой вид страниц, вероятней всего, будет оценен выше? Тот, у которого много ссылок на другие страницы, или тот, у которого ссылок меньше? Упражнения в этом разделе представляют много возможностей изучить поведение при случайной навигации. Поскольку программа `randomsurfer.py` использует стандартный ввод, можно составлять простые программы, создающие большие входные модели, передающие свой вывод программе `randomsurfer.py`, и, таким образом, изучать случайную навигацию на довольно больших моделях. Такая гибкость — важнейшая причина использования стандартного ввода и вывода.

Непосредственное моделирование поведения при случайной навигации весьма полезно для понимания структуры веб, но у него есть ограничения. Обдумаем следующий вопрос: можно ли использовать модели веб для вычисления рангов миллионов (или миллиардов!) веб-страниц со ссылками? Быстрый ответ на этот вопрос — *нет*, хотя бы потому, что вы не можете позволить себе хранить матрицу перехода для такого большого количества страниц. У матрицы для миллионов страниц были бы *триллионы* элементов. У вас хватит места на компьютере? Можно ли использовать программу `randomsurfer.py` для вычисления рангов страниц на меньшей модели, скажем, для тысяч страниц? Чтобы ответить на этот вопрос, можно запустить несколько моделей, записать результаты для последующих исследований, а затем интерпретировать эти экспериментальные результаты. Мы действительно используем этот подход для многих научных задач (первый пример — задача о разорении игрока; второй приведен в разделе 2.4), но он может отнимать очень много времени, поскольку для достижения желаемой точности может потребоваться огромное количество экспериментов. Даже в нашем крошечном примере, как можно убедиться, он требует миллионов

итераций для получения ранга страницы с точностью в три или четыре позиции после десятичной точки. Для больших моделей необходимое количество итераций для точной оценки становится действительно огромным.

Смешение цепи Маркова. Важно помнить, что ранги страниц — это свойство модели веб, а не конкретный подход для ее расчета. Таким образом, программа `randomsurfer.py` — это только *один* из способов вычисления рангов страниц. К счастью, простая вычислительная модель на основании хорошо изученной области математики обеспечивает намного более эффективный подход, чем модель рангов страницы. Эта модель использует простые арифметические операции с двумерными матрицами, рассматривавшиеся в разделе 1.4.



Возведение в квадрат цепи Маркова

Возведение в квадрат цепи Маркова. Какова вероятность, что при случайной навигации будет осуществлен переход со страницы `i` на страницу `j` за *два* шага? Первый переход осуществляется на промежуточную страницу `k`, поэтому мы вычисляем вероятность перехода от `i` до `k`, затем от `k` до `j` для всех возможных `k`, а результаты суммируем. В нашем примере вероятность перехода от `1` до `2` за два шага является вероятностью перехода от `1` до `0` и до `2` ($0{,}02 \times 0{,}02$), плюс вероятность перехода от `1` до `1` и до `2` ($0{,}02 \times 0{,}38$), плюс вероятность перехода от `1` до `2` и до `2` ($0{,}38 \times 0{,}02$), плюс вероятность перехода от `1` до `3` и до `2` ($0{,}38 \times 0{,}02$), плюс вероятность перехода от `1` до `4` и до `2` ($0{,}20 \times 0{,}47$), что составляет в целом 0,1172. Тот же процесс работает для каждой пары страниц. *Это то же вычисление, что*

и прежде, в определении матричного умножения: элемент в ряду `i` и столбце `j` содержит результат скалярного произведения ряда `i` и столбца `j` в исходной матрице. Другими словами, результатом умножения матрицы `p[][]` на саму себя является матрица, где элементы ряда `i` и столбца `j` являются вероятностями того, что при случайной навигации произойдет переход со страницы `i` на страницу `j` за два шага. Изучение элементов матрицы перехода за два шага в нашем примере стоит потраченного времени и поможет вам лучше понять перемещение при случайной навигации. Например, наибольший элемент в квадрате находится в ряду 2 и столбце 0 и отражает тот факт, что после начала перемещений со страницы `2` есть только одна ссылка, на страницу `3`, где также есть только одна ссылка на страницу `0`. Поэтому, безусловно, наиболее вероятный результат начала навигации на странице `2` — это ее завершение на странице `0` после двух шагов. Все другие маршруты в два шага имеют больше возможных выборов и менее вероятны. Следует заметить, что это точное вычисление (ограниченное только точностью чисел Python с плавающей точкой), в отличие от программы `randomsurfer.py`, осуществляющей оценку и нуждающейся в большем количестве итераций для получения более точной оценки.

Метод степеней. Мы могли бы затем вычислить вероятности для трех шагов следующим умножением матрицы `p[][]`, для четырех — следующим умножением матрицы `p[][]` и так далее. Однако последовательное матрично-матричное умножение довольно дорого, а нас фактически интересуют матрично-векторные вычисления. В нашем примере начнем с вектора `[1.0 0.0 0.0 0.0 0.0]`, задающего начало случайной навигации со страницы 0. Умножение этого вектора на матрицу переходов дает вектор `[.02 .92 .02 .02 .02]`, содержащий вероятности того, что навигация завершится на каждой из страниц после одного шага. Следующее умножение *этого* вектора на матрицу переходов дает вектор `[.05 .04 .36 .37 .19]`, содержащий вероятности завершения навигации на каждой из страниц после *двух* шагов. Например, вероятность перехода с `0` на `2` за два шага является вероятностью перехода с `0` на `0` и на `2` (0,02 × 0,02), плюс вероятность перехода `0 – 1 – 2` (0,92 × 0,38), плюс вероятность перехода `0 – 2 – 2` (0,02 × 0,02), плюс вероятность перехода `0 – 3 – 2` (0,02 × 0,02), плюс вероятность перехода `0 – 4 – 2` (0,02 × 0,47), что составит в целом 0,36. Шаблон этих вычислений понятен: *вектор дает вероятности того, что случайная навигация с каждой страницы после t шагов является произведением соответствующего вектора для t–1 шагов и матрицы переходов.* Согласно базовой теореме предела для цепей Маркова, этот процесс сводится к тому же вектору независимо от начальной позиции; другими словами, после достаточного количества шагов вероятность завершения навигации на любой конкретной странице не зависит от отправной точки.

Программа 1.6.3 (`markov.py`) — это реализация, которую можно использовать для проверки схождения в нашем примере. Она дает те же результаты

(ранги страниц с точностью две позиции после десятичной точки), что и программа `randomsurfer.py`, но всего за 20 матрично-векторных умножений вместо десятков тысяч итераций, необходимых программе `randomsurfer.py`. Еще 20 умножений дают результаты с точностью в три позиции после десятичной точки, по сравнению с миллионами итераций для программы `randomsurfer.py`, и всего несколько дополнительных умножений дают результаты с полной точностью (см. упр. 1.6.6).

Метод степеней для вычисления рангов страниц (предельные значения вероятностей перехода)

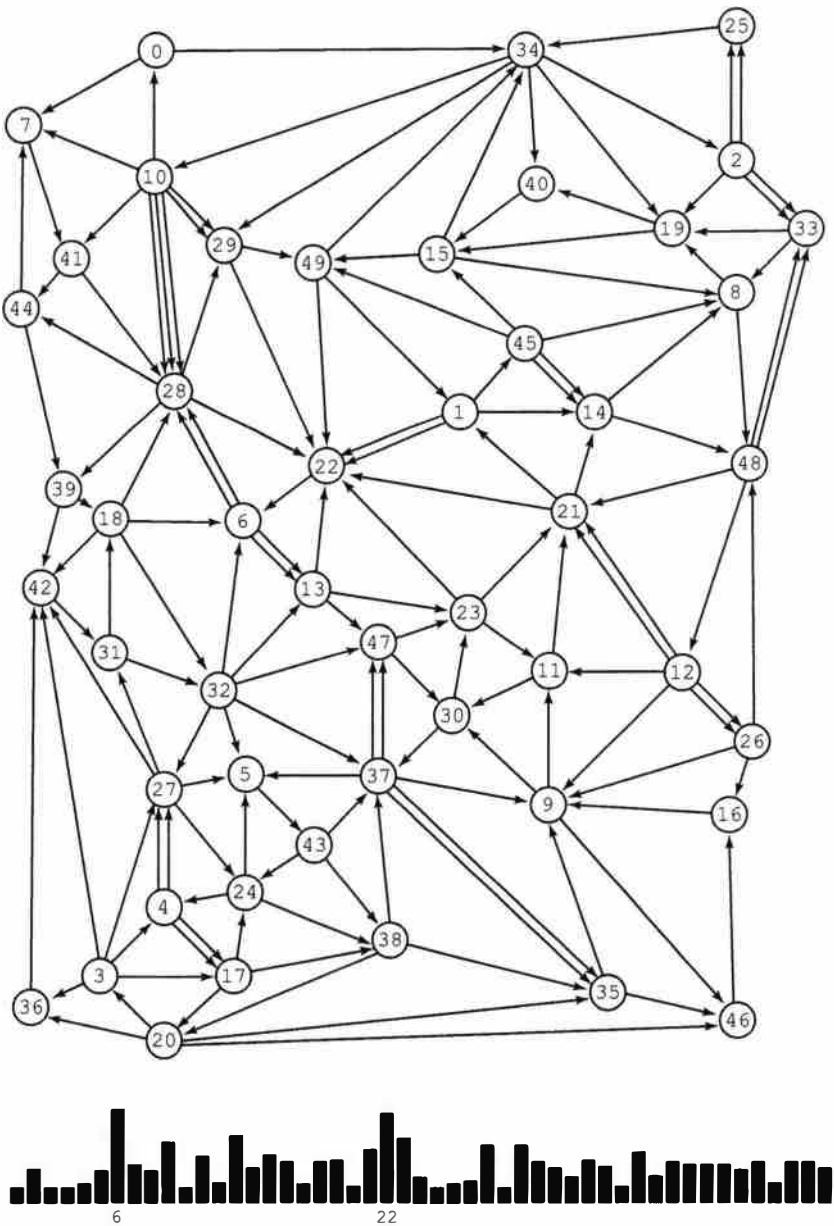
Программа 1.6.3. Смешение цепи Маркова (markov.py)

```
import sys
import stdarray
import stdio

moves = int(sys.argv[1])
n = stdio.readInt()
stdio.readInt()

p = stdarray.create2D(n, n, 0.0)
for i in range(n):
    for j in range(n):
        p[i][j] = stdio.readFloat()

ranks = stdarray.create1D(n, 0.0)
ranks[0] = 1.0
for i in range(moves):
    newRanks = stdarray.create1D(n, 0.0)
    for j in range(n):
        for k in range(n):
            newRanks[j] += ranks[k] * p[k][j]
    ranks = newRanks

for i in range(n):
    stdio.writef('%8.5f', ranks[i])
stdio.writeln()
```

`moves`	Количество итераций
`n`	Количество страниц
`p[][]`	Матрица переходов
`ranks[]`	Ранги страниц
`newRanks[]`	Новые ранги страниц

Эта программа получает из командной строки целое число `moves`, читает со стандартного ввода матрицу переходов, вычисляет вероятности перехода при случайной навигации на каждую страницу (ранги страниц) в результате `moves` матрично-векторных умножений и выводит ранги страниц на стандартное устройство вывода.

```
% python transition.py < tiny.txt | python markov.py 20
0.27245 0.26515 0.14669 0.24764 0.06806

% python transition.py < tiny.txt | python markov.py 40
0.27303 0.26573 0.14618 0.24723 0.06783
```



Ранги страниц с гистограммой в большем примере

Цепи Маркова хорошо изучены, но их влияние на веб не замечалось до 1998 года, когда у двух аспирантов, Сергея Михайловича Брина и Ларри Пейджа, хватило смелости построить цепь Маркова и вычислить вероятности достижения при случайной навигации каждой страницы *всего веб*. Их работа вызвала революцию в сетевом

поиске и стала основой для методов ранжирования страниц, используемых в основанной ими поисковой системе Google. В частности, компания периодически вычисляет вероятность посещения при случайной навигации для каждой страницы. Затем, когда вы осуществляете поиск, она выводит страницы, связанные с вашими ключевыми словами поиска, в порядке этих рангов. Ныне ранги страниц доминируют, поскольку они так или иначе соответствуют ожиданиям обычных пользователей веб, надежно обеспечивая их *подходящими* веб-страницами при поиске. Используемое вычисление отнимает чрезвычайно много времени в связи с огромным количеством страниц в веб, но результат оказывается весьма выгодным и стоящим затрат. Метод, используемый в программе `markov.py`, намного эффективнее моделирования поведения при случайной навигации, но он все еще слишком медленный, чтобы фактически вычислять вероятности для огромной матрицы, соответствующей всем страницам в веб. Для этого вычисления необходима лучшая структура данных графов (подробно об этом — в главе 4).

Уроки. Выработка полного понимания модели случайной навигации выходит за рамки этой книги. Наша задача — продемонстрировать ее применение, задействующее немного больше кода, чем короткие программы, используемые нами для демонстрации конкретных концепций. Какие же уроки мы можем извлечь из изучения данного случая?

Полная компьютерная модель. Встроенные типы данных, совместно с условными выражениями, циклами, массивами и стандартным вводом/выводом, позволяют решать интересные задачи всех видов. Действительно, это основа теоретической информатики, и данной модели достаточно для осуществления любых вычислений, которые могут быть выполнены на любом подходящем вычислительном устройстве. В следующих двух главах мы обсудим два критически важных способа расширения модели, позволяющих существенно сократить время и усилия по разработке больших и сложных программ.

Управляемый данными код. Концепция использования потоков стандартного ввода и вывода, а также хранения данных в файлах весьма полезна. Можно составлять фильтры для преобразования одного вида ввода к другой, генераторы, способные создавать огромные входные файлы для исследования, и программы, способные обработать широкое разнообразие моделей. Мы можем сохранять данные для архивирования или последующего использования. Мы можем также обрабатывать данные, полученные из какого-либо другого источника, а затем сохранять их в файле на собственном компьютере или на удаленном веб-сайте. Концепция управляемого данными кода — это простой и гибкий способ обеспечения целого комплекса действий.

Точность может быть неуловима. Было бы ошибкой предполагать, что программа дает точные ответы просто потому, что она способна выводить числа с точностью до многих позиций после десятичной точки. Зачастую самые большие трудности создает гарантия точности ответов.

Равномерные случайные числа — это только начало. Когда в неофициальном разговоре мы упоминаем случайное поведение, то зачастую подразумеваем нечто более сложное, чем "каждое значение одинаково вероятно" (модель, обеспечиваемая функцией `random.random()`). Большинство рассматриваемых нами задач задействовало случайные числа из других распределений, таких как `randomsurfer.py`.

Вопросы эффективности. Ошибкой было бы также подразумевать, что ваш компьютер настолько быстрый, что может осуществлять *любые* вычисления. Некоторые задачи требуют намного больше вычислительных мощностей, чем другие. Глава 4 полностью посвящена обсуждению оценки эффективности составляемых вами программ. Отложим подробное рассмотрение таких задач до тех пор, пока не получим общее представление о требованиях к производительности своих программ.

Возможно, важнейший урок, извлеченный из создания программ для решения сложных задач, как в примерах данного раздела, в том, что *отладка трудна.* Отполированные программы в книге маскируют этот урок, но можете быть уверены, что каждая из них — результат долгих проверок, устранения ошибок и многочисленных запусков с разнообразными входными данными. Обычно мы избегаем описания ошибок и процесса их устранения, поскольку это делает книгу скучной и чрезмерно сосредоточивает внимание на плохом коде, но на сайте книги вы найдете несколько примеров с описаниями и упражнениями.

Упражнения

1.6.1. Измените программу `transition.py` так, чтобы получать вероятность прямого перехода из командной строки, и используйте модифицированную версию для исследования влияния на ранги страниц переключения на правила 80 на 20 или 95 на 5.

1.6.2. Измените программу `transition.py` так, чтобы игнорировать влияние множественных ссылок. Таким образом, если есть несколько ссылок с одной страницы на другую, то они считаются одной ссылкой. Создайте пример, демонстрирующий, как эта модификация может изменить порядок рангов страниц.

1.6.3. Измените программу `transition.py` так, чтобы соответствующие страницам без исходящих ссылок ряды заполнялись значением 1/n.

1.6.4. Создающий случайные ходы фрагмент кода программы `randomsurfer.py` дает сбой, если вероятности в ряду `p[page]` не составляют в целом 1. Объясните, что происходит в этом случае, и предложите способ устранения проблемы.

1.6.5. Определите (в степенях числа 10) количество итераций, необходимых программе `randomsurfer.py` для вычисления рангов страниц с точностью в четыре позиции после десятичной точки и пять позиций после десятичной точки для файла `tiny.txt`.

1.6.6. Определите количество итераций, необходимых программе `markov.py` для вычисления рангов страниц с точностью в три, четыре и десять позиций после десятичной точки для файла `tiny.txt`.

1.6.7. Загрузите с сайта книги файл `medium.txt` (содержащий рассматриваемый в этом разделе пример на 50 страниц) и добавьте в него ссылки *со* страницы 23 *на* любую другую страницу. Выясните влияние этого на ранги страниц и обдумайте результат.

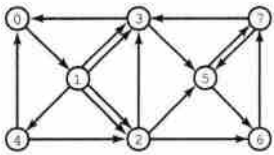
1.6.8. Добавьте в файл `medium.txt` (см. предыдущее упражнение) ссылки *на* страницу 23 *с* любой страницы. Выясните результат для рангов страниц и обдумайте его.

1.6.9. Предположим, ваша страница в файле `medium.txt` — это страница 23. Можно ли добавить ссылку с вашей страницы на другую страницу, чтобы *поднять* ранг *своей* страницы?

1.6.10. Предположим, ваша страница в файле `medium.txt` — это страница 23. Можно ли добавить ссылку с вашей страницы на другую страницу, которая *понизит* ранг *той* страницы?

1.6.11. Используйте программы `transition.py` и `randomsurfer.py` для определения вероятностей переходов в представленном ниже примере на восемь страниц.

1.6.12. Используйте программы `transition.py` и `markov.py` для определения вероятностей переходов в представленном ниже примере на восемь страниц.



Пример на восемь страниц

Практические упражнения

1.6.13. *Возведение матрицы в квадрат.* Составьте программу, подобную `markov.py`, вычисляющую ранги страниц последовательным возведением в квадрат матрицы, как при вычислении последовательности p, p^2, p^4, p^8, p^{16} и т.д. Проверьте, чтобы все ряды в матрице сходились к тем же значениям.

1.6.14. *Случайный веб.* Составьте генератор для программы `transition.py`, получающий на входе количество страниц `n`, количество ссылок `m` и выводящий на стандартный вывод `n` сопровождаемое `m` случайных пар целых чисел от `0` до `n-1`. (Более реалистичные модели веб обсуждаются в разделе 4.5.)

1.6.15. *Концентраторы и авторитеты* (алгоритм HITS). Добавьте к своему генератору из предыдущего упражнения фиксированное количество *концентраторов* (hub), обладающих ссылками на 10 процентов выбранных наугад страниц, и *авторитетов* (authorities), ссылки на которых есть у 10 процентов страниц. Вычислите ранги страницы. Что занимает место выше, концентраторы или авторитеты?

1.6.16. *Ранги страниц.* Разработайте массив страниц и ссылок, где у страниц высшего ранга меньше ссылок, указывающих на них, чем у других страниц.

1.6.17. *Время достижения.* Время достижения страницы — это ожидаемое количество переходов при случайной навигации для посещения страницы. Проведите эксперименты, чтобы оценить время достижения страниц из файла `tiny.txt`, и сравните результаты с рангами страницы, чтобы сформулировать гипотезу об их отношениях. Проверьте свою гипотезу на файле `medium.txt`.

1.6.18. *Время покрытия.* Составьте программу, оценивающую время, необходимое при случайной навигации для посещения каждой страницы по крайней мере однажды, начиная со случайной страницы.

1.6.19. *Графическое моделирование.* Создайте графическую модель, где размер точки, представляющей каждую страницу, пропорционален ее рангу. Чтобы сделать программу управляемой данными, разработайте формат файла, учитывающий координаты отображения каждой страницы. Проверьте свою программу на файле `medium.txt`.

Глава 2 Функции и модули

2.1. Определение функций 212
2.2. Модули и клиенты.....247
2.3. Рекурсия.....285
2.4. Случай из практики: просачивание 316

Эта глава посвящена конструкции, имеющей существенное влияние и на контроль потока, и на условные выражения, и на циклы. Речь пойдет о *функции* (function), позволяющей передавать управление между различными частями кода. Функции важны, поскольку они позволяют четко разделять задачи в пределах программы, а следовательно, предоставляют общий механизм, обеспечивающий возможность многократного использования кода. Определение и использование функций является центральным компонентом программирования на языке Python.

Когда приходится работать со многими функциями, их можно сгруппировать в *модуль* (module). Используя модули, можно разделить вычислительную задачу на подзадачи разумного размера. В этой главе вы узнаете, как создавать собственные модули и как их использовать в стиле *модульного программирования* (modular programming). В частности, мы рассмотрим здесь модули для создания случайных чисел, анализа данных и ввода-вывода массивов. Модули значительно расширяют набор операций, доступных для использования в наших программах.

Мы уделим особое внимание функциям, передающим управление самим себе. Это — *рекурсия* (recursion). Сначала рекурсия может показаться интуитивно непонятной, но она позволяет разрабатывать простые программы, способные решать сложные задачи, решить которые иначе было бы намного труднее.

Каждый раз, когда вы можете четко разделить задачи в пределах вычисления, так и поступайте. В данной главе мы регулярно повторяем эту мантру и заканчиваем главу примером, демонстрирующим, как можно решить сложную задачу программирования, разделив ее на меньшие подзадачи, разработав независимые модули, которые взаимодействуют друг с другом, решая эти подзадачи. Повсюду в этой главе мы используем функции и модули, разработанные ранее, чтобы подчеркнуть удобство модульного программирования.

2.1. Определение функций

Программы этого раздела...

Программа 2.1.1. Гармонические числа (повторно) (harmonicf.py) 215
Программа 2.1.2. Гауссовы функции (gauss.py) 225
Программа 2.1.3. Коллекционер купонов (повторно) (coupon.py) 226
Программа 2.1.4. Проигрывание мелодии (повторно) (playthattunedeluxe.py) 235

Код, вызывающий функции Python, вы составляли с самого начала этой книги, от вывода строк при помощи функции `stdio.writeln()`, использования функций преобразования типов, таких как `str()` и `int()`, применения таких математических функций, как `math.sqrt()`, и до использования функций модулей `stdio`, `stddraw` и `stdaudio`. В этом разделе вы узнаете, как определять и вызывать собственные функции.

В математике функция сопоставляет исходные значения одного типа (домена) с выходными значениями другого типа (диапазона). Например, квадратная функция $f(x) = x^2$ сопоставляет число 2 с числом 4, число 3 — с числом 9, число 4 — с числом 16 и т.д. С самого начала мы работаем с функциями Python, реализующими математические функции, поскольку они хорошо знакомы нам. Модуль Python `math` реализует множество стандартных математических функций, но ученые и инженеры работают с весьма широким разнообразием математических функций, которые не могут быть включены в модуль все. В начале этого раздела вы узнаете, как реализовать и использовать такие функции самостоятельно.

Позже вы узнаете, что с функциями Python можно сделать куда больше, чем реализовать математические функции: они способны получать на входе строки и данные других типов (или их диапазоны), у них могут быть побочные эффекты, такие как вывод. В этом разделе мы также рассмотрим использование функций Python для организации программ, чтобы упростить решение сложных задач программирования.

С этого момента мы используем термин *функция* для обозначения *функций Python* или *математических функций* в зависимости от контекста. Более конкретную терминологию мы используем только в случае, если этого требует контекст.

Функции обеспечивают ключевую концепцию, определяющую ваш подход к программированию с этого момента и далее: *всякий раз, когда можете четко разделить задачи в пределах вычисления, так и поступайте*. Мы будем подчеркивать этот пункт повсюду в данном разделе и укреплять это мнение в остальной части главы (и остальной части книги также). Когда вы пишете текст, вы разбиваете его на параграфы; когда вы составляете программу, вы разбиваете ее на функции. Разделение большей задачи на меньшие весьма важно и при программировании, и при написании текста, поскольку это весьма облегчает отладку, обслуживание и обеспечивает многократное использование кода, что критически важно для разработки хорошего программного обеспечения.

Определение и использование функций. Как известно из уже полученного опыта использования функций, результат их вызова прост и понятен. Например, помещение вызова `math.sqrt(a-b)` в код программы равнозначно помещению в него *возвращаемого значения* (return value), создаваемого функцией Python `math.sqrt()` при передаче ей выражения `a - b` в качестве *аргумента* (argument). Это настолько интуитивно понятно, что едва ли имеет смысл его комментировать. Если задуматься о том, что система должна сделать для получения такого эффекта, то станет ясно, что это подразумевает управление потоком выполнения программы. Значение способности изменять поток выполнения так же велико, как и в условных выражениях или циклах.

При создании функции в программе Python используется оператор `def`, определяющий сигнатуру функции, затем следуют операторы, составляющие функцию. Подробности мы рассмотрим вскоре, а пока начнем с простого примера, иллюстрирующего влияние функции на поток выполнения. Наш первый пример, программа 2.1.1 (`harmonicf.py`), включает функцию `harmonic()`, получающую аргумент `n` и вычисляющую `n`-е гармоническое число (см. программу 1.3.5). Он также иллюстрирует типичную структуру программы Python, имеющую три компонента.

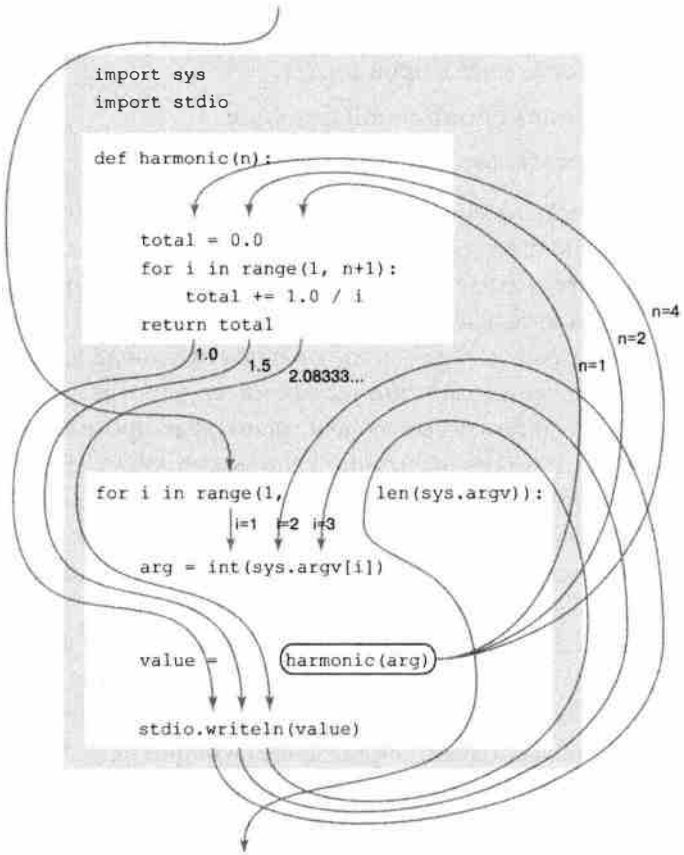
- Последовательность операторов `import`.
- *Последовательность* определений функции.
- Произвольный *глобальный код*, или тело программы.

В программе 2.1.1 есть два оператора `import`, одно определение функции и четыре строки произвольного глобального кода. Python выполняет глобальный код, когда мы запускаем программу, введя в командной строке `python harmonicf.py`; а этот глобальный код вызывает функцию `harmonic()`, определенную ранее.

Реализация программы `harmonicf.py` предпочтительнее для нашей первоначальной задачи вычисления гармонических чисел (программа 1.3.5), поскольку она четко отделяет две главные задачи, решаемые программой: вычисление гармонических чисел и взаимодействие с пользователем. (В иллюстративных целях мы сделали часть взаимодействия с пользователем немного сложней, чем в программе 1.3.5.) *Всякий раз, когда можете четко разделить задачи в пределах вычисления, так и поступайте.* Впоследствии мы подробнее рассмотрим, как именно программа `harmonicf.py` достигает этой цели.

Контроль потока. Схема, приведенная ниже, иллюстрирует управление потоком для команды `python harmonicf.py 1 2 3`. Сначала Python обрабатывает операторы `import`, делая доступными для программы все средства, определенные в модулях `sys` и `stdio`. Затем Python обрабатывает определение функции `harmonic()` в строках 4–8, но *не выполняет функцию*, он выполнит ее только при вызове. После определения функции Python выполняет первый оператор глобального кода, оператор `for`, выполнение которого продолжается обычным образом, пока не встретится оператор `value = harmonic(arg)`. Вначале вычисляется выражение `harmonic(arg)`, когда

параметр `arg` содержит аргумент `1`. Для этого управление передается функции `harmonic()` — поток выполнения проходит к коду в определении функции. Python инициализирует "параметрическую" переменную `n` значением `1` и "локальную" переменную `total` значением `0.0`, а затем выполняет цикл `for` в пределах функции `harmonic()`, который заканчивается после одной итерации с `total`, равным `1.0`. Затем Python выполняет оператор `return` в конце определения функции `harmonic()`, заставляя управление вернуться назад к вызывающему оператору `value = harmonic(arg)`, продолжиться с того места, где оно остановилось, но с выражением `harmonic(arg)`, замененным значением `1.0`. Таким образом, Python присвоит переменной `value` значение `1.0` и запишет его в стандартный вывод. Затем выполнит следующую итерацию цикла и еще раз вызовет функцию `harmonic()` при `n`, инициализированной значением `2`, что приведет к выводу `1.5`. Затем процесс повторяется в третий раз со значением `4` параметра `arg` (а затем и `n`), что приведет к выводу значения `2.083333333333333`. И наконец, цикл `for` завершается и завершается весь процесс. Следующая схема демонстрирует, как простой код маскирует довольно запутанный поток.



Поток выполнения при вызове `python harmonicf.py 1 2 4`

Программа 2.1.1. Гармонические числа (повторно) (harmonicf.py)

```
import sys
import stdio

def harmonic(n):
    total = 0.0
    for i in range(1, n+1):
        total += 1.0 / i
    return total

for i in range(1, len(sys.argv)):
    arg = int(sys.argv[i])
    value = harmonic(arg)
    stdio.writeln(value)
```

n	Параметрическая переменная
i	Индекс цикла
total	Возвращаемое значение

i	Индекс аргумента
arg	Аргумент
value	Гармоническое число

Эта программа пишет в стандартный вывод гармонические числа, определенные как аргументы командной строки. Программа определяет функцию `harmonic()`, получающую аргумент `n` типа `int` и вычисляющую n-е гармоническое число $1 + 1/2 + 1/3 + \ldots + 1/n$.

```
% python harmonicf.py 1
2 4
1.0
1.5
2.083333333333333
```

```
% python harmonicf.py
10 100 1000 10000
2.9289682539682538
5.187377517639621
7.485470860550343
9.787606036044348
```

Предупреждение о сокращении. Мы продолжаем использовать для функций и вызовов функций сокращения, описанные в разделе 1.2. Например, мы могли бы написать "вызов функции `harmonic(2)` возвращает значение `1.5`", вместо более точного, но подробного "в результате передачи функции `harmonic()` ссылки на объект типа `int` со значением `2` возвращается ссылка на объект типа `float` со значением `1.5`". Мы стремимся быть по возможности краткими и точными, а подробности — только по мере необходимости.

Свободная трассировка вызовов функции. Один из простейших подходов к отслеживанию потока выполнения при вызове функций подразумевает вывод каждой функцией при вызове своего имени и аргументов, а также ее возвращаемого значения непосредственно перед его возвращением. Вывод осуществляется с соответствующими отступами. Вывод значений переменных и результатов вызовов улучшает наглядность трассировки программы, используемой начиная с раздела 1.2. Свободная трассировка данного примера представлена ниже.

```
i = 1
arg = 1
harmonic(1)
    total = 0.0
    total = 1.0
    return 1.0
value = 1.0
i = 2
arg = 2
harmonic(2)
    total = 0.0
    total = 1.0
    total = 1.5
    return 1.5
value = 1.5
i = 3
arg = 4
harmonic(4)
    total = 0.0
    total = 1.0
    total = 1.5
    total = 1.8333333333333333
    total = 2.083333333333333
    return 2.083333333333333
value = 2.083333333333333
```

Свободная трассировка при вызове `python harmonicf.py 1 2 4`

Добавленный отступ отражает представление потока выполнения и помогает проверить соответствие результата вызова каждой функции нашим ожиданиям. Обычно добавление вызовов функции `stdio.writef()` для отслеживания потока выполнения *любой* программы является прекрасным подходом для понимания ее работы. Если возвращаемые значения соответствуют ожиданиям, нет смысла подробно отслеживать код функций, что существенно экономит объем работ.

Код в остальной части этой главы будет сосредоточен на создании и использовании функций, поэтому имеет смысл подробнее рассмотреть их основные свойства и, в частности, связанную с функциями терминологию. Затем рассмотрим несколько примеров реализации функций и их применения.

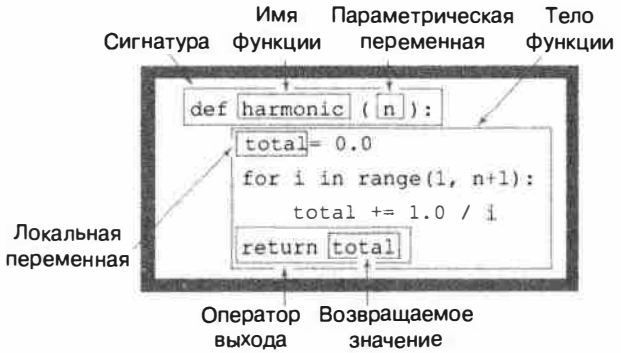
Основная терминология. Как обычно, имеет смысл обсудить различия между абстрактными концепциями и реализующими их механизмами Python (оператор Python `if` реализует условное выражение, оператор `while` реализует цикл и так далее). Идея математической функции объединяет несколько концепций, и для каждой из них есть соответствующая конструкция Python, как отмечено в таблице, приведенной далее. Можете быть уверены, что этот формализм хорошо служил математикам в течение многих столетий (и хорошо послужит ныне программистам), поэтому мы воздержимся от подробного рассмотрения всех его преимуществ и сосредоточимся на тех, которые помогут вам научиться программировать.

Концепция	Конструкция Python	Описание
Функция	Функция	Сопоставление
Входное значение	Аргумент	Ввод функции
Выходное значение	Возвращаемое значение	Вывод функции
Формула	Тело функции	Определение функции
Независимая переменная	Параметрическая переменная	Символьное знакоместо для входного значения

При использовании символьного имени в формуле, определяющей такую математическую функцию, как $f(x) = 1 + x + x^2$, символ x является знакоместом для некоего входного значения, замена которого в формуле конкретным значением определит выходное значение. Язык Python использует в качестве символьного

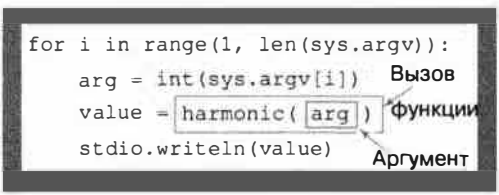
знакоместа *параметрическую переменную* (parameter variable), а конкретное входное значение, обрабатываемое функцией, является *аргументом* (argument).

Определение функции. Первая строка определения функции, *сигнатура* (signature), предоставляет имя функции и все параметрические переменные. Сигнатура состоит из ключевого слова `def`; *имени функции* (function name); последовательности из любого количества имен параметрических переменных, отделенных запятыми и заключенных в круглые скобки; и двоеточия. Операторы с отступом после сигнатуры — это *тело функции* (function body). Тело функции может состоять из операторов, обсуждавшихся в главе 1. Оно может также содержать *оператор выхода* (return statement), возвращающий управление в место вызова функции и возвращающий результат вычисления или *возвращаемое значение* (return value). В теле функции можно определить *локальные переменные* (local variable), доступные только в той функции, в которой они определены.



Анатомия определения функции

Вызовы функции. Как уже упоминалось, вызов функции Python — это не более чем имя функции, сопровождаемое ее аргументами, разделенными запятыми и заключенными в круглые скобки. Форма аналогична общепринятой для математических функций. В разделе 1.2 также упоминалось, что каждый аргумент может быть выражением, результат вычисления которого передается на вход функции. Когда функция завершает работу, возвращаемое ей значение занимает место вызова функции, как будто это было значение переменной (возможно, даже в пределах выражения).



Анатомия вызова функции

Несколько аргументов. Как и математическая функция, функция Python может иметь больше одной параметрической переменной, а следовательно, больше одного аргумента. В сигнатуре функции имя каждой параметрической переменной отделяется запятой. Например, следующая функция вычисляет длину гипотенузы прямоугольного треугольника со сторонами длиной `a` и `b`:

```
def hypot(a, b)
    return math.sqrt(a*a + b*b)
```

Несколько функций. В файле `.py` можно определить столько функций, сколько необходимо. Функции независимы, если они не вызывают друг друга. В файле они могут присутствовать в любом порядке:

```
def square(x):
    return x*x

def hypot(a, b):
    return math.sqrt(square(a) + square(b))
```

Однако определение функции должно присутствовать перед любым глобальным кодом, который ее использует. Именно поэтому типичная программа Python содержит: (1) операторы `import`, (2) определения функций и (3) произвольный глобальный код, причем именно в таком порядке.

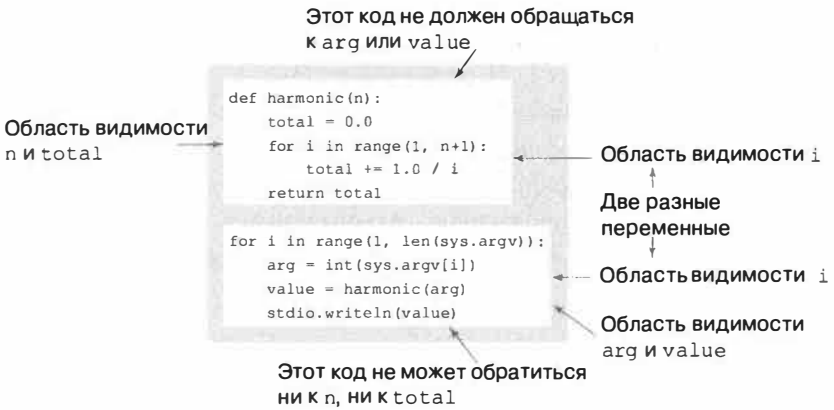
Несколько операторов выхода. Операторы `return` можно помещать в функцию везде, где это необходимо; они возвращают управление вызывающей стороне, как только встретится первый оператор `return`. Следующая функция, проверяющая, является ли число простым, — это пример функции, в которой вполне уместно использование нескольких операторов `return`:

```
def isPrime(n):
    if n < 2: return False
    i = 2
    while i*i <= n:
        if n % i == 0: return False
        i += 1
    return True
```

Одно возвращаемое значение. Функция Python возвращает вызывающей стороне только одно значение (точнее, ссылку на один объект). Это правило не является столь уж ограничивающим, как могло бы показаться, поскольку типы данных Python способны содержать больше информации, чем одно число, логическое значение или строка. Например, далее в этом разделе будет продемонстрировано, что в качестве возвращаемого значения можно использовать массивы.

Область видимости. Область видимости (scope) переменной — это набор операторов, способных обращаться к этой переменной непосредственно. Область видимости локальных и параметрических переменных функции ограничивается самой функцией; область видимости переменной, определенной в глобальном

коде — *глобальной переменной* (global variable), — ограничивается файлом `.py`, содержащим эту переменную. Следовательно, глобальный код не может обратиться к локальной или параметрической переменной функции. Функция также не может обратиться к локальной или параметрической переменной, определенной в другой функции. Когда функция определяет локальную (или параметрическую) переменную с тем же именем, что и у глобальной переменной (как `i` в программе 2.1.1), то в функции имя переменной относится к значению только локальной (или параметрической) переменной, но не глобальной.



Область видимости локальных и параметрических переменных

Руководящий принцип разработки программного обеспечения подразумевает определение каждой переменной так, чтобы ее область видимости была как можно меньше. Одной из важнейших причин использования функций является обеспечение независимости одной части программы от другой. Таким образом, код функции *может* обращаться к глобальным переменным, но он *не должен* этого делать: все взаимодействие вызывающей стороны с функцией должно осуществляться через параметрические переменные функции, а вся связь функции с вызывающей стороной — через возвращаемое значение функции. В разделе 2.2 приведена методика устранения большей части глобального кода, позволяющая ограничить области видимости и потенциальные непредвиденные взаимодействия.

Стандартные аргументы. Функция Python может определить аргумент как *необязательный*, задав для него *стандартное значение* (default value). Если в вызове функции пропустить необязательный аргумент, то Python заменяет его стандартным значением. Несколько примеров этой возможности уже встречалось нам. Например, функция `math.log(x, b)` возвращает логарифм `x` по основанию `b`. Если пропустить второй аргумент `b`, т.е. применить вызов `math.log(x)`, то функция `math.e` использует стандартное значение аргумента `b` и вернет

натуральный логарифм x. Может показаться, что в модуле math есть две разные функции логарифма, но фактически есть только одна, с необязательным аргументом и стандартным значением.

В пользовательской функции также можно определить необязательный аргумент со стандартным значением. Для этого в сигнатуре функции достаточно расположить знак равенства после параметрической переменной и указать стандартное значение. В сигнатуре функции можно определить несколько необязательных аргументов, но все они должны располагаться после всех обязательных аргументов.

Рассмотрим, например, обобщенную задачу вычисления *энного обобщенного гармонического числа порядка r*: $H_{n,r} = 1 + 1/2^r + 1/3^r + \ldots + 1/n^r$. Например, $H_{1,2} = 1$, $H_{2,2} = 5/4$, а $H_{2,2} = 49/36$. Обобщенные гармонические числа тесно связаны с дзета-функцией Римана из теории чисел. Обратите внимание, что *энное* обобщенное гармоническое число при порядке $r = 1$ равно *энному* гармоническому числу. Поэтому вполне резонно использовать для r стандартное значение 1, если вызывающая сторона пропустит второй аргумент. Для этого укажем в сигнатуре r=1:

```
def harmonic(n, r=1):
    total = 0.0
    for i in range(1, n+1):
        total += 1.0 / (i ** r)
    return total
```

В этом определении вызов harmonic(2, 2) возвращает 1.25, а вызовы harmonic(2, 1) и harmonic(2) возвращают 1.5. Клиенту может показаться, что это две разные функции, одна с одним аргументом, вторая с двумя аргументами, но на самом деле это одна реализация.

Побочные эффекты. В математике функция сопоставляет одно или несколько входных значений некоему выходному значению. В программировании множество функций используют ту же модель: они получают один или несколько аргументов, а их единственная цель — возвратить результирующее значение. *Чистая функция* (pure function) всегда возвращает те же результирующие значения, получив те же аргументы, причем без заметных *побочных эффектов* (side effect), таких как ввод, вывод или иное изменение состояния системы. До сих пор в этом разделе рассматривались только чистые функции.

Но в программировании популярно также определение функций, создающих побочные эффекты. Фактически мы очень часто определяем функции, единственная цель которых — побочные эффекты. Явный оператор return в такой функции необязателен: управление возвращается вызывающей стороне после того, как Python выполнит последний оператор функции. Функции без явно определенного возвращаемого значения фактически возвращают специальное значение None, которое обычно игнорируется.

Например, у функции `stdio.write()` есть побочный эффект записи переданного аргумента на стандартное устройство вывода (и нет никакого определенного возвращаемого значения). Точно так же у следующей функции есть побочный эффект рисования треугольника на стандартном графическом устройстве (и также нет никакого определенного возвращаемого значения):

```
def drawTriangle(x0, y0, x1, y1, x2, y2):
    stddraw.line(x0, y0, x1, y1)
    stddraw.line(x1, y1, x2, y2)
    stddraw.line(x2, y2, x0, y0)
```

Вообще-то, это плохой стиль — создавать функцию, которая производит побочный эффект и возвращает значение. Одно известное исключение — функция, читающая ввод. Например, функция `stdio.readInt()` производит побочный эффект (читает одно целое число из стандартного ввода) и возвращает значение (целое число).

Контроль соответствия типов. В математике функция определяет и домен, и диапазон. Для гармонических чисел, например, домен — это положительные целые числа, а диапазон — положительные вещественные числа. В языке Python мы не определяем типы параметрических переменных и типы возвращаемого значения. Пока Python способен выполнить все операции в пределах функции, он их выполняет и возвращает значение.

Если Python не может применить операцию к данному объекту из-за неправильного типа, он передает сообщение об ошибке времени выполнения, указывающее на недопустимость типа. Например, если вызвать определенную ранее функцию `square()` с аргументом типа `int`, то результат будет типа `int`; если вызвать ее с аргументом типа `float`, то результат будет типа `float`. Но если вызывать ее со строковым аргументом, то во время выполнения произойдет ошибка `TypeError`.

Эта гибкость — весьма популярное средство Python (известное как *полиморфизм* (polymorphism)), поскольку оно позволяет определить одну функцию для использования с объектами разных типов. При вызове функции с аргументами непредвиденных типов это может привести к непредвиденным ошибкам. В принципе, для предотвращения таких ошибок можно добавить код проверки типов передаваемых функции данных. Но, как и большинство программистов Python, мы воздержимся от этого. В этой книге мы рекомендуем *всегда знать типы* передаваемых функциям данных, и рассматриваемые здесь функции соответствуют этой философии, которая по общему признанию конфликтует с тенденцией Python к полиморфизму. Подробно эту проблему мы обсудим в разделе 3.3.

Таблица, приведенная ниже, резюмирует наше обсуждение на примерах определений функции, рассмотренных до сих пор. Чтобы проверить свое понимание, уделите время тщательному изучению этих примеров.

Примеры кода реализации функции

Проверка простоты чисел

```
def isPrime(n):
    if n < 2: return False
    i = 2
    while i*i <= n:
        if n % i == 0: return False
        i += 1
    return True
```

Гипотенуза прямоугольного треугольника

```
def hypot(a, b)
    return math.sqrt(a*a + b*b)
```

Обобщенное гармоническое число

```
def harmonic(n, r=1):
    total = 0.0
    for i in range(1, n+1):
        total += 1.0 / (i ** r)
    return total
```

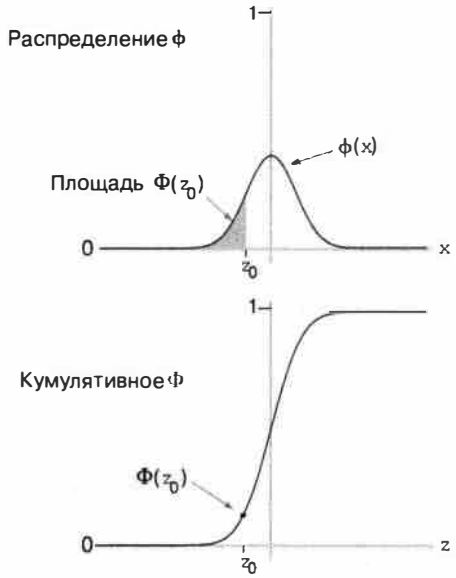
Рисование треугольника

```
def drawTriangle(x0, y0, x1, y1, x2, y2):
    stddraw.line(x0, y0, x1, y1)
    stddraw.line(x1, y1, x2, y2)
    stddraw.line(x2, y2, x0, y0)
```

Реализация математических функций. Почему бы не использовать только встроенные функции Python и те, которые определены в стандартных или дополнительных модулях Python? Например, почему бы не использовать функцию `math.hypot()` вместо определения собственной функции `hypot()`? Мы *действительно* используем такие функции, когда в этом есть смысл (поскольку они работают быстрее или точнее). Нам может понадобиться любая функция без ограничений, но в стандартных модулях Python и модулях расширения определено, безусловно, только конечное количество функций. Если необходимой функции нет среди определенных в стандартных модулях Python и модулях расширения, то ее следует определить самостоятельно.

В качестве примера рассмотрим код, применимый для задачи, знакомой и важной для многих потенциальных студентов колледжей в Соединенных Штатах. За последний год более 1 миллиона учеников сдавало Академический оценочный тест (Scholastic Aptitude Test — SAT). Тест состоит из двух главных разделов: анализ текста и математика. Диапазон баллов в каждом разделе — от 200 (самый низкий) до 800 (самый высокий); таким образом, общий диапазон результатов экзамена составляет от 400 до 1600. Многие университеты учитывают эти результаты при принятии важных решений. Например, Национальной ассоциации студенческого спорта (National Collegiate Athletic Association — NCAA) нужны спортивные студенты, а следовательно, и многим университетам тоже. Для них достаточно по крайней мере 820 баллов (из 1600), а в некоторых академических вузах минимальный балл для стипендии составляет 1500 (из 1600).

Какой процент сдавших тест не подходит для атлетов? Какой процент имеет шанс на стипендию?



Функции Гауссовой вероятности

Точные ответы на эти вопросы дают две статистические функции. Функция *стандартной нормальной (Гауссовой) плотности вероятностей* характеризуется знакомой колоколообразной кривой и определяется формулой $\varphi(x) = e^{-x^2/2} / \sqrt{2\pi}$. Стандартная нормальная (Гауссова) функция *кумулятивного распределения* $\Phi(z)$ определяет площадь ниже кривой, определенной функцией $\phi(x)$ выше оси X и налево от вертикальной линии $x = z$. Эти функции играют важную роль в науке, технике и финансах, поскольку они точно моделируют реальный мир и являются основанием для понимания экспериментальных ошибок. В частности, как известно, они точно описывают ежегодно публикуемое распределение экзаменационных отметок как функции среднего (среднее значение баллов) и среднеквадратичного отклонения (квадратный корень среднего из квадратов разниц между каждым баллом и средним). Имея среднее μ и среднеквадратичное отклонение σ экзаменационных баллов, процент абитуриентов с баллом ниже заданного значения z стремится к функции $\Phi(z, \mu, \sigma) = \Phi((z - \mu)/\sigma)$. Функций для вычисления ϕ и Φ в модуле Python `math` нет, поэтому их придется реализовать самостоятельно.

Замкнутая форма. В самом простом случае есть замкнутая форма математической формулы, определяющая нашу функцию в терминах функций, реализованных в модуле Python `math`. Эта ситуация имеет место для функции φ — модуль `math` включает функции вычисления экспонент и квадратного корня (а также константу π). Таким образом, реализовать функцию `pdf()`, соответствующую математическому определению, довольно просто. Для удобства программа 2.1.2 (`gauss.py`) использует стандартные аргументы $\mu = 0$ и $\sigma = 1$ и фактически вычисляет уравнение $\phi(x, \mu, \sigma) = \varphi((x - \mu)/\sigma)/\sigma$.

Отсутствие замкнутой формы. Если известной формулы нет, то для вычисления значения функции, возможно, понадобится более сложный алгоритм. Эта ситуация имеет место для функции Φ — для нее нет никакого выражения замкнутой формы. Алгоритмы вычисления значения функции иногда непосредственно происходят от аппроксимаций рядов Тейлора, но разработка надежных и точных реализаций математических функций — это наука и искусство. Она должна быть

тщательно проработана с использованием математических знаний, выработанных за несколько прошлых столетий. Для вычисления функции Ф известно много разных подходов. Например, аппроксимация рядов Тейлора к соотношению Ф и ϕ оказывается весьма эффективным основанием для вычисления функции

$$\Phi(z) = S + \phi(z) \ (z + z^3/3 + z^5/(3 \times 5) + z^7/(3 \times 5 \times 7) + ...).$$

Эта формула легко преобразуется в код Python для функции `cdf()` программы 2.1.2. Для малого z (соответственно большого) значение чрезвычайно близко к 0 (соответственно 1), таким образом, код непосредственно возвращает значение 0 (соответственно 1), в противном случае он использует ряд Тейлора для суммирования, пока сумма не сойдется. Также для удобства программа 2.1.2 фактически вычисляет выражение $\Phi(z, \mu, \sigma) = \Phi((z - \mu) / \sigma)$, используя стандартные значения $\mu = 0$ и $\sigma = 1$.

Запуск программы `gauss.py` с соответствующими аргументами командной строки покажет, что примерно 17% сдавших тест за год не проходят даже как атлеты, поскольку среднее составило 1019, а среднеквадратичное отклонение — 209. В том же году примерно 1% получит академическую стипендию.

Вычисления с использованием различных математических функций играет важную роль в науке и технике. В очень многих приложениях для создания необходимых функций можно обойтись функциями из модуля Python `math`, как в функции `pdf()`, или использовать аппроксимации рядов Тейлора, или иную форму упрощения вычислений, как в функции `cdf()`. Действительно, поддержка таких вычислений сыграла главную роль в эволюции вычислительных систем и языков программирования.

Использование функций для организации кода. Кроме вычисления математических функций, процесс вычисления результирующего значения важен как общая методика организации контроля потока в *любом* вычислении. Это простой пример чрезвычайно важного руководящего принципа любого хорошего программиста: *всякий раз, когда можете четко разделить задачи в пределах вычисления, так и поступайте.*

Функции — естественный и универсальный механизм выражения вычислительных задач. Действительно, общая панорама программы Python, с которой мы начали в разделе 1.1, была эквивалентна функции: мы начали рассматривать программу Python как функцию, преобразующую аргументы командной строки в строку вывода. Это представление проявляется на многих разных уровнях. В частности, и это обычно имеет место, длинную программу можно выразить более естественно в терминах функций вместо последовательности операторов присвоения, условных выражений и операторов цикла Python. Обладая способностью определять функции, вы можете лучше организовать свои программы, создавая функции в их пределах, когда это возможно.

Программа 2.1.2. Гауссовы функции (gauss.py)

```
import math
import sys
import stdio

def pdf(x, mu=0.0, sigma=1.0):
    x = float(x - mu) / sigma
    return math.exp(-x*x/2.0) / math.sqrt(2.0*math.pi) / sigma

def cdf(z, mu=0.0, sigma=1.0):
    z = float(z - mu) / sigma
    if z < -8.0: return 0.0
    if z > +8.0: return 1.0
    total = 0.0
    term = z
    i = 3
    while total != total + term:
        total += term
        term *= z * z / i
        i += 2
    return 0.5 + total * pdf(z)

z     = float(sys.argv[1])
mu    = float(sys.argv[2])
sigma = float(sys.argv[3])
stdio.writeln(cdf(z, mu, sigma))
```

`total`	Накопленная сумма
`term`	Текущий предел

Этот код реализует Гауссову (нормальную) плотность вероятностей (`pdf`) и кумулятивное распределение (`cdf`), функции которых не реализованы в библиотеке Python `math`. Реализация функции `pdf()` следует непосредственно из ее определения, а реализация функции `cdf()` использует ряд Тейлора и также вызывает функцию `pdf()` (см. упр. 1.3.36). *Примечание*: если вы собираетесь использовать этот код в другой программе, пожалуйста, обратитесь к программе 2.2.1 (`gaussian.py`), разработанной для многократного использования.

```
% python gauss.py 820 1019 209
0.17050966869132106
% python gauss.py 1500 1019 209
0.9893164837383885
```

Например, программа 2.1.3 (`coupon.py`), приведенная далее, является улучшенной версией программы 1.4.2 (`couponcollector.py`), в ней лучше отделены индивидуальные компоненты вычислений. Если изучить программу 1.4.2, то можно выявить три отдельных задачи.

- Дано количество купонов *n*, вычислить случайное значение купона.
- Дано *n*, провести эксперимент коллекционирования купонов.
- Получить *n* из командной строки, а затем вычислить и вывести результат.

Программа 2.1.3 перестраивает реальный код, в основе которого лежат эти три действия. Первые два реализуются как функции, а третье действие — как глобальный код.

Программа 2.1.3. Коллекционер купонов (повторно) (coupon.py)

```
import random
import sys
import stdarray
import stdio

def getCoupon(n):
    return random.randrange(0, n)

def collect(n):
    isCollected = stdarray.create1D(n, False)
    count = 0
    collectedCount = 0
    while collectedCount < n:
        value = getCoupon(n)
        count += 1
        if not isCollected[value]:
            collectedCount += 1
            isCollected[value] = True
    return count

n = int(sys.argv[1])
result = collect(n)
stdio.writeln(result)
```

`n`	Количество значений купонов (`0 – n-1`)
`isCollected[i]`	Есть ли уже купон `i` в коллекции?
`count`	Количество собранных купонов
`collectedCount`	Количество разных собранных купонов
`value`	Значение текущего купона

Эта версия программы 1.4.2 иллюстрирует стиль инкапсуляции вычислений в функциях. Этот код имеет тот же результат, что и `couponcollector.py`, но лучше разделяет код на три составляющие части: создание случайного целого числа от `0` до `n-1`, запуск эксперимента и осуществление ввода-вывода.

```
% python coupon.py 1000
6522
% python coupon.py 1000
6481
% python coupon.py 1000000
12783771
Hello, World
```

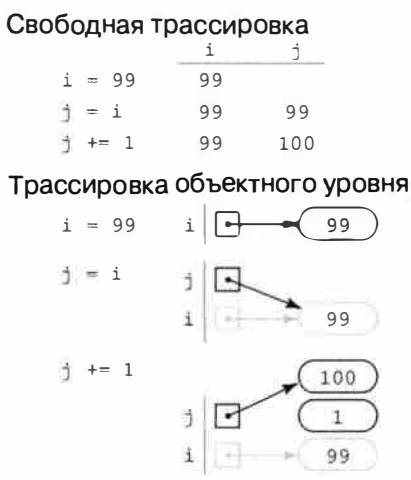
При такой организации мы могли бы изменить функцию `getCoupon()` (например, могло бы понадобиться получить случайные числа другого распределения) или глобальный код (например, могло бы понадобиться получить несколько входных значений или провести несколько экспериментов), не волнуясь о воздействии этих изменений на код в функции `collect()`.

Использование функций изолирует реализацию каждого компонента от других, т.е. *инкапсулирует* их. Как правило, у программ есть множество независимых компонентов, что увеличивает преимущество их разделения на отдельные функции. Подробно мы обсудим эти преимущества после рассмотрения нескольких других примеров, но вы, конечно, и сами понимаете, что легче выразить вычисления, разделив их на функции, так же как легче выразить идею в тексте, разделив его на параграфы. *Всякий раз, когда можете четко разделить задачи в пределах вычисления, так и поступайте.*

Передача аргументов и возвращение значений. Рассмотрим специфические особенности механизмов передачи аргументов и возвращения значений из функций Python. Концептуально эти механизмы очень просты, но имеет смысл уделить время их полному рассмотрению, поскольку от них зависит многое. Понимание механизмов передачи аргументов и возвращения значения является ключом к изучению *любого* нового языка программирования. В случае Python ключевую роль играют концепции *неизменности* (immutability) и *применение псевдонимов* (aliasing).

Вызов по ссылке на объект. В теле функции параметрические переменные можно использовать повсюду, равно как и локальные. Единственное различие между параметрической и локальной переменной в том, что Python инициализирует параметрическую переменную соответствующим аргументом, предоставленным вызывающим кодом. Это фактически *вызов по ссылке на объект* (call by object reference), весьма похожий на *вызов по значению* (call by value), но значением является не сам объект, а только ссылка на него. Одно последствие этого подхода в том, что если параметрическая переменная ссылается на изменяемый объект и вы измените его значение в пределах функции, то это изменит также значение данного объекта в вызывающем коде (поскольку это тот же объект). Далее мы рассмотрим этот подход подробней.

Неизменность и псевдонимы. Как упоминалось в разделе 1.4, массивы — это *изменяемый* (mutable) тип данных, поскольку элементы массива можно изменять. В отличие от него,



Неизменность целых чисел

неизменный (immutable) тип данных не позволяет изменять значение своего объекта. К неизменным относятся такие типы данных, как `int`, `float`, `str` и `bool`. Операции, казалось бы, изменяющие значение неизменяемого типа, фактически приводят к созданию нового объекта, как иллюстрирует простой пример ниже. Сначала оператор `i = 99` создает целое число `99` и присваивает переменной `i` ссылку на него. Затем оператор `j = i` присваивает значение переменной `i` (объектную ссылку) переменной `j`, в результате переменные `i` и `j` ссылаются на тот же объект — целое число `99`. Две переменные, ссылающиеся на тот же объект, являются *псевдонимами* (alias). Далее, в результате выполнения оператора `j += 1`, переменная `j` ссылается на объект со значением `100`, но *это не изменение прежнего целочисленного значения* `99` на `100`! Действительно, поскольку объекты типа `int` неизменны, *никакой* оператор не может изменить значение существующего целого числа. Вместо этого создается новое целое число `1`, суммируется с целым числом `99`, результат, `100`, записывается в другое новое целое число, а ссылка на него присваивается переменной `j`. Однако переменная `i` все еще ссылается на исходный объект `99`. Обратите внимание, что по завершении оператора никакой ссылки на новое целое число `1` больше нет, и о его удалении система позаботится сама, без нашего участия. Неизменность целых, вещественных, булевых чисел и строк является фундаментальным аспектом Python. Подробно преимущества и недостатки этого подхода мы рассмотрим в разделе 3.3.

Целые, вещественные, логические числа и строки как аргументы. Вот ключевой момент, который следует помнить об аргументах функций Python: при передаче они (и параметрические переменные функции) становятся псевдонимами. Практически это преобладающий способ использования псевдонимов в Python, и очень важно понимать его смысл. В иллюстративных целях предположим, что необходима функция инкремента целого числа (наше обсуждение относится также и к более сложным функциям). Начинающий программист Python мог бы определить ее так:

```
def inc(j):
    j += 1
```

а затем ожидать приращения значения целочисленной переменной `i` вызовом `inc(i)`. В некоторых языках программирования такой код сработал бы, но не в Python, как показано на рисунке. Вначале оператор `i = 99` присваивает глобальной переменной `i` ссылку на целое число `99`. Затем оператор `inc(i)` передает ссылку на объект `i` функции `inc()`. Эта объектная ссылка присваивается параметрической переменной `j`. На настоящий момент `i` и `j` — псевдонимы. Как и прежде, оператор `j += 1` в функции `inc()` не изменяет целое число `99`, а создает новое целое число `100` и присваивает ссылку на это него переменной `j`. Но когда функция `inc()` возвращает значение вызывающей стороне, ее параметрическая

переменная j выходит из области видимости, а переменная i все еще продолжает ссылаться на целое число 99.

Этот пример демонстрирует, что в языке Python *функция не может произвести такой побочный эффект, как изменение значения целочисленного объекта* (и ничто другое тоже). Для инкремента переменной i можно использовать такое определение:

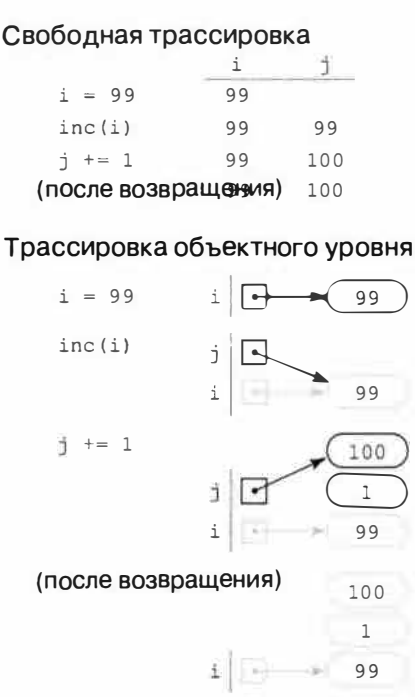
```
def inc(j):
    j += 1
    return j
```

и вызвать функцию с оператором присвоения i = inc(i).

То же справедливо для любого неизменяемого типа. Функция не может изменить значение переменных типа int, float, bool и str.

Массивы как аргументы. Когда функция получает в качестве аргумента массив, она способна обработать произвольное количество объектов. Например, следующая функция вычисляет среднее значение массива типа float или int:

```
def mean(a):
    total = 0.0
    for v in a:
        total += v
    return total / len(a)
```



Применение псевдонимов в вызове функции

Мы использовали массивы как аргументы с самого начала книги. Например, согласно соглашению, Python собирает строки, вводимые после имени программы в команде python, в массив sys.argv[] и неявно вызывает глобальный код с этим массивом строк в качестве аргумента.

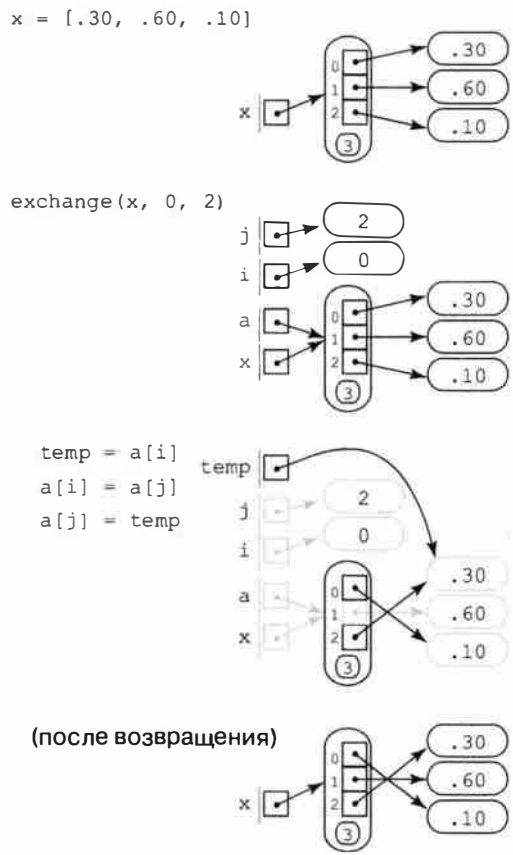
Побочные эффекты с массивами. Поскольку массивы *изменяемы*, целью функции зачастую является получение массива в качестве аргумента и осуществление побочного эффекта (например, изменение порядка элементов массива). В качестве примера рассмотрим прототип функции, обменивающей местами два заданных по индексам элемента в переданном массиве. Адаптируем для этого код, рассматриваемый в начале раздела 1.4:

```
def exchange(a, i, j):
    temp = a[i]
    a[i] = a[j]
    a[j] = temp
```

Основа этой реализации естественно проистекает из представления массива в языке Python. Первая параметрическая переменная функции `exchange()` — это ссылка на сам массив, а не все элементы массива: при передаче функции массива в качестве аргумента обеспечивается возможность работы с самим массивом (а не его копией). Формальная трассировка обращения к этой функции представлена ниже. Эта диаграмма достойна внимательного исследования, она позволит проверить ваше понимание механизма вызова функции Python.

Пример второго прототипа функции получает массив в виде аргумента и создает побочный эффект случайной перестановки его элементов с использованием алгоритма, описанного в разделе 1.4 (и только что определенной функции `exchange()`):

```
def shuffle(a):
    n = len(a)
    for i in range(n):
        r = random.randrange(i, n)
        exchange(a, i, r)
```



Обмен двух элементов массива

Кстати, стандартная функция Python `random.shuffle()` делает то же самое. В качестве другого примера функции в разделе 4.2 мы рассмотрим сортировку массива (перестройку его элементов в заданном порядке).

Массивы как возвращаемые значения. Функция сортировки, перестановки или иной модификации массива, переданного в качестве аргумента, не должна возвращать ссылку на этот массив, поскольку она изменяет содержимое самого массива, а не его копию. Но есть много ситуаций, когда функции имеет смысл возвратить массив как значение. К ним относятся функции, создающие массивы для возвращения клиенту нескольких объектов того же типа.

В качестве примера рассмотрим следующую функцию, возвращающую массив случайных чисел типа `float`:

```
def randomarray(n):
    a = stdarray.create1D(n)
    for i in range(n):
```

```
        a[i] = random.random()
    return a
```

Далее в этой главе мы создадим несколько функций, возвращающих огромные объемы данных таким же образом.

В таблице ниже резюмируется наше обсуждение массивов как аргументов функции на примерах некоторых типичных функций обработки массивов.

Типичный код реализации функций с массивами

Среднее массива

```
def mean(a):
    total = 0.0
    for v in a:
        total += v
    return total / len(a)
```

Скалярное произведение двух векторов одинаковой длины

```
def dot(a, b):
    total = 0
    for i in range(len(a)):
        total += a[i] * b[i]
    return total
```

Обмен двух элементов массива

```
def exchange(a, i, j):
    temp = a[i]
    a[i] = a[j]
    a[j] = temp
```

Вывод одномерного массива (и его длины)

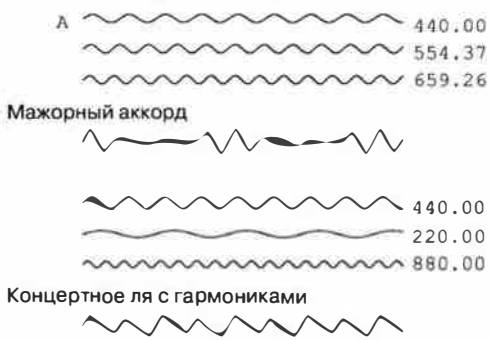
```
def write1D(a):
    stdio.writeln(len(a))
    for v in a:
        stdio.writeln(v)
```

Чтение двумерного массива типа `float` (с размерностями)

```
def readFloat2D():
    m = stdio.readInt()
    n = stdio.readInt()
    a = stdarray.create2D(m, n, 0.0)
    for i in range(m):
        for j in range(n):
            a[i][j] = stdio.readFloat()
    return a
```

Пример: суперпозиция звуковых волн. Давайте улучшим простую звуковую модель, обсуждавшуюся в разделе 1.5, так, чтобы получить звук, напоминающий звук музыкального инструмента. Функции позволяют сделать много разных усовершенствований, и мы можем систематически применить их для создания звуковых волн, намного более сложных, чем просто синусоиды, как в разделе 1.5. В качестве иллюстрации эффективности использования функций для решения интересующей вычислительной задачи рассмотрим программу, имеющую, по существу, те же возможности, что и программа 1.5.8 (`playthattune.py`), но с добавлением гармонических тонов на одну октаву выше и одну октаву ниже каждой ноты, чтобы создать более реалистичный звук.

Аккорды и гармоники. Такие ноты, как концертное *ля*, имеют чистый звук, который не очень музыкален, поскольку у звуков, которые мы привыкли слышать, есть много компонентов. Звук гитарной струны отражается от деревянных частей инструмента, стен комнаты, в которой вы находитесь, и т.д. Такие эффекты можно считать изменением базовой синусоидальной волны. Например, большинство музыкальных инструментов создает *гармоники* (то же ноты в разных октавах, но тише), или *аккорды* (разные ноты одновременно). Для объединения нескольких звуков используется *суперпозиция*: простое наложение волн и изменение масштаба, чтобы все значения остались в интервале от –1 до +1. Кроме того, при наложении синусоидальных волн разных частот можно получить произвольно сложные волны. Действительно, одним из триумфов математики XIX века стала идея о том, что любая гладкая периодическая функция может быть выражена как сумма синусоидальных и косинусоидальных волн, известных как *ряды Фурье*. На практике эта математическая идея означает, что, используя музыкальные инструменты и голосовые связки, можно получить большой диапазон звуков, состоящих из композиции различных колебательных кривых. Любой звук соответствует кривой, а любая кривая соответствует звуку, поэтому суперпозицией (наложением) можно создать произвольно сложные кривые.



Наложение волн позволяет создать сложные звуки

Вычисления со звуковыми волнами. В разделе 1.5 было продемонстрировано, как представить звуковые волны в виде массивов чисел. Теперь мы используем такие массивы, как возвращаемые значения и аргументы функций, обрабатывающих такие данные. Например, следующая функция получает как аргументы частоту (в Герцах) и продолжительность (в секундах), а возвращает представление звуковой волны (точнее, массив, содержащий значения выборок для определенной волны при стандартных 44 100 выборках в секунду):

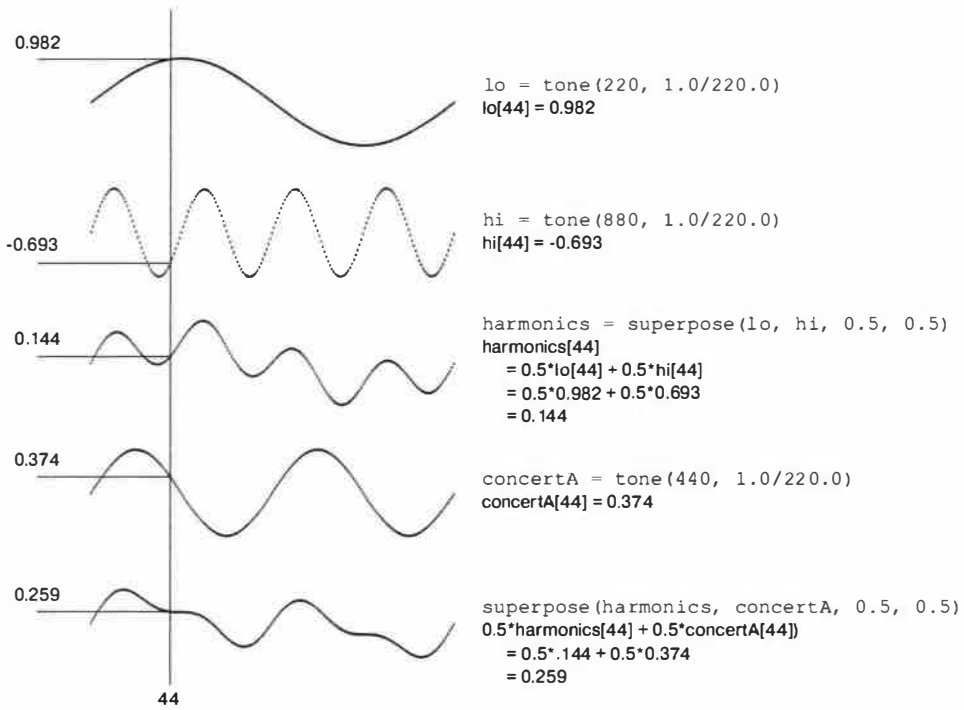
```
def tone(hz, duration, sps=44100):
    n = int(sps * duration)
    a = stdarray.create1D(n+1, 0.0)
    for i in range(n+1):
        a[i] = math.sin(2.0 * math.pi * i * hz / sps)
    return a
```

Размер возвращаемого массива зависит от продолжительности звука: он содержит `sps*duration` чисел типа `float` (для 10 секунд почти полмиллиона). Но теперь мы можем обработать этот массив (возвращаемое значение функции `tone()`) как единую сущность и составить код обработки звуковых волн, как будет вскоре продемонстрировано в программе 2.1.4.

Взвешенная суперпозиция. Поскольку мы представляем звуковые волны массивами чисел, описывающими их значения в тех же точках выборки, реализовать суперпозицию довольно просто: достаточно суммировать их значения в каждой выборке и получить объединенный результат. Для полноты контроля определяем также относительный (весовой) коэффициент для каждой из двух налагаемых волн в следующей функции:

```
def superpose(a, b, aWeight, bWeight):
    c = stdarray.create1D(len(a), 0.0)
    for i in range(len(a)):
        c[i] = aWeight*a[i] + bWeight*b[i]
    return c
```

(Этот код подразумевает, что массивы `a[]` и `b[]` имеют одинаковую длину.) Например, если представленный массивом `a[]` звук должен в три раза превышать налагаемый звук, представленный массивом `b[]`, то мы вызвали бы функцию `superpose(a, b, 0.75, 0.25)`. В верхней части рисунка, приведенного ниже, демонстрируется использование двух вызовов этой функции для добавления гармоник к тону (мы налагаем гармоники, а затем налагаем результат на исходный тон, в результате получается двойной коэффициент исходного тона для каждой гармоники). Пока коэффициенты позитивны и дают в сумме 1, функция `superpose()` выполняет соглашение о нахождении всех значений волн в диапазоне от –1 до +1.

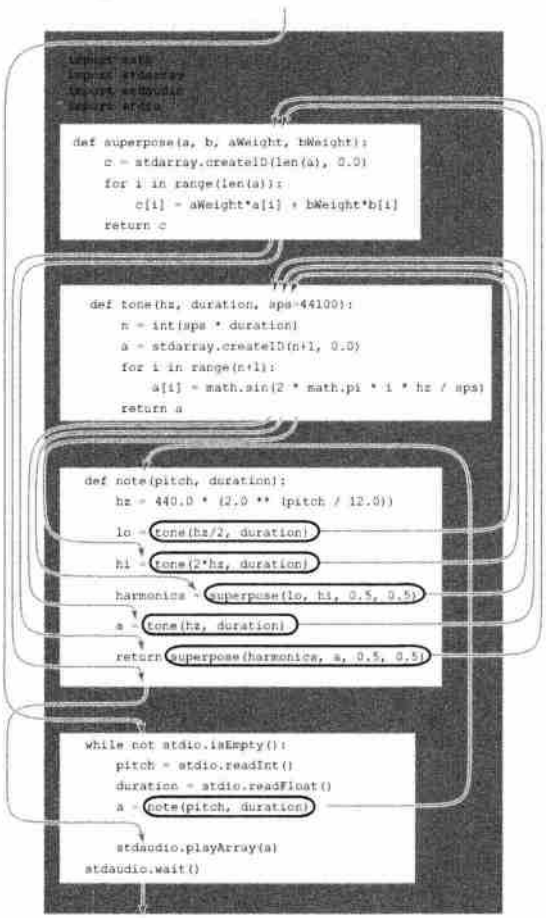


Добавление гармоник к концертному ля (1/220 секунды при 44 100 выборках в секунду)

Программа 2.1.4 (playthattunedeluxe.py) реализует эти концепции для создания более реалистичного звука, чем в программе 1.5.8. Для этого она использует функции, разделяющие вычисление на четыре части.

- По заданной частоте и продолжительности создает чистый тон.
- По двум заданным звуковым волнам и относительным коэффициентам создает их наложение.
- По заданной частоте и продолжительности создает ноту с гармониками.
- Читает со стандартного ввода и проигрывает последовательность пар частота–продолжительность.

Реализации функций, решающих эти задачи, все еще зависят друг от друга. Каждая функция хорошо определяется и проста в реализации. Все они (и модуль stdaudio) представляют звук как набор дискретных значений, сохраненных в массиве и соответствующих выборке звуковой волны, при 44 100 выборках в секунду.



Поток выполнения при нескольких функциях

До этого момента мы использовали функции для удобства написания кода. Например, поток выполнения программ 2.1.1, 2.1.2 и 2.1.3 довольно прост: каждая функция вызывается в коде только один раз. В отличие от них программа 2.1.4 — убедительный пример эффективности определения функций для организации вычислений, поскольку каждая функция вызывается многократно. Например, как представлено на рисунке ниже, функция note() вызывает функцию tone() три раза, а функция superpose() — дважды. Без функций пришлось бы копировать код функций tone() и superpose() несколько раз; функции позволяют иметь дело непосредственно с концепциями. Подобно циклам, функции — это простое, но мощное средство: одна последовательность операторов (в определении

функции) многократно выполняется во время выполнения программы (по разу при каждом вызове функции).

Программа 2.1.4. Проигрывание мелодии (повторно) (playthattunedeluxe.py)

```
import math
import stdarray
import stdaudio
import stdio

def superpose(a, b, aWeight, bWeight):
    c = stdarray.create1D(len(a), 0.0)
    for i in range(len(a)):
        c[i] = aWeight*a[i] + bWeight*b[i]
    return c

def tone(hz, duration, sps=44100):
    n = int(sps * duration)
    a = stdarray.create1D(n+1, 0.0)
    for i in range(n+1):
        a[i] = math.sin(2.0 * math.pi * i * hz / sps)
    return a

def note(pitch, duration):
    hz = 440.0 * (2.0 ** (pitch / 12.0))
    lo = tone(hz/2, duration)
    hi = tone(2*hz, duration)
    harmonics = superpose(lo, hi, 0.5, 0.5)
    a = tone(hz, duration)
    return superpose(harmonics, a, 0.5, 0.5)

while not stdio.isEmpty():
    pitch = stdio.readInt()
    duration = stdio.readFloat()
    a = note(pitch, duration)
    stdaudio.playSamples(a)
stdaudio.wait()
```

hz	Частота
lo[]	Нижняя гармоника
hi[]	Верхняя гармоника
h[]	Объединенная гармоника
a[]	Чистый тон

Эта программа читает звуковые выборки, улучшает звук, добавляя гармоники, чтобы получить более реалистичный тон, чем в программе 1.5.8, и проигрывает полученный звук на стандартном аудиоустройстве.

```
% python playthattunedeluxe.py < elise.txt
```

```
% more elise.txt
7 .125  6 .125
7 .125  6 .125 7  .125
2 .125  5 .125 3  .125
0 .25
```

Функции важны еще потому, что они позволяют *дополнить* язык Python в пределах программы. Реализовав и отладив такие функции, как `harmonic()`, `pdf()`, `cdf()`, `mean()`, `exchange()`, `shuffle()`, `isPrime()`, `superpose()`, `tone()` и `note()`, мы можем использовать их почти так же, как и встроенные функции Python. Такая гибкость открывает целый новый мир программирования. Прежде мы могли рассматривать программу Python как последовательность операторов, а теперь ее можно считать набором *функций*, способных вызывать друг друга. Привычный поток передачи управления от оператора к оператору все еще имеет место в пределах функций, но теперь у программ есть более высокоуровневые средства контроля потока (вызов функций и возвращение значений). Эта способность позволяет мыслить в терминах операций, необходимых приложению, а не только встроенных в Python.

Всякий раз, когда можете четко разделить задачи в пределах вычисления, так и поступайте. Примеры в этом разделе (и в остальной части книги) ясно иллюстрируют преимущества соблюдения этого принципа. Функции позволяют:

- разделить длинную последовательность операторов на независимые части;
- многократно использовать код без необходимости копировать его;
- работать с высокоуровневыми концепциями (такими, как звуковые волны).

Это позволяет создавать более понятный код, который проще поддерживать и отлаживать, по сравнению с длинной программой, состоящей исключительно из операторов присвоения, условных выражений и операторов цикла Python. В следующем разделе мы обсудим идею использования функций, определенных в других файлах, что снова переводит нас на следующий уровень программирования.

Вопросы и ответы

Могу ли я использовать в функции оператор `return`, не определяя значение?

Да. Технически он возвращает объект `None` — единственное значение типа `NoneType`.

Что будет, если один поток выполнения функции приводит к оператору `return`, возвращающему значение, а другой просто достигает конца тела функции?

Определение такой функции является плохим стилем программирования, поскольку это налагает серьезное бремя на вызывающую сторону: пользователь должен знать, при каких обстоятельствах функция возвращает значение, а при каких нет.

Что будет, если в теле функции я расположу код после оператора `return`?

Как только встретится оператор `return`, управление вернется вызывающей стороне, поэтому любой код в теле функции после оператора `return` бесполезен; он никогда не выполняется. В языке Python — это плохой стиль, но ничего некорректного в такой функции нет.

Что будет, если в том же файле `.py` определить две функции с тем же именем, но, возможно, с разным количеством аргументов?

Произойдет *перегрузка функций* (function overloading), поддерживаемая многими языками программирования, но не Python, поэтому второе определение функции заменит первое. Зачастую тот же эффект можно получить с использованием стандартных аргументов.

Что будет, если определить две функции с теми же именами, но в разных файлах?

Все будет прекрасно. Например, вполне нормально иметь функцию `pdf()` в файле `gauss.py` для вычисления функции Гауссовой плотности вероятностей и другую функцию `pdf()` в файле `cauchy.py` для вычисления функции плотности вероятностей Коши. В разделе 2.2 описано, как вызывать функции, определенные в разных файлах `.py`.

Может ли функция изменить объект, с которым связана параметрическая переменная?

Да, параметрическую переменную вполне можно использовать с левой стороны оператора присвоения. Однако многие программисты Python считают это

плохим стилем. Обратите внимание, что такой оператор присвоения никак не воздействует на вызывающую сторону.

Проблема с побочными эффектами и изменяемыми объектами сложна. Действительно ли это так важно?

Да. Правильный контроль побочных эффектов является одной из самых важных задач в больших системах. Поэтому безусловно имеет смысл уделить время полному пониманию различий между передачей массивов (изменяемых) и целых, вещественных, логических чисел и строк (неизменяемых). Для всех других типов данных используются те же механизмы, как будет описано в главе 3.

Как передать массив функции так, чтобы она не могла изменять его элементы?

Напрямую — никак. В разделе 3.3 будет показано, как получить тот же эффект за счет создания типа данных *оболочки* (wrapper) и передачи объекта этого типа вместо массива. Будет также продемонстрировано использование встроенного типа данных Python `tuple`, представляющего неизменяемую последовательность объектов.

Можно ли использовать изменяемый объект в качестве стандартного значения для необязательного аргумента?

Да, но это может привести к неожиданным последствиям. Python вычисляет стандартное значение только однажды, когда функция определяется (а не при каждом вызове функции). Таким образом, если тело функции изменит стандартное значение, то последующие вызовы функции будут использовать измененное значение. Подобные трудности возникают, если инициализировать стандартное значение при вызове функции с побочным эффектом. Например, после выполнения следующего фрагмента кода:

```
def append(a=[], x=random.random()):
    a += [x]
    return a

b = append()
c = append()
```

- `b[]` и `c[]` окажутся псевдонимами того же массива длиной 2 (а не 1), содержащего одно значение типа `float`, повторенное дважды, а не два разных числа типа `float`.

Упражнения

2.1.1. Составьте функцию `max3()`, получающую три аргумента типа `int` или `float` и возвращающую наибольший из них.

2.1.2. Составьте функцию `odd()`, получающую три аргумента типа `bool` и возвращающую значение `True`, если получено нечетное количество аргументов `True` и значение `False` в противном случае.

2.1.3. Составьте функцию `majority()`, получающую три аргумента типа `bool` и возвращающую значение `True`, если по крайней мере два из аргументов `True`, и значение `False` в противном случае. Не используйте оператор `if`.

2.1.4. Составьте функцию `areTriangular()`, получающую три числовых аргумента и возвращающую значение `True`, если они могли бы быть длинами сторон треугольника (ни один из них не больше и не равен сумме двух других), и значение `False` в противном случае.

2.1.5. Составьте функцию `sigmoid()`, получающую аргумент `x` типа `float` и возвращающую значение типа `float`, полученное по формуле $1/(1 + e^{-x})$.

2.1.6. Составьте функцию `lg()`, получающую как аргумент целое число `n` и возвращающую логарифм `n` по основанию 2. Можете использовать модуль Python `math`.

2.1.7. Составьте функцию `lg()`, которая получает целое число `n` как аргумент и возвращает наибольшее целое число, не большее логарифма `n` по основанию 2. *Не используйте* модуль Python `math`.

2.1.8. Составьте функцию `signum()`, получающую аргумент `n` типа `float` и возвращающую `-1`, если `n` меньше `0`, `0`, если `n` равно `0`, и `+1`, если `n` больше `0`.

2.1.9. Рассмотрим следующую функцию `duplicate()`:

```
def duplicate(s):
    t = s + s
```

Что выводит следующий фрагмент кода?

```
s = 'Hello'
s = duplicate(s)
t = 'Bye'
t = duplicate(duplicate(duplicate(t)))
stdio.writeln(s + t)
```

2.1.10. Рассмотрим следующую функцию `cube()`:

```
def cube(i):
    i = i * i * i
```

Сколько раз выполняется следующий цикл `while`?

```
i = 0
while i < 1000:
    cube(i)
    i += 1
```

Решение: Только 1 000 раз. Вызов функции `cube()` никак не влияет на клиентский код. Он изменяет значение параметрической переменной `i`, но никак не затрагивает переменную `i` в цикле `while`, являющуюся совсем другой переменной. Если заменить вызов функции `cube(i)` оператором `i = i * i * i` (возможно, так вы и подумали), то цикл выполняется пять раз, пока переменная `i` имеет значения 0, 1, 2, 9 и 730.

2.1.11. Что выводит следующий фрагмент кода?

```
for i in range(5):
    stdio.write(i)
for j in range(5):
    stdio.write(i)
```

Решение: `0123444444`. Обратите внимание: второй вызов функции `stdio.write()` использует аргумент `i`, а не `j`. В отличие от аналогичных циклов во многих других языках программирования, когда первый цикл `for` завершается, содержащая значение `4` переменная `i` не удаляется и остается в области видимости.

2.1.12. Следующая формула *контрольной суммы* (checksum) широко используется банками и кредитными компаниями для проверки корректности номеров счетов:

$$d_0 + f(d_1) + d_2 + f(d_3) + d_4 + f(d_5) + \ldots = 0 \ (\mathrm{mod}\, 10),$$

d_i — это десятичные цифры номера счета; $f(d)$ — сумма десятичных цифр $2d$ (например, $f(7) = 5$ поскольку $2 \times 7 = 14$, а $1 + 4 = 5$). Например, номер счета 17327 вполне допустим, поскольку $1 + 5 + 3 + 4 + 7 = 20$, что кратно 10. Реализуйте функцию f и составьте программу, получающую в аргументе командной строки целое число из 10 цифр и выводящую допустимый номер из 11 цифр с заданным целым числом в виде первых 10 цифр и контрольной суммы в виде последней цифры.

2.1.13. Даны две звезды с углами склонения и прямыми восхождениями (d_1, a_1) и (d_2, a_2) соответственно. Их противолежащий угол вычисляется по формуле

$$2\arcsin((\sin^2(d/2) + \cos(d_1)\cos(d_2)\sin^2(a/2))^{1/2}),$$

где a_1 и a_2 — это углы между –180 и 180 градусами; d_1 и d_2 — углы между –90 и 90 градусами; $a = a_2 - a_1$ и $d = d_2 - d_1$. Составьте программу, которая

получает в аргументах командной строки склонение и прямое восхождение двух звезд и выводит их противолежащий угол. *Подсказка*: будьте внимательны с преобразованием градусов в радианы.

2.1.14. Составьте функцию `readBool2D()`, читающую двумерную матрицу значений 0 и 1 (с размерностями) в массив булевых переменных. *Решение*: тело функции практически то же, что и у соответствующей функции в таблице, приведенной выше, для двумерных массивов типа `float`:

```
def readBool2D():
    m = stdio.readInt()
    n = stdio.readInt()
    a = stdarray.create2D(m, n, False)
    for i in range(m):
        for j in range(n):
            a[i][j] = stdio.readBool()
    return a
```

2.1.15. Составьте функцию, получающую как аргумент массив `a[]` сугубо положительных вещественных чисел, и измените масштаб массив так, чтобы все элементы оказались в диапазоне от 0 до 1 (вычитая минимальное значение из каждого элемента, а затем деля каждый элемент на разницу между минимальным и максимальным значениями). Используйте встроенные функции `max()` и `min()`.

2.1.16. Составьте функцию `histogram()`, получающую как аргументы массив `a[]` целых чисел, целое число `m` и возвращающую массив длиной `m`, каждый `i`-й элемент которого — количество чисел `i` в исходном массиве. Подразумевается, что все значения в массиве `a[]` находятся в диапазоне от `0` до `m-1`, чтобы сумма значений в возвращаемом массиве была равна `len(a)`.

2.1.17. Соберите фрагменты кода из этого раздела и раздела 1.4, чтобы составить программу, которая получает из командной строки целое число `n` и выводит `n` сдач по пять карт из случайно перетасованной колоды, отделенных пустыми строками, по одной карте в строку с использованием названий карт, например `Ace of Clubs`.

2.1.18. Составьте функцию `multiply()`, которая получает как аргументы две квадратных матрицы одинаковой размерности и возвращает их произведение (еще одна квадратная матрица той же размерности). *Дополнительное задание*: обеспечьте в программе обязательное равенство количества столбцов в первой матрице количеству рядов во второй матрице.

2.1.19. Составьте функцию `any()`, получающую как аргумент массив логических переменных и возвращающую `True`, если *любой* из элементов в массиве содержит значение `True`, и `False` в противном случае. Составьте функцию `all()`, которая получает как аргумент массив логических переменных и возвращающую `True`, если *все* элементы в массиве содержат значение `True`, и `False` в противном случае. Обратите внимание, что функции `all()` и `any()` являются встроенными функциями Python (задача этого упражнения в том, чтобы вы лучше изучили их, создавая собственные версии).

2.1.20. Разработайте версию функции `getCoupon()`, лучше моделирующую ситуацию, когда один из n купонов редок: выберите одно значение наугад и возвращайте его с вероятностью $1/(1000n)$, а все остальные — с равной вероятностью. *Дополнительное задание*: как это изменение влияет на среднее значение функции коллекции купонов?

2.1.21. Измените программу `playthattune.py` так, чтобы добавить гармоники в две октавы от каждой ноты с половинным коэффициентом относительно гармоник в одну октаву.

Практические упражнения

2.1.22. *Задача дня рождения.* Составьте программу с соответствующими функциями для решения задачи дня рождения (см. упр. 1.4.36).

2.1.23. *Функция Эйлера.* Это важная функция в теории чисел: $\phi(n)$ определяется как количество положительных целых чисел, меньших или равных n, взаимно простых с n (никаких общих множителей, кроме n и 1). Составьте функцию, получающую целочисленный аргумент n и возвращающую $\phi(n)$. Глобальный код должен получать из командной строки целое число, вызывать функцию и выводить результат.

2.1.24. *Гармонические числа.* Создайте программу `harmonic.py`, определяющую для вычисления гармонических чисел три функции: `harmonic()`, `harmonicSmall()` и `harmonicLarge()`. Функция `harmonicSmall()` должна вычислить лишь сумму (как в программе 2.1.1), функция `harmonicLarge()` должна использовать аппроксимацию $H_n = \log_e(n) + \gamma + 1/(2n) - 1/(12n^2) - 1/(120n^4)$ (число $\gamma = 0.577215664901532\ldots$ — это *константа Эйлера*), а функция `harmonic()` должна вызвать функцию `harmonicSmall()` для $n<100$ и функцию `harmonicLarge()` в противном случае.

2.1.25. *Гауссовы случайные значения.* Поэкспериментируйте со следующей функцией, создающей случайные значения из Гауссова распределения. На

основании этих значений создайте случайную точку в единичном круге с использованием формулы Бокса–Мюллера (см. упр. 1.2.24).

```
def gaussian():
    r = 0.0
    while (r >= 1.0) or (r == 0.0):
        x = -1.0 + 2.0 * random.random()
        y = -1.0 + 2.0 * random.random()
        r = x*x + y*y
    return x * math.sqrt(-2.0 * math.log(r) / r)
```

Получите аргумент командной строки `n` и создайте `n` случайных чисел, используя массив `a[]` из 20 целых чисел для подсчета количества создаваемых чисел, попадающих в интервал между `i*.05` и `(i+1)*.05` для `i` от `0` до `19`. Затем используйте модуль `stddraw`, чтобы нарисовать получаемые значения и сравнить результат с кривой нормального распределения. *Комментарий*: этот подход быстрее и точенее описанного в упражнении 1.2.24. Хоть здесь и задействован цикл, он выполняется (в среднем) только $4/\pi$ раз (примерно 1,273). Это сокращает общее ожидаемое количество вызовов трансцендентальной функции.

2.1.26. *Бинарный поиск.* Общая функция `cdf()`, подробно рассматриваемая в разделе 4.2, эффективна для вычисления инверсии кумулятивного распределения. Такие функции непрерывны и неубывающи от (0, 0) до (1, 1). Чтобы найти значение x_0, для которого $f(x_0) = y_0$, проверьте значение $f(0.5)$. Если оно больше y_0, то x_0 должно быть между 0 и 0,5; в противном случае оно должно быть между 0.5 и 1. Так или иначе, мы делим пополам интервал, о котором известно, что он содержит x_0. Итеративно можно вычислить x_0 в пределах заданного допуска. Добавьте в программу `gauss.py` функцию `cdfInverse()`, использующую бинарный поиск для вычисления инверсии. Измените глобальный код так, чтобы получать как третий аргумент командной строки число p от 0 до 100 и выводить минимальный балл, необходимый студенту, чтобы оказаться во главе p процентов студентов, сдавших тест SAT в год, когда среднее и среднеквадратичное отклонение были первыми двумя аргументами командной строки.

2.1.27. *Модель ценообразования опционов Блэка–Шоулза.* Формула Блэка–Шоулза определяет теоретическую цену на европейские опционы, подразумевающую, что если базовый актив продается на рынке, то цена опциона на него неявным образом уже устанавливается самим рынком. Таким образом, если есть текущий курс акций s, цена исполнения опциона x,

непрерывно начисляемая устойчивая процентная ставка r, среднеквадратичное отклонение σ доходности (волатильности) и срок погашения t (в годах), значение стоимости вычисляется по формуле $s\Phi(a) - x\ e^{-rt}\ \Phi(b)$, где $\Phi(z)$ — это Гауссова функция кумулятивного распределения. Составьте программу, которая получает из командной строки `s`, `x`, `r`, `sigma` и `t`, а затем выводит стоимость по Блэку–Шоулзу.

2.1.28. *Подразумеваемая волатильность.* Обычно доходность — это неизвестное значение в формуле Блэка–Шоулза. Составьте программу, которая читает из командной строки значения `s`, `x`, `r`, `t` и текущей цены европейских опционов, а затем использует бинарный поиск (см. упр. 2.1.26) для вычисления σ.

2.1.29. *Метод Горнера.* Составьте программу `horner.py` с функцией `evaluate(x, a)`, вычисляющей многочлен $a(x)$, коэффициенты которого являются элементами массива `a[]`:

$$a_0 + a_1x^1 + a_2x^2 + \ldots + a_{n-2}x^{n-2} + a_{n-1}x^{n-1}$$

Использование метода Горнера — это эффективный способ вычисления многочлена, записанного в виде суммы мономов (одночленов), при заданном значении переменной:

$$a_0 + x\,(a_1 + x\,(a_2 + \ldots + x\,(a_{n-2} + x\,a_{n-1})\ldots))$$

Затем составьте функцию `exp()`, вызывающую функцию `evaluate()` для вычисления аппроксимации к e^x, используя первые n терминов выражения ряда Тейлора $e^x = 1 + x + x^2/2! + x^3/3! + \ldots$. Получите из командной строки аргумент `x` и сравните свой результат с результатом функции `math.exp(x)`.

2.1.30. *Закон Бенфорда.* Американский астроном Саймон Ньюком обнаружил странность в сборнике логарифмических таблиц: начала страниц были разреженней, чем их завершения. Он заподозрил, что в вычислениях чаще встречаются числа, начинающиеся с 1, чем с 8 или 9, и постулировал закон первой цифры, гласящий, что при общих обстоятельствах первой цифрой, вероятней всего, будет 1 (примерно 30 %), а не 9 (меньше 4 %). Этот закон Бенфорда часто используется для статистической проверки. Например, Федеральное налоговое управление США (IRS) полагается на него при поиске налоговых мошенничеств. Составьте программу, которая читает со стандартного устройства ввода последовательности целых чисел и сводит в таблицу количество значений, начинающихся с цифр 1–9, разделяя вычисление на ряд соответствующих функций.

Используйте свою программу для проверки закона на информационных таблицах с вашего компьютера или из веб. Затем составьте программу, сбивающую со следа IRS за счет создания случайных сумм от 1.00 до 1 000.00$, с распределением в соответствии с законом Бенфорда.

2.1.31. *Биномиальное распределение.* Составьте функцию `binomial()`, получающую целые числа `n` и `k`, а также вещественное число `p` и вычисляющую вероятность получения точно `k` орлов при `n` бросках монеты (с вероятностью `p` орлов), используя формулу

$$f(k, n, p) = p^k(1-p)^{n-k}\, n!/(k!(n-k)!)$$

Подсказка. Во избежание вычислений с огромными целыми числами вычислите $x = \ln f(k, n, p)$, а затем возвратите e_x. В глобальном коде получите из командной строки числа `n` и `p` и удостоверьтесь, что сумма всех значений `k` от `0` до `n` составляет приблизительно 1. Кроме того, сравните каждое вычисленное значение с нормальной аппроксимацией

$$f(k, n, p) \approx \Phi(k+1/2, np, \sqrt{np(1-p)}) - \Phi(k-1/2, \sqrt{np(1-p)})$$

2.1.32. *Коллекция купонов при биномиальном распределении.* Составьте версию функции `getCoupon()`, использующую функцию `binomial()` из предыдущего упражнения, чтобы получать значения купонов согласно биномиальному распределению при $p = 1/2$. *Подсказка*: создайте однородно случайное число x в диапазоне от 0 до 1, а затем возвратите наименьшее значение k, для которого сумма $f(j, n, p)$ для всех $j<k$ превышает x. *Дополнительное задание*: выработайте гипотезу для описания поведения функции коллекционирования купонов согласно этому предположению.

2.1.33. *Аккорды.* Составьте версию программы `playthattunedeluxe.py`, способную проигрывать музыку с аккордами (три или более разных нот, включая гармоники). Разработайте входной формат, позволяющий определять различные продолжительности для всех аккордов и различные весовые коэффициенты для амплитуд каждой ноты в пределах аккорда. Создайте тестовые файлы, позволяющие проверить программу с разными аккордами и гармониками, а затем создайте использующий их файл мелодии *К Элизе* Людвига ван Бетховена.

2.1.34. *Почтовые штрих-коды.* Штрих-коды почтовой службы США, используемые для доставки почты, определяются следующим образом: каждая десятичная цифра почтового индекса кодируется последовательностью из трех штрихов половинной высоты и двух полной. Штрих-код начинается

и завершается полноразмерными штрихами (ограждение) и включает цифру контрольной суммы (после пяти цифр индекса или ZIP+4), вычисляемой как результат деления по модулю 10 суммы первых цифр индекса. Определите следующие функции.

- Использующую модуль `stddraw` для рисования штрихов половинной и полной высоты.
- Рисующую последовательность штрихов по заданной цифре.
- Вычисляющую цифру контрольной суммы.

Определите также глобальный код, получающий из командной строки пять (или девять) цифр почтового индекса и рисующий соответствующий почтовый штрих-код.

2.1.35. *Календарь.* Составьте программу `cal.py`, получающую в аргументах командной строки два числа, `m` и `y`, и выводящую месячный календарь для месяца `m` в году `y`, как в следующем примере:

```
% python cal.py 2 2015
February 2015
 S  M Tu  W Th  F  S
 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
```

Подсказка: см. программу 1.2.5 (`leapyear.py`) и упражнение 1.2.26.

2.1.36. *Выбросы Фурье.* Составьте программу, получающую аргумент командной строки `n` и выводящую график функции

$$(\cos(t) + \cos(2t) + \cos(3t) + ... + \cos(Nt))/N$$

для 500 равноудаленных выборок t от –10 до 10 (в радианах). Запустите программу для $n = 5$ и $n = 500$. *Примечание.* Можно заметить схождение суммы к выбросу (везде 0, кроме одного значения). Это свойство — основание для доказательства того, что *любая* плавная функция может быть выражена как сумма синусоид.

2.2. Модули и клиенты

Программа 2.2.1. Модуль Гауссовых функций (gaussian.py) 249
Программа 2.2.2. Пример клиента модуля Гауссовых функций (gaussiantable.py) 250
Программа 2.2.3. Модуль случайных чисел (stdrandom.py) 259
Программа 2.2.4. Система итерационных функций (ifs.py) 266
Программа 2.2.5. Модуль анализа данных (stdstats.py) 269
Программа 2.2.6. Рисование значений данных (stdstats.py, продолжение) 272
Программа 2.2.7. Исследование Бернулли (bernoulli.py) 274

Каждая написанная до сих пор программа состояла из кода Python, расположенного в едином файле `.py`. Хранение всего кода в одном файле является ненужным ограничением для больших программ. К счастью, Python позволяет вызывать функции, определенные в другом файле. У этой возможности есть два важных последствия.

Во-первых, это позволяет *многократно использовать код*. Одна программа может использовать код, написанный и отлаженный прежде, простым его вызовом, без копирования и вставки. Возможность определения многократно используемого кода является основой современного программирования. Определяя и используя собственный набор операций с данными, можно существенно расширить базовые возможности языка Python.

Во-вторых, это обеспечивает *модульное программирование*. Программу можно не только разделить на функции, как описано в разделе 2.1, но и сохранить их в разных файлах, группируя согласно потребностям приложения. Модульное программирование важно, поскольку оно позволяет независимо составлять и отлаживать части больших программ по одной часть за раз, оставляя каждую законченную часть в ее собственном файле для последующего использования без необходимости заботиться о ее подробностях снова. Можно составлять модули функций для использования любыми другими программами, сохраняя каждый модуль в его собственном файле и используя его функции в других программах. Примерами являются модуль Python `math` и модули ввода-вывода `std*` с сайта книги. Однако важней всего то, что определять собственные модули очень просто. Возможность определять модули, а затем использовать их в нескольких программах — это критически важный аспект, позволяющий создавать программы для решения сложных задач.

Подобно тому, как в разделе 2.1 мы стали считать программы Python не просто последовательностью операторов, а набором *функций* (и глобального кода), пришло время рассматривать программы Python как наборы *файлов*, каждый из которых является независимым модулем, состоящим из ряда функций. Поскольку каждая функция может вызвать функцию в другом модуле, весь код можно

считать взаимодействующей сетью функций, способных вызывать друг друга и группироваться в модулях. Это позволят также упростить разработку сложных программ за счет разделения задач на модули, которые можно реализовать и проверить независимо.

Использование функций в других программах. Чтобы в программе Python обратиться к функции, определенной в другом месте, используется тот же механизм, что и для вызова функций из наших модулей `std*` или модулей Python `math` и `random`. Рассмотрим сначала базовый языковой механизм Python. Для этого будем различать два типа программ Python.

- *Модуль* (module), содержащий функции, доступные для использования другими программами.
- *Клиент* (client) — программа, использующая функцию в модуле.

Программа может быть и модулем, и клиентом: терминология относится только к конкретной функции.

Чтобы создать и использовать модуль, необходимо пять простых этапов: в коде клиента импортировать модуль, квалифицировать вызовы функций, составить проверки для модуля, в коде модуля устранить глобальный код и сделать модуль доступным для клиента. Впоследствии мы обсудим каждый из этих этапов подробно, а пока используем имя *модуль*`.py` для обозначения модуля и *клиент*`.py` для обозначения клиента. Далее мы рассмотрим полный процесс на примере модуля в программе 2.2.1 (`gaussian.py`), собранного из блоков версий программы 2.1.2 (`gauss.py`) для вычисления функций Гауссовых распределений и клиентской программы 2.2.2 (`gaussiantable.py`), использующей модуль для вычисления и вывода таблицы значений.

На стороне клиента: импортируйте модуль. Чтобы использовать модуль, поместите в код файла *клиент*`.py` оператор `import` *модуль* (обратите внимание на отсутствие расширения `.py`). Задача этого оператора в том, чтобы уведомить Python о вызове кодом клиента одной или нескольких функций, определенных в модуле *модуль*`.py`. В нашем примере клиент `gaussiantable.py` содержит оператор `import gaussian`; теперь он может вызвать любую функцию, определенную в модуле `gaussian.py`. Обычно в коде Python (включая все программы книги) операторы `import` располагают в начале программы, причем все стандартные модули Python подключались до всех пользовательских модулей.

На стороне клиента: квалифицируйте вызовы функции. Чтобы вызвать функцию, определенную в модуле *модуль*`.py`, из любой другой (клиентской) программы Python, перед именем функции следует указать имя модуля, сопровождаемое точкой (`.`). Вы уже привыкли к этому на примере вызовов таких функций, как `stdio.writeln()` или `math.sqrt()`. В данном примере клиент `gaussiantable.py` использует вызов функции `gaussian.cdf(score, mu, sigma)`, подразумевая функцию `cdf()`, определенную в модуле `gaussian.py`.

Программа 2.2.1. Модуль Гауссовых функций (gaussian.py)

```
import math
import sys
import stdio

def pdf(x, mu=0.0, sigma=1.0):
    x = float(x - mu) / sigma
    return math.exp(-x*x/2.0) / math.sqrt(2.0*math.pi) / sigma

def cdf(z, mu=0.0, sigma=1.0):
    z = float(z - mu) / sigma
    if z < -8.0: return 0.0
    if z > +8.0: return 1.0
    total = 0.0
    term = z
    i = 3
    while total != total + term:
        total += term
        term *= z * z / i
        i += 2
    return 0.5 + total * pdf(z)

def main():
    z     = float(sys.argv[1])
    mu    = float(sys.argv[2])
    sigma = float(sys.argv[3])
    stdio.writeln(cdf(z, mu, sigma))

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

Эта программа переупаковывает функции `pdf()` и `cdf()` из программы 2.1.2 (`gauss.py`) в модуль, пригодный для использования клиентским кодом, располагающимся в другом файле, таком, как `gaussiantable.py` (программа 2.2.2). Здесь также определена клиентская проверка `main()`, получающая как аргументы командной строки числа `z`, `mu` и `sigma` типа `float` и использующая их для проверки функций `pdf()` и `cdf()`.

```
% python gaussian.py 820 1019 209
0.17050966869132106
% python gaussian.py 1500 1019 209
0.9893164837383885
% python gaussian.py 1500 1025 231
0.9801220907365491
```

Программа 2.2.2. Пример клиента модуля Гауссовых функций (gaussiantable.py)

```
import sys
import stdio
import gaussian

mu    = float(sys.argv[1])
sigma = float(sys.argv[2])

for score in range(400, 1600+1, 100):
    percent = gaussian.cdf(score, mu, sigma)
    stdio.writef('%4d %.4f\n', score, percent)
```

Этот клиент модулей `gaussian` (а также `sys` и `stdio`) выводит таблицу процентов учеников, сдавших тест SAT ниже заданного балла. Подразумевается, что экзаменационные отметки подчиняются Гауссову распределению при заданном среднем и среднеквадратичном отклонении. Это иллюстрирует вызов функции из других модулей: импорт модуля с последующим вызовом любых функций в его пределах с указанием имени модуля и точки перед именем функции. В частности, этот код вызывает функцию `cdf()` из программы 2.2.1 (`gaussian.py`).

```
% python gaussiantable.py 1019 209
 400  0.0015
 500  0.0065
 600  0.0225
 700  0.0635
 800  0.1474
 900  0.2845
1000  0.4638
1100  0.6508
1200  0.8068
1300  0.9106
1400  0.9658
1500  0.9893
1600  0.9973
```

На стороне модуля: составьте клиентскую проверку. Хорошая практика программирования, выработанная на протяжении многих десятилетий, подразумевает создание кода проверки функции в модуле и *включение этого кода в сам модуль*. По давней традиции этот код помещают в функцию `main()`. В нашем примере модуль `gaussian.py` также содержит функцию `main()`, получающую из

командной строки три аргумента, вызывающую функции модуля и выводящую результат на стандартный вывод.

На стороне модуля: устраните ненужный глобальный код. Оператор Python `import` выполняет весь глобальный код, присутствующий в импортированном модуле (включая определения функции и глобальный код). Мы не можем оставить произвольный глобальный код в модуле (такой, как проверочный код, выводящий текущие значения переменных), поскольку Python будет выполнять его каждый раз при импорте модуля. Вместо этого мы помещаем проверочный код в функцию `main()`, как было описано выше. Можно также принять меры, чтобы Python вызвал функцию `main()` при запуске программы из командной строки (и только в этом случае). Для этого используется следующий трюк:

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

Неофициально этот код указывает Python, что если этот файл `.py` запущен непосредственно командой `python` (а не через оператор `import`), то следует вызвать функцию `main()`. В результате определенная в файле *модуль*`.py` функция `main()` выполняется при отладке модуля командой `python` *модуль*`.py`, но она *не выполняется* в процессе импорта модуля клиентом с использованием оператора `import`.

Сделайте модуль доступным для клиента. Выполняя оператор `import` *модуль* в файле *клиент*`.py` Python должен быть в состоянии найти файл *модуль*`.py`. Если модуль не встроенный и не стандартный, Python в первую очередь ищет его в том же каталоге, что и файл *клиент*`.py`. Поэтому проще всего располагать файлы клиента и модуль в том же каталоге. В разделе "Вопросы и ответы" в конце этого раздела описан альтернативный подход.

Таким образом, оператор `import gaussian` делает функции в модуле `gaussian.py` доступными для использования любой другой программой. Напротив, клиентская программа `gaussiantable.py` содержит произвольный глобальный код и не предназначена для использования другими программами — ее код может быть введен в интерактивном режиме. Такой код мы называем *сценарием* (script). Между модулем и сценарием нет большого различия: программисты Python зачастую начинают с создания сценария, а в конечном счете получают модуль, удалив из него произвольный глобальный код.

Предупреждение о сокращении. С этого момента и далее мы резервируем термин *модуль* для файлов с расширением `.py`, структурированных так, чтобы его средства могли быть многократно использованы в других программах Python (в результате он не содержит произвольного глобального кода), а термин *сценарий* описывает файл с расширением `.py`, не предназначенный для многократного использования (поскольку он содержит произвольный глобальный код), хотя обычно мы называем их *программами*.

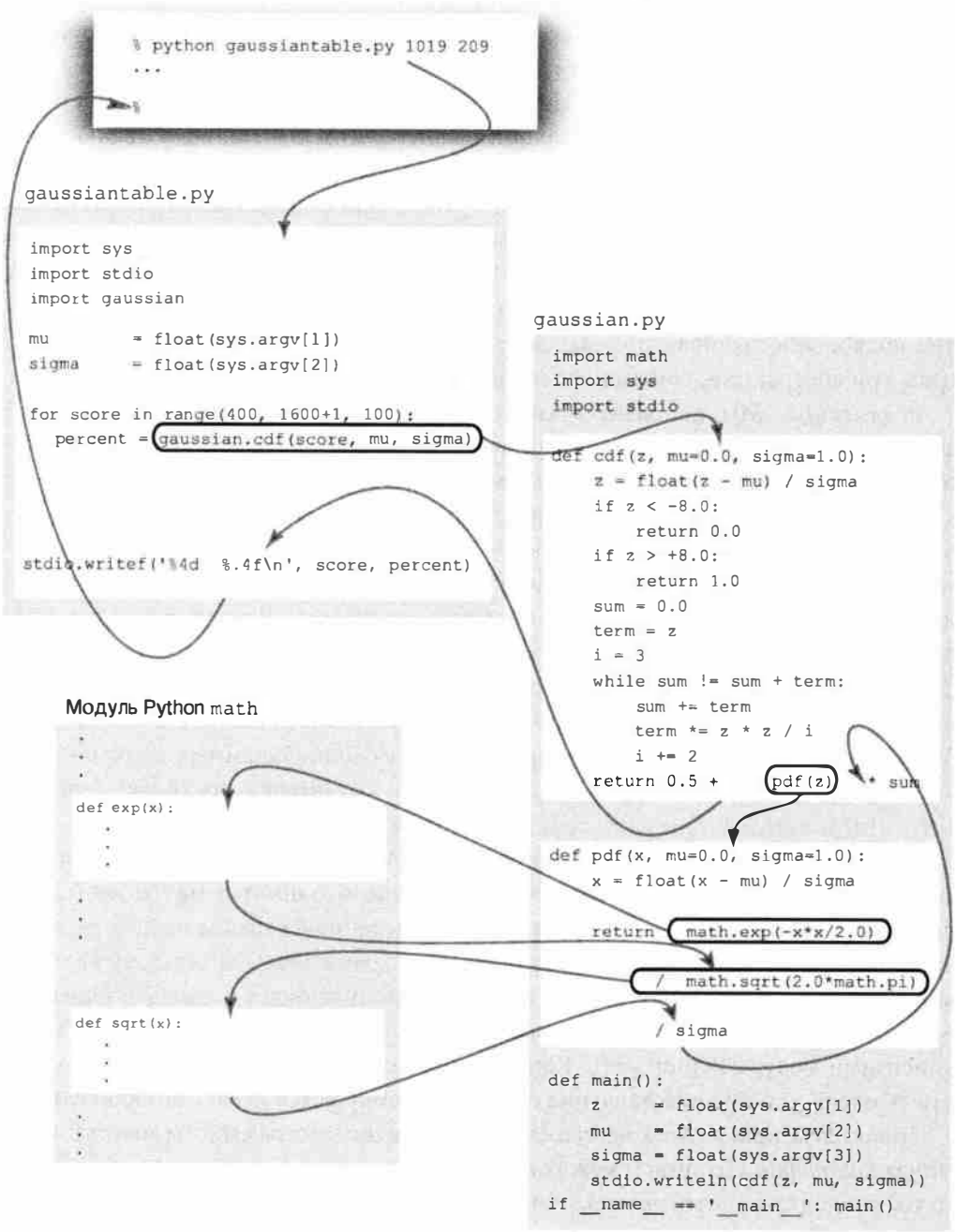
Модульное программирование. Определение нескольких файлов, каждый из которых является независимым модулем с несколькими функциями, является еще одним серьезным изменением в нашем стиле программирования. Этот подход называется *модульным программированием.* Мы независимо разрабатываем и отлаживаем функции, а затем используем их в последующих разработках. Многочисленные примеры данного раздела помогут привыкнуть к этой идее.

Если вы будете рассматривать каждую составляемую программу как нечто, что можно использовать позднее, то довольно скоро обзаведетесь всякого рода полезными инструментами. Модульное программирование позволяет рассматривать решение каждой вычислительной задачи как возможность дополнить свою среду разработки.

Предположим, например, что необходимо вычислять Гауссову функцию кумулятивного распределения для некоторого будущего приложения. Почему бы просто не скопировать и не вставить код реализации функции `cdf()` из предыдущей программы `gauss.py`? Это вполне сработало бы, но появятся две копии кода, что затруднит его поддержку. Если впоследствии понадобится исправить или улучшить код, то придется сделать это для обеих копий. Вместо этого можете собрать свой код из блоков, как мы сделали только что, превращая программу 2.1.2 (`gauss.py`) в программу 2.2.1 (`gaussian.py`). Впоследствии вы сможете использовать любую функцию из своего модуля, добавив оператор `import` *`модуль`* в начало программы и предваряя вызовы функций частью "*`модуль`*.".

Воздействие модульного программирования на управление потоком выполнения программы Python заслуживает особого внимания. В нашем следующем весьма простом примере управление передается от сценария `gaussiantable.py` к функции `cdf()` из модуля `gaussian.py`, затем к функции `pdf()`, затем к функциям `exp()` и `sqrt()` из модуля Python `math`, потом назад к функциям `pdf()` и `cdf()` из модуля `gaussian.py` и наконец назад, к сценарию `gaussiantable.py`. В типичном модульном приложении поток выполнения проходит среди нескольких модулей, как будет продемонстрировано далее, в примерах этого раздела. Каждый модуль полезен и используется многими другими модулями и сценариями.

Ключевое преимущество модульного программирования и основная причина его использования программистами в том, что оно позволяет разделять вычисление на меньшие части, которые удобнее отлаживать и проверять индивидуально. В принципе *все* программы следует составлять, разумно разделяя вычисление на части управляемого размера, и реализовать каждую часть так, как будто кто-то будет впоследствии их использовать. Такой подход важен и предпочтителен даже для небольших программ. Даже если не кто-то, то вы сами поблагодарите себя за экономию усилий по повторной разработке и отладке кода. В этой книге мы собираем из блоков любую программу, содержащую функцию, которую собираемся использовать многократно.



Поток выполнения в модульной программе

Модульные абстракции программирования. Одно из важнейших преимуществ программирования на языке Python заключается в том, что много функций уже предопределено автоматически и для использования вам доступны буквально тысячи модулей Python. Но в этом разделе мы сосредоточимся на более важной идее — возможности создания *пользовательских модулей*, являющихся не более чем файлами, содержащими ряд связанных функций для использования другими программами. Ни один модуль Python (или библиотека модулей) не может содержать все функции, которые могут понадобиться для данного вычисления, таким образом, эта способность — решающий фактор в решении сложных программных задач. Для управления процессом применим проверенный временем подход, обеспечивающий максимум гибкости во время разработки. Рассмотрим три абстракции, служащие основанием для этого подхода.

Реализация. Мы используем общий термин *реализация* (implementation) для обозначения кода, реализующего набор функций, предназначенный для многократного использования. Модуль Python — это реализация: мы обращаемся к функциям из набора, указывая предварительно имя *модуль*, и сохраняем их в файле *модуль*.py. Пример реализации — файл gaussian.py. Выбор правильного набора функций для группировки и реализации является искусством и одной из ключевых задач при разработке большой программы. Руководящий принцип при проектировании модуля — *предоставить клиентам только необходимые функции, и никаких других.* Модуль с огромным количеством функций может быть тяжело реализовать, а модуль с недостатком важных функций не будет удобен для клиентов. Примеры этого принципа уже были. Например, модуль Python math не содержит функций секанса, косеканса и котангенса, поскольку функции math.sin(), math.cos() и math.tan() позволяют вычислять их.

Клиент. Мы употребляем общий термин *клиент* (client) для обозначения программы, использующей реализацию. Мы говорим, что программа Python (сценарий или модуль), вызывающая функцию, определенную в файле *модуль*.py, являются клиентом модуля *модуль*. Например, программа gaussiantable.py является клиентом модуля gaussian.py. Как правило, у модуля будет несколько клиентов: все созданные ранее программы, вызывавшие функцию math.sqrt(), являются клиентами модуля Python math. Когда вы реализуете новый модуль, необходимо иметь очень четкое представление о том, что собираются делать его клиенты.

Интерфейс прикладных программ (API). Обычно программисты мыслят в терминах *контракта* (contract) между клиентом и реализацией, четко определяющего то, что должна делать реализация. Этот подход позволяет многократно использовать код. Возможность составлять программы, являющиеся клиентами модулей math, random и многих других стандартных модулей Python, обусловлена неформальным контрактом (англоязычным описанием их назначения) наряду с точной спецификацией сигнатур функций, доступной пользователям. Совместно

эта информация составляет *интерфейс прикладных программ* (Application Programming Interface — API). Для модулей, определяемых пользователем, применим тот же механизм. API позволяет любому клиенту использовать модуль без всякой необходимости исследовать код модуля, как это было с модулями `math` и `random`. Создавая новый модуль, всегда предоставляйте его API. Например, API для нашего модуля `gaussian.py` представлены далее.

Сколько информации должен содержать API? Эта спорная тема горячо обсуждается программистами. Можно было бы попытаться поместить в API так много информации, сколько возможно, но (как и с любым контрактом!) есть ограничения на объем продуктивной информации. В этой книге мы придерживаемся принципа, отвечающего нашему общему принципу проекта: *предоставить клиентам только необходимую информацию, и не больше.* Это дает нам значительно больше гибкости, чем альтернатива, подразумевающая предоставление подробной информации о реализациях. Действительно, любая дополнительная информация создает неявное расширение контракта, что нежелательно.

API для нашего модуля `gaussian`

Вызов функции	Описание
`gaussian.pdf(x, mu, sigma)`	Функция Гауссовой плотности вероятностей $\varphi(x, \mu, \sigma)$
`gaussian.cdf(z, mu, sigma)`	Функция Гауссова кумулятивного распределения $\Phi(x, \mu, \sigma)$

Примечание. Стандартное значение для `mu` — `0.0`, для `sigma` — `1.0`.

Закрытые функции. Иногда в модуле имеет смысл определить вспомогательную функцию, не предназначенную для непосредственного вызова клиентами. Это *закрытая функция* (private function). В соответствии с соглашением языка Python, первым символом имени закрытой функции является подчеркивание. Следующий пример — альтернативная реализация функции `pdf()` из файла `gaussian.py`, которая вызывает закрытую функцию `_phi()`:

```
def _phi(x):
    return math.exp(-x*x/2.0) / math.sqrt(2*math.pi)
def pdf(x, mu=0.0, sigma=1.0):
    return _phi(float((x - mu) / sigma)) / sigma
```

Закрытые функции не включают в API, поскольку они не являются частью контракта между клиентами и реализациями. Первый символ подчеркивания в имени функции предупреждает клиентов *не вызвать* эту функцию явно. (К сожалению, у языка Python нет никакого механизма жесткого обеспечения выполнения этого соглашения.)

Библиотеки. Библиотека (library) — это коллекция взаимосвязанных модулей. Например, у Python есть стандартная библиотека (включающая модули `random` и `math`) и много библиотек расширения (таких, как `NumPy` для научных

вычислений и Pygame для графики и звука). Кроме того, для этой книги мы предоставляем собственную библиотеку (включающую модули stdio и stddraw). Информацию о представляющих интерес модулях и библиотеках мы предоставляем повсюду в книге. Получив больше опыта в программировании на языке Python, вы будете лучше справляться с диапазонами и областями видимости доступных для вас библиотек.

Документация. API всех стандартных модулей, модулей расширения и книжных модулей доступны благодаря встроенной функции help(), вызываемой в интерактивном режиме Python. Как показано ниже, для этого достаточно ввести команду python (чтобы перейти в интерактивный режим Python), ввести оператор import *модуль* (чтобы загрузить модуль), а затем ввести команду help(*модуль*), чтобы просмотреть API модуля *модуль*. API стандартных модулей и модулей расширения Python доступны также в форме сетевой документации Python; подробности приведены на сайте книги. API некоторых стандартных модулей Python и некоторых из наших книжных модулей обсуждались в предыдущих разделах. В разделе 1.5 предоставлены API модулей stdio, stddraw и stdaudio, а в разделе 1.4 предоставлена часть API модуля stdarray.

```
% python
...
>>> import stddraw
>>> help(stddraw)

Help on module stddraw:

NAME
    stddraw - stddraw.py

FILE
    .../stddraw.py

DESCRIPTION
    The stddraw module defines functions that allow the user
    to create a drawing. A drawing appears on the canvas.
    The canvas appears in the window.

FUNCTIONS
    circle(x, y, r)
        Draw a circle of radius r centered at (x, y).

    filledCircle(x, y, r)
        Draw a filled circle of radius r centered at (x, y).

    filledPolygon(x, y)
        Draw a filled polygon with coordinates (x[i], y[i]).
...
```

Доступ к API в интерактивном режиме Python

Каждый модуль Python и каждый модуль, созданный нами, являются реализацией некого API, но ни один интерфейс API не имеет смысла без некой реализации, а никакая реализация не имеет смысла без клиента. Наша задача при разработке реализации — соблюсти условия контракта. Зачастую для этого есть много путей. Идея отделения клиентского кода от кода реализации при помощи API дает нам свободу замены и улучшения реализации. Эта мощная идея хорошо служит программистам на протяжении многих десятилетий.

Далее мы представим API нашего модуля `stdrandom` (для создания случайных чисел), нашего модуля `stdarray` (для одно- и двумерных массивов) и нашего модуля `stdstats` (для статистических вычислений). Мы обсудим реализации некоторых из функций этих модулей, чтобы проиллюстрировать, как вы могли бы реализовать собственный модуль. Но мы не будем рассматривать все реализации, поскольку для этого нет никаких разумных причин, как и нет причин демонстрировать реализации всех функций в модуле Python `math`. Мы также опишем некоторые из наиболее интересных клиентов этих модулей. Нашей задачи рассмотреть все эти модули вполне достаточно. Во-первых, они предоставляют богатейший набор средств программирования для вашего использования, применимых при разработке сложных клиентских программ. Во-вторых, они служат примером для самостоятельного изучения, когда вы начнете разрабатывать собственные модули и создавать собственные модульные программы.

Случайные числа. Мы уже составили несколько программ, использующих модуль Python `random`, но наш код зачастую использует специфические идиомы для обеспечения типа распределения случайных чисел, необходимого для конкретного приложения. Например, мы рассматривали код случайной перетасовки массива в разделе 1.4 и код рисования случайных значений из дискретного распределения в разделе 1.6.

Чтобы эффективнее многократно использовать наш код, реализующий эти идиомы, мы будем с этого момента использовать модуль `stdrandom` (программа 2.2.3), включающий функции создания случайных чисел в соответствии с различными распределениями и функции перетасовки массива. API этих функций приведены далее. Поскольку эти функции уже вам знакомы, коротких описаний в API вполне достаточно для определения их действий. Как обычно, для использования этих функций код клиента должен включать оператор `import`, а модуль `stdrandom.py` должен находиться в том же каталоге, что и код клиента, или быть доступным Python с использованием механизма вашей операционной системы (см. раздел "Вопросы и ответы").

Собрав все функции, использующие модуль `random` для создания случайных чисел различных типов в одном файле (`stdrandom.py`), мы концентрируем свои усилия по созданию случайных чисел на этом файле (и многократно используем его код), а не разбрасываем их по всем программам, использующим эти функции.

Каждая программа, использующая одну из этих функций, куда понятней, чем код, непосредственно вызывающий функцию `random.random()`, поскольку его цель ясно сформулирована выбором функции модуля `stdrandom`. В некоторых случаях реализации доступны или могли бы стать доступными в других библиотеках Python. Практически мы могли бы использовать эти реализации (действительно, наш книжный код мог бы отличаться от программы 2.2.3). Создание собственного API предоставляет нам свободу делать это без необходимости изменять любой клиентский код.

API для нашего модуля `stdrandom`

Вызов функции	Описание
`uniformInt(lo, hi)`	Однородно случайное целое число в диапазоне `[lo, hi)`
`uniformFloat(lo, hi)`	Однородно случайное целое число типа `float` в диапазоне `[lo, hi)`
`bernoulli(p)`	`True` с вероятностью `p` (стандартное значение `p` — `0.5`)
`binomial(n, p)`	Количество орлов в `n` бросках монеты с вероятностью выпадения орла `p` (стандартное значение `p` — `0.5`)
`gaussian(mu, sigma)`	Нормальное при среднем `mu` и среднеквадратичном отклонении `sigma` (стандартные значения `mu` и `sigma` — `0.0` и `1.0` соответственно)
`discrete(a)`	`i` с вероятностью, пропорциональной `a[i]`
`shuffle(a)`	Случайно перетасовывает массив `a[]`

Проект API. Мы делаем определенные предположения об объектах, передаваемых каждой функции в модуле `stdrandom`. Например, мы подразумевали, что клиенты передадут функции `stdrandom.bernoulli()` число типа `float` в диапазоне от `0.0` до `1.0`, а функции `stdrandom.discrete()` — массив неотрицательных чисел (не все из которых являются нулем). Такие предположения — часть контракта между клиентом и реализацией. Мы стремимся разрабатывать модули таким образом, чтобы контракт был ясен и однозначен (он не должен иметь скрытых деталей). Подобно многим задачам в программировании, хороший проект API зачастую является результатом нескольких циклов испытаний при разных возможностях. Мы всегда проявляем особую осторожность при разработке API, поскольку при изменении API может понадобиться изменить код как всех клиентов, так и всех реализаций. *Наша задача заключается в том, чтобы четко сформулировать то, что клиенты должны ожидать от API, вне зависимости от кода.* Эта практика позволяет изменять код и, возможно, реализацию, достигая желаемого эффекта эффективней или точней.

Программа 2.2.3. Модуль случайных чисел (stdrandom.py)

```
import math
import random

def uniformInt(lo, hi):
    return random.randrange(lo, hi)

def uniformFloat(lo, hi):
    return random.uniform(lo, hi)

def bernoulli(p=0.5):
    return random.random() < p

def binomial(n, p=0.5):
    heads = 0
    for i in range(n):
        if bernoulli(p): heads += 1
    return heads

def gaussian(mu=0.0, sigma=1.0):
    # См. упражнение 2.1.25.

def discrete(a):
    r = uniformFloat(0.0, sum(a))
    subtotal = 0.0
    for i in range(len(a)):
        subtotal += a[i]
        if subtotal > r: return i

def shuffle(a):
    # См. упражнение 2.2.13.
```

Этот модуль определяет функции, реализующие различные типы случайности: случайные целые или вещественные числа, однородно распределенные в данном диапазоне, случайное логическое значение (Бернулли), случайные целые числа, полученные из биномиального распределения, случайные вещественные числа, полученные из Гауссова распределения, случайные целые числа, полученные из данного дискретного распределения, а также функции случайной перестановки содержимого массива. Клиентская проверка приведена ниже.

```
% python stdrandom.py 5
90 26.36076 False 47 8.79269 0
13 18.02210 False 55 9.03992 1
58 56.41176 True  51 8.80501 0
29 16.68454 False 58 8.90827 0
85 86.24712 True  47 8.95228 0
```

Модульное тестирование. Мы реализуем модуль `stdrandom` независимо от любого конкретного клиента, однако хорошей практикой программирования является включение функции *клиентской проверки* `main()`, которая как минимум

- проверяет весь код;
- обеспечивает некую гарантию работоспособности кода;
- получает аргумент из командной строки, обеспечивая гибкость проверки.

Хотя функция `main()` и не предназначается для клиентов, она используется при отладке, проверке и улучшении функций в модуле. Это практика *модульного тестирования* (unit testing). Например, клиентская проверка для модуля `stdrandom` представлена ниже (проверка функции `shuffle()` опущена, см. упр. 1.4.22). Как показано в программе 2.2.1, при вызове этой функции после ввода команды **`python stdrandom.py 10`** неожиданностей в выводе не будет: в первом столбце будут случайные целые числа от `0` до `99` с одинаковой вероятностью; во втором столбце — однородно распределенные случайные числа от `10.0` до `99.0`; примерно половиной значений в третьем столбце будет `True`; числа в четвертом столбце близки к `50`; среднее чисел в пятом столбце составляет приблизительно `9.0`, и маловероятно наличие значений, слишком далеких от `9.0`; значения последнего столбца недалеки от 50% нулей, 30% единиц, 10% двоек и 10% троек. Если что-то кажется неправильным, мы можем получить еще много результатов, введя команду **`python stdrandom.py 100`**. Как правило, хорошая практика программирования подразумевает детализацию функции `main()`, позволяющую получить более исчерпывающую проверку, если библиотека будет использоваться интенсивно.

Надлежащее модульное тестирование может быть достаточно сложной программной задачей само по себе. В данном случае уместна более обширная проверка в отдельном клиенте, позволяющая удостовериться в том, что у случайных чисел будут именно те свойства, которые соответствуют заданным распределениям (см. упр. 2.2.2). *Примечание.* Эксперты все еще обсуждают наилучший способ проверки характеристик чисел, подобных создаваемым функциями модуля `stdrandom`, например, действительно ли эти числа случайны.

```
main():
    trials = int(sys.argv[1])
    for i in range(trials):
        stdio.writef('%2d ' , uniformInt(10, 100))
        stdio.writef('%8.5f ', uniformFloat(10.0, 99.0))
        stdio.writef('%5s ' , bernoulli(0.5))
        stdio.writef('%2d ' , binomial(100, 0.5))
        stdio.writef('%7.5f ', gaussian(9.0, 0.2))
        stdio.writef('%1d ' , discrete([5, 3, 1, 1]))
        stdio.writeln()

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

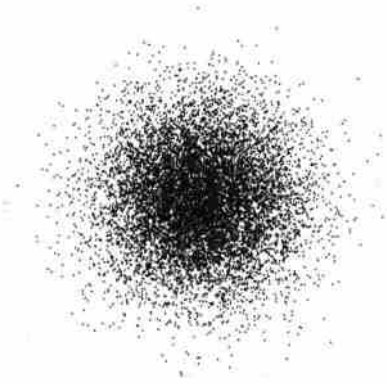
Простая клиентская проверка модуля `stdrandom`

Один из эффективных подходов создания клиентской проверки, использующей модуль stddraw, подразумевает визуализацию результирующих данных, наглядно демонстрирующую правильность поведения программы. В данном случае рисование большого количества точек, чьи координаты x и y получены из разных распределений, зачастую создает узор, непосредственно демонстрирующий важнейшие свойства распределения. Однако важней всего то, что на таких рисунках ошибки создания случайных чисел обнаруживаются немедленно. Пример сценария проверки функции stdrandom.gaussian() будет представлен далее.

Стресс-проверка. Интенсивно используемый модуль, такой как stdrandom, должен также пройти *стресс-проверку* (stress test), позволяющую удостовериться в отказоустойчивости модуля, даже когда клиент не следует контракту или делает некое предположение, явно идущее вразрез с ним. Все стандартные модули Python прошли такую проверку, требующую тщательного исследования каждой строки кода и выявления всех условий, способных вызвать проблему. Что должна делать функция stdrandom.discrete(), если некоторые элементы массива отрицательны? Или если являющийся аргументом массив имеет нулевую длину? Что должна делать функция stdrandom.uniform(), если второй аргумент меньше или равен первому аргументу? Справедлив любой вопрос, который может прийти вам в голову. Это *крайние случаи* (corner case). Уверен, вам встречались учителя или начальники, являвшиеся сторонниками крайних случаев. На практике большинство программистов стремятся решать такие вопросы на ранних этапах, избегая неприятной встречи с ними впоследствии, во время отладки. И снова имеет смысл реализовать стресс-проверку как отдельный клиент.

```
import sys
import stddraw
import stdrandom

trials = int(sys.argv[1])
stddraw.setPenRadius(0.0)
for i in range(trials):
    x = stdrandom.gaussian(0.5, 0.2)
    y = stdrandom.gaussian(0.5, 0.2)
    stddraw.point(x, y)
stddraw.show()
```

API модуля обработки массива. В разделе 1.4 мы ознакомились с удобной функцией создания одномерных массивов заданной длины и двумерных массивов с заданным количеством рядов и столбцов. Таким образом, мы уже знакомы с модулем stdarray из библиотеки с сайта книги, а именно с ее функциями для создания и инициализации массивов stdarray.create1D() и stdarray.create2D().



Клиент проверки функции stdrandom.gaussian()

Кроме того, вскоре мы рассмотрим несколько примеров, где будем читать значения со стандартного ввода в массив и выводить значения из массива на стандартный вывод. Соответственно, чтобы читать массивы целых, вещественных и логических значений со стандартного ввода, а также выводить их на стандартный вывод, мы подключим функции модуля `stdarray`, дополнив, таким образом, модуль `stdio`. Вот полный API для модуля `stdarray`:

API для нашего модуля `stdarray`

Вызов функции	Описание
`create1D(n, val)`	Создает массив длиной `n` элементов, инициализированных значением `val`
`create2D(m, n, val)`	Создает массив `m` на `n` элементов, инициализированных значением `val`
`readInt1D()`	Читает со стандартного ввода массив целых чисел
`readInt2D()`	Читает со стандартного ввода двумерный массив целых чисел
`readFloat1D()`	Читает со стандартного ввода массив вещественных чисел
`readFloat2D()`	Читает со стандартного ввода двумерный массив вещественных чисел
`readBool1D()`	Читает со стандартного ввода массив логических значений
`readBool2D()`	Читает со стандартного ввода двумерный массив логических значений
`write1D(a)`	Выводит массив `a[]` на стандартное устройство вывода
`write2D(a)`	Выводит двумерный массив `a[]` на стандартное устройство вывода

Примечание 1. Формат 1D — целое число `n`, сопровождаемое `n` элементами.
Формат 2D — два целых числа `m` и `n`, сопровождаемых `m` и `n` элементов в порядке по рядам.

Примечание 2. Логические переменные вводятся как `0` и `1`, а не `False` и `True`.

Как обычно, вы можете также найти эту информацию при работе в интерактивном режиме Python: для просмотра API введите сначала команду `import stdarray`, а затем команду `help(stdarray)`.

Функции чтения и записи массива должны "договориться" о *формате файла.* Для простоты и гармонии имеет смысл принять соглашение о том, как выглядит массив на стандартном устройстве ввода, включая размерности. Функции `read*()` ожидают данные в этом формате, а функции `write*()` — выводят в этом формате. Мы можем легко создать в этом формате файл для данных, поступающих из другого источника.

Для массивов логических данных наш формат файла использует `0` и `1` вместо `False` и `True`. Для больших массивов это соглашение намного экономичней. Однако важнее всего то, что с этим форматом файла намного проще определять данные, как будет продемонстрировано в разделе 2.4.

При наличии кода обработки массива, рассматриваемого в разделах 1.4 и 2.1, реализация этих функций довольно проста. Опустим реализацию этого модуля,

поскольку лежащий в его основе код уже изучен. Если интересно, найдите полную реализацию в файле `stdarray.py` на сайте книги.

Упаковка всех этих функций в один файл (`stdarray.py`) облегчает многократное использование кода и позволяет не волноваться о подробностях создания, вывода и чтения массивов при написании впоследствии клиентских программ. Кроме того, наличие в клиентских программах только вызовов этих функций вместо реализующего их кода делает их куда компактней и понятней.

Форматы файлов для массивов

		Инициализатор массива	Файл
`float`	Одномерный	`[0.01, 0.85, 0.07, 0.07]`	`4` `.01 .85 .07 .07`

	Двумерный	`[[ 0.00, 0.00, 0.500],`
		`[ 0.85, 0.04, -0.075],`
		`[ 0.20, -0.26, 0.400],`
		`[-0.15, 0.28, 0.575]]`
			`4 3`
			`.00 .00 .500`
			`.85 .04 -.075`
			`.20 -.26 .400`
			`-.15 .28 .575`

`bool`	Одномерный	`[False, True, True]`	`3` `0 1 1`

	Двумерный	`[[False, True, False],`
		`[ True, False, True ],`
		`[ True, False, True ],`
		`[False, True, False]]`
			`4 3`
			`0 1 0`
			`1 0 1`
			`1 0 1`
			`0 1 0`

Системы итерационных функций. Оказалось, что простые вычислительные процессы способны создавать весьма сложные визуальные изображения. Модули `stdrandom`, `stddraw` и `stdarray` позволяют легко изучить поведение таких систем.

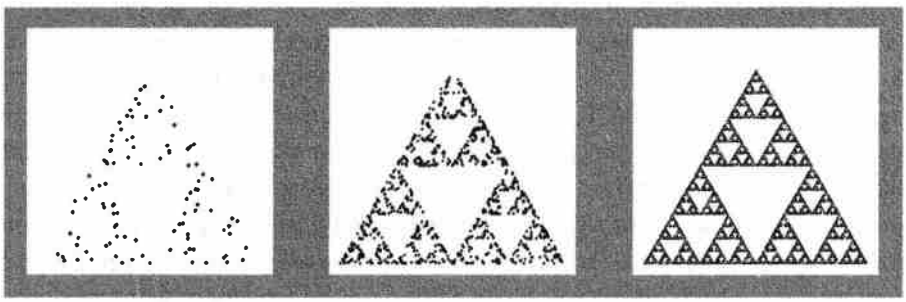
Треугольник Серпинского. В качестве первого примера рассмотрим следующий простой процесс: сначала рисуем точку в одной из вершин заданного равностороннего треугольника. Затем выбираем наугад одну из этих трех вершин и рисуем новую точку на полпути между первой точкой и этой вершиной. Продолжаем выполнять те же действия. Чтобы провести линию, середина которой будет следующей рисуемой точкой, каждый раз выбирается случайная вершина треугольника. Поскольку выбор каждый раз осуществляется случайно, у набора точек должны были бы быть некоторые из характеристик случайных значений, и после нескольких первых итераций это, казалось бы, действительно имеет место.

Случайный процесс

Но мы можем изучить процесс для больших количеств итераций, создав сценарий для рисования `n` точек согласно правилам:

```
cx = [0.000, 1.000, 0.500]
cy = [0.000, 0.000, 0.866]
x = 0.0
y = 0.0
for i in range(n):
    r = stdrandom.uniformInt(0, 3)
    x = (x + cx[r]) / 2.0
    y = (y + cy[r]) / 2.0
    stddraw.point(x, y)
stddraw.show()
```

Сохраним координаты x и y вершин треугольника в массивах `cx[]` и `cy[]` соответственно. Для выбора случайного индекса `r` в этих массивах используем функцию `stdrandom.uniformInt()`, а координатами выбранной вершины будут элементы `cx[r]` и `cy[r]`. Координата x середины линии из точки (x, y) к этой вершине вычисляется выражением `(x + cx[r]) / 2.0`, а координата y — подобным выражением. Добавление вызова функции `stddraw.point()` и помещение этого кода в цикл завершает реализацию. Просто замечательно, но, несмотря на случайность, после большого количества итераций всегда получается тот же рисунок! Это *треугольник Серпинского* (Sierpinski triangle), см. упражнение 2.3.27. Весьма интересный вопрос: почему такой регулярный узор является результатом случайного процесса?

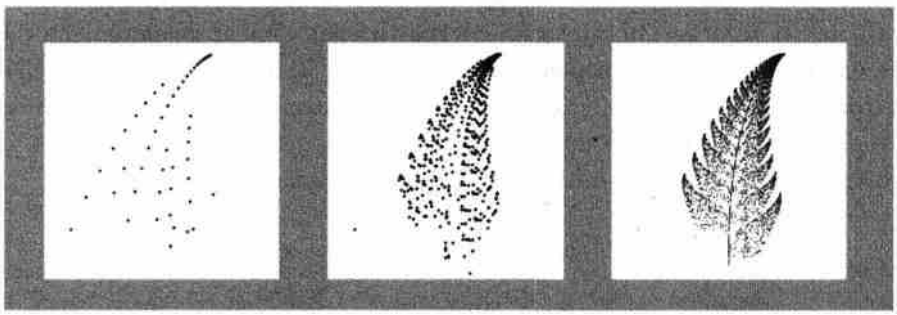


Случайный ли процесс?

Папоротник Барнсли. Добавим интригу — играя в ту же игру, но с разными правилами, можно получить множество разнообразных узоров. Один из примеров — *папоротник Барнсли* (Barnsley fern). Для его создания используем тот же процесс, но на сей раз управляемый следующей таблицей формул. На каждом этапе мы выбираем формулы, используемые для модификации координат x и y, с указанной вероятностью (первую пару формул мы используем с вероятностью 1%, вторую пару — с вероятностью 85% и так далее).

Вероятность	Модификация по x	Модификация по y
1%	$x = 0.500$	$x = 0.16y$
85%	$x = 0.85x + 0.04y + 0.075$	$y = -0.04x + 0.85y + 0.180$
7%	$x = 0.20x - 0.26y + 0.400$	$y = 0.23x + 0.22y + 0.045$
7%	$x = -0.15x + 0.28y + 0.575$	$y = 0.26x + 0.24y - 0.086$

Для итерации этих правил мы могли бы составить код точно так же, как код создания треугольника Серпинского, но матричная обработка предоставляет единообразный способ обобщения этого кода, позволяющий выполнять любой свод правил. Имеется m разных преобразований, выбираемых из вектора 1 на m функцией `stdrandom.discrete()`. Для каждого преобразования имеется уравнение, модифицирующее координату x, и уравнение, модифицирующее координату y. Таким образом, мы используем две матрицы m на 3 для коэффициентов уравнения: одну для x и одну для y. Программа 2.2.4 (`ifs.py`) реализует эту управляемую данными версию вычисления. Эта программа предоставляет безграничное поле для исследования: она осуществляет итерацию для любого содержащего вектор ввода, определяющего распределение вероятности, и две матрицы, определяющие коэффициенты для модификации x и y. При заданных коэффициентах, даже при том, что на каждом этапе выбираются случайные уравнения, каждый раз получается то же изображение, очень похожее на лист папоротника, который вы могли бы увидеть в лесу, но созданное случайным процессом на компьютере.



Создание папоротника Барнсли

Эта короткая программа получает несколько чисел со стандартного устройства ввода и, рисуя точки на стандартном графическом устройстве, может (согласно введенным данным) создать треугольник Серпинского или папоротник Барнсли (и многие, многие другие изображения). Из-за ее простоты и привлекательности результатов вычисления такого сорта полезны при создании синтетических изображений с реалистичным внешним видом в компьютерных мультфильмах и играх.

Возможно, еще более значительным последствием способности легко создавать такие реалистичные изображения является вполне резонный научный интерес: что вычисление может сказать нам о природе и что природа говорит нам о вычислении?

Программа 2.2.4. Система итерационных функций (`ifs.py`)

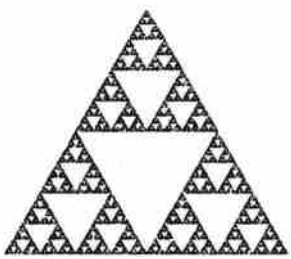
```
import sys
import stdarray
import stddraw
import stdrandom

n = int(sys.argv[1])
probabilities = stdarray.readFloat1D()
cx = stdarray.readFloat2D()
cy = stdarray.readFloat2D()
x = 0.0
y = 0.0
stddraw.setPenRadius(0.0)
for i in range(n):
    r = stdrandom.discrete(probabilities)
    x0 = cx[r][0]*x + cx[r][1]*y + cx[r][2]
    y0 = cy[r][0]*x + cy[r][1]*y + cy[r][2]
    x = x0
    y = y0
    stddraw.point(x, y)
stddraw.show()
```

`n`	Итерации
`probabilities[]`	Вероятности
`cx[][]`	Коэффициенты x
`cy[][]`	Коэффициенты y
`x, y`	Текущая точка

Этот управляемый данными сценарий является клиентом модулей `stdarray`, `stdrandom` и `stddraw`. Он запускает систему итерационных функций, определенную вектором $1 \times m$ (вероятностей) и двумя матрицами $m \times 3$ (коэффициенты модификации x и y соответственно), полученными со стандартного устройства ввода, и рисующую точками результат на стандартном графическом устройстве. Любопытно, но этот код не обращается к `m`.

```
% more sierpinski.txt
3
  .33  .33  .34
3 3
  .50  .00  .00
  .50  .00  .50
  .50  .00  .25
3 3
  .00  .50  .00
  .00  .50  .00
  .00  .50  .433
```

```
% python ifs.py 100000 < sierpinski.txt
```



```
% more barnsley.txt
4
  .01   .85   .07   .07
4 3
  .00   .00   .500
  .85   .04   .075
  .20  -.26   .400
 -.15   .28   .575
4 3
  .00   .16   .000
 -.04   .85   .180
  .23   .22   .045
  .26   .24  -.086
```

```
% python ifs.py 100000 < barnsley.txt
```

```
% more tree.txt
6
  .1   .1   .2   .2   .2   .2
6 3
  .00   .00   .550
 -.05   .00   .525
  .46  -.15   .270
  .47  -.15   .265
  .43   .26   .290
  .42   .26   .290
6 3
  .00   .60   .000
 -.50   .00   .750
  .39   .38   .105
  .17   .42   .465
 -.25   .45   .625
 -.35   .31   .525
```

```
% python ifs.py 100000 < tree.txt
```

```
% more coral.txt
3
  .40   .15   .45
3 3
  .3077  -.5315   .8863
  .3077  -.0769   .2166
  .0000   .5455   .0106
3 3
 -.4615  -.2937  1.0962
  .1538  -.4476   .3384
  .6923  -.1958   .3808
```

```
% python ifs.py 100000 < coral.txt
```

Примеры систем итерационных функций

Стандартная статистика. Теперь рассмотрим модуль для набора математических вычислений и базовых инструментальных средств визуализации (не все из которых реализованы в стандартных модулях Python), применимых во всякого

рода научных и технических приложениях. Эти вычисления связаны с задачами выявления статистических свойств последовательности чисел. Такой модуль пригодится, например, при обработке результатов серии научных экспериментов, полученных в результате измерений. Одной из главнейших задач, стоящих перед современными учеными, является надлежащий анализ таких данных, и этот анализ играет все более важную роль. Эти базовые функции анализа данных нетрудно реализовать, а их API представлены далее.

Базовая статистика. Предположим, имеется n измерений $x_0, x_1, \ldots, x_{n-1}$. *Среднее* (mean) значение этих измерений вычисляется по формуле $\mu = (x_0 + x_1 + \ldots + x_{n-1})/n$ и является оценкой полученного значения. Минимальное и максимальное значения также представляют интерес, как и медиана (значение в середине отсортированного набора, если количество его элементов нечетно, или средние для двух значений в середине, если оно четно). Представляет также интерес *выборочная дисперсия* (sample variance), вычисляемая по формуле

$$\sigma^2 = ((x_0 - \mu)^2 + (x_1 - \mu)^2 + \ldots + (x_{n-1} - \mu)^2)/(n-1)$$

и *выборочное среднеквадратичное отклонение* — квадратный корень выборочной дисперсии. Программа 2.2.5 (`stdstats.py`) — это модуль, содержащий функции для вычисления базовых статистических данных (поскольку вычисление медианы трудней остальных, рассмотрим реализацию функции `median()` в разделе 4.2). Функция клиентской проверки `main()` модуля `stdstats` читает со стандартного ввода числа в массив и вызывает каждую из функций. Подобно модулю `stdrandom`, данному модулю требуется более обширная проверка вычислений. Как обычно, мы отлаживаем и проверяем новые функции в модуле и соответственно корректируем код модульного тестирования, проверяя функции по одной. Столь зрелый и широко используемый модуль, как `stdstats`, также заслуживает клиента стресс-проверки, чтобы полностью проверять все после любого изменения. Если интересно узнать, как мог бы выглядеть такой клиент, найдите таковой для модуля `stdstats` на сайте книги (см. также упр. 2.2.2). Опытные программисты рекомендуют регулярно проводить модульное тестирование и стресс-проверки при разработке (это сторицей окупится позже).

API для нашего модуля `stdstats`

Вызов функции	Описание
`mean(a)`	Среднее значение чисел в массиве `a[]`
`var(a)`	Выборочная дисперсия значений в числовом массиве `a[]`
`stddev(a)`	Выборочное среднеквадратичное отклонение значений в числовом массиве `a[]`
`median(a)`	Медиана значений в числовом массиве `a[]`
`plotPoints(a)`	Рисует точки согласно значениям в числовом массиве `a[]`
`plotLines(a)`	Рисует линию согласно значениям в числовом массиве `a[]`
`plotBars(a)`	Рисует полосу согласно значениям в числовом массиве `a[]`

Программа 2.2.5. Модуль анализа данных (stdstats.py)

```
import math
import stdarray
import stddraw
import stdio

def mean(a):
    return sum(a) / float(len(a))

def var(a):
    mu = mean(a)
    total = 0.0
    for x in a:
        total += (x - mu) * (x - mu)
    return total / (len(a) - 1)

def stddev(a):
    return math.sqrt(var(a))

def median(a):
    # См. упражнение 4.2.16.

# Графические функции см. в программе 2.2.6.

def main():
    a = stdarray.readFloat1D()
    stdio.writef(' mean %7.3f\n', mean(a))
    stdio.writef('std dev %7.3f\n', stddev(a))
    stdio.writef(' median %7.3f\n', median(a))

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

Этот модуль реализует функции вычисления минимального, максимального и среднего значений, а также дисперсии и среднеквадратичного отклонения чисел в клиентском массиве. Графические функции находятся в программе 2.2.6.

```
% more tiny1D.txt
7
3.0 1.0 4.0 7.0 8.0 9.0 6.0
```

```
% python stdstats.py < tiny1D.txt
   mean  5.429
std dev  2.878
 median  6.000
```

Графика. Одним из важнейших способов использования модуля `stddraw` является куда более наглядная визуализация данных, чем таблицы чисел. В типичной ситуации мы ставим эксперименты, сохраняем экспериментальные данные в массиве,

а затем сравниваем результаты с некой моделью, возможно, описывающей данные математической функцией. Чтобы ускорить этот процесс для типичного случая, где значения переменной равноудалены, наш модуль `stdstats` определяет функции, которые можно использовать для графического представления данных в массиве. Программа 2.2.6 — это реализация функций `plotPoints()`, `plotLines()` и `plotBars()` для модуля `stdstats`. Эти функции отображают значения переданного в качестве аргумента массива в графическом окне с регулярными интервалами, или соединяют их сегментами (`lines`), или заполняют круги точками (`points`), или рисуют полосы от оси *x* до заданного значения (`bars`). Все они рисуют точки со значением `i` по координате x и значением `a[i]` по координате *y*, используя заполненные круги, линии из точек и полосы соответственно. Они все подбирают масштаб изображения так, чтобы полностью заполнить окно рисунка по координате *x* (точки равномерно расположены вдоль оси *x*), а за масштаб по координате *y* отвечает клиент.

Эти функции не предназначены для формирования универсального графического модуля. Конечно, вы можете придумать всякого рода элементы, которые имеет смысл добавить: различные типы пятен, помеченные оси, цвет и много других элементов, обычно присутствующих в современных системах графического отображения данных. В некоторых немногих ситуациях могли бы понадобиться более сложные функции, чем эти, но наше намерение с модулем `stdstats` заключается в том, чтобы познакомить вас с анализом данных и продемонстрировать простоту определения модуля, а не решать утилитарные задачи. Фактически этот модуль все же оказался полезным — мы использовали его графические функции для рисования иллюстраций в этой книге, изображающих графики функций, звуковые волны и экспериментальные результаты. Затем мы рассмотрим несколько примеров их использования.

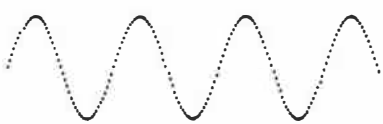
Рисование графиков функции. Функции `stdstats.plot*()` можно использовать для рисования графиков любых функций: выберите интервал по оси *x*, где необходимо нарисовать график функции, вычислите значения функции, равномерно распределенные в этом интервале, и сохраните их в массиве, определите и установите масштаб по оси *y*, а затем вызовите функцию `stdstats.plotLines()` или `plot*()`, как показано в примере ниже. Плавность кривой определяется свойствами функции и количеством рисуемых точек. Как упоминалось при первом рассмотрении модуля `stddraw`, вы должны позаботиться о достаточном количестве выборок для учета флуктуаций функции. В разделе 2.4 мы рассмотрим и другой подход рисования графиков функций на основании выборки не равноудаленных значений. Зачастую необходимо изменение масштаба по оси *y* (масштаб по оси *x* автоматически выбирается функциями модуля `stdstats`). Например, при рисовании графика синусоидальной функции масштаб по оси *y* должен покрывать значения от –1 до +1. Вообще, для выбора масштаба по оси *y* можно вызвать функцию `stddraw.setYscale(min(a), max(a))`.

```
n = int(sys.argv[1])
a = stdarray.create1D(n+1, 0.0)
for i in range(n+1):
    a[i] = gaussian.pdf(-4.0 + 8.0 * i / n)
stdstats.plotPoints(a)
stdstats.plotLines(a)
stddraw.show()
```

Рисование графика функции

Рисование звуковых волн. Модули `stdaudio` и `stdstats` рисуют графики функций, используя массивы, содержащие значения равномерно распределенных выборок, что упрощает рисование звуковых волн. Графики звуковых волн в разделе 1.5 и в начале этого раздела создавались в результате предварительного масштабирования по оси *y*, для получения приемлемого внешнего вида кривой и последующего рисования точек функцией `stdstats.plotPoints()`. Как вы уже видели, такие графики дают прямое представление об обрабатываемом аудио. Вы можете также получить интересные эффекты, рисуя звуковые волны по мере их проигрывания средствами модуля `stdaudio`, хотя эта задача немного сложна из-за огромного объема задействованных данных (см. упр. 1.5.27).

```
def tone(hz, t):
    # См. программу 2.4.7
stddraw.setYscale(-6.0, 6.0)
hi = tone(440, 0.01)
stdstats.plotPoints(hi)
stddraw.show()
```



Рисование звуковой волны

Рисование результатов экспериментов. В тот же рисунок можно поместить несколько графиков. Обычно это делают при необходимости сравнить экспериментальные результаты с теоретической моделью. Например, программа 2.2.7 (`bernoulli.py`) подсчитывает количество орлов при *n* бросках монеты и сравнивает результат с предсказанным функцией Гауссова (нормального) распределения. Известное следствие теории вероятности заключается в том, что это практически биномиальное распределение чрезвычайно хорошо аппроксимирует Гауссова функция φ при плотности вероятностей *n*/2 и среднеквадратичном отклонении. Чем больше испытаний мы проводим, тем точнее приближение. Созданный программой `bernoulli.py` и представленный ниже рисунок является отображением результатов эксперимента и убедительным доказательством

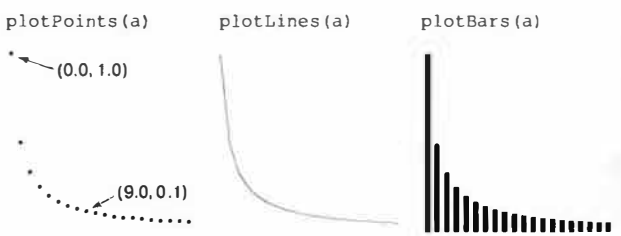
правильности теории. Это пример прототипа научного подхода к программированию приложений, используемый повсюду в этой книге, и вы должны использовать его всякий раз, когда начинаете эксперимент. Если доступна теоретическая модель, способная объяснить ваши результаты, то визуальный график, сравнивающий эксперимент с теорией, поможет сравнить их.

Программа 2.2.6. Рисование значений данных (stdstats.py, продолжение)

```
def plotPoints(a):
    n = len(a)
    stddraw.setXscale(0, n-1)
    stddraw.setPenRadius(1.0 / (3.0 * n))
    for i in range(n):
        stddraw.point(i, a[i])

def plotLines(a):
    n = len(a)
    stddraw.setXscale(0, n-1)
    stddraw.setPenRadius(0.0)
    for i in range(1, n):
        stddraw.line(i-1, a[i-1], i, a[i])

def plotBars(a):
    n = len(a)
    stddraw.setXscale(0, n-1)
    for i in range(n):
        stddraw.filledRectangle(i-0.25, 0.0, 0.5, a[i])
```

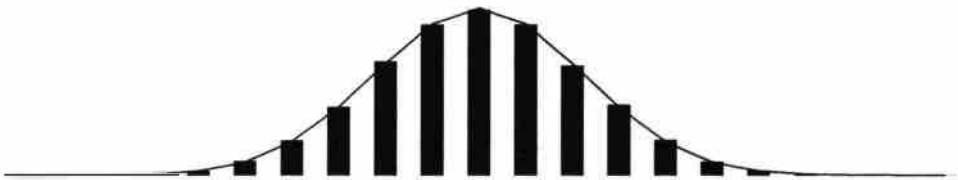
Этот код реализует в программе 2.2.5 (`stdstats.py`) три функции для рисования данных. Получив массив `a[]`, они рисуют по точкам график (`i`, `a[i]`) заполненными кругами, сегментами линий и полосами соответственно. За вызов функции `stddraw.show()` отвечает клиент.

```
n = 20
a = stdarray.create1D(n, 0.0)
for i in range(n):
    a[i] = 1.0 / (i + 1)
```



Эти несколько примеров должны продемонстрировать создание хорошо проработанного модуля функций анализа данных. В упражнениях рассматривается несколько дополнительных модификаций и идей. Вы можете найти, что модуль `stdstats` полезен для простых графиков, поэтому имеет смысл поэкспериментировать с его реализациями и модифицировать их или добавить функции, чтобы создать собственный модуль, позволяющий рисовать графики для собственного проекта. По мере решения новых и расширения круга собственных задач программирования вы, вполне естественно, будете разрабатывать собственные инструментальные средства для своего использования.

```
% python bernoulli.py 20 100000
```



Модульное программирование. Разрабатываемые нами модули иллюстрируют стиль *модульного программирования* (modular programming). Составляя новую программу для решения какой-либо задачи, вовсе не обязательно располагать ее целиком в собственном файле: каждую большую задачу можно разделить на меньшие, более управляемые подзадачи, а затем реализовать и отладить код решения каждой подзадачи независимо. *Всякий раз, когда можете четко разделить задачи в пределах вычисления, так и поступайте.* Язык Python обеспечивает такое разделение, позволяя независимо разрабатывать функции в *модулях*, используемых затем клиентами. Таким образом, Python облегчает модульное программирование, разрешая определять модули для решения важных задач и последующего использования клиентами.

Наша программа 2.2.4, сценарий системы итерационных функций (`ifs.py`), иллюстрирует модульное программирование, поскольку это относительно сложное вычисление, реализуемое несколькими относительно небольшими взаимодействующими модулями, разработанными независимо. Он использует функции модулей `stdrandom` и `stdarray`, а также подходящие функции из модулей `sys` и `stddraw`. Если бы мы поместили весь необходимый код в один файл `ifs.py`, то получился бы очень большой объем кода, сложный в поддержке и отладке; при модульном программировании мы можем изучать системы итерационных функций, будучи вполне уверенными в правильности чтения массивов и создании случайных чисел с правильным распределением значений, поскольку в созданных по отдельности модулях уже есть реализованный и проверенный код для решения этих задач.

Программа 2.2.7. Исследование Бернулли (bernoulli.py)

```
import sys
import math
import stdarray
import stddraw
import stdrandom
import stdstats
import gaussian

n      = int(sys.argv[1])
trials = int(sys.argv[2])

freq = stdarray.create1D(n+1, 0)
for t in range(trials):
    heads = stdrandom.binomial(n, 0.5)
    freq[heads] += 1

norm = stdarray.create1D(n+1, 0.0)
for i in range(n+1):
    norm[i] = 1.0 * freq[i] / trials

phi = stdarray.create1D(n+1, 0.0)
stddev = math.sqrt(n)/2.0
for i in range(n+1):
    phi[i] = gaussian.pdf(i, n/2.0, stddev)

stddraw.setCanvasSize(1000, 400)
stddraw.setYscale(0, 1.1 * max(max(norm), max(phi)))
stdstats.plotBars(norm)
stdstats.plotLines(phi)
stddraw.show()
```

`n`	Количество бросков в эксперименте
`trials`	Количество экспериментов
`freq[]`	Результаты экспериментов
`norm[]`	Нормализованный результат
`phi[]`	Гауссова модель

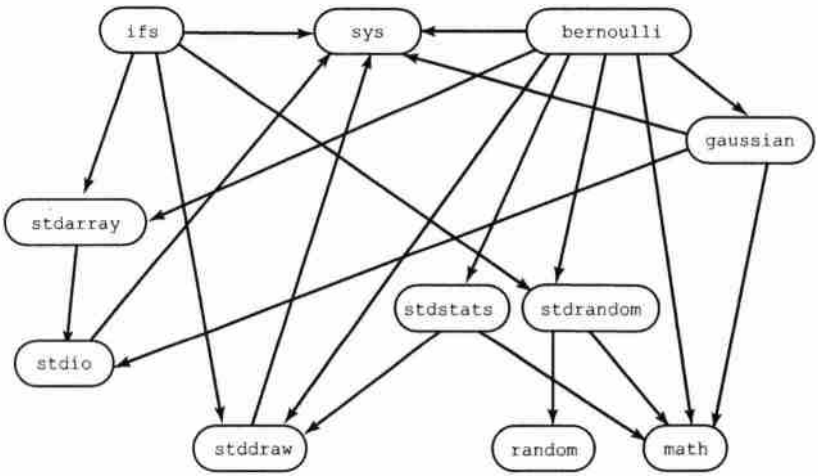
Этот сценарий демонстрирует убедительное визуальное доказательство совпадения наблюдаемого количества орлов при n бросках монеты и Гауссова распределения.

Модули и сценарии этого раздела

Модуль/сценарий	Описание
`gaussian`	Функции Гауссова распределения
`stdrandom`	Функции для случайных чисел
`stdarray`	Ввод и вывод массивов
`ifs`	Клиент для систем итерационных функций
`stdstats`	Функции анализа данных
`bernoulli`	Клиент для экспериментов Бернулли

Программа 2.2.7 (bernoulli.py) также иллюстрирует модульное программирование. Это клиент модулей gaussian, sys, math, stdrandom, stdarray, stddraw и stdstats. И снова у нас есть уверенность, что функции в данных модулях приводят к ожидаемым результатам, поскольку это системные модули или модули, проверенные, отлаженные и использованные прежде.

Для описания отношений между модулями в модульной программе можно нарисовать *граф зависимостей* (dependency graph), где имена модулей соединяются стрелками, если первый модуль использует средства, определенные во втором модуле. Такие схемы играют важную роль, поскольку понимание отношений между модулями необходимо для надлежащей разработки и обслуживания. Граф зависимостей для файлов ifs.py и bernoulli.py, а также используемых ими модулей представлен ниже.



Граф зависимостей для клиентов и модулей данного раздела

Мы подчеркиваем важность модульного программирования повсюду в этой книге, поскольку у него есть много преимуществ, ставших общепринятыми в современном программировании, включая следующие.

- Возможность составлять программы разумного размера, даже для больших систем.
- Отладка ограничивается малыми частями кода.
- Возможность многократно использовать код без необходимости повторно реализовать его.
- Поддержка (и модификация) кода намного проще.

Важность этих преимуществ трудно преувеличить, поэтому рассмотрим их подробней.

Разумный размер программ. Никакая большая задача не настолько сложна, чтобы ее нельзя было разделить на меньшие подзадачи. Если программа простирается на несколько страниц кода, то имеет смысл задаться вопросами: можно ли реализовать некоторые подзадачи отдельно? Можно ли сгруппировать некоторые из этих подзадач логически в отдельном модуле? Могут ли другие клиенты использовать этот код в будущем? С другой стороны, при наличии огромного количества крошечных модулей возникают другие вопросы: принадлежит ли логически некая группа подзадач к тому же модулю? Вероятно ли, что каждый модуль будет использоваться несколькими клиентами? Нет никакого жесткого правила о размере модуля: одна реализация критически важной абстракции может занимать лишь несколько строк кода, тогда как другой модуль с большим количеством функций мог бы простираться на сотни строк кода.

Отладка. Увеличение количества операторов и взаимодействующих переменных быстро усложняет трассировку программы. Трассировка программы с сотнями переменных требует отслеживания сотен объектов, поскольку любой оператор может затрагивать любую переменную или взаимодействовать с ней. Для сотен, тысяч и более операторов сделать это весьма затруднительно. При модульным программировании и согласно нашим руководящим принципам разграничения областей видимости локальных переменных мы строго ограничиваем количество возможностей, рассматриваемых при отладке. Не менее важным результатом является идея контракта между клиентом и реализацией. Удостоверившись, что реализация соответствует своей части контракта, можно приступать к отладке всех ее клиентов.

Многократное использование кода. Реализовав модули, такие как `stdrandom` и `stdstats`, можно больше не волноваться о создании кода для вычисления среднего или среднеквадратичного отклонения, а также о создании случайных чисел — можно просто многократно использовать уже имеющийся код. Кроме того, больше не нужно делать копии кода — любой модуль может обратиться к любой функции в любом другом модуле.

Поддержка. Как и хорошую книгу, хорошую программу всегда можно улучшить, и модульное программирование облегчает процесс непрерывного улучшения ваших программ Python, поскольку улучшение модуля улучшает все его клиенты. Обычно есть несколько разных подходов к решению той же проблемы. При модульном программировании вы можете реализовать их все и опробовать независимо. Пока клиентский код полагается только на действия, документированные в API, он должен нормально продолжать работать, несмотря на модификацию или замену реализации. Однако важней всего другой случай: предположим, что при разработке очередного клиента обнаруживается ошибка в некоем модуле. При модульном программировании устранение этой ошибки фактически устранит ее во всех клиентах модуля.

Если вы столкнетесь со старой программой (или новой, составленной старым программистом!), то вероятнее всего она представляет собой один огромный

модуль из длинной последовательности операторов на нескольких страницах или больше, где любой оператор может обратиться к любой переменной в программе. Огромные модули такого вида чрезвычайно трудно понимать, поддерживать и отлаживать. Старые программы этого типа находятся в критически важных частях нашей вычислительной инфраструктуры (например, на некоторых атомных электростанциях и в некоторых банках). Дело в том, что занятые их поддержкой программисты просто не в состоянии понять их достаточно хорошо, чтобы повторно реализовать на современном языке! Поддержка модульного программирования новыми языками, такими как Python, позволяет избежать подобных ситуаций, ведя разработку отдельных наборов функций в независимых файлах.

Способность совместно использовать функции из разных файлов существенно дополняет нашу модель программирования двумя разными способами. Во-первых, позволяет многократно использовать код без необходимости поддерживать несколько их копий. Во-вторых, это позволяет организовать программу в файлы управляемого размера, которые можно независимо проверять и отлаживать, строго соблюдая наше основное правило: *всякий раз, когда можете четко разделить задачи в пределах вычисления, так и поступайте.*

В этом разделе мы дополнили библиотеки `std*` из раздела 1.5 несколькими другими библиотеками, которые вы можете использовать: `gaussian`, `stdarray`, `stdrandom` и `stdstats`. Кроме того, их использование было проиллюстрировано на примерах нескольких клиентских программ. Эти инструментальные средства сосредоточены на базовых математических концепциях, применимых в любом научном или техническом проекте. Наше намерение не заключалось в предоставлении реальных инструментов, а только в иллюстрации того, насколько просто самому создать собственные инструментальные средства. Первый вопрос, задаваемый современными программистами при решении сложной задачи: какие инструменты мне нужны? Если необходимых инструментов нет, возникает второй вопрос: насколько трудно реализовать их? Хороший программист должен точно знать, когда нужно создавать собственный программный инструмент, а когда имеет смысл искать решение в существующем модуле.

Для обладания полной современной моделью программирования, кроме модулей и модульного программирования, остается еще один этап: *объектно-ориентированное программирование*, рассматриваемое в главе 3. При объектно-ориентированном программировании вы можете создавать модули функций, использующие побочные эффекты (жестко контролируемым способом) для значительного расширения программной модели Python. Прежде чем переходить к объектно-ориентированному программированию, рассмотрим развитие идеи, согласно которой любая функция вполне может вызвать сама себя (раздел 2.3), а также глубже, чем небольшие клиенты, в этом разделе проанализируем проблемы модульного программирования (раздел 2.4).

Вопросы и ответы

Как сделать такой модуль, как `gaussian` или `stdrandom`, доступным для моих программ Python?

Проще всего загрузить файл `gaussian.py` с сайта книги и поместить его в тот же каталог, что и клиент. Но в результате может получиться множество копий файла `gaussian.py` в разных каталогах, что затруднит поддержку кода. Вместо этого можно поместить файл `gaussian.py` в один каталог и включить его в список переменной окружения `PYTHONPATH`. Сайт книги содержит инструкции для установки переменной `PYTHONPATH` в вашей операционной системе. Если вы следовали поэтапным инструкциям установки Python с сайта книги, то все наши стандартные модули (включая `stdio`, `stddraw`, `stdarray`, `stdrandom` и `stdstats`) будут уже доступны для использования в ваших программах Python.

Я попытался импортировать модуль `gaussian`, но получил следующее сообщение об ошибке. Что не так?

```
ImportError: No module named gaussian
```

Вы не сделали модуль `gaussian` доступным для Python, как описано выше.

Я попытался вызвать функцию `gaussian.pdf()`, но получил следующее сообщение об ошибке. Что не так?

```
NameError: name 'gaussian' is not defined
```

Вы пропустили оператор `import gaussian`.

Есть ли ключевое слово, идентифицирующее файл `.py` как модуль (а не сценарий)?

Нет. Но чисто технически ключевым моментом модуля является отсутствие произвольного глобального кода, как описано ранее в этом разделе. Если в файле `.py` нет произвольного глобального кода, то он может быть импортирован в некий другой файл `.py`, а следовательно, это модуль. Существенная разница есть только с концептуальной точки зрения, поскольку одно дело создать файл `.py`, запускаемый немедленно (а возможно, и когда-нибудь позже с другими данными), другое дело создать файл `.py`, на который придется полагаться в будущем, и совсем другое дело создать файл `.py`, который будет использоваться в будущем не только вами.

Как разрабатывать новую версию модуля, который я уже использую?

Осторожно. Каждое изменение в API может нарушить любую клиентскую программу, поэтому работать имеет смысл в отдельном каталоге и только с копией кода. Если вы изменяете модуль, обладающий множеством клиентов, то можете оценить проблемы, стоящие перед компаниями, выпускающими новые версии своего программного обеспечения. Если необходимо лишь добавить в модуль несколько функций, то обычно это не слишком опасно.

Как узнать, что реализация ведет себя правильно? Почему нет автоматической проверки соответствия API?

Мы используем неформальные спецификации, поскольку создание детальной спецификации не очень сильно отличается по сложности от создания программы. Кроме того, фундаментальный принцип теоретической информатики гласит, что даже ее соблюдение не решает основных проблем, поскольку вообще нет никакого способа гарантировать, что две разные программы будут выполнять одинаковые вычисления.

Я заметил, что при запуске программ из этого раздела создаются файлы с расширением `.pyc`**. Например, когда я ввожу команду** `python gaussiantable.py`**, Python автоматически создает файл** `gaussian.pyc`**. Что это за файлы?**

Как уже упоминалось в разделе 1.1, всякий раз, когда Python выполняет программу, он компилирует ее во внутренний нечитаемый человеком формат — *код виртуальной машины* (bytecode). При первом импорте модуля Python компилирует код и сохраняет полученный код виртуальной машины в файле `.pyc`. Это ускоряет загрузку модуля, поскольку его не нужно компилировать повторно каждый раз (выполнение программы от этого быстрее не становится). Эти файлы `.pyc` можно удалить в любой момент; Python восстановит их, когда понадобится. Эти файлы `.pyc` можно и *не* удалять, поскольку после редактирования файла `.py` Python автоматически создаст соответствующий файл `.pyc`.

Упражнения

2.2.1. Составьте модуль, реализующий гиперболические тригонометрические функции на основании определений $\sin h(x) = (e^x - e^{-x})/2$ и $\cos h(x) = (e^x + e^{-x})/2$, при $\tan h(x)$, $\cot h(x)$, $\sec h(x)$ и $\csc$ определенных аналогичным способом стандартными тригонометрическими функциями.

2.2.2. Составьте клиент проверки для модулей `stdstats` и `stdrandom`, проверяющий все функции обоих модулей (за исключением функции `shuffle()`; см. упр. 1.4.22). Получите из командной строки аргумент `n`, создайте `n` случайных чисел, используя каждую из функций модуля `stdrandom`, и выведите их статистику. *Дополнительное задание*: сравните полученные результаты с результатом, ожидаемым из математического анализа.

2.2.3. Разработайте клиент стресс-проверки для модуля `stdrandom`. Обратите особое внимание на функцию `discrete()`. Например, действительно ли вероятности являются неотрицательными? Или все нули?

2.2.4. Составьте функцию, которая получает аргументы `ymin`, `ymax` (`ymin` строго меньше `ymax`) и массив `a[]` типа `float`, а затем линейно масштабирует элементы массива `a[]` так, чтобы все они оказались между `ymin` и `ymax`.

2.2.5. Составьте клиенты для модулей `gaussian` и `stdstats`, исследующие влияние изменения значений среднего и среднеквадратичного отклонения кривой Гауссова распределения. Создайте один график кривых с фиксированным средним и разными среднеквадратичными отклонениями и другой график кривых с фиксированным среднеквадратичным отклонением и разными средними значениями.

2.2.6. Добавьте в модуль `stdrandom` функцию `maxwellBoltzmann()`, возвращающую случайное значение, полученное из распределения Максвелла–Больцмана, с параметром σ. Для создания такого значения возвратите квадратный корень суммы квадратов трех Гауссовых случайных переменных со средним 0 и среднеквадратичным отклонением σ. (Распределение Максвелла–Больцмана описывает скорости молекул в идеальном газе.)

2.2.7. Модифицируйте программу `bernoulli.py`, анимировав гистограмму, — перерисовывайте ее после каждого эксперимента, чтобы можно было отследить ее схождение к нормальному распределению.

2.2.8. Модифицируйте программу `bernoulli.py` так, чтобы она получала дополнительный аргумент командной строки `p`, определяющий вероятность выпадения орлов при броске монеты. Проведите эксперименты, чтобы получить представление о распределении, соответствующем броскам монеты. Опробуйте значения p ближе к 0 и ближе к 1.

2.2.9. Составьте модуль `matrix.py`, реализующий следующие API для векторов и матриц (см. раздел 1.4).

API для модуля `matrix`

Вызов функции	Описание
`rand(m, n)`	Матрица `m` на `n` случайных чисел типа `float` от 0 до 1
`identity(n)`	Единичная матрица `n` на `n`
`dot(v1, v2)`	Скалярное произведение векторов `v1` и `v2`
`transpose(m)`	Транспозиция матрицы `m`
`add(m1, m2)`	Сумма матриц `m1` и `m2`
`subtract(m1, m2)`	Разница матриц `m1` и `m2`
`multiplyMM(m1, m2)`	Произведение матриц `m1` и `m2`
`multiplyMV(m, v)`	Матрично-векторное произведение матрицы `m` и вектора `v` (вектор)
`multiplyVM(v, m)`	Векторно-матричное произведение вектора `v` и матрицы `m` (вектор)

В качестве клиента проверки используйте следующий код, выполняющий то же вычисление, что и программа 1.6.3 (`markov.py`):

```
moves = int(sys.argv[1])
p = stdarray.readFloat2D()
ranks = stdarray.create1D(len(p), 0.0)
ranks[0] = 1.0
for i in range(moves):
    ranks = matrix.multiplyVM(ranks, p)
stdarray.write1D(ranks)
```

На практике для таких задач ученые и математики используют готовые библиотеки, такие как `NumPy` (или специальные языки обработки матриц, такие как Matlab), поскольку они, вполне вероятно, эффективней, точней и надежней самодельных. Более подробная информация об использовании библиотеки `NumPy` приведена на сайте книги.

2.2.10. Составьте клиент модуля `matrix.py`, реализующий версию программы `markov.py`, описанной в разделе 1.6, но на сей раз на базе квадратов матриц, а не последовательности векторно-матричного умножения.

2.2.11. Переделайте программу 1.6.2 (`randomsurfer.py`), используя модули `stdarray` и `stdrandom`.

Частичное решение:

```
...
p = stdarray.readFloat2D()
page = 0 # Начало на стр. 0.
```

```
hits = stdarray.create1D(n, 0)
for i in range(moves):
    page = stdrandom.discrete(p[page])
    hits[page] += 1
```

2.2.12. Добавьте в программу `stdrandom.py` функцию `exp()`, получающую аргумент λ и возвращающую случайное число из *экспоненциального распределения* (exponential distribution) с коэффициентом λ: если x — случайное число однородного распределения от 0 до 1, то $-\ln x / \lambda$ — это случайное число экспоненциального распределения с коэффициентом λ.

2.2.13. Реализуйте в программе `stdrandom.py` функцию `stdrandom.py`, получающую как аргумент массив и перетасовывающую ее элементы. Используйте алгоритм перетасовки, описанный в разделе 1.4.

Практические упражнения

2.2.14. *Игральная кость Зикермана.* Предположим, у вас есть две шестигранных игральных кости: одна с метками 1, 3, 4, 5, 6 и 8, вторая с метками 1, 2, 2, 3, 3 и 4. Сравните вероятности выпадения каждой суммы из значений этих игральных костей с таковыми для стандартной пары игральных костей. Используйте модули `stdrandom` и `stdstats`.

2.2.15. *Кости.* Ниже приведены правила игры в *кости*. Пусть x будет суммой значений двух шестигранных игральных костей после броска.

- Если при первом броске выпало в сумме 7 или 11, вы победили.
- Если при первом броске выпало в сумме 2, 3 или 12, вы проиграли.
- Если не выпало ни одно из этих значений, броски продолжаются, пока не выпадет 7 или 4, 5, 6, 8, 9, 12.
- Если выпало в сумме 4, 5, 6, 8, 9 или 12, вы выиграли.
- Если выпало в сумме 7, вы проиграли.

Составьте модульную программу, оценивающую вероятность выигрыша. Модифицируйте свою программу так, чтобы получать вероятность выпадения 1 из командной строки и чтобы вероятность выпадения 6 вычислялась как 1/6 минус вероятность единицы, а 2–5 имели одинаковую вероятность. *Подсказка*: используйте функцию `stdrandom.discrete()`.

2.2.16. *Динамическая гистограмма.* Предположим, что поток стандартного ввода — это последовательность вещественных чисел. Составьте программу, которая получает из командной строки целое число `n` и два вещественных

числа `l` и `r`, а затем использует модуль `stdstats` для рисования гистограммы количества чисел в потоке стандартного ввода, попадающих в каждый из `n` одинаковых интервалов, получаемых делением диапазона (`l`, `r`) на `n`. Используйте эту программу для пополнения кода решения упражнения 2.2.2 для рисования гистограммы распределения чисел, созданных каждой функцией, получая `n` из командной строки.

2.2.17. *График Тьюки* — это обобщенная гистограмма для визуализации данных; он применим в случае, когда каждое целое число в заданном диапазоне связано с несколькими значениями по оси *y*. Для каждого целого числа `i` в диапазоне мы вычисляем среднее значение, среднеквадратичное отклонение, а также по 10 и 90 процентов всех связанных значений по оси *y*; затем рисуется вертикальная линия со значением `i` по координате *x* длиной от 10-го до 90-го процента по координате *y*; затем рисуется тонкий прямоугольник с центром на линии, начинающийся от одного среднеквадратичного отклонения ниже среднего значения к среднеквадратичному отклонению выше среднего. Предположим, что поток стандартного ввода содержит последовательность пар чисел, где первое число целое, а второе вещественное. Составьте клиенты модулей `stdstats` и `stddraw`, получающие из командной строки целое число `n`, а из входного потока целые числа в диапазоне от `0` до `n-1`, а затем использующие стандартное графическое устройство для вывода графика Тьюки по этим данным.

2.2.18. *IFS.* Поэкспериментируйте с различными входными данными программы `ifs.py` и создайте узоры по собственному проекту, подобные треугольнику Серпинского, папоротнику Барнсли и др. Эксперименты можно начать с незначительных модификаций входных данных.

2.2.19. *Реализация матрицы IFS.* Составьте версию программы `ifs.py`, использующую функцию `matrix.multiplyMV()` (см. упр. 2.2.9) вместо уравнений, вычисляющих новые значения `x` и `y`.

2.2.20. *Стресс-проверка.* Составьте клиент, осуществляющий стресс-проверку модуля `stdstats`. Поработайте с напарником: один составляет код, другой проверяет его.

2.2.21. *Разорение игрока.* Составьте клиент модуля `stdrandom` для изучения задачи разорения игрока (см. программу 1.3.8 и упражнение 1.3.23).

2.2.22. *Модуль для свойств целых чисел.* Составьте модуль на основании функций вычисления свойств целых чисел, описанных в этой книге. Включите функции выяснения принадлежности заданного целого числа к простым; взаимной простоты двух целых чисел; выявления всех множителей данного

целого числа; выявления наибольшего общего делителя и наименьшего общего множителя двух целых чисел. Включите также функцию Эйлера (см. упр. 2.1.23) и любые другие функции, которые, по-вашему, могли бы быть полезны. Создайте API, клиент стресс-проверки и клиенты, решающие некоторые из упражнений, приведенных ранее в этой книге.

2.2.23. *Машина для подсчета голосов.* Составьте клиент модуля `stdrandom` (с соответствующими собственными функциями) для изучения следующей задачи: предположим, есть 100 миллионов избирателей, 51% голосует за кандидата A, а 49% — за кандидата B. Однако машины для подсчета голосов склонны к ошибкам: в 5% случаев они дают неправильный ответ. Если ошибки происходят независимо и случайно, достаточно ли 5%-го коэффициента ошибок для объявления неверных результатов выборов между кандидатами с почти равными шансами? Какой коэффициент ошибок может быть допустим?

2.2.24. *Анализ сдачи в покере.* Составьте клиент модулей `stdrandom` и `stdstats` (с соответствующими собственными функциями) для оценки вероятности получить одну пару, две пары, три одинаковых, фул-хаус и флеш при сдаче пяти карт. Разделите программу на соответствующие функции и реализуйте свои решения. *Дополнительное задание*: добавьте в список возможностей флеш-рояль и стрит.

2.2.25. *Модуль музыки.* На основании функции в программе 2.1.4 (`playthattune-deluxe.py`) разработайте модуль для использования клиентскими программами, создающими и манипулирующими музыкой.

2.2.26. *Анимированные графики.* Составьте программу, получающую аргумент командной строки `m` и рисующую гистограмму для `m` последних вещественных чисел, поступивших со стандартного ввода. Используйте ту же методику анимации, что и для программы 1.5.7 (`bouncingball.py`): многократно вызывайте функцию `clear()`, рисуйте график и вызывайте функцию `show()`. Каждый раз, когда программа читает новое число, она должна перерисовать весь график. Поскольку большая часть изображения не изменяется, при перерисовке оно смещается немного влево. Программа произведет впечатление окна фиксированного размера, динамически скользящего по входным значениям. Используйте свою программу для рисования огромного изменяющегося во времени файла данных, таких как курсы акций.

2.2.27. *Модуль рисования массива.* Разработайте собственные графические функции, улучшающие таковые в модуле `stdstats`. Проявите творческий подход! Попробуйте сделать модуль рисования полезным для будущих приложений.

2.3. Рекурсия

Способность одной функции вызывать другую сразу наводит на мысль о возможности функции вызвать *себя саму*. Механизм вызова функции в Python и в большинстве современных языков программирования обеспечивает эту возможность, известную как *рекурсия* (recursion). В этом разделе мы изучим примеры изящных и эффективных рекурсивных решений множества задач. Освоившись с идеей, вы увидите, что рекурсия — это мощнейшая универсальная методика программирования со многими привлекательными свойствами. Этот фундаментальный инструмент довольно часто используется в данной книге. Рекурсивные программы зачастую компактней и проще своих не рекурсивных аналогов. Некоторые программисты находят рекурсию достаточно удобной, чтобы использовать ее в коде регулярно, однако изящное решение задачи при помощи рекурсии является обычной практикой, доступной для каждого программиста.

Программы этого раздела...

Программа 2.3.1. Алгоритм Евклида (`euclid.py`) 290
Программа 2.3.2. Ханойская башня (`towersofhanoi.py`) 293
Программа 2.3.3. Код Грея (`beckett.py`) 299
Программа 2.3.4. Рекурсивная графика (`htree.py`) 301
Программа 2.3.5. Броуновский мост (`brownian.py`) 303

Рекурсия — это намного больше, чем методика программирования. Во многих случаях это естественный способ описания реального мира. Например, рекурсивное дерево (приведено ниже) напоминает реальное дерево и имеет естественное рекурсивное описание. Многие явления хорошо описываются рекурсивными моделями. В частности, рекурсия играет центральную роль в информатике. Она представляет простую вычислительную модель, охватывающую все, что может быть вычислено любым компьютером; это помогает организовать и анализировать программы; это также ключ к многочисленным критически важным вычислительным приложениям от комбинаторного поиска и древовидных структур данных, обеспечивающих обработку информации, до быстрого преобразования Фурье при обработке сигналов.

Рекурсивная модель реального мира

Одной из важнейших причин рассмотрения рекурсии является ее способность простым и понятным способом создавать математические модели, применимые для доказательства важных фактов о наших программах. Используемая методика доказательства известна как *математическая индукция* (mathematical induction). Вообще-то, мы избегаем подробностей математических доказательств в этой книге, но в данном разделе вы увидите, что предпринятые усилия по изучению рекурсивных программ будут потрачены не зря.

Рекурсия изображения

Первая рекурсивная программа. Первой рекурсивной программой, реализуемой большинством программистов (типа “Hello, World”), является функция *факториала* (factorial), определенная для n положительных целых чисел уравнением

$$n! = n \times (n-1) \times (n-2) \times ... \times 2 \times 1$$

Другими словами, n! — это произведение положительных целых чисел, меньших или равных n. Пора вычислить *n*! Довольно просто сделать это при помощи цикла for, но еще проще, используя следующую рекурсивную функцию:

```
def factorial(n)
    if n == 1: return 1
    return n * factorial(n-1)
```

Эти вызовы функцией самой себя окажут желаемый эффект. Можете убедиться сами, что обращение к функции `factorial()` возвращает 1, когда `n` равно 1, а если нет, то она вычисляет значение

$$(n-1)! = (n-1) \times (n-2) \times \ldots \times 2 \times 1,$$

а затем значение

$$n! = n \times (n-1)! = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1$$

Чтобы вычислить `factorial(5)`, рекурсивная функция должна вычислить `factorial(4)`; чтобы вычислить `factorial(4)`, она должна вычислить `factorial(3)`, и т.д. Этот процесс повторяется до вызова `factorial(1)`, который непосредственно возвращает значение `1`. Затем `factorial(2)` умножает это значение на `2` и возвращает `2`; `factorial(3)` умножает это значение на `3` и возвращает `6` и т.д. Это вычисление можно трассировать точно так же, как любую другую последовательность вызовов функций. Поскольку мы просматриваем все вызовы как независимые экземпляры кода, тот факт, что они рекурсивны, несущественен.

```
1 1
2 2
3 6
4 24
5 120
6 720
7 5040
8 40320
9 362880
10 3628800
11 39916800
12 479001600
13 6227020800
14 87178291200
15 1307674368000
16 20922789888000
17 355687428096000
18 6402373705728000
19 121645100408832000
20 2432902008176640000
```

Значения n!

```
factorial(5)
    factorial(4)
        factorial(3)
            factorial(2)
            factorial(1)
                return 1
            return 2*1 = 2
        return 3*2 = 6
    return 4*6 = 24
return 5*24 = 120
```

Трассировка вызовов функции `factorial()`

Наша реализация функции `factorial()` демонстрирует два основных компонента, присущих каждой рекурсивной функции. *Конечный случай* (base case) возвращает значение без последующих рекурсивных вызовов. Он обусловлен одним или несколькими специальными аргументами, при которых функция вычисляется без

рекурсии. Для функции `factorial()` конечный случай — это $n = 1$. *Этап редукции*[1] (reduction step) — центральная часть рекурсивной функции. Он связывает один (или несколько) аргументов функции с вычислением одного (или нескольких других) аргумента. Для функции `factorial()` этап редукции — `n * factorial(n-1)`. У всех рекурсивных функций должны быть эти два компонента. Кроме того, последовательность значений аргумента должна сходиться к конечному случаю. У функции `factorial()` значение `n` уменьшается на 1 при каждом вызове, поэтому последовательность значений аргумента сходится к конечному случаю ($n = 1$).

Небольшие функции, такие как `factorial()`, становятся немного понятней, если поместить этап редукции в директиву `else`. Однако принятие этого соглашения излишне усложнило бы большие функции, поскольку бо́льшая часть их кода (этапа редукции) оказалась бы с отступом в блоке `else`. Вместо этого используется другое соглашение: конечный случай помещается в первый оператор, завершающийся оператором `return`, а остальная часть кода посвящается этапу редукции.

Та же методика эффективна при определении функций любого рода. Например, рекурсивная функция

```
def harmonic(n):
    if n == 1: return 1.0
    return harmonic(n-1) + 1.0/n
```

эффективно вычисляет гармонические числа (см. программу 1.3.5) при небольших `n` на основании следующего уравнения:

$$H_n = 1 + 1/2 + ... + 1/n = (1 + 1/2 + ... + 1/(n-1)) + 1/n = H_{n-1} + 1/n$$

На самом деле этот подход эффективен для вычисления значений *любой* дискретной суммы, для которой есть компактная формула, задействующая всего несколько строк кода. Подобные рекурсивные программы — это скрытые циклы, но рекурсия способна лучше выполнять вычисления такого типа.

Математическая индукция. Рекурсивное программирование непосредственно связано с *математической индукцией* (mathematical induction) — методикой доказательства фактов о дискретных функциях.

То, что выражение задействует целое число `n`, является истиной для бесконечно многих значений `n`. Для доказательства этого математическая индукция использует следующие два этапа.

- *Конечный случай.* Докажите истинность выражения для небольшого количества конкретных значений `n` (обычно 1).
- *Этап индукции* (центральная часть доказательства). Подразумевая, что выражение истинно для всех положительных целых чисел меньше `n`, используем этот факт для доказательства истинности для `n`.

[1] Редукция — логико-методологический прием сведения сложного к простому. В данном случае переход на следующий уровень рекурсии. — *Примеч. ред.*

Такого доказательства достаточно для демонстрации того, что выражение истинно для *всех* положительных целых чисел n: можно начать с конечного случая и использовать наше доказательство для утверждения того, что выражение истинно для каждого большего значения n, одно за другим.

Всеобщее, первое доказательство индукции должно продемонстрировать, что сумма положительных целых чисел, меньших или равных n, вычисляется по формуле $n(n+1)/2$. Таким образом, необходимо доказать, что следующее уравнение истинно для всех $n \geq 1$:

$$1 + 2 + 3 \ldots + (n-1) + n = n\,(n+1)\,/\,2$$

Уравнение, безусловно, истинно для $n = 1$ (конечный случай), поскольку $1 = 1$. Если подразумевать, что оно истинно для всех целых чисел, меньших n, то, в частности, это истинно для $n - 1$, таким образом,

$$1 + 2 + 3 \ldots + (n-1) = (n-1)\,n\,/\,2,$$

и n можно добавить с обеих сторон этого уравнения и упростить его так, чтобы получить желаемое уравнение (этап индукции).

Каждый раз, составляя рекурсивную функцию, необходима математическая индукция, гарантирующая, что функция имеет желаемый эффект. Соответствие между индукцией и рекурсией самоочевидно. Различие в названии связано с различием цели: цель рекурсивной функции — осуществить вычисление, сведя его к меньшей задаче, поэтому мы используем термин *этап редукции* (reduction step); в доказательстве индукции наша цель заключается в установлении истинности выражения для больших задач, поэтому мы используем термин *этап индукции* (induction step).

Составляя рекурсивные функции, мы обычно не записываем полное формальное доказательство того, что они приводят к желаемому результату, но мы всегда зависим от существования такого доказательства. Мы действительно часто обращаемся к неформальному доказательству индукции для гарантии правильности работы рекурсивной функции. Например, мы только что обсудили неформальное доказательство того, что функция `factorial(n)` вычисляет произведение положительных целых чисел, меньших или равных `n`.

Алгоритм Евклида. *Наибольший общий делитель* (НОД) двух положительных целых чисел — это наибольшее целое число, делящее нацело их обоих. Например, наибольший общий делитель чисел 102 и 68 — это 34, поскольку и 102, и 68 делятся на 34, но никакое другое целое число больше 34 не делит нацело ни 102, ни 68. Наибольшие общие делители вы, вероятно, изучали одновременно с правилами сокращения дробей. Например, мы можем сократить 68/102 до 2/3, разделив и числитель, и знаменатель на 34, т.е. на их НОД. Поиск НОД больших целых чисел является важной задачей, встречающейся во многих коммерческих приложениях, включая известную криптографическую систему RSA.

Программа 2.3.1. Алгоритм Евклида *(euclid.py)*

```
import sys
import stdio

def gcd(p, q):
    if q == 0: return p
    return gcd(q, p % q)

def main():
    p = int(sys.argv[1])
    q = int(sys.argv[2])
    stdio.writeln(gcd(p, q))

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

Этот сценарий получает из командной строки два положительных целых числа и вычисляет их наибольший общий делитель, используя рекурсивную реализацию алгоритма Евклида:

```
% python euclid.py 1440 408
24
% python euclid.py 314159
271828
1
```

Для эффективного вычисления НОД можно использовать следующее свойство положительных целых чисел p и q:

если $p>q$, то их НОД тот же, что и НОД q и $p\%q$.

Чтобы убедиться в этом факте, обратите внимание на то, что НОД p и q тот же, что и НОД q и $p - q$, поскольку если число делит и p, и q, то оно делит $p - q$. Согласно той же логике, тот же НОД будет и у q и $p-2q$, и у q и $p - 3q$ и т.д.; один из способов вычисления $p\%q$ подразумевает вычитание q из p, пока не получится число меньше, чем q.

Функция `gcd()` в программе 2.3.1 (`euclid.py`) является компактной рекурсивной функцией, этап редукции которой основан на этом свойстве. Конечный случай наступает, когда `q` равно `0`, поскольку `gcd(p, 0) = p`. Чтобы этап редукции сходился к конечному случаю, значение второго аргумента уменьшается при каждом рекурсивном вызове до `p % q < q`. Если `p < q`, то первый рекурсивный вызов включает два аргумента. Фактически значение второго аргумента уменьшается по крайней мере на множитель числа 2 для каждого второго рекурсивного вызова, поэтому значение аргумента довольно быстро сходится к конечному случаю (см. упр. 2.3.13). Это рекурсивное решение задачи вычисления

наибольшего общего делителя известно как *алгоритм Евклида* и является одним из самых старых известных алгоритмов — ему более 2 000 лет.

```
gcd(1440, 408)
    gcd(408, 216)
        gcd(216, 24)
            gcd(192, 24)
            gcd(24, 0)
                return 24
            return 24
        return 24
    return 24
return 24
```

Трассировка вызова функции `gcd()`

Ханойская башня. Никакое обсуждение рекурсии не обходится без известной задачи *Ханойской башни*: имеется три стержня и n дисков на первом из них. Диски отличаются по размеру и располагаются на стержне от наибольшего снизу (диск n) до наименьшего сверху (диск 1). Задача заключается в том, чтобы переместить все n дисков на другой стержень, соблюдая следующие правила.

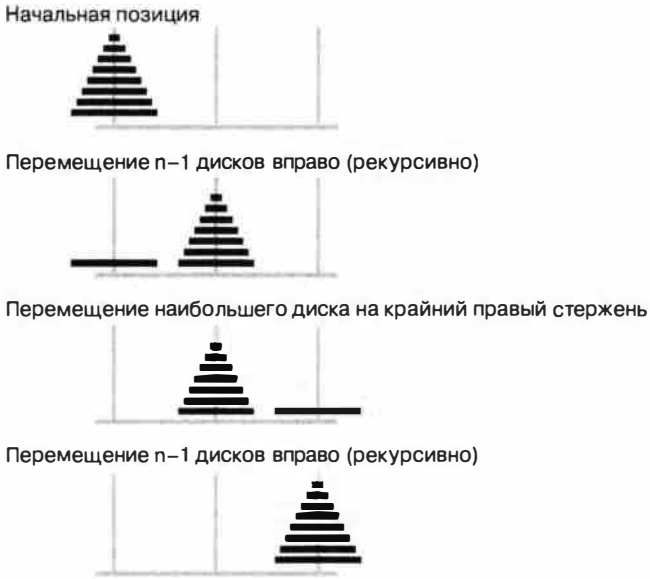
- Переносится только один диск за раз.
- Больший диск нельзя поместить на меньший.

Эту головоломку придумал в 1883 году французский математик Эдуард Люка. По его легенде в самом начале времен в Великом храме города Бенарес, который является серединой мира, Брахма воздвиг бронзовый диск с 3 алмазными стержнями высотой в один локоть и толщиной с пчелу. На один из них он возложил 64 золотых диска, лежащих меньший на большем. Монахи храма должны были переложить диски на другой стержень. Как только они выполнят задачу, мир прекратит существование.

Но могут ли монахи выполнить задачу вообще, играя по правилам?

Для решения задачи необходимо выработать последовательность инструкций по перемещению дисков. Подразумевается, что стержни расположены в ряд и каждая инструкция по перемещению дисков определяет его номер и направление перемещения (влево или вправо). Если диск находится на левом стержне, инструкция по перемещению влево означает перенос вокруг на крайний правый стержень; если диск находится на правом стержне, инструкция по перемещению вправо означает перенос вокруг на крайний левый стержень. Когда все диски находятся на одном стержне, возможны два хода (переместить наименьший диск влево или вправо); в противном случае есть три возможных хода (переместить наименьший диск влево или вправо либо сделать один из допустимых ходов, задействуя два других стержня). Выбор между этими возможностями на каждом этапе решения задачи и является главной сложностью плана. Рекурсия обеспечивает реализацию плана на основании

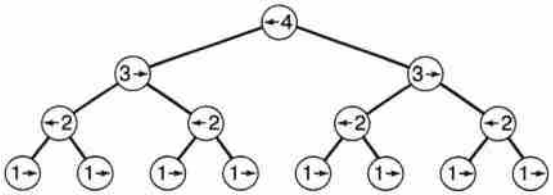
следующей идеи: сначала перемещаем вершину (n–1-й диск) на пустой стержень, затем перемещаем больший диск на другой пустой стержень (не с меньшим диском) и завершаем работу, перенося n–1-й диск на больший. Для упрощения инструкций будем перемещать диски влево или вправо с переносом вокруг, т.е. перемещение с крайнего левого стержня влево означает перенос на крайний правый, а перенос вправо с крайнего правого — перенос на крайний левый стержень.



Рекурсивный план решения задачи Ханойской башни

Программа 2.3.2 (`towersofhanoi.py`) — это прямая реализация данной стратегии. Она получает в аргументе командной строки число `n` и выводит решение задачи Ханойской башни для `n` дисков. Рекурсивная функция `moves()` выводит последовательность переходов для перемещения стопки дисков влево (если аргумент `left` имеет значение `True`) или вправо (если `left` — `False`), с переносом вокруг. Все это происходит согласно плану, описанному выше.

Деревья вызова функции. Чтобы лучше понять поведение модульных программ, обладающих несколькими рекурсивными вызовами (таких, как `towersofhanoi.py`), используют визуальное представление — *дерево вызова функции* (function-call tree). Каждый вызов функции представляется как *узел древа* (tree node), изображаемый в виде круга, помеченного значениями аргументов этого вызова. Под каждым узлом располагаются следующие узлы, соответствующие каждому следующему вызову функции (в порядке слева направо) и соединяющие их линии. Эта схема содержит всю информацию, необходимую для понимания поведения программы, — для каждого вызова функции представлен свой узел.



Дерево вызовов функции `moves(4, true)`
в программе `towersofhanoi.py`

Программа 2.3.2. Ханойская башня (`towersofhanoi.py`)

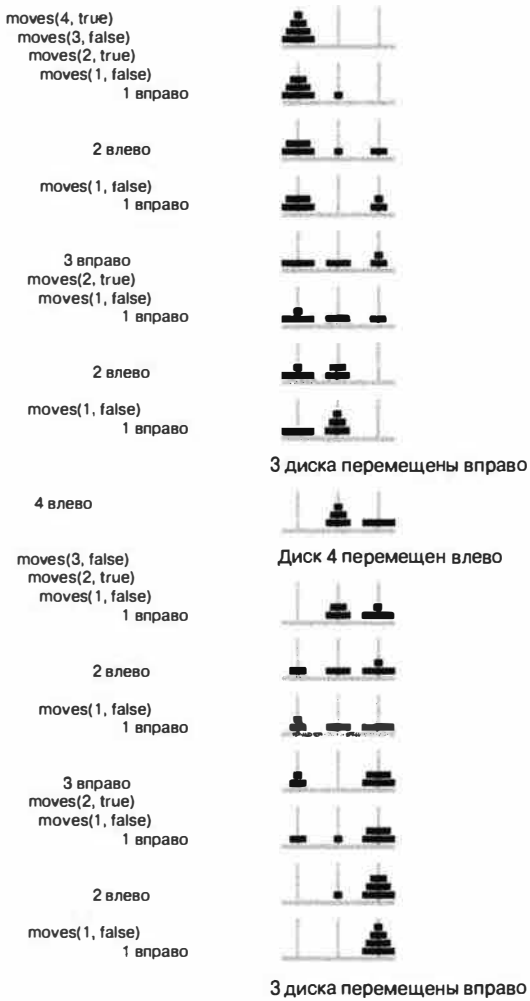
```
import sys
import stdio

def moves(n, left):
    if n == 0: return
    moves(n-1, not left)
    if left:
        stdio.writeln(str(n) + ' left')
    else:
        stdio.writeln(str(n) + ' right')
    moves(n-1, not left)

n = int(sys.argv[1])
moves(n, True)
```

`n`	Количество дисков
`left`	Направление перемещения

Этот сценарий выводит инструкции по решению задачи Ханойской башни. Рекурсивная функция `moves()` выводит инструкции по перемещению `n` дисков влево (если аргумент `left` имеет значение `True`) или вправо (если `left` — `False`).

```
% python towersofhanoi.py 1
1 left

% python towersofhanoi.py 2
1 right
2 left
1 right

% python towersofhanoi.py 3
1 left
2 right
1 left
3 left
1 left
2 right
1 left
```

```
% python towersofhanoi.py 4
1 right
2 left
1 right
3 right
1 right
2 left
1 right
4 left
1 right
2 left
1 right
3 right
1 right
2 left
1 right
```



Трассировка вызовов функции `moves(4, true)`

Деревья вызова функции можно использовать для понимания поведения любой модульной программы, но они особенно полезны для демонстрации поведения рекурсивных программ. Например, создать дерево вызовов функции `moves()` в программе `towersofhanoi.py` вовсе не сложно. Начнем с узла древа, помеченного значениями аргументов командной строки. Первый аргумент — количество дисков в пирамиде (и метка фактически перемещаемого диска); второй — направление перемещения пирамиды. Для ясности мы изображаем направление (логическое значение) как стрелку, указывающую влево или вправо, поскольку это наша интерпретация значения направления перемещения диска. Затем нарисуем два узла древа ниже, с количеством дисков, уменьшенным на 1, и измененным направлением. Продолжаем процесс до тех пор, пока не останутся узлы только с метками, соответствующими значению 1 первого аргумента, ниже них никаких узлов нет. Эти узлы соответствуют обращениям к функции `moves()`, не ведущим к дальнейшим рекурсивным вызовам.

Уделите немного времени изучению дерева вызовов функции, приведенного ранее в этом разделе, и сравните его с соответствующей трассировкой вызовов функции, приведенной ниже. Сделав это, вы увидите, что дерево вызовов функции — это только компактное представление трассировки. В частности, чтение меток узлов слева направо дает переходы, решающие задачу.

Кроме того, при изучении дерева можно заметить несколько шаблонов, включая следующие два.

- Альтернативные переходы задействуют наименьший диск.
- Этот диск всегда двигается в том же направлении.

Эти наблюдения вполне уместны, поскольку они дают решение задачи, не требующее рекурсии (или даже компьютера): любой переход задействует наименьший диск (включая первый и последний) и каждый предыдущий переход — единственный допустимый в момент, когда наименьший диск не задействован. Мы можем *доказать*, что эта стратегия дает тот же результат, что и рекурсивная программа, использующая индукцию. Начав несколько тысячелетий назад (без преимуществ компьютера), наши монахи, возможно, используют эту стратегию.

Деревья уместны и важны для понимания рекурсии, поскольку дерево — это наиболее существенный рекурсивный объект. Как абстрактная математическая модель, деревья играют основную роль во многих приложениях, а в главе 4 мы увидим, что использование деревьев как вычислительной модели позволяет структурировать данные для более эффективной обработки.

Экспоненциальное время. Одним из преимуществ использования рекурсии является возможность разработки математических моделей, позволяющих доказывать важные факты о поведении рекурсивных программ. Для задачи Ханойской башни мы можем оценить период времени до конца мира (если легенда истинна). Это упражнение важно не только потому, что оно доказывает, что конец мира еще весьма далек (даже если легенда истинна), но также и потому, что оно позволяет понять суть рекурсии и помочь нам избежать ситуаций, когда программы могут и не завершить работу примерно до тех же пор.

Математическая модель задачи Ханойской башни проста: если мы определяем функцию $T(n)$ для количества директив перехода, выдаваемых программой `towersofhanoi.py` для перемещения n дисков с одного стержня на другой, то рекурсивный код подразумевает, что функция $T(n)$ должна удовлетворить следующим уравнениям:

$$T(n) = 2\ T(n-1) + 1 \text{ для } n > 1, \text{ при } T(1) = 1.$$

В дискретной математике такое уравнение известно как *рекуррентное соотношение* (recurrence relation). Рекуррентные соотношения естественно возникают при исследовании рекурсивных программ. Их зачастую используют при получении замкнутых выражений. Для функции $T(n)$ вы, возможно, уже предположили по исходным значениям $T(1) = 1$, $T(2) = 3$, $T(3) = 7$ и $T(4) = 15$, что $T(n) = 2^n - 1$. Рекуррентное соотношение позволяет *доказать* это с использованием математической индукции.

- *Конечный случай*: $T(1) = 2^h - 1 = 1$.
- *Этап индукции*: если $T(h-1) = 2^{h-1} - 1$, $T(h) = 2(2^{h-1} - 1) + 1 = 2^h - 1$.

Следовательно, по индукции $T(h) = 2_h - 1$ для всех $h > 0$. Минимально возможное количество переходов также удовлетворяет тому же рекуррентному соотношению (см. упр. 2.3.11).

Экспоненциальный рост

Зная значение T(n), можно оценить период времени, необходимый для выполнения всех переходов. Если бы монахи перемещали по одному диску в секунду, потребовалась бы не одна неделя для завершения решения задачи с 20 дисками, для 30 дисков понадобилось бы больше 34 лет, а для 40 дисков — больше 348 *столетий* непрерывной работы (если не делать ошибок). Решение задачи с 64 дисками заняло бы 584 *миллиарда* лет[2]. Конец мира, вероятно, наступит раньше, поскольку у монахов не было преимуществ использования программы 2.3.2, и они были бы не в состоянии так быстро выяснять, какой диск куда перемещать.

Даже компьютеры не справляются с экспоненциальным ростом. Компьютеру, способному выполнять миллиард операций в секунду, все еще понадобятся столетия для осуществления 2^{64} операций, и никакому компьютеру никогда не выполнить $2^{1\,000}$ операций. Отсюда вывод: при рекурсии можно легко составить простую, короткую и вполне работоспособную программу, но с экспоненциальной продолжительностью выполнения. Для достаточно большого *n* она окажется просто не в состоянии завершить работу за приемлемый промежуток времени.

Новички нередко сомневаются в этом факте, поэтому имеет смысл остановиться на нем подробнее. Чтобы убедиться в его истинности, уберем из программы `towersofhanoi.py` вызовы функции `stdio.writeln()` и запустим ее сначала для *n*, равного 20. Вы можете легко убедиться, что каждое последующее увеличение значения *n* на 1 удваивает продолжительность выполнения, и вы быстро начнете терять терпение, ожидая его завершения. Если некоего значения *n* хватит на час работы, то для $n + 5$ понадобится уже больше дня, для $n + 10$ — больше месяца, а для $n + 20$ — больше столетия (ни у кого не хватит терпения *так* надолго). Мощности вашего компьютера вполне может не хватить для выполнения короткой программы Python, несмотря на ее кажущуюся простоту! *Остерегайтесь программ, способных потребовать экспоненциального времени выполнения.*

Нас часто интересует ориентировочная продолжительность выполнения наших программ. В разделе 4.1 мы обсудим использование только что рассмотренного процесса для оценки продолжительности выполнения других программ.

Коды Грея. Задача Ханойской башни — не только игрушка. Она тесно связана с фундаментальными алгоритмами манипулирования числами и дискретными объектами. В качестве примера рассмотрим *коды Грея* (Gray codes) — математическую абстракцию с многочисленными случаями применения.

[2] Продолжительность существования таких звезд, как Солнце, составляет 10–12 миллиардов лет. — *Примеч. ред.*

Драматург Сэмюель Беккет, известный как автор пьесы "В ожидании Годо", поставил спектакль "Quad", начинавшийся с пустой сцены, на которую по одному входят и выходят персонажи так, чтобы в каждый момент времени набор персонажей на сцене был уникальным. Как Беккет осуществлял постановку этого спектакля?

Представления кода Грея

Код	Подмножество	Переходы
0 0 0 0	Пусто	
0 0 0 1	1	enter 1
0 0 1 1	2 1	enter 2
0 0 1 0	2	exit 1
0 1 1 0	3 2	enter 3
0 1 1 1	3 2 1	enter 1
0 1 0 1	3 1	exit 2
0 1 0 0	3	exit 1
1 1 0 0	4 3	enter 4
1 1 0 1	4 3 1	enter 1
1 1 1 1	4 3 2 1	enter 2
1 1 1 0	4 3 2	exit 1
1 0 1 0	4 2	exit 3
1 0 1 1	4 2 1	enter 1
1 0 0 1	4 1	exit 2
1 0 0 0	4	exit 1

Один из способов представления подмножества из n дискретных объектов подразумевает использование строки из n битов. Для задачи Беккета мы используем 4-битовую строку, где биты пронумерованы от 1 до n справа налево, и значение 1 каждого бита означает наличие персонажа на сцене. Например, строка 0 1 0 1 соответствует нахождению на сцене персонажей 3 и 1. Это представление дает быстрое доказательство простого факта: *количество уникальных подмножеств из n объектов составляет* 2^n. В спектакле "Quad" четыре персонажа, в результате получается 2^4, или 16 различных сцен. Наша задача — создание постановки.

Код Грея для n бит — это список из 2^n разных двоичных чисел размером по n бит, где каждая запись отличается от предшествующей только одним битом. Коды Грея имеют непосредственное отношение к задаче Беккета, поскольку изменение значения бита с 0 на 1 соответствует выходу персонажа на сцену, а с 1 на 0 — уходу со сцены.

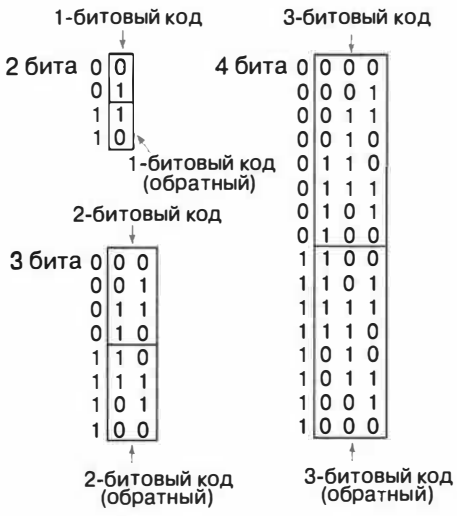
Как мы создаем код Грея? Подойдет рекурсивный план, очень похожий на использованный для задачи Ханойской башни. *Рефлексивный бинарный код Грея* (binary reflected Gray code) для n бит рекурсивно определяется следующим образом:

- (n–1)-битовый код с 0, предшествующем каждому кодовому слову, сопровождаемый
- (n–1)-битовым кодом *в обратном порядке*, с 1, предшествующей каждому кодовому слову.

Битовый код 0 определяется как нуль, таким образом, однобитовый код 0 сопровождается 1. Это рекурсивное определение позволяет проверить индукцией, что у n-битового рефлексивного бинарного кода Грея есть необходимое свойство: соседние кодовые слова отличаются одной позицией двоичного разряда. Это истинно в соответствии с индуктивной гипотезой, кроме, возможно, последнего кодового слова в первой половине и первого кодового слова во второй половине: эта пара различается только по первому биту.

Рекурсивные определения ведут, после некоторого осмысления, к реализации в программе 2.3.3 (`beckett.py`) решения задачи Беккета. Эта программа очень похожа на `towersofhanoi.py`. Действительно, за исключением спецификации, единственное различие находится в значениях вторых аргументов рекурсивных вызовов!

Подобно направлениям переноса дисков в программе `towersofhanoi.py`, выход и уход со сцены избыточны в программе `beckett.py`, поэтому слово `exit` выводится, когда актер уходит со сцены, а `enter` — когда выходит на сцену. Действительно, и программа `beckett.py`, и `towersofhanoi.py` непосредственно задействуют линейчатую функцию, которую мы рассматривали в одной из первых программ — в программе 1.2.1 (`ruler.py`). Без избыточных инструкций они обе реализуют простую рекурсивную функцию, применимую для вывода результатов линейчатой функции для любого значения, заданного аргументом командной строки.



2-, 3- и 4-битовые коды Грея

Коды Грея применяются довольно часто, от аналого-цифровых преобразователей до планирования экспериментов. Они использовались в импульсно-кодовой коммуникации, в минимизации логических каналов, в архитектуре гиперкуба и предлагались для организации книг на библиотечных полках.

Программа 2.3.3. Код Грея (beckett.py)

```
import sys
import stdio

def moves(n, enter):
    if n == 0: return
    moves(n-1, True)
    if enter:
        stdio.writeln('enter ' + str(n))
    else:
        stdio.writeln('exit ' + str(n))
    moves(n-1, False)

n = int(sys.argv[1])
moves(n, True)
```

n	Количество персонажей
enter	Движение по сцене

Этот сценарий читает из командной строки целое число n и использует рекурсивную функцию для вывода инструкций для сцены Беккета на *n* персонажей (позиции двоичного разряда, изменяющегося в рефлексивном бинарном коде Грея). Позиции изменяемых двоичных разрядов точно описываются линейчатой функцией, и (конечно) каждый персонаж входит и выходит поочередно.

```
% python beckett.py 1
enter 1

% python beckett.py 2
enter 1
enter 2
exit  1

% python beckett.py 3
enter 1
enter 2
exit  1
enter 3
enter 1
exit  2
exit  1
```

```
% python beckett.py 4
enter 1
enter 2
exit  1
enter 3
enter 1
exit  2
exit  1
enter 4
enter 1
enter 2
exit  1
enter 3
enter 1
exit  2
exit  1
```

Рекурсивная графика. Простые рекурсивные графические схемы способны создать на удивление сложные изображения. Рекурсивные рисунки не только украшают многочисленные приложения, но и обеспечивают платформу

для изучения свойств рекурсивных программ, поскольку они позволяют наглядно отслеживать процесс рекурсивного формирования изображения.

Для первого простого примера мы определим *H-дерево порядка n* (H-tree of order *n*) следующим образом: в конечном случае, при $n = 0$, ничего рисовать не нужно. Этап редукции подразумевает рисование в пределах единичного квадрата следующим образом:

- три линии в форме символа H;
- четыре H-дерева порядка $n-1$ на каждом конце H

с дополнительной оговоркой: размер H-деревьев порядка $n-1$ вдвое меньше, и располагаются они по четырем углам квадрата. Программа 2.3.4 (`htree.py`) получает параметр командной строки *n* и рисует H-дерево порядка *n*.

H-деревья

У подобных рисунков множество практических применений. Рассмотрим, например, кабельную компанию, которая должна провести кабель по всем домам в районе. Разумная стратегия подразумевает использование H-дерева для расположения необходимого количества концентраторов по всему региону и последующей прокладки кабелей от каждого из них до ближайших домов. Та же задача стоит перед разработчиками компьютеров, когда необходимо распределить питание или сигнал по всей интегральной микросхеме.

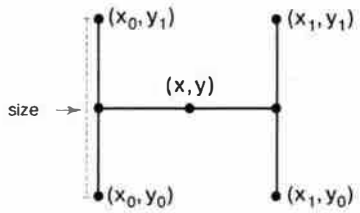
H-деревья обладают экспоненциальным ростом. H-дерево порядка *n* соединяет 4^n центров, таким образом, при $n = 10$ пришлось бы нарисовать больше 1 миллиона линий, а при $n = 15$ — больше 1 миллиарда. При $n = 30$ программе вряд ли удастся закончить рисунок.

Упражнение 2.3.14 требует модифицировать программу `htree.py` так, чтобы анимировать рисунок H-дерева. Если запустить полученную программу на компьютере, то рисование займет приблизительно одну минуту. Вы сможете понаблюдать процесс создания рисунка и лучше понять природу рекурсивных функций, поскольку он наглядно демонстрирует порядок возникновения H-образных фигур и формирования H-деревьев. Еще более поучителен тот факт, что тот же рисунок получается независимо от порядка рекурсивных вызовов функций `draw()` и `stddraw.line()`. Изменение порядка этих вызовов должно влиять на порядок появления линий на рисунке (см. упр. 2.3.14).

Броуновский мост. H-дерево — это простой пример *фрактала*: геометрической фигуры, которая может быть разделена на части, каждая из которых является уменьшенной копией (приблизительно) исходной. Фракталы довольно просто создавать с помощью рекурсивных функций, хотя ученые, математики и программисты изучают их с немного разных точек зрения. В этой книге фракталы уже встречались несколько раз, например, в программе 2.2.4 (`ifs.py`).

Программа 2.3.4. Рекурсивная графика (htree.py)

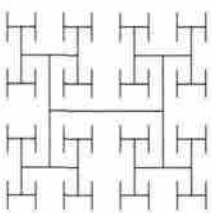
```
import sys
import stddraw

def draw(n, size, x, y):
    if n == 0: return
    x0 = x - size/2.0
    x1 = x + size/2.0
    y0 = y - size/2.0
    y1 = y + size/2.0

    stddraw.line(x0, y, x1, y)
    stddraw.line(x0, y0, x0, y1)
    stddraw.line(x1, y0, x1, y1)

    draw(n-1, size/2.0, x0, y0)
    draw(n-1, size/2.0, x0, y1)
    draw(n-1, size/2.0, x1, y0)
    draw(n-1, size/2.0, x1, y1)

n = int(sys.argv[1])
stddraw.setPenRadius(0.0)
draw(n, 0.5, 0.5, 0.5)
stddraw.show()
```



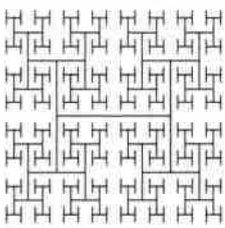
n	Глубина
size	Длина линии
x, y	Центр

Этот сценарий получает аргумент командной строки n и использует рекурсивную функцию для рисования H-дерева порядка n: он рисует три линии в форме символа H, соединяющих центр (x, y) квадрата с четырьмя его вершинами, а затем вызывается для каждой из вершин. Целочисленный аргумент контролирует глубину рекурсии, а аргумент типа float — размер начальной буквы H.

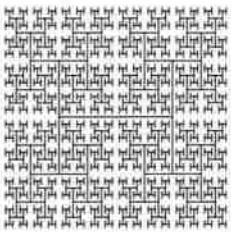
```
% python htree.py 3
```



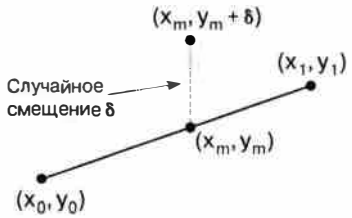
```
% python htree.py 4
```



```
% python htree.py 5
```



Исследование фракталов играет важную роль в художественном выражении, экономическом анализе и научном поиске. Художники и ученые используют их для создания компактных моделей сложных форм, существующих в реальной природе и не поддающихся описанию с использованием обычной геометрии, например, облаков, растений, гор, русел рек, человека и многих других форм. Экономисты также используют фракталы для графиков функции экономических показателей.



Расчет броуновского моста

Фракционное броуновское движение — это математический механизм создания реалистичных рекурсивных моделей для многих естественных форм. Он используется в финансовых вычислениях и исследовании многих естественных явлений, включая океанские течения и мембраны нервных клеток. Вычисление точных фракталов, определенных моделью, может быть довольно трудной задачей, но приблизительные вычисления не столь трудны с рекурсивными функциями. Здесь мы рассмотрим один простой пример; больше подробной информации о модели можно найти на сайте книги.

Программа 2.3.5 (`brownian.py`) рисует приблизительный график функции простого примера фракционного броуновского движения, известного как *броуновский мост* (Brownian bridge). Его можно считать схемой пути при случайных перемещениях между двумя точками от (x_0, y_0) к (x_1, y_1), контролируемого несколькими параметрами. В основе реализации лежит *метод смещения средней точки* (midpoint displacement method), являющийся рекурсивным планом рисунка в интервале $[x_0, x_1]$. Конечный случай подразумевает прямую линию, соединяющую эти две конечных точки. Случай редукции подразумевает деление интервала на две половины со следующим продолжением:

- вычисляется середина интервала (x_m, y_m);
- к значению координаты y (y_m) добавляется случайное значение δ, полученное из Гауссова распределения со средним 0 и заданной дисперсией;
- процесс повторяется для полученных интервалов делением дисперсии на заданный коэффициент масштабирования β.

Форму кривой контролируют два параметра: *волатильность* (volatility) (исходное значение дисперсии), которая определяет расстояние отклонения графика от соединяющей точки прямой линии, и *показатель Херста* (Hurst exponent), определяющий плавность кривой. Обозначаем показатель Херста как H и поделим дисперсию на $\beta = 2^{2H}$ на каждом рекурсивном уровне. Когда H равно 1/2 (деление на 2 на каждом уровне), кривая представляет собой броуновский мост: непрерывная версия задачи разорения игрока (см. программу 1.3.8). Когда $0 < H < 1/2$, смещения начинают увеличиваться, приводя к более грубой кривой; а когда $1/2 < H < 2$, смещения уменьшаются, приводя к более плавной кривой. Значение $2 - H$ известно как *фрактальная размерность* (fractal dimension) кривой.

Программа 2.3.5. Броуновский мост (brownian.py)

```
import math
import sys
import stddraw
import stdrandom

def curve(x0, y0, x1, y1, var, beta, n=7):
    if n == 0:
        stddraw.line(x0, y0, x1, y1)
        return

    xm = (x0 + x1) / 2.0
    ym = (y0 + y1) / 2.0
    delta = stdrandom.gaussian(0.0, math.sqrt(var))
    curve(x0, y0, xm, ym+delta, var/beta, beta, n-1)
    curve(xm, ym+delta, x1, y1, var/beta, beta, n-1)

hurstExponent = float(sys.argv[1])
stddraw.setPenRadius(0.0)
beta = 2.0 ** (2.0 * hurstExponent)
curve(0.0, 0.5, 1.0, 0.5, 0.01, beta)
stddraw.show()
```

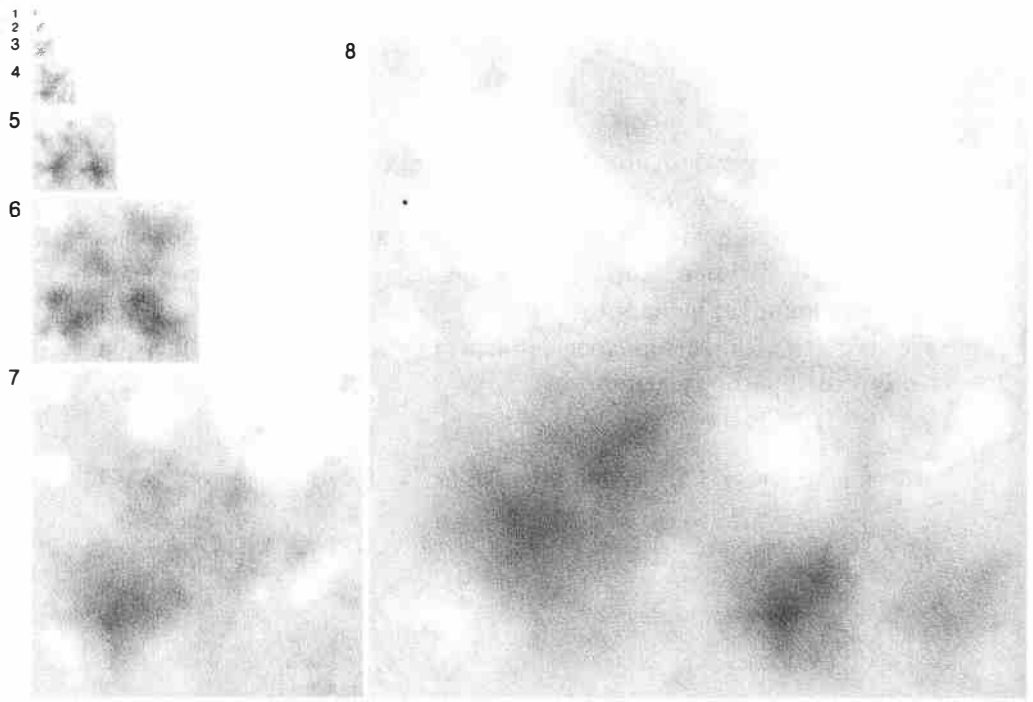
`n`	Глубина рекурсии
`x0, y0`	Левая конечная точка
`x1, y1`	Правая конечная точка
`xm, ym`	Середина
`delta`	Смещение
`var`	Дисперсия
`beta`	Плавность

Этот сценарий рисует броуновский мост через середину окна, с добавлением небольшого случайного числа (полученного из Гауссова распределения) к рекурсивной функции, которая в противном случае рисовала бы прямую линию. Аргумент командной строки, известный как показатель Херста, контролирует плавность кривых — рекурсивная функция использует его для вычисления коэффициента `beta`, корректирующего дисперсию Гауссова распределения.

```
% python brownian.py 1   % python brownian.py .5   % python brownian.py .05
```

Волатильность и конечные точки начального интервала имеют отношение к масштабу и позиции. Глобальный код в программе `brownian.py` позволяет экспериментировать с параметром плавности *H*. При значении, большем 1/2, вы получаете графики, выглядящие как горизонт в гористой местности; при значениях, меньших 1/2, вы получаете графики наподобие биржевых курсов.

Расширение метода смещения средней точки до двух размерностей создает фракталы, известные как *плазменные облака* (plasma cloud). Чтобы получить прямоугольное плазменное облако, используем рекурсивный план, где конечный случай соответствует прямоугольнику заданного цвета, а этап редукции создает плазменное облако, цвета каждого из квадрантов которого зависят от среднего со случайным числом, полученным из Гауссова распределения. Используя ту же волатильность и коэффициент плавности, как в программе `brownian.py`, можно создать чрезвычайно реалистичные искусственные облака. Тот же код можно использовать для создания искусственного ландшафта, если интерпретировать номера цвета как высоту. Варианты этой схемы широко используются в индустрии развлечений для создания фонового пейзажа мультфильмов и компьютерных игр.



Плазменные облака

Проблемы рекурсии. К настоящему времени вы, возможно, убедились, что рекурсия позволяет составлять компактные и изящные программы. Поскольку вы начинаете разрабатывать собственные рекурсивные программы, имеет смысл

узнать о нескольких присущих им общих проблемах. Одна из них уже обсуждалась (продолжительность выполнения программы может возрастать экспоненциально). После выявления этих проблем преодолеть их вообще-то не трудно, но следует соблюдать осторожность и избегать их при создании рекурсивных функций.

Отсутствие конечного случая. Рассмотрите следующую рекурсивную функцию, которая, как предполагается, вычисляет гармонические числа, но конечный случай отсутствует:

```
def harmonic(n):
    return harmonic(n-1) + 1.0/n
```

Если запустить клиент, вызывающий эту функцию, то она будет многократно вызывать сама себя без выхода, в результате программа никогда не завершит работу. Вы, вероятно, уже сталкивались с бесконечными циклами — при запуске программы ничего не происходит (или выводится бесконечная последовательность строк). При бесконечной рекурсии результат будет иным, поскольку система следит за каждым рекурсивным вызовом (используя обсуждаемый в разделе 4.3 механизм на основании стека) и в конечном счете предпринимает попытку записи за пределы памяти. В итоге Python передает сообщение об ошибке `RuntimeError` и уведомление `maximum recursion depth exceeded` (превышена максимальная глубина рекурсии). Запуская рекурсивную программу, всегда следует удостовериться в правильности ее работы, создав неформальный аргумент на основании математической индукции. Это могло бы выявить отсутствие конечного случая.

Нет гарантии схождения. Еще одна общая проблема рекурсивных функций заключается в том, что решаемая в ее пределах задача не менее сложна, чем первоначальная. Например, следующая функция входит в бесконечный рекурсивный цикл для любого значения своего аргумента `n`, кроме `1`:

```
def harmonic(n):
    if n = 1: return 1.0
    return harmonic(n) + 1.0/n
```

Подобные ошибки выявить легко, но более сложные версии той же проблемы обнаружить куда трудней.

Перерасход памяти. Если рекурсивные вызовы функцией самой себя занимают чрезмерно большие промежутки времени до завершения, то Python, вынужденный хранить в памяти каждый вызов, может исчерпать доступную память и выдать сообщение о превышении максимальной глубины рекурсии во время выполнения. Чтобы выяснить объем используемой памяти, проведите небольшую серию экспериментов, используя рекурсивную функцию, например, для вычисления гармонических чисел, постепенно увеличивая значение `n` (начав, скажем, с 1 000 и увеличивая потом в 10 раз):

```
def harmonic(n):
    if n == 1: return 1.0
    return harmonic(n-1) + 1.0/n
```

Момент, когда вы получите сообщение о превышении максимальной глубины рекурсии, это даст вам некоторое представление о количестве памяти, используемой Python для реализации рекурсии. В отличие от этой, программа 1.3.5 способна вычислить H_n для очень большого n, используя совсем мало памяти.

Чрезмерный объем вычислений. Искушение составить для решения задачи простую рекурсивную функцию иногда приходится умерять, поскольку даже довольно простая функция способна создать чрезмерный объем вычислений. Этот эффект возможен даже у самых простых рекурсивных функций, и о нем, безусловно, нужно знать и избегать его. Рассмотрим пример *последовательности Фибоначчи* (Fibonacci sequence):

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, \ldots$$

определяемой как $F_n = F_{n-1} + F_{n-2}$ для $n \geq 2$ и $F_0 = 0$ и $F_1 = 1$. Последовательность Фибоначчи имеет много интересных свойств и применяется довольно часто. Начинающий программист мог бы реализовать рекурсивную функцию для вычисления чисел последовательности Фибоначчи так:

```
def fib(n):
    if n == 0: return 0
    if n == 1: return 1
    return fib(n-1) + fib(n-2)
```

Эта функция абсолютно неправильна, не используйте ее! Начинающие программисты зачастую полагают, что компьютер достаточно быстр, чтобы вычислить и вернуть ответ. Попробуйте проверить, достаточно ли быстр ваш компьютер, чтобы вычислить `fib(50)`. Чтобы узнать, почему это бесполезно, рассмотрим, как функция вычисляет `fib(7)` = 13. Сначала она вычисляет `fib(6)` = 8 и `fib(5)` = 5. Чтобы вычислить `fib(6)`, она *снова* рекурсивно вычисляет `fib(5)` = 5 и `fib(4)` = 3. Дело быстро ухудшается, поскольку оба раза, вычисляя `fib(5)`, она игнорирует тот факт, что `fib(4)`, и так далее она уже вычисляла. Фактически, используя индукцию, вы можете доказать, что количество вычислений этой программой `fib(1)` при начальном `fib(n)` составляет F_n (см. упр. 2.3.12). Количество повторных вычислений нарастает экспоненциально. Например, чтобы вычислить `fib(200)`, данной рекурсивной функции понадобится вычислять `fib(1)` как F_{200}, т.е. более 10^{43} раз! Никакой реальный компьютер никогда не сможет осуществить столько вычислений. *Остерегайтесь программ, способных потребовать экспоненциального времени выполнения.* Множество вычислений, осуществляемых с использованием рекурсивных функций, относится к этой категории. Не попадайтесь в эту ловушку, пытаясь запускать их.

Такая систематическая методика, как *мемоизация* (memoization), позволяет избежать этой проблемы, продолжая в то же время использовать компактные рекурсивное вычисления. Она подразумевает использование массива для хранения уже вычисленных значений, что позволяет избежать их повторного рекурсивного вычисления и применять его только для новых значений. Эта методика относится к *динамическому программированию* — хорошо известной технике организации вычислений, изучаемой на курсах алгоритмических и операционных исследований.

Перспектива. Программисты, не используя рекурсию, упускают две возможности. Во-первых, рекурсия ведет к компактным решениям сложных проблем. Во-вторых, рекурсивные решения воплощают аргумент, который программа использует, как предполагалось. На заре компьютерных вычислений связанные с рекурсивными функциями дополнительные затраты были неприемлемы на некоторых системах, и многие избегали рекурсии. В современных системах, таких как Python, рекурсия зачастую является предпочтительным выбором.

Если вас заинтриговала тайна того, как Python создает иллюзию независимого использования экземпляров той же части кода, то мы рассмотрим эту проблему в главе 4. Простота решения может удивит вас. Реализовать это настолько просто, что программисты использовали рекурсию задолго до появления таких высокоуровневых языков программирования, как Python. Как ни удивительно, но вы вполне могли бы составить программы, эквивалентные рассматриваемым в этой главе, только с использованием базовых циклов, условных выражений и массивов в соответствии с моделью программирования, обсуждаемой в главе 1.

Рекурсивные функции действительно иллюстрируют мощь тщательно сформулированной абстракции. В то время как концепция функции, способной вызвать саму

```
fib(7)
    fib(6)
        fib(5)
            fib(4)
                fib(3)
                    fib(2)
                        fib(1)
                            return 1
                        fib(0)
                            return 0
                        return 1
                    fib(1)
                        return 1
                    return 2
                fib(2)
                    fib(1)
                        return 1
                    fib(0)
                        return 0
                    return 1
                return 3
            fib(3)
                fib(2)
                    fib(1)
                        return 1
                    fib(0)
                        return 0
                    return 1
                fib(1)
                    return 1
                return 2
            return 5
        fib(4)
            fib(3)
                fib(2)
                  .
                  .
                  .
```

Неправильный способ вычисления чисел Фибоначчи

себя, кажется поначалу многим абсурдной, рассмотренные здесь примеры однозначно доказывают, что владение рекурсией является необходимым для понимания роли и использования вычислительных моделей в изучении естественных явлений.

Рекурсия — последняя часть в модели программирования, служившей для построения большей части вычислительной инфраструктуры, разработанной с момента появления компьютеров и до того, как они начали играть центральную роль в повседневной жизни. Программы, созданные из модулей функций, состоящих в свою очередь из операторов, работающих со встроенными типами данных, условными выражениями, циклами и функциями (включая рекурсивные), способны решать важные прикладные задачи всех видов. В следующем разделе мы подчеркнем этот момент и сделаем обзор этих концепций в контексте более широкого применения. В главах 3 и 4 мы рассмотрим продолжение этих фундаментальных идей, обусловливающее более экспансивный стиль программирования, доминирующий ныне в компьютерных вычислениях.

Вопросы и ответы

Бывают ли ситуации, когда итерация — единственная возможность решения задачи?

Нет, любой цикл может быть заменен рекурсивной функцией, хотя рекурсивная версия могла бы потребовать большего расхода памяти.

Бывают ли ситуации, когда рекурсия — единственная возможность решения задачи?

Нет, любая рекурсивная функция может быть заменена итерационным аналогом. В разделе 4.3 описано, как компиляторы создают код вызова функции, используя такую структуру данных, как *стек*.

Что предпочесть, рекурсию или итерацию?

То, что позволит создать более простой, понятный или эффективный код.

Я предупрежден о чрезмерном расходе памяти и чрезмерном объеме вычислений в рекурсивном коде. Мне стоит беспокоиться о чем-то еще?

Особо остерегайтесь создавать массивы в рекурсивном коде. Объем используемого пространства может накапливаться очень быстро, а следовательно, и возрастает период времени, необходимый для управления памятью.

Упражнения

2.3.1. Что будет при вызове функции `factorial()` с отрицательным значением `n`? Или с большим значением `n`, скажем, 35?

2.3.2. Напишите рекурсивную функцию, вычисляющую значение ln(n!)

2.3.3. Получите последовательность целых чисел, выведенных вызовом `ex233(6)`:

```
ef ex233(n):
    if n <= 0: return
    stdio.writeln(n)
    ex233(n-2)
    ex233(n-3)
    stdio.writeln(n)
```

2.3.4. Получите значение `ex234(6)`:

```
def ex234(n):
    if n <= 0: return ''
    return ex234(n-3) + str(n) + ex234(n-2) + str(n)
```

2.3.5. Раскритикуйте следующую рекурсивную функцию:

```
def ex235(n):
    s = ex233(n-3) + str(n) + ex235(n-2) + str(n)
    if n <= 0: return ''
    return s
```

Решение: конечный случай никогда не будет достигнут. Вызов `ex235(3)` приведет к вызовам `ex235(0)`, `ex235(-3)`, `ex235(-6)` и так далее, пока не произойдет превышение максимальной глубины рекурсии.

2.3.6. Дано четыре положительных целых числа, `a`, `b`, `c` и `d`, объясните смысл вычисления `gcd(gcd(a, b), gcd(c, d))`.

2.3.7. Объясните в терминах целых чисел и делителей действие следующей функции, подобной Евклидовой:

```
def gcdlike(p, q):
    if q == 0: return p == 1
    return gcdlike(q, p % q)
```

2.3.8. Рассмотрим следующую рекурсивную функцию:

```
def mystery(a, b):
    if b == 0:
        return 0
    if b % 2 == 0:
        return mystery(a+a, b/2)
    return mystery(a+a, b/2) + a
```

Каковы значения `mystery(2, 25)` и `mystery(3, 11)`? Даны положительные целые числа `a` и `b`, опишите значение, вычисляемое функцией `mystery(a, b)`. Ответьте на тот же вопрос, но замените `+` на `*` и `return 0` на `return 1`.

2.3.9. Напишите рекурсивную программу `rules.py`, рисующую график секций линейки, используя модуль `stddraw` (см. программу 1.2.1).

2.3.10. Решите следующие рекуррентные соотношения, все при $T(1) = 1$. Подразумевается, что n — степени числа 2.

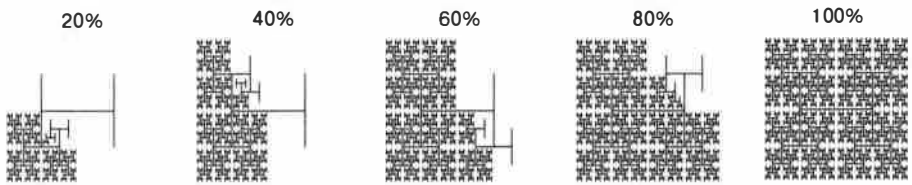
$$T(n) = T(n/2) + 1$$
$$T(n) = 2T(n/2) + 1$$
$$T(n) = 2T(n/2) + n$$
$$T(n) = 4T(n/2) + 3$$

2.3.11. Докажите индукцией, что минимально возможное количество переходов при решении задачи Ханойской башни должно удовлетворять той же рекурренции, что и количество переходов, используемых в соответствии с нашим рекурсивным решением.

2.3.12. Докажите индукцией, что рекурсивная программа, данная в тексте, делает именно F_n рекурсивных вызовов `fib(1)` при вычислении вызова `fib(n)`.

2.3.13. Докажите, что второй аргумент функции `gcd()` уменьшается по крайней мере на коэффициент 2 для каждого второго рекурсивного вызова, а затем докажите, что `gcd(p, q)` использует по крайней мере $2 \log_2 n + 1$ рекурсивных вызов, где n — большее из `p` и `q`.

2.3.14. Измените программу 2.3.4 (`htree.py`) так, чтобы анимировать рисование H-дерева:



Затем перестройте порядок рекурсивных вызовов (и конечный случай), просмотрите полученную анимацию и объясните каждый результат.

Практические упражнения

2.3.15. *Бинарное представление.* Составьте программу, получающую из командной строки положительное целое число n (в десятичном виде) и выводящую его двоичное представление. Помните, что в программе 1.3.7 (`binary.py`)

использовался метод вычитания степеней числа 2. Используйте вместо него следующий упрощенный метод: многократно делите 2 на n и читайте остатки наоборот. Сначала составьте цикл `while`, чтобы выполнить это вычисление и вывести биты в неправильном порядке. Затем используйте рекурсию, чтобы вывести биты в правильном порядке.

2.3.16. *Лист A4.* Отношение ширины к высоте бумаги в формате ISO — квадратный корень из 2 к 1. Площадь листа формата A0 составляет 1 квадратный метр. Формат A1 — половина A0 по вертикали, формат A2 — половина A1 по горизонтали и т.д. Составьте программу, получающую из командной строки целое число n и использующую модуль `stddraw` для демонстрации разделения листа бумаги A0 на 2^n частей.

2.3.17. *Перестановки.* Составьте программу `permutations.py`, получающую из командной строки параметр n и выводящую все $n!$ перестановок для n символов, начиная с `a` (подразумевая, что n не превышает 26). Перестановка n элементов — это один из $n!$ возможных вариантов расстановки элементов. Например, когда $n = 3$, вы должны получить следующий вывод. Не волнуйтесь о порядке вывода перестановок.

```
bca cba cab acb bac abc
```

2.3.18. *Перестановки размером k.* Модифицируйте свое решение предыдущего упражнения так, чтобы оно получало два аргумента командной строки, `n` и `k`, а выводило все перестановки, содержащие только k из n элементов. Количество таких перестановок — $P(n, k) = n!/(n-k)!$. Ниже приведен вывод, когда $k = 2$ и $n = 4$. Не волнуйтесь о порядке вывода перестановок.

```
ab ac ad ba bc bd ca cb cd da db dc
```

2.3.19. *Комбинации.* Составьте программу `combinations.py`, получающую из командной строки один аргумент `n` и выводящую все 2^n комбинаций любого размера. Комбинация — это подмножество из n элементов, независимо от порядка. Например, когда $n = 3$, вы должны получить следующий вывод:

```
 a ab abc ac b bc c
```

Обратите внимание, что программа должна выводить и пустую строку (подмножество размером 0).

2.3.20. *Комбинации размером k.* Модифицируйте свое решение предыдущего упражнения так, чтобы оно получало два аргумента командной строки, `n` и `k`, а выводило все комбинации размером k. Количество таких

комбинаций — $C(n, k) = n!/(k!(n-k)!)$. Например, когда $n = 5$ и $k = 3$, вы должны получить следующий вывод:

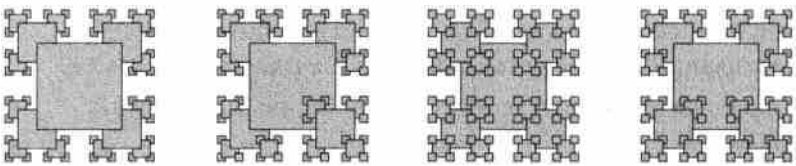
```
abc abd abe acd ace ade bcd bce bde cde
```

2.3.21. *Расстояние Хемминга.* Расстояние Хемминга (Hamming distance) между двумя строками битов длиной `n` равно количеству битов, по которым отличаются эти две строки. Составьте программу, получающую из командной строки целое число `k` и битовую строку `s`, а выводящую все битовые строки, у которых расстояние Хемминга больше `k` из `s`. Например, если `k` = 2, а `s` — `0000`, то ваша программа должна вывести

```
0011 0101 0110 1001 1010 1100
```

Подсказка: выберите `k` из `n` битов в строке `s` для перестановки.

2.3.22. *Рекурсивные квадраты.* Составьте программы, создающие каждый из следующих рекурсивных узоров. Соотношение размеров квадратов 2.2:1. Чтобы получить квадрат с тенью, получите заполненный серый квадрат, затем незаполненный черный квадрат.



2.3.23. *Переворачивание блинов.* На сковородке лежит стопка из n блинов разного размера. Ваша задача — переложить стопку в порядке убывания так, чтобы наибольший блин был внизу, а наименьший — вверху. Переворачивать можно только вершину из k блинов, изменяя таким образом их порядок на обратный. Разработайте рекурсивную схему упорядочивания блинов в надлежащем порядке, использующую не более $2n - 3$ переворачиваний.

2.3.24. *Код Грея.* Модифицируйте программу 2.3.3 (`beckett.py`) так, чтобы выводить код Грея (а не только последовательность изменяемых двоичных позиций).

2.3.25. *Вариант Ханойской башни.* Рассмотрите следующий вариант задачи Ханойской башни. Есть `2n` дисков возрастающего размера на трех стержнях. Первоначально все диски с нечетным размером (`1`, `3`, ..., `2n-1`) сложены на левом стержне сверху вниз в порядке увеличения размера; все диски с четными размерами (`2`, `4`, ..., `2n`) сложены на правом стержне.

Составьте программу, предоставляющую инструкции для перемещения нечетных дисков на правый стержень и четных дисков на левый стержень, согласно правилам Ханойской башни.

2.3.26. *Анимированная Ханойская башня.* Используйте модуль `stddraw` для анимации решения проблемы Ханойской башни, перемещая диски с частотой примерно по 1 в секунду.

2.3.27. *Треугольники Серпинского.* Составьте рекурсивную программу, рисующую треугольники Серпинского (см. программу 2.2.4). Для контроля глубины рекурсии используйте аргумент командной строки, как в программе `htree.py`.

Треугольники Серпинского

2.3.28. *Биномиальное распределение.* Оценочное количество рекурсивных вызовов, которое использовалось бы кодом

```
def binomial(n, k):
    if (n == 0) or (k == 0): return 1.0
    if (n < 0) or (k < 0): return 0.0
    return (binomial(n-1, k) + binomial(n-1, k-1)) / 2.0
```

при вычислении `binomial(100, 50)`. Разработайте лучшую реализацию на базе мемоизации. *Подсказка*: см. упр. 1.4.39.

2.3.29. *Странная функция.* Рассмотрите функцию McCarthy 91:

```
def mcCarthy(n):
    if n > 100: return n - 10
    return mcCarthy(mcCarthy(n+11))
```

Определите значение `mcCarthy(50)`, не используя компьютер. Какое количество рекурсивных вызовов использует функция `mcCarthy()` для вычисления этого результата. Докажите, что конечный случай достижим для всех положительных целых чисел `n`, или укажите значение `n`, для которого эта функция входит в бесконечный рекурсивный цикл.

2.3.30. *Функция Коллатца.* Рассмотрите следующую рекурсивную функцию, связанную с известной нерешенной задачей из теории чисел — *задачей Коллатца* (Collatz problem) или *задачей 3n+1*:

```
def collatz(n):
    stdio.write(str(n) + ' ')
    if n == 1: return
    if n % 2 == 0:
        collatz(n // 2)
    else:
        collatz(3*n + 1)
```

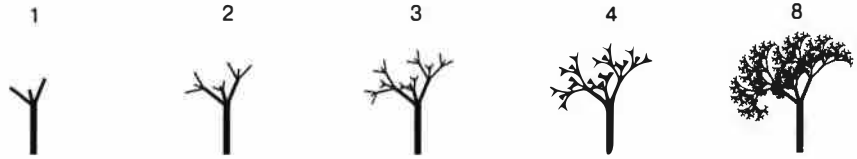
Например, вызов `collatz(7)` выводит следующую последовательность из 17 целых чисел

```
7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
```

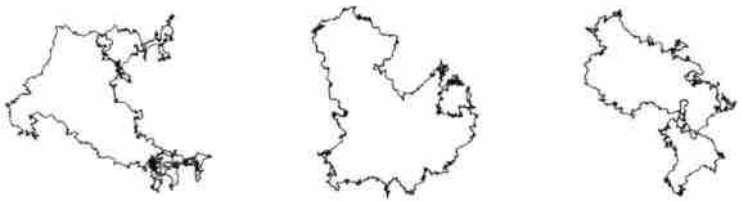
после 17 вызовов функции. Составьте программу, получающую из командной строки аргумент `m` и возвращающую значение `n < m`, для которого количество рекурсивных вызовов функции `collatz(n)` максимально. Нерешенная задача заключается в том, что никто не знает, завершится ли функция для всех положительных значений `n` (математическая индукция не поможет). Разработайте лучшую реализацию на основании мемоизации.

2.3.31. *Плазменные облака.* Составьте программу, использующую для получения плазменных облаков рекурсивный подход, предложенный в тексте.

2.3.32. *Рекурсивное дерево.* Составьте программу `tree.py`, получающую аргумент командной строки `n` и рисующую древовидные рекурсивные узоры, как эти, для *n* равных 1, 2, 3, 4 и 8:



2.3.33. *Броуновский остров.* Б. Мандельброт задал известный вопрос: "*Какова длина побережья Великобритании?*" Модифицируйте программу 2.3.5 (`brownian.py`) так, чтобы получить программу `brownianisland.py`, рисующую *броуновские острова*, береговые линии которых напоминают береговые линии Великобритании. Модификации просты: сначала измените функцию `curve()`, чтобы добавить Гауссову случайную переменную к координате *x*, так же, как к координате *y*; затем измените глобальный код так, чтобы он рисовал замкнутую кривую из точки в центре холста. Поэкспериментируйте с различными значениями параметров, чтобы заставить программу рисовать острова с реалистичным внешним видом.



Броуновские острова с показателем Херста 0,76

2.4. Случай из практики: просачивание

Рассматривавшиеся до сих пор инструменты программирования позволяют решать весь спектр важнейших задач. Мы завершаем свое исследование функций и модулей рассмотрением разработки программы для решения интересной научной проблемы. Наша задача заключается в обзоре уже рассматривавшихся фундаментальных элементов в контексте различных трудностей, с которыми можно столкнуться при решении конкретных задач, а также проиллюстрировать хороший стиль программирования.

Программы этого раздела...

Программа 2.4.1. Скаффолдинг просачивания (percolation0.py) 320
Программа 2.4.2. Обнаружение вертикального просачивания (percolationv.py) 322
Программа 2.4.3. Ввод-вывод просачивания (percolationio.py) 324
Программа 2.4.4. Клиент визуализации (visualizev.py) 325
Программа 2.4.5. Оценка вероятности просачивания (estimatev.py) 327
Программа 2.4.6. Обнаружение просачивания (percolation.py) 329
Программа 2.4.7. Адаптивный графический клиент (percplot.py) 322

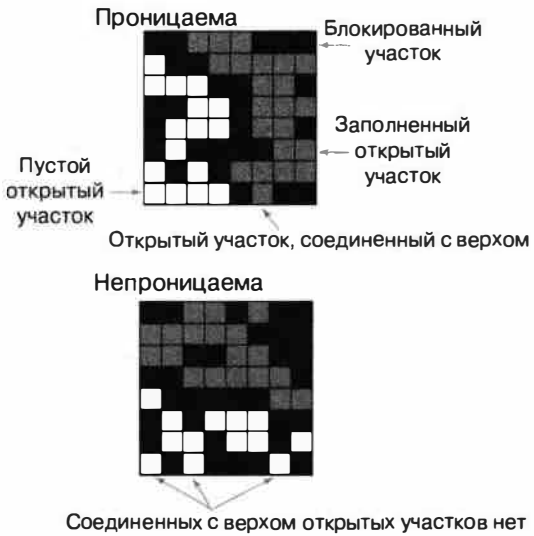
В качестве примера рассмотрим простую модель, которая была чрезвычайно полезна и помогала ученым и инженерам во многих контекстах. Имеется в виду широко распространенная методика, известная как модель Монте-Карло, позволяющая изучить такое природное явление, как *просачивание* (percolation). Оно не только имеет непосредственное отношение к материаловедению и геологии, но и объясняет многие другие природные явления.

Термин *модель Монте-Карло* (Monte Carlo simulation) широко используется и охватывает все вычислительные методики, использующие случайность для оценки неизвестных величин при осуществлении нескольких экспериментов (*моделировании*). Модель Монте-Карло используется и в нескольких других контекстах, например, в уже рассматривавшихся задачах разорения игрока и коллекционера купонов. Вместо разработки полной математической модели или снятия всех возможных результатов эксперимента мы полагаемся на законы вероятности и статистики.

Здесь вы узнаете о задаче просачивания, но основное внимание в данном случае уделяется процессу разработки модульных программ для решения вычислительных задач. Мы выявим подзадачи, которые могут быть решены независимо, ключевые моменты, лежащие в основе абстракций и поднимающие такие вопросы, как: может ли некая конкретная подзадача помочь решить эту задачу? Каковы основные характеристики данной конкретной подзадачи? Может ли это решение, обладающее данными характеристиками, быть полезным в решении других задач? Постановка таких вопросов сулит существенные дивиденды,

поскольку они ведут к разработке такого программного обеспечения, которое проще создавать, отлаживать и многократно использовать, а следовательно, быстрее решить основную задачу, представляющую практический интерес.

Просачивание. Весьма распространена ситуация, когда локальные взаимодействия в системе определяют ее глобальные свойства. Например, инженера-электрика мог бы заинтересовать следующий вопрос о композитном материале, состоящем из случайно перемешанных металлических и изолирующих компонентов: какова должна быть доля металла, чтобы композит стал электрическим проводником? Геолога мог бы заинтересовать другой пример: если есть пористая порода с водой на поверхности (или нефтью под поверхностью), то при каких условиях вода будет просачиваться вглубь (или нефть подниматься на поверхность)? Абстрактный процесс, моделирующий такие ситуации, ученые назвали *просачиванием* (percolation). Он хорошо изучен, и точность моделей на его базе доказана множеством разнообразных приложений, кроме изоляционных материалов, пористых пород, тушения лесных пожаров, подавления эпидемий и исследований развития Интернета.



Просачивание в таблице участков 8 на 8

Для простоты мы начнем работу в двух размерностях и смоделируем систему как таблицу *участков* (site) размером *n* на n. Каждый участок или *блокирован* (blocked), или *открыт* (open); открытые участки первоначально *пусты* (empty). *Заполненный* (full) участок — это открытый участок, который может быть соединен с открытым участком в верхнем ряду по цепочке соседних открытых участков (слева, справа, выше, ниже). Если в нижнем ряду есть заполненный участок, то система *проницаема* (percolate). Другими словами, система проницаема, если есть возможность, заполняя только открытые участки, соединить верхний ряд с нижним. В примере изолятора/проводника открытые участки соответствуют металлу, таким образом, у проницаемой системы (проводящей) был бы металлический путь сверху донизу, отмеченный заполненными участками. Для примера пористой породы открытые участки соответствуют пустому пространству, через которое могла бы течь вода, а проницаемая система позволила бы воде заполнить открытые участки, просачиваясь сверху донизу.

В известной научной задаче исследователей интересует следующий вопрос: если участки независимо устанавливаются открытыми с *вероятностью вакансии p* (а следовательно, блокированными с вероятностью $1-p$), то какова вероятность проницаемости системы? Несмотря на десятилетия научных исследований, математическое решение этой проблемы еще не найдено. Наша задача заключается в составлении компьютерных программ, помогающих изучить проблему.

Простой скаффолдинг. Решение задачи просачивания с использованием программы Python требует преодоления множества затруднений, и в результате получится куда больше кода, чем в предыдущих коротких программах. Наша задача — проиллюстрировать высокий стиль программирования, подразумевающий независимую разработку модулей, решающих части задачи, а также создать атмосферу доверия к вычислительной инфраструктуре из небольших модулей нашего изготовления.

Первый этап — выбор представления данных. Поскольку это решение окажет существенное влияние на вид составляемого впоследствии кода, не стоит принимать его поспешно. В действительности это нередко имеет место, поскольку, когда вы узнаете больше о работе с выбранным представлением, зачастую приходится пересматривать все сначала.

Для задачи просачивания эффективное представление пути очевидно — использование массива *n* на n. Для обозначения пустого участка можно использовать такой код, как 0, для блокированного участка — 1, а для заполненного — 2. Но есть и альтернатива. Обратите внимание, что участки, как правило, описывают в терминах вопросов: является ли участок открытым или блокированным? Заполнен ли он или пуст? Эта характеристика элементов предполагает возможность использования массива *n* на *n* с элементами `True` или `False`. В информатике такие двумерные массивы известны как *логические матрицы* (boolean matrices).

Логические матрицы — это фундаментальные математические объекты со многими применениями. Python не поддерживает операции с логическими матрицами напрямую, но мы можем использовать функции модуля `stdarray` для чтения и вывода логических матриц. Этот выбор иллюстрирует основной принцип программирования: усилия по созданию общих инструментов обычно окупаются. Использование естественной абстракции, такой как логические матрицы, весьма предпочтительно для специализированного представления. В данном случае использование логических матриц вместо целочисленных позволяет создавать более простой и понятный код.

В конечном счете нам предстоит работать со случайными данными, а также понадобится возможность читать и писать файлы, поскольку отладка программ со случайными входными данными может быть непродуктивной. При случайных исходных данных вы будете получать при каждом запуске программы разный ввод; после исправления ошибки вы захотите проверить код на том же вводе. Таким образом, лучше начинать с неких частных понятных нам случаев,

сохраненных в файлах формата, читаемого функцией `stdarray.readBool2D()` (размерности, сопровождаемые значением `0` и `1` в порядном порядке).

Начиная работу над новой задачей, подразумевающей использование нескольких файлов, обычно создают новую папку, в которую помещают ее файлы отдельно от других, над которыми вы, возможно, работаете. Например, для хранения кода, составляемого в этом разделе, можно было бы создать папку `percolation`. Затем можно реализовать и отладить код чтения и записи проницаемых систем, создать файлы проверки, проверить совместимость файлов с кодом и т.д., а затем действительно заняться просачиванием вообще. Код этого типа, иногда называемый *скаффолдинг* (подмости), прост, но его реализация и отладка в самом начале позволит не отвлекаться от решения основной задачи.

Теперь можно обратиться к коду, проверяющему, представляет ли логическая матрица проницаемую систему. Что касается корректности интерпретации, которую мы можем рассматривать как модель задачи, то наше первое решение в проекте, с учетом заполнения водой сверху (т.е. вода просачивается сверху вниз), подразумевает создание функции `flow()`, получающей как аргумент логическую матрицу `isOpen[][]`, определяющую открытые участки и возвращающую другую логическую матрицу `isFull[][]`, определяющую заполненные участки. Не будем пока беспокоиться о реализации этой функции; достаточно решить, как организовать вычисление. Также вполне понятно, что необходим клиентский код, способный использовать функцию `percolates()`, проверяющую, есть ли у возвращенной функцией `flow()` матрицы заполненные участки в самом низу.

Эти решения суммирует программа 2.4.1 (`percolation0.py`). Никаких интересных вычислений она не выполняет, но после реализации и отладки этого кода вполне можно начать думать о фактическом решении проблемы. Функция, не выполняющая никаких вычислений, такая как `flow()`, — это *заглушка* (stub). Наличие заглушки позволяет проверять и отлаживать функции `percolates()` и `main()` в том контексте, в котором они будут использоваться. Код программы 2.4.1 относится к *скаффолдингу*. Подобно подмостьям, используемым рабочими на стройке, код этого вида оказывает поддержку, необходимую для разработки программы. Полная реализация и отладка этого кода (особенно, если не все решения еще приняты) в самом начале обеспечивает надежный фундамент для построения кода собственно решения задачи. Продолжая аналогию, по завершении некоего этапа подмостья перемещают (или заменяют чем-то получше), а после завершения реализации — удаляют.

Вертикальное просачивание. Имея логическую матрицу, представляющую открытые участки, следует выяснить, представляет ли она проницаемую систему? Как будет продемонстрировано в конце этого раздела, данное вычисление непосредственно связано с фундаментальным вопросом информатики, а пока мы будем рассматривать лишь упрощенную версию задачи — *вертикальное просачивание* (vertical percolation).

Программа 2.4.1. Скаффолдинг просачивания (percolation0.py)

```
import stdarray
import stdio

def flow(isOpen):
    n = len(isOpen)
    isFull = stdarray.create2D(n, n, False)
    # Здесь будет вычисление пути просачивания.
    # см. программы 2.4.2 и 2.4.6.
    return isFull

def percolates(isOpen):
    isFull = flow(isOpen)
    n = len(isFull)
    for j in range(n):
        if isFull[n-1][j]: return True
    return False

def main():
    isOpen = stdarray.readBool2D()
    stdarray.write2D(flow(isOpen))
    stdio.writeln(percolates(isOpen))

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

n	Размер системы (n на n)
isFull[][]	Заполненные участки
isOpen[][]	Открытые участки

Приступая к просачиванию, мы составляем этот код для решения всех простых задач, сопутствующих основному вычислению. Главная функция `flow()` возвращает двумерный массив, представляющий заполненные участки (здесь только заглушка). Вспомогательная функция `percolates()` проверяет нижний ряд возвращенного массива и выясняет, проницаема ли система. Клиент проверки читает логический двумерный массив из стандартного ввода, а затем выводит результат вызова функций `flow()` и `percolates()` для этого массива.

```
% more test5.txt
5 5
0 1 1 0 1
0 0 1 1 1
1 1 0 1 1
1 0 0 0 1
0 1 1 1 1
```

```
% python percolation0.py < test5.txt
5 5
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
False
```

Упрощение ограничивает наше внимание только *вертикальными* путями просачивания. Если такой путь соединяет верх с низом, то система считается *вертикально проницаемой* (vertically percolates) по пути. Это ограничение, возможно, интуитивно понятно, если говорить о песке, просыпающемся через бетон, но не о воде, просачиваемой через цемент, или об электрической проводимости. Простая как есть, задача вертикального просачивания интересна сама по себе, поскольку она поднимает различные математические вопросы. Имеет ли это ограничение существенное значение? Сколько вертикальных путей просачивания мы ожидаем?

Вычисление вертикального просачивания

Вычисление заполненных участков, соединенных неким вертикальным путем с верхом, не сложно. Мы инициализируем верхний ряд результирующего массива значениями верхнего ряда массива системы просачивания и заполняем участки, соответствующие открытым. Затем, двигаясь сверху вниз, заполняем каждый ряд массива, проверяя соответствующий ряд системы просачивания. Продолжая сверху вниз, мы заполняем ряды массива `isFull[][]`, отмечая как `True` все элементы, соответствующие участкам в массиве `isOpen[][]`, которые вертикально соединяются с заполненным участком в предыдущем ряду. Программа 2.4.2 (`percolationv.py`) та же, что и `percolation0.py`, за исключением реализации функции `flow()`, возвращающей логическую матрицу заполненных участков (`True`, если соединено с верхом по вертикальному пути, и `False` в противном случае).

Проверка. Убедившись, что код ведет себя так, как планировалось, хотелось бы запустить его на более широком разнообразии тестовых наборов и решить некоторые из наших научных вопросов. На настоящий момент первоначальный скаффолдинг становится менее полезен, как представление больших логических матриц с нулями и единицами на стандартных устройствах ввода и вывода. Большие наборы тестовых данных быстро становятся неинформативными и громоздкими. Вместо этого мы хотели бы автоматически создавать наборы тестов и наблюдать его

Вертикальное просачивание

обработку нашим кодом, чтобы убедиться в правильности его работы. Конкретно, чтобы завоевать доверие к нашему коду и выработать лучшее понимание просачивания, наши следующие задачи таковы.

- Проверить код на больших случайных логических матрицах.
- Получив вероятность вакансии p, оценить вероятность проницаемости системы.

Для решения этих задач необходимы новые клиенты, немного сложнее, чем скаффолдинг, позволивший только создать и запустить программу. Наш модульный стиль программирования подразумевает разработку таких клиентов в независимых модулях *без изменения кода просачивания вообще.*

Программа 2.4.2. Обнаружение вертикального просачивания (percolationv.py)

```
# То же, что и percolation0.py, кроме замены flow() этим:

def flow(isOpen):
    n = len(isOpen)
    isFull = stdarray.create2D(n, n, False)
    for j in range(n):
        isFull[0][j] = isOpen[0][j]
    for i in range(1, n):
        for j in range(n):
            if isOpen[i][j] and isFull[i-1][j]:
                isFull[i][j] = True
    return isFull
```

n	Размер системы (n на n)
isFull[][]	Заполненные участки
isOpen[][]	Открытые участки

Замена заглушки этой функцией в программе 2.4.2 дает решение задачи только вертикального просачивания, что и нужно для нашего конкретного случая проверки (см. текст).

```
% more test5.txt
5 5
0 1 1 0 1
0 0 1 1 1
1 1 0 1 1
1 0 0 0 1
0 1 1 1 1
```

```
% python percolationv.py < test5.txt
5 5
0 1 1 0 1
0 0 1 0 1
0 0 0 0 1
0 0 0 0 1
0 0 0 0 1
True
```

Модель Монте-Карло. Мы хотим, чтобы наш код работал правильно для *любой* логической матрицы. Кроме того, представляющий интерес научный вопрос относится к случайным логическим матрицам. Для этого мы составим функцию `random()`, получающую два аргумента, `n` и `p`, и создающую случайный логический массив *n* на *n* с в вероятностью `p`, что каждый элемент окажется `True`. Отладив код на нескольких конкретных наборах тестов, мы сможем использовать эту функцию для проверки больших случайных систем. Вполне возможно, что эти случаи смогут раскрыть еще несколько ошибок, поэтому отнесемся к проверке результатов ответственно. Однако, отладив наш код для малых систем, мы сможем продолжать уже более уверенно. Куда проще сосредоточиться на новых ошибках после устранения вполне очевидных ошибок.

Визуализация данных. Нам будет легче работать с большими наборами данных, если для вывода мы используем модуль `stddraw`. Таким образом, мы разработаем функцию `draw()` для визуализации в окне стандартного графического устройства содержимого логической матрицы в виде квадратов, по одному для каждого участка. Для обеспечения гибкости мы включим второй аргумент, определяющий, какие квадраты следует заполнить — соответствующие элементам `True` или `False`. Как вы увидите, это окупится при визуализации больших экземпляров матриц. Использование функции `draw()` для вывода массивов с блокированными и заполненными участками различных цветов дает неотразимое визуальное представление просачивания.

При таком наборе инструментов проверка нашего кода просачивания на больших логических матрицах и больших наборах экспериментов довольно проста. Программа 2.4.3 (`percolationio.py`) является небольшим модулем, содержащим две только что описанные функции ввода и вывода, предназначенные для проверки нашего кода просачивания; программа 2.4.4 (`visualizev.py`) — это клиентский сценарий, получающий из командной строки размер системы просачивания, вероятность и количество испытаний, а затем выполняющий указанное количество испытаний, создавая новую случайную логическую матрицу и отображая результат вычисления пути просачивания для каждого, делая короткую паузу между испытаниями.

Возможно, наша задача заключается в точном вычислении оценки вероятности просачивания при большом количестве испытаний. В то же время эти инструменты помогут нам лучше ознакомиться с задачей на примере некоторых больших случаев (а также быть уверенными, что наш код работает правильно). Когда вы запускаете программу `visualizev.py` для умеренного размера *n* (скажем, 50–100) и различных значений *p*, вы сможете сразу попытаться ответить на некоторые вопросы о просачивании. Безусловно, система не будет проницаема при низких *p* и всегда будет проницаема при очень высоких *p*. Но как она поведет себя при промежуточных значениях *p*? Как изменяется поведение при увеличении *n*?

Программа 2.4.3. Ввод-вывод просачивания (percolationio.py)

```
import sys
import stdarray
import stddraw
import stdio
import stdrandom

def random(n, p):
    a = stdarray.create2D(n, n, False)
    for i in range(n):
        for j in range(n):
            a[i][j] = stdrandom.bernoulli(p)
    return a

def draw(a, which):
    n = len(a)
    stddraw.setXscale(-1, n)
    stddraw.setYscale(-1, n)
    for i in range(n):
        for j in range(n):
            if a[i][j] == which:
                stddraw.filledSquare(j, n-i-1, 0.5)

def main():
    n = int(sys.argv[1])
    p = float(sys.argv[2])
    test = random(n, p)
    draw(test, False)
    stddraw.show()

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

Эти вспомогательные функции используются для проверки при изучении просачивания. Вызов `random(n, p)` создает случайный логический массив размером `n` на `n`, где каждый элемент содержит значение `True` с вероятностью `p`, а вызов `draw(test, False)` визуализирует заданный двумерный массива в окне стандартного графического устройства с заполненными квадратами, соответствующими элементам `False`.

```
% python percolationio.py 10 0.8
```

Программа 2.4.4. Клиент визуализации (visualizev.py)

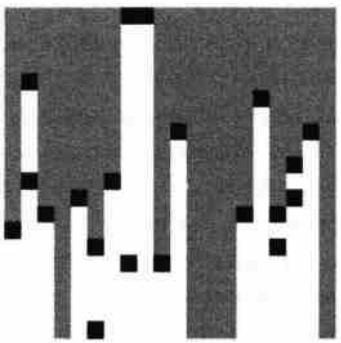
```
import sys
import stddraw
import percolationv
import percolationio

n = int(sys.argv[1])
p = float(sys.argv[2])
trials = int(sys.argv[3])

for i in range(trials):
    isOpen = percolationio.random(n, p)
    stddraw.clear()
    stddraw.setPenColor(stddraw.BLACK)
    percolationio.draw(isOpen, False)
    stddraw.setPenColor(stddraw.BLUE)
    isFull = percolationv.flow(isOpen)
    percolationio.draw(isFull, True)
    stddraw.show(1000.0)
stddraw.show()
```

n	Размер системы (n на n)
p	Вероятность вакансии участка
trials	Количество испытаний
isOpen[][]	Открытые участки
isFull[][]	Заполненные участки

Этот клиент получает из командной строки аргументы n, p и trials, а затем создает случайные логические массивы n на n с вероятностью вакансии участка p, вычисляет прямой путь просачивания и рисует результат в окне стандартного графического устройства. Такие рисунки увеличивают уверенность в правильности работы нашего кода и помогают создать интуитивно понятное представление просачивания.

```
% python visualizev.py 20 0.95 1
```

```
% python visualizev.py 20 0.95 1
```



Оценка вероятностей. Следующий шаг процесса разработки нашей программы подразумевает составление кода оценки вероятности проницаемости случайной системы — *вероятности просачивания* (percolation probability). Для оценки этого значения следует провести много экспериментов. Ситуация не отличается от нашего исследования бросков монеты (см. программу 2.2.7), но вместо бросков монеты мы создаем случайную систему и проверяем ее проницаемость.

Программа 2.4.5 (`estimatev.py`) инкапсулирует это вычисление в функции `evaluate()`, возвращающей оценку вероятности проницаемости системы n на n при вероятности вакансии участков p, полученную в результате проверки заданного количества случайных систем и вычисления доли проницаемых из них. Функция получает три аргумента: `n`, `p` и `trials`.

Сколько испытаний необходимо, чтобы получить точную оценку вероятности просачивания? Этот вопрос решается простыми методами вероятности и статистики, не рассматриваемыми в этой книге, но мы можем получить представление о проблеме на практике. За несколько запусков программы `estimatev.py` вполне можно убедиться, что при вероятности вакансии участков, близкой к 0 или 1, большого количества испытаний не нужно, но есть значения, для которых понадобится порядка 10 000 испытаний, чтобы оценить вероятность просачивания с точностью в две позиции после десятичной точки. Чтобы подробней изучить ситуацию, мы могли бы изменить программу `estimatev.py`, обеспечив ей вывод, как у программы 2.2.7 (`bernoulli.py`), рисующей гистограммы для точек, чтобы увидеть распределение значений (см. упр. 2.4.11).

Применение функции `estimatev.evaluate()` значительно увеличит объем осуществляемых вычислений. Внезапно появляется смысл выполнять тысячи испытаний. Было бы неблагоразумно пытаться сделать это, не отладив сначала полностью функции просачивания. Кроме того, необходимо уделить время завершению необходимых вычислений. Базовая методология этого процесса является темой раздела 4.1, однако структура подобных программ достаточно проста, и вполне можно осуществить быстрое вычисление, проверяемое при запуске программы. Каждое испытание задействует n^2 участков, поэтому полная продолжительность работы функции `estimatev.evaluate()` пропорциональна n^2 умноженное на количество испытаний. Если увеличить количество испытаний в 10 раз (чтобы получить большую точность), продолжительность выполнения также увеличится примерно в 10 раз. Если мы увеличим n в 10 раз (чтобы изучить просачивание больших систем), продолжительность выполнения увеличится примерно в 100 раз.

Можно ли запустить эту программу для определения вероятности просачивания в системе с миллиардом участков при нескольких цифрах точности? Нет. Никакой компьютер не работает так быстро, чтобы использовать функцию `estimatev.evaluate()` в таких условиях. Кроме того, в научном эксперименте на просачивание значение n, вероятно, будет намного выше. Мы можем надеяться сформулировать гипотезу на базе нашей модели и проверить ее

экспериментально на значительно большей системе, но нельзя смоделировать систему, которая точно соответствует атом за атомом реальному миру. Подобное упрощение широко используется в науке.

Программа 2.4.5. Оценка вероятности просачивания (estimatev.py)

```
import sys
import stdio
import percolationv
import percolationio

def evaluate(n, p, trials):
    count = 0
    for i in range(trials):
        isOpen = percolationio.random(n, p)
        if (percolationv.percolates(isOpen)):
            count += 1
    return 1.0 * count / trials

def main():
    n = int(sys.argv[1])
    p = float(sys.argv[2])
    trials = int(sys.argv[3])
    q = evaluate(n, p, trials)
    stdio.writeln(q)

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

n	Размер системы (n на n)
p	Вероятность вакансии участка
trials	Количество испытаний
isOpen[][]	Открытые участки
q	Заполненные участки

Для оценки вероятности проницаемости системы мы создаем случайную матрицу размером n на n с вероятностью вакансии участка p и вычисляем долю проницаемых. Это процесс Бернулли, он не отличается от процесса бросков монеты (см. программу 2.2.6). Увеличение количества испытаний увеличивает точность оценки. Если вероятность вакансии участка близка к 0 или 1, испытаний необходимо не много.

```
% python estimatev.py 20 .75 10
0.0
% python estimatev.py 20 .95 10
1.0
% python estimatev.py 20 .85 10
0.7
% python estimatev.py 20 .85 1000
0.564
% python estimatev.py 20 .85 1000
0.561
% python estimatev.py 40 .85 100
0.11
```

Загрузите программу `estimatev.py` с сайта книги, чтобы получить представление и о вероятности просачивания, и о периоде времени, необходимого для их вычисления. В этом случае вы не только узнаете больше о просачивании, но и проверите гипотезу, что описанные нами модели применимы для дальнейшего моделирования процесса просачивания.

Какова вероятность вертикальной проницаемости системы при вероятности вакансии участка p? Вертикальное просачивание достаточно просто, чтобы элементарные вероятностные модели смогли дать точные формулы для его вычисления, и мы можем проверить его экспериментально программой `estimatev.py`. Таким образом, нашей единственной причиной изучения вертикального просачивания было создание простой отправной точки для разработки программного обеспечения, необходимого для изучения методов просачивания. Дальнейшие исследования вертикального просачивания мы оставляем для упражнения 2.4.13, а сами вернемся к основной задаче.

Рекурсивное решение для просачивания. Как проверить проницаемость системы в общем случае, когда *любой* путь, начинающийся вверху и заканчивающийся внизу (не только вертикальный), решает задачу?

Замечательно, что мы можем решить эту задачу компактной программой на основании классической рекурсивной схемы, известной как *поиск вглубь* (depth-first search). Программа 2.4.6 (`percolation.py`) является реализацией функции `flow()`, вычисляющей матрицу пути `isFull[][]` на основании рекурсивной функции `_flow()` с четырьмя аргументами. Рекурсивная функция получает как аргументы матрицу вакансий участков `isOpen[][]`, матрицу пути `isFull[][]` и позицию участка, определенную индексом ряда `i` и индексом столбца `j`. Конечный случай — просто выход из рекурсии (*пустой вызов* (null call)) по одной из следующих причин:

- значение `i` или `j` вне границ двумерного массива;
- участок блокирован (`isOpen[i][j]` содержит `False`);
- участок уже отмечен как заполненный (`isFull[i][j]` содержит `True`).

Случай редукции должен отмечать участок как заполненный и осуществлять рекурсивный вызов для четырех соседних участков: `isOpen[i+1][j]`, `isOpen[i][j+1]`, `isOpen[i][j-1]` и `isOpen[i-1][j]`. Реализация функции `flow()` вызывает рекурсивную функцию `_flow()` для каждого участка в верхнем ряду. Рекурсия всегда заканчивается, поскольку каждый рекурсивный вызов является либо пустым, либо отмечающим новый участок как заполненный. На основании индукции аргумента (как обычно для рекурсивных программ) мы можем предсказать, что участок отмечается как заполненный, если и только если он является открытым и связанным с одним из участков в верхнем ряду по цепи соседних открытых участков.

Программа 2.4.6. Обнаружение просачивания (percolation.py)

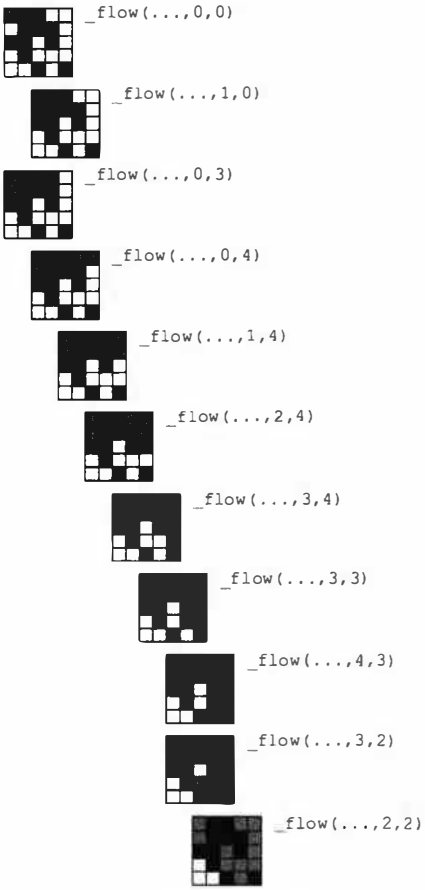
```
# То же, что и percolation0.py, кроме замены flow() этим кодом:

def _flow(isOpen, isFull, i, j):
    n = len(isFull)
    if (i < 0) or (i >= n): return
    if (j < 0) or (j >= n): return
    if not isOpen[i][j]:    return
    if isFull[i][j]:        return

    isFull[i][j] = True
    _flow(isOpen, isFull, i+1, j  ) # Вниз.
    _flow(isOpen, isFull, i  , j+1) # Вправо.
    _flow(isOpen, isFull, i  , j-1) # Влево.
    _flow(isOpen, isFull, i-1, j  ) # Вверх.

def flow(isOpen):
    n = len(isOpen)
    isFull = stdarray.create2D(n, n, False)
    for j in range(n):
        _flow(isOpen, isFull, 0, j)
    return isFull
```

`n`	Размер системы (n на n)
`isOpen[][]`	Открытые участки
`isFull[][]`	Заполненные участки
`i, j`	Ряд и столбец текущего участка

Замените этими функциями заглушки `flow()` в программе 2.4.1, получив решение задачи просачивания на базе поиска вглубь. Вызов `_flow(isOpen, isFull, i, j)` заполняет участки, присваивая значение `True` элементу массива `isFull[][]`, соответствующему каждому участку, который может быть достигнут из `isOpen[i][j]` по пути открытых участков. Функция `flow()` с одним аргументом вызывает рекурсивную функцию для каждого участка в верхнем ряду.

```
% more test8.txt
8 8
0 0 1 1 1 0 0 0
1 0 0 1 1 1 1 1
1 1 1 0 0 1 1 0
0 0 1 1 0 1 1 1
0 1 1 1 0 1 1 0
0 1 0 0 0 0 1 1
1 0 1 0 1 1 1 1
1 1 1 1 0 1 0 0
```

```
% python percolation.py < test8.
txt
8 8
0 0 1 1 1 0 0 0
0 0 0 1 1 1 1 1
0 0 0 0 0 1 1 0
0 0 0 0 0 1 1 1
0 0 0 0 0 1 1 0
0 0 0 0 0 0 1 1
0 0 0 0 1 1 1 1
0 0 0 0 0 1 0 0
True
```

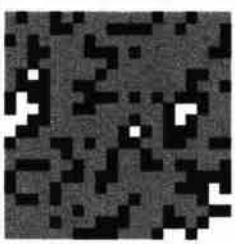


Рекурсивное просачивание (пустые вызовы опущены)

Трассировка функции `_flow()` на крошечном случае проверки является поучительным примером исследования динамики процесса, как показано на рисунке ниже. Представлены вызовы функции для каждого участка, который может быть достигнут по пути открытых участков от верха. Этот пример иллюстрирует, что простые рекурсивные программы способны маскировать вычисления, которые в противном случае были бы весьма сложными. Эта функция — частный случай классического алгоритма поиска вглубь, у которого есть еще много важных применений.

Используя такие сценарии, как `visualizev.py` и `estimatev.py`, а также только что разработанные инструменты, мы можем выполнять эксперименты по этому алгоритму и визуализировать их результаты. Для этого можно создать файл `visualize.py` как копию файла `visualizev.py`, а затем отредактировать его, заменив два экземпляра `percolationv` на `percolation`. Точно так же мы можем создать файл `estimate.py` как отредактированную копию `estimatev.py`. (Для экономии места мы не приводим файлы `visualize.py` и `estimate.py` в книге; они доступны на сайте.) Если вы сделаете это и опробуете разные значения n и p, то быстро получите представление о ситуации: системы всегда проницаемы, когда значение p высоко, и никогда непроницаемы, когда оно низко (особенно при увеличении n); есть такие значения p, выше которых системы почти всегда проницаемы, и есть такие, ниже которых они почти никогда непроницаемы (см. примеры).

```
% python visualize.py 20 .65 1
```



```
% python visualize.py 20 .60 1
```

```
% python visualize.py 20 .55 1
```

Просачивание тем вероятнее, чем менее вероятна вакансия участков

Отладив программы `visualizev.py` и `estimatev.py` на простом вертикальном процессе просачивания, мы можем использовать их с большей уверенностью для исследования просачивания, а затем быстро вернуться к изучению реальной научной проблемы, представляющей интерес. Обратите внимание, что если мы хотим изучить обе версии просачивания, то могли бы составить клиенты и `percolationv`, и `percolation`, оба из которых вызывают функцию `flow()`, а затем сравнить их.

Адаптивный график. Чтобы получить больше представления о просачивании, на следующем этапе разработки составим программу, рисующую график вероятностей просачивания q как функции от вероятности вакансии участка p для заданного значения n. Возможно, наилучший способ построения такого графика состоит в предварительном получении математического уравнения для функции и последующем использовании этого уравнения для создания графика. Но в случае просачивания такое уравнение еще не получено, поэтому используем метод Монте-Карло: запустим модель и нарисуем график результатов.

Сразу возникает множество вопросов. Для скольких значений p следует вычислить оценку вероятности просачивания? Какие значения p следует выбрать? Какую точность следует обеспечить в вычислениях? Принимаемые решения — это задача постановки эксперимента. Поскольку желательно немедленно получить точный преобразованный экземпляр кривой для любого заданного n, стоимость вычислений может стать проблемой. Например, первое, что приходит на ум, — это использование программы 2.2.5 (`stdstats.py`) для графика, скажем, из 100–1 000 равноудаленных точек. Но, как уже было сказано об использовании `estimate.py`, достаточно точные вычисления значений вероятности просачивания по каждой точке могли бы занять несколько секунд и даже больше, поэтому расчет всего графика мог бы занять минуты, часы и даже больше. Кроме того, вполне понятно, что большая часть этого времени тратится совершенно впустую, поскольку уже известно, что для малых p результат равен 0, а для больших p — 1. Это время мы могли бы потратить на получение более точных результатов для промежуточных p. Как же быть?

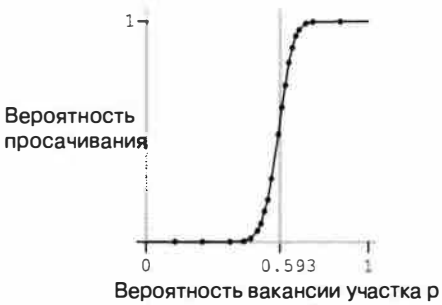
Программа 2.4.7. Адаптивный графический клиент (`percplot.py`)

```
import sys
import stddraw
import estimate

def curve(n, x0, y0, x1, y1, trials=10000, gap=0.01, err=0.0025):

    xm = (x0 + x1) / 2.0
    ym = (y0 + y1) / 2.0

    fxm = estimate.evaluate(n, xm, trials)
    if (x1 - x0 < gap) or (abs(ym - fxm) < err):
        stddraw.line(x0, y0, x1, y1)
        stddraw.show(0.0)
        return

    curve(n, x0, y0, xm, fxm)

    stddraw.filledCircle(xm, fxm, 0.005)
    stddraw.show(0.0)

    curve(n, xm, fxm, x1, y1)

n = int(sys.argv[1])
stddraw.setPenRadius(0.0)
curve(n, 0.0, 0.0, 1.0, 1.0)
stddraw.show()
```

`n`	Размер системы (n на n)
`x0, y0`	Левая конечная точка
`x1, y1`	Правая конечная точка
`xm, ym`	Середина
`fxm`	Значение в середине
`trials`	Количество испытаний
`gap`	Допустимый промежуток
`err`	Допустимая ошибка

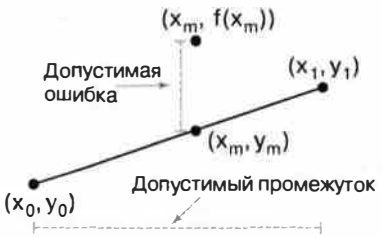
Эта программа получает как аргумент командной строки целое число `n`, а затем проводит эксперименты, чтобы построить адаптивный график зависимости вероятности просачивания (экспериментальные наблюдения) от вероятности вакансии участка (управляющая переменная) для систем размером `n` на `n`.

```
% python percplot.py 20
```




```
% python percplot.py 100
```

Программа 2.4.7 (`percplot.py`) реализует рекурсивный подход, широко применяемый при решении подобных проблем. Фундаментальная идея проста: мы выбираем минимальное расстояние, которое хотим получить между значениями по координате x (*допустимый промежуток* (gap tolerance)), минимум известной ошибки по координате y, которую мы согласны терпеть (*допустимая ошибка* (error tolerance)), и количество испытаний для точки, которое мы желаем проводить. Рекурсивная функция рисует график в пределах заданного интервала по координате x $[x_0, x_1]$, от (x_0, y_0) до (x_1, y_1). В нашем случае график от (0, 0) до (1, 1). Конечный случай (если расстояние между x_0 и x_1 меньше допустимого промежутка или расстояние между линией, соединяющей эти две конечные точки, и значение функции в середине меньше допустимой ошибки) должен просто рисовать линию от точки (x_0, y_0) к точке (x_1, y_1). Этап редукции должен рекурсивно рисовать две половины кривой, от точки (x_0, y_0) к точке (x_m, y_m) и от точки (x_m, y_m) к точке (x_1, y_1).



Адаптивные графические допуски

Код программы `percplot.py` относительно прост и рисует красивую кривую относительно дешево. Мы можем использовать ее для изучения формы кривой при разных значениях n. Для большей уверенности в том, что кривая близка к фактическим значениям, можно выбрать меньшие допуски. В принципе, конкретные математические формулы о качестве приближения могут быть любыми, но, возможно, не стоит слишком увлекаться подробностями при таком исследовании, как наши эксперименты, поскольку их цель только в том, чтобы просто выработать гипотезу о просачивании, которую можно проверить в ходе научного эксперимента.

Созданные программой `percplot.py` кривые сразу подтверждают гипотезу о наличии *порогового значения* (приблизительно 0,593): если p больше порогового значения, то система почти наверняка проницаема; если p меньше порогового значения, то система почти наверняка непроницаема. При увеличении n форма кривой приближается к скачкообразной, как при смене значения с 0 на 1 при достижении порогового значения. Это явление, известное как *фазовый переход* (*p*hase tra*n*sitio*n*), встречается во многих физических системах.

Простота формы вывода программы `percplot.py` маскирует огромный объем скрывающихся позади вычислений. Например, у нарисованной кривой для $n = 100$ есть 18 точек, каждая из которых — результат 10 000 испытаний, в каждом испытании задействовано n^2 участков. Создание и проверка каждого участка задействует несколько строк кода, таким образом, этот график — результат выполнения *миллиардов* операторов. Из этого урока можно сделать два вывода. Во-первых, необходимо быть уверенным в каждой строке кода, способной

```
percplot.curve()
    estimate.evaluate()
        percolation.random()
            stdrandom.bernoulli()
            # .
            # . n² раз
            # .
            stdrandom.bernoulli()
        percolation.percolates()
            flow()
            _flow()
            # .
            # . от n до n² раз
            # .
            _flow()
        # .
        # . trials раз
        # .
        percolation.random()
            stdrandom.bernoulli()
            # .
            # . n² раз
            # .
            stdrandom.bernoulli()
        percolation.percolates()
           flow()
               _flow()
               # .
               # . от n до n² раз
               # .
               _flow()
        # .
        # . trials раз
        # .
```

```
        # .
        # . по разу для каждой точки
        # .
        estimate.evaluate()
            percolation.random()
                stdrandom.bernoulli()
                # .
                # . n² раз
                # .
                stdrandom.bernoulli()
            percolation.percolates()
                flow()
                    _flow()
                    # .
                    # . от n до n² раз
                    # .
                    _flow()
            # .
            # . trials раз
            # .
            percolation.random()
                stdrandom.bernoulli()
                # .
                # . n² раз
                # .
                stdrandom.bernoulli()
            percolation.percolates()
               flow()
                   _flow()
                   # .
                   # . от n до n² раз
                   # .
                   _flow()
```

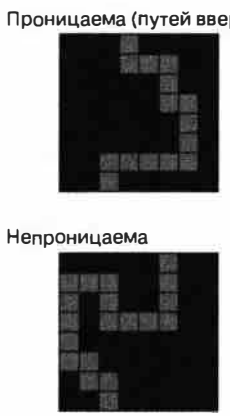
Трассировка вызова `python percplot.py`

выполняться миллиарды раз, поэтому при разработке и отладке кода с отступами следует быть очень внимательным. Во-вторых, хотя нас могли бы интересовать куда *б*ольшие системы, необходимы дальнейшие исследования в области информатики, чтобы появилась возможность обрабатывать большие случаи, т.е. должны быть разработаны более быстрые алгоритмы и среды выполнения, позволяющие существенно улучшить характеристики производительности.

Поскольку все наше программное обеспечение допускает многократное использование, мы можем изучать всякого рода варианты задачи просачивания, реализуя только различные функции `flow()`. Например, если вы не учитываете последний рекурсивный вызов в рекурсивной функции `_flow()` программы 2.4.6, это приведет получению к *направленного просачивания* (directed percolation),

когда пути вверх не рассматриваются. Эта модель могла бы пригодиться для ситуации, когда жидкость просачивается через пористую породу с учетом гравитации, но не для ситуации с электропроводимостью. Если запустить программу `percplot.py` для обеих функций, вы заметите различие (см. упр. 2.4.5)?

Для моделирования физических ситуаций, таких как вода, просачивающаяся сквозь пористые породы, необходимо использовать трехмерные массивы. Есть ли в трехмерной задаче подобие порогового значения? Если да, то каково это значение? Поиск вглубь эффективен для изучения этого вопроса, хотя добавление еще одной размерности требует обратить еще больше внимания на стоимость вычисления проницаемости системы (см. упр. 2.4.21). Ученые изучают также и более сложные структуры, не очень хорошо описываемые многомерными массивами (моделирование таких структур рассматривается в разделе 4.5).



Направленное просачивание

Просачивание интересно изучать на симуляции эксперимента, поскольку никто еще не вывел математических формул для некоторых физических моделей. Единственный известный ученым способ — использование модели Монте-Карло, как было описано в данном разделе. Ученый должен провести эксперименты, чтобы убедиться, отражает ли модель просачивания то, что наблюдается в реальности (возможно, понадобится усовершенствование модели, например, использование другой структуры решетки). Просачивание — это пример увеличения количества проблем, когда программные подходы, подобные описанным здесь, являются основной частью научного процесса.

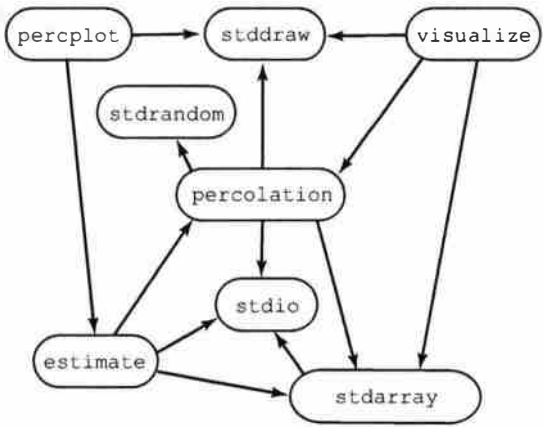
Уроки. Возможно, мы приблизились к решению задачи просачивания, занимаясь всего лишь разработкой и реализацией единственной программы, выполняющей, вероятно, сотни строк кода и создающей графики, подобные рисуемым программой 2.4.7. На заре компьютерной эры у программистов был небольшой выбор, но они работали с подобными программами и тратили массу времени на поиск ошибок и корректировку проектных решений. При наличии современных инструментов программирования, таких как Python, мы можем добиться большего успеха, используя представленный в этой главе новейший модульный стиль программирования, а также учитывая некоторые из выполненных здесь упражнений.

Устраняйте ошибки. В каждой существенной части составляемого вами кода будет по крайней мере одна или две ошибки, если не больше. При запуске малых частей кода для небольших тестовых наборов данных, которые вполне понятны, вам будет проще изолировать ошибки, а затем легче устранить их. Как только

код будет отлажен, вы сможете полагаться на этот модуль и использовать его как стандартный блок для любого клиента.

Ограничивайте размер модулей. Вы можете сосредоточить внимание максимум на нескольких дюжинах строк кода за раз, поэтому при составлении кода имеет смысл разделять его на маленькие модули. Некоторые модули, содержащие коллекции связанных функций, могут в конечном счете достигать сотен строк кода; однако удобней работать с маленькими файлами.

Ограничивайте взаимозависимости. В хорошо проработанной модульной программе большинство модулей должно зависеть от как можно меньшего количества других модулей. В частности, модуль, обращающийся к большому количеству других модулей, должен быть разделен на меньшие части. Модули, к которым обращается большое количество других модулей (у вас такие должны быть), нуждаются в особом внимании, поскольку необходимость внести изменения в API модуля повлечет внесение изменений во все его клиенты.



Анализ зависимостей (без системных обращений)

Разрабатывайте код поэтапно. Запускайте и отлаживайте каждый небольшой модуль по мере реализации. Так вы никогда не будете работать с более чем несколькими дюжинами строк кода одновременно. Если вы поместите весь свой код в один большой модуль, будет достаточно трудно убедиться в отсутствии в нем ошибок. Регулярный запуск кода вынуждает вас также подумать о форматах ввода и вывода, характере задачи и других проблемах. Опыт, полученный из размышлений о таких проблемах, и отладка соответствующего кода делает его более эффективным.

Решайте задачи попроще. Любое рабочее решение лучше отсутствия решения, поэтому обычно начинают со сборки самого простого кода, который вы можете реализовать и который решает данную задачу, как было с вертикальным просачиванием. Эта реализация — первый этап процесса непрерывных

усовершенствований, осуществляемых по мере обретения более полного понимания проблемы в результате исследования более широкого круга тестовых наборов и разработки вспомогательного программного обеспечения, такого, как наши программы `visualize.py` и `estimate.py`.

Рассматривайте возможность рекурсивного решения. Рекурсия — необходимый инструмент в современном программировании, который следует знать и которому стоит доверять. Если вы еще не убедились в этом факте на примере простых и элегантных программ `percplot.py` и `percolation.py`, то можете попытаться разработать не рекурсивную программу для проверки проницаемости системы, а затем сравните их.

По возможности создавайте инструменты. Наши функции визуализации `draw()` и создания случайной логической матрицы `random()` в программе `percolationio.py`, конечно, пригодятся и во многих других приложениях, равно как и функция адаптивной графики `curve()` в программе `percplot.py`. Объединение этих функций в соответствующие модули вовсе на сложно. Реализовать функции общего назначении, как эти, не трудней (а возможно, и проще), чем реализовать функции специального назначения, например, для просачивания.

По возможности многократно используйте код. Наши модули `stdio`, `stdrandom` и `stddraw` упростили процесс разработки кода в этом разделе. Мы также смогли непосредственно многократно использовать такие программы, как `estimate.py` и `visualize.py`, для задачи просачивания после их разработки для случая вертикального просачивания. Составив несколько программ подобного вида, вы сможете разработать такие версии этих программ, которые сможете многократно использовать для других моделей Монте-Карло или других задач экспериментального анализа данных.

Главная задача этой главы — убедить вас в том, что модульное программирование способно завести вас намного дальше, чем вы могли бы добраться без него. Хотя ни один из подходов к программированию — не панацея для всех проблем, инструменты и подходы, обсуждавшиеся в этом разделе, позволят решать сложнейшие задачи программирования, которые в противном случае далеко не всегда могли бы быть решены.

Успех модульного программирования — это только начало. Современные системы программирования полагаются на значительно более гибкую модель, чем только что рассмотренная модель модуля как коллекции функций. В следующих двух главах мы углубим эту модель и приведем множество примеров, иллюстрирующих ее удобство.

Вопросы и ответы

Замена в коде программ `visualize.py` и `estimate.py` всех вхождений слова `percolation` на `percolationv` (или любого другого исследуемого модуля) вызывает беспокойство. Есть ли способ избежать этого?

Да. Проще всего сохранить несколько экземпляров файла `percolation.py` в разных папках. Затем скопировать желаемый файл `percolation.py` из одной из папок в ваш рабочий каталог, выбрав, таким образом, одну конкретную реализацию. Альтернативный подход подразумевает использование оператора Python `import as`, позволяющего определить идентификатор для обращения к модулю:

```
import percolationv as percolation
```

Теперь любой вызов `percolation.percolates()` использует функцию, определенную в файле `percolationv.py`, вместо определенного в файле `percolation.py`. В этой ситуации замена реализаций подразумевает редактирование только одной строки исходного кода.

Рекурсивная функция `flow()` меня тревожит. Как мне лучше понять, что она делает?

Начните с небольших самостоятельно продуманных примеров, оснащенных средствами вывода трассировки вызова функции. Запустив ее несколько раз, и вы удостоверитесь, что она всегда заполняет участки, соединенные с начальной точкой.

Есть ли простой не рекурсивный подход?

Известно несколько подходов осуществления тех же вычислений. Мы вернемся к этой теме в конце книги, в разделе 4.5, а пока работа над разработкой не рекурсивной реализации функции `flow()` будет бесспорно занимательным упражнением, если вам интересно.

Программа 2.4.7 (`percplot.py`), кажется, задействует огромный объем вычислений, чтобы получить простой график функции. Это что, наилучший способ?

Наилучшая стратегия подразумевала бы математическое доказательство порогового значения, но оно пока ускользает от ученых.

Упражнения

2.4.1. Составьте программу, получающую из командной строки аргумент `n` и создающую логический массив *n* на *n*, элемент в ряду *i* и столбце *j* которого содержит `True`, если *i* и *j* взаимно просты, а затем выводит полученный массив на стандартное графическое устройство (см. упр. 1.4.14). Затем составьте подобную программу для вывода *матрицы Адамара* (Hadamard matr*ix*) порядка *n* (см. упр. 1.4.27) и другую программу для вывода матрицы, элемент в ряду *n* и столбце *j* которой установлен в `True`, если коэффициент x^j в $(1+x)^i$ (биномиальный коэффициент) нечетен (см. упр. 1.4.39). Создаваемый им узор просто удивителен.

2.4.2. Составьте функцию `write()` для программы `percolationIO.py`, выводящую `1` для блокированных участков, `0` — для открытых участков и `*` — для заполненных.

2.4.3 Составьте рекурсивные вызовы для программы `percolation.py`, получающей следующий ввод:

```
3 3
1 0 1
0 0 0
1 1 0
```

2.4.4. Составьте клиент для программы `percolation.py`, подобный `visualize.py`, осуществляющий серию экспериментов для аргумента командной строки *n*, где вероятность вакансии участка *p* увеличивается от 0 до 1 с заданным инкрементом (также получаемым из командной строки).

2.4.5. Составьте программу `percolationd.py`, осуществляющую проверку на *направленное* просачивание (удалив последний рекурсивный вызов в рекурсивной функции `_flow()` программы 2.4.6, как было описано в тексте), а затем используйте программу `percplot.py` (соответственно модифицированную) для получения графика вероятности направленного просачивания как функции от вероятности вакансии участка.

2.4.6. Составьте клиент для `percolation.py` и `percolationd.py`, получающий вероятность вакансии участка *p* из командной строки и выводящий оценку вероятности проницаемости системы без просачивания вниз. Проведите достаточно много экспериментов, чтобы получить оценку с точностью в три позиции после десятичной точки.

2.4.7. Опишите порядок, в котором отмечаются участки при использовании программы `percolation.py` для системы без блокированных участков. Какой участок отмечается последним? Какова глубина рекурсии?

2.4.8. Измените программу `percolation.py` так, чтобы анимировать вычисление пути, демонстрируя заполнение участков один за другим. Проверьте результат на предыдущем упражнении.

2.4.9. Поэкспериментируйте с использованием программы `percplot.py` для рисования графиков различных математических функций (достаточно заменить обращение а функции `estimate.evaluate()` выражением, вычисляющим функцию). Опробуйте функцию `sin(x) + cos(10*x)` и убедитесь, что график адаптируется к волнообразной кривой, а также придумайте три или четыре функции для графиков по своему выбору.

2.4.10. Измените программу `percolation.py` так, чтобы вычислить максимальную глубину рекурсии, используемой в вычислении пути. Нарисуйте график ожидаемого значения как функцию от вероятности вакансии участка p. Как изменится ответ, если порядок рекурсивных вызовов изменится на обратный?

2.4.11. Измените программу `estimate.py` так, чтобы создать вывод, как у программы 2.2.7 (`bernoulli.py`). *Дополнительное задание*: используйте свою программу для проверки гипотезы о подчинении данных Гауссову (нормальному) распределению.

2.4.12. Измените программы `percolationio.py`, `estimate.py`, `percolation.py` и `visualize.py` так, чтобы обрабатывать таблицы и логические матрицы размером m на n. Используйте необязательный аргумент со стандартным значением, чтобы присвоить m значение n, если задана только одна из этих двух размерностей.

Практические упражнения

2.4.13. *Вертикальное просачивание.* Продемонстрируйте, что система просачивания n на n с вероятностью вакансии участка p вертикально проницаема с вероятностью $1 - (1 - p^n)^n$. Используйте программу `estimate.py` для проверки результата вашего анализа при различных значениях n.

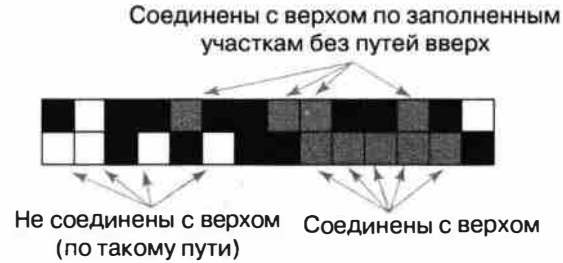
2.4.14. *Прямоугольные системы просачивания.* Измените код этого раздела так, чтобы изучать просачивание в прямоугольных системах. Сравните графики вероятности просачивания систем при соотношении ширины к высоте 2 : 1 с таковыми при соотношении 1 : 2.

2.4.15. *Адаптивное рисование.* Модифицируйте программу `percplot.py` так, чтобы она получала свои управляющие параметры (допустимый промежуток, допустимую ошибку и количество испытаний) из командной

строки. Поэкспериментируйте с различными значениями параметров, чтобы изучить их влияние на качество кривой и стоимость вычисления. Кратко опишите свои поиски.

2.4.16. *Порог просачивания.* Составьте клиент программы `percolation.py`, использующий бинарный поиск для оценки порогового значения (см. упр. 2.1.26).

2.4.17. *Не рекурсивное направленное просачивание.* Составьте не рекурсивную программу проверки направленного просачивания сверху вниз, подобную нашему коду вертикального просачивания. Выработайте свое решение на базе следующих вычислений: если какой-нибудь участок в последовательности открытых участков текущего ряда соседствует с неким заполненным участком в предыдущем ряду, то все участки последовательности заполняются.

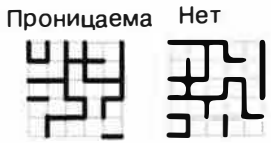


Вычисление направленного просачивания

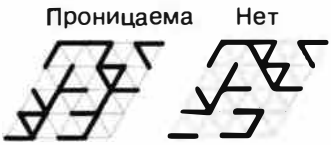
2.4.18. *Быстрая проверка просачивания.* Модифицируйте рекурсивную функцию `_flow()` в программе 2.4.6 так, чтобы она завершалась, как только обнаружит участок в нижнем ряду (и нечего будет больше заполнять). *Подсказка*: используйте аргумент `done`, имеющий значение `True`, если достигнут низ, и `False` в противном случае. Дайте грубую оценку коэффициента улучшения производительности при этом изменении, используя программу `percplot.py`. Используйте значение n, для которого программа выполняется по крайней мере несколько секунд, но не больше нескольких минут. Обратите внимание, что усовершенствование неэффективно, если только первый рекурсивный вызов функции `_flow()` осуществляется не для участка ниже текущего участка.

2.4.19. *Просачивание по границе.* Составьте модульную программу для изучения просачивания согласно предположению, что *край* таблицы обеспечивает

соединение. Таким образом, край может быть или пустым, или заполненным, а система проницаема, если есть путь, проходящий по краю сверху донизу. *Примечание*: эта задача *была* решена аналитически, поэтому ваша модель должна проверить гипотезу, что порог просачивания приближается к 1/2 при достаточно больших n.



2.4.20. *Просачивание по границе в треугольной решетке.* Составьте модульную программу для изучения просачивания по границе *треугольной решетки* (triangular grid), где система состоит из $2n^2$ равносторонних треугольников, упакованных в ромбовидную таблицу n на n, и, как в предыдущем упражнении; край таблицы обеспечивает соединение. У каждой внутренней точки есть шесть связей, у каждой точки на краю — четыре, и у каждой угловой точки — две.



2.4.21. *Просачивание в объеме.* Реализуйте модули `percolation3d.py` и `percolation3dio.py` (для ввода, вывода и создания случайных значений), чтобы изучить просачивание в трехмерных кубах, обобщая двумерный случай, рассмотренный в этой главе. Система просачивания в n на n на n единичных кубических участках с вероятностью их открытости p и вероятностью их блокирования $1-p$. Пути способны соединять открытый куб с любым открытым кубом, имеющим общую грань (один из шести соседей, за исключением границы). Система проницаема, если в ней существует путь, соединяющий любой открытый участок на нижней грани с любым открытым участком на верхней. Используйте рекурсивную версию функции `_flow()`, как в программе 2.4.6, но с восемью рекурсивными вызовами вместо четырех. Нарисуйте график зависимости вероятности просачивания от вероятности вакансии участка для такого большого значения n, как сможете. Разрабатывайте свое решение поэтапно, как и повсюду в этом разделе.

2.4.22. *Игра "Жизнь".* Реализуйте модуль `life.py`, моделирующий игру Джона Конвея "Жизнь". Представьте логическую матрицу, соответствующую системе клеток, которые могут быть живыми или мертвыми. Игра заключается в проверке и, возможно, модификации значения каждой клетки в зависимости от значений ее соседей (смежные клетки в каждом направлении, включая диагонали). Живые клетки остаются живыми, а мертвые остаются мертвыми, со следующими исключениями.

- Мертвая клетка с тремя живыми соседями становится живой.
- Живая клетка с одним живым соседом становится мертвой.
- Живая клетка с более чем тремя живыми соседями становится мертвой.

Проверьте свою программу на *планере* (glider), известном узоре, спускающемся вправо за каждые четыре поколения, как показано на схеме ниже. Затем опробуйте два планера, которые сталкиваются. После этого опробуйте случайную логическую матрицу или используйте один из стартовых узоров на сайте книги. Эта игра была хорошо изучена и имеет отношение к основам информатики (дополнительная информация есть на сайте книги).

Пять поколений планера

Глава 3 Объектно-ориентированное программирование

3.1. Использование типов данных 346
3.2. Создание типов данных.....393
3.3. Разработка типов данных 440
3.4. Случай из практики: моделирование N тел.....486

Следующий шаг к эффективному программированию концептуально прост. Теперь, когда вы умеете использовать встроенные типы данных, рассмотрим в этой главе, как *проектировать*, *создавать* и *использовать* высокоуровневые типы данных.

Абстракция (abstraction) — это упрощенное описание чего-то, включающее лишь основные элементы без всех остальных подробностей. Сложные системы в науке, технике и программировании всегда стараются понять через абстракцию. В программировании на языке Python это осуществляется с использованием *объектно-ориентированного программирования* (object-oriented programming), когда большая и потенциально сложная программа разделяется на набор взаимодействующих элементов или *объектов* (object). Идея берет свое начало в компьютерном моделировании реальных сущностей, таких как электроны, люди, строения или солнечная система, и легко распространяется на моделирование таких абстрактных сущностей, как биты, числа, цвета, изображения или программы.

Как упоминалось в разделе 1.2, тип данных — это набор возможных значений и операций, допустимых для этих значений. В языке Python предопределены значения и операции для многих типов данных, таких как `int` и `float`. В объектно-ориентированном программировании используется код, создающий *новые* типы данных.

Способность определять новые типы данных и манипулировать объектами, содержащими значения этих типов, известна также как *абстракция данных* (data abstraction). Она ведет нас к стилю модульного программирования, вполне естественно дополняющего стиль *функциональной абстракции* (function abstraction), лежавшего в основе материала главы 2. Тип данных позволяет изолировать

данные точно так же, как и функции. Наша мантра в этой главе такова: *всякий раз, когда можете четко разделить данные и связанные с ними задачи в пределах вычисления, так и поступайте.*

3.1. Использование типов данных

Организация данных для обработки является основным этапом разработки компьютерной программы. Программирование на языке Python в значительной степени основано на разработке типов данных, обеспечивающих объектно-ориентированное программирование. Этот стиль программирования подразумевает интеграцию кода и данных.

В первых двух главах этой книги вы, конечно, обратили внимание на то, что наши программы были ограничены операциями с числами, логическими переменными и строками. Причина, безусловно, в том, что типы данных Python, с которыми мы работали до сих пор (`int`, `float`, `bool` и `str`), представляли числа, логические значения и строки, использующие знакомые операции. В этой главе мы начнем рассматривать другие типы данных.

Программы этого раздела...

Программа 3.1.1. Идентификация потенциального гена (`potentialgene.py`) 353
Программа 3.1.2. Клиент заряженной частицы (`chargeclient.py`) 357
Программа 3.1.3. Квадраты Альберса (`alberssquares.py`) 362
Программа 3.1.4. Модуль яркости (`luminance.py`) 364
Программа 3.1.5. Преобразование цвета в полутон (`grayscale.py`) 368
Программа 3.1.6. Масштабирование изображений (`scale.py`) 369
Программа 3.1.7. Эффект постепенного изменения (`fade.py`) 370
Программа 3.1.8. Визуализация электрического потенциала (`potential.py`) 373
Программа 3.1.9. Конкатенация файлов (`cat.py`) 376
Программа 3.1.10. Анализ экранных данных для котировки акций (`stockquote.py`) 378
Программа 3.1.11. Разделение файла (`split.py`) 379

Сначала исследуем набор новых операций с объектами типа `str` как введение в объектно-ориентированное программирование, поскольку большинство этих операций реализуется как методы, манипулирующие объектами. Методы очень похожи на функции, за исключением того, что каждый вызов метода явно ассоциируется с определенным объектом. Для иллюстрации обычного стиля программирования, подразумевающего вызов методов, рассмотрим приложение обработки строк генома.

В разделе 3.2 вы научитесь определять собственные типы данных для реализации любых абстракций вообще. Эта возможность крайне важна для современного программирования. Никакая библиотека модулей не сможет удовлетворить

потребности всех возможных приложений, поэтому для решения конкретных задач разработчики обычно создают собственные типы данных.

В этом разделе мы сосредоточимся на клиентских приложениях, использующих существующие типы данных, чтобы дать вам некоторое представление о новых концепциях и проиллюстрировать их широкую доступность. Мы рассмотрим конструкторы, создающие объекты типа данных, и методы для работы с их значениями. Мы также рассмотрим программы, манипулирующие электрическим током, цветом, изображениями, файлами и веб-страницами, — настоящий скачок по сравнению со встроенными типами данных, использовавшимися в предыдущих главах.

Методы. В разделе 1.2 уже обсуждалось, что тип данных — это набор возможных значений и операций, допустимых для этих значений; вы также узнали подробности о таких встроенных типах данных Python, как `int`, `bool`, `float` и `str`. Вы также узнали, что все данные в программах Python представляются объектами и отношениями между ними. До сих пор мы рассматривали программы, использовавшие операции, связанные со встроенными типами и манипулировавшие объектами этих типов. Для этого использовались условные выражения, циклы и функции, позволявшие создавать большие программы с хорошо понятным управлением потоком выполнения операций. В этом разделе мы объединим все эти концепции.

В программах, встречавшихся до сих пор, мы применяли операции с типами данных, используя такие встроенные операторы, как `+`, `-` `*`, `/` и `[]`. Теперь мы собираемся представить новый, более общий способ применения операций с типами данных. *Метод* (method) — это функция, ассоциируемая с определенным объектом (точнее, с типом этого объекта). Таким образом, метод соответствует операции с типом данных.

Мы можем *вызвать* (invoke) метод, используя имя переменной, сопровождаемое *точечным оператором* (`.`), именем метода и заключенным в круглые скобки списком аргументов, разделяемых запятыми. Например, у встроенного типа Python `int` есть метод `bit_length()`, позволяющий определить количество битов в двоичном представлении значения типа `int` следующим образом:

```
x = 3 ** 100
bits = x.bit_length()
stdio.writeln(bits)
```

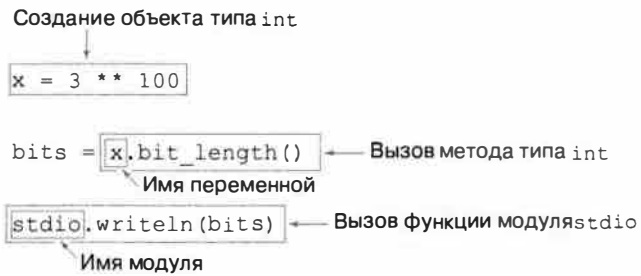
Этот код выводит на стандартное устройство вывода значение `159`, указывая, что 3^{100} (огромное целое число) имеет в двоичном выражении 159 битов.

Синтаксис и действие вызовов методов почти такой же, как синтаксис и действие вызовов функции. Например, метод может получать любые количества аргументов, эти аргументы передаются по ссылке, и метод возвращает значение своей вызывающей стороне. Подобно вызову функции, вызов метода — это

выражение, поэтому его можно использовать везде, где в программе использовалось бы выражение. Основное различие в синтаксисе: для вызова метода используются конкретный объект и точечный оператор.

В объектно-ориентированном программировании обычно предпочитают синтаксис вызова метода синтаксису вызова функции, поскольку он подчеркивает роль объекта. Этот подход прошел проверку в течение многих десятилетий разработки программ, особенно тех, которые предназначены для понимания модели реального мира.

Различие между методами и функциями. Основное различие между функцией и методом в том, что метод ассоциируется с определенным объектом. Этот определенный объект можно считать дополнительным аргументом, передаваемым функции, кроме обычных аргументов метода. Чтобы отличить вызов метода и вызов функции, в клиентском коде достаточно посмотреть влево от точечного оператора: в вызове функции обычно используется имя модуля, а в вызове метода — имя переменной. Эти различия показаны слева на рисунке, приведенном ниже.



Вызовы метода и функции

Обработка строк. С начала этой книги мы использовали тип данных `str`, чтобы создать удобочитаемый вывод наших программ. Наш опыт применения типа `str` показывает, что *вы не обязаны знать, как реализован тип данных, чтобы использовать его* (это одна из нескольких повторяемых в книге мантр в связи с ее важностью). Вы уже знаете, что значения типа `str` — это последовательности символов, и вы можете выполнить операцию конкатенации двух значений типа `str`, чтобы получить результат типа `str`.

Тип данных Python `str` подключает много и других операций, как указано в API далее. Это один из самых важных типов данных Python, поскольку обработка строк критически важна для многих приложений. Строки лежат в основе нашей способности компилировать и выполнять программы Python, а также осуществлять множество других базовых вычислений; на них основано большинство систем обработки данных, критически важных для бизнес-систем. Люди используют их каждый день в электронных сообщениях, блогах, чатах или при подготовке документов к публикации; они критически важны в научном прогрессе, в частности в молекулярной биологии.

Методы и функции

	Метод	Функция
Типичный вызов	`x.bit_length()`	`stdio.writeln(bits)`
Обычно вызывается с	именем переменной	именем модуля
Параметры	Ссылка на объект и аргумент(ы)	Аргумент(ы)
Главная задача	Манипулирование значением объекта	Вычисление возвращаемого значения

Часть API для встроенного типа данных Python `str`

Операция	Описание
`len(s)`	Длина `s`
`s + t`	Новая строка в результате конкатенации `s` и `t`
`s += t`	Присвоение `s` новой строки, полученной в результате конкатенации `s` и `t`
`s[i]`	`i`-й символ `s` (строка)
`s[i:j]`	`i`-й из `j-1` символов `s` (стандартное значение `i` — 0; стандартное значение `j` — `len(s)`)
`s[i:j:k]`	Часть от `i` до `j` с шагом размером `k`
`s < t`	Меньше ли `s`, чем `t`?
`s <= t`	Меньше ли `s`, чем `t`, или равно ему?
`s == t`	Равны ли `s` и `t`?
`s != t`	Не равны ли `s` и `t`?
`s >= t`	Больше ли `s`, чем `t`, или равно ему?
`s > t`	Больше ли `s`, чем `t`?
`s in t`	Присутствует ли `s` как подстрока в `t`?
`s not in t`	Не присутствует ли `s` как подстрока в `t`?
`s.count(t)`	Количество вхождений подстроки `t` в `s`
`s.find(t, start)`	Индекс первого вхождения `t` в `s` (`-1`, если не найдено), `start` начало поиска (стандартно `0`)
`s.upper()`	Копия `s`, где строчные буквы заменены на прописные
`s.lower()`	Копия `s`, где прописные буквы заменены на строчные
`s.startswith(t)`	Начинается ли `s` с `t`?
`s.endswith(t)`	Заканчивается ли `s` на `t`?
`s.strip()`	Копия `s` с удаленными предваряющим и замыкающим пробелами
`s.replace(old, new)`	Копия `s` со всеми вхождениями `old`, замененными на `new`
`s.split(delimiter)`	Массив подстрок `s`, разделенных `delimiter` (стандартно пробелами)
`delimiter.join(a)`	Конкатенация строк в `a[]`, разделенных `delimiter`

При ближайшем рассмотрении операции из API типа `str` можно разделить на три категории.

- *Встроенные операторы* +, +=, [ ], [:], in, not in и операторы сравнения, характеризующиеся специальными символами и синтаксисом.
- *Встроенная функция* len() со стандартным синтаксисом вызова функции.
- *Методы* upper(), startswith(), find() и т.д., отличающиеся в API именем переменной перед точечным оператором.

С этого момента и далее у любого рассматриваемого API будут эти виды операций. Впоследствии мы рассмотрим каждый из них по очереди.

```
a = 'now is '
b = 'the time '
c = 'to'
```

Примеры строковых операций

Вызов	Возвращаемое значение
len(a)	7
a[4]	'i'
a[2:5]	'w i'
c.upper()	'TO'
b.startswith('the')	True
a.find('is')	4
a + c	'now is to'
b.replace('t','T')	'The Time '
a.split()	['now','is']
b == c	False
a.strip()	'now is'

Встроенные операторы. Оператор (или функция), применимый к нескольким типам данных, является *полиморфным*. *Полиморфизм* (polymorphism) — это важнейшее достоинство языка программирования Python и нескольких его встроенных операторов, позволяющее создавать компактный код, используя знакомые операторы при обработке любых типов данных. Для конкатенации строк вы уже использовали оператор +, знакомый по работе с числами. Для извлечения одного символа из строки, согласно API, вы можете использовать оператор [ ], знакомый по работе с массивами, а для извлечения подстроки из строки — оператор [:]. Не все типы данных предоставляют реализации для всех операторов. Например, оператор / не определяется для строк, поскольку нет никакого смысла в делении одной строки на другую.

Встроенные функции. Python обладает также множеством полиморфных функций, таких как len(), которые, вероятно, будут иметь смысл для множества типов данных. Когда тип данных реализует такую функцию, Python автоматически

вызывает эту реализацию на основании типа аргумента. Полиморфные функции похожи на полиморфные операторы, но без специального синтаксиса.

Методы. Мы включили сюда встроенные операторы и встроенные функции для полноты (и в соответствии с нормами Python), однако большая часть усилий при создании типов данных уходит на разработку методов, работающих с их объектами, таких как `upper()`, `startswith()`, `find()`, и других методов из API типа `str`.

Фактически эти три вида операций сводятся к одному: к реализации, как будет продемонстрировано в разделе 3.2. Язык Python автоматически сопоставляет встроенные операторы и функции со *специальными* методами, используя соглашение, согласно которому у таких специальных методов есть по два символа подчеркивания до и после их имен. Например, оператор `s + t` эквивалентен вызову метода `s.__add__(t)`, а вызов `len(s)` эквивалентен вызову метода `s.__len__()`. В клиентском коде мы никогда не используем форму двойного подчеркивания, она используется при реализации специальных методов, как будет продемонстрировано в разделе 3.2.

В таблице ниже приведено несколько простых примеров обработки строк, демонстрирующих удобство различных операций с типом данных Python `str`. Эти примеры — только введение; далее рассматриваются куда более сложные случаи.

Типичный код обработки строк

Преобразование ДНК в мРНК (замена `'T'` на `'U'`)	`def translate(dna):`
	`  dna = dna.upper()`
	`  rna = dna.replace('T', 'U')`
	`  return rna`

Является ли строка `s` палиндромом?	`def isPalindrome(s):`
	`  n = len(s)`
	`  for i in range(n // 2):`
	`  if s[i] != s[n-1-i]:`
	`  return False`
	`  return True`

Извлечение имени и расширения файла из аргумента командной строки	`s = sys.argv[1]`
	`dot = s.find('.')`
	`base = s[:dot]`
	`extension = s[dot+1:]`

Ввод на стандартное устройство всех строк, содержащих подстроку, переданную в аргументе командной строки	`query = sys.argv[1]`
	`while stdio.hasNextLine():`
	`  s = stdio.readLine()`
	`  if query in s:`
	`  stdio.writeln(s)`

Содержит ли массив строки в порядке возрастания?	`def isSorted(a):`
	`  for i in range(1, len(a)):`
	`  if a[i] < a[i-1]:`
	`  return False`
	`  return True`

Приложение обработки строк: геномика. Для приобретения большей практики в обработке строк исследуем очень короткий пример из области *геномики* (genomics) и создадим программу, которая могла бы пригодиться биоинформатикам для идентификации потенциальных генов. Для представления стандартных блоков жизни биологи используют простую модель, где символы `A`, `G`, `C` и `T`[1] представляют четыре основных азотистых основания в ДНК живых организмов. В каждом живом организме эти базовые стандартные блоки существуют в виде ряда длинных последовательностей (по одной в каждой хромосоме) и известны как *геном*. Понимание свойств генома является ключом к пониманию процессов, происходящих в живых организмах. Ныне известны генные последовательности для многих живых существ, включая человека (последовательность приблизительно из 3 миллиардов оснований). С момента открытия этих последовательностей ученые начали составлять компьютерные программы для изучения их структуры. Обработка строк — это теперь одна из самых важных методологий (экспериментальной и компьютерной) молекулярной биологии.

Прогноз генов. Ген — это подстрока в геноме, представляющая функциональный блок, критически важный для понимания жизненных процессов. Ген состоит из последовательности *кодонов* (codon), каждый из которых является последовательностью из трех оснований и представляет одну аминокислоту. *Кодон начала* (start codon), `ATG`, отмечает начало гена, а *кодон конца* (stop codon), `TAG`, `TAA` или `TGA`, отмечает конец гена (внутри гена ни один из вариантов кодонов конца не используется). Один из первых этапов анализ генома подразумевает идентификацию потенциальных генов, что является задачей по обработке строк, для решения которой пригодится тип данных Python `str`.

Первым этапом будет программа 3.1.1 (`potentialgene.py`). Она получает как аргумент командной строки последовательность ДНК и определяет, соответствует ли она потенциальному гену на основании следующих критериев: длина кратна 3, начинается с кодона начала, завершается кодоном конца и не имеет никаких кодонов конца внутри. Для этого программа одновременно использует строковые методы, встроенные операторы и встроенные функции.

Хотя реальные правила, определяющие гены, немного сложнее, чем этот набросок, программа `potentialgene.py` иллюстрирует, как элементарные знания программирования позволяют ученому эффективней изучать последовательности генов.

В данном контексте наш интерес в том, чтобы проиллюстрировать, что тип данных `str` — это хорошо проработанная инкапсуляция важной абстракции, весьма полезная для клиентов. В решении этой задачи нам помогут языковые механизмы Python, от полиморфных функций и операторов до методов, работающих с объектами. Впоследствии мы рассмотрим множество других примеров.

[1] Аденин (A), гуанин (G), цитозин (C), тимин (T). В РНК тимин заменен урацилом (U). — *Примеч. ред.*

Программа 3.1.1. Идентификация потенциального гена (potentialgene.py)

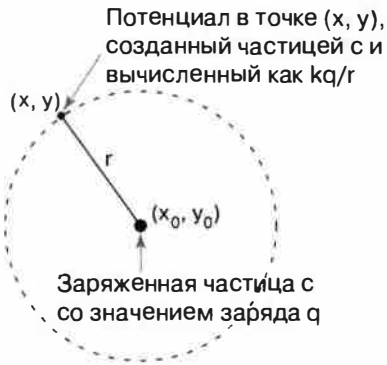
```
import sys
import stdio

def isPotentialGene(dna):
    # количество оснований кратно 3
    if (len(dna) % 3) != 0: return False

    # начинается с кодона начала
    if not dna.startswith('ATG'): return False

    # нет кодонов конца внутри
    for i in range(len(dna) - 3):
        if i % 3 == 0:
            if dna[i:i+3] == 'TAA': return False
            if dna[i:i+3] == 'TAG': return False
            if dna[i:i+3] == 'TGA': return False

    # завершается кодоном конца
    if dna.endswith('TAA'): return True
    if dna.endswith('TAG'): return True
    if dna.endswith('TGA'): return True

    return False

dna = sys.argv[1]
stdio.writeln(isPotentialGene(dna))
```

В аргументе командной строки эта программа получает последовательность ДНК и определяет, соответствует ли она правилам гена: длина кратна 3, начинается с кодона начала (ATG), завершается кодоном конца (TAA, TAG или TGA) и не имеет кодонов конца внутри.

```
% python potentialgene.py ATGCGCCTGCGTCTGTACTAG
True

% python potentialgene.py ATGCGCTGCGTCTGTACTAG
False
```

Пользовательский тип данных. В качестве примера пользовательского типа данных рассмотрим тип `Charge`, представляющий заряженную частицу. В частности, нас интересует ее двумерная модель согласно *закону Кулона*, гласящему, что *электрический потенциал* (electric potential) в точке рядом с данной заряженной

частицей описывается формулой $V = kq / r$, где q — это значение заряда; r — расстояние от точки до заряженной частицы; $k = 8.99 \times 10^9$ N m² / C^2 — электростатическая постоянная, или *постоянная Кулона* (Coulomb's constant). Для полноты в этой формуле мы использовали Международную систему единиц СИ (Système International d'Unités — SI): N — это Ньютоны (сила); m — метры (расстояние); C — кулоны (электрический заряд). Когда есть несколько заряженных частиц, электрический потенциал в любой точке — это сумма потенциалов, созданных каждым зарядом. Нас интересует вычисление потенциала в различных точках на плоскости при заданном наборе заряженных частиц. Для этого мы составим программы, создающие и манипулирующие объектами типа `Charge`.



Закон Кулона для заряженной частицы на плоскости

Интерфейс прикладных программ. Согласно нашей мантре, *вы не обязаны знать, как реализован тип данных, чтобы использовать его.* Определим сначала поведение типа данных `Charge` в списке API для операций с ним, а обсуждение их реализации отложим до раздела 3.2.

API для нашего пользовательского типа данных `Charge`

Операция	Описание
`Charge(x0, y0, q0)`	Новый заряд со значением `q0` расположен в точке `(x0, y0)`
`c.potentialAt(x, y)`	Электрический потенциал заряда `c` в точке `(x, y)`
`str(c)`	`'q0 в (x0, y0)'` (строковое представление заряда `c`)

У первого элемента в API то же имя, что и у типа данных, — это *конструктор* (constructor). Клиенты вызывают конструктор, чтобы создать новый объект; каждый вызов конструктора `Charge` создает один новый объект типа `Charge`. Два других элемента определяют операции с типом данных. Первый метод, `potentialAt()`, вычисляет и возвращает потенциал, созданный зарядом в заданной точке `(x, y)`. Второй метод — это встроенная функция `str()`, возвращающая строковое

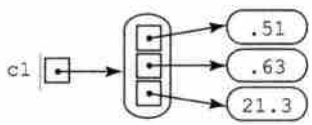
представление заряда частицы. Теперь рассмотрим использование этого типа данных в клиентском коде.

Файловые соглашения. Код, определяющий пользовательский тип данных, содержится в файле с расширением `.py`. В соответствии с соглашением, мы определяем каждый тип данных в отдельном файле `.py`, имя которого совпадает с названием типа данных (но буквами в нижнем регистре). Таким образом, тип данных `Charge` находится в файле `charge.py`. Чтобы использовать тип данных `Charge` в клиентской программе, в начало его файла `.py` следует поместить следующий оператор `import`:

```
from charge import Charge
```

Обратите внимание, что формат оператора `import`, используемого с пользовательскими типами данных, отличается от формата, используемого с функциями. Как обычно, файл `charge.py` должен быть доступен Python, он либо должен располагаться в том же каталоге, что и клиентский код, либо использовать механизм путей операционной системы (см. вопросы и ответы в конце раздела 2.2).

Создание объектов. При создании объекта пользовательского типа данных вызывается его конструктор, инструктирующий Python о том, как создать новый индивидуальный объект. Конструктор вызывают так же, как функцию, используя имя типа данных, сопровождаемое заключенными в круглые скобки аргументами, разделенными запятыми. Например, вызов `Charge(x0, y0, q0)` создает новый объект типа `Charge` с позицией `(x0, y0)` и зарядом `q0`, а затем возвращает ссылку на новый объект. Как правило, конструктор вызывают для создания нового объекта и присвоения переменной ссылки на этот объект в той же строке кода, как в примере ниже. Как только объект создан, значения `x0`, `y0` и `q0` принадлежат *объекту*. Как обычно, вновь созданные в памяти переменную и объект имеет смысл представить, как на схеме ниже.



Создание объекта Charge

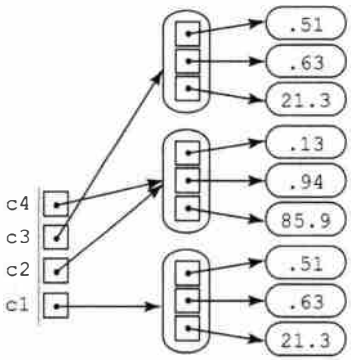
```
c1 = Charge(.51, .63, 21.3)
```

Можно создать любое количество объектов того же типа данных. Как упоминалось в разделе 1.2, у каждого объекта есть собственный идентификатор, тип и значение. В то же время два разных объекта, располагающиеся в разных участках памяти компьютера, вполне могут иметь тот же тип и хранить то же значение. Например, код на схеме ниже создает три разных объекта типа `Charge`. Переменные `c1` и `c3` ссылаются на разные объекты, даже при том, что эти два объекта хранят то же значение. Другими словами, объекты, на которые ссылаются переменные `c1` и `c3`, равны (т.е. это объекты того же типа, случайно имеющие то же значение), но не *идентичны* (т.е. у них разные идентификаторы, поскольку

они располагаются в разных участках машинной памяти). Переменные c2 и c4, напротив, ссылаются на *тот же* объект — они являются псевдонимами.

Вызов метода. Как обсуждалось в начале этого раздела, для идентификации объекта, ассоциированного с вызываемым методом, обычно используют имя переменной. В нашем примере вызов метода `c1.potentialAt(.20, .50)` возвращает значение типа `float`, представляющее потенциал в точке (0.20, 0.50), созданный объектом типа `Charge`, на который ссылается переменная `c1`. Расстояние между запрошенной точкой и местом расположения составляет 0,34; таким образом, потенциал рассчитывается как $8{,}99 \times 10^9 \times 0{,}51 / 0{,}34 = 1{,}35 \times 10^{10}$.

```
c1 = Charge(.51, .63, 21.3)
c2 = Charge(.13, .94, 85.9)
c3 = Charge(.51, .63, 21.3)
c4 = c2
```



Четыре переменные, ссылающиеся на три объекта типа Charge

Строковое представление. В любую реализацию типа данных имеет смысл включить операцию преобразования значения объекта в строку. У Python для этого есть встроенная функция `str()`, которую вы используете с самого начала для преобразования целых и вещественных чисел в строки для вывода. Поскольку в API нашего типа `Charge` есть реализация метода `str()`, любой клиент может вызвать его для получения строкового представления значения объекта типа `Charge`. В нашем примере вызов `str(c1)` возвращает строку `'21.3 at (0.51, 0.63)'`. Характер преобразования полностью зависит от реализации, но обычно строка представляет значение объекта в удобочитаемой форме. Мы исследуем лежащий в основе этого соглашения механизм после обсуждения реализации в разделе 3.2.

Эти механизмы представлены в клиенте `chargeclient.py` (программа 3.1.2), создающем два объекта типа `Charge` и вычисляющем полный потенциал в указанной из командной строки точке, созданный этими двумя зарядами. Данный код иллюстрирует идею разработки абстрактной модели (для заряженной частицы) и отделения кода, реализующего эту абстракцию (которой вы еще даже не видели), от кода, использующего его. Это поворотный момент в книге: вы еще не встречали подобного кода, но практически весь код, создаваемый с этого момента, будет основан на определении и вызове методов, реализующих операции с типами данных.

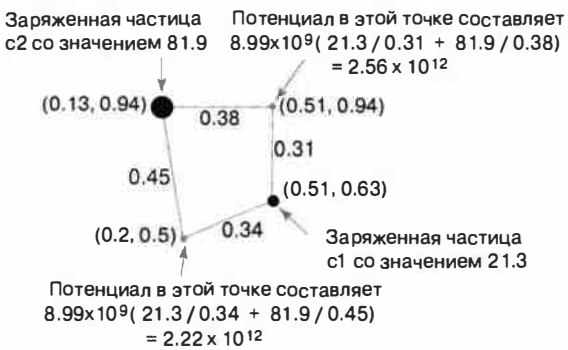
Программа 3.1.2. Клиент заряженной частицы (`chargeclient.py`)

```
import sys
import stdio
from charge import Charge

x = float(sys.argv[1])
y = float(sys.argv[2])
c1 = Charge(.51, .63, 21.3)
c2 = Charge(.13, .94, 81.9)
v1 = c1.potentialAt(x, y)
v2 = c2.potentialAt(x, y)
stdio.writef('potential at (%.2f, %.2f) due to\n', x, y)
stdio.writeln('  ' + str(c1) + ' and')
stdio.writeln('  ' + str(c2))
stdio.writef('is %.2e\n', v1+v2)
```

x, y	Запрашиваемая точка
c1	Первый заряд
c2	Второй заряд
v1	Потенциал от заряда c1
v2	Потенциал от заряда c2

Этот объектно-ориентированный клиент получает в аргументе командной строки точку (x, y), создает два заряда, $c1$ и $c2$, с фиксированной позицией и значениями заряда, а затем выводит на стандартное устройство вывода два заряда и потенциал в точке (x, y), созданный двумя зарядами. Потенциал в точке (x, y) является суммой потенциалов, созданных зарядами $c1$ и $c2$.

```
% python chargeclient.py .2 .5
potential at (0.20, 0.50) due to
    21.3 at (0.51, 0.63) and
    81.9 at (0.13, 0.94)
is 2.22e+12
% python chargeclient.py .51 .94
potential at (0.51, 0.94) due to
    21.3 at (0.51, 0.63) and
    81.9 at (0.13, 0.94)
is 2.56+12
```



Рассмотренные фундаментальные концепции являются отправной точкой объектно-ориентированного программирования, поэтому имеет смысл сделать здесь их обзор. На концептуальном уровне *тип данных — это набор возможных значений и операций, допустимых для этих значений.* На конкретном уровне мы используем тип данных для создания *объектов* (object). Объект характеризуется тремя основными свойствами: *идентификатор*, *тип* и *значение*.

- *Идентификатор* (identity) объекта — это местоположение ячейки в памяти компьютера, где хранится объект. Он однозначно определяет объект.
- *Тип* (type) объекта полностью определяет его *поведение* — это набор операций, поддерживаемых для объекта.
- *Значение* (value) или *состояние* (state) объекта — это значение типа данных, представляемое в настоящий момент объектом.

Для создания объекта в объектно-ориентированном программировании осуществляется вызов *конструктора*, а для последующего манипулирования его значением используется вызов его *методов*. В Python мы обращаемся к объектам по ссылкам. *Ссылка* (reference) — это имя, связанное с ячейкой объекта в памяти (идентификатор).

Известный бельгийский художник Рене Магритт (René Magritte) уловил концепцию ссылки в цикле работ *Вероломство образов*, одной из которых является знаменитое изображение трубки с подписью *Это не трубка* (Ceci n'est pas une pipe). Подпись вполне резонно уведомляет, что изображение трубки — это фактически не сама трубка, а только ее изображение. Либо, возможно, Магритт подразумевал, что подпись — это не трубка и даже не ее изображение, а только подпись! В нынешнем контексте данная картина иллюстрирует идею, что ссылка на объект — это не более чем ссылка, а ничуть не сам объект.

© 2015 С. Г ярославич / Общество защиты прав художников (ARS), Нью-Йорк

Это рисунок трубки

Подобия между пользовательскими и встроенными типами данных. По большей части пользовательские типы данных (т.е. стандартные, дополнительные типы данных и типы данных с сайта книги, а также типы данных, которые вы могли бы определить) никак не отличаются от встроенных типов данных, таких как `int`, `float`, `bool` и `str`. Объекты любого типа можно использовать

- в операторах присвоения;
- как элементы в массивах;
- как аргументы или возвращаемые значения методов или функций;
- как операнды таких встроенный операторов, как `+`, `-`, `*`, `/`,`+=`, `<`, `<=`, `>`, `>=`, `==`, `!=`, `[]` и `[:]`;
- как аргументы таких встроенных функций, как `str()` и `len()`.

Эти возможности позволяют создавать изящный и понятный клиентский код, способный непосредственно манипулировать нашими данными естественным способом, как уже было показано в нашем клиенте `potentialgene.py` и еще будет показано во многих других примерах далее в этом разделе.

Различия между пользовательскими и встроенными типами данных. В языке Python у встроенных типов все же есть особый статус, что доказывают следующие соображения.

- Для использования встроенных типов данных не нужен оператор `import`.
- Для создания объектов встроенных типов данных Python предоставляет специальный синтаксис. Например, литерал `123` создает объект типа `int`, а выражение `['Hello', 'World']` создает массив, элементами которого являются объекты типа `str`. Для создания объектов пользовательского типа данных приходится вызывать конструктор.
- В соответствии с соглашением, названия встроенных типов начинаются со строчных букв, в то время как имена пользовательских типов — с прописных.
- Python предоставляет автоматическое преобразование типов для встроенных арифметических типов данных, таких как `int` и `float`.
- Python предоставляет встроенные функции для преобразования типов *во* встроенные типы, включая `int()`, `float()`, `bool()` и `str()`.

Для демонстрации степени объектной ориентации далее мы рассмотрим еще несколько примеров. Наша главная цель в представлении этих примеров заключается в том, чтобы вы привыкли к идее определения и работы с абстракциями, а также к разработке типов данных, которые полезны вообще и нередко используются в остальной части книги. Сначала мы рассмотрим знакомые задачи обработки изображения, где мы используем объекты типа `Color` и `Picture`. Это две наиболее существенные абстракции, и мы можем сократить их до простых типов данных, которые позволят нам составлять программы обработки изображения, подобные используемым при съемке камерой и представлении на экране. Они также очень полезны при визуализации научных данных, как будет продемонстрировано далее. Затем мы вернемся к теме ввода-вывода на уровне существенно выше средств, предоставляемых книжным модулем `stdio`. В том числе мы рассмотрим абстракции, позволяющие составлять программы Python, способные непосредственно обрабатывать веб-страницы и файлы на вашем компьютере

Цвет (color) — это распознаваемая глазом частота электромагнитного излучения. Поскольку на компьютерах приходится часто просматривать цветные изображения и манипулировать ими, цвет стал широко используемой абстракцией в компьютерной графике. В профессиональной полиграфии, в печати и в веб работа с цветом — непростая задача. Например, внешний вид цветного изображения существенно зависит от среды, используемой для его представления. Наш тип данных `Color`, определенный в модуле `color.py`, отделяет творческую дизайнерскую задачу по определению желаемого цвета от системной задачи по его созданию. API типа данных `Color` представлен далее.

Для представления значения цвета тип `Color` использует *цветовую модель RGB*, в которой цвет определяется тремя целыми числами, от 0 до 255, каждый из которых отвечает за интенсивность красного, зеленого и синего компонентов цвета. Другие значения цветов получаются при смешении красного, зеленого и синего компонентов. Используя эту модель, можно представить 256^3 (т.е. примерно 16,7 миллиона) различных цветов. Ученые установили, что человеческий глаз способен различить лишь порядка 10 миллионов различных цветов.

Некоторые значения цветов

Красный	Зеленый	Синий	
255	0	0	Красный
0	255	0	Зеленый
0	0	255	Синий
0	0	0	Черный
100	100	100	Темно-серый
255	255	255	Белый
255	255	0	Желтый
255	0	255	Сиреневый
9	90	166	*Этот цвет*

У типа `Color` есть конструктор, получающий три целочисленных аргумента, чтобы можно было составить следующий код:

```
red  = color.Color(255,   0,   0)
blue = color.Color(  0,   0, 255)
```

создающий объекты, значения которых представляют чистые красный и синий цвета соответственно. Мы использовали цвета из модуля `stddraw`, начиная с раздела 1.5, но были ограничены рядом предопределенных цветов, таких как `stddraw.BLACK`, `stddraw.RED` и `stddraw.PINK`. Теперь в вашем распоряжении миллионы цветов.

API для нашего типа данных `Color (color.py)`

Операция	Описание
`Color(r, g, b)`	Новый цвет, компоненты которого `r` (красный), `g` (зеленый) и `b` (синий) представлены целыми числами от 0 до 255
`c.getRed()`	Красный компонент `c`
`c.getGreen()`	Зеленый компонент `c`
`c.getBlue()`	Синий компонент `c`
`str(c)`	`'(R, G, B)'` (строковое представление `c`)

Программа 3.1.3 (`alberssquares.py`) является клиентом типа `Color` и модуля `stddraw` и позволяет экспериментировать с цветами. Программа получает из командной строки два цвета и отображает их в формате, разработанным в 1960-х годах

художником и теоретиком Джозефом Альберсом (Josef Albers), сделавшим революцию в способе восприятия цвета людьми.

Наша главная цель — использовать тип `Color` как пример для иллюстрации объектно-ориентированного программирования. Если запускать ее для достаточно большого количества аргументов, то можно убедиться, что даже такая простая программа, как `alberssquares.py`, — весьма полезный и интересный способ изучения взаимодействия цветов. В то же время мы вполне можем также разработать несколько полезных инструментов, которые могут пригодиться при составлении программ с использованием основных цветов. Далее мы выберем одно из цветовых свойств в качестве примера того, что создание объектно-ориентированного кода для обработки абстрактных концепций, таких как цвет, вполне удобный и полезный подход.

Яркость. Качество изображений на современных дисплеях, таких как мониторы LCD, телевизоры LED и экраны сотовых телефонов, зависит от такого свойства цвета, как *монохромная яркость* (monochrome luminance) или *эффективная яркость* (effective brightness). Стандартная формула яркости выведена исходя из чувствительности глаза к красному, зеленому и синему цветам. Это линейная комбинация из трех интенсивностей: если значения красного, зеленого и синего цветов — это r, g и b соответственно, то яркость определяется по следующей формуле:

$$Y = 0.299r + 0.587g + 0.114b$$

Поскольку коэффициенты позитивны и в сумме дают 1, а все интенсивности представлены целыми числами от 0 до 255, то значение яркости — это вещественное число от 0 до 255.

Полутон. У цветовой модели RGB есть интересное свойство: при совпадении интенсивностей всех трех цветов получается серый цвет — полутон, в диапазоне от черного (все 0) до белого (все 255). Для печати цветной фотографии в черно-белой газете (или книге) необходима функция преобразования цветов в полутона. Проще всего преобразовать цвет в полутон — заменить цвет новым, чьи значения красного, зеленого и синего равны его монохромной яркости.

Пример полутонов

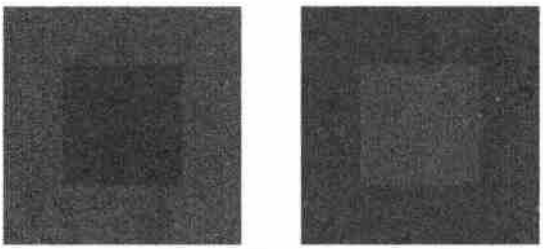
Программа 3.1.3. Квадраты Альберса (`alberssquares.py`)

```
import sys
import stddraw
from color import Color

r1 = int(sys.argv[1])
g1 = int(sys.argv[2])
b1 = int(sys.argv[3])
c1 = Color(r1, g1, b1)

r2 = int(sys.argv[4])
g2 = int(sys.argv[5])
b2 = int(sys.argv[6])
c2 = Color(r2, g2, b2)

stddraw.setPenColor(c1)
stddraw.filledSquare(.25, .5, .2)
stddraw.setPenColor(c2)
stddraw.filledSquare(.25, .5, .1)

stddraw.setPenColor(c2)
stddraw.filledSquare(.75, .5, .2)
stddraw.setPenColor(c1)
stddraw.filledSquare(.75, .5, .1)

stddraw.show()
```

`r1, g1, b1`	Значения RGB
`c1`	Первый цвет
`r2, g2, b2`	Значения RGB
`c2`	Второй цвет

Эта программа отображает два цвета, введенных в командной строке в знакомом формате RGB, разработанным в 1960-х годах художником и теоретиком Джозефом Альберсом.

```
% python alberssquares.py 9 90 166 100 100 100
```



Совместимость цветов. Значение яркости также крайне важно для определения совместимости двух цветов, в смысле читабельности текста, набранного одним цветом на фоне другого. Широко используется эмпирическое правило: различие между яркостями цветов переднего плана и фона должно составлять минимум 128. Например, для черного текста на белом фоне различие яркости составляет 255, а для черного текста на синем фоне различие яркости лишь 74. Это правило важно при создании дизайна объявлений, дорожных знаков, веб-сайтов и многого другого. Программа 3.1.4 (`luminance.py`) представляет собой модуль, позволяющий преобразовать цвет в полутон и проверить, совместимы ли два цвета, например, в приложениях модуля `stddraw`. Функции `luminance()`, `toGray()` и `areCompatible()` в программе `luminance.py` иллюстрируют удобство использования типов данных для организации информации. Использование типа данных `Color` и передачи объектов как аргументов существенно упрощает эти реализации по сравнению с необходимостью передачи трех значений интенсивности. Без типа данных `Color` возвращение нескольких значений из функции также было бы мало изящным и склонным к ошибкам.

Пример совместимости

Абстракция цвета важна не только при прямом использовании, но и при создании высокоуровневых типов данных, обладающих значениями цвета. Далее мы проиллюстрируем этот момент, используя абстракцию цвета при разработке типа данных, используемого при составлении программ по обработке цифровых изображений.

Цифровая обработка изображения. Вы, конечно, знакомы с концепцией фотографии. Технически мы могли бы определить фотографию как двумерное изображение, созданное в результате фокусировки и регистрации электромагнитного излучения с длиной волн в диапазоне видимого света, составляющее представление некой сцены в некий момент времени. Техническое определение не суть важно, но стоит обратить внимание на то, что история фотографии — это история технологий. На протяжении всего прошлого столетия фотография была основана на химических процессах, но отныне ее будущее за цифровыми технологиями. Ваше устройство мобильной связи — это компьютер с линзами и светочувствительными устройствами, способный захватывать изображения в цифровой форме, а у вашего компьютера есть программное обеспечение

редактирования фотографий, позволяющее обработать эти изображения. Их можно обрезать, увеличить, уменьшить, откорректировать контраст, осветлить или затемнить, убрать красные глаза и т.д. Большинство этих операций реализуется на удивление просто, если есть простой тип данных, представляющий идею цифрового изображения.

Программа 3.1.4. Модуль яркости (`luminance.py`)

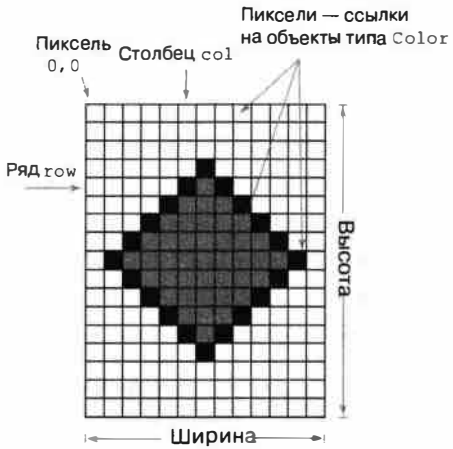
```
import sys
import stdio
from color import Color

def luminance(c):
    red   = c.getRed()
    green = c.getGreen()
    blue  = c.getBlue()
    return .299*red + .587*green + .114*blue

def toGray(c):
    y = int(round(luminance(c)))
    return Color(y, y, y)

def areCompatible(c1, c2):
    return abs(luminance(c1) - luminance(c2)) >= 128.0

def main():
    r1 = int(sys.argv[1])
    g1 = int(sys.argv[2])
    b1 = int(sys.argv[3])
    r2 = int(sys.argv[4])
    g2 = int(sys.argv[5])
    b2 = int(sys.argv[6])
    c1 = Color(r1, g1, b1)
    c2 = Color(r2, g2, b2)
    stdio.writeln(areCompatible(c1, c2))

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

`y`	Яркость `c`
`c1`	Первый цвет
`c2`	Второй цвет

Этот модуль содержит три важных функции манипулирования цветом: яркости, преобразования в полутон и совместимости фона с передним планом.

```
% python luminance.py 232 232 232     0   0   0
True
% python luminance.py   9  90 166   232 232 232
True
% python luminance.py   9  90 166     0   0   0
False
```

Цифровые изображения. Для рисования геометрических объектов (точек, линий, кругов, квадратов) в окне на экране компьютера мы использовали модуль `stddraw`. Какой набор значений необходимо обработать в цифровом изображении и какие операции необходимо осуществить с этими значениями? Фундаментальная абстракция компьютерных дисплеев совпадает с используемой для цифровой фотографии: цифровое изображение — это прямоугольная таблица пикселей (элементов изображения), где цвет каждого пикселя определяется индивидуально. Цифровые изображения иногда называют растром или растровым изображением, в отличие от векторных изображений, создаваемых с использованием модуля `stddraw`.



Анатомия цифрового изображения

Наш тип данных `Picture`, определенный в модуле `picture.py`, реализует цифровую абстракцию изображения. Набор значений — двумерный массив значений типа `Color`, и операции с ним вполне ожидаемы: создание изображения (пустого с заданной шириной и высотой или инициализированного из файла), установка цвета заданного пикселя, возвращение цвета заданного пикселя, возвращение ширины и высоты изображения, отображение изображения в окне на экране компьютера и сохранение изображения в файле.

API для нашего типа данных `Picture (picture.py)`

Операция	Описание
`Picture(w, h)`	Новый массив пикселей размером `w` на `h` (первоначально черного цвета)
`Picture(filename)`	Новое изображение, инициализированное из файла `filename`
`pic.save(filename)`	Сохранение `pic` в файле `filename`
`pic.width()`	Ширина `pic`
`pic.height()`	Высота `pic`
`pic.get(col, row)`	Возвращение цвета пикселя `(col, row)`
`pic.set(col, row, c)`	Установка цвета пикселя `(col, row)`

Примечание. Имя файла должно иметь расширение `.png` или `.jpg`, означающее формат файла.

Согласно соглашению, пиксель (0, 0) находится в верхнем левом углу, поэтому пиксели изображения располагаются в порядке, общепринятом для массивов (в отличие от соглашения модуля `stddraw`, по которому точка (0, 0) располагается

в левом нижнем углу, чтобы рисунки ориентировались способом, общепринятым для Декартовых координат). Большинство программ обработки изображений подобны фильтрам, которые рассматривают пиксели исходного изображения как двумерный массив, а затем, выполнив некую обработку, определяют цвет каждого пикселя в результирующем изображении. Для составления собственных программ по обработке фотографий и добавления результатов в альбом или на вебсайт поддерживаются широко распространенные форматы файлов `png` и `jpg`. Типы данных `Picture` и `Color` открывают дверь в мир обработки изображений.

Наличие метода `save()` позволяет просматривать впоследствии полученные изображения таким же образом, как фотографии или другие изображения. Кроме того, модуль `stddraw` предоставляет функцию `picture()`, позволяющую отображать заданный объект типа `Picture` в окне стандартного графического устройства наряду с линиями, прямоугольниками, кругами и т.д.

API для отображения объекта `Picture`

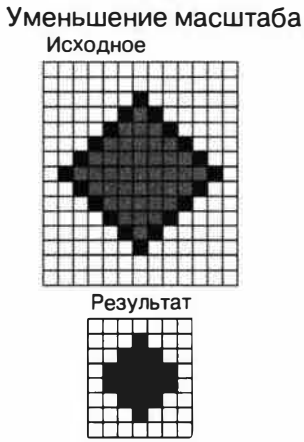
`stddraw.picture(pic, x, y)`	Выводит `pic` в `stddraw` с центром в `(x, y)`

Примечание. Стандартно координаты `x` и `y` отсчитываются от центра холста стандартного графического устройства.

Полутон. На сайте книги можно найти множество примеров цветных изображений, а все описываемые нами методы применимы для цветных изображений, но все примеры изображений в тексте будут полутоновыми. Таким образом, наша первая задача — составить программу, способную преобразовать цветные изображения в полутоновые. Эта задача — прототип задачи по обработке изображения. Каждому пикселю исходного изображения соответствует пиксель в результирующем объекте, но с другим цветом. Программа 3.1.5 (`grayscale.py`) является фильтром, который получает из командной строки имя файла и создает полутоновую версию этого изображения. Она создает новый объект типа `Picture`, инициализированный цветным изображением, а затем присваивает каждому пикселю новый полутоновый цвет `Color`, значение которого вычислено при помощи функции `toGray()` из программы 3.1.4 (`luminance.py`), получающей цвет соответствующего пикселя в исходном изображении.

Масштабирование. Одна из наиболее распространенных задач обработки изображения — это изменение его размера. Примерами этой простой операции *масштабирования* (scaling) является уменьшение фотографий до миниатюр для использования в чате или сотовом телефоне, изменение размера фотографии высокого разрешения, чтобы вписать ее в заданное пространство печатной публикации или веб-страницы, а также увеличение спутниковой фотографии или изображения, полученного при помощи микроскопа. В оптических системах для достижения желаемого масштаба достаточно переместить линзу, но с цифровыми изображениями работы намного больше.

В некоторых случаях стратегия ясна. Например, если результирующее изображение должно быть в половину размера (по каждой размерности) исходного, то достаточно выбрать только половину пикселей или, скажем, удалить половину рядов и половину столбцов. Эта методика известна как *выборка* (sampling). Если результирующее изображение должно удвоить размер (по каждой размерности) исходного, мы можем заменить каждый исходный пиксель четырьмя пикселями того же цвета. Обратите внимание, что при уменьшении масштаба мы теряем информацию, поэтому, уменьшив изображение наполовину, а затем удвоив его, мы не получим прежнего изображения.



Масштабирование цифрового изображения

Для уменьшения и увеличения масштаба эффективна одна стратегия. Наша задача заключается в получении результирующего изображения, потому мы просматриваем его пиксель за пикселем, масштабируя их координаты и получая соответствующий ему цвет из исходного изображения. Если ширина и высота исходного изображения — w_s и h_s соответственно, а ширина и высота результирующего w_t и h_t соответственно, то коэффициент масштабирования для столбцов составит w_s / w_t, а для рядов — h_s / h_t. Таким образом, мы получаем цвет пикселя в столбце c и ряду r результирующего изображения из столбца $c \times w_s / w_t$ и ряда $r \times h_s / h_t$ исходного. Например, если мы делим размер изображения на два, то коэффициенты масштабирования составят 2. Таким образом, пиксель в столбце 4 и ряду 6 результата получит цвет пикселя из столбца 8 и ряда 12 первоисточника. При удвоении размера изображения коэффициенты масштабирования составят 1 / 2; таким образом, пиксель в столбце 4 и ряду 6 результата получит цвет пикселя из столбца 2 и ряда 3 первоисточника. Программа 3.1.6 (`scale.py`) реализует эту стратегию. Для изображений низкого разрешения, встречающихся на старых веб-страницах и камерах, могут быть эффективны более сложные стратегии. Например, при уменьшении вдвое можно было бы вычислить среднее значение четырех пикселей первоисточника и получить цвет одного пикселя результата. Для современных изображений высокого разрешения в большинстве случаев вполне применим и эффективен простой подход, используемый в программе `scale.py`.

Программа 3.1.5. Преобразование цвета в полутон (grayscale.py)

```
import sys
import stddraw
import luminance
from picture import Picture

pic = Picture(sys.argv[1])

for col in range(pic.width()):
    for row in range(pic.height()):
        pixel = pic.get(col, row)
        gray = luminance.toGray(pixel)
        pic.set(col, row, gray)

stddraw.setCanvasSize(pic.width(), pic.height())
stddraw.picture(pic)
stddraw.show()
```

`pic`	Изображение из файла
`col, row`	Координаты пикселя
`pixel`	Цветной пиксель
`gray`	Полутоновый пиксель

Эта программа — иллюстрация простого клиента обработки изображения. Сначала он создает объект типа `Picture`, инициализированный изображением из файла, имя которого передано в аргументе командной строки. Затем он преобразует цвет каждого пикселя изображения в полутон и отображает изображение. На изображении справа можно заметить отдельные пиксели, поскольку оно было увеличено из изображения низкого разрешения (см. обсуждение масштабирования выше).

```
% python grayscale.py mandrill.jpg
```

```
% python grayscale.py darwin.jpg
```

Та же фундаментальная идея вычисления значения цвета каждого выходного пикселя как функции от значений цветов определенных исходных пикселей эффективна для обработки изображения любого рода. Далее мы рассмотрим еще два примера, а в упражнениях и на сайте книги можно найти еще множество других примеров.

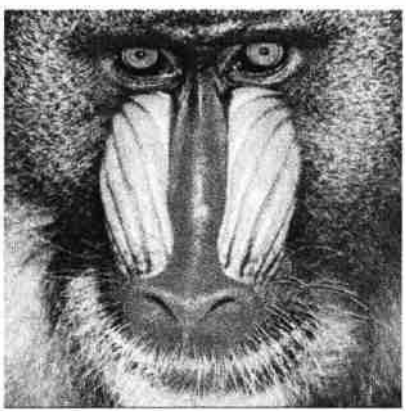
Программа 3.1.6. Масштабирование изображений (scale.py)

```
import sys
import stddraw
from picture import Picture

file = sys.argv[1]
wT = int(sys.argv[2])
hT = int(sys.argv[3])

source = Picture(file)
target = Picture(wT, hT)

for colT in range(wT):
    for rowT in range(hT):
        colS = colT * source.width()  // wT
        rowS = rowT * source.height() // hT
        target.set(colT, rowT, source.get(colS, rowS))

stddraw.setCanvasSize(wT, hT)
stddraw.picture(target)
stddraw.show()
```

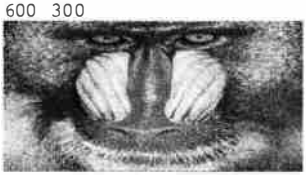
wT, hT	Конечные размеры
source	Исходное изображение
target	Конечное изображение
colT, rowT	Координаты конечного пикселя
colS, rowS	Координаты исходного пикселя

Эта программа получает как аргументы командной строки имя файла изображения `.jpg` или `.png` и два целых числа, `wT` и `hT`. Она отображает изображение в масштабе по ширине `wT` и высоте `hT`.

```
% python scale.py mandrill.jpg 800 800
```



600 300



200 400

200 200

Программа 3.1.7. Эффект постепенного изменения (fade.py)

```
import sys
import stddraw
from color import Color
from picture import Picture

def blend(c1, c2, alpha):
    r = (1-alpha)*c1.getRed() + alpha*c2.getRed()
    g = (1-alpha)*c1.getGreen() + alpha*c2.getGreen()
    b = (1-alpha)*c1.getBlue() + alpha*c2.getBlue()
    return Color(int(r), int(g), int(b))

sourceFile = sys.argv[1]
targetFile = sys.argv[2]
n = int(sys.argv[3])

source = Picture(sourceFile)
target = Picture(targetFile)

width = source.width()
height = source.height()

stddraw.setCanvasSize(width, height)
pic = Picture(width, height)
for t in range(n+1):
    for col in range(width):
        for row in range(height):
            c0 = source.get(col, row)
            cn = target.get(col, row)
            alpha = 1.0 * t / n
            pic.set(col, row, blend(c0, cn, alpha))
    stddraw.picture(pic)
    stddraw.show(1000.0)

stddraw.show()
```

Чтобы постепенно превратить одно изображение в другое за `n-1` промежуточных этапа, для каждого пикселя в `t`-м изображении устанавливается средневзвешенное значение соответствующих пикселей исходного и конечного изображений, причем для исходного значения используется весовой коэффициент `1-t/n`, а для конечного — `t/n`. Пример преобразования, приведенный ниже, создан вызовом `python fade.py mandrill.jpg darwin.jpg 9`.

Эффект постепенного изменения. Наш следующий пример обработки изображения — преобразование одного изображения в другое в ходе серии отдельных этапов. Речь идет о преобразовании, известном как *эффект постепенного изменения* (fade effect). Программа 3.1.7 (`fade.py`) — это клиент типов `Picture`, `Color` и модуля `stddraw`, использующий для реализации этого эффекта стратегию линейной интерполяции. Она вычисляет n–1 промежуточных изображений, где каждый пиксель в t-м изображении является средневзвешенным средним от соответствующих пикселей в исходном и результирующем изображениях. Функция `blend()` реализует интерполяцию: исходный цвет взвешен коэффициентом $1 - t / n$, а результирующий цвет — коэффициентом t / n (когда t равно 0, имеется исходный цвет; когда t равно n — результирующий). Это простое вычисление может привести к удивительным результатам. При запуске программы `fade.py` на компьютере изменения отображаются динамически. Попробуйте запустить ее для некоторых изображений из вашей библиотеки фотографий. Обратите внимание: программа `fade.py` подразумевает, что ширина и высота изображений совпадают; если дело обстоит не так, то можно использовать программу `scale.py` для получения масштабной версии одного или обоих изображений.

Визуализация значения потенциала. Обработка изображения полезна также для визуализации научных данных. В качестве примера рассмотрим клиент типа `Picture` для визуализации свойств объекта типа `Charge`, определенного в начале этого раздела. Программа 3.1.8 (`potential.py`) визуализирует значения потенциалов, созданных рядом заряженных частиц. Сначала программа `potential.py` создает массив частиц со значениями, полученными со стандартного ввода. Затем она создает объект типа `Picture` и устанавливает для каждого пикселя изображения полутоновый оттенок, пропорциональный значению потенциала в соответствующей точке. В основе подхода очень простое вычисление: для каждого пикселя (x, y) вычисляется соответствующее значение в единичном квадрате, затем для каждого заряда вызывается функция `potentialAt()`, а возвращаемые потенциалы для этой точки суммируются, и значение возвращается. При присвоении полутоновых значений, соответствующих значениям потенциалов (отмасштабированных так, чтобы попасть в диапазон от 0 до 255), мы получаем интересное визуальное представление электрического потенциала, что прекрасно помогает понять взаимодействие таких частиц. Подобные изображения можно получить, используя функцию `filledSquare()` из

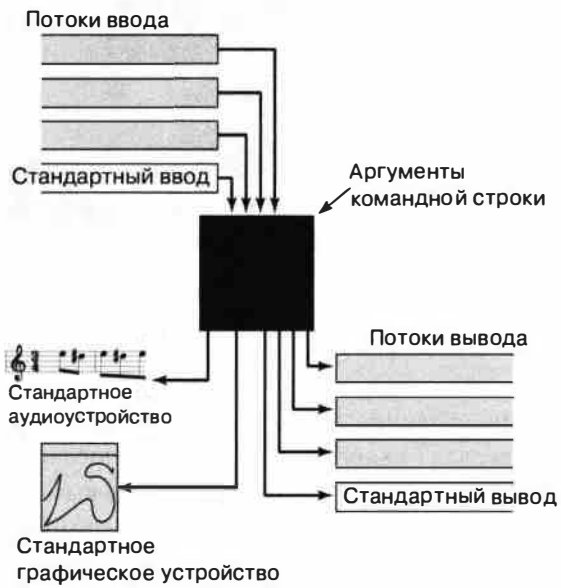
```
% more charges.txt
9
.51   .63 -100
.50   .50   40
.50   .72   10
.33   .33    5
.20   .20  -10
.70   .70   10
.82   .72   20
.85   .23   30
.90   .12  -50
% python potential.py < charges.txt
```

Визуализация значений потенциалов, созданных набором зарядов

модуля stddraw, но тип данных Picture обеспечивает более точный контроль цвета каждого пикселя на экране. Тот же фундаментальный подход применим и во многих других случаях, примеры которых можно найти на сайте книги.

Рассмотрим кратко код программы potential.py, поскольку он иллюстрирует абстракцию данных и объектно-ориентированное программирование. Мы хотим получить изображение, демонстрирующее взаимодействие заряженных частиц, и наш код точно отражает процесс создания этого изображения. Для изображения используется объект типа Picture (манипулирующий объектами типа Color), а для частиц используются объекты типа Charge. Когда необходима информация о заряде, мы непосредственно вызываем соответствующий метод объекта типа Charge; если мы хотим создать цвет, то используем конструктор типа Color; а если хотим установить цвет пикселя, то непосредственно задействуем соответствующий метод типа Picture. Эти типы данных разработаны независимо, но их совместное использование в одном клиенте осуществляется просто и естественно. Далее мы рассмотрим еще несколько примеров, иллюстрирующих широкую доступность абстракции данных, а также добавим еще несколько полезных типов данных в нашу простую модель программирования.

Ввод и вывод еще раз. В разделе 1.5 уже было описано, как читать и выводить числа и текст с помощью книжного модуля stdio. Вы, конечно, оценили удобство этих механизмов передачи информации в программы и ее вывода. Одна из причин их удобства в том, что "стандартные" соглашения делают их доступными в любом месте в пределах программы. Но у этих соглашений есть один недостаток — они ставят нас в зависимость от механизма пересылки и перенаправления операционной системы, обеспечивающей доступ к файлам, а также ограничивают нас работой только с одним входным и одним выходным файлом. Объектно-ориентированное программирование позволяет определять механизмы, подобные используемым модулем stdio, но способные работать с несколькими потоками ввода и вывода в пределах одной программы.



Общая панорама программы Python (еще раз)

Программа 3.1.8. Визуализация электрического потенциала (potential.py)

```
import stddraw
import stdio
import stdarray
from charge import Charge
from color import Color
from picture import Picture

n = stdio.readInt()
charges = stdarray.create1D(n)
for i in range(n):
    x0 = stdio.readFloat()
    y0 = stdio.readFloat()
    q0 = stdio.readFloat()
    charges[i] = Charge(x0, y0, q0)

pic = Picture()
for col in range(pic.width()):
    for row in range(pic.height()):
    x = 1.0 * col / pic.width()
    y = 1.0 * row / pic.height()
    v = 0.0
    for i in range(n):
        v += charges[i].potentialAt(x, y)
    v = (255 / 2.0) + (v / 2.0e10)
    if   v < 0:   gray = 0
    elif v > 255: gray = 255
    else:         gray = int(v)
    color = Color(gray, gray, gray)
    pic.set(col, pic.height()-1-row, color)

stddraw.setCanvasSize(pic.width(), pic.height())
stddraw.picture(pic)
stddraw.show()
```

`n`	Количество зарядов
`charges[]`	Массив зарядов
`x0, y0`	Позиция заряда
`q0`	Значение заряда
`col, row`	Позиция пикселя
`x, y`	Точка в единичном квадрате
`gray`	Масштабированное значение потенциала
`color`	Цвет пикселя

Эта программа читает значения с устройства стандартного ввода, создает массив заряженных частиц, устанавливает для каждого пикселя в изображении значение полутонового цвета, пропорциональное общему значению потенциала, созданного заряженными частицами в соответствующих точках, а затем выводит полученное изображение.

В этом разделе мы определяем типы данных `InStream` и `OutStream` для потоков ввода и вывода соответственно. Как обычно, файлы `instream.py` и `outstream.py` следует сделать доступными для Python, поместив их в тот же каталог, что и клиентский код, или использовав механизм путей вашей операционной системы (см. вопросы и ответы в конце раздела 2.2).

Типы `InStream` и `OutStream` должны обеспечить гибкость, необходимую для решения множества общих задач обработки данных в пределах программ Python. Вместо того чтобы ограничиваться только одним потоком ввода и одним потоком вывода, мы сможем создать несколько объектов каждого типа данных, связав потоки с различными отправителями и получателями данных. Мы также получим гибкость присвоения переменным ссылок на такие объекты, передачи их в аргументах и возвращения из функций или методов, создания их массивов и других манипулирований, подобно объектам любого другого типа. После изучения API мы рассмотрим несколько примеров их использования.

Тип данных потока ввода. Наш тип данных `InStream`, определенный в модуле `instream.py`, является более общей версией операций чтения по сравнению с предоставляемыми модулем `stdio`; он обеспечивает чтение чисел и текста из файлов, с веб-сайтов и из потока стандартного ввода. Его API представлены ниже.

API для нашего типа данных `InStream (instream.py)`

Операция	Описание
`InStream(filename)`	Новый поток ввода, инициализированный именем файла (если аргумента нет, подразумевается стандартный ввод)
	Методы, читающие лексемы из стандартного ввода
`s.isEmpty()`	`s` пуст? (он состоит исключительно из отступа?)
`s.readInt()`	Читает лексему из `s`, преобразует ее в целое число и возвращает
`s.readFloat()`	Читает лексему из `s`, преобразует ее в вещественное число и возвращает
`s.readBool()`	Читает лексему из `s`, преобразует ее в логическое значение и возвращает
`s.readString()`	Читает лексему из `s`, преобразует ее в строку и возвращает
	Методы, читающие строки из стандартного ввода
`s.hasNextLine()`	Есть ли у `s` следующая строка?
`s.readLine()`	Читает следующую строку из `s` и возвращает ее как строку

Примечание 1. Лексема (token) — это максимальная последовательность значащих символов (не отступов).

Примечание 2. Ведет себя как стандартный ввод; методы `readAll()` также поддерживаются (см. раздел 1.5).

Этот подход позволяет обрабатывать по нескольку файлов в пределах той же программы. Кроме того, это позволяет непосредственно обращаться ко всему веб как к потенциальному источнику ввода для наших программ. Например,

позволяет обрабатывать данные, предоставляемые и поддерживаемые кем-то еще. Такие файлы можно найти в веб повсюду. Ученые ныне регулярно публикуют файлы данных с результатами экспериментов от последовательностей генов до структур белков, спутниковых фотографий и результатов астрономических наблюдений; финансовые компании, такие как фондовые биржи, также регулярно публикуют в веб подробную информацию о состоянии рынка и финансов; правительства публикуют результаты выборов и т.д.. Теперь вы можете составлять программы Python, способные непосредственно читать такие файлы. Тип данных `InStream` обеспечивает высокую гибкость и позволяет использовать множество источников данных, ставших теперь доступными.

Тип данных потока вывода. Наш тип данных `OutStream`, определенный в модуле `outstream.py`, является более общей версией операций вывода по сравнению с предоставляемыми модулем `stdio`, он обеспечивает вывод строк в несколько потоков, включая стандартный вывод и файлы. API снова определяет те же методы, что и аналог в модуле `stdio`. Вы указываете имя используемого для вывода файла как аргумент конструктора. Объект типа `OutStream` интерпретирует эту строку как имя нового файла на вашем компьютере и посылает свой вывод туда. Если не предоставить аргумент конструктору, то вы получите стандартный вывод.

API для нашего типа данных `OutStream` (`outstream.py`)

Операция	Описание
`OutStream(filename)`	Новый поток вывода, пишущий в файл `filename` (если аргумента нет, подразумевается стандартный вывод)
`out.write(x)`	Выводит `x` в `out`
`out.writeln(x)`	Выводит `x` в `out`, а затем новую строку (стандартно `x` — пустая строка)
`out.writef(fmt, arg1, ...)`	Выводит аргументы `arg1...` в `out` согласно строке формата `fmt`

Конкатенация файлов и фильтрация. Программа 3.1.9 (`cat.py`) является типичным клиентом типов `InStream` и `OutStream`. Она использует несколько потоков ввода для конкатенации нескольких входных файлов в один выходной файл. В некоторых операционных системах есть такая команда, как `cat`, реализующая эту функцию. Однако осуществляющая тоже действие программа Python, возможно, окажется полезней, поскольку она способна *фильтровать* входные файлы различными способами: мы могли бы игнорировать неподходящую информацию, изменять формат или выбирать только некоторые данные. Давайте рассмотрим один из примеров такой обработки, а другие вы можете найти в упражнениях.

Анализ экранных данных. Комбинация типа `InStream` (позволяющего создавать потоки ввода из любых страниц в веб) и типа `str` (предоставляющего мощнейшие инструменты для обработки текстовых строк) открывает весь веб для непосредственного доступа нашим программам Python независимо

```
...
(GOOG)</h2> <span class="rtq_
exch"><span class="rtq_dash">-</span>
NMS </span><span class="wl_sign">
</span></div></div>
<div class="yfi_rt_quote_summary_rt_top
sigfig_promo_1"><div>
<span class="time_rtq_ticker">
<span id="yfs_l84goog">1,100.62</span>
</span> <span class="down_r time_rtq_
content"><span id="yfs_c63_goog">
...
```

Код HTML из веб

от операционной системы или браузера. Парадигма *анализ экранных данных* (screen scraping) подразумевает извлечение некой информации с веб-страницы программными средствами без необходимости просмотра и поиска ее вручную. Сделав это, мы используем тот факт, что многие веб-страницы представляют собой высоко структурированные текстовые файлы (поскольку они создаются компьютерными программами!). У браузера есть механизм, позволяющий исследовать исходный код, создающий веб-страницу, которую вы просматриваете, а исследовав ее исходный код, зачастую можно выяснять, что она делает.

Программа 3.1.9. Конкатенация файлов (cat.py)

```
import sys
from instream import InStream
from outstream import OutStream

inFilenames = sys.argv[1:len(sys.argv)-1]
outFilename = sys.argv[len(sys.argv)-1]

outstream = OutStream(outFilename)
for filename in inFilenames:
    instream = InStream(filename)
    s = instream.readAll()
    outstream.write(s)
```

`outstream`	Поток вывода
`filename`	Текущее имя файла
`instream`	Текущий поток ввода
`s`	Содержимое `filename`

Эта программа создает выходной файл, имя которого задано последним аргументом командной строки, а содержимое является копией входных файлов, имена которых задаются другими аргументами.

```
% more in1.txt
This is
% more in2.txt
a tiny
test.
```

```
% python cat.py in1.
txt in2.txt out.txt
% more out.txt
This is
a tiny
test.
```

Предположим, необходимо получить в аргументе командной строки торговый символ компании и вывести текущую цену торгов ее акций. Такая информация публикуется в веб финансовыми компаниями и провайдерами услуг Интернета. Например, вы можете найти курс акций компании с символом `goog` по адресу

`http://finance.yahoo.com/q?s=goog`. Подобно множеству веб-страниц, имя кодируется как аргумент (`goog`), и мы можем заменить его любым другим символом, чтобы получить веб-страницу с финансовой информацией для любой другой компании. Кроме того, подобно многим другим файлам в веб, это текстовый файл, написанный на языке HTML. С точки зрения программы Python это значение типа `str`, доступное через объект `InStream`. Для просмотра исходного кода этого файла вы можете использовать возможности браузера или вызов `python cat.py "http://finance.yahoo.com/q?s=goog" mycopy.txt`, чтобы поместить исходный код в локальный файл `mycopy.txt` на вашем компьютере (хотя никакой реальной потребности в этом нет). Теперь предположим, что `goog` в настоящее время идет по 1 100,62 доллара. Если вы будете искать в исходном коде этой страницы строку `'1,100.62'`, то найдете курс акций, заключенный в некоем коде HTML. Чтобы выяснить нечто о контексте, в котором представлена цена, необходимо иметь знания о языке HTML. В данном случае, как можно заметить, курс акций заключен между подстроками `<span id="yfs_l84goog">` и `</span>`.

Используя метод `find()` типа данных `str` и разделение строк, совсем не сложно вычленить эту информацию, как иллюстрирует программа 3.1.10 (`stockquote.py`). Эта программа зависит от формата веб-страницы, используемой сайтом `http://finance.yahoo.com`; если этот формат изменится, то программа `stockquote.py` работать не будет. Действительно, к тому времени, когда вы будете читать эту страницу, формат, вероятно, изменится. Однако внесение соответствующих изменений вряд ли окажется трудным. Вы можете сами усовершенствовать программу `stockquote.py` любыми способами. Например, можно периодически извлекать курс акций и рисовать его график, вычислить скользящее среднее значение или сохранить результаты в файле для последующего анализа. Конечно, та же методика работает и для других источников данных, широко доступных в веб. Несколько примеров тому можно найти в упражнениях в конце этого раздела и на сайте книги.

Извлечение данных. Способность поддерживать несколько потоков ввода и вывода обеспечивает гибкость в обработке больших объемов данных, получаемых из множества источников. Рассмотрим еще один пример. Предположим, у ученого или финансового аналитика есть большой объем данных в формате программы электронной таблицы. Как правило, такие электронные таблицы имеют относительно большое количество рядов и относительно небольшое количество столбцов. Вас вряд ли будут интересовать все данные в электронной таблице, возможно, лишь некоторые из столбцов. Некоторые из вычислений можно сделать и в пределах программы электронной таблицы (в конце концов, в этом ее цель), но у нее, конечно, нет той гибкости, которую может реализовать программа Python. Один из способов решения этой задачи подразумевает экспорт данных электронной таблицы в текстовый файл с использование некоего специального символа для разграничения столбцов и последующего составления программы Python, читающей этот файл из потока ввода. В качестве разделителей обычно используют запятые: ряд

записывается в одну строку, а запятые отделяют элементы столбцов. Такие файлы известны как *значения, разделяемые запятыми* (comma-separated-value) или `.csv`. Метод `split()` типа данных Python `str` позволяет читать файл построчно и извлекать необходимые данные. Несколько примеров этого подхода будет представлено позже. Программа 3.1.11 (`split.py`), являющаяся клиентом типов `InStream` и `OutStream`, идет на шаг дальше: она создает несколько потоков вывода и формирует по одному файлу для каждого столбца.

Программа 3.1.10. Анализ экранных данных для котировки акций (`stockquote.py`)

```
import sys
import stdio
from instream import InStream

def _readHTML(stockSymbol):
    WEBSITE = 'http://finance.yahoo.com/q?s='
    page = InStream(WEBSITE + stockSymbol)
    html = page.readAll()
    return html

def priceOf(stockSymbol):
    html  = _readHTML(stockSymbol)
    trade = html.find('yfs_l84', 0)
    beg   = html.find('>', trade)
    end   = html.find('</span>', beg)
    price = html[beg+1:end]
    price = price.replace(',', '')
    return float(price)

def main():
    stockSymbol = sys.argv[1]
    price = priceOf(stockSymbol)
    stdio.writef('%.2f\n', price)

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

`page`	**Поток ввода**
`html`	**Содержимое** `page`
`trade`	**Индекс** `yfs_l84`
`beg`	**>** **после индекса** `trade`
`end`	**** **после индекса** `to`
`price`	**Текущая цена**

Эта программа получает как аргумент командной строки биржевой торговый символ и выводит на стандартное устройство вывода текущий курс акций, сообщаемый веб-сайтом `http://finance.yahoo.com`. Для разделения строк она использует методы `find()` и `replace()` типа `str`.

```
% python stockquote.py goog
1100.62
% python stockquote.py adbe
70.51
```

Эти примеры убедительно демонстрируют удобство работы с текстовыми файлами, несколькими потоками ввода и вывода, а также непосредственным доступом к веб-страницам. Веб-страницы написаны на языке HTML и доступны любой программе, способной читать строки. Люди зачастую используют текстовые форматы, такие как файлы `.csv`, а не форматы данных, специфичные для конкретных приложений, ведь это позволяет многим другим людям получать доступ к данным, используя такие простые программы, как `split.py`.

Программа 3.1.11. Разделение файла (`split.py`)

```
import sys
import stdarray
from instream import InStream
from outstream import OutStream

basename = sys.argv[1]
n = int(sys.argv[2])

instream = InStream(basename + '.csv')
out = stdarray.create1D(n)

for i in range(n):
    out[i] = OutStream(basename + str(i) + '.txt')

while instream.hasNextLine():
    line = instream.readLine()
    fields = line.split(',')
    for i in range(n):
        out[i].writeln(fields[i])
```

`basename`	Имя исходного файла
`n`	Количество полей
`instream`	Поток ввода
`out[]`	Потоки вывода
`line`	Текущая строка
`fields[]`	Значения в текущей строке

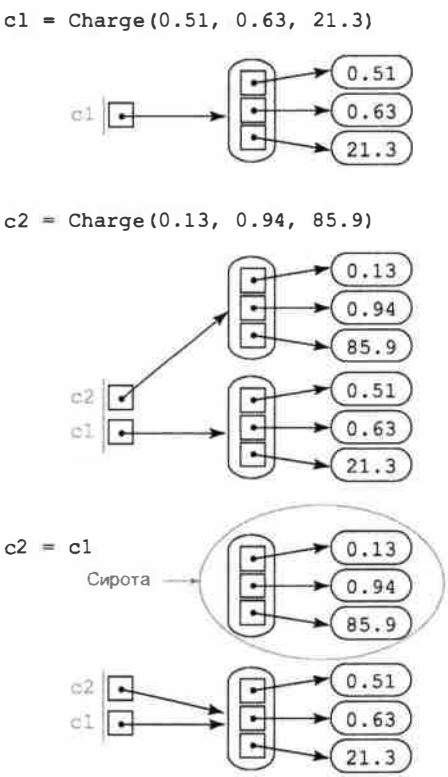
Эта программа использует несколько потоков вывода для разделения файла `.csv` на отдельные файлы, по одному для каждого из полей, разделенных запятыми. Она получает в аргументах командной строки строку `basename` и целое число `n`, а затем разделяет файл по имени `basename.csv`, поля которого разделены запятыми, на `n` файлов с именами `basename0.txt`, `basename1.txt` и т.д.

```
% more djia.csv
...
31-Oct-29,264.97,7150000,273.51
30-Oct-29,230.98,10730000,258.47
29-Oct-29,252.38,16410000,230.07
28-Oct-29,295.18,9210000,260.64
25-Oct-29,299.47,5920000,301.22
24-Oct-29,305.85,12900000,299.47
23-Oct-29,326.51,6370000,305.85
22-Oct-29,322.03,4130000,326.51
21-Oct-29,323.87,6090000,320.91
...
```

```
% python split.py djia 4

% more djia2.txt
...
7150000
10730000
16410000
9210000
5920000
12900000
6370000
4130000
6090000
...
```

Управление памятью. Теперь, рассмотрев несколько примеров объектно-ориентированных типов данных (`str`, `Charge`, `Color`, `Picture`, `InStream` и `OutStream`), а также способные использовать их клиентские программы, можно подробнее обсудить проблемы поддержки таких программ в языке Python. Язык Python по большей части защищает начинающего программиста от необходимости знать их подробности, но для написания правильной и эффективной объектно-ориентированной программы необходимо иметь некоторое представление о происходящем в системе.

Объекты в языке Python создаются при вызове *конструктора*. Каждый раз, когда создается объект, Python резервирует для него область машинной памяти, но освобождается ли для последующего использования занимаемая им область памяти, когда объект удаляется? Кратко рассмотрим этот вопрос.



Осиротевший объект

Осиротевшие объекты. Возможность связать переменную с другим объектом позволяет программе создать объект, на который вообще больше нельзя сослаться. Рассмотрим, например, три оператора присвоения, приведенные на рисунке ниже. Мало того, что после третьего оператора присвоения переменные `c1` и `c2` ссылаются на тот же объект типа `Charge` (с координатами 0,51, 0,63 и зарядом 21,3), но также нет больше никакой ссылки на объект типа `Charge`, первоначально созданный для инициализации переменной `c2`. Единственной ссылкой на этот объект было значение переменной `c2`, а поскольку эта ссылка оказалась переписанной, не осталось никакого способа обратиться к объекту снова. Таким образом, объект остался *сиротой* (orphan). Объекты могут также стать сиротами при выходе переменных из области видимости. Программисты Python обращают не много внимания на осиротевшие объекты, поскольку система автоматически освобождает занимаемую ими область многократно используемой памяти.

Управление памятью. Программы имеют тенденцию создавать огромные количества объектов, но одновременно использовать лишь немногие из них. Таким

образом, языки программирования и системы нуждаются в механизмах создания объектов (и резервирования памяти) и в механизмах удаления объектов (и освобождения памяти), когда они становятся сиротами. Большинство систем программирования самостоятельно заботится о резервировании памяти для переменных, когда они появляются, и освобождении памяти, когда они выходят из области видимости. Управление памятью объектов куда сложней: Python знает, что резервировать память для объекта нужно при его создании, но он не может знать точно, когда освобождать связанную с объектом память, поскольку именно динамика выполнения программы определяет, когда объект станет сиротой и потребует удаления. Система никак не может предсказать то, что сделает программа, поэтому вынуждена контролировать работу программы и принимать соответствующие меры.

Утечки памяти. Во многих языках (таких, как C и C++) программист сам несет ответственность и за резервирование, и за освобождение памяти. Но это и утомительно, и ведет к ошибкам. Предположим, например, что программа освобождает память объекта, а затем (возможно, намного позже) продолжает обращаться к нему. Тем временем система уже, возможно, перераспределила эту память для другого объекта, и в результате все может закончиться плохо. Еще одна довольно коварная проблема возникает тогда, когда программист не удосуживается гарантировать освобождение осиротевших объектов. Эта проблема известна как *утечка памяти* (memory leak), поскольку она приводит к устойчивому увеличению объема памяти, занятой осиротевшими объектами (а потому недоступной для использования). В результате производительность падает, как будто память вытекает из вашего компьютера. Через некоторое время компьютер придется перезагрузить, поскольку он будет работать все медленней и хуже. Причиной такого поведения обычно является утечка памяти в одном из ваших приложений.

Сбор мусора. Одним из наиболее значительных достоинств языка Python является автоматическое управление памятью. Идея заключается в том, чтобы освободить программистов от ответственности за управление памятью за счет отслеживания осиротевших объектов и возвращения используемой ими памяти в пул свободной памяти. Подобное обслуживание памяти известно как *сбор мусора* (garbage collection), и система типов Python реализует ее эффективно и автоматически. Сбор мусора — довольно старая идея, но до сих пор еще не прекратился спор о том, компенсируются ли дополнительные затраты по автоматическому сбору мусора удобством, связанным с отсутствием необходимости заботиться об управлении памятью. Тем не менее очень многие (включая программистов Python и Java, например) предпочитают пользоваться этим преимуществом и не волноваться об утечках памяти и других проблемах.

Для справки резюмируем типы данных, рассмотренные в этом разделе. Мы выбрали эти примеры, чтобы помочь вам понять основные свойства типов данных и объектно-ориентированного программирования.

Тип данных — это набор возможных значений и операций, допустимых для этих значений. Встроенные числовые типы обладают небольшим и довольно простым набором значений. Цвета, изображения, строки и потоки ввода-вывода — это высокоуровневые типы данных, реализующие широко популярные абстракции данных. *Вы не обязаны знать, как реализован тип данных, чтобы использовать его.* Каждый тип данных (в стандартных модулях Python и в модулях расширения их сотни, и скоро вы научитесь создавать собственные типы) характеризуется API (интерфейсом прикладных программ), предоставляющим информацию, необходимую для их использования. Клиентская программа создает объекты, содержащие значения типа данных, и вызывает методы, манипулирующие этими значениями. Мы составляем клиентские программы, использующие те же простые выражения и управляющие структуры, что и в главах 1 и 2, но теперь можем работать с обширным разнообразием типов данных, а не только встроенных. По мере приобретения опыта вы найдете, что эта способность открывает новые горизонты в программировании.

Типы данных этого раздела

Тип данных	Описание
`str`	Последовательность символов
`Charge`	Электрический заряд
`Color`	Цвет
`Picture`	Цифровое изображение
`InStream`	Поток ввода
`OutStream`	Поток вывода

Наш пример `Charge` продемонстрировал возможность приспособления одного или нескольких типов данных к потребностям вашего приложения. Возможность сделать это — серьезное преимущество, а также тема следующего раздела. Будучи правильно разработанными и реализованными, типы данных упрощают создание клиентских программам и облегчают их поддержку по сравнению с эквивалентными программами, не использующими абстракцию данных. Клиентские программы данного раздела — прямое доказательство этого заявления. Кроме того, как будет продемонстрировано в следующем разделе, для реализации простого типа данных достаточно базовых навыков программирования, которые вы уже получили. В частности, при реализации большого и сложного приложения необходимо понять, каковы его данные и выполняемые с ними операции, а затем уже можно составить программы, непосредственно отражающие это понимание. Как только вы научитесь делать это, у вас сразу может вопрос: а как программисты раньше разрабатывали большие программы, не используя абстракцию данных?

Вопросы и ответы

Что будет, если вызвать метод, не определенный для данного объекта?

Во время выполнения Python передаст сообщение об ошибке `AttributeError`.

Почему мы можем использовать функцию `stdio.writeln(x)` вместо функции `stdio.writeln(str(x))` при выводе объекта `x`, не являющегося строкой?

Для удобства всякий раз, когда необходим строковый объект, функция `stdio.writeln()` автоматически вызывает встроенную функцию `str()`.

Я обратил внимание, что программа 3.1.8 (`potential.py`) вызывает функцию `stdarray.create1D()` для создания массива объектов типа `Charge`, но ей предоставляется только один аргумент (желаемое количество элементов). Почему функция `stdarray.create1D()` не требует предоставления двух аргументов: желаемого количества элементов и исходных значений элементов?

Если исходные значения не определены, функции `stddarray.create1D()` и `stdarray.create2D()` используют специальное значение `None`, не ссылающееся ни на какой объект. Непосредственно после вызова функции `stdarray.create1D()` программа `potential.py` изменяет каждый элемент массива, в результате они ссылаются на новые объекты `Charge`.

Можно ли вызвать метод с литералом или другим выражением?

Да, с точки зрения клиента при вызове метода можно использовать любое выражение. Когда Python вызывает метод, он вычисляет результат выражения и вызывает получающий его метод. Например, вызов `'python'.upper()` возвращает строку `'Python'`, а `(3 ** 100).bit_length()` возвращает `159`. Но с целочисленными литералами необходимо быть внимательным: например, вызов `1023.bit_length()` закончится ошибкой `SyntaxError`, поскольку Python интерпретирует `1023.` как литерал с плавающей точкой; вместо этого следует использовать синтаксис `(1023).bit_length()`.

Можно ли объединить несколько вызовов строковых методов в одном выражении?

Да. Например, выражение `s.strip().lower()` работает как надо. В результате получится новая строка, являющаяся копией `s`, но без предваряющих и завершающих пробелов, со всеми символами, переведенными в нижний регистр. Это работает, потому что (1) каждый из методов возвращает свой результат как строку и (2) точечный оператор имеет левосторонний порядок, поэтому Python вызывает методы слева направо.

Почему красный, зеленый и синий, а не красный, желтый и синий?

Теоретически работали бы любые три цвета, содержащие некий объем каждого основного цвета, но развились две разные цветовые модели: RGB создает хорошие цвета на экранах (телевизоров, компьютеров и камер), а CMYK обычно используется в полиграфии (см. упр. 1.2.28). Система CMYK (синий, красный, желтый и черный) действительно включает желтый цвет. Обе системы соответствуют своим целям, поскольку при печати чернила поглощают свет, при наложении двух чернил разных цветов больше света поглощается и меньше отражается. Экраны, напротив, испускают свет; поэтому два пикселя разных цветов испускают больше света.

Не возникнет ли проблем при создании тысяч объектов типа `Color`, как в программе 3.1.5 (`grayscale.py`)? Не кажется ли это расточительным?

Все конструкции языка программирования имеют цену. В данном случае цена разумна, поскольку время, необходимое для создания объектов `Color`, ничтожно по сравнению со временем, необходимым для фактического получения изображений.

Может ли у типа данных быть два метода (или конструктора) с одинаковым именем, но разным количеством аргументов?

Нет, подобно функциям, не может быть двух методов (или конструкторов) с тем же именем. Как и функции, методы (и конструкторы) могут использовать необязательные аргументы со стандартными значениями. Именно так тип данных `Picture` создает иллюзию наличия двух конструкторов.

Упражнения

3.1.1. Составьте программу, которая получает из командной строки аргумент `w` типа `float` и создает четыре объекта типа `Charge`, находящиеся на расстоянии `w` в четырех направлениях по осям от точки (0.5, 0.5), а затем выводит потенциал в точке (0,25, 0,5).

3.1.2. Составьте программу, получающую из командной строки три целых числа от 0 до 255, представляющие значения красного, зеленого и синего цветов, а затем создает и отображает объект типа `Picture` размером 256×256 этого цвета.

3.1.3. Измените программу 3.1.3 (`albersquares.py`) так, чтобы она получала из командной строки девять аргументов, определяющих три цвета, а затем рисовала шесть квадратов Альберса, у которых большие квадраты имеют введенные цвета, а малые квадраты — цвета, отличные от них.

3.1.4. Составьте программу, получающую в аргументе командной строки имя файла полутонового изображения и использующая модуль `stddraw` для рисования графика частоты вхождений каждой из 256 полутоновых оттенков.

3.1.5. Составьте программу, получающую в аргументе командной строки имя файла изображения и переворачивающую изображение горизонтально.

3.1.6. Составьте программу, которая получает в аргументе командной строки имя файла изображения и создает три изображения: одно только с красными компонентами, одно только с зелеными и одно только с синими.

3.1.7. Составьте программу, которая получает в аргументе командной строки имя файла изображения и выводит координаты пикселя левого нижнего и верхнего правого углов наименьшей рамки (прямоугольник, параллельный осям x и y), содержащей все не белые пиксели.

3.1.8. Составьте программу, которая получает в аргументе командной строки имя файла изображения и координаты прямоугольника (в пикселях) внутри этого изображения. Прочитайте из стандартного потока ввода список значений типа `Color` (представленных тремя целыми числами). Программа служит фильтром, она выводит те значения `Color`, цвет которых совместим со всеми пикселями в прямоугольнике (как фон и передний план). (Такой фильтр применяется при выборе цвета текста, нанесенного поверх изображения.)

3.1.9. Составьте функцию `isValidDNA()`, получающую как аргумент строку и возвращающую значение `True`, если и только если она состоит исключительно из символов `A`, `C`, `T` и `G`.

3.1.10. Составьте функцию `complementWC()`, получающую как аргумент строку ДНК и возвращающую для нее *дополнение Уотсона–Крика* (Watson-Crick complement), заменив `A` на `T`, `C` на `G`, и наоборот.

3.1.11. Составьте функцию `palindromeWC()`, получающую как аргумент строку ДНК и возвращающую значение `True`, если строка является комплементарным палиндромом Уотсона–Крика, и значение `False` в противном случае. *Комплементарный палиндром Уотсона–Крика* (Watson-Crick complemented palindrome) — это строка ДНК, обратная ее дополнению Уотсона–Крика.

3.1.12. Составьте программу проверки допустимости номера ISBN (см. упр. 1.3.33) с учетом возможности вставки дефисов в произвольных местах.

3.1.13. Что выводит следующий фрагмент кода?

```
s = 'Hello World'
s.upper()
s[6:11]
stdio.writeln(s)
```

Решение: `'Hello World'`. Строковые объекты неизменяемы — строковый метод возвращает новый объект типа `str` с соответствующим значением, но не изменяет значение строки, использованной для его вызова. Поэтому данный код игнорирует возвращенные объекты и выводит только исходную строку. Для изменения строки `s` используйте операторы `s = s.upper()` и `s = s[6:11]`.

3.1.14. Строка `s` является *циклическим сдвигом* (circular shift) строки `t`, если она получается в результате циклического сдвига символов на любое количество позиций. Например, `ACTGACG` — это циклический сдвиг `TGACGAC`, и наоборот. Обнаружение этого условия важно в исследовании последовательностей генов. Составьте функцию проверки, не являются ли две предоставленные строки `s` и `t` циклическим сдвигом друг друга. *Подсказка*: решение состоит из одной строки с оператором `in` и конкатенацией строк.

3.1.15. Составьте фрагмент кода, который получает строку URL веб-сайта и определяет тип его домена. Например, типом домена для URL сайта нашей книги `http://introcs.cs.princeton.edu/python` является `edu`.

3.1.16. Составьте функцию, получающую как аргумент имя домена и возвращающую обратный домен (порядок строк между точками изменен

на обратный). Например, для `introcs.cs.princeton.edu` обратный домен будет `edu.princeton.cs.introcs`. Это вычисление полезно при анализе веб-журнала. (См. упр. 4.2.33.)

3.1.17. Что возвращает следующая рекурсивная функция?

```
def mystery(s):
    n = len(s)
    if (n <= 1): return s
    a = s[0 : n//2]
    b = s[n//2 : n]
    return mystery(b) + mystery(a)
```

3.1.18. Составьте версию программы 3.1.1 (`potentialgene.py`), находящую все потенциальные гены в длинной строке ДНК. Добавьте аргумент командной строки, позволяющий пользователю определять минимальную длину потенциального гена.

3.1.19. Составьте программу, которая получает как аргументы строки начала и конца, а выводит все подстроки данной строки, которые начинаются с первого аргумента и кончаются вторым, в противном случае не выводит ничего. *Примечание*: будьте особенно осторожны с перекрытиями!

3.1.20. Составьте фильтр, который читает из потока ввода текст и выводит его в поток вывода, удалив любые строки, состоящие только из отступов.

3.1.21. Модифицируйте программу 3.1.8 (`potential.py`) так, чтобы она получала из командной строки целое число `n` и создавала в единичном квадрате `n` случайных объектов типа `Charge` со значениями, случайно присвоенными согласно Гауссову распределению со средним 50 и среднеквадратичным отклонением 10.

3.1.22. Модифицируйте программу 3.1.10 (`stockquote.py`) так, чтобы она получала из командной строки несколько символов.

3.1.23. Файл примера `djia.csv`, использованный для программы 3.1.11 (`split.py`), содержит дату, верхнюю цену, объем и нижнюю цену, а выводит индекс Доу-Джонса для каждого дня с начала наблюдений. Загрузите этот файл с сайта книги и составьте программу, составляющую график цен и объемов с частотой, получаемой из командной строки.

3.1.24. Составьте программу `merge.py`, получающую из командной строки разделитель и произвольное количество имен файлов; конкатенируйте соответствующие строки каждого файла, разделенные разделителем, а затем

выведите результат на стандартный вывод, выполнив таким образом операцию, противоположную выполняемой программой 3.1.11 (`split.py`).

3.1.25. Найдите веб-сайт, публикующий текущую температуру в вашей области, и напишите программу `weather.py` для анализа экранных данных, чтобы можно было, введя `python weather.py` и свой индекс, получить прогноз погоды.

Практические упражнения

3.1.26. *Фильтрация изображений.* Составьте модуль `rawpicture.py` с функциями `read()` и `write()`, позволяющими использовать стандартные устройства ввода и вывода. Функция `write()` получает как аргумент объект типа `Picture` и выводит на стандартное устройство изображение, используя следующий формат: если изображение имеет размер w на h, то выводится w, затем h, затем w*h триплетов целых чисел, представляющих значения цветов пикселей в порядном порядке. Функция `read()` не получает никаких аргументов и возвращает объект типа `Picture`, создаваемый ею при чтении со стандартного ввода изображения в только что описанном формате. *Примечание*: результат фильтрации изображения расходует намного больше дискового пространства, чем само изображение, — стандартные форматы используют сжатие такой информации, чтобы она не занимала много места.

3.1.27. *Шифр Камасутры.* Составьте фильтр `KamaSutra`, получающий как аргумент командной строки две строки (*ключевые* строки), читающий стандартный ввод и заменяющий каждый символ так, как определено ключевыми строками, и выводит результат на стандартный вывод. Эта операция — основа одной из самых ранее известных криптографических систем. Ключевые строки должны быть равной длины и содержать все символы, поступающие со стандартного ввода. Например, если ввод осуществляется только прописными буквами, можно использовать ключи `THEQUICKBROWN` и `FXJMPSVRLZYDG`, чтобы получить следующую таблицу:

```
T H E Q U I C K B R O W N
F X J M P S V L A Z Y D G
```

Таким образом, при копировании ввода в вывод мы должны заменять `F` на `T`, `T` на `F`, `H` на `X`, `X` на `H` и т.д. Сообщение кодируется заменой каждого символа его парой. Например, сообщение `MEET AT ELEVEN` превращается в `QJJF BF JKJCJG`. Для чтения сообщения получатель должен использовать те же ключи.

3.1.28. *Проверка надежности пароля.* Составьте функцию, получающую как аргумент строку и возвращающую значение `True`, если она удовлетворяет следующему условию, и значение `False` в противном случае:

Длина не менее восьми символов.

Содержит по крайней мере одну цифру (0–9).

Содержит по крайней мере одну прописную букву.

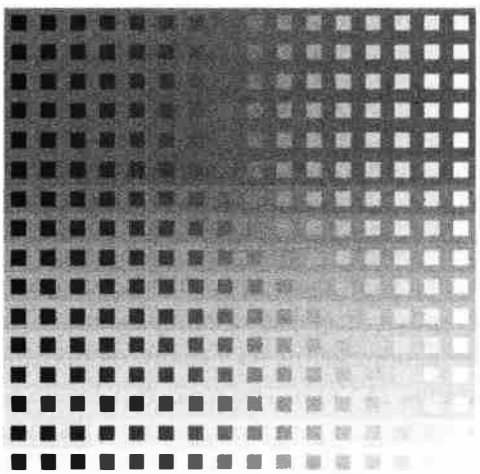
Содержит по крайней мере одну строчную букву.

Содержит по крайней мере один символ, не являющийся ни буквой, ни цифрой.

Такие проверки общеприняты для паролей в веб.

3.1.29. *Визуализация звука.* Составьте программу, использующую модуль `stdaudio` и тип `Picture` для создания двумерной цветовой визуализации звукового файла по мере его проигрывания. Проявите творческий подход!

3.1.30. *Исследование цветов.* Составьте программу, отображающую исследуемые цвета в 256 квадратах Альберса, где синий цвет переходит в белый сверху вниз рядами, а для внутренних квадратов использован полутон (оттенки серого от черного до белого столбцами справа налево).



Исследование цветов

3.1.31. *Энтропия. Энтропия Шэннона* измеряет информационное содержимое входной строки и играет ключевую роль в сжатии данных и теории информации. Если есть строка из n символов, то f_c будет частотой вхождения

символа c, а значение $p_c = f_c / n$ будет оценкой вероятности наличия символа c в случайной строке. Энтропия определяется как сумма значений $-p_c \log_2 p_c$ по всем присутствующим в строке символам. Энтропия измеряет *информационное содержимое* (information content) строки: если все символы распределены равномерно, энтропия имеет минимальное значение. Составьте программу, вычисляющую и выводящую на стандартное устройство вывода энтропию строки, прочитанной со стандартного ввода. Запустите свою программу для веб-страницы, которую вы регулярно читаете, которую вы написали, и для генома плодовой мушки, найденного на веб-сайте.

3.1.32. *Минимальный потенциал.* Составьте функцию, получающую как аргумент массив объектов типа `Charge` с положительным зарядом и находящую точку, потенциал в которой находится в пределах 1% от минимального потенциала где-нибудь в единичном квадрате. Составьте клиент проверки, вызывающий эту функцию для вывода координат точки и значение заряда для данных, заданных в тексте, и для случайных зарядов, описанных в упражнении 3.1.21.

3.1.33. *Показ слайдов.* Составьте программу, получающую в аргументах командной строки имена нескольких файлов изображений и отображающих их по очереди (по одному каждые две секунды), используя эффект постепенного изменения к черному, а затем из черного к следующему изображению.

3.1.34. *Плитка.* Составьте программу, получающую как аргумент командной строки имя файла изображения и два целых числа `m` и `n`, а затем создающую плитку `m` на `n` из изображения.

3.1.35. *Фильтр поворота.* Составьте программу, которая получает два аргумента командной строки (имя файла изображения и вещественное число θ) и поворачивает изображение на θ градусов против часовой стрелки. Для поворота копии цвет каждого пикселя (s_i, s_j) в исходном изображении переносится пикселю (t_i, t_j) результирующего, чьи координаты вычисляются по следующим формулам:

$$t_i = (s_i - c_i)\cos\theta - (s_j - c_j)\sin\theta + c_i;$$
$$t_j = (s_i - c_i)\sin\theta + (s_j - c_j)\cos\theta + c_j;$$

где (c_i, c_j) — центр изображения.

3.1.36. *Фильтр скручивания.* Создание эффекта скручивания подобно повороту, за исключением того, что угол изменяется как функция от расстояния к центру. Используйте те же формулы, что и в предыдущем упражнении, но θ вычисляйте как функцию от (s_i, s_j), а именно как $\pi / 256$ расстояний к центру.

3.1.37. Составьте фильтр, как в двух предыдущих упражнениях, создающий эффект волны в результате копирования цвета каждого пикселя (s_i, s_j) исходного изображения в пиксель (t_i, t_j) результирующего, где $t_i = s_i$ и $t_j = s_j + 20\ sin(2\pi s_i / 64)$. Добавьте код, получающий амплитуду и частоту в аргументах командной строки. Поэкспериментируйте с различными значениями этих параметров.

3.1.38. Составьте программу, которая получает как аргумент командной строки имя файла изображения и применяет *фильтр стекло* (glass filter): каждому пикселю p присваивается цвет случайного соседнего пикселя (координаты которого находятся в пределах 5 пикселей от координат p).

3.1.39. *Преобразование.* Изображения в примере `fade.py` плохо совмещаются по вертикали (рот мандрила намного ниже, чем рот Дарвина). Измените программу `fade.py`, добавив преобразование по вертикали, чтобы сделать переход более плавным.

3.1.40 *Кластеры.* Составьте программу, которая получает из командной строки имя файла изображения, а затем создает и отображает с использованием модуля `stddraw` изображение с заполненными кругами, покрывающими подходящие области. Сначала просмотрите изображение, чтобы определить цвет фона (доминирующий цвет, покрывающий более половины пикселей). Используйте поиск вглубь (см. программу 2.4.6) для нахождения непрерывных наборов пикселей, цвет которых совместим с фоном. Ученый мог бы использовать эту программу для изучения таких природных явлений, как птицы в полете или

Примеры применения фильтров

частицы в движении. Возьмите фотографию шаров на бильярдном столе и попробуйте заставить вашу программу выявить шары и их позиции.

3.1.41. *Цифровое увеличение.* Составьте программу `zoom.py`, получающую из командной строки имя файла изображения и три вещественных числа: *s*, *x* и *y*. Программа выводит изображение, увеличенное по сравнению с исходным. Все числа должны быть в пределах от 0 до 1, где *s* интерпретируется как коэффициент масштабирования, а *x* и *y* — как относительные координаты точки, которая должна оказаться в центре результирующего изображения. Используйте эту программу для увеличения цифрового изображении по вашему выбору на своем компьютере.

```
% python zoom.py boy.jpg 1 0.5 0.5
```

© 2014 Жанин Дитц

```
% python zoom.py boy.jpg 0.5 0.5 0.5
```

```
% python zoom.py boy.jpg 0.2 0.48 0.5
```

Цифровое увеличение

3.2. Создание типов данных

В принципе, вполне можно составлять все программы, используя только встроенные типы данных. Но, как мы уже видели в предыдущем разделе, куда удобнее составлять программы с использованием высокоуровневых абстракций. Таким образом, в стандартных модулях Python и модулях расширения определено много других типов данных. Но мы, конечно, не можем ожидать, что в этих модулях определены все мыслимые типы данных, которые кто-либо когда-либо пожелал бы использовать, поэтому необходимо быть в состоянии *определять* собственные типы. Задача этого раздела состоит в том, чтобы объяснить, как создавать собственные типы данных в Python.

Программы этого раздела...

Программа 3.2.1. Заряженная частица (charge.py) 400
Программа 3.2.2. Секундомер (stopwatch.py) 404
Программа 3.2.3. Гистограмма (histogram.py) 406
Программа 3.2.4. Черепашья графика (turtle.py) 409
Программа 3.2.5. Удивительная спираль (spiral.py) 412
Программа 3.2.6. Комплексные числа (complex.py) 417
Программа 3.2.7. Множество Мандельброта (mandelbrot.py) 421
Программа 3.2.8. Биржевая учетная запись (stockaccount.py) 425

Тип данных — это набор возможных значений и операций, допустимых для этих значений. В языке Python мы реализуем тип данных, используя *класс* (class). API определяет операции, необходимые для реализации, но мы вольны сами выбрать любое удобное представление для значений. Реализация типа данных как класса Python не очень отличается от реализации модуля функций как набора функций. Основное различие в том, что мы ассоциируем значения (в виде переменных экземпляра) с методами и что каждый вызов метода ассоциируется с объектом, используемым для его вызова.

Для закрепления фундаментальных концепций начнем с определения класса, реализующего тип данных для заряженных частиц, с которыми мы познакомились в разделе 3.1. Затем мы рассмотрим процесс определения классов на нескольких примерах, от комплексных чисел, используемых при учете, до нескольких программных инструментов, которые будем использовать далее в книге. Полезность в клиентском коде — это доказательство значимости любого типа данных, поэтому мы рассмотрим также несколько клиентов, включая отображающий известное и очаровательное множество Мандельброта.

Процесс определения типа данных — это *абстракция данных* (в отличие от *абстракции функций*, обсуждавшейся в главе 2). Мы сосредоточимся на данных и реализуем операции с ними. *Всякий раз, когда можете четко разделить*

данные и связанные с ними задачи в пределах вычисления, так и поступайте. Моделирование физических объектов и математических абстракций просто и удобно, но истинная степень абстракции данных заключается в том, что она позволяет моделировать нечто, что мы можем определить точно. Приобретя опыт этого стиля программирования, вы увидите, что он позволяет решать задачи программирования любой сложности.

Составные элементы типа данных. Для иллюстрации процесса реализации типа данных как класса Python рассмотрим реализацию типа данных `Charge` из раздела 3.1 во всех подробностях. Теперь, уже имея *клиентские* программы, демонстрирующие удобство применения такого типа данных (программы 3.1.2 и 3.1.8), остановимся на подробностях *реализации*. У каждой разрабатываемой реализации типа данных будут те же составные компоненты, что и в этом примере.

API. Интерфейс прикладных программ — это контракт со всеми клиентами, а потому он является отправной точкой для любой реализации. Чтобы подчеркнуть критическую важность API для реализаций, мы повторяем далее API типа `Charge`. Для реализации типа данных `Charge` как класса Python необходимо определить значения типа данных, составить код инициализации новых объектов типа `Charge` определенными значениями и реализовать два метода, манипулирующих этими значениями. Когда сталкиваешься с проблемой создания совершенно нового типа данных для некого приложения, первым этапом становится разработка API. Это этап *проектирования* (design), рассматриваемый в разделе 3.3. API можно рассматривать и как спецификацию для *использования* типа данных в клиентском коде; но теперь мы рассмотрим ее как спецификацию для *реализации* типов данных.

Класс. В языке Python мы реализуем тип данных как *класс* (class). Согласно соглашению Python, код типа данных помещается в файле с тем же именем, что и класс, но строчными буками и с последующим расширением `.py`. Таким образом, код типа `Charge` мы сохраняем в файле `charge.py`. Для определения класса используется ключевое слово `class`, сопровождаемое именем класса, двоеточием и списком определений методов. Наш класс будет придерживаться соглашения и определит три основные характеристики типа данных: *конструктор* (constructor), *переменные экземпляра* (instance variable) и *методы* (method). Поскольку эти средства взаимосвязаны, хорошая стратегия их обсуждения (чтобы не повторяться) — рассмотреть все три описания, а затем снова рассмотреть их.

API для нашего пользовательского типа данных `Charge`

Операция	Описание
`Charge(x0, y0, q0)`	Новый заряд со значением $q0$ расположен в точке ($x0$, $y0$)
`c.potentialAt(x, y)`	Электрический потенциал заряда c в точке (x, y)
`str(c)`	`'q0 в (x0, y0)'` (строковое представление заряда c)

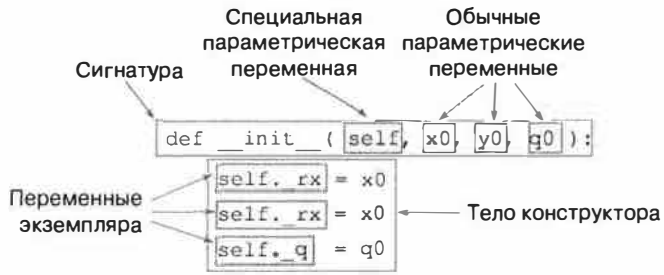
Конструктор. Создает объект определенного типа и возвращает ссылку на него. Следующий пример клиентского кода возвращает правильно инициализированный новый объект типа `Charge`.

```
c = Charge(x0, y0, q0)
```

Язык Python предоставляет общий и гибкий механизм создания объектов, но мы рассмотрим лишь его подмножество, хорошо подходящее нашему стилю программирования. В частности, каждый тип данных в этой книге определяет специальный метод `__init__()`, задача которого заключается в определении и инициализации *переменных экземпляра*, как описано ниже. Двойные подчеркивания перед и после имени свидетельствуют о том, что он "особенный". Вскоре мы встретимся и с другими *специальными методами* (special method).

Когда клиент вызывает конструктор, запускается стандартный процесс Python, создающий новый объект заданного типа, вызывается метод `__init__()`, определяющий и инициализирующий переменные экземпляра, а затем возвращающий ссылку на новый объект. В этой книге мы называем метод `__init__()` *конструктором* типа данных, хотя чисто технически он только часть процесса создания объекта.

Код, приведенный ниже, — это реализация метода `__init__()` для типа `Charge`. Поскольку это метод, его первая строка — сигнатура, состоящая из ключевого слова `def`, имени (`__init__`), списка параметрических переменных и двоеточия. В соответствии с соглашением первая параметрическая переменная имеет имя `self`. *В ходе стандартного процесса создания объекта Python при вызове метода* `__init()__` *значением параметрической переменной* `self` *является ссылка на создаваемый объект.* За специальной параметрической переменной `self` следуют обычные клиентские параметрические переменные. Остальные строки составляют тело конструктора. Наше соглашение в этой книге подразумевает, что метод `__init__()` состоит из кода инициализации вновь созданного объекта, определяющего и инициализирующего переменные экземпляра.



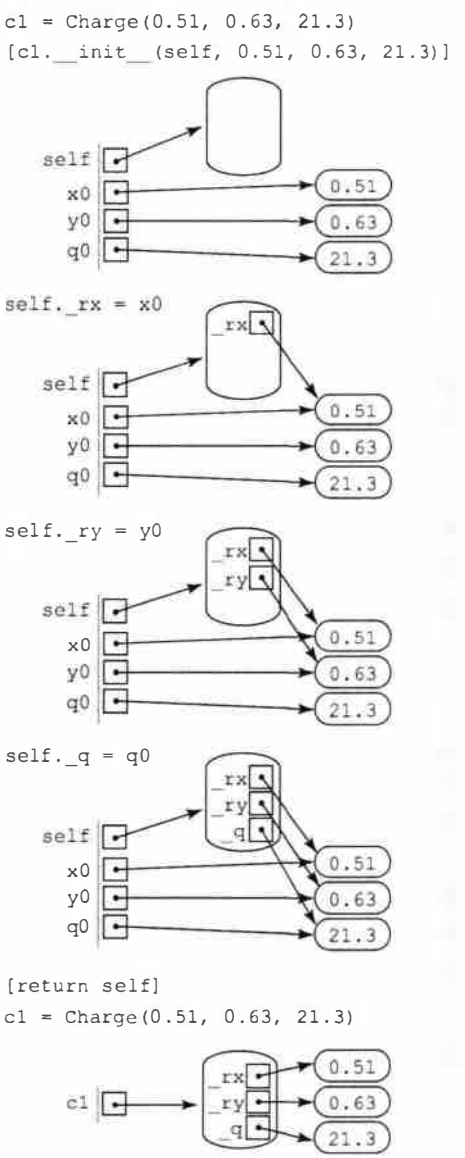
Анатомия конструктора

Переменные экземпляра. Тип данных — это набор возможных значений и операций, допустимых для этих значений. В языке Python *переменные экземпляра*

реализуют значения. Переменная экземпляра принадлежит конкретному *экземпляру* класса, т.е. конкретному объекту. В этой книге принято соглашение определять и инициализировать каждую переменную экземпляра вновь созданного объекта *в конструкторе и только в конструкторе.* Стандартное соглашение Python гласит, что имена переменных экземпляра начинаются с символа подчеркивания. В наших реализациях вы можете просмотреть конструктор и увидеть весь набор переменных экземпляра. Например, реализация метода `__init__()` выше гласит, что у типа `Charge` есть три переменные экземпляра, `_rx`, `_ry` и `_q`. Когда объект создается, параметрическая переменная `self` метода `__init__()` — это ссылка на этот объект. Подобно тому, как мы можем вызвать метод для заряда `c`, используя синтаксис `c.potentialAt()`, мы можем обратиться к переменной экземпляра `self` заряда, используя синтаксис `self._rx`. Таким образом, эти три строки кода в пределах конструктора `__init__()` типа `Charge` определяют и инициализируют переменные `_rx`, `_ry` и `_q` нового объекта.

Детали создания объекта. На схеме распределения памяти представлена точная последовательность событий при создании клиентом нового объекта типа `Charge` следующим кодом:

```
c1 = Charge(0.51, 0.63, 21.3)
```



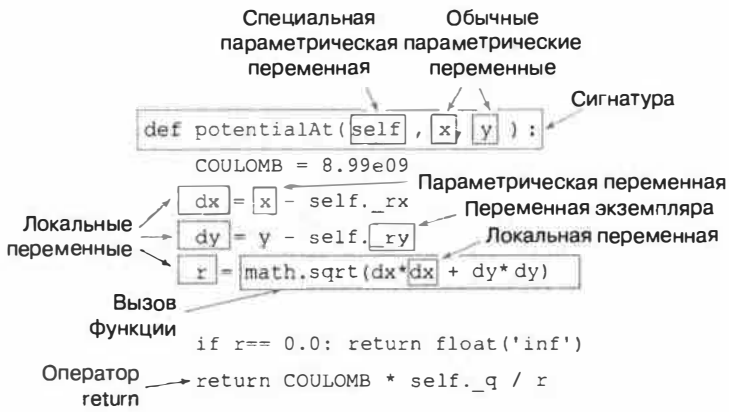
Создание и инициализация объекта

- Python создает объект и вызывает конструктор `__init__()`, инициализировав его параметрическую переменную `self` так, чтобы она ссылалась на вновь созданный объект, его параметрическую переменную `x0` так, чтобы она ссылалась на значение `0.51`, его параметрическую переменную `y0` так, чтобы она ссылалась на значение `0.63`, и его параметрическую переменную `q0` так, чтобы она ссылалась на значение `21.3`.

- Конструктор определяет и инициализирует переменные экземпляра `_rx`, `_ry` и `_q` в пределах вновь созданного объекта, на который ссылается параметр `self`.
- По завершении выполнения конструктора Python автоматически возвращает клиенту ссылку `self` на вновь созданный объект.
- Клиент присваивает эту ссылку переменной `c1`.

По завершении выполнения конструктора `__init__()` параметрические переменные `x0`, `y0` и `q0` выходят из области видимости, но объекты, на которые они ссылались, все еще доступны через переменные экземпляра нового объекта.

Методы. Тип данных — это набор возможных значений и операций, допустимых для этих значений. В языке Python *методы* (method), или *методы экземпляра* (instance method), реализуют операции с типами данных, ассоциированными с конкретным объектом. При определении методов мы составляем код, в точности похожий на применявшийся в главе 2 при определении функций, но с одним (существенным) исключением — методы способны обращаться к переменным экземпляра. Для примера ниже представлен код метода `potentialAt()` нашего типа данных `Charge`. Первая строка — сигнатура метода: ключевое слово `def`, имя метода, имена параметрических переменных в круглых скобках и двоеточие. Первая параметрическая переменная каждого метода имеет имя `self`. Когда клиент вызывает метод, Python автоматически присваивает параметрической переменной `self` ссылку на объект, которым предстоит манипулировать, — это *объект, использованный при вызове метода.* Например, когда клиент вызывает метод `c.potentialAt(x, y)`, значением параметрической переменной `self` будет `c`. Обычные клиентские параметрические переменные (в данном случае `x` и `y`) следуют за специальной параметрической переменной `self`. Остальные строки составляют *тело метода* `potentialAt()`.



Анатомия метода

Переменные в пределах методов. Чтобы понять реализацию метода, очень важно знать, что метод обычно использует три вида переменных:

- переменные экземпляра объекта `self`;
- параметрические переменные метода;
- локальные переменные.

Есть критически важное различие между переменными экземпляра, параметрами и локальными переменными, к которым вы уже привыкли: каждой локальной переменной или параметру в один момент времени соответствует только одно значение, а переменной экземпляра может соответствовать много значений, по одному для каждой из переменных экземпляра из каждого объекта, являющегося экземпляром данного типа данных. В этом нет никакого противоречия, поскольку при каждом вызове метода для этого используется объект, и код реализации вполне может непосредственно обращаться к переменным экземпляра того объекта, используя параметрическую переменную `self`. *Убедитесь, что понимаете различия между этими тремя видами переменных.* Различия приведены в таблице ниже, и это ключ к объектно-ориентированному программированию. В нашем примере метод `potentialAt()` использует для вычисления и возвращения значения переменные экземпляра `_rx`, `_ry` и `_q` объекта, на который ссылается параметр `self`, параметрические переменные `x` и `y`, а также локальные переменные `COULOMB`, `dx`, `dy` и `r`. В качестве примера рассмотрим оператор, использующий все три вида переменных:

```
dx = x - self._rx
```

С учетом этих различий еще раз удостоверьтесь, что понимаете, как работает метод `potentialAt()`.

Переменные в пределах методов

Переменная	Цель	Пример	Область видимости
Параметра	Передает аргумент от клиента к методу	`self, x, y`	Метод
Экземпляра	Значение типа данных	`_rx, _ry`	Класс
Локальная	Временно используется в пределах метода	`dx, dy`	Метод

Методы — это функции. Метод — это специальный вид функции, определяемой в классе и ассоциированной с объектом. Поскольку методы являются функциями, они могут получать любое количество аргументов, определять необязательные аргументы со стандартными значениями и возвращать значение своей вызывающей стороне. В нашем примере метод `potentialAt()` получает два обычных аргумента и возвращает клиенту значение типа `float`. Основное отличие между функциями и методами в том, что метод ассоциирован с определенным объектом и имеет непосредственный доступ к его переменным экземпляра. Далее в этом разделе мы рассмотрим примеры, где цель метода заключается

в создании побочного эффекта — изменении значения переменной экземпляра, а не возвращения значения клиенту.

Встроенные функции. Третья операция в API типа `Charge` — это встроенная функция `str(c)`. Согласно соглашению, Python автоматически преобразует этот вызов функции в вызов стандартного метода `c.__str()__`. Таким образом, для поддержки этой операции мы реализуем *специальный метод* `__str__()`, использующий переменные экземпляра вызывающего объекта для достижения желаемого результата. Напомним, что во время выполнения Python знает тип каждого объекта, поэтому он может выяснить, какой именно метод `__str__()` вызывать (тот, что определен в классе, соответствующем вызывающему объекту).

Закрытость. Клиент должен обращаться к типу данных только через методы в его API. Иногда удобно определить в реализации вспомогательные методы, не предназначенные для непосредственного вызова клиентом. Специальный метод `__str__()` является типичным примером. В разделе 2.2 упоминались *закрытые функции*, согласно соглашению Python имена таких методов начинаются символом подчеркивания. Первый символ подчеркивания — это однозначный сигнал клиенту не вызвать этот *закрытый метод* (private method) непосредственно. Аналогично начинающееся символом подчеркивания имя переменной экземпляра требует от клиента не пытаться получать доступ к таким *закрытым переменным экземпляра* (private instance variable) непосредственно. Хотя в языке Python нет встроенной поддержки этого соглашения, большинство программистов Python его соблюдают.

Для реализации типов данных как классов Python следует знать его базовые компоненты. У каждой рассмотренной нами реализации типа данных (класса Python) есть переменные экземпляра, конструкторы и методы. Каждый разрабатываемый нами тип данных имеет те же элементы. Вместо того чтобы думать, какое действие необходимо предпринять далее для достижения цели (как это было вначале изучения программирования), нужно подумать о потребностях клиента, а затем приспособить к ним тип данных. Программа 3.2.1 — полная реализация нашего типа данных `Charge`; она иллюстрирует все обсуждаемые средства. Обратите внимание: есть также клиент проверки этого типа — программа `charge.py`. Подобно модулям, для каждой реализации типа данных имеет смысл создавать клиент проверки.

Резюме. Рискуя повториться, подведем итог основным этапам, описанным в процессе создания нового типа данных.

Первый этап создания типа данных подразумевает определение API. Задача API — отделить клиента от реализации и облегчить таким образом модульное программирование. При определении API преследуются две цели. Во-первых, мы хотим обеспечить простой и правильный клиентский код. На самом деле имеет смысл составить некий клиентский код прежде, чем завершить определение API. Это позволит удостовериться, что определенные операции с типами данных именно те, которые нужны клиенту. Во-вторых, необходимо быть в состоянии реализовать операции. Нет никакого смысла определять операции, которые не знаешь, как реализовать.

Второй этап создания типа данных подразумевает реализацию класса Python в соответствии со спецификацией API. Сначала мы составляем конструктор, чтобы определить и инициализировать переменные экземпляра; затем составляем методы, манипулирующие переменными экземпляра и реализующие необходимые функциональные возможности. В языке Python обычно реализуют три вида методов.

Программа 3.2.1. Заряженная частица (charge.py)

```
import math
import sys
import stdio

class Charge:
    def __init__(self, x0, y0, q0):
        self._rx = x0
        self._ry = y0
        self._q = q0

    def potentialAt(self, x, y):
        COULOMB = 8.99e09
        dx = x - self._rx
        dy = y - self._ry
        r = math.sqrt(dx*dx + dy*dy)
        if r == 0.0: return float('inf')
        return COULOMB * self._q / r

    def __str__(self):
        result = str(self._q) + ' at ('
        result += str(self._rx) + ', ' + str(self._ry) + ')'
        return result

def main():
    x = float(sys.argv[1])
    y = float(sys.argv[2])
    c = Charge(.51, .63, 21.3)
    stdio.writeln(c)
    stdio.writeln(c.potentialAt(x, y))

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

	Переменные экземпляра
_rx	Координата x
_ry	Координата y
_q	Значение заряда
COULOMB	Расстояние кулона
dx, dy	Разница расстояний от запрошенной точки
r	Дистанция от запрошенной точки
x, y	Запрошенная точка
c	Заряд

Эта реализация нашего типа данных для заряженных частиц содержит базовые компоненты, имеющиеся в каждом типе данных: переменные экземпляра _rx, _ry и _q; конструктор __init__(); методы potentialAt() и __str__(); а также клиент проверки main() (см. также программу 3.1.1).

```
% python charge.py .5 .5
21.3 at (0.51, 0.63)
1468638248194.164
```

- Для реализации *конструктора* мы реализуем специальный метод `__init__()` с первой параметрической переменной `self`, сопровождаемой обычными параметрическими переменными конструктора.
- Для реализации *метода* мы реализуем функцию с необходимым именем и первой параметрической переменной `self`, сопровождаемой обычными параметрическими переменными метода.
- Для реализации *встроенной функции* мы реализуем специальный метод с тем же именем, заключенным в двойные символы подчеркивания, и первой параметрической переменной `self`.

Примеры типа `Charge` представлены в таблице ниже. Во всех трех случаях мы используем параметр `self` для доступа к переменным экземпляра вызывающего объекта, а в других случаях используем параметры и локальные переменные для вычисления результата и возвращения значения вызывающей стороне.

Методы в реализации типа данных Python

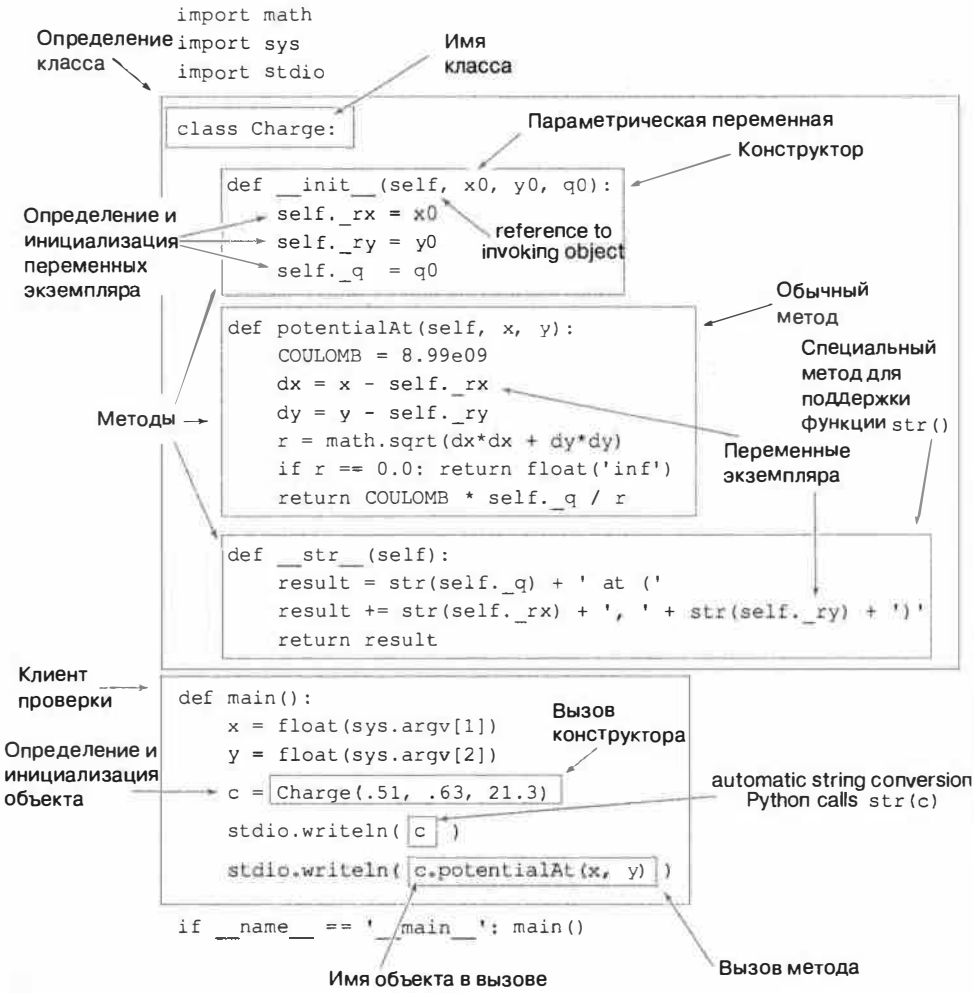
Операция	Вызов (в клиентском коде)	Сигнатура (в коде реализации)
Конструктор	`c = Charge(x0, y0, q0)`	`__init__(self, x0, y0, q0)`
Метод	`c.potentialAt(x, y)`	`potentialAt(self, x, y)`
Встроенная функция	`str(c)`	`__str__(self)`

Третий этап создания типа данных подразумевает составление *клиента проверки*, позволяющего проверять проектные решения, принятые на первых двух этапах.

Обсуждаемая терминология приведена на рисунке ниже. Она заслуживает внимания, поскольку мы будем часто использовать ее далее в книге.

В оставшейся части данного раздела мы применяем эти основные этапы для создания многих типов данных и клиентов. Мы начинаем каждый пример с API, а затем рассматриваем реализации и клиенты. В конце этого раздела вы найдете много упражнений, позволяющих приобрести опыт в создании типов данных. Раздел 3.3 предоставляет краткий обзор процесса проектирования и соответствующих языковых механизмов.

Какие значения определяют тип данных и какие операции клиенты должны выполнять с этими значениями? Ответив на эти простые вопросы, вы сможете создавать новые типы данных и составлять клиенты, использующие их таким же образом, как и встроенные типы данных.



Анатомия определения класса (типа данных)

Секундомер. Один из признаков объектно-ориентированного программирования — легкость моделирования реальных объектов при создании абстрактных программных объектов. В качестве простого примера рассмотрим программу 3.2.2 (`stopwatch.py`), реализующую наш тип данных `Stopwatch`, определенный следующим API:

API для нашего типа данных `Stopwatch`

Операция	Описание
`Stopwatch()`	Новый секундомер (сразу запущен)
`watch.elapsedTime()`	Время (в секундах), прошедшее с момента создания объекта `watch`

Другими словами, объект Stopwatch — это упрощенная версия старомодного секундомера. Он запускается при создании, и вы можете запросить его, как долго он выполняется, вызвав метод elapsedTime(). Вы вполне можете добавлять в программу Stopwatch всякого рода усовершенствования, ограничиваясь только своим воображением. Хотите быть в состоянии сбросить секундомер? Запускать и останавливать его? Все это можно добавить (см. упр. 3.2.11).

Реализация программы stopwatch.py использует в своих интересах функцию Python time() из модуля time, возвращающую значение типа float, соответствующее текущему времени в секундах (обычно количество секунд с полуночи 1 января 1970 года в стандарте UTC). Реализация этого типа данных едва ли могла быть проще. Объект Stopwatch сохраняет время своего создания в переменной экземпляра, а затем возвращает разницу между этим и текущим временем всякий раз, когда клиент вызывает его метод elapsedTime(). Сам объект Stopwatch не тикает (такты для всех объектов Stopwatch отсчитывают внутренние системные часы на вашем компьютере); это только иллюзия, что работу для клиентов осуществляет он. Почему бы не использовать функцию time.time() непосредственно на клиенте? Вполне можно, но использование высокоуровневой абстракции Stopwatch ведет к более понятному и простому в обслуживании клиентскому коду.

Клиент проверки типичен. Он создает два объекта типа Stopwatch, использует их для регистрации времени запуска двух разных вычислений, а затем выводит соотношение продолжительностей.

Вопрос в том, лучше ли один подход для решения проблемы другого. В разделе 4.1 мы выработаем научный подход к пониманию стоимости вычисления. Класс Stopwatch — весьма полезный инструмент в этом подходе.

Гистограмма. Программа 3.2.3 (histogram.py) определяет тип данных Histogram, позволяющий графически представить распределение данных, используя полосы различных высот. Такой график называется *гистограммой* (histogram). Объект Histogram содержит *массив* частот вхождения целочисленных значений в данном интервале. Функция stdstats.plotBars() используется для отображения гистограммы значений, контролируемых следующими API.

API для нашего типа данных Histogram

Операция	Описание
Histogram(n)	Новая гистограмма для целочисленных значений 0, 1, ..., $n - 1$
h.addDataPoint(i)	Добавление вхождения целого числа i к гистограмме h
h.draw()	Вывести h на стандартное графическое устройство

Программа 3.2.2. Секундомер (stopwatch.py)

```
import math
import sys
import time
import stdio

class Stopwatch:
    def __init__(self):
        self._start = time.time()
    def elapsedTime(self):
        return time.time() - self._start

def main():
    n = int(sys.argv[1])

    total1 = 0.0
    watch1 = Stopwatch()
    for i in range(1, n+1):
        total1 += i**2
    time1 = watch1.elapsedTime()

    total2 = 0.0
    watch2 = Stopwatch()
    for i in range(1, n+1):
        total2 += i*i
    time2 = watch2.elapsedTime()

    stdio.writeln(total1/total2)
    stdio.writeln(time1/time2)

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

Переменная экземпляра	
_start	Время создания

Этот модуль определяет класс `Stopwatch`, реализующий тип данных, который мы можем использовать для сравнения продолжительности критически важных для производительности методов (см. раздел 4.1). Клиент проверки получает из командной строки целое число `n` и сравнивает стоимость операции возведения в квадрат числа способами `i**2` и `i*i` для задачи вычисления суммы квадратов чисел от `1` до `n`. Эта быстрая проверка указывает, что способ `i*i` примерно в три раза быстрее, чем `i**2`.

```
% python stopwatch.py 1000000
1.0
3.179422835633626
```

Имея этот простой тип данных, мы можем воспользоваться всеми обсуждавшимися в главе 2 преимуществами модульного программирования (многократное использование кода, независимая разработка небольших программ и т.д.), а также дополнительным преимуществом — *отделением данных*. Клиент гистограммы не обязан поддерживать данные (или знать что-либо о способе их представления); он просто создает гистограмму и вызывает метод `addDataPoint()`.

При изучении кода этого и нескольких следующих примеров имеет смысл тщательно просматривать клиентский код. Каждый реализуемый нами класс, по существу, расширяет язык Python, позволяя создавать объекты нового типа данных и выполнять операции с ними. Концептуально все клиентские программы — это те же программы со встроенными типами данных, которые мы изучали вначале. Однако теперь вы можете определять типы и операции, необходимые в клиентском коде!

В данном случае использование типа `Histogram` фактически *повышает* удобочитаемость клиентского кода, подобно тому, как вызов метода `addDataPoint()` обращает внимание на изучаемые данные. Без типа `Histogram` мы были бы вынуждены смешать код создания гистограммы с кодом представляющих интерес вычислений, что загромоздит программу и затруднит ее понимание и поддержку по сравнению с двумя отдельными программами. *Всякий раз, когда можете четко разделить данные и связанные с ними задачи в пределах вычисления, так и поступайте.*

Хорошо усвоив, как тип данных будет использоваться в клиентском коде, можно переходить к реализации. Реализация характеризуется ее переменными экземпляра (значением типа данных). Тип `Histogram` содержит массив частот каждой точки. Его метод `draw()` масштабирует рисунок так, чтобы самая высокая полоса помещалась на холсте, а затем рисует график частот.

Черепашья графика. *Каждый раз, когда можете четко разделить данные и связанные с ними задачи в пределах вычисления, так и поступайте.* В объектно-ориентированном программировании мы дополняем эту мантру *состоянием* (state) и задачами. Небольшой набор состояний может быть весьма ценен для упрощения вычислений. Рассмотрим *черепашью графику* (turtle graphics) (или графику с относительными командами) на базе типа данных, определенного следующим API.

API для нашего типа данных Turtle

Операция	Описание
`Turtle(x0, y0, a0)`	Новая черепаха в точке (`x0`, `y0`), смотрящая в направлении `a0` градусов по оси *X*
`t.turnLeft(delta)`	Поворот черепахи `t` влево (против часовой стрелки) на `delta` градусов
`t.goForward(step)`	Перемещение черепахи `t` на расстояние `step` с рисованием линии

Программа 3.2.3. Гистограмма (histogram.py)

```
import sys
import stdarray
import stddraw
import stdrandom
import stdstats

class Histogram:
    def __init__(self, n):
        self._freq = stdarray.create1D(n, 0)

    def addDataPoint(self, i):
        self._freq[i] += 1

    def draw(self):
        stddraw.setYscale(0, max(self._freq))
        stdstats.plotBars(self._freq)

def main():
    n      = int(sys.argv[1])
    p      = float(sys.argv[2])
    trials = int(sys.argv[3])
    histogram = Histogram(n+1)
    for t in range(trials):
        heads = bernoulli.binomial(n, p)
        histogram.addDataPoint(heads)
    stddraw.setCanvasSize(500, 200)
    histogram.draw()
    stddraw.show()

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

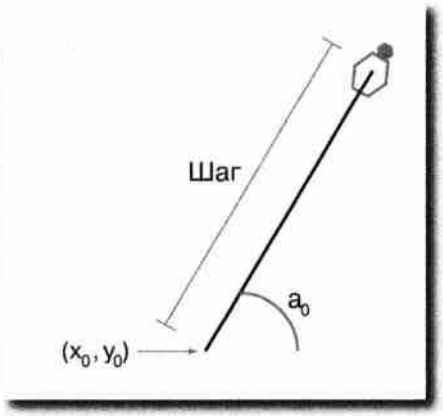
	Переменная экземпляра
`_freq[]`	Значения частот

Этот модуль определяет класс `Histogram`, реализующий тип данных для создания динамических гистограмм. Значения частот сохраняются в массиве экземпляра `_freq`. Клиент проверки получает в аргументах командной строки целое число `n`, вещественное число `p` и целое число `trials`. Он проводит `trials` экспериментов, в каждом из которых подсчитывается количество выпавших орлов при `n` бросках монеты (с вероятностью орла `p` и вероятностью решки `1 - p`). Затем он выводит результаты на стандартное графическое устройство.

```
% python histogram.py 50 .5 100000
```

Вообразите черепаху, живущую в единичном квадрате и рисующую линии по мере движения. Она может перемещаться на определенное расстояние по прямой линии и поворачиваться (против часовой стрелки) на определенное количество градусов. Согласно API, при создании черепахи мы помещаем ее в определенную точку в указанном направлении. Затем, отдавая черепахе последовательность команд `goForward()` и `turnLeft()`, создаем на экране рисунки.

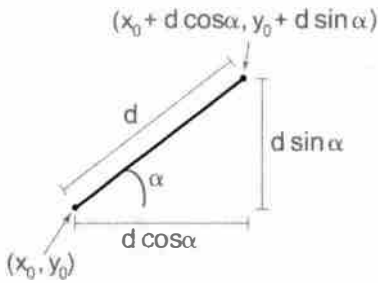
```
x0 = 0.5
y0 = 0.0
a0 = 60.0
step = math.sqrt(3)/2
turtle = Turtle(x0, y0, a0)
turtle.goForward(step)
```



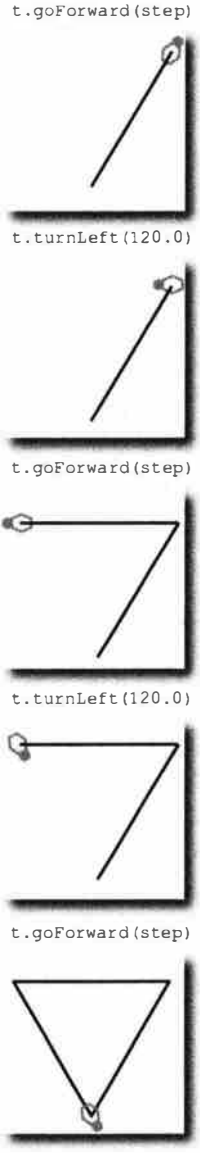
Первый шаг черепахи

Например, чтобы получить треугольник, мы создаем черепаху в точке (0, 0,5), смотрящую под углом 60 градусов против часовой стрелки от оси *X*, затем указываем ей сделать шаг вперед, повернуться на 120 градусов против часовой стрелки, сделать второй шаг вперед, повернуться еще на 120 градусов против часовой стрелки и сделать третий шаг вперед, чтобы завершить треугольник. Действительно, все исследуемые нами клиенты черепахи просто создают ее, а затем передают ей серию команд шагов и поворотов, изменяя размер шага и угол поворота. Как будет продемонстрировано далее, эта простая модель позволяет создавать рисунки произвольной сложности во многих важных приложениях.

Класс `Turtle`, определенный в программе 3.2.4 (`turtle.py`), является реализацией этого API, использующего модуль `stddraw`. Он содержит три переменные экземпляра: координаты позиции черепахи и ее текущее направление в градусах от оси *X* против часовой стрелки (*полярный угол*). Реализация двух методов класса требует *изменения* значений этих переменных, поэтому объекты типа `Turtle` изменяемы. Необходимые изменения просты: `turnLeft(delta)` добавляет `delta` к текущему углу, а `goForward(step)` — размер шага.



Черепашья тригонометрия



Ваш первый рисунок черепашьей графики

Клиент проверки в `Turtle` получает как аргумент командной строки целое число `n` и рисует многоугольник с `n` сторонами. Если вас интересует элементарная аналитическая геометрия, то вам, возможно, захочется проверить этот факт. Сделали вы это или нет, давайте подумаем о том, что пришлось бы сделать для вычисления координат всех точек многоугольника. Простота черепашьего подхода подкупает. Короче говоря, черепашья графика — весьма полезная абстракция для описания всех геометрических форм. Например, при достаточно большом значении `n` получается хорошее приближение к кругу.

Объект `Turtle` можно использовать как любой другой объект. Программы могут создать массивы объектов `Turtle`, передавать их как аргументы функциям и т.д. Наши примеры иллюстрируют эти возможности и убедят вас, что создание таких типов данных, как `Turtle`, и очень просто, и очень полезно. Для каждого из них, подобно обычным многоугольникам, вполне *возможно* вычислить координаты всех точек и прямых линий, чтобы получить рисунки, но с классом `Turtle` это *проще* сделать. Черепашья графика иллюстрирует значение абстракции данных.

Фрактальная графика. Кривая Коха порядка 0 — это прямая линия. Для формирования кривой Коха порядка n получают кривую Коха порядка $n - 1$, поворачивают влево на 60 градусов, рисуют вторую кривую Коха порядка n – 1, поворачивают вправо на 120 градусов (–120 градусов влево), рисуют третью кривую Коха порядка $n - 1$, поворачивают влево на 60 градусов и рисуют четвертую кривую Коха порядка $n - 1$. Эти фрактальные инструкции непосредственно ведут к коду клиента черепахи. При соответствующей модификации подобные фрактальные шаблоны могут быть весьма полезны при моделировании самоповторяющихся узоров, весьма распространенных в природе, таких как снежинки.

Приведенный ниже клиентский код прост и понятен, за исключением значения размера шага. Если тщательно исследовать несколько первых примеров, то можно увидеть (и доказать индукцией), что ширина кривой порядка n составляет 3^n размера шага, таким образом, установив размер шага, равный $1 / 3^n$, получим кривую шириной 1. Точно так же количество шагов кривой порядка n составляет 4^n, таким образом, программа `koch.py` может и не завершить работу за разумный период, если вызвать ее для большого n.

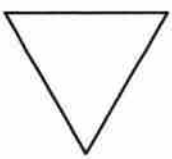
Программа 3.2.4. Черепашья графика (`turtle.py`)

```
import math
import sys
import stddraw

class Turtle:
    def __init__(self, x0, y0, a0):
        self._x = x0
        self._y = y0
        self._angle = a0

    def turnLeft(self, delta):
        self._angle += delta

    def goForward(self, step):
        oldx = self._x
        oldy = self._y
        self._x += step * math.cos(math.radians(self._angle))
        self._y += step * math.sin(math.radians(self._angle))
        stddraw.line(oldx, oldy, self._x, self._y)

def main():
    n = int(sys.argv[1])
    step = math.sin(math.radians(180.0/n))
    turtle = Turtle(.5, .0, 180.0/n)
    for i in range(n):
        turtle.goForward(step)
        turtle.turnLeft(360.0/n)
    stddraw.show()

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

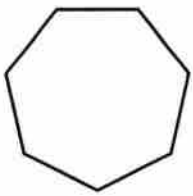
	Переменные экземпляра
`_x`, `_y`	Позиция (в единичном квадрате)
`_angle`	Полярный угол (в градусах)

Этот тип данных поддерживает черепашью графику, которая зачастую упрощает создание рисунков.

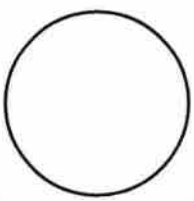
```
% python turtle.py 3
```



```
% python turtle.py 7
```



```
% python turtle.py 1000
```



Можно найти много примеров фрактальных узоров подобного типа, они изучались и разрабатывались математиками, учеными и художниками во многих контекстах. Здесь наш интерес заключается в том, что абстракция черепашьей графики весьма упрощает рисующий их клиентский код.

```
import sys
import stddraw
from turtle import Turtle

def koch(n, step, turtle):
    if n == 0:
        turtle.goForward(step)
        return
    koch(n-1, step, turtle)
    turtle.turnLeft(60.0)
    koch(n-1, step, turtle)
    turtle.turnLeft(-120.0)
    koch(n-1, step, turtle)
    turtle.turnLeft(60.0)
    koch(n-1, step, turtle)
    n = int(sys.argv[1])

stddraw.setPenRadius(0.0)
step = 3.0 ** -n
turtle = Turtle(0.0, 0.0, 0.0)
koch(n, step, turtle)
stddraw.show()
```

```
% python koch.py 0
```



```
% python koch.py 1
```

```
% python koch.py 2
```

```
% python koch.py 3
```

```
% python koch.py 4
```

Рисование кривых Коха с использованием черепашьей графики

Удивительная спираль. Возможно, черепаха немного устала, сделав 4^n шага, рисуя кривую Коха (или просто обленилась). Поэтому предположим, что каждый шаг черепахи становится немного меньше (с постоянным коэффициентом, близким к 1). Что будет с нашими рисунками? Для ответа на этот вопрос изменим клиента проверки в программе 3.2.4, рисующей многоугольник, и получим изображение *логарифмической спирали* (logarithmic spiral) — кривой, встречающейся в природе довольно часто.

Программа 3.2.5 (`spiral.py`) является реализацией этой кривой. Она инструктирует черепаху поочередно шагать и поворачиваться, пока она не пройдет по кругу заданное количество раз при уменьшении размера каждого шага с постоянным коэффициентом. Сценарий получает три аргумента командной строки, контролирующих форму и характер спирали. Как можно заметить в четырех примерах запуска программы, спирали стремятся к центру рисунка. Чтобы понять, как параметры контролируют поведение спирали, поэкспериментируйте с программой `spiral.py`.

Впервые логарифмическая спираль была описана Рене Декартом (René Descartes) в 1638 году. Якоб Бернулли (Jacob Bernoulli) был так поражен ее математическими свойствами, что назвал ее *удивительной спиралью* (miraculous spiral) и даже попросил выгравировать ее на своей надгробной плите. Многие также находят ее удивительной, поскольку она точно описывает широкое разнообразие естественных явлений. Ниже приведены три примера: раковина наутилуса, ветви спиральной галактики и формирующиеся облака тропического шторма. Ученые наблюдали также ее как путь ястреба, приближающегося к добыче, и как путь заряженной частицы, перемещенной в перпендикулярное равномерное магнитное поле.

Раковина наутилуса

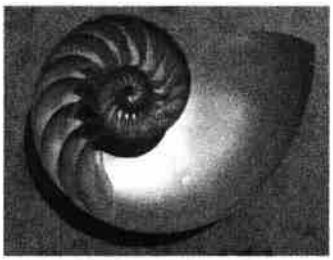


Фото: Chris 73 (лицензия CC-by-SA)

Спиральная галактика

Фото: NASA и ESA

Штормовые облака

Фото: NASA

Примеры удивительной спирали в природе

Одна из задач научного исследования заключается в предоставлении простых, но точных моделей сложных естественных явлений. Наша усталая черепаха, конечно, проходит этот тест!

Броуновское движение. Возможно, черепаха немного перебрала. В результате дезориентации она поворачивает в *случайном* направлении перед каждым шагом. Снова нарисуем путь черепахи для миллионов этапов, и снова окажется, что такие пути встречаются в природе во многих контекстах. В 1827 году ботаник Роберт Броун увидел через микроскоп, что погруженные в воду крошечные частицы пыльцы перемещаются случайным образом. Этот процесс был назван *броуновским движением* (Brownian motion), и впоследствии он привел Альберта Эйнштейна к проникновению в суть природы элементарных частиц.

Возможно, у нашей черепахи есть друзья, также немного переоценившие свои возможности. После достаточно долгих блужданий их пути переплетаются и становятся неразличимы по отдельности. Астрофизики сегодня используют эту модель для понимания наблюдаемых свойств отдаленных галактик.

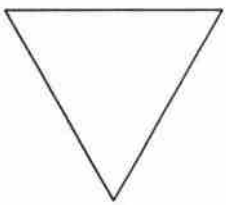
Программа 3.2.5. Удивительная спираль (spiral.py)

```
import math
import sys
import stddraw
from turtle import Turtle

n = int(sys.argv[1])
wraps = int(sys.argv[2])
decay = float(sys.argv[3])
angle = 360.0 / n

step = math.sin(math.radians(angle/2.0))
turtle = Turtle(0.5, 0, angle/2.0)

stddraw.setPenRadius(0)
for i in range(wraps * n):
    step /= decay
    turtle.goForward(step)
    turtle.turnLeft(angle)
stddraw.show()
```

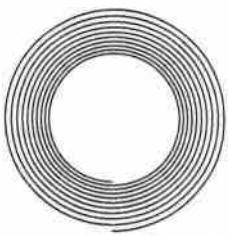
`n`	Количество сторон
`wraps`	Количество кругов
`step`	Размер шага
`decay`	Коэффициент уменьшения
`angle`	Величина поворота
`step`	Размер шага
`turtle`	Ленивая черепаха

Этот код — модификация клиента проверки в программе 3.2.4, где размер каждого шага уменьшается и осуществляется заданное количество кругов. Значение `n` контролирует форму, `wraps` — продолжительность, а `decay` — характер спирали.

```
% python spiral.py 3 1 1.0
```

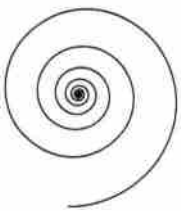


```
% python spiral.py 3 10 1.2
```



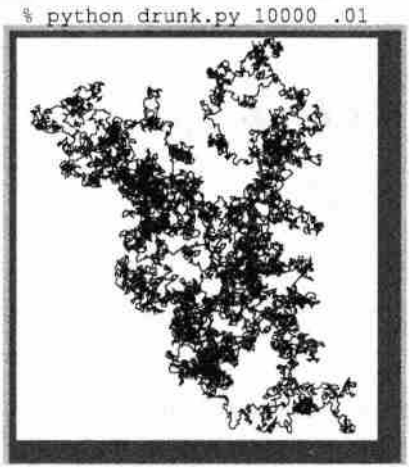
```
% python spiral.py 1440 10 1.00004
```

```
% python spiral.py 1440 10 1.0004
```



Первоначально черепашья графика была разработана Сеймуром Пейпертом (Seymour Papert) в Массачусетсском технологическом институте в 1960-х годах как часть учебного языка программирования Logo, который до сих пор используется в игрушках. Но черепашья графика не игрушка, как можно заметить по многочисленным научным примерам. У черепашьей графики есть также многочисленные коммерческие применения. Например, она лежит в основе языка программирования PostScript, используемого для создания печатаных страниц большинства газет, журналов и книг. В данном контексте тип `Turtle` — это самый существенный пример объектно-ориентированного программирования, демонстрирующий, что даже небольшой объем сохраняемых состояний (объекты используют также абстракции данных, а не только функций) способен значительно упростить вычисление.

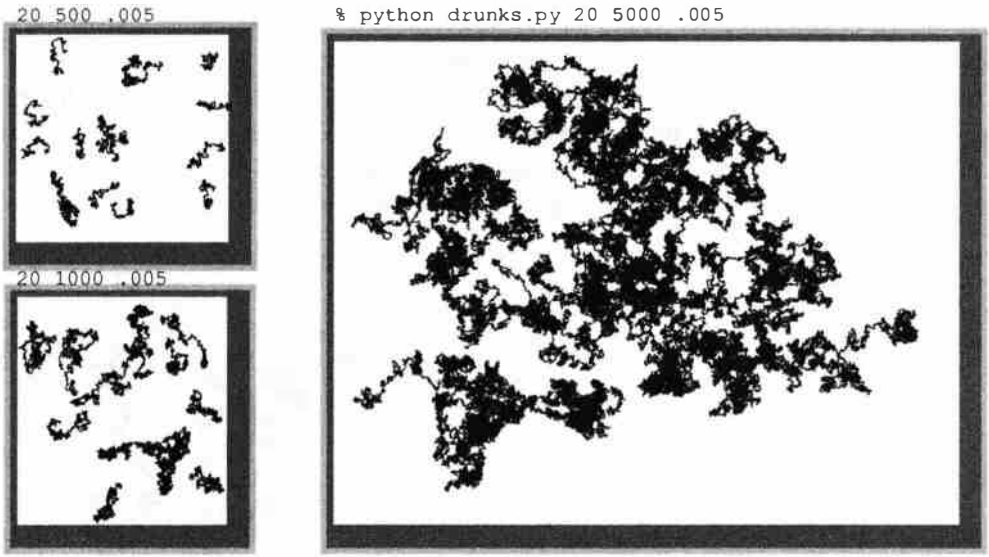
```
import sys
import stddraw
import stdrandom
from turtle import Turtle

trials = int(sys.argv[1])
step = float(sys.argv[2])
stddraw.setPenRadius(0.0)
turtle = Turtle(0.5, 0.5, 0.0)
for t in range(trials):
    angle = stdrandom.uniformFloat(0.0, 360.0)
    turtle.turnLeft(angle)
    turtle.goForward(step)
    stddraw.show(0)
stddraw.show()
```



Броуновское движение пьяной черепахи
(перемещение на фиксированное расстояние в случайном направлении)

```
import sys
import stdarray
import stddraw
import stdrandom
from turtle import Turtle

n = int(sys.argv[1])
trials = int(sys.argv[2])
step = float(sys.argv[3])
stddraw.setPenRadius(0.0)
turtles = stdarray.create1D(n)
for i in range(n):
    x = stdrandom.uniformFloat(0.0, 1.0)
    y = stdrandom.uniformFloat(0.0, 1.0)
    turtles[i] = Turtle(x, y, 0.0)
for t in range(trials):
    for i in range(n):
```

```
        angle = stdrandom.uniformFloat(0.0, 360.0)
        turtles[i].turnLeft(angle)
        turtles[i].goForward(step)
        stddraw.show(0)
stddraw.show()
```



Броуновское движение компании пьяных черепах

Комплексные числа. *Комплексное число* (complex number) имеет форму $x+yi$, где x и y — вещественные числа, i — квадратный корень -1. Число x — это *вещественная* часть комплексного числа, а y — *мнимая*. Эта терминология основана на идее, что квадратный корень -1 должен быть мнимым числом, поскольку ни у какого вещественного числа не может быть этого значения. Комплексные числа — это квинтэссенция математической абстракции: хоть и не каждый полагает, что имеет смысл физически брать квадратный корень из -1, комплексные числа помогают нам понять естественный мир. Они интенсивно используются в прикладной математике и играют важную роль во многих отраслях науки и техники. Они используются в моделировании физических систем всех сортов, от каналов до звуковых волн и электромагнитных полей. Как правило, эти модели требуют интенсивных вычислений и манипулирования комплексными числами согласно четким арифметическим правилам, поэтому мы и хотим составлять компьютерные программы для осуществления этих вычислений. Короче говоря, необходим новый тип данных.

Ни один язык программирования не может предоставить реализации всех математических абстракций, в которых могла бы возникнуть нужда, однако благодаря возможности реализовать типы данных мы можем не только составлять программы, легко манипулирующие такими абстракциями, как комплексные числа, полиномы, векторы и матрицы, но и думать в терминах новых абстракций.

Язык Python предоставляет тип данных `complex` (со строчной `c`). Однако разработка типа данных для комплексных чисел — отличное упражнение для объектно-ориентированного программирования, поэтому создадим собственный тип данных `Complex` (с прописной `C`). Это позволит нам решить нетривиальным способом много интересных задач с типами данных для математических абстракций.

Операции с комплексными числами, необходимые для базовых вычислений, включают сложение и умножение применительно к коммутативному, ассоциативному и дистрибутивному законам алгебры (наряду с идентификатором $i^2 = -1$); вычисление модуля, а также извлечение вещественной и мнимой частей, согласно следующим уравнениям.

- *Сложение*: $(x+yi) + (v+wi) = (x+v) + (y+w)i$.
- *Умножение*: $(x + yi) * (v + wi) = (xv - yw) + (yv + xw)\ i$.
- *Модуль*: $|x + yi| = \sqrt{x^2 + y^2}$.
- *Вещественная часть*: $\mathrm{Re}(x + yi) = x$.
- *Мнимая часть*: $\mathrm{Im}(x + yi) = y$.

Например, если $a = 3 + 4i$ и $b = -2 + 3i$, то $a+b = 1 + 7i$, $a*b = -18 + i$, $\mathrm{Re}(a) = 3$, $\mathrm{Im}(a) = 4$ и $|a| = 5$.

При этих базовых определениях реализация типа данных для комплексных чисел ясна. Как обычно, мы начинаем с API, определяющих операции с типом данных. В этом API для простоты мы сосредоточимся только на базовых операциях, но упражнение 3.2.18 требует учесть несколько других полезных операций и обеспечить их поддержку. Кроме прописной буквы `C`, встроенный тип данных Python `complex` реализует те же API.

API для пользовательского типа данных `Complex`

Клиентская операция	Специальный метод	Описание
`Complex(x, y)`	`__init__(self, re, im)`	Новый объект `Complex` со значением `x+yi`
`a.re()`		Вещественная часть `a`
`a.im()`		Мнимая часть `a`
`a + b`	`__add__(self, other)`	Сумма `a` и `b`
`a * b`	`__mul__(self, other)`	Произведение `a` и `b`
`abs(a)`	`__abs__(self)`	Модуль `a`
`str(a)`	`__str__(self)`	`'x + yi'` (строковое представление `a`)

Специальные методы. Как реализовать операции с типом данных так, чтобы клиенты могли вызывать их, используя такие арифметические операторы, как `+` и `*`? Ответ — в другом наборе *специальных методов* (special method) Python, предназначенных именно для этой цели. Например, когда Python встречает в клиентском коде выражение `a + b`, он заменяет его вызовом метода `a.__add__(b)`. Аналогично Python заменяет выражение `a * b` вызовом метода `a.__mul__(b)`. Следовательно, чтобы тип

работал так, как ожидается, необходимо лишь реализовать специальные методы `__add__()` и `__mul__()` для сложения и умножения. Это тот же механизм, что и использовавшийся ранее для поддержки встроенной функции Python `str()` типом `Charge` (реализация специального метода `__str__()`), за исключением того, что специальные методы для арифметических операторов получают два аргумента. Вышеупомянутый API включает дополнительный столбец, сопоставляющий клиентские операции со специальными методами. Обычно в API мы опускаем этот столбец, поскольку эти имена стандартны и безотносительны к клиенту. Список специальных методов Python обширен, и мы обсудим его далее, в разделе 3.3.

Программа 3.2.6 (`Complex`) — это класс, реализующий данный API. У него есть все те же компоненты, что и у типа `Charge` (как и у каждой реализации типа данных Python): переменные экземпляра (`_re` и `_im`), конструктор, методы и клиент проверки. Сначала клиент проверки присваивает z_0 значение $1+i$, затем присваивает z к z_0 и наконец вычисляет выражения:

$$z = z^2 + z_0 = (1+i)^2 + (1+i) = (1+2i-1) + (1+i) = 1+3i;$$
$$z = z^2 + z_0 = (1+3i)^2 + (1+i) = (1+6-9) + (1+i) = -7+7i.$$

Этот код прост и подобен коду, уже встречавшемуся в этой главе.

Доступ к переменным экземпляра в объектах того же типа. Реализации обоих методов, `__add__()` и `__mul__()`, нуждаются в доступе к значениям в *двух* объектах: в объекте, переданном как аргумент, и в объекте вызова метода (т.е. объекте по ссылке `self`). Когда клиент вызывает метод `a.__add__(b)`, параметрическая переменная `self` ссылается на объект `a`, а параметрическая переменная `other` — на объект `b`. Как обычно, мы можем обращаться к переменным экземпляра объекта `a` с использованием синтаксиса `self._re` и `self._im`, а к переменным объекта `b` — с помощью синтаксиса `other._re` и `other._im`. Согласно соглашению о закрытых переменных экземпляра, мы не должны непосредственно обращаться к переменным экземпляра в другом классе. Доступ к переменным экземпляра в пределах другого объекта того же класса не нарушает этого правила.

Неизменность. Значения этих двух переменных экземпляра типа `Complex` устанавливаются при создании каждого из объектов и не изменяются на протяжении их существования. Таким образом, объекты типа `Complex` неизменны. Мы обсудим преимущества этого проектного решения в разделе 3.3.

Комплексные числа — это основа сложных вычислений в прикладной математике, имеющих много применений. Наша основная причина разработки типа `Complex` (несмотря на наличие встроенного типа Python `complex`) заключается в демонстрации использования специальных функций Python в простом, но реалистичном случае. В реальных приложениях, конечно, нужно использовать тип `complex`, если только вы не нуждаетесь в большем количестве операций, чем обеспечивает Python.

Чтобы вы получили представление о характере вычислений, задействующих комплексные числа, и об удобстве абстракции комплексного числа, рассмотрим известный пример клиента типа `complex` (или `Complex`).

Программа 3.2.6. Комплексные числа (`complex.py`)

```
import math
import stdio

class Complex:

    def __init__(self, re=0, im=0):
        self._re = re
        self._im = im

    def re(self): return self._re
    def im(self): return self._im

    def __add__(self, other):
        re = self._re + other._re
        im = self._im + other._im
        return Complex(re, im)

    def __mul__(self, other):
        re = self._re * other._re - self._im * other._im
        im = self._re * other._im + self._im * other._re
        return Complex(re, im)

    def __abs__(self):
        return math.sqrt(self._re*self._re + self._im*self._im)

    def __str__(self):
        return str(self._re) + ' + ' + str(self._im) + 'i'

    def main():
        z0 = Complex(1.0, 1.0)
        z = z0
        z = z*z + z0
        z = z*z + z0
        stdio.writeln(z)

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

	Переменные экземпляра
`_re`	Вещественная часть
`_im`	Мнимая часть

Этот тип данных позволяет составлять программы Python, манипулирующие комплексными числами.

```
% python complex.py
-7.0 + 7.0i
```

Множество Мандельброта (Mandelbrot set) — это специфический набор комплексных чисел, обнаруженных Бенуа Мандельбротом (Benoît B. Mandelbrot),

и обладающий многими замечательными свойствами. Это фрактальный шаблон, подобный папоротнику Барнсли, треугольнику Серпинского, броуновскому мосту, кривой Коха, пьяной черепахе и другим рекурсивным (самоповторяющимся) узорам, программы создания которых приведены в этой книге. Узоры подобного вида нередко встречаются в естественной природе, и их программные модели очень важны в современной науке.

Набор точек во множестве Мандельброта не может быть описан единым математическим уравнением, он определяется *алгоритмом*, а потому является идеальным кандидатом в клиенты для типа `complex` (мы изучим этот набор, создавая программы, рисующие его график).

Правило, позволяющее определить, принадлежит ли комплексное число z_0 к множеству Мандельброта, обманчиво просто. Рассмотрим последовательность комплексных чисел z_0, z_1, z_2, ..., z_t, ..., где $z_{t+1} = (z_t)^2 + z_0$. Например, в следующей таблице демонстрируется несколько первых элементов последовательности, соответствующей $z_0 = 1 + i$.

Вычисление последовательности Мандельброта

t	z_t	$(z_t)^2$	$(z_t)^2 + z_0$
0	$1 + i$	$1 + 2i + i^2 = 2i$	$2i + (1 + i) = 1 + 3i$
1	$1 + 3i$	$1 + 6i + 9i^2 = -8 + 6i$	$-8 + 6i + (1 + i) = -7 + 7i$
2	$-7 + 7i$	$49 - 98i + 49i^2 = -98i$	$-98i + (1 + i) = 1 - 97i$

Теперь, если $|z_t|$ последовательности стремится к бесконечности, то z_0 *не находится* во множестве Мандельброта; если последовательность ограничена, то z_0 *находится* во множестве Мандельброта. Для одних точек проверка проста, для других требует большего количества вычислений, как свидетельствуют примеры в этой таблице.

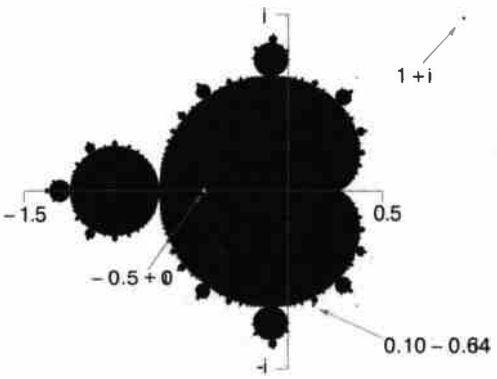
Последовательность Мандельброта для нескольких отправных точек

z_0	$0 + 0i$	$2 + 0i$	$1 + i$	$0 + i$	$-0.5 + 0i$	$-0.10 - 0.64i$
z_1	0	6	$1 + 3i$	$-1 + i$	-0.25	$-0.30 - 0.77i$
z_2	0	36	$-7 + 7i$	$-i$	-0.44	$-0.40 - 0.18i$
z_3	0	1446	$1 - 97i$	$-1 + i$	-0.31	$0.23 - 0.50i$
z_4	0	2090918	$-9407 - 193i$	$-i$	-0.40	$-0.09 - 0.87i$
...	...	...	...	...	...	...
В наборе?	Да	Нет	Нет	Да	Да	Да

Для краткости числа количеств в двух самых правых столбцах этой таблицы даны только с двумя позициями после десятичной точки. В некоторых случаях мы можем доказать принадлежность чисел к набору; например, число $0 + 0i$ безусловно находится в наборе (поскольку модуль всех чисел в его последовательности равен 0), и $2 + 0i$ также, конечно, не находится в наборе (поскольку его последовательность доминирует над степенями числа 2, что отвергается). В других

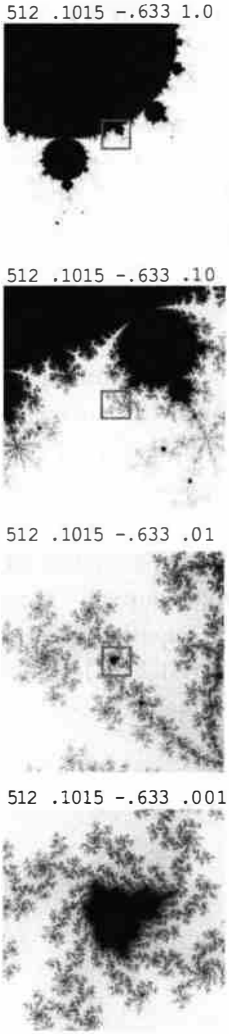
случаях рост очевиден; например, 1 + i не кажется принадлежащим набору. Другие последовательности демонстрируют периодическое поведение; например, последовательность $0 + i$ чередует $-1 + i$ и $-i$. А некоторые последовательности тянутся слишком долго, прежде чем модуль их значений начинает расти.

Для визуализации множества Мандельброта мы осуществляем выборку *комплексных* точек как и выборку вещественных значений при рисовании графика функции вещественных значений. Каждое комплексное число $x+yi$ соответствует точке (x, y) на плоскости, поэтому мы можем нарисовать график результатов следующим образом: для заданного разрешения n мы определяем регулярно разграфленную таблицу пикселей размером n на n в пределах определенного квадрата, а затем рисуем черный пиксель, если соответствующая точка находится во множестве Мандельброта, и белый пиксель, если это не так. Полученный график имеет вид поразительного узора, все черные точки которого группируются в пределах квадрата примерно 2 на 2 с центром в точке $-1/2 + 0i$. Большие значения n дают изображения более высокого разрешения, хоть и за счет большего количества вычислений. Приближенный просмотр демонстрирует самоповторение повсюду, как показано в примере, приведенном ниже. Например, тот же самый выпуклый узор с самоповторяющимися придатками присутствует по всему контуру основной черной области кардиоиды, а их размеры напоминают простую линейчатую функцию из программы 1.2.1. При увеличении краев кардиоиды проявляются точно такие же крошечные кардиоиды!



Множество Мандельброта

Но как именно мы создаем такие графики? Фактически никто не знает наверняка, поскольку нет никакого простого способа проверки безусловной принадлежности точки к набору. Имея комплексную точку, мы можем вычислить условия в начале ее последовательности, но не можем знать, что последовательность остается ограниченной. Есть простая математическая проверка, позволяющая узнать наверняка, что точка *не* принадлежит набору: если модуль какого-нибудь числа в последовательности когда-либо превысит 2 (например, $1 + 3i$), то последовательность, конечно, отклонятся.



Увеличение графика набора

Программа 3.2.7 (`mandelbrot.py`) использует эту проверку для визуального представления множества Мандельброта. Поскольку наше знание набора не является абсолютно черно-белым, мы используем в нашем визуальном представлении полутона. Основа вычислений — функция `mandel()`, получающая комплексный аргумент `z0` и целочисленный аргумент `limit`, а затем вычисляющая итерации последовательности Мандельброта, начиная с `z0` и возвращающая числа из итераций, для которых модуль остается меньшим (или равным) 2, до заданного предела.

Для каждого пикселя главный сценарий в программе `mandelbrot.py` вычисляет пиксель, соответствующий точке `z0`, а затем вычисляет `255 - mandel(z0, 255)`, чтобы получить полутоновый цвет пикселя. Любой не являющийся черным пиксель соответствует точке, о которой известно, что она может быть и не во множестве Мандельброта, поскольку модуль чисел в его последовательности превышает 2 (а следовательно, стремится к бесконечности). Черные пиксели (полутоновое значение 0) соответствуют точкам, считающимся принадлежащим набору, поскольку модуль остается меньше (или равен) 2 для 255 итераций, но мы не обязательно знаем это наверняка.

Сложность изображений, создаваемых этими простыми программами, просто поражает, особенно при увеличении крошечных деталей. Для придания изображениям большего драматизма можно использовать цвет (см. упр. 3.2.32). Кроме того, множество Мандельброта происходит от перебора только одной функции ($z^2 + z_0$); однако никто не мешает изучить свойства и других функций.

Простота кода маскирует существенный объем вычисления. В изображении 512 на 512 примерно 250 000 пикселей; для всех черных требуется 255 итераций, таким образом, для создания изображения программе `mandelbrot.py` потребуются сотни миллионов операций с комплексными значениями. Поэтому мы используем тип данных Python `complex`, который бесспорно эффективней только что рассмотренного типа данных `Complex` (см. упр. 3.2.14).

Хотя все это и очень интересно, но наш основной интерес во множестве Мандельброта заключается в иллюстрации того, что вычисления с использованием типа данных, являющегося чистой абстракцией, вполне естественное и полезное действие в программировании. Клиентский код в программе `mandelbrot.`

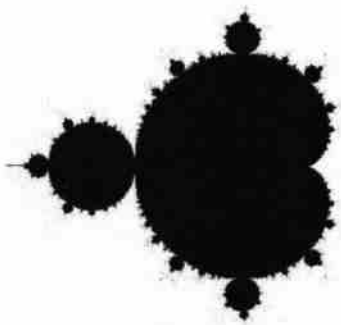
py — это простое и естественное выражение вычисления, легко проектируемого и реализуемого благодаря использованию типов данных в языке Python.

Программа 3.2.7. Множество Мандельброта (mandelbrot.py)

```
import sys
import stddraw
from color import Color
from picture import Picture

def mandel(z0, limit):
    z = z0
    for i in range(limit):
        if abs(z) > 2.0: return i
        z = z*z + z0
    return limit

n = int(sys.argv[1])
xc = float(sys.argv[2])
yc = float(sys.argv[3])
size = float(sys.argv[4])

pic = Picture(n, n)
for col in range(n):
    for row in range(n):
        x0 = xc - size/2 + size*col/n
        y0 = yc - size/2 + size*row/n
        z0 = complex(x0, y0)
        gray = 255 - mandel(z0, 255)
        color = Color(gray, gray, gray)
        pic.set(col, n-1-row, color)

stddraw.setCanvasSize(n, n)
stddraw.picture(pic)
stddraw.show()
```

x0, y0	Точка в квадрате
z0	x0 + y0i
limit	Предел итераций
xc, yc	Центр квадрата
size	Квадрат size на size
n	Таблица пикселей n на n
pic	Изображение для вывода
color	Цвет пикселя для вывода

```
% python mandelbrot.py 512 -.5 0 2
```



Эта программа получает три аргумента командной строки, определяющие таблицу n на n, центр (xc, yc) и размер квадратной области, представляющей интерес, а затем создает цифровое изображение, демонстрирующее результат выборки множества Мандельброта в области n на n таблицы равноудаленных пикселей. Она закрашивает каждый пиксель полутоновым цветом, значение которого определяется подсчетом количества итераций (не более 255), пока последовательность Мандельброта для соответствующего комплексного числа не превысит значения 2.0.

Обработка коммерческих данных. Одной из движущих сил разработки объектно-ориентированного программирования была потребность в надежном программном обеспечении для обработки коммерческих данных. Давайте для примера рассмотрим тип данных, который мог бы использоваться финансовым учреждением для отслеживания информации о клиенте.

Предположим, биржевой брокер должен обслуживать учетные записи клиентов, содержащие доли различных акций. Таким образом, брокеру необходимо обрабатывать набор значений, включающий имя клиента, количество различных собственных акций, количество долей и биржевой символ всех акций, а также кассовую наличность. Для обработки учетных записей брокер нуждается, по крайней мере, в следующих операциях, определенных в данном API.

API для пользовательского типа данных `Account`

Операция	Описание
`StockAccount(filename)`	Новая учетная запись, созданная по данным из файла `filename`
`c.valueOf()`	Общая стоимость по учетной записи `c`
`c.buy(amount, symbol)`	Добавить `amount` акций с биржевым символом `symbol` к учетной записи `c` (и вычесть их стоимость из ее наличных)
`c.sell(amount, symbol)`	Вычесть `amount` акций с биржевым символом `symbol` из учетной записи `c` (и добавить их стоимость в наличные учетной записи)
`c.write(filename)`	Записать данные учетной записи `c` в файл `filename`
`c.writeReport()`	Вывести на стандартное устройство вывода подробный отчет по учетной записи `c` (кассовая наличность плюс все акции и их стоимость)

Конечно, брокер должен покупать, продавать и отчитываться перед клиентом, но для понимания обработки данных этого вида в первую очередь следует рассмотреть конструктор `StockAccount()` и метод `write()` этого API. Информация клиента имеет долговременный характер, поэтому она должна храниться в *файле* или *базе данных*. Для обработки учетной записи клиентская программа должна читать информацию из соответствующего файла, обрабатывать ее соответствующим образом и, если она изменилась, записать обратно в файл для дальнейшего использования. Для обеспечения такой обработки необходим *формат файла* и *внутреннее представление* (или структура) данных учетной записи. Ситуация похожа на обработку матрицы из главы 1, где мы определяли формат файла (количества рядов и столбцов, сопровождаемые значениями элементов в порядном порядке) и внутреннее представление данных (двумерный массив), чтобы позволить составлять программы случайной навигации и для других применений.

Предположим, например, что брокер обслуживает небольшой портфель акций ведущих компаний по разработке программного обеспечения, принадлежащий

Алану Тьюрингу, отцу компьютеров. *Кстати*, биография Тьюринга весьма занимательна и заслуживает внимания. Кроме всего прочего, он работал над компьютерной криптографией, которая помогла приблизить конец Второй мировой войны, он создал фундамент современной теоретической информатики, разработал и построил один из первых компьютеров, а также был пионером в исследовании искусственного интеллекта. Вероятно, вполне можно предположить, что Тьюринг, безотносительно от своей финансовой ситуации академического ученого середины прошлого века, был достаточным оптимистом, чтобы поверить в потенциальное воздействие компьютерного программного обеспечения на современный мир и сделать некоторые небольшие инвестиции.

Формат файла. Современные системы зачастую используют текстовые файлы даже для данных, чтобы минимизировать зависимость от форматов, определяемых каждой программой. Для простоты мы используем прямое представление и будем выводить имя владельца счета (строка), баланс наличности (вещественное число) и имеющееся количество акций (целое число), сопровождаемые строками по всем акциям, включающими количество долей и биржевой символ, как показано в примере ниже. Для маркировки всей информации и дальнейшей минимизации зависимости от любых других программ имеет смысл использовать такие *дескрипторы*, как `<Name>`, `<Number of shares>` и т.д. (для краткости мы опускаем остальные дескрипторы).

```
% more turing.txt
Turing, Alan
10.24
4
100 ADBE
 25 GOOG
 97 IBM
250 MSFT
```

Структура данных. Для представления информации на обработку программам Python мы определяем тип данных и используем *переменные экземпляра* для организации информации. Переменные экземпляра определяют тип информации и ее структуру, что необходимо для обращения к ней в коде. Например, для реализации типа `StockAccount` используются следующие переменные экземпляра.

- Строка для имени учетной записи.
- Вещественное число для наличности.
- Целое число для количества акций.
- Массив строк для символов акций.
- Массив целых чисел для количества долей.

Мы непосредственно отражаем этот выбор в объявлениях переменных экземпляра класса StockAccount, определенного в программе 3.2.8. Массивы _stocks[] и _shares[] известны как *параллельные массивы* (parallel arrays). По индексу i массив _stocks[] возвращает символ акции, а _shares[] — количество долей в этих акциях для учетной записи. В качестве альтернативы можно было бы определить отдельный тип данных для акций, чтобы манипулировать информацией по каждой акции и поддерживать массив объектов этого типа в типе StockAccount.

Тип StockAccount включает конструктор, читающий файл и создающий учетную запись по прочитанным данным. Он также включает метод valueOf(), использующий программу 3.1.10 (stockquote.py), для получения из веб цены каждой акции. Для предоставления регулярных подробных отчетов клиентам наш брокер мог бы использовать следующий метод writeReport() класса StockAccount:

```
def writeReport(self):
    stdio.writeln(self._name)
    total = self._cash
    for i in range(self._n):
        amount = self._shares[i]
        price = stockquote.priceOf(self._stocks[i])
        total += amount * price
        stdio.writef('%4d %4s ', amount, self._stocks[i])
        stdio.writef(' %7.2f %9.2f\n', price, amount*price)
    stdio.writef('%21s %10.2f\n', 'Cash:', self._cash)
    stdio.writef('%21s %10.2f\n', 'Total:', total)
```

С одной стороны, этот клиент иллюстрирует тот тип вычислений, который обусловил эволюцию компьютеров в 1950-х годах. Банки и другие компании покупали первые компьютеры именно из-за необходимости составлять такие финансовые отчеты. Например, форматирование вывода было разработано именно для таких приложений. С другой стороны, этот клиент иллюстрирует современное веб-ориентированное вычисление, получающее информацию непосредственно из веб без использования браузера.

Поскольку реализации методов buy() и sell() требуют использования фундаментальных механизмов, рассматриваемых в разделе 4.4, отложим их до упражнения 4.4.39. Кроме этих методов, другие идеи находят широкое применение во многих других клиентах. Например, брокер мог бы захотеть создать массив всех учетных записей, а затем обработать список транзакций, изменяя информацию в учетных записях и фактически осуществляя транзакции через веб. Конечно, такой код следует разрабатывать с большой осторожностью!

Программа 3.2.8. Биржевая учетная запись (stockaccount.py)

```
import sys
import stdarray
import stdio
import stockquote
from instream import InStream

class StockAccount:
    def __init__(self, filename):
        instream = InStream(filename)
        self._name = instream.readLine()
        self._cash = instream.readFloat()
        self._n = instream.readInt()
        self._shares = stdarray.create1D(self._n, 0)
        self._stocks = stdarray.create1D(self._n, 0)
        for i in range(self._n):
            self._shares[i] = instream.readInt()
            self._stocks[i] = instream.readString()

    def valueOf(self):
        total = self._cash
        for i in range(self._n):
            price = stockquote.priceOf(self._stocks[i])
            amount = self._shares[i]
            total += amount * price
        return total

def main():
    acct = StockAccount(sys.argv[1])
    acct.writeReport()

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

	Переменные экземпляра
_name	Имя клиента
_cash	Кассовая наличность
_n	Количество акций
_shares[]	Количество долей
_stocks[]	Биржевой символ акции
instream	Поток ввода

Этот предназначенный для работы с учетными записями класс иллюстрирует типичное применение объектно-ориентированного программирования для обработки коммерческих данных. Текст метода `writeReport()` приведен выше, текст метода `write()` см. в упражнении 3.2.20, а методов `buy()` и `sell()` — в упражнении 4.4.39.

```
% python stockaccount.py turing.txt
Turing, Alan
 100  ADBE   70.56   7056.00
  25  GOOG  502.30  12557.50
  97   IBM  156.54  15184.38
 250  MSFT   45.68  11420.00
             Cash:     10.24
            Total:  46228.12
```

Когда в главе 2 мы изучали определение функций, применимых в нескольких местах программы (или даже в других программах), мы перешли от мира, где программы — это просто последовательности операторов в едином файле, к миру модульного программирования, кратко описываемой так: *всякий раз, когда можете четко разделить данные и связанные с ними задачи в пределах вычисления, так и поступайте.* Аналогичная возможность для данных, представленная в этой главе, переводит нас из мира, где данные должны быть одним из нескольких элементарных типов данных, в мир, где можно определять собственные типы данных. Эта новая возможность существенно расширяет горизонт программирования. Подобно концепции функций, как только вы научитесь реализовывать и использовать типы данных, вы поразитесь примитивности программ, не использующих их.

Но объектно-ориентированное программирование — это намного больше, чем структурирование данных. Это позволяет ассоциировать необходимые данные с операциями, манипулирующими этими данными, и сохранить их раздельно в независимом модуле. В объектно-ориентированном программировании наша мантра такова: *каждый раз, когда можете четко разделить данные и связанные с ними задачи в пределах вычисления, так и поступайте.*

Рассмотренные нами примеры убедительно доказывают, что объектно-ориентированное программирование способно сыграть ключевую роль в широком диапазоне программ. Пытаетесь ли вы разработать и создать некий физический предмет, разработать программную систему, понять естественный мир или обработать информацию — первым ключевым этапом является определение соответствующей абстракции, такой как геометрическое описание физического предмета, проекта модулей программной системы, математической модели естественного мира или структуры данных для информации. Когда мы хотим составить программу для манипулирования экземплярами хорошо проработанной абстракции, мы реализуем абстракцию как тип данных в классе Python и составляем программы, создающие объекты этого типа и манипулирующие ими.

Каждый раз, когда мы разрабатываем новый класс, использующий другие классы, создаем и манипулируем объектами типа определенного классом, мы переходим на более высокий уровень абстракции. В следующем разделе мы обсудим некоторые из сложностей, присущих этому виду программирования.

Вопросы и ответы

Могу ли я определить класс в файле, имя которого не совпадает с именем класса? Могу ли я определить несколько классов в одном файле `.py`?

Да и да, но в этой главе мы так не поступаем. В главе 4 мы встретимся с несколькими ситуациями, когда эта возможность вполне уместна.

Если метод `__init__()` технически не является конструктором, то что он собой представляет?

Другая специальная функция `__new__()`. Чтобы создать объект, Python сначала вызывает метод `__new__()`, а затем метод `__init__()`. Для программ в этой книге стандартная реализация метода `__new__()` вполне приемлема, поэтому мы не обсуждаем его.

У каждого ли класса должен быть конструктор?

Да, но если вы не определите конструктор, Python автоматически предоставит стандартный конструктор (без аргументов). При наших соглашениях такой тип данных бесполезен, поскольку у него не будет никаких переменных экземпляра.

Почему при обращении к переменным экземпляра я должен явно использовать ссылку `self`?

Синтаксически Python нуждается в некотором способе узнать, присваиваете ли вы значение локальной переменной или переменной экземпляра. Во многих других языках программирования (таких, как C++ и Java) вы *явно объявляете* переменные экземпляра типа данных, поэтому никаких неоднозначностей нет. Переменная `self` облегчает также программистам понимание того, обращается ли код к локальной переменной или к переменной экземпляра.

Предположим, я не включил метод `__str__()` в мой тип данных. Что будет, если я вызову метод `str()` или `stdio.writeln()` с объектом того типа?

Python предоставляет стандартную реализацию, возвращающую строку, содержащую тип объекта и его идентификатор (адрес в памяти). Это вряд ли имеет широкое применение, поэтому все же имеет смысл определить собственный метод.

Возможны ли в классе другие виды переменных кроме параметров, локальных переменных и переменных экземпляра?

Да. В главе 1 упоминалась возможность определять *глобальные переменные* в глобальном коде, за пределами определения любой функции, класса или метода. Область видимости глобальных переменных — весь файл `.py`. В современном программировании предпочитают ограничивать область видимости,

поэтому глобальные переменные используются редко (кроме некоторых небольших сценариев, не предназначенных для многократного использования). Python поддерживает также *переменные класса*, определяемые в классе, но вне любого из методов. Каждая переменная класса совместно используется всеми объектами класса, в отличие от переменных экземпляра, которыми каждый объект владеет индивидуально. Переменные класса используются в специальных целях, но в этой книге они не применяются.

Это только наше соглашение, или Python требует применения столь сложных имен?

Да, но это также верно и для многих других языков программирования. Вот краткое резюме соглашения об именовании, используемого в этой книге.

- Имя переменной начинается со строчной буквы.
- Имя константной переменной начинается с прописной буквы.
- Имя переменной экземпляра начинается с символа подчеркивания и строчной буквы.
- Имя метода начинается со строчной буквы.
- Имя специального метода начинается с двойного подчеркивания и строчной буквы и завершается двойным подчеркиванием.
- Имя пользовательского класса начинается с прописной буквы.
- Имя встроенного класса начинается со строчной буквы.
- Сценарий или модуль хранится в файле, имя которого состоит из строчных букв и завершается расширением `.py`.

Большинство этих соглашений *не является* частью языка, хотя многие программисты Python придерживаются их. Может возникнуть вопрос: если они так важны, то почему бы не сделать их частью языка? Хороший вопрос. Однако не всем программистам нравятся такие соглашения, и вы, вероятно, когда-нибудь встретите преподавателя, начальника или коллегу, придерживающегося совершенно иного стиля, вы также вполне можете следовать своим путем. Действительно, многие программисты Python разделяют многословные имена переменных символами подчеркивания, а не прописными буквами, предпочитая писать `is_prime` и `hurst_exponent` вместо `isPrime` и `hurstExponent`.

Как я могу определить литерал для типа `complex`?

Добавление символа `j` к числовому литералу дает мнимое число (вещественная часть которого — нуль). Вы можете добавить этот символ к числовому

литералу и получить такое комплексно сопряженное число, как `3 + 7j`. Символ `j` вместо `i` принят в некоторых технических дисциплинах. Обратите внимание, что сам по себе символ `j` *не является* литералом комплексного числа, следует использовать `1j`.

Упоминалось, что программа `mandelbrot.py` создает сотни миллионов объектов `complex`. Не замедлят ли работу дополнительные затраты на создание всех этих объектов?

Да, но не на столько, чтобы мы не смогли создавать свои графики. Наша задача в том, чтобы сделать программы читабельными и простыми в составлении и поддержке. Ограничение области видимости через абстракцию комплексного числа помогает решить эту задачу. Если по некоторым причинам необходимо значительно ускорить работу программы `mandelbrot.py`, можно попробовать избежать применения абстракции комплексного числа и использовать низкоуровневый язык, где числа не являются объектами. Обычно программы Python не оптимизируются по производительности. Мы вернемся к этой проблеме в главе 4.

Почему метод `__add__(self, other)` в программе 3.2.6 (`complex.py`) вполне нормально обращается к переменным экземпляра и параметрической переменной другого объекта? Разве эти переменные не являются закрытыми?

Программы Python обеспечивают закрытость относительно классов, а не отдельных объектов. Поэтому метод может обратиться к переменным экземпляра любого объекта своего класса. У языка Python нет никаких соглашений о "сверхзакрытых" именах, к которым можно обращаться только в экземпляре вызывающего объекта. Доступ к переменным экземпляра `other` может быть немного опасен, поскольку невнимательный клиент может передать аргумент и не имеющий тип `Complex`, и тогда мы (непреднамеренно) обратимся к переменной экземпляра в другом классе! При изменяемом типе можно было бы даже (непреднамеренно) изменить или создать переменные экземпляра в другом классе!

Если методы — это действительно функции, могу ли я вызвать метод, используя синтаксис вызова функции?

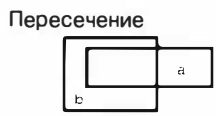
Да, определенную в классе функцию можно вызвать либо как метод, либо как обычную функцию. Например, если `c` — это объект типа `Charge`, то вызов функции `Charge.potentialAt(c, x, y)` эквивалентен вызову метода `c.potentialAt(x, y)`. В объектно-ориентированном программировании предпочитают синтаксис вызова метода, он подчеркивает роль вызывающего объекта и позволяет избежать ошибок ввода имени класса в вызове функции.

Упражнения

3.2.1. Рассмотрим следующую реализацию типа данных для прямоугольников (ориентированных по осям), представляющую каждый прямоугольник координатами его средней точки, шириной и высотой:

```
class Rectangle:
    # Создает прямоугольник с центром (x, y),
    # шириной w и высотой h.

    def __init__(self, x, y, w, h):
        self._x = x
        self._y = y
        self._width = w
        self._height = h

    # Площадь self.
    def area(self):
        return self._width * self._height

    # Периметр self.
    def perimeter(self):
        ...

    # True, если self пересекает other; в противном случае False.
    def intersects(self, other):
        ...

    # True, если self содержит other; в противном случае False.
    def contains(self, other):
        ...

    # Выводит self на stddraw.
    def draw(self):
        ...
```

Составьте API для этого типа данных и завершите его реализацию как класса, заполнив код методов `perimeter()`, `intersects()`, `contains()` и `draw()`. *Примечание*: считайте совпадающие линии пересечением, когда `a.intersects(a)` дает `True` и `a.contains(a)` дает `True`.



3.2.2. Составьте для типа `Rectangle` клиент проверки, получающий три аргумента командной строки `n`, `lo` и `hi`; создающий `n` случайных прямоугольников, ширина и высота которых однородно распределена от `lo` до `hi`

в единичном квадрате; выводящий эти прямоугольники на стандартное графическое устройство, а на стандартное устройство вывода — их среднюю площадь и средний периметр.

3.2.3. Добавьте к своему клиенту проверки из предыдущего упражнения код вычисления среднего количества пар прямоугольников, которые пересекаются и содержатся друг в друге.

3.2.4. Разработайте реализацию API своего типа `Rectangle` из предыдущих упражнений, где прямоугольник представляется координатами левого нижнего и правого верхнего углов. *Не изменяйте* API.

3.2.5. Что не так в этом коде?

```
class Charge:
    def __init__(self, x0, y0, q0):
        _rx = x0
        _ry = y0
        _q = q0
    ...
```

Решение: операторы присвоения в конструкторе создают *локальные* переменные `_rx`, `_ry` и `_q`, которым присваиваются значения параметрических переменных, которые никогда не используются. Они исчезают, когда конструктор завершает выполнение. Вместо них конструктор должен создать переменные экземпляра предварив имя каждой из них словом `self`, точкой и символом подчеркивания:

```
class Charge:
    def __init__(self, x0, y0, q0):
        self._rx = x0
        self._ry = y0
        self._q = q0
    ...
```

Символ подчеркивания не строго обязателен, но мы придерживаемся стандартных соглашений Python повсюду в этой книге, поэтому и обозначаем переменные экземпляра как закрытые.

3.2.6. Создайте тип данных `Location`, представляющий широту и долготу точки на поверхности Земли. Включите метод `distanceTo()`, вычисляющий расстояния по длине большой окружности (см. упр. 1.2.30).

3.2.7. Python предоставляет тип данных `Fraction`, определенный в стандартном модуле `fraction.py` и реализующий рациональные числа. Реализуйте

собственную версию этого типа данных. А именно: разработайте реализацию следующего API для типа данных рациональных чисел.

API для пользовательского типа данных `Rational`

Клиентская операция	Специальный метод	Описание
`Rational(x, y)`	`__init__(self)`	Новый объект `Rational` со значением `x/y`
`a + b`	`__add__(self, b)`	Сумма `a` и `b`
`a - b`	`__sub__(self, b)`	Разница `a` и `b`
`a * b`	`__mul__(self, b)`	Произведение `a` и `b`
`abs(a)`	`__abs__(self)`	Модуль `a`
`str(a)`	`__str__(self)`	`'x/y'` (строковое представление `a`)

Используйте функцию `euclid.gcd()` (программа 2.3.1) для гарантии отсутствия у числителя и знаменателя общих делителей. Включите клиент проверки, использующий все методы.

3.2.8. *Интервал* определяется как набор всех точек линии, больших или равных `left` и меньших или равных `right`. В частности, интервал с `right` меньшим `left` пуст. Составьте тип данных `Interval`, реализующий следующий API. Включите клиент проверки, получающий из командной строки `x` типа `float` и выводящий на стандартное устройство вывода (1) все полученные со стандартного ввода интервалы (каждый определяется парой вещественных чисел), содержащих `x` и (2) все пары полученных со стандартного ввода интервалов, пересекающих друг друга.

API для пользовательского типа данных `Interval`

Клиентская операция	Описание
`Interval(left, right)`	Новый объект `Interval` с конечными точками `left` и `right`
`a.contains(b)`	Содержит ли интервал `a` интервал `b`?
`a.intersects(b)`	Пересекает ли интервал `a` интервал `b`?
`str(a)`	`'[left, right]'` (строковое представление `a`)

3.2.9. Разработайте реализацию API типа `Rectangle` из упражнения 3.2.1, используя тип `Interval` для упрощения кода.

3.2.10. Составьте тип данных `Point`, реализующий следующий API. Включите клиент проверки собственного изготовления.

API для пользовательского типа данных `Point`

Клиентская операция	Описание
`Point(x, y)`	Новый объект `Point` со значением `(x, y)`
`a.distanceTo(b)`	Евклидово расстояние между `a` и `b`
`str(a)`	`'(x, y)'` (строковое представление `a`)

3.2.11. Добавьте в тип `Stopwatch` методы, позволяющие клиентам останавливать и перезапускать секундомер.

3.2.12. Используйте тип `Stopwatch` для сравнения стоимости вычисления гармонических чисел при помощи цикла `for` (см. программу 1.3.5) и рекурсивного метода, предоставленного в разделе 2.3.

3.2.13. Измените клиент проверки в файле `turtle.py` так, чтобы получать `n`-конечные звезды для нечетных `n`.

3.2.14. Составьте версию программы `mandelbrot.p`, использующую тип `Complex` вместо типа Python `complex`, как описано выше. Для вычисления соотношения продолжительности выполнения двух программ используйте тип `Stopwatch`.

3.2.15. Модифицируйте метод `__str__()` в файле `complex.py` так, чтобы он выводил комплексные числа в традиционном формате. Например, значение `3 - i` должно выводиться как `3 - i`, а не как `3.0 + -1.0i`, значение `3` как `3`, а не как `3.0 + 0.0i`, и значение `3i` как `3i`, а не как `0.0 + 3.0i`.

3.2.16. Составьте клиент типа `complex`, который получает в аргументах командной строки вещественные числа `a`, `b` и целое число `n`, а затем выводит комплексные корни уравнения $x^2 + bx + c$.

3.2.17. Составьте для типа `complex` клиент `Roots`, получающий в аргументах командной строки вещественные числа `a`, `b` и целое число `n`, а затем выводящий энные корни $a + bi$. *Примечание*: пропустите это упражнение, если не знакомы с операцией получения корней комплексных чисел.

3.2.18. Реализуйте следующие добавления к API типа `Complex`.

API для типа данных `Complex` (продолжение)

Клиентская операция	Специальный метод	Описание
`a.theta()`		Полярный угол (фаза) `a`
`a.conjugate()`		Комплексно сопряженная величина от `a`
`a - b`	`__sub__(self, b)`	Разница `a` и `b`
`a / b`	`__truediv__(self, b)`	Частное `a` и `b`
`a ** b`	`__pow__(self, b)`	`a` в степени `b`

Добавьте клиент проверки, использующий все эти методы.

3.2.19. Найдите число типа `complex`, для которого метод `mandel()` возвращает количество итераций больше 100, а затем обеспечьте увеличение этого числа, как описано выше.

3.2.20. Реализуйте в файле `stockaccount.py` метод `write()`, получающий как аргумент имя файла и выводящий содержимое учетной записи в файл, используя формат файла, определенный в тексте.

Практические упражнения

3.2.21. *Изменяемые заряды.* Модифицируйте тип `Charge` так, чтобы значение заряда `q0` могло изменяться. Для этого добавьте метод `increaseCharge()`, получающий аргумент типа `float` и добавляющий это значение к `q0`. Составьте клиент, инициализирующий массив значениями

```
a = stdarray.create1D(3)
a[0] = charge.Charge(.4, .6, 50)
a[1] = charge.Charge(.5, .5, -5)
a[2] = charge.Charge(.6, .6, 50)
```

а затем отображающий результат, постепенно уменьшая значения заряда `a[1]` при помощи кода, рассчитывающего изображение в цикле следующим образом:

```
for t in range(100):
    # Рассчет изображения p.
    stddraw.clear()
    stddraw.picture(p)
    stddraw.show(0)
    a[1].increaseCharge(-2.0)
```

3.2.22. *Хронометраж типа* `complex`. Напишите клиент типа `Stopwatch`, сравнивающий стоимость использования типа `complex` со стоимостью кода, непосредственно манипулирующего двумя значениями типа `float` для вычислений в программе `mandelbrot.py`. А именно: создайте версию программы `mandelbrot.py`, только осуществляющей вычисления (удалите весь

Изменение заряда

код, относящийся к типу `Picture`), затем создайте версию этой программы, не использующую тип `complex`, и вычислите соотношение продолжительностей их выполнения.

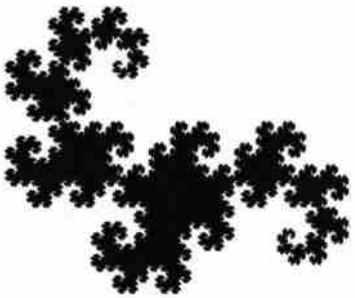
3.2.23. *Кватернионы.* В 1843 году сэр Уильям Роуэн Гамильтон (William Hamilton) обнаружил расширение комплексных чисел — кватернионы. *Кватернион* — это вектор $a = (a_0, a_1, a_2, a_3)$ со следующими операциями:

- *Модуль*: $|a| = \sqrt{a_0^2 + a_1^2 + a_2^2 + a_3^2}$
- *Сопряжение*: сопряжение из $(a_0, -a_1, -a_2, -a_3)$
- *Инверсия*: $a^1 = (a_0 / |a|^2, -a_1 / |a|^2, -a_2 / |a|^2, -a_3 / |a|^2)$
- *Сумма*: $a+b = (a_0 + b_0, a_1, + b_1, a_2, + b_2, a_3, + b_3)$
- *Произведение*: $a*b = (a_0b_0 - a_1b_1 - a_2b_2 - a_3b_3,\ a_0b_1 - a_1b_0 + a_2b_3 - a_3b_2,\ a_0b_2 - a_1b_3 + a_2b_0 + a_3b_1,\ a_0b_3 + a_1b_2 - a_2b_1 + a_3b_0)$
- *Частное*: $a / b = ab^{-1}$

Создайте тип данных для кватернионов и клиент проверки всего вашего кода. Кватернионы расширяют концепцию трехмерного вращения до четырехмерного. Они используются в компьютерной графике, теории контроля, обработке сигналов и орбитальной механике.

3.2.24. *Кривые дракона.* Напишите рекурсивный клиент `Dragon` типа `Turtle`, рисующий кривые дракона (см. упр. 1.2.32 и 1.5.9).

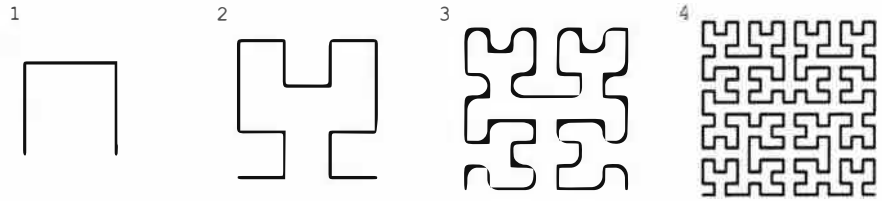
```
% python dragon.py 15
```



Решение. Эти кривые, которые были первоначально обнаружены тремя физиками из NASA, популяризированы в 1960-х годах Мартином Гарднером и впоследствии использованы Майклом Крайтоном в книге и фильме *Парк юрского периода.* Код решения этого упражнения на удивление компактен, он основан на двух взаимодействующих рекурсивных функциях, происходящих непосредственно из условия упражнения 1.2.32. Одна из них, `dragon()`, должна рисовать кривую, как и ожидается; другая, `nogard()`, должна рисовать кривую в *обратном* порядке. См. подробности на сайте книги.

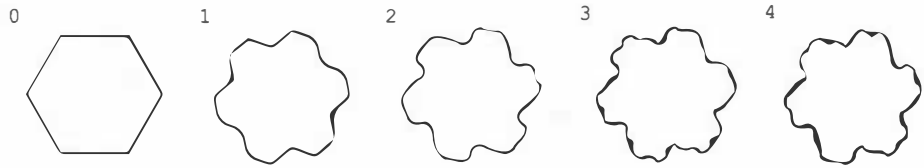
3.2.25. *Кривые Гильберта. Заполняющая пространство кривая* (кривая Пеано) — это непрерывная линия в единичном квадрате, проходящая

через каждую точку. Напишите рекурсивный клиент для типа `Turtle`, создающий фрактальные узоры, приближающиеся к кривой заполнения, определенной математиком Давидом Гильбертом в конце XIX столетия.



Частичное решение. См. предыдущее упражнение. Необходимы два метода: `hilbert()`, рисующий кривую Гильберта, и `treblih()`, рисующий кривую Гильберта *в обратном порядке*. См. подробности на сайте книги.

3.2.26. *Остров Госпера.* Составьте рекурсивный клиент для типа `Turtle`, создающий следующие фрактальные узоры.



3.2.27. *Анализ данных.* Составьте тип данных для проведения экспериментов, где управляющая переменная — целое число в диапазоне [0, *n*), а зависимая переменная имеет тип `float`. К таким экспериментам, например, относится изучение продолжительности работы программы, получающей целочисленный аргумент. Для визуализации полученных статистических данных весьма подходит *график Тьюки* (см. упр. 2.2.17). Реализуйте представленный ниже API.

Для статистических вычислений и получения графики можно использовать функции модуля `stdstats`. Задействовав модуль `stddraw`, клиенты смогут использовать различные цвета в функциях `plot()` и `plotTukey()` (например, светло-серый для всех точек и черный для графика Тьюки). Составьте клиент проверки, рисующий график результатов (вероятность просачивания) для экспериментов с просачиванием (см. раздел 2.4) при увеличении размера таблицы.

API для пользовательского типа данных `Data`

Клиентская операция	Описание
`Data(n)`	Новый объект `Data` для `n` целочисленных значений в `[0, n)`
`d.addDataPoint(i, x)`	Добавляет к `d` точку на графике с абсциссой `i` и ординатой `x`
`d.plot()`	Рисует `d` в окне стандартного графического устройства
`d.plotTukey()`	Отображает график Тьюки для `d` в окне стандартного графического устройства

3.2.28. *Элементы.* Составьте тип данных `Element` для записей периодической таблицы элементов. Включите в тип данных значение элемента, атомный номер, символ и атомный вес, а также методы доступа для каждого из этих значений. Затем составьте тип данных `PeriodicTable`, читающий значения из файла, создающий массив объектов `Element` (сам файл и описание его формата можно найти на сайте книги) и отвечающий на запросы со стандартного устройства ввода, чтобы пользователь мог ввести такую молекулярную формулу, как `H2O`, и получить в ответ, например, молекулярную массу. Разработайте API и реализации для каждого типа данных.

3.2.29. *Курсы акций.* Файл `DJIA.csv` на сайте книги содержит все заключительные курсы акций в истории Промышленного индекса Доу-Джонса (используется формат со значениями, отделенными запятыми). Составьте тип данных `Entry`, способный содержать одну запись из таблицы, со значениями даты, цены при открытии, наибольшей и наименьшей ценой за день, ценой на момент закрытия биржи и т.д. Затем составьте тип данных `Table`, читающий файл и создающий массив объектов `Entry`, а также предоставляющий методы для вычисления среднего за различные промежутки времени. И наконец, создайте клиенты для типа `Table`, рисующие графики для интересующих вас данных. Проявите творчество: этот путь слишком хорошо протоптан.

3.2.30. *Хаос с методом Ньютона.* У полиномиальной функции $f(z) = z^4 - 1$ четыре корня: 1, –1, i, и $-i$. Мы можем найти корни, используя метод Ньютона в комплексной плоскости: $z_{k+1} = z_k - f(z_k) / f'(z_k)$. Здесь $f(z) = z^4 - 1$ и $f'(z) = 4z^3$. Метод сходится к одному из этих четырех корней, в зависимости от отправной точки z_0. Составьте клиент `Newton` для типа `Complex`, получающий аргумент командной строки `n` и закрашивающий пиксели рисунка размером `n` на `n` белым, красным, зеленым или синим цветом. Комплексная точка сопоставляется с пикселем в регулярной квадратной

таблице размером 2 и центром в исходной точке. Каждый пиксель окрашивается согласно тому, к какому из этих четырех корней сходится соответствующая точка (если после 100 итераций никакого сближения нет, выбирается черный цвет).

3.2.31. *Эквипотенциальные поверхности.* Эквипотенциальная поверхность — это набор всех точек с одинаковым электрическим потенциалом. Имея группу точечных зарядов, полезно визуализировать электрический потенциал, нарисовав эквипотенциальные поверхности (известные также как *контурный график*). Составьте программу `equipotential.py`, рисующую линии через каждые 5 В, вычислив потенциал в каждом пикселе и проверив, кратен ли потенциал в этой соответствующей 1 пикселю точке 5 вольтам. *Примечание*: *очень простое* приближенное решение этого упражнения получается из программы 3.1.8 при присвоении каждому пикселю значений цветов вместо пропорциональных полутонов. Например, рисунок ниже получен при вставке предыдущего кода выше места создания объекта `Color`. Объясните, почему это работает, и поэкспериментируйте со своей собственной версией.

3.2.32. *Цветной график Мандельброта.* Создайте файл из 256 целочисленных триплетов, представляющих значения типа `Color`, а затем используйте эти цвета вместо полутоновых значений, чтобы получить все пиксели в программе `mandelbrot.py`. Прочитайте значения и создайте массив из 256 значений типа `Color`, а затем индексируйте его возвращаемым значением функции `mandel()`. Поэкспериментировав с выбором различных цветов в различных местах множества, можно получить удивительные изображения. См. пример `mandel.txt` на сайте книги.

3.2.33. *Множество Жюлиа.* Множество Жюлиа для данного комплексного числа c — это множество точек, связанных с функцией Мандельброта. Вместо того чтобы зафиксировать z и варьировать c, мы фиксируем c и изменяем z. Эти точки z, для которых модифицированная функция Мандельброта остается ограниченной, относятся к *множеству Жюлиа*; а те, для которых

последовательность стремится к бесконечности, не входят в множество. Все интересующие точки z сосредоточены в квадрате 4 на 4 с центром в исходной точке. Множество Жюлиа для c замкнуто, если и только если c находится во множестве Мандельброта! Составьте программу `colorjulia.py`, получающую из командной строки два аргумента, `a` и `b`, а затем рисующую цветную версию графика множества Жюлиа для $c = a + bi$, используя метод таблицы цветов, описанный в предыдущем упражнении.

3.2.34. *Наибольший победитель и наибольший неудачник.* Составьте клиент для класса `StockAccount`, который создает массив объектов `StockAccount`, вычисляет общую стоимость каждой учетной записи и выводит отчет для учетных записей с наибольшей и наименьшей стоимости учетной записи. Подразумевается, что данные в учетной записи хранятся в отдельном файле, содержащем информацию каждой учетной записи, одна за другой, в формате, описанном в тексте.

3.3. Разработка типов данных

Способность создавать типы данных превращает каждого программиста в разработчика языка. Мы не обязаны соглашаться на встроенные в язык типы данных и связанные с ними операции, поскольку легко можем создать собственные типы данных, а затем составлять использующие их клиентские программы. Например, у Python нет встроенного типа для заряженных частиц, но мы вполне можем определить тип данных `Charge` и составить клиентские программы, непосредственно использующие эту абстракцию. Даже когда у языка Python действительно есть специфический тип, мы вполне можем использовать его, приспособив к своим потребностям, как мы делаем, используя свои книжные модули вместо более общих, предоставляемых языком Python.

Программы этого раздела...

Программа 3.3.1. Комплексные числа (`complexpolar.py`) 446
Программа 3.3.2. Счетчик (`counter.py`) 449
Программа 3.3.3. Пространственные векторы (`vector.py`) 456
Программа 3.3.4. Эскиз документа (`sketch.py`) 473
Программа 3.3.5. Обнаружение подобия (`comparedocuments.py`) 475

Главное, что мы вынесли в настоящий момент при создании программ — это понимание необходимых нам типов данных. Выработка этого понимания является *проектным* действием. В этом разделе мы сосредоточимся на разработке API как критически важном этапе в разработке любой программы. Необходимо рассмотреть различные альтернативы, понять их воздействие и на клиентские программы, и на реализации, а затем детализировать проект, чтобы соблюсти баланс между потребностями клиентов и возможностями реализации.

Если пройти курс системного программирования, то можно узнать, что этап проектирования критически важен при создании больших систем и что в языке Python и подобных языках есть мощные высокоуровневые механизмы поддержки многократного использования кода при создании больших программ. Большинство этих механизмов предназначено для использования высококвалифицированными специалистами при построении больших систем, но общий подход доступен для каждого программиста, а некоторые из этих механизмов полезны при создании и небольших программ.

В этом разделе мы рассмотрим инкапсуляцию, неизменность и наследование. Особое внимание мы уделим использованию этих идей при проектировании типов данных, обеспечивающих модульное программирование, облегчающих отладку и составление простого и понятного кода.

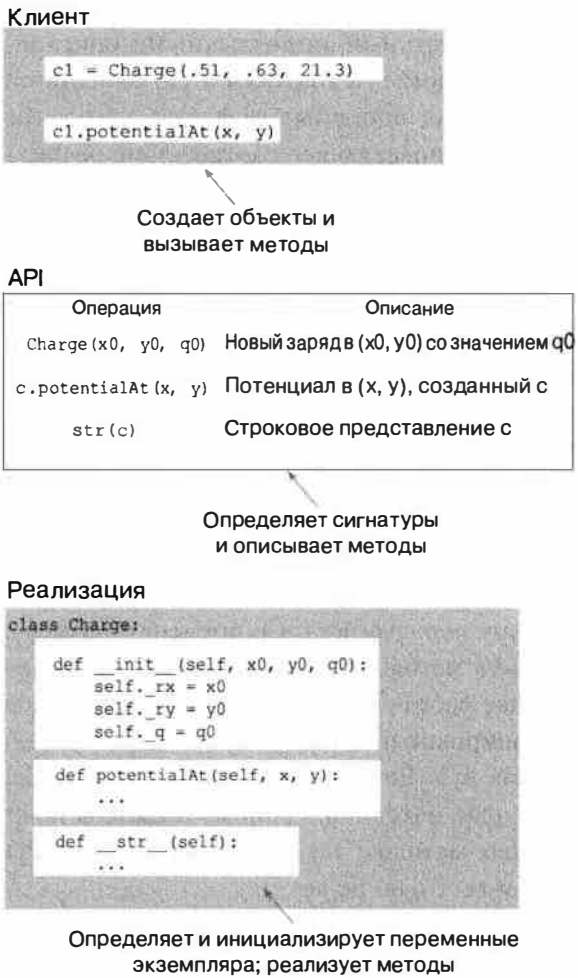
В конце раздела мы обсудим механизмы Python, используемые для проверки проектных предположений по сравнению с фактическими условиями во время

выполнения. Такие инструменты — неоценимые средства при разработке надежного программного обеспечения.

Разработка API. В разделе 3.1 мы составляли клиентские программы, *использующие* эти API, а в разделе 3.2 *реализовали* API. Теперь мы рассмотрим сложности *разработки* API. Обсуждение этих тем в этом порядке и с этим акцентом вполне оправдано, поскольку по большей части вы будете составлять клиентские программы. Основная цель разработки хороших API заключается в том, чтобы упростить клиентский код.

Зачастую самый важный и самый сложный этап создания программного обеспечения — это разработка API. Это требует практики, тщательного обдумывания и множества итераций. Однако любое время, потраченное на разработку хорошего API, бесспорно окупится экономией времени на отладку и многократное использование кода.

Столь тщательная проработка API могла бы показаться чрезмерной при создании маленькой программы, но такое внимание необходимо при создании любой программы, как если бы вы должны были многократно использовать их код, и вовсе не потому, что вы знаете, что будете многократно использовать этот код, а потому, что часть вашего кода может быть многократно использована, причем вы не можете знать, какой именно код понадобится.



Объектно-ориентированная абстракция типа данных

Стандарты. Довольно просто понять, почему соответствие API настолько важно. Известно, что использование общего стандарта находит самое широкое применение в технике, от железнодорожных путей, гаек и болтов, MP3, цифровой

видеозаписи и радиочастот до стандартов Интернета. Сам язык Python является еще одним примером: ваши программы Python — это клиенты *виртуальной машины Python* — стандартного интерфейса, реализуемого широким разнообразием аппаратных и программных платформ. При использовании API для отделения клиента от реализации мы получаем преимущества стандартных интерфейсов для каждой составляемой программы.

Проблемы спецификации. Наши API для типов данных — это наборы методов наряду с краткими англоязычными описаниями того, что эти методы должны делать. В идеале API должен явно описать поведение всех возможных аргументов, включая побочные эффекты, кроме того, необходимо программное обеспечение, проверяющее соответствие реализации спецификации. К сожалению, фундаментальное следствие теоретической информатики, *проблема спецификации* (specification problem), гласит, что эта задача фактически невыполнима. Короче говоря, такая спецификация должна была бы быть написана на таком формальном языке, как язык программирования, и проблема определения, когда две программы осуществляют то же вычисление, математикам известна как *неразрешимое* (unsolvable). (Если вас заинтересовала эта идея, то можете узнать больше о природе неразрешимых проблем и их роли в нашем понимании характера вычислений, пройдя курс теоретической информатики.) Поэтому мы прибегаем к неформальным описаниям с примерами, такими как в тексте, сопутствующим нашим API.

Широкие интерфейсы. Широкий интерфейс (wide interface) обладает чрезмерным количеством методов. Важнейший принцип разработки API — *избежание широких интерфейсов*. Со временем размер API естественно растет, поскольку добавить методы к существующему API довольно просто, а вот удалить их, не нарушая работу существующих клиентов, довольно трудно. В некоторых ситуациях широкие интерфейсы вполне резонны, например, в таких встроенных типах, как `str`. Для сокращения фактической ширины интерфейса используются различные методики. Один из подходов подразумевает включение только независимых методов; так, например, модуль Python `math` включает методы `sin()`, `cos()` и `tan()`, но не `sec()`.

Начинаем с клиентского кода. Одна из главных задач разработки типов данных — упрощение клиентского кода. Поэтому на клиентский код имеет смысл обратить внимание с самого начала, при разработке API. Как правило, это не проблема вообще, поскольку первоочередная причина разработки типа данных, как правило, и заключается в упрощении клиентского кода, который становится слишком громоздким. Когда ваш клиентский код оказывается никуда не годным, один из способов исправить ситуацию подразумевает составление упрощенной версии кода, выражающей вычисление неким высокоуровневым способом, не задействующим подробности кода. Либо, если вы выполнили свою работу хорошо

и создали сжатые комментарии для описания своих вычислений, одной из реальных возможностей будет преобразование комментариев в код.

Помните нашу мантру для типов данных: *всякий раз, когда можете четко разделить данные и связанные с ними задачи в пределах вычисления, так и поступайте.* Безотносительно источника, весьма мудро составлять клиентский код (и разрабатывать API) *перед* работой над реализацией. Создание двух клиентов даже лучше. Начинать с клиентского кода — это один из способов гарантировать, что разработка реализации будет стоить усилий.

Избегайте зависимости от представления. Обычно при разработке API в уме имеется какое-то представление. В конце концов, тип данных — это набор возможных значений и операций, допустимых для этих значений, и нет особого смысла говорить об операциях, не имея представления о значениях. Но это отличается от смысла *представления* значений. Одна из целей типа данных — упрощение клиентского кода за счет избегания деталей и зависимости от конкретного представления. Например, наша клиентская программа для типа `Picture` и модуля `stdaudio` имеет простые абстрактные представления для изображения и звука соответственно. Первоочередное значение API для этих абстракций в том, что они позволяют клиентскому коду игнорировать существенный объем подробностей, находящихся в стандартных представлениях этих абстракций.

Проблемы в проекте API. API может быть *слишком трудно реализовать.* Имеется в виду, что его очень трудно или даже невозможно развить либо *слишком трудно использовать*, т.е. создавать клиентский код, который оказывается сложнее, чем без API. API мог бы оказаться или *слишком узким*, без всех необходимых клиентам методов, или *слишком широким*, включающим много методов, не все из которых необходимы любым клиентам. API может быть или *слишком общим* и не поддерживать полезных абстракций, или *слишком специфическим* и поддерживать абстракции столь подробные или столь распыленные, что они становятся бесполезными. Эти соображения иногда обобщаются в следующую сентенцию: *предоставить клиентам только необходимые методы, и никаких других.*

Приступая к изучению программирования, вы набрали программу `helloworld.py`, не понимая о ней очень многого, кроме производимого результата. С этого момента вы изучаете программирование, имитируя код из книги, и в конечном счете начинаете разрабатывать собственный код для решения различных проблем. То же относится и к проектированию API. В книге доступно множество API, их еще больше на сайте книги и в сетевой документации Python. Вы можете изучить их и использовать понравившиеся при проектировании и разработке собственных API.

Инкапсуляция. Процесс отделения клиентов от реализаций при сокрытии информации — это *инкапсуляция* (encapsulation). Подробности реализации остаются скрытыми от клиентов, а у реализации нет никакого способа узнать детали клиентского кода, который может быть создан в будущем.

Как вы, возможно, заметили, инкапсуляция широко практикуется в наших реализациях типов данных. В разделе 3.1 мы начали повторять мантру: *вы не обязаны знать, как реализован тип данных, чтобы использовать его.* Это выражение описывает одно из главных преимуществ инкапсуляции. Мы полагаем, что это настолько важно, что даже не описываем все остальные способы разработки типов данных. Давайте теперь подробнее обсудим три главных причины использования инкапсуляции.

- Обеспечение модульного программирования.
- Облегчение отладки.
- Упрощение кода программы.

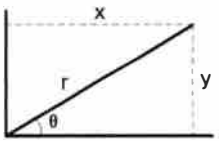
Эти причины взаимосвязаны (хорошо разработанный модульный код проще понять и отладить, чем код исключительно на базе встроенных типов).

Модульное программирование. Модульный стиль программирования, изучаемый нами начиная с функций в главе 2, обусловлен идеей разделения больших программ на небольшие модули, которые могут быть разработаны и отлажены независимо. Этот подход повышает отказоустойчивость нашего программного обеспечения, ограничивая и локализуя воздействие вносимых изменений, и обеспечивает многократное использование кода, позволяя заменять реализации типа данных новыми, чтобы улучшить их производительность, точность или объем занимаемой памяти. Залог успеха модульного программирования в обеспечении *независимости* модулей. Для этого создаются API, являющиеся *единственной* точкой соприкосновения между клиентом и реализацией, — код реализации типа данных может подразумевать, что клиент знает *только* API.

Изменение API. Используя стандартные модули, мы регулярно пользуемся преимуществами инкапсуляции. Например, новые версии Python вполне могут включить новые реализации различных типов данных или модулей, определяющих функции. Причины регулярного улучшения *реализации* типов данных вполне очевидны, а потенциально пользу из этого могут извлечь *все* клиенты. *API Python изменяются редко.* Однако, когда изменения действительно происходят, они обходятся довольно дорого всему сообществу Python — всем приходится модифицировать свои клиенты. В качестве весьма яркого примера рассмотрим изменение значения оператора / для целочисленных операндов в Python версий 2 и 3 (см. стр. 43.). Несмотря на преимущества этого изменения, оно повлекло изменение в API для типа данных `int` (изменено поведение оператора /), что *потребовало переписать, проверить и отладить все*

программы, использующие оператор / с целыми числами! Некоторые посчитали цену этого изменения чрезмерной, что задержало широкое распространение языка Python 3 на десятилетия. Эта ситуация ведет к следующей мантре: *как только количество клиентов модуля станет существенным, не изменяйте API.*

Изменение реализации. Рассмотрим класс `Complex`, определенный в программе 3.3.1 (`complexpolar.py`). Он имеет то же имя и API, что и класс `Complex` из программы 3.2.6, но использует иное представление комплексных чисел. Класс `Complex`, определенный в файле `complex.py`, использует Декартово представление, где переменные экземпляра `_re` и `_im` представляют комплексное число как x + yi. Класс `Complex`, определенный в программе `complexpolar.py`, использует *полярное представление*, где переменные экземпляра `_r` и `_theta` представляют комплексные числа как r(cosθ + i sinθ). В этом представлении r — это *модуль*, а θ — *полярный угол*. Полярное представление интересно потому, что некоторые операции с комплексными числами проще выполнять в полярном представлении. Сложение и вычитание проще в Декартовом представлении, а умножение и деление — в полярном. Как вы узнаете в разделе 4.1, эффективность разных подходов может отличаться весьма существенно. Идея инкапсуляции состоит в том, что можно заменить одну из этих программ другой (по любой причине), *не изменяя клиентский код*, кроме оператора `import`, чтобы использовать `complexpolar` вместо `complex`. Выбор между этими двумя реализациями зависит от клиента. В принципе, единственное различие для клиента должно заключаться в производительности. Эта возможность критически важна по многим причинам. Одна из самых важных в том, что можно постоянно улучшать программное обеспечение: разработка лучшего способа реализации типа данных может быть полезна для всех его клиентов. Вы используете это свойство каждый раз, когда устанавливаете новую версию программного обеспечения, включая сам язык Python.



Полярное представление

Закрытость. Инкапсуляцию поддерживают многие языки программирования. Например, язык Java предоставляет модификатор видимости `private`. Объявив переменную экземпляра (или метод) закрытой, вы лишаете любого клиента (код в другом модуле) возможности непосредственно обращаться к данной переменной экземпляра (или методу). В результате клиенты могут использовать класс только через его открытые методы и конструкторы, т.е. через его API.

Программа 3.3.1. Комплексные числа (`complexpolar.py`)

```
import math
import stdio

class Complex:
    def __init__(self, re=0, im=0):
        self._r     = math.hypot(im, re)
        self._theta = math.atan2(im, re)

    def re(self): return self._r * math.cos(self._theta)
    def im(self): return self._r * math.sin(self._theta)

    def __add__(self, other):
        re = self.re() + other.re()
        im = self.im() + other.im()
        return Complex(re, im)

    def __mul__(self, other):
        c = Complex()
        c._r = self._r * other._r
        c._theta = self._theta + other._theta
        return c

    def __abs__(self): return self.r

    def __str__(self):
        return str(self.re()) + ' + ' + str(self.im()) + 'i'

def main():
    z0 = Complex(1.0, 1.0)
    z = z0
    z = z*z + z0
    z = z*z + z0
    stdio.writeln(z)

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

	Переменные экземпляра
`_r`	Модуль
`_theta`	Полярный угол

Этот тип данных реализует тот же API, что и программа 3.2.6. Он использует те же методы, но с другими переменными экземпляра. Поскольку переменные экземпляра являются закрытыми, эта программа могла бы использоваться вместо программы 3.2.6 без изменения клиентского кода (за исключением оператора `import`).

```
% python complexpolar.py
-7.000000000000002 + 7.000000000000003i
```

Язык Python не предоставляет модификатора закрытой области видимости, а значит, клиенты могут непосредственно обращаться ко всем переменным экземпляра, методам и функциям. Однако сообщество программистов Python придерживается такого соглашения: если имя переменной экземпляра, метода или функции начинается символом подчеркивания, то клиенты должны считать такую переменную экземпляра, метод или функцию закрытыми и не обращаться к ним в клиентском коде непосредственно.

Программисты, следующие соглашению о символе подчеркивания, вполне могут изменять реализацию закрытых функций или методов (или использовать различные закрытые переменные экземпляра) и при этом знать, что ни на какой из клиентов, также следующих соглашению, это никак не повлияет. Например, если в классе `Complex`, определенном в программе 3.2.6 (`complex.py`), переименовать переменные экземпляра `_re` и `_im` (обратите внимание на начальные символы подчеркивания, означающие, что переменные закрыты) в `re` и `im` (без предваряющих символов подчеркивания), то клиент сможет составить код, непосредственно обращающийся к ним. Если `z` — это переменная, ссылающаяся на объект типа `Complex`, то клиент может использовать синтаксис `z.re` и `z.im`, чтобы обратиться к его переменным экземпляра. Но любой подобный клиентский код станет зависящим от данной конкретной реализации API, нарушая главное правило инкапсуляции. Переход на другую реализацию, как в программе 3.3.1, не должен нарушать работу клиента.

Для защиты от таких ситуаций мы в этой книге идем на шаг дальше и делаем *все* переменные экземпляра закрытыми в наших классах. Мы настоятельно рекомендуем это и вам, поскольку нет оправданных причин для доступа к переменным экземпляра непосредственно из клиента. Впоследствии мы рассмотрим некоторые из последствий этого соглашения.

Планирование будущего. Есть много примеров тому, когда отсутствие инкапсуляции в типах данных приводило к существенным расходам на ликвидацию последствий.

- *Проблема 2000-го года.* В конце прошлого тысячелетия многие программы представляли год только двумя десятичными цифрами, для экономии. Такие программы не могли различить 1900 и 2000 годы. К 1 января 2000 года программисты постарались устранить такие ошибки и предотвратить катастрофические отказы, предсказанные многими научными фантастами.
- *Почтовые индексы.* В 1963 году Почтовая служба Соединенных Штатов (USPS) начала использовать почтовые индексы из пяти цифр, чтобы улучшить сортировку и передачу почты. Программисты писали программное обеспечение в расчете, что эти пятизначные коды останутся навсегда. В 1983 году USPS ввела расширенный почтовый индекс — ZIP + 4,

состоящий из первоначального почтового индекса (пять цифр) и четырех дополнительных цифр.

- *Протокол IPv4 против протокола IPv6.* Протокол Интернета (IP) является стандартом, используемым электронными устройствами для обмена информацией по Интернету. Каждому устройству присваивается уникальное целое число или адрес. Протокол IPv4 использует 32-разрядные адреса и обеспечивает приблизительно 4,3 миллиарда адресов. В связи со взрывообразным ростом Интернета протокол новой версии, IPv6, использует 128-битовые адреса и обеспечивает 2^{128} адресов.

В каждом из этих случаев, если бы программисты правильно инкапсулировали данные, изменение внутреннего представления (для приспособления к новому стандарту) распространилось бы на огромный объем клиентского кода (зависимого от старого стандарта). Предполагаемая цена изменений в каждом из этих случаев составляет порядка сотен миллионов долларов! Это огромная цена за отсутствие инкапсуляции одного числа. Эти затруднения могли бы показаться надуманными, но вы можете убедиться, что каждый программист (включая вас), отказавшись от предоставляемой инкапсуляцией защиты, рискует потратить существенные объемы времени и сил на исправление кода после смены стандартов.

Наше соглашение о том, что *все* переменные экземпляра являются закрытыми, обеспечивает некоторую защиту от таких проблем. Если бы этого соглашения придерживались при реализации типов данных для года, почтового индекса, IP-адреса или чего бы то ни было, то внутреннее представление можно было бы изменять, не затрагивая клиентов. *Реализация типа данных* знает представление данных, *объект* содержит сами данные; *клиент* содержит только ссылку на объект и не знает деталей.

API для пользовательского типа данных `Counter`

Операция	Описание
`Counter(id, maxCount)`	Новый счетчик `id`, инициализированный значением `0` и максимальным значением `maxCount`
`c.increment()`	Инкремент `c`, если его значение не `maxCount`
`c.value()`	Значение счетчика `c`
`str(c)`	`'id: value'` (строковое представление счетчика `c`)

Программа 3.3.2. Счетчик *(counter.py)*

```
import sys
import stdarray
import stdio
import stdrandom

class Counter:
    def __init__(self, id, maxCount):
        self._name = id
        self._maxCount = maxCount
        self._count = 0

    def increment(self):
        if self._count < self._maxCount:
            self._count += 1

    def value(self):
        return self._count

    def __str__(self):
        return self._name + ': ' + str(self._count)

def main():
    n = int(sys.argv[1])
    p = float(sys.argv[2])
    heads = Counter('Heads', n)
    tails = Counter('Tails', n)
    for i in range(n):
        if stdrandom.bernoulli(p): heads.increment()
        else: tails.increment()
    stdio.writeln(heads)
    stdio.writeln(tails)

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

	Переменные экземпляра
_name	Имя счетчика
_maxCount	Максимальное значение
_count	Значение

Этот класс инкапсулирует простой целочисленный счетчик, присваивая ему строковое имя и инициализируя значением `0`. Его значение увеличивается при каждом вызове клиентом метода `increment()`. Когда клиент вызывает метод `value()`, он сообщает значение счетчика, а вызов метода `str()` создает строку с именем и значением счетчика.

```
% python counter.py 1000000 .75
Heads: 750056
Tails: 249944
```

Ограничение возможностей для ошибки. Инкапсуляция позволяет также программистам гарантировать, что их код сработает так, как предназначено. В качестве примера рассмотрим еще один триллер: на президентских выборах в 2000 году электронная машина для голосования в округе Волуси штата Флорида выдала за Альберта Гора (Al Gore) *минус* 16 022 голоса. Переменная счетчика не была правильно инкапсулирована в программном обеспечении машины для подсчета голосов! Чтобы понять проблему, рассмотрим тип `Counter` (программа 3.3.2), реализующий простой счетчик согласно API, приведенному ниже. Эта абстракция полезна во многих контекстах, включая, например, электронную машину для подсчета голосов. Он инкапсулирует одно целое число и гарантирует, что единственная операция, которая может быть выполнена с целом числом, — это *инкремент на единицу*. Поэтому оно никогда не сможет стать отрицательным. Задача абстракции данных заключается в *ограничении* операций с данными, а также *изоляции* операций с данными. Например, мы могли бы добавить новую реализацию с возможностью регистрации, чтобы метод `counter.increment()` записывал временную метку для каждого голосования и некую другую информацию, применяемую для проверок на достоверность. Но самая большая проблема в том, что соглашение Python никак не защищает от злонамеренного клиентского кода. Например, где-нибудь в машине для подсчета голосов мог бы быть следующий клиентский код:

```
counter = Counter('Volusia', VOTERS_IN_VOLUSIA_COUNTY)
counter._count = -16022;
```

В языке программирования, поддерживающем инкапсуляцию, такой код даже не будет откомпилирован; а без такой защиты количество голосов у Гора вполне может оказаться отрицательным. Надлежащая инкапсуляция — далеко не исчерпывающее решение проблем защиты машины для голосования, но это уже хорошее начало. Поэтому эксперты по защите даже не рассматривают язык Python как возможный для создания таких приложений.

Понятность кода. Точное определение типа данных улучшает проект, поскольку это ведет к клиентскому коду с более четким определением его вычислений. Вы видели много примеров такого клиентского кода в разделах 3.1 и 3.2, от заряженных частиц до изображений и комплексных чисел. Ключевым качеством хорошего проекта является код, составленный с надлежащими абстракциями, он может быть почти самодокументирован.

Преимущества инкапсуляции подчеркиваются повсюду в этой книге. И сейчас мы резюмируем их здесь в контексте разработки типов данных. Инкапсуляция обеспечивает модульное программирование, позволяющее нам следующее:

- независимую разработку кода реализации и клиента;
- замену реализации улучшенной без нарушения клиентов;

- поддержка клиентов еще не составлена (любой клиент может составить код для API).

Инкапсуляция изолирует также операции с типами данных, обеспечивая следующие возможности.

- Добавление в реализацию проверок на достоверность и других средств отладки.
- Разъяснение клиентского кода.

Правильно реализованный (инкапсулируемый) тип данных дополняет язык Python, позволяя использовать его в любой клиентской программе.

Неизменность. Объект типа данных *неизменяем* (immutable), если его значение не может быть изменено после создания. Все объекты *неизменяемого типа данных* (immutable data type), такого как строка Python, неизменны. В отличие от него, значения объектов *изменяемого* (mutable) типа данных, такого как список или массив Python, предназначены для изменений. Из рассматриваемых в этой главе типов данных `Charge`, `Color` и `Complex` — неизменны, а `Picture`, `Histogram`, `Turtle`, `StockAccount` и `Counter` — изменчивы. Возможность сделать тип данных неизменным — это фундаментальное решение проекта, которое зависит от конкретного случая.

Неизменяемые типы данных. Цель многих типов данных заключается в инкапсуляции значений, которые не должны изменяться. Например, программист, реализующий клиент для типа `Complex`, вполне резонно мог бы ожидать возможности составить такой код, как `z = z0`, чтобы заставить две переменные ссылаться на тот же объект типа `Complex`, точно так же, как два числа типа `float` или `int`. Но если бы тип `Complex` был изменяем и объект, на который ссылается переменная `z`, должен был быть изменен после присвоения `z = z0`, то объект, на который ссылается переменная `z0`, также изменился бы (ведь это псевдонимы того же объекта или две ссылки на тот же объект). Концептуально изменение значения `z` изменило бы значение `z0`! Этот неожиданный результат, *ошибка псевдонимов* (aliasing bug), становится неожиданностью для многих новичков в объектно-ориентированном программировании. Одна из важнейших причин реализации неизменяемых типов — возможность использовать неизменяемые объекты в операторах присвоения и как аргументы или возвращаемые значения из функций без необходимости волноваться об изменении их значений.

Изменяемый	Неизменный
`Picture`	`Charge`
`Histogram`	`Color`
`Turtle`	`Complex`
`StockAccount`	`str`
`Counter`	`int`
`list`	`float`
	`bool`
	`complex`

Изменяемые типы данных. Для многих других типов данных главная цель абстракции заключается в инкапсуляции значений при их изменении. Хорошим

примером является класс Turtle, определенный в программе 3.2.4 (turtle.py). Главная причина, по которой мы используем тип Turtle, — уменьшение размера клиентских программ и избавление их от ответственности за отслеживание изменяющихся значений. Точно так же от типов Picture, Histogram, StockAccount, Counter, а также от типа Python list ожидается изменение значений. В клиенте, где мы передаем объект типа Turtle как аргумент функции или методу (например, koch.py), мы ожидаем, что позиция и ориентация черепахи изменятся.

Массивы и строки. Как разработчик клиентских программ вы уже встречались с этим различием, когда использовали типы Python для массивов и списков (изменяемые) и строк (неизменяемый). При передаче строки в метод (функцию) вы можете не волноваться о том, что метод (функция) изменит последовательность символов в строке. Напротив, когда вы передаете массив в метод (функцию), он вполне может изменить значения элементов массива. Строки Python неизменны, поскольку обычно мы *не хотим*, чтобы значения строк изменились; массивы Python изменяемы, поскольку мы часто *хотим* изменять элементы массива. Бывают также ситуации, когда мы хотим изменять строки и не хотим изменять массивы, как будет описано далее в этом разделе.

Преимущества неизменности. Как правило, неизменяемые типы данных легче использовать и труднее применить неправильно, поскольку область видимости кода, способного изменить значение объекта, намного меньше, чем у изменяемых типов. Использующий неизменяемые типы данных код проще отлаживать, поскольку они гарантируют, что объекты остаются в том же состоянии. При использовании изменяемых типов данных всегда следует беспокоиться о том, где и когда могло бы измениться значение объекта.

Стоимость неизменности. Недостаток неизменности в том, что для каждого значения приходится создавать новый объект. Например, при использовании типа данных Complex выражение z = z*z + z0 задействует создание третьего объекта (чтобы содержать значение z*z), а затем использует этот объект с оператором + (не сохраняя явную ссылку на него) и создает четвертый объект для содержания значения z*z + z0 и присвоения ссылки на него переменной z (в результате исходная ссылка z осиротеет). Такая программа, как 3.2.7 (mandelbrot.py), создает огромное количество промежуточных объектов. Но этот расход — обычный, и система управления памятью Python оптимизирована для таких ситуаций.

Применение неизменности. Некоторые языки обладают прямой поддержкой неизменности. Например, для переменных экземпляра, значения которых никогда не должны изменяются, язык Java предоставляет модификатор final, документирующий запрет изменения значения и предотвращающий их случайную модификацию. Это упрощает программы и их отладку. Смоделировать подобное поведение в языке Python вполне возможно, но давайте оставим такой код для экспертов. В этой книге лучшее, что мы можем сделать, — это заявить

о нашем намерении сделать тип данных неизменным и гарантировать, что в нашем коде реализации (который не всегда так прост, как вы могли бы подумать) значение такого объекта не будет изменяться.

Защитные копии. Предположим, нужно разработать неизменяемый тип данных `Vector`, конструктор которого получает как аргумент массив `float` для инициализации переменных экземпляра. Рассмотрим эту попытку:

```
class Vector:
    def __init__(self, a):
        self._coords = a # массив координат
    ...
```

Этот код делает `Vector` изменяемым типом данных. Клиентская программа могла бы создать объект `Vector`, определив элементы массива, а затем (в обход API) изменить элементы уже после создания:

```
a = [3.0, 4.0]
v = new Vector(a)
a[0] = 17.0 # обход открытых API
```

Переменная экземпляра `_coords` отмечена как закрытая (начальный символ подчеркивания), но тип `Vector` изменяем, поскольку реализация содержит ссылку на тот же массив, что и клиент; если клиент изменит элемент в этом массиве, то клиент изменит также и объект `Vector`.

Чтобы гарантировать неизменность типа данных, включающего переменную экземпляра изменяемого типа, реализация должна сделать локальную *защитную копию* (defensive copy). Как упоминалось в разделе 1.4, выражение `a[:]` создает копию массива `a[]`. Как следствие, этот код создает защитную копию:

```
class Vector:
    def __init__(self, a):
        self._coords = a[:] # массив координат
    ...
```

А теперь рассмотрим полную реализацию такого типа данных.

Неизменность должна быть учтена в любом проекте типа данных. Идеально неизменность типа данных должна быть определена в API, чтобы клиенты знали, что значения объектов не будут изменяться. Реализация неизменяемого типа может быть трудной. Создание защитной копии для сложных типов тоже проблематично, как и гарантия того, что ни один из методов не изменит значения объектов.

Пример: пространственный вектор. Для иллюстрации этих идей в контексте полезной математической абстракции рассмотрим тип данных *вектора* (vector). Как и комплексные числа, базовое определение абстракции вектора общеизвестно, поскольку оно играет центральную роль в прикладной математике уже более ста лет. В свойствах векторов заинтересована также такая область математики,

как *линейная алгебра*. Линейная алгебра — это улучшенная и успешная теория с многочисленными применениями, играющая важную роль во многих областях социальных и естественных наук. Полное описание линейной алгебры, безусловно, не является темой этой книги, но некоторые из ее важнейших приложений основаны на ее элементах, поэтому мы задействуем векторы и линейную алгебру повсюду в книге (например, в основе примера случайной навигации из раздела 1.6 лежит линейная алгебра). Естественно, такую абстракцию стоит инкапсулировать в типе данных.

Пространственный вектор (spatial vector) — это абстрактная сущность, обладающая *модулем* и *направлением*. Пространственные векторы представляют естественный способ описания таких свойств физического мира, как сила, скорость, импульс и ускорение. Вектор обычно представляют как стрелку из начала Декартовых координат к некой точке: направление — это луч из начала координат, а модуль — это длина стрелки (расстояние от начала координат до точки). Чтобы определить вектор, достаточно определить точку.

Пространственный вектор

Эта концепция распространяется на любое количество размерностей: для определения вектора в N-мерном пространстве достаточно упорядоченного списка из n вещественных чисел (координаты N-мерной точки). В соответствии с соглашением, для обозначения вектора мы используем букву, выделенную полужирным шрифтом, а для обозначения его значения — числа или индексированные имена переменной (тот же символ, но курсивом), отделенные запятыми в круглых скобках. Например, мы могли бы использовать x для обозначения вектора $(x_0, x_1..., x_{n-1})$ и y для обозначения вектора $(y_0, y_1..., y_{n-1})$.

API. К базовым операциям с векторами относятся: сложение двух векторов, умножение вектора на скаляр (вещественное число), вычисление скалярного произведения двух векторов, а также вычисление модуля и направления следующим образом:

- *Сложение*: $\mathbf{x} + \mathbf{y} = (x_0 + y_0, x_1 + y_1, ..., x_{n-1} + y_{n-1})$.
- *Скалярное произведение*: $\alpha\mathbf{x} = (\alpha x_0, \alpha x_1, ..., \alpha x_{n-1})$.
- *Произведение векторов*: $\mathbf{x} \cdot \mathbf{y} = x_0y_0 + x_1y_1 ... + x_{n-1}y_{n-1}$.
- *Модуль*: $|\mathbf{x}| = (x_0^2 + x_1^2 + ... + x_{n-1}^2)^{1/2}$.
- *Направление*: $x / |\mathbf{x}| = (x_0 / |\mathbf{x}|, x_1 / |\mathbf{x}|, ..., x_{n-1} / |\mathbf{x}|)$.

Результатом сложения, скалярного произведения и направления являются векторы, а модуля и произведения векторов — скалярные значения (`float`). Например, если $x = (0, 3, 4, 0)$ и $y = (0, -3, 1, -4)$, то $x + y = (0, 0, 5, -4)$, $3x = (0, 9, 12, 0)$, $\mathbf{x} \cdot \mathbf{y} = -5$, $|\mathbf{x}| = 5$ и $x / |\mathbf{x}| = (0, 0{,}6, 0{,}8, 0)$. Вектор направления — *единичный вектор*:

его модуль равен 1. Эти определения приводят к API, представленному ниже. Как и API типа `Complex`, этот не определяет явно, что тип данных неизменен. Однако известно, что разработчики клиентов (вероятно, мыслящие в терминах математических абстракции) будут, конечно, ожидать этого. Возможно, мы не объяснили им в соглашении, что пытаемся защитить их от ошибок применения псевдонимов!

Представление. Как обычно, первый этап разработки реализации — выбор представления данных. Использование массива для содержания Декартовых координат, предоставленных в конструкторе, — это вполне логичный, но не единственный разумный выбор. Действительно, один из фундаментальных принципов линейной алгебры заключается в том, что основанием системы координат является набор из *n* векторов: любой вектор может быть выражен как линейная комбинация набора из *n* векторов, удовлетворяющих определенному условию, — *линейной независимости* (linear independence). Способность изменять системы координат хорошо согласуется с инкапсуляцией. Большинство клиентов вообще не обязано знать ничего о представлении и быть способным работать с объектами `Vector` и его операциями. Если это гарантировано, то реализация может изменять систему координат, не затрагивая клиентский код.

API для пользовательского типа данных `Vector`

Клиентская операция	Специальный метод	Описание
`Vector(a)`	`__init__(self, a)`	Новый объект `Vector` с Декартовыми координатами, взятыми из массива `a[]`
`x[i]`	`__getitem__(self, i)`	`i`-я Декартова координата `x`
`x + y`	`__add__(self, other)`	Сумма `x` и `y`
`x - y`	`__sub__(self, other)`	Разница `x` и `y`
`x.dot(y)`		Произведение `x` и `y`
`x.scale(alpha)`		Скалярное произведение `alpha` и `x`
`x.direction()`		Единичный вектор с тем же направлением, что и `y` `x` (если `x` — нулевой вектор, происходит ошибка)
`abs(x)`	`__abs__(self)`	Модуль `x`
`len(x)`	`__len__(self)`	Длина `x`
`str(x)`	`__str__(self)`	Строковое представление `x`

Программа 3.3.3. Пространственные векторы (vector.py)

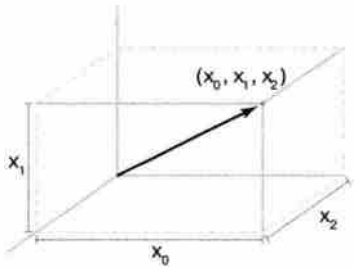
```
import math
import stdarray
import stdio
class Vector:
    def __init__(self, a):
        self._coords = a[:]
        self._n = len(a)
    def __add__(self, other):
        result = stdarray.create1D(self._n, 0)
        for i in range(self._n):
            result[i] = self._coords[i] + other._coords[i]
        return Vector(result)
    def dot(self, other):
        result = 0
        for i in range(self._n):
            result += self._coords[i] * other._coords[i]
        return result
    def scale(self, alpha):
        result = stdarray.create1D(self._n, 0)
        for i in range(self._n):
            result[i] = alpha * self._coords[i]
        return Vector(result)
    def direction(self):      return self.scale(1.0 / abs(self))
    def __getitem__(self, i): return self._coords[i]
    def __abs__(self):        return math.sqrt(self.dot(self))
    def __len__(self):        return self._n
    def __str__(self):        return str(self._coords)
```

	Переменные экземпляра
_coords[]	Декартовы координаты
_n	Размерность

Эта реализация инкапсулирует математическую пространственно-векторную абстракцию в неизменяемом типе данных Python. Программы 3.3.4 (sketch.py) и 3.4.1 (body.py) являются типичными клиентами. Реализации клиента проверки и функции __sub__() оставлены для упражнений 3.3.5 и 3.3.6.

Реализация. При наличии представления код реализации всех этих операций довольно прост, как можно заметить в классе Vector, определенном в программе 3.3.3 (vector.py). Конструктор создает защитную копию клиентского массива, и ни один из методов не присваивает значений этой копии, чтобы объекты типа Vector были неизменны. Обратите внимание на простоту реализации x[i] (__getitem__(self, i)) в нашем Декартовом представлении: достаточно возвратить соответствующую координату в массиве. Это фактически реализует математическую функцию, определяемую для любого представления

вектора — геометрическую проекцию на i-ю Декартову ось.



Проекции вектора (трехмерного)

Как можно гарантировать неизменность, когда клиенту, казалось бы, ничто не мешает составить такой код, как `x[i] = 2.0`? Ответ на этот вопрос кроется в специальном методе, который мы *не реализуем* в неизменном типе данных: в таком случае Python вызывает специальный метод `__setitem__()` вместо метода `__getitem__()`. Поскольку тип `Vector` не реализует этот метод, такой клиентский код привел бы во время выполнения к ошибке `AttributeError`. Далее в этом разделе мы рассмотрим специальные методы Python, чтобы у вас было некоторое представление о них на случай возникновения подобных проблем.

Зачем такие сложности с типом данных `Vector`, когда все эти операции легко реализуются с массивами? На настоящий момент ответ на этот вопрос должен быть для вас очевиден: для поддержки модульного программирования, облегчения отладки и упрощения кода. Массив — это низкоуровневый механизм Python, допускающий все виды операций. Ограничивая операции с типом `Vector` только указанными в API (единственными необходимыми для многих клиентов), мы упрощаем процесс разработки, реализации и поддержки наших программ. Поскольку тип неизменен, мы можем использовать его как любой встроенный тип (`int`, `float`, `bool` и `str`). Например, когда мы передаем объект типа `Vector` функции, мы гарантируем, что его значение не будет изменено, но с массивом у нас нет такой гарантии. Создание программ, использующих объекты типа `Vector` и операции с ними, — это простой и естественный способ использования в своих интересах обширного набора математических знаний, разработанных для этой абстрактной концепции.

Кортежи. Встроенный тип данных Python `tuple` представляет *неизменяемую* последовательность объектов. Он подобен встроенному типу данных `list` (который мы используем для массивов), за исключением того, что после создания кортежа изменить его элементы нельзя. В ситуациях, когда необходимо изменять элементы в последовательности (обратный порядок, перетасовка), следует использовать массивы; в ситуациях, когда элементы не должны изменяться (как координаты в нашем типе данных `Vector`), следует использовать кортежи.

Для манипулирования кортежами можно использовать синтаксис, знакомый по массивам, как указано в API ниже. Для создания кортежа можно использовать либо встроенную функцию `tuple()`, либо список выражений, отделенных запятыми и (необязательно) заключенных в скобки.

Использование кортежей позволяет улучшить проект программы. Например, если заменить первый оператор в конструкторе `vector.py` на

```
self._coords = tuple(a)
```

то любая попытка изменить координату вектора в пределах класса `Vector` приведет во время выполнения к ошибке `TypeError`, обеспечив неизменность объектов типа `Vector`.

```
a = (1, 7, 3)
```

Примеры операций с типом `tuple`

Операция	Комментарий
`len(a)`	3
`a[1]`	7
`a[1] = 9`	Ошибка времени выполнения
`a += [8]`	Ошибка времени выполнения
`(x, y, z) = a`	Распаковка кортежа
`b = (y, z, x)`	Упаковка кортежа
`b[1]`	3

Язык Python предоставляет также такие мощные средства, как *упаковка кортежа* (присвоение кортежа) и *распаковка кортежа*, позволяющие применять кортеж в выражении с правой стороны от оператора присвоения, где переменные кортежа находятся слева (предоставляемое количество переменных слева должно соответствовать количеству выражений справа). Это средство можно использовать для присвоения нескольких переменных одновременно. Например, следующий оператор обменивает объектные ссылки переменных `x` и `y`:

```
x, y = y, x
```

Упаковку и распаковку кортежа можно также использовать для возвращения нескольких значений из функции (см. упр. 3.3.14).

Часть API для встроенного типа данных Python `tuple`

Операция	Комментарий
`len(a)`	Длина `a`
`a[i]`	`i`-й элемент `a`
`for v in a:`	Перебор элементов `a`

Полиморфизм. Составляя методы (или функции), мы зачастую намереваемся использовать их с объектами только определенных типов, но иногда мы хотим использовать их с объектами разных типов. Метод (или функция), способный получать аргументы различных типов, *полиморфен*.

Наилучший вид полиморфизма — неожиданный: когда вы применяете существующий метод / функцию к новому типу данных (что никогда и не планировалось) и внезапно обнаруживаете, что у метода / функции именно то поведение, которое и нужно. Наихудший вид полиморфизма — также неожиданный: когда вы применяете существующий метод / функцию к новому типу данных и внезапно получаете непредвиденный результат! Поиск подобной ошибки может быть чрезвычайно сложным.

Утиная типизация. Утиная типизация (duck typing) — это стиль программирования, при котором язык формально не определяет требования для аргументов функции; вместо этого осуществляется попытка вызова функции, удачная при совместимости (в противном случае во время выполнения происходит ошибка). Название взято из цитаты, приписываемой поэту Джеймсу Уиткому Рили (J. W. Riley):

Если это выглядит, как утка, плавает, как утка,
и крякает, как утка, то, вероятно, это утка.

В языке Python, если объект выглядит, как утка, плавает, как утка, и крякает, как утка, то его можно обработать, как утку; его даже не нужно явно объявлять уткой. Во многих языках (таких, как Java или C++) *необходимо* явно объявлять типы переменных, но не в языке Python, — здесь используется утиная типизация для всех операций (вызов функций, метода или оператора). Если операция неприменима к объекту, то во время выполнения произойдет ошибка `TypeError`, свидетельствующая о несоответствии типа. Один из принципов утиной типизации состоит в том, что метод / функция не должны заботиться о типе объекта, а только о том, может ли клиент выполнять желаемые операции с этим объектом. Таким образом, если все операции в пределах метода / функции могут быть применены к типу, то метод / функция применимы к этому типу. Такой подход ведет к простому и гибкому клиентскому коду, перенося основное внимание на фактически используемые операции, а не на тип.

Недостатки утиной типизации. Главный недостаток утиной типизации в невозможности точно узнать контракт между клиентом и реализацией, особенно когда необходимый метод требуется только косвенно. В API этой информации просто нет. Такая нехватка информации может привести к ошибкам времени выполнения. Хуже того, конечный результат может быть неправильным семантически, вообще без передачи сообщения об ошибке. Далее мы рассмотрим простой пример этой ситуации.

Показательный пример. Мы разрабатываем свой тип данных `Vector` согласно косвенным предположениям, что компоненты вектора будут иметь тип `float` и что клиент будет создавать новый вектор, передавая конструктору массив объектов типа `float`. Если клиент создает два вектора, `x` и `y`, именно так, то `x[i]` и `x.dot(y)` возвратят тип `float`, а `x + y` и `x - y` возвратят векторы с компонентами типа `float`, как и ожидается.

Предположим теперь, что вместо этого клиент создает объект `Vector` с целочисленными элементами, передав конструктору массив объектов типа `int`. Если клиент создаст два вектора, `x` и `y`, таким способом, то `x[i]` и `x.dot(y)` возвратят целые числа, а `x + y` и `x - y` возвратят векторы с целочисленными элементами, как и нужно. Конечно, функции `abs(x)` и `x.direction()` возвращают вектор с компонентами типа `float`. Это лучший вид полиморфизма, когда благодаря счастливому стечению обстоятельств утиная типизация работает правильно.

Теперь предположим, что клиент создает вектор с комплексными элементами, передав конструктору массив объектов типа `complex`. Со сложением векторов и скалярным умножением никаких проблем нет, но реализация операции произведения векторов (наряду с реализациями операций модуля и направления, зависящих от произведения векторов) *безуспешна*. Например:

```
a = [1 + 2j, 2 + 0j, 4 + 0j]
x = Vector(a)
b = abs(x)
```

Во время выполнения этот код завершается ошибкой `TypeError`, поскольку вызов `math.sqrt()` попытается взять квадратный корень комплексного числа. Проблема в том, что произведение двух векторов, $\boldsymbol{x}$ и $\boldsymbol{y}$, с комплексными значениями потребует взятия *комплексно сопряженной величины* из элементов второго вектора:

$$\mathbf{x} \cdot \mathbf{y} = x_0\overline{y}_0 + x_1\overline{y}_1 + \ldots + x_{n-1}\overline{y}_{n-1}.$$

В нашем примере специальный метод `__abs__()` в типе `Vector` вызывает метод `x.dot(x)`, который должен был бы вычислить результат

$$(1 + 2i)\,(1 - 2i) + 2 \cdot 2 + 4 \cdot 4 = 25,$$

но фактически вычисляется

$$(1 + 2i)\,(1 + 2i) + 2 \cdot 2 + 4 \cdot 4 = 17 + 4i.$$

Получившееся комплексное число приведет к ошибке `TypeError` в функции `math.sqrt()`. Конечно, внезапная ошибка времени выполнения — это плохо, но предположим, что последней строкой кода была `b = abs(x.dot(x))`. В этом случае `b` присваивается целое число `33` (а предполагалось `25`), *и нет никакого сообщения о проблеме вообще*. Будем надеяться, что такой код не будет использован в программе управления атомной электростанцией или ракетой "земля–воздух". Вычисление с неправильным результатом бесспорно способно нанести ущерб.

Если мы узнаем о проблеме, то можем ее решить. Чтобы сделать тип `Vector` совместимым с комплексными числами, достаточно внести два изменения, одно в метод `dot()`, другое — в метод `__abs__()`:

```
def dot(self, other):
    result = 0
    for i in range(self._length):
        result += self._coords[i] * other._coords[i].conjugate()
```

```
    return result

def __abs__(self):
    return math.sqrt(abs(self.dot(self)))
```

Модификация `dot()` сработает потому, что все числовые типы Python включают метод `conjugate()`, возвращающий комплексно сопряженную величину числа (являющееся самим числом, если это целое или вещественное число). Модификация `__abs__()` необходима потому, что функция `math.sqrt()` передает сообщение об ошибке `TypeError`, если ее аргумент имеет тип `complex` (даже если его мнимая часть — нуль). Произведение комплексного числа с самим собой гарантированно будет неотрицательным вещественным числом, но это комплексное число, поэтому мы берем его модуль (`float`), используя функцию `abs()`, прежде чем извлечь квадратный корень.

В данном случае утиная типизация — самый плохой вид полиморфизма. Конечно, клиент вполне резонно будет ожидать, что реализация типа `Vector` будет работать правильно, даже когда компонентами вектора будут комплексные числа. Как реализация может упредить и подготовиться ко всем потенциальным случаям использования типов данных? Эту проблему проекта решить невозможно. Все, что мы можем сделать, — это предупредить о необходимости проверять, могут ли все используемые вами типы данных обрабатывать те типы данных, которые вы намереваетесь использовать с ними.

Перегрузка. Возможность определить тип данных, предоставляющий собственные определения операторов, — это одна из форм полиморфизма, известная как *перегрузка оператора* (operator overloading). В языке Python вы можете перегрузить почти каждый оператор, включая арифметические операторы, операторы сравнений, индексирования и разделения. Можно также перегрузить *встроенные функции*, включая абсолютное значение, длину, хеширование и преобразование типов. Перегрузка операторов и встроенных функций позволяет пользовательским типам вести себя, как встроенные типы.

Специальные методы. Механизм поддержки перегрузки Python ассоциирует специальный метод с определенным оператором или встроенной функцией. Хотя вы уже познакомились с использованием специальных методов на нескольких примерах реализации типов данных (в частности, `Complex` и `Vector`), кратко повторим эту концепцию здесь.

Для выполнения операций Python внутренне преобразует выражение в вызов соответствующего специального метода; для вызова встроенной функции Python внутренне вызывает вместо него соответствующий специальный метод. Чтобы перегрузить оператор или встроенную функцию, в реализацию включают соответствующий специальный метод с собственным кодом. Например, всякий раз, когда Python встречает в клиентском коде выражение `x + y`, он преобразует его

в вызов специального метода `x.__add__(y)`. Таким образом, для перегрузки оператора + в вашем типе данных достаточно включить в него реализацию специального метода `__add__()`. Точно так же, когда Python встречает в клиентском коде выражение `str(x)`, он преобразует его в вызов специального метода `x.__str__()`. Таким образом, для перегрузки встроенной функции `str()` достаточно включить реализацию специального метода `__str__()`.

Арифметические операторы. В языке Python такие арифметические операции, как `x + y` и `x * y`, применимы не только для целых и вещественных чисел. Python ассоциирует специальный метод с каждым из арифметических операторов, поэтому вы можете перегрузить любую арифметическую операцию, реализовав соответствующий специальный метод, как показано в таблице ниже.

Специальные методы для арифметических операторов

Клиентская операция	Специальный метод	Описание
`x + y`	`__add__(self, other)`	Сумма x и y
`x - y`	`__sub__(self, other)`	Разница x и y
`x * y`	`__mul__(self, other)`	Произведение x и y
`x ** y`	`__pow__(self, other)`	x в степени y
`x / y`	`__truediv__(self, other)`	Частное x и y
`x // y`	`__floordiv__(self, other)`	Неполное частное x и y
`x % y`	`__mod__(self, other)`	Остаток при делении x на y
`+x`	`__pos__(self)`	x
`-x`	`__neg__(self)`	Арифметическое отрицание x

Примечание. Python 2 использует `__div__` вместо `__truediv__`.

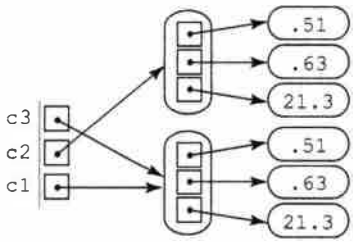
Равенство. Проверяющие равенство операторы `==` и `!=` требуют особого внимания. Рассмотрим, например, код в схеме, приведенной ниже, где создаются два объекта типа `Charge`, на которые ссылаются три переменные: `c1`, `c2` и `c3`. Как показано на схеме, переменные `c1` и `c3` ссылаются на тот же объект, отличный от объекта, на который ссылается переменная `c2`. Понятно, что `c1 == c3` дает `True`, но как насчет `c1 == c2`? Ответ на этот вопрос неясен, поскольку равенство в Python рассматривается двояко:

```
c1 = Charge(.51, .63, 21.3)
c2 = Charge(.51, .63, 21.3)
c3 = c1
```

- *Ссылочное равенство (равенство идентификаторов).* При ссылочном равенстве две ссылки считаются равными, когда они ссылаются на тот же объект. Встроенная функция `id()` возвращает идентификатор объекта (его адрес памяти), а операторы `is` и `is not` проверяют, ссылаются ли две переменные на тот же объект. Таким образом, оператор `c1 is c2` проверяет совпадение результатов вызовов `id(c1)` и `id(c2)`. В нашем примере `c1 is`

c3 дает True, как и ожидалось, а c1 is c2 дает False, поскольку c1 и c2 располагаются в памяти по разным адресам.

- *Объектное равенство (равенство значений).* При объектном равенстве два объекта считаются равными, когда совпадают значения их типа данных. Операторы == и !=, определенные с использованием специальных методов __eq__() и __ne__(), применяют для проверки равенства объектов. Если не определить метод __eq__(), то Python сам предоставит оператор is. Таким образом, стандартный оператор == реализует ссылочное равенство. Поэтому в нашем прежнем примере c1 == c2 дало False, даже при том, что c1 и c2 имеют те же значения позиции и заряда. Если мы хотим считать равными два заряда с идентичными значениями позиции и заряда, то можем обеспечить этот результат, включив следующий код в программу 3.2.1 (charge.py):



Три переменные, ссылающиеся на два объекта типа Charge

```
def __eq__(self, other):
    if self._rx != other._rx: return False
    if self._ry != other._ry: return False
    if self._q != other._q: return False
    return True

def __ne__(self, other):
    return not __eq__(self, other)
```

Теперь при наличии этого кода выражение c1 == c2 дает True в нашем примере. В данном случае, вполне типичном, совсем не очевидно, ожидают ли клиенты именно такого поведения (поэтому мы и не включали эти методы в нашу реализацию). Решение об объектном равенстве — это важное (и трудное) проектное решение для любого типа данных.

Для проверки своего понимания этих идей изучим приведенный ниже интерактивный сеанс Python. В данном случае переменные a и b ссылаются на разные объекты типа int с тем же значением, а a и c ссылаются на тот же объект типа int.

```
>>> a = 123456789
>>> b = 123456789
>>> c = a
>>> a == b
True
>>> a == c
True
>>> id(a)
140461201279248
```

```
>>> id(b)
140461201280816
>>> id(c)
140461201279248
>>> a is b
False
>>> a is c
True
```

Хеширование. Теперь рассмотрим *хеширование* (hashing) — связанную с проверкой равенства фундаментальную операцию сопоставления объекта с целым числом — *хеш-кодом* (hash code). Эта операция настолько важна, что Python предоставляет для нее специальный метод `__hash__()`, поддерживающий встроенную функцию `hash()`. Объект считается *хешируемым* (hashable), если он удовлетворяет следующим трем требованиям:

- объект допускает сравнение с другими объектами при помощи оператора `==`;
- всякий раз, когда два объекта сравниваются на равенство, у них одинаковый хеш-код;
- на протяжении существования объекта его хеш-код не изменяется.

Например, в интерактивном сеансе Python, приведенном ниже, переменные `a` и `b` ссылаются на равные объекты типа `str`. Таким образом, у них должен быть одинаковый хеш-код; переменные `a` и `c` ссылаются на разные объекты типа `str`, поэтому мы ожидаем, что их хеш-коды будут разными.

В типичных приложениях мы используем хеш-код для сопоставления объекта `x` с целым числом в небольшом диапазоне, скажем, от `0` до `m-1`, используя *хеш-функцию* (hash function)

```
hash(x) % m
```

Впоследствии значение хеш-функции можно использовать как целочисленный индекс в массиве длиной `m` (см. программы 3.3.4 и 4.4.3). По определению, у равных хешируемых объектов тот же хеш-код, а следовательно, то же значение хеш-функции. У не равных объектов может быть то же значение хеш-функции, но мы ожидаем, что хеш-функция разделит объекты на m групп примерно равной длины. Все неизменяемые типы данных Python (включая `int`, `float`, `str` и `tuple`) являются хешируемым и спроектированы так, чтобы распределять объекты разумным способом.

```
>>> a = 'Python'
>>> b = 'Python'
>>> c = 'programmer'
>>> hash(a)
-2359742753373747800
>>> hash(b)
-2359742753373747800
>>> hash(c)
7354308922443094682
```

Пример хеша строки

Пользовательский тип данных можно сделать хешируемым, реализовав два специальных метода: `__hash__()` и `__eq__()`. Создание хорошей хеш-функции требует грамотной комбинации науки и техники и в этой книге не рассматривается. Вместо этого мы опишем простой

способ, как это сделать в Python, эффективно и применимо в широком разнообразии ситуаций.

- Удостовериться в неизменности типа данных.
- Реализовать метод `__eq__()`, сравнив все существенные переменные экземпляра.
- Реализовать метод `__hash__()`, поместив те же переменные экземпляра в кортеж и вызвав встроенную функцию `hash()` кортежа.

Вот, например, реализация метода `__hash__()` для типа данных `Charge` (программа 3.2.1), сопровождающая реализацию только что рассмотренного метода `__eq__()`:

```
def __hash__(self):
    a = (self._rx, self._ry, self._q)
    return hash(a)
```

Операции по проверке равенства

Клиентская операция	Специальный метод	Описание
`x is y`	[не может быть перегружен]	Ссылаются ли `x` и `y` на тот же объект?
`x is not y`	[не может быть перегружен]	Ссылаются ли `x` и `y` на разные объекты?
`id(x)`	[не может быть перегружен]	Идентификатор (адрес памяти) `x`
`hash(x)`	`__hash__(self)`	Хеш-код `x`
`x == y`	`__eq__(self, other)`	Равны ли `x` и `y`?
`x != y`	`__ne__(self, other)`	Не равны ли `x` и `y`?

Операторы сравнения. В Python операторы сравнения, такие как `x < y` и `x >= y`, также применимы не только для целых и вещественных чисел или строк. С каждым из операторов сравнения Python также ассоциирует специальный метод, поэтому вы можете перегрузить любой оператор сравнения, реализовав соответствующий специальный метод, как описано в следующей таблице. Кстати, если вы определяете любой из методов сравнения, то должны определить и все остальные единообразным способом. Например, если `x < y`, то вы должны гарантировать, что и `y > x` и `x <= y`. Кроме того, в любом типе данных, где клиент мог бы ожидать сортировки объектов, методы сравнения должны определять *полный порядок* (total order). А именно: должны выполняться три условия.

- *Антисимметрия.* Если и `x <= y` и `y <= x`, то `x == y`.
- *Транзитивность.* Если и `x <= y` и `y <= z`, то `x <= z`.
- *Совокупность.* Или `x <= y`, или `y <= x`.

Тип данных *сравним* (comparable), или сопоставим, если он реализует шесть методов сравнения и они определяют полный порядок. Такие встроенные типы Python, как `int`, `float` и `str`, сравнимы. Чтобы сделать сравнимым

пользовательский тип, необходимо реализовать шесть специальных методов, как мы сделали для типа `Counter` в программе 3.3.2:

```
def __lt__(self, other): return self._count <  other._count
def __le__(self, other): return self._count <= other._count
def __eq__(self, other): return self._count == other._count
def __ne__(self, other): return self._count != other._count
def __gt__(self, other): return self._count >  other._count
def __ge__(self, other): return self._count >= other._count
```

Мы рассмотрим несколько применений сопоставимых объектов в разделе 4.1 (сортировка) и разделе 4.3 (таблицы идентификаторов).

Специальные методы для операторов сравнения

Клиентская операция	Специальный метод	Описание
`x < y`	`__lt__(self, other)`	x меньше, чем y?
`x <= y`	`__le__(self, other)`	x меньше или равен y?
`x >= y`	`__ge__(self, other)`	x больше или равен y?
`x > y`	`__gt__(self, other)`	x больше, чем y?

Другие операторы. Почти каждый оператор в Python может быть перегружен. Например, вы вполне можете перегрузить такие операторы присвоения, как `+=` и `*=`, или такие побитовые логические операторы, как `&`, `|` и `^`. Если вы хотите перегрузить такой оператор, то можете разыскать соответствующий специальный метод на сайте нашей книги или в сетевой документации Python. Существует множество операций обработки строк, массивов и подобных типов данных (например, оператор `[]` и соответствующий специальный метод `__getitem__()`, использованный в типе `Vector`), которые мы рассмотрим подробнее в разделе 4.4.

Встроенные функции. В каждом разрабатываемом классе мы перегружаем встроенную функцию `str()`, но есть еще несколько встроенных функций, которые мы можем перегрузить таким же образом. Функции, используемые в этой книге, приведены в таблице ниже. Мы использовали уже почти все эти функции, за исключением функции `iter()`, рассматриваемой в разделе 4.4.

Специальные методы для встроенных функций

Клиентская операция	Специальный метод	Описание
`len(x)`	`__len__(self)`	Длина x
`float(x)`	`__float__(self)`	Вещественный эквивалент x
`int(x)`	`__int__(self)`	Целочисленный эквивалент x
`str(x)`	`__str__(self)`	Строковое представление x
`abs(x)`	`__abs__(self)`	Абсолютное значение x
`hash(x)`	`__hash__(self)`	Целочисленный хеш-код x
`iter(x)`	`__iter__(self)`	Итератор для x

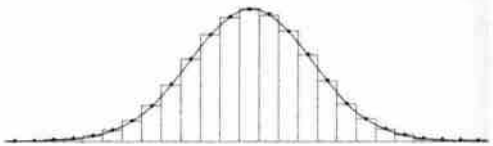
Способность перегружать операторы и встроенные функции — это мощнейшее достоинство языка Python. Оно позволяет программисту создавать пользовательские типы, обладающие поведением, подобным поведению встроенных типов. Однако большие возможности — это большая ответственность. В то время как перегрузка арифметических операторов для знакомых математических сущностей вполне естественна (таких, как комплексные числа или пространственные векторы), перегрузка арифметических операторов для нематематических типов данных иногда неуместна, поскольку от этого может пострадать понятность кода.

Например, разработчику компьютерной графики мог бы понадобиться способ вычисления цветов при определении операторов `a + b` или `a * b`, когда `a` и `b` — объекты типа `Color`; или музыкант мог бы пожелать испытать нечто подобное для тонов или аккордов. Такой креатив вас вряд ли обескуражит, но действительно ли имеют смысл добавление, вычитание и умножение объектов таких типов, как `Turtle`? Использование обычных методов (со смысловыми именами) может быть лучшим подходом. Фактически множество языков программирования (таких, как Java) не поддерживает перегрузку операторов, поскольку программисты зачастую злоупотребляют этой возможностью. Корректная перегрузка операторов, способная потребовать существенных объемов работ, имеет смысл только тогда, когда она ведет к естественному и ожидаемому поведению.

Функции — это объекты. В языке Python "все" является объектами, включая функции. Это значит, что функции можно использовать как аргументы функций и возвращать их как результаты. Определение так называемых *функций высшего порядка* (higher-order function), манипулирующих другими функциями, весьма распространено и в математике, и в научных вычислениях. Например, *производная* (derivative) — это функция, получающая как аргумент функцию и создающая другую функцию. В научных вычислениях нередко необходимо дифференцировать и интегрировать функции, находить корни функций и т.д.

Рассмотрим, например, задачу вычисления *интеграла Римана* — положительной вещественной функции f (площадь ниже кривой) в интервале (a, b). Это вычисление известно как *квадратура*, или *численное интегрирование*. Для вычисления квадратуры было разработано много методов. Вероятно, самый простой из них — *метод прямоугольников*, когда приближенное значение интеграла вычисляется как сумма площадей n прямоугольников равной ширины ниже кривой. Приведенная ниже функция `integrate()` вычисляет интеграл вещественной функции $f()$ в интервале (a, b), используя метод прямоугольников для n прямоугольников.

```
def square(x):
    return x*x

def integrate(f, a, b, n=1000):
    total = 0.0
    dt = 1.0 * (b - a) / n
    for i in range(n):
        total += dt * f(a + (i + 0.5) * dt)
    return total
```



Аппроксимация интеграла

Первообразная x^2 — это $x^3 / 3$, таким образом, интеграл от 0 до 10 составит 1000 / 3. Вызов функции `integrate(square, 0, 10)` даст `333.33324999999996`, что является правильным ответом для шести цифр точности. Аналогично вызов `integrate(gaussian.pdf, -1, 1)` даст `0.6826895727940137` — правильный ответ для семи цифр точности (вспомните Гауссову функцию плотности вероятностей и программу 2.2.1).

Квадратура — не всегда самый эффективный или точный способ вычисления функции. Например, функция `gaussian.cdf()` в программе 2.2.1 — это более быстрый и точный способ интегрирования Гауссовой функции плотности вероятностей. Но квадратура важна потому, что она применима для любой функции вообще, вопрос только в технических условиях и плавности.

Наследование. Python обеспечивает языковую поддержку для определения таких отношений между классами, как *наследование* (inheritance). Разработчики программ используют наследование весьма широко, поэтому если вы берете курс на разработку программного обеспечения, изучите его внимательно. Эффективное использование наследования выходит за рамки рассмотрения этой книги, но мы кратко описываем его здесь, поскольку вы можете встретиться с ним в некоторых ситуациях.

При правильном использовании наследование представляет собой одну из форм многократного использования кода — *создание производных классов* (subclassing). Это мощная методика, позволяющая программисту изменить поведение класса и добавить функциональные возможности, не переделывая весь класс с самого начала. Идея заключается в том, чтобы определить новый *производный класс* (subclass или derived class), наследующий переменные экземпляра и методы из другого, *базового класса* (superclass или base class). Производный класс содержит больше методов, чем базовый. Системные программисты используют наследование для построения *модулей расширения* (extensible module). Идея в том, что один программист (даже вы) может добавить методы в класс, созданный другим программистом (или группой системных программистов), и фактически многократно использовать код в потенциально огромном модуле. Этот подход широко используется, особенно при

разработке пользовательских интерфейсов, когда для обеспечения всех ожидаемых пользователем средств (раскрывающиеся меню, копирование и вставка, обращение к файлам и т.д.) требуется большой объем кода, который желательно использовать многократно.

Важнейший из случаев использования наследования в Python — это *числовая башня* (numeric tower), реализующая числовые типы Python в модуле `numbers.py`. Этот модуль включает классы `Integral`, `Rational`, `Real` и `Complex`, структурированные в соответствии с иерархией наследования. Например, класс `Real` включает (наследует) все методы класса `Complex` плюс методы работы с вещественными числами, такие как операторы сравнения <, <=, >, и >=. Встроенные числовые типы — это производные классы, например, класс `int` происходит от класса `Integral`, а класс `float` — от класса `Real`.

Несмотря на все преимущества, использование наследования вызывает споры даже среди системных программистов. Мы не используем его в этой книге, так как оно отрицает инкапсуляцию. Наследование затрудняет модульное программирование по двум причинам. Во-первых, любое изменение в базовом классе затрагивает все производные классы. Производный класс не может быть разработан *независимо* от базового класса; действительно, он *полностью зависит от* базового класса. Это известно как *проблема хрупкости базового класса* (fragile base class problem). Во-вторых, код производного класса, имея доступ к переменным экземпляра, способен извратить намерение кода базового класса. Например, разработчик такого класса, как `Vector`, возможно, сделал все возможное для обеспечения неизменности класса `Vector`, но производный класс при полном доступе к переменным экземпляра вполне способен изменять их, нанося ущерб любому клиенту, считающему класс неизменным.

Применение: поиск данных. Для иллюстрации некоторых из обсуждаемых в этом разделе концепций (в контексте применения) рассмотрим программную технологию, весьма важную при решении проблемы *поиска данных* (data mining), т.е. больших объемов информации. Эта технология может лечь в основу существенных усовершенствований поиска в сети, поиска мультимедийной информации, поиска в биомедицинских базах данных, обнаружения плагиата, исследования генома, учреждения стипендии во многих областях, в новейших коммерческих приложениях и во многих других областях. Таким образом, эта технология представляет интерес и имеет перспективу.

У вас есть прямой доступ к тысячам файлов на вашем компьютере и косвенный доступ к миллиардам файлов в веб. Как известно, эти файлы удивительно разнообразны: есть коммерческие веб-страницы, музыка, видео, электронная почта, программный код и всякого рода другая информация. Для простоты мы ограничимся рассмотрением *текстовых* документов (хотя изучаемый нами подход относится также и к файлам изображения, музыки

и других). Даже при этом ограничении доступно замечательное разнообразие типов документов. Для справки: описанные ниже документы можно найти на сайте книги.

Нас интересует эффективный способ поиска файлов по их *содержимому*, характеризующему документ. Один из перспективных подходов решения этой проблемы подразумевает ассоциацию с каждым документом вектора *эскиза* (sketch), являющегося ультракомпактным представлением его содержимого. Основная идея эскиза в том, что он должен охватить самые существенные статистические особенности документа так, чтобы эскизы разных документов отличались, а сходных были подобны. Вас, вероятно, не удивит, что этот подход позволяет отличить роман от программы Python или генома, но вам, наверное, будет интересно узнать, что он может указать на различие между романами, написанными разными авторами, и может быть эффективен как основа для многих других подробных критериев поиска.

Некоторые из текстовых документов

Имя файла	Описание	Пример текста
constitution.txt	Официальный документ	... of both Houses shall be determined by ...
tomsawyer.txt	Американский роман	..."Say, Tom, let ME whitewash a little." ...
huckfinn.txt	Американский роман	...was feeling pretty good after breakfast...
prejudice.txt	Английский роман	... dared not even mention that gentleman....
Picture.py	Код Python	...import sys import color import stdarray...
djia.csv	Финансовые данные	...01-Oct-28,239.43,242.46,3500000,240.01 ...
amazon.html	Исходный код веб-страницы	...<table width="100%" border="0" cellspac...
actg.txt	Геном вируса	...GTATGGAGCAGCAGACGCGCTACTTCGAGCGGAGGCATA...

Для начала нужна абстракция документа. Что такое документ? Какие операции мы хотим выполнить с документами? Ответы на эти вопросы определят наш проект, а следовательно, в конечном счете и составленный нами код. В нашем случае документ — это содержимое любого текстового файла. Как было указано, мы храним только эскиз документа (а не весь документ) и используем эскизы для измерения подобия ассоциированных с ними документов. Эти соображения ведут к API, приведенному ниже. Аргументы конструктора — это строка и два целых числа, контролирующих качество эскиза. Для определения степени сходства между двумя эскизами в диапазоне от 0 (разные) до 1 (совпадающие) клиенты могут использовать метод `similarTo()`. Этот простой тип данных

обеспечивает хорошее разделение реализации меры подобия и реализации клиентов, использующих меру для поиска среди документов.

Расчет эскизов. Первая задача — расчет эскиза документа. Наш первый выбор для представления эскиза документа — класс `Vector`. Но какая информация должна войти в эскиз и как ее вычислить? Для этой задачи уже выработано много разных подходов, и исследования по поиску эффективных алгоритмов все еще активно продолжаются. Наша реализация в программе 3.3.4 (`sketch.py`) использует простой подход *подсчета частот*. Кроме строки, конструктор получает еще два аргумента: целое число k и размерность вектора d. Он просматривает документ и исследует все его *k-граммы* (k-gram) — подстроки длиной k, начинающиеся в каждой позиции. В своей самой простой форме эскиз — это вектор, предоставляющий относительную частоту вхождений k-граммов в строке: по элементу для каждого возможного k-грамма, предоставляющего количество k-граммов с тем же значением в документе. Предположим, например, что для генных данных мы используем $k = 2$ при $d = 16$ (поскольку есть 4 возможных символьных значения, существует 16 возможных 2-грамма). В строке `ATAGATGCATAGCGCATAGC` 2-грамм `AT` встречается 4 раза, таким образом, соответствующим `AT` элементом вектора было бы 4. Для построения вектора частот необходимо преобразовать каждый из k-грамм в целое число от 0 до 15 (функция сопоставления строк с целыми числами — это *хеш-функция*). Для генных данных это простое упражнение (см. упр. 3.3.27). Затем мы можем вычислить массив для построения вектора частот за один проход текста, увеличивая значение элемента массива, соответствующего каждому встретившемуся k-грамму. Хотя мы, конечно, теряем информацию, игнорируя последовательность k-граммов, ее информационная ценность ниже, чем у частоты k-граммов. (Та же парадигма модели Маркова, что и для случайной навигации в разделе 1.6, такие модели эффективны, но сложны в реализации.) Инкапсулируем это вычисление в класс `Sketch`, определенный в программе 3.3.4 (`sketch.py`). Это даст нам гибкость и позволит экспериментировать с различными проектами без необходимости переписывать клиенты класса `Sketch`.

API для пользовательского типа данных `Sketch`

Операция	Описание
`Sketch(text, k, d)`	Новый эскиз, созданный из строки `text` с использованием `k`-граммов и размерности `d`
`a.similarTo(b)`	Мера подобия между эскизами `a` и `b` (`float` от `0.0` до `1.0`)
`Str(a)`	Строковое представление эскиза `a`

				CTTTCGGTTT GGAACCGAAG CCGCGCGTCT	
		ATAGATGCAT AGCGCATAGC		TGTCTGCTGC AGCATCGTTC	
2-грамм	**Хеш**	**Счет**	**Единица**	**Счет**	**Единица**
AA	0	0	0	2	.137
AC	1	0	0	1	.069
AG	2	4	.508	1	.069
AT	3	3	.381	2	.137
CA	4	3	.381	3	.206
CC	5	0	0	2	.137
CG	6	0	0	4	.275
CT	7	1	.127	6	.412
GA	8	3	.381	0	0
GC	9	0	0	5	.343
GG	10	0	0	6	.412
GT	11	1	.127	4	.275
TA	12	1	.127	2	.137
TC	13	4	.508	6	.412
TG	14	0	0	4	.275
TT	15	0	0	2	.137

Профилирование генных данных

Предупреждение пользователям Python 2. Все хеш-функции начинают свое вычисление с "начального числа". В языке Python 2 начальное число хеша постоянно, поэтому значение хеш-функции любого данного объекта то же при каждом запуске программы. Начальное число хеш в Python 3, напротив, изменяется (стандартно). Поэтому в Python 3 значение хеш-функции любого данного объекта, даже в пределах любого данного запуска программы, весьма вероятно будет отличаться. Таким образом, в Python 3 программа `sketch.py` создаст разный вывод при каждом запуске.

Хеширование. Во многих системах для каждого символа есть 128 разных возможных значений, таким образом, возможно 128^k k-грамм, а размерность d должна составить 128^k для простой, только что описанной схемы. Это число предельно велико даже для умеренно большого k. Для кодировки Unicode с более чем 65 536 символами даже 2-грамм приводит к огромным векторным эскизам. Для решения этой проблемы мы используем *хеширование* (hashing) — фундаментальную операцию (рассматривавшуюся ранее в этом разделе) преобразования

объекта в целое число. Для любой строки `s` `hash(s) % d` является целым числом от `0` до `d - 1`, которое мы можем использовать как индекс массива при вычислении частоты. Используемый нами эскиз является направлением вектора, определенного частотами этих значений для всех *k*-граммов в документе (единичный вектор с тем же направлением).

Программа 3.3.4. Эскиз документа (sketch.py)

```
import sys
import stdarray
import stdio
from vector import Vector

class Sketch:
    def __init__(self, text, k, d):
        freq = stdarray.create1D(d, 0)
        for i in range(len(text) - k):
            kgram = text[i:i+k]
            freq[hash(kgram) % d] += 1
        vector = Vector(freq)
        self._sketch = vector.direction()

    def similarTo(self, other):
        return self._sketch.dot(other._sketch)

    def __str__(self):
        return str(self._sketch)

def main():
    text = stdio.readAll()
    k = int(sys.argv[1])
    d = int(sys.argv[2])
    sketch = Sketch(text, k, d)
    stdio.writeln(sketch)

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

	Переменные экземпляра
`_sketch`	Единичный вектор
`text`	Строка документа
`k`	Длина грамма
`d`	Размерность
`kgram`	k последовательных символов в документе
`freq[]`	Хеш частот
`vector`	Вектор частот

Этот клиент типа `Vector` создает единичный вектор из `k`-граммов документа, который можно использовать для измерения его подобия другим документам (см. текст).

```
% more genome20.txt
ATAGATGCATAGCGCATAGC
% python sketch.py 2 16 < genome20.txt
[ 0.0, 0.0, 0.0, 0.0, 0.504, 0.504, ..., 0.126, 0.0, 0.0, 0.378 ]
```

Сравнение эскизов. Вторая задача — вычисление меры подобия двух эскизов. Для сравнения двух векторов также есть много разных способов. Возможно, проще всего вычислить *Евклидово расстояние* между ними. Если даны векторы **x** и **y**, то это расстояние определяется следующим образом:

$$|\mathbf{x} - \mathbf{y}| = ((x_0 - y_0)^2 + (x_1 - y_1)^2 + ... + (x_{d-1} - y_{d-1})^2)^{1/2}.$$

Вам знакома эта формула для d = 2 или d = 3. Имея класс `Vector`, вычислить расстояние просто. Если `x` и `y` — это два объекта типа `Vector`, то Евклидово расстояние между ними — `abs(x - y)`. Если документы подобны, мы ожидаем, что и их эскизы будут подобны, а расстояние между ними будет коротким. Другая широко использованная мера подобия — это *косинусный коэффициент Отиаи* (cosine similarity), он даже проще, поскольку наши эскизы — это единичные векторы с неотрицательными координатами, их произведение является числом от 0 до 1:

$$\mathbf{x} \cdot \mathbf{y} = x_0 y_0 + x_1 y_1 + ... + x_{d-1} y_{d-1}$$

Геометрически это значение косинуса угла, образованного двумя векторами (см. упр. 3.3.9). Чем более подобны документы, тем ближе эта мера будет к 1. Если `x` и `y` — это два объекта типа `Vector`, то их векторное произведение — `x.dot(y)`.

Сравнение всех пар. Программа 3.3.5 (`comparedocuments.py`) является простым и полезным клиентом типа `Sketch`, предоставляющим информацию, необходимую для решения следующей задачи: дан ряд документов, найти два наиболее похожих. Поскольку эта спецификация немного субъективна, программа `comparedocuments.py` выводит меру по косинусному коэффициенту для всех пар документов. Для умеренных величин k и d эскизы дают на удивление хорошие результаты по характеристике нашего небольшого набора документов. Результаты доказывают, что не только генные, финансовые данные, код Python и исходный код веб-страниц весьма отличаются от официальных документов и романов, но и то, что *Том Сойер* и *Гекльберри Финн* намного более подобны друг другу, чем *Гордость и предубеждение*. Исследователь в области сравнительного литературоведения мог бы использовать эту программу для обнаружения отношений между текстами; преподаватель — для обнаружения плагиата в студенческих работах, а биолог — для обнаружения отношений среди геномов.

Поиск подобных документов. Следующий вполне логичный клиент класса `Sketch` использует эскизы для поиска среди множества документов, подобных данному. Например, сетевые поисковые системы используют клиенты этого типа для предоставления вам страниц, подобных тем, которые вы посетили ранее, сетевые книжные магазины используют клиенты этого типа для рекомендации книг, подобных купленным вами ранее, а веб-сайты социальных сетей — для идентификации людей, личные интересы которых подобны вашим. Поскольку конструктор класса `Instream` может получать в качестве аргумента сетевой адрес, вместо имени файла, вполне можно составить сетевую программу, вычисляющую эскизы и возвращающую ссылки на страницы с эскизами, подобными искомым. Мы оставляем этот клиент для самостоятельного упражнения.

Программа 3.3.5. Обнаружение подобия (comparedocuments.py)

```
import sys
import stdarray
import stdio
from instream import InStream
from sketch import Sketch

k = int(sys.argv[1])
d = int(sys.argv[2])
filenames = stdio.readAllStrings()

sketches = stdarray.create1D(len(filenames))
for i in range(len(filenames)):
    text = InStream(filenames[i]).readAll()
    sketches[i] = Sketch(text, k, d)

stdio.write('  ')
for i in range(len(filenames)):
    stdio.writef('%8.4s', filenames[i])
stdio.writeln()

for i in range(len(filenames)):
    stdio.writef('%.4s', filenames[i])
    for j in range(len(filenames)):
        stdio.writef('%8.2f ', sketches[i].similarTo(sketches[j]))
    stdio.writeln()
```

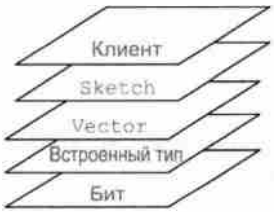
	Переменные экземпляра
k	Длина грамма
d	Размерность
filenames[]	Имена документов
sketches[]	Все эскизы

Этот клиент типа Sketch читает со стандартного устройства ввода список документов, вычисляет их эскизы на основании частоты k-грамм и выводит таблицу мер подобия между всеми парами документов. В аргументах командной строки она получает значения k и размерность d.

```
% more documents.txt
constitution.txt
tomsawyer.txt
huckfinn.txt
prejudice.txt
picture.py
djia.csv
amazon.html
actg.txt
```

```
% python comparedocuments.py 5 10000 < documents.txt
      cons  toms  huck  prej  pict  djia  amaz  actg
cons  1.00  0.69  0.63  0.67  0.06  0.15  0.19  0.12
toms  0.69  1.00  0.93  0.89  0.05  0.18  0.19  0.14
huck  0.63  0.93  1.00  0.83  0.03  0.16  0.16  0.13
prej  0.67  0.89  0.83  1.00  0.04  0.20  0.20  0.14
pict  0.06  0.05  0.03  0.04  1.00  0.01  0.13  0.01
djia  0.15  0.18  0.16  0.20  0.01  1.00  0.11  0.07
amaz  0.19  0.19  0.16  0.20  0.13  0.11  1.00  0.06
actg  0.12  0.14  0.13  0.14  0.01  0.07  0.06  1.00
```

Это решение — лишь краткий обзор. Для эффективного расчета эскизов и их сравнения уже создано множество сложных алгоритмов, и многие



Уровни абстракции

разрабатываются. Наша задача здесь заключается в том, чтобы ознакомить вас с основами в этой области и проиллюстрировать мощь абстракции при решении столь сложных вычислительных задач. Векторы — это фундаментальная математическая абстракция, и мы можем построить решение для поиска, разработав *уровни абстракции* (layers of abstraction): `Vector` создается с массивом, `Sketch` — с объектами `Vector`, а клиентский код использует объекты `Sketch`. Как обычно, мы избавим вас от длинных рассуждений и множества наших попыток по выработке этих API, однако следует заметить, что типы данных разрабатываются в ответ на поставленную задачу, причем с учетом требований реализации. Выявление и реализация соответствующих абстракций является ключом к эффективному объектно-ориентированному программированию. Мощь абстракции — в математике, физических моделях и компьютерных программах, это и демонстрируют все наши примеры. По мере приобретения опыта в разработке типов данных для решения собственных вычислительных задач ваша оценка этой мощи будет, конечно, увеличиваться.

Проектирование по контракту. В заключение кратко обсудим механизмы языка Python, позволяющие вам проверять предположения о вашей программе *во время ее выполнения*. Например, если у вас есть тип данных, представляющий частицу, то вы могли бы утверждать, что ее масса положительна, а скорость меньше скорости света. Или, если у вас есть метод сложения двух векторов той же размерности, вы могли бы утверждать, что у результирующего вектора будет такая же размерность.

Исключение (exception) — это деструктивное событие, происходящее во время выполнения программы и зачастую сообщающее об ошибке. Предпринимаемое действие известно как *передача исключения* (raising an exception), или сообщение об ошибке. С некоторыми из исключений вы уже встречались, изучая стандартные модули Python, типичные примеры — `IndexError` и `ZeroDivisionError`. Вы можете также передавать исключения собственных типов. Самый простой тип, `Exception`, прерывает выполнение программы и выводит сообщение об ошибке.

```
raise Exception('Error message here.')
```

Использование исключений, когда они могут быть полезными для клиента, считается хорошей практикой. Например, в методе `__add__()` класса `Vector` (программа 3.3.3) мы должны передать исключение, если у двух суммируемых векторов разные размерности. Для этого в начало метода `__add__()` следует вставить следующий оператор:

```
if len(self) != len(other):
    raise Exception('vectors have different dimensions')
```

Это даст более информативное сообщение об ошибке, чем `IndexError`, которое в противном случае получил бы клиент при несовпадении размерностей `self` и `other`.

Утверждение (assertion) — это логическое выражение, истинность (`True`) которого подтверждается в данной точке программы. Если выражение будет ложно (`False`), то во время выполнения программа передаст сообщение об ошибке `AssertionError`. Программисты используют утверждения для обнаружения ошибок и для того, чтобы завоевать доверие к правильности их программ. Утверждения служат также для документирования намерений программиста. Например, в классе `Counter` (программа 3.3.2) мы могли бы проверить, что счетчик никогда не будет иметь отрицательных значений, добавив в метод `increment()` как последний оператор следующее утверждение:

```
assert self._count >= 0
```

Этот оператор привлек бы внимание к отрицательному значению счетчика. Можно также добавить необязательное сообщение, чтобы помочь идентифицировать ошибку:

```
assert self._count >= 0, 'Negative count detected!'
```

Стандартно утверждения разрешены, но вы можете отключить их в командной строке, использовав флаг `-O` (знак "минус" и прописная буква "O") в команде `python`. (`O` — сокращение от Optimize (оптимизировать).) Утверждения используются только при отладке; при нормальной работе ваша программа не должна полагаться на утверждения, поскольку они могут быть отключены.

Когда вы будете проходить курс системного программирования, вы научитесь использовать утверждения для гарантии того, что ваш код *никогда* не завершится системной ошибкой или входом в бесконечный цикл. Одна из программных моделей, выражающих эту идею, — *проектирование по контракту* (design-by-contract). Разработчик типа данных выражает *предусловие* (precondition), т.е. условие, которое клиент обещает выполнять при вызове метода, *постусловие* (postcondition), т.е. условие, которого реализация обещает достичь при выходе из метода, *инварианты* (invariant), т.е. все условия, которые реализация обещает соблюдать во время выполнения метода, и *побочные эффекты* (side effect) — все остальные изменения в состоянии, к которым могло бы привести выполнение метода. Утверждения позволяют проверять все эти условия во время разработки. Многие программисты весьма интенсивно используют утверждения для облегчения отладки.

Обсуждаемые в этом разделе механизмы языка иллюстрируют тот факт, что эффективный проект типа данных выводит нас в фарватер проектирования языка программирования. Эксперты все еще обсуждают наилучшие способы реализации некоторых из рассматриваемых здесь концепций проектирования.

Почему Python не поддерживает управление доступом и закрытые переменные экземпляра? Почему в Java функции не являются объектами первого класса? Почему Matlab передает в функции копии массивов и матриц вместо ссылок? Как упоминалось в главе 1, это скользкий путь от жалоб на возможности языка программирования к квалификации разработчика языка программирования. Если вы не планируете сделать это, то лучше всего использовать широко доступные языки. У большинства систем есть обширные библиотеки, которыми при необходимости нужно пользоваться, но зачастую вы можете упростить свой клиентский код, создавая абстракции, которые могут быть легко переведены на другие языки. Ваша основная задача — разрабатывать типы данных так, чтобы бо́льшая часть работы осуществлялась на уровне абстракции, подходящем для решения текущей задачи.

Вопросы и ответы

Почему соглашение о символе подчеркивания не является частью языка Python (хотя и соблюдается)?

Хороший вопрос.

Почему начальные символы подчеркивания?

Это только один из многих примеров, когда разработчик языка программирования руководствуется личным предпочтением, а мы получаем результат. К счастью, большинство составляемых нами программ Python будет клиентскими программами, которые не вызывают специальные методы непосредственно и не обращаются к закрытым переменным экземпляра, а следовательно, они не будут нуждаться в начальных символах подчеркивания. Собственные типы данных реализует относительно немного программистов Python (теперь это и вы), они должны следовать соглашению о символах подчеркивания, но даже они, вероятно, будут составлять больше клиентского кода, чем реализаций классов, поэтому символы подчеркивания не будут столь тягостны в конечном счете.

Метод `__mul__()` в программе 3.3.1 (`complexpolar.py`) неуклюж, поскольку он создает объект `Complex` (представляющий 0 + 0*i*), а затем немедленно изменяет его переменные экземпляра на желаемые полярные координаты. Разве проект не был бы лучше, если бы я мог добавить второй конструктор, получающий как аргументы полярные координаты?

Да, но у нас уже есть конструктор, получающий как аргументы Декартовы координаты. Лучший проект подразумевал бы наличие в API двух обычных функций (не методов) `createRect(x, y)` и `createPolar(r, theta)`, создающих и возвращающих новые объекты. Такой проект, возможно, лучше, поскольку он позволил бы *клиенту* использовать и полярные координаты. Этот пример демонстрирует, что при разработке типа данных имеет смысл подумать и о больше чем одной реализации. Конечно, внесение такого изменения потребует модификации всех существующих реализаций и клиентов API, поэтому эти резоны стоит по возможности учесть уже на раннем этапе процесса проектирования.

Как определить кортеж, состоящий из нуля или одного элемента?

Можно использовать синтаксис `()` и `(1,)` соответственно. Без запятой во втором выражении Python примет это за арифметическое выражение, заключенное в круглые скобки.

Нужно ли перегрузить все шесть методов сравнения, если я хочу сделать мой тип данных сравнимым?

Да. Это пример того, когда соглашение обеспечивает максимальную гибкость клиентам за счет дополнительного кода в реализациях. Зачастую для сокращения фактического объема кода реализации вы можете использовать симметрию. Кроме того, Python 3 имеет несколько сокращений. Например, если определить для типа данных метод `__eq__()`, то метод `__ne__()` можно не определять — Python сам предоставит реализацию, вызывающую метод `__eq__()` и инвертирующую результат. Однако в Python 2 этих сокращений нет, поэтому лучше не полагаться на них в коде.

Возможна ли ситуация, когда нежелательно, чтобы шесть методов сравнения реализовали полный порядок?

Да, вы могли бы реализовать *частичный порядок*, где можно сравнить не каждую пару объектов. Например, вы могли бы сравнивать людей согласно генеалогическому порядку, где один человек считается меньше другого, если он (или она) — потомок (ребенок, внук, правнук и т.д.) другого. Но не все люди состоят в родстве. Кроме того, вам могло бы понадобиться сравнивать объекты `Counter` по их счету (при сортировке), но считать при этом равными только объекты с совпадающим счетом и именем.

Каков диапазон целочисленных значений, возвращаемых встроенной функцией `hash()`?

Как правило, Python использует 64-битовое целое число, соответственно диапазон от -2^{63} до $2^{63} - 1$. Для криптографических приложений следует использовать модуль Python `hashlib`, предоставляющий "защищенные" хеш-функции, обеспечивающие намного бо́льшие диапазоны.

Какие операторы Python не могут быть перегружены?

В Python нельзя перегружать:

- Логические операторы `and`, `or` и `not`.
- Операторы `is` и `is not`, проверяющие идентификаторы объектов.
- Оператор формата строк `%`, применимый только к строкам.
- Оператор присвоения `=`.

Упражнения

3.3.1. Создайте тип данных `Location` для работы с локациями на поверхности Земли, используя сферические координаты (широта / долгота). Включите методы создания случайной точки на поверхности Земли, скажем, "25.344 N, 63.5532 W", и вычисления длины по большому кругу между двумя локациями.

3.3.2. Создайте тип данных для трехмерной частицы с позицией (r_x, r_y, r_z), массой (m) и скоростью (v_x, v_y, v_z). Включите метод для возвращения ее кинетической энергии, равной $1/2\ m\ (v_x^2 + v_y^2 + v_z^2)$. Используйте тип данных `Vector`.

3.3.3. Если вы знаете физику, разработайте альтернативную реализацию типа данных из предыдущего упражнения на основании *импульса* (p_x, p_y, p_z) в качестве переменной экземпляра.

3.3.4. Разработайте реализацию класса `Histogram` (программа 3.2.3), использующего тип данных `Counter` (программа 3.3.2).

3.3.5. Разработайте клиент проверки для класса `Vector`.

3.3.6. Составьте реализацию метода `__sub__()` класса `Vector`, вычитающий два вектора.

3.3.7. Реализуйте тип данных `Vector2D` для двумерных векторов, обладающий тем же API, что и `Vector`, за исключением того, что конструктор получает в аргументах два значения типа `float`. Используйте для переменных экземпляра два значения типа `float` (вместо массива).

3.3.8. Реализуйте тип данных `Vector2D` из предыдущего упражнения, используя в качестве единственной переменной экземпляра один объект типа `Complex`.

3.3.9. Докажите, что произведение двух двумерных единичных векторов — это косинус угла между ними.

3.3.10. Реализуйте тип данных `Vector3D` для трехмерных векторов, обладающий тем же API, что и `Vector`, за исключением того, что конструктор получает в аргументах три значения типа `float`. Кроме того, добавьте метод *перекрестного произведения* двух векторов, определенный уравнением $\mathbf{a} \times \mathbf{b} = \mathbf{c}\ |\mathbf{a}|\ |\mathbf{b}|\ \sin\theta$, где $\mathbf{c}$ — единичный перпендикуляр к нормали векторов $\mathbf{a}$ и $\mathbf{b}$; θ — угол между ними. В Декартовых координатах перекрестное произведение определяет следующее уравнение:

$$(a_0, a_1, a_2) \times (b_0, b_1, b_2) = (a_1 b_2 - a_2 b_1, a_2 b_0 - a_0 b_2, a_0 b_1 - a_1 b_0).$$

Перекрестное произведение — это псевдовектор, перпендикулярный плоскости, построенной по двум сомножителям, являющийся результатом бинарной операции "векторное умножение" над векторами в трехмерном Евклидовом пространстве. Кроме того, $|\mathbf{a} \times \mathbf{b}|$ — это площадь параллелограмма со сторонами **a** и **b**.

3.3.11. Какие модификации необходимы (если необходимы), чтобы выполнить работу класса `Vector` с компонентами типа `Complex` (см. программу 3.2.6) или компонентами типа `Rational` (см. упр. 3.2.7)?

3.3.12. Добавьте в программу `charge.py` код, делающий объекты `Charge` сравнимыми по значению заряда, чтобы определить полный порядок.

3.3.13. Составьте функцию `fibonacci()`, получающую целочисленный аргумент `n` и вычисляющий `n`-е число Фибоначчи. Используйте упаковку и распаковку кортежа.

3.3.14. Пересмотрите функцию `gcd()` в программе 2.3.1 так, чтобы она получала два неотрицательных целочисленных аргумента `p` и `q`, а возвращала кортеж целых чисел `(d, a, b)`, где `d` — это наибольший общий делитель `p` и `q`, а соотношение `a` и `b` соответствует соотношению Безу: `d = a*p + b*q`. Используйте упаковку и распаковку кортежа.

Решение — расширенный алгоритм Евклида:

```
def gcd(p, q):
    if q == 0: return (p, 1, 0)
    (d, a, b) = gcd(q, p % q)
    return (d, b, a - (p // q) * b)
```

3.3.15. Обсудите преимущества и недостатки проекта Python по созданию встроенного типа `bool` как производного от встроенного типа `int`.

3.3.16. Добавьте к классу `Counter` код передачи во время выполнения исключения `ValueError`, если клиент пытается создать объект `Counter`, используя отрицательное значение для `maxCount`.

3.3.17. Используя исключения, разработайте реализацию класса `Rational` (см. упр. 3.2.7) так, чтобы во время выполнения передавалось исключение `ValueException`, если знаменатель имеет нулевое значение.

Упражнения по разработке типов данных

Эти упражнения помогут приобрести практический опыт в разработке типов данных. Для каждой задачи разработайте один или несколько API, реализуйте их, проверьте свои проектные решения при реализации

на типичном клиентском коде. Некоторые из упражнений требуют знаний в специфических областях или возможности найти эти знания в веб.

3.3.18. *Статистика.* Разработайте тип данных для хранения статистических данных в виде набора значений типа `float`. Снабдите его методом добавления новых пунктов данных, а также методами возвращения количества пунктов, их среднего, среднеквадратичного отклонения и дисперсии. Разработайте две реализации: у одной из которых значения экземпляра — это количество пунктов, сумма значений и сумма квадратов значений, а другая хранит массив, содержащий все пункты. Для простоты в конструкторе можно получать максимальное количество пунктов. Первая реализация, вероятно, окажется быстрее и потребует существенно меньше пространства, но, вероятно, будет также восприимчива к ошибке округления. Хорошо спроектированные альтернативы приведены на сайте книги.

3.3.19. *Геном.* Разработайте тип данных для хранения генома организма. Биологи зачастую абстрагируют геном до последовательностей оснований (`A`, `C`, `G` и `T`). Тип данных должен поддерживать такие методы, как `addCodon(c)` и `baseAt(i)`, а также `isPotentialGene()` (см. программу 3.1.1). Разработайте три реализации.

- Используйте строку как единственную переменную экземпляра; реализуйте метод `addCodon()` с использованием конкатенации строк.
- Используйте массив односимвольных строк как единственную переменную экземпляра; реализуйте метод `addCodon()` с использованием оператора `+=`.
- Используйте массив логических значений, где каждое основание закодировано двумя битами.

3.3.20. *Время.* Разработайте тип данных для времени дня. Предоставьте клиентские методы, один из которых возвращает текущий час, минуту и секунду, а второй подобен методу `__str__()`. Разработайте две реализации: одна из которых хранит время как единое значение типа `int` (количество секунд с полуночи), а вторая — три значения типа `int` (по одному для часов, минут и секунд).

3.3.21. *Векторное поле.* Разработайте тип данных для векторов силы в двух размерностях. Предоставьте конструктор, метод добавления двух векторов и клиент проверки.

3.3.22. *Даты.* Разработайте API для дат (год, месяц, день). Включите методы для сравнения двух дат (хронологически), вычисления количества дней между двумя датами, определения дня недели заданной даты и любые другие операции, в которых, по-вашему, мог бы нуждаться клиент. Разработав собственный API, посмотрите тип данных Python `datetime.date`.

3.3.23. *Полиномы.* Разработайте тип данных для одномерных полиномов с целочисленными коэффициентами, такими как $x^3 + 5x^2 + 3x + 7$. Включите методы для стандартных операций с полиномами, такие как сложение, вычитание, умножение, порядок, вычисление, композиция, дифференцирование, определенная интеграция и проверка равенства.

3.3.24. *Рациональные полиномы.* Повторите предыдущее упражнение, но гарантируя, что полиноминальный тип данных будет вести себя правильно при предоставлении коэффициентов типа `int`, `float`, `complex` и `Fraction` (см. упр. 3.2.7).

Практические упражнения

3.3.25. *Календарь.* Разработайте API для типов `Appointment` и `Calendar`, применяемых для отслеживания встреч (по дням) в календаре на год. Ваша задача заключается в том, чтобы позволить клиентам планировать встречи, которые не вступят в противоречие, и сообщать о текущих встречах клиентам. Используйте модуль Python `datetime`.

3.3.26. *Векторное поле.* Векторное поле ассоциирует вектор с каждой точкой Евклидова пространства. Составьте версию программы 3.1.8 (`potential.py`), которая получает размер таблицы n, вычисляет значение `Vector` потенциала, созданного точечным зарядом для каждой равноудаленной точки в таблице n на n, и рисует в каждой точке поля единичный вектор в суммарном направлении.

3.3.27. *Создание эскизов генома.* Составьте функцию `hash()`, которая получает как аргумент k-грамм (строка длиной k) из символов только `A`, `C`, `G` или `T` и возвращает значение типа `int` от 0 до 4^k, соответствующее результату обработки строки как числа по основанию 4 при {`A`, `C`, `G`, `T`}, замененных на {`0`, `1`, `2`, `3`}, как предложено в таблице ранее. Затем составьте функцию `unhash()`, осуществляющую обратное преобразование. Используйте свои методы при создании класса `Genome`, производного от класса `Sketch`, но основанного на точном подсчете k-граммов в геномах. И наконец, составьте версию программы `comparealldocuments.py` для объектов класса

`Genome`, а затем используйте ее для поиска сходства в наборе файлов генома.

3.3.28. *Создание эскизов.* Загрузите набор документов с сайта книги (или используйте собственную коллекцию) и запустите программу `comparealldocuments.py` с различными параметрами командной строки, чтобы узнать об их влиянии на вычисление.

3.3.29. *Мультимедийный поиск.* Разработайте стратегии создания эскизов для звука и изображений, а затем используйте их для обнаружения подобия в музыкальных файлах библиотеки и фотографиях в альбоме на вашем компьютере.

3.3.30. *Поиск данных.* Составьте рекурсивную программу поиска в веб, которая начинается со страницы, заданной первым аргументом командной строки, и отыскивает страницы, подобные заданной вторым аргументом командной строки. Для обработки имени откройте поток ввода, выполните `readAll()`, создайте его эскиз и выведите имя, если его дистанция к выходной странице больше порогового значения, заданного третьим аргументом командной строки. Затем просмотрите страницы для всех строк, содержащих подстроку `http://`, и (рекурсивно) обработайте страницы с этими именами. *Примечание*: эта программа способна читать очень большое количество страниц!

3.4. Случай из практики: моделирование N тел

Программы этого раздела...

Программа 3.4.1. Гравитационное тело (body.py) 490

Программа 3.4.2. Моделирование N тел (universe.py) 493

Некоторые из примеров, рассматривавшихся в главах 1 и 2, можно выразить яснее как объектно-ориентированные программы. Например, мы можем реализовать программу 1.5.7 (bouncingball.py) как тип данных со значениями позиции и скорости шара и как клиент этого типа, вызывающий методы для перемещения и возвращения позиции шара. Такой тип данных и клиент позволят, например, моделировать движение нескольких шаров сразу (см. упр. 3.4.1). Точно так же наш анализ задачи просачивания в разделе 2.4 является хорошим кандидатом на упражнение по объектно-ориентированному программированию, равно как и анализ задачи случайной навигации в разделе 1.6. Но мы по-прежнему оставляем их для упражнения (3.4.11) и повторного рассмотрения в разделе 4.5. В этом разделе мы рассмотрим новую программу, иллюстрирующую объектно-ориентированное программирование.

Наша задача — составить программу, динамически моделирующую движение N тел под воздействием взаимного гравитационного притяжения. Задача *моделирования N тел* была впервые сформулирована Исааком Ньютоном более 350 лет назад, и ученые все еще изучают ее до сих пор.

Каков набор значений и каковы операции с этими значениями? Эта задача является ярким примером объектно-ориентированного программирования, поскольку она демонстрирует прямое и естественное соответствие между физическими объектами в реальном мире и абстрактными объектами, используемыми в программировании. Переход от решения задачи при помощи набора последовательных операторов к проектированию типов данных бывает трудным для многих новичков. По мере приобретения практического опыта вы ощутите все преимущества применения этого подхода к решению задач.

Сначала повторим некоторые из фундаментальных физических концепций и уравнений, изучаемых в школе. Для понимания кода не обязательно полностью помнить эти уравнения — из-за *инкапсуляции* этими уравнениями ограничивается лишь несколько методов, а из-за *абстракции данных* большая часть кода интуитивно понятна. В некотором смысле это исключительно объектно-ориентированная программа.

Моделирование N тел. Модель подпрыгивающего мяча (см. раздел 1.5) была создана на основании *первого закона Ньютона*: тело остается в движении с той же скоростью, если не подвергается действию внешней силы. Усовершенствование данного примера — в применении *второго закона Ньютона*, объясняющего влияние внешних сил на скорость. Это подводит нас к фундаментальной задаче,

занявшей умы ученых на столетия. Дана система из N тел, взаимодействующих под воздействием гравитационных сил, задача заключается в описании их движения. Эта базовая модель распространяется на множество задач: от астрофизики до молекулярной динамики.

В 1687 году Ньютон сформулировал принципы движения двух тел под воздействием их взаимного гравитационного притяжения. Однако Ньютону не удалось разработать математическое описание движения *трех* тел. С тех пор высказывались предположения, что не только невозможно их описание в терминах элементарных функций, но и что возможно хаотическое поведение в зависимости от исходных значений. Для изучения таких проблем у ученых нет пока иного средства, кроме разработки точных моделей. В этом разделе мы разработаем объектно-ориентированную программу, реализующую такую модель. Ученых интересует изучение таких задач для огромных количеств тел, поэтому наше решение — это только введение в тему, но вы, вероятно, будете удивлены легкостью, с которой мы сможем получать реалистичные изображения столь сложных движений.

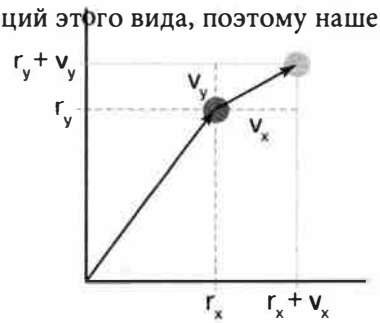
Тип данных тела. В программе 1.5.7 (`bouncingball.py`) мы хранили смещение от исходной точки в двух переменных `rx` и `ry` типа `float`, скорость — в переменных `vx` и `vy`, а перемещение шара вычисляли как сумму перемещений за единицу времени при помощи следующих операторов:

```
rx = rx + vx
ry = ry + vy
```

Используя тип `Vector` (программа 3.3.3), можно хранить позицию в векторе `r`, скорость в векторе `v` и перемещать тело за единицу времени `dt` одиночным оператором:

```
r = r + v.scale(dt)
```

При моделировании N тел есть несколько операций этого вида, поэтому наше первое проектное решение заключается в том, чтобы работать с объектами `Vector` вместо индивидуальных компонентов. Это решение приводит к более понятному, компактному и гибкому коду, чем альтернатива с индивидуальными компонентами. Программа 3.4.1 (`body.py`) реализует класс `Body` для типа данных Python, позволяющего перемещать тела. Класс `Body` — это клиент типа `Vector`, его значениями являются объекты типа `Vector`, содержащие позицию и скорость тела, а также переменную типа `float` для его массы. Операции типа данных позволяют клиентам перемещать и рисовать тело (а также вычислять вектор силы гравитационного притяжения другого тела), как

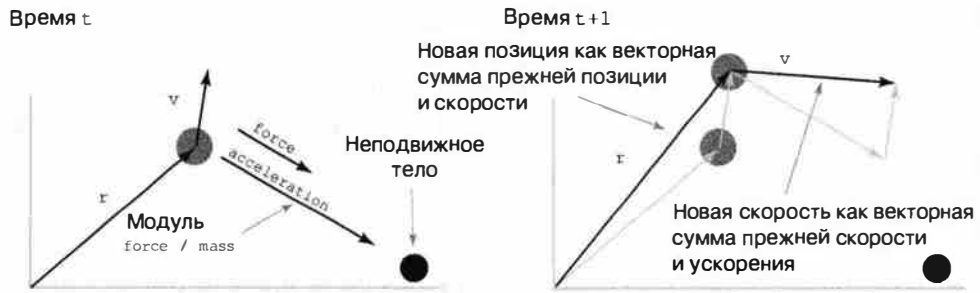


Сложение векторов при перемещении шара

определено в API далее. Технически позиция тела (смещение от начала координат) не является вектором (это точка в пространстве, а не направление и модуль), но нам удобней представить ее как тип `Vector`, поскольку операции с ним ведут к компактному коду, необходимому нам для перемещения тела. Когда мы перемещаем объект класса `Body`, необходимо изменить не только его позицию, но и скорость.

API для пользовательского типа данных `Body`

Операция	Описание
`Body(r, v, mass)`	Новое тело массой `mass` в позиции `r`, двигающееся со скоростью `v`
`b.move(f, dt)`	Переместить `b`, приложив силу `f` на протяжении `dt` секунд
`b.forceFrom(a)`	Вектор силы от тела `a` для тела `b`
`b.draw()`	Рисует тело `b` на стандартном графическом устройстве

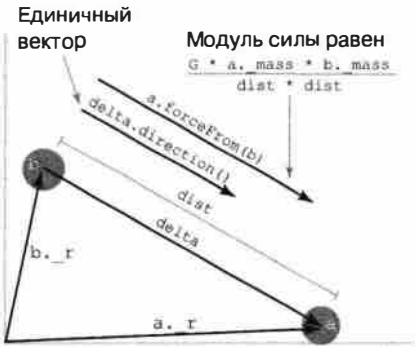


Движение вблизи неподвижного тела

Сила и движение. Второй закон Ньютона гласит, что приложенная к телу сила (вектор) равна скалярному произведению его массы на ускорение (также вектор): $\mathbf{F} = m\,\mathbf{a}$. Другими словами, чтобы вычислить ускорение тела, мы вычисляем силу, а затем делим ее на массу. В классе `Body` сила — это аргумент `f` типа `Vector` метода `move()`, поэтому для вычисления вектора ускорения достаточно деления на массу (скалярное значение, хранимое в переменной экземпляра типа `float`), а затем можно вычислить изменение в скорости, добавив значение изменения этого вектора за интервал времени (таким же образом мы использовали бы скорость для изменения позиции). Этот закон непосредственно преобразуется в следующий код модификации позиции и скорости тела благодаря приложенному вектору силы `f` за период времени `dt`:

```
a = f.scale(1.0 / mass)
v = v + a.scale(dt)
r = r + v.scale(dt)
```

Этот код располагается в методе `move()` класса `Body` для корректировки его значения под воздействием силы, приложенной на протяжении определенного времени: тело двигается, и его скорость изменяется. Это вычисление подразумевает, что на протяжении интервала времени ускорение остается постоянным.



Воздействие силы одного тела на другое

Силы среди тел. Вычисление силы, прилагаемой одним телом к другому, инкапсулируется в методе `forceFrom()` класса `Body`, получающий как аргумент объект `Body` и возвращающий объект `Vector`. В основе вычислений лежит *закон всемирного тяготения*: модуль силы гравитации между двумя телами равняется произведению их масс, деленных на квадрат расстояния между ними, и умноженный на гравитационную постоянную G, равную 6.67×10^{-11} $Нм^2кг^{-2}$, а направление силы — это линия между центрами тел. Этот закон преобразуется в следующий код вычисления `a.forceFrom(b)`:

```
G = 6.67e-11
delta = b._r - a._r
dist = abs(delta)
magnitude = G * a.mass * b.mass / (dist * dist)
f = delta.direction().scale(magnitude)
```

Модуль вектора силы — это переменная `magnitude` типа `float`, а *направление* вектора силы совпадает с направлением вектора разницы между позициями двух тел. Вектор силы `f` является произведением модуля и направления.

Тип данных области. Тип данных `Universe` (программа 3.4.2) реализует следующий API.

API для пользовательского типа данных `Universe`

Операция	Описание
`Universe(file)`	Создает новую область из описания в файле `file`
`u.increaseTime(dt)`	Обновляет область `u` в течение `dt` секунд
`u.draw()`	Рисует область `u` на стандартном графическом устройстве

Программа 3.4.1. Гравитационное тело (body.py)

```
import stddraw
from vector import Vector

class Body:
    def __init__(self, r, v, mass):
        self._r = r
        self._v = v
        self._mass = mass

    def move(self, f, dt):
        a = f.scale(1.0 / self._mass)
        self._v = self._v + a.scale(dt)
        self._r = self._r + self._v.scale(dt)

    def forceFrom(self, other):
        G = 6.67e-11
        delta = other._r - self._r
        dist = abs(delta)
        m1 = self._mass
        m2 = other._mass
        magnitude = G * m1 * m2 / (dist * dist)
        return delta.direction().scale(magnitude)

    def draw(self):
        stddraw.setPenRadius(0.0125)
        stddraw.point(self._r[0], self._r[1])
```

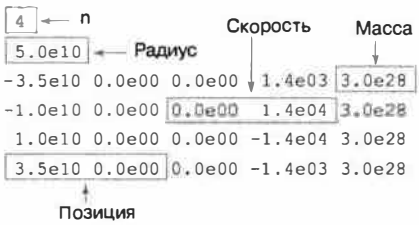
	Переменные экземпляра
_r	Позиция
_v	Скорость
_mass	Масса
f	Сила, приложенная к этому телу
dt	Приращение времени
a	Ускорение

Этот тип данных предоставляет операции, необходимые для моделирования движения физических тел, таких как планеты или элементарные частицы. Это изменяемый тип, его переменными экземпляра являются позиция и скорость тела, значения которых изменяют метод `move()` в ответ на внешние силы (масса тела неизменяема). Метод `forceFrom()` возвращает вектор силы. Модуль проверки мы отложим до упражнения 3.4.2.

	Переменные экземпляра
self	Вызывающее тело
other	Другое тело
G	Гравитационная постоянная
delta	Вектор между телами
dist	Расстояние между телами
magnitude	Модуль силы

Значение этого типа данных определяет область (ее размер, количество и массив тел). Тип данных имеет две операции: метод `increaseTime()` корректирует позиции (и скорости) всех тел, а метод `draw()` рисует все тела. Ключ к моделированию N тел — это реализация метода `increaseTime()` в классе `Universe`. Первая часть вычисления — двойной вложенный цикл, вычисляющий вектор силы, описывая силу гравитации, прикладываемую каждым телом к другому телу. Здесь применяется *принцип суперпозиции*, гласящий, что мы можем сложить воздействующие на тело векторы силы и получить единый вектор, представляющий результирующую силу. После вычисления всех сил для каждого тела вызывается метод `move()`, чтобы применить вычисленную силу за фиксированный отрезок времени.

```
% more 2body.txt
2
5.0e10
0.0e00 4.5e10 1.0e04 0.0e00 1.5e30
0.0e00 -4.5e10 -1.0e04 0.0e00 1.5e30

% more 3body.txt
3
1.25e11
0.0e00 0.0e00 0.05e04 0.0e00 5.97e24
0.0e00 4.5e10 3.0e04 0.0e00 1.989e30
0.0e00 -4.5e10 -3.0e04 0.0e00 1.989e30

% more 4body.txt
```



Примеры форматов файлов областей

Формат файла. Мы используем управляемый данными проект с вводом, получаемым из файла. Конструктор читает параметры области и описания тел из файла, содержащего следующую информацию:

- количество тел;
- радиус области;
- позиции, скорости и массы каждого тела.

Как обычно, все измерения даны в стандартных единицах СИ (помните, что гравитационная постоянная G также присутствует в нашем коде). При этом определении формата файла несложно составить код конструктора типа `Universe`:

```
def __init__(self, filename):
    instream = InStream(filename)
    n = instream.readInt()
    radius = instream.readFloat()

    stddraw.setXscale(-radius, +radius)
    stddraw.setYscale(-radius, +radius)
```

```
self._bodies = stdarray.create1D(n)
for i in range(n):
    rx   = instream.readFloat()
    ry   = instream.readFloat()
    vx   = instream.readFloat()
    vy   = instream.readFloat()
    mass = instream.readFloat()
    r = Vector([rx, ry])
    v = Vector([vx, vy])
    self._bodies[i] = Body(r, v, mass)
```

Каждый объект `Body` характеризуется пятью переменными типа `float`: координатами *x* и *y* его позиции, компонентами по *x* и *y* его начальной скорости и массой.

Таким образом, клиент проверки `main()` в классе `Universe` — это управляемая данными программа, моделирующая движение *n* тел, взаимно притягивающихся гравитацией. Конструктор создает массив из *n* объектов типа `Body`, читает начальную позицию, начальную скорость и массу каждого тела из файла, имя которого указано в аргументе командной строки. Метод `increaseTime()` вычисляет действующие на тела силы и использует эту информацию для изменения ускорения, скорости и позиции каждого тела за отрезок времени `dt`. Клиент проверки `main()` вызывает конструктор, а затем, оставаясь в цикле, вызывает методы `increaseTime()` и `draw()` для моделирования движения. На сайте книги можно найти множество файлов, определяющих "области" всех разновидностей, и вы можете, запустив программу `universe.py`, понаблюдать за их движением. Когда вы посмотрите движение даже небольшого количества тел, то поймете, почему Ньютону было затруднительно вывести уравнения, определяющие их траектории. На рисунках, приведенных ниже, показан результат выполнения программы `universe.py` для примеров на 2, 3 и 4 тела из описанных ранее файлов данных. Пример с 2 телами — взаимно орбитальная пара; пример с 3 телами — хаотическая ситуация с луной, движущейся между двумя орбитальными планетами; а в примере с 4 телами медленно вращаются две пары взаимно орбитальных тел. (В примере с 3 телами используется немного модифицированная версия класса `Body`, где радиус рисуемого тела пропорционален его массе.) Статические изображения на этих страницах были получены с использованием измененных классов `Universe` и `Body`, рисующих тела сначала белым, а затем черным на сером фоне, как в программе 1.5.7 (`bouncingball.py`). Сравните динамические изображения, получаемые при запуске программы `universe.py`, позволяющие получить реалистичное представление о вращении тел вокруг друг друга, с фиксированными изображениями, где различить движение затруднительно. Запустив программу `universe.py` на примере с большим количеством тел, вы можете оценить, почему моделирование столь важный инструмент для ученых, пытающихся понять сложную проблему. Модель N тел на удивление универсальна, как можно заметить, поэкспериментировав с некоторыми из исходных файлов.

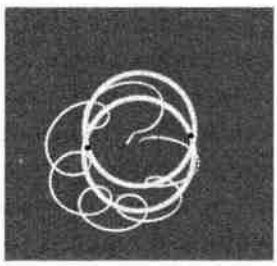
Программа 3.4.2. Моделирование N тел (universe.py)

```
import sys
import stdarray
import stddraw
from body import Body
from instream import InStream
from vector import Vector

class Universe:

    def __init__(self, filename):
    // См. текст.

    def increaseTime(self, dt):
        n = len(self._bodies)
        f = stdarray.create1D(n, Vector([0, 0]))
        for i in range(n):
            for j in range(n):
                if i != j:
                    bodyi = self._bodies[i]
                    bodyj = self._bodies[j]
                    f[i] = f[i] + bodyi.forceFrom(bodyj)
        for i in range(n):
            self._bodies[i].move(f[i], dt)

    def draw(self):
        for body in self._bodies:
            body.draw()

def main():
    universe = Universe(sys.argv[1])
    dt = float(sys.argv[2])
    while True:
        universe.increaseTime(dt)
        stddraw.clear()
        universe.draw()
        stddraw.show(10)

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

	Переменные экземпляра
_bodies	Массив тел
n	Количество тел
f[]	Сила, воздействующая на каждое тело

```
% python universe.py 3body.txt 20000
```

880 шагов



Эта управляемая данными программа моделирует движение в области, определенной по данным из файла, и с частотой dt, указанной в командной строке. Конструктор описан выше.

Вы, конечно, пожелаете разработать собственную область (см. упр. 3.4.10). Самая большая сложность создания файла данных заключается в выборе подходящих масштабов, чтобы радиус области, временны*е* рамки, массы и скорости тел привели к интересному поведению. Вы можете изучать движение планет, вращающихся вокруг Солнца, или элементарных частиц, взаимодействующих друг с другом, но вы не сможете изучить взаимодействие планеты с элементарной частицей. Когда вы будете работать с собственными данными, у вас, вероятно, некоторые тела улетят в бесконечность, другие столкнутся, однако пробуйте!

Планетарный масштаб

```
% more 2body.txt
2
5.0e10
0.0e00 4.5e10 1.0e04 0.0e00 1.5e30
0.0e00 -4.5e10 -1.0e04 0.0e00 1.5e30
```

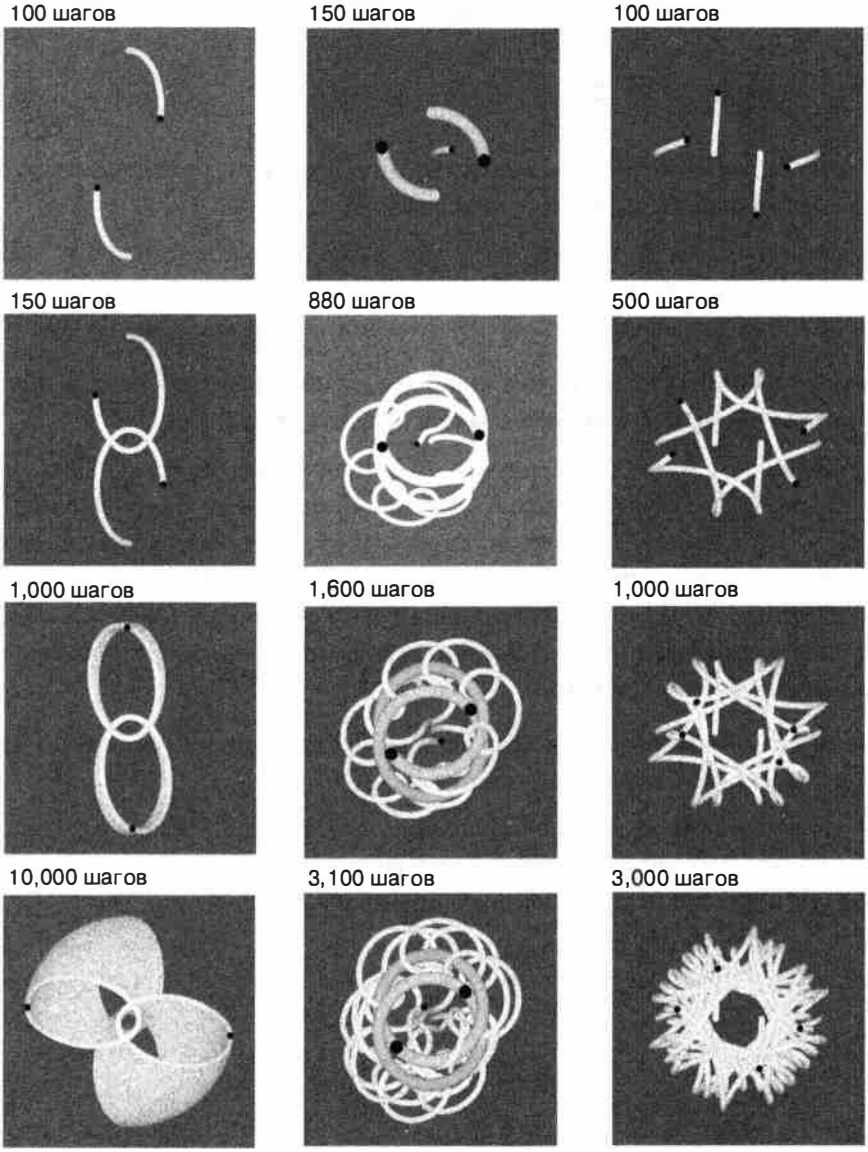
Масштаб элементарных частиц

```
% more 2bodyTiny.txt
2
5.0e-10
0.0e00 4.5e-10 1.0e-16 0.0e00 1.5e-30
0.0e00 -4.5e-10 -1.0e-16 0.0e00 1.5e-30
```

Два входных файла разного масштаба

Задача этого примера — продемонстрировать удобство типов данных, а не предоставить код модели для рабочего использования. При использовании этого подхода для изучения естественных явлений ученым приходится преодолевать множество проблем. Одна из них — *числовая точность*: для погрешностей в вычислениях характерно накапливаться и приводить к существенному воздействию на модель, чего не наблюдалось бы в природе. Для модификации позиции и скорости мы используем *пошаговый метод*, который дает более точные результаты, чем многие альтернативные подходы. Вторая проблема — *эффективность*: метод `move()` класса `Universe` требует времени, пропорционального n^2, а потому он не пригоден для большого количества тел. Кроме того, наша модель игнорирует другие силы. Например, наш код ничего не предпринимает при столкновении двух (или более) тел. Решение научных задач, связанных с проблемой n-тела, требует знаний не только в основной области.

Для простоты мы работаем с *двумерной* областью, реалистичной только при рассмотрении движения тел в плоскости. Однако важнейшее следствие создания реализации класса `Body` на базе класса `Vector` заключается в том, что клиент может использовать *трехмерные* векторы и моделировать движение шаров в трех размерностях (фактически в любом количестве размерностей), не изменяя существенно код (см. упр. 3.4.9).



Модели областей с 2, 3 и 4 телами

Клиент проверки в классе `Universe` — это только одна из возможностей. Мы можем использовать ту же базовую модель во всякого рода других ситуациях (например, задействовав различные виды взаимодействующих тел). Одна из таких возможностей подразумевает наблюдение и измерение текущего движения некоторых существующих тел и последующий запуск модели назад! Астрофизики используют такой метод для попыток поиска начальной точки вселенной. В науке мы пытаемся понять прошлое и предсказать будущее, а хорошая модель способна помочь в этом.

Вопросы и ответы

API класса `Universe` весьма невелик. Почему бы не реализовать этот код в клиенте проверки `main()` для класса `Body`?

Наш проект — это выражение того, что большинство людей знают о вселенной: она была создана, а затем пошло время. Не усложняя код, разъясняем и учитываем в модели максимум происходящего в области вселенной.

Почему `forceFrom()` сделан методом? Не лучше ли было сделать его функцией, получающей как аргументы два объекта типа `Body`?

Реализация `forceFrom()` как метода — это только одна из нескольких возможных альтернатив, включающих и функцию, получающую как аргументы два объекта `Body` (что, конечно, является разумным выбором). Однако некоторые программисты предпочитают полностью избегать функций в реализациях типа данных. Еще одна из возможностей — хранить действующую на каждое тело силу как переменную экземпляра. Наш выбор — это компромисс между двумя этими возможностями.

Не должен ли метод `move()` в программе `body.py` модифицировать позицию, используя *старую* скорость вместо *новой*?

Использование новой скорости (*пошаговый метод*) дает более точный результат, чем использование старой скорости (*метод Эйлера*). Если пройти курс математического анализа, то можно узнать, почему.

Упражнения

3.4.1. Разработайте объектно-ориентированную версию программы 1.5.7 (`bouncingball.py`). Включите конструктор, запускающий каждый шар в случайном направлении со случайной скоростью (в разумных пределах), и клиент проверки, получающий из командной строки целое число `n` и моделирующий движение `n` подпрыгивающих шаров.

3.4.2. Добавьте в программу 3.4.1 (`body.py`) метод `main()`, проверяющий тип данных `Body`.

3.4.3. Модифицируйте программу `body.py` так, чтобы радиус рисуемого тела был пропорционален его массе.

3.4.4. Что произошло бы в области без гравитационной силы? Это соответствовало бы ситуации, когда метод `forceFrom()` класса `Body` всегда возвращал бы нулевой вектор.

3.4.5. Создайте тип данных `Universe3D` для моделирования трехмерной области. Разработайте файл данных для моделирования движения планет в нашей солнечной системе вокруг солнца.

3.4.6. Составьте клиент проверки, моделирующий движение в двух разных областях пространства (определенных двумя разными файлами и отображаемых в двух разных частях окна стандартного графического устройства). Придется также изменить метод `draw()` класса `Body`.

3.4.7. Составьте класс `RandomBody`, который инициализирует свои переменные экземпляра (тщательно подобранными) случайными значениями, а не получает их в аргументах. Затем составьте клиент, получающий из командной строки один аргумент `n` и моделирующий движение в случайной области с `n` телами.

3.4.8. Измените класс `Vector`, включив в него метод `__iadd__(self, other)`, для поддержки оператора `+=`, позволяющего клиенту составлять такой код, как `r += v.scale(dt)`. Пересмотрите программы `body.py` и `universe.py` с использованием этого метода.

3.4.9. Измените конструктор `Vector` так, чтобы при передаче в аргументах положительного целого числа `d` он создавал и возвращал нулевой вектор размерности `d`. Используя этот измененный конструктор, пересмотрите программу `universe.py` так, чтобы она работала с областями в трех (или более) размерностях. Не волнуйтесь об изменении в программе `body.py` метода `draw()` — пусть проецирует позицию на плоскость, определенную первыми координатами x и y.

Практические упражнения

3.4.10. *Новая область.* Разработайте новую область с интересными свойствами и смоделируйте движение в ней при помощи класса `Universe`. Это упражнение — хорошая возможность проявить творческий подход!

3.4.11. *Просачивание.* Составьте объектно-ориентированную версию программы 2.4.6 (`percolation.py`). Тщательно продумайте проект, прежде чем начинать его реализацию, и будьте готовы защищать свои проектные решения.

Глава 4 Алгоритмы и структура данных

4.1. Эффективность 500
4.2. Сортировка и поиск 542
4.3. Стеки и очереди 576
4.4. Таблицы идентификаторов 619
4.5. Случай из практики: феномен "тесного мира" 667

Эта глава знакомит с фундаментальными типами данных, являющимися основой стандартных блоков разнообразных приложений. Даже при том, что некоторые из них встроены в Python, мы представляем их полные реализации, чтобы у вас было четкое представление о том, как они работают и почему они важны.

Объекты могут содержать ссылки на другие объекты, поэтому мы можем создавать *связанные структуры* (linked structure) произвольной сложности. Обладая связанными структурами и массивами, можно создавать *структуры данных* (data structure) для организации информации таким способом, чтобы эффективно обработать ее с соответствующими *алгоритмами*. В типе данных мы используем набор значений, чтобы создавать структуры данных и методы для работы с этими значениями и реализации алгоритмов.

Рассматриваемые в этой главе алгоритмы и структуры данных представляют собой совокупность знаний, выработанных за несколько прошедших десятилетий и составляющих основу эффективного использования компьютеров и широкого разнообразия приложений. Описываемые здесь фундаментальные подходы стали основой в научных исследованиях: от задач моделирования N тел в физике и упорядочения генов в биоинформатике до систем баз данных в поисковых механизмах и методов организации вычислений коммерческих данных. Поскольку область применения компьютерных приложений продолжает расширяться, растет и важность этих фундаментальных подходов.

Алгоритмы и структуры самих данных — вполне допустимые темы современных научных исследований. Поэтому мы начинаем с описания научного подхода

для анализа эффективности алгоритмов, используемых повсюду в этой главе, чтобы изучить характеристики производительности наших реализаций.

4.1. Эффективность

Программы этого раздела...

Программа 4.1.1. Задача суммирования триплетов (`threesum.py`) 503

Программа 4.1.2. Проверка гипотезы удвоения (`doublingtest.py`) 505

В этом разделе вы научитесь соблюдать принцип, кратко выражаемый еще одной мантрой программирования: *обращайте внимание на стоимость*. Если вы станете инженером, то это будет вашей работой; если станете биологом или физиком, то стоимость продиктует, какие научные задачи вы сможете решать; если будете заниматься бизнесом или станете экономистом, то этот принцип не будет нуждаться ни в каких комментариях; а если вы станете разработчиком программного обеспечения, то именно стоимость продиктует, будет ли созданное вами программное обеспечение использоваться вашими клиентами.

Чтобы узнать стоимость выполнения своих программ, мы применяем для их изучения тот же *научный метод*, что и ученые в поисках знаний о естественном мире. Для получения упрощенных моделей стоимости мы применяем также *математический анализ*.

Какие особенности естественного мира мы изучаем? В большинстве ситуаций нас интересует одна фундаментальная характеристика: *время*. Всякий раз запуская программу, мы проводим эксперимент в реальном мире, проводя сложную систему электронных схем через последовательность изменяющихся состояний, подразумевающих огромное количество отдельных событий, чтобы в конечном счете она пришла в состояние, интерпретируемое нами как результат. Хотя и разработанные в абстрактном мире программирования Python, эти события вполне определенно происходят в естественном мире. Сколько пройдет времени, пока мы не увидим результат? Для нас имеет большое значение, будет ли это миллисекунда, секунда, день или неделя. Поэтому нам нужен научный метод, позволяющий корректно контролировать ситуацию, так же, как и при запуске ракеты, постройке моста или расщеплении атомного ядра.

С одной стороны, современные программы и среды программирования сложны; с другой стороны, они разработаны на базе набора простых (но мощных) абстракций. Это маленькое чудо, что программа приходит к тому же результату при каждом запуске. Для прогнозирования необходимого времени мы используем ту же относительно простую вспомогательную инфраструктуру, что и при создании программы. Вас может удивить легкость разработки оценок стоимости и предсказания характеристик производительности большинства составляемых программ.

Научный метод. Следующие пять этапов кратко характеризуют научный подход.

- *Наблюдение* явлений естественного мира.
- *Гипотеза* как модель, объясняющая наблюдения.
- *Прогноз* событий на основе гипотезы.
- *Подтверждение* прогноза на основании дальнейших наблюдений.
- *Проверка* в ходе многократных подтверждений совпадения гипотезы и наблюдений.

Один из ключевых принципов научного метода заключается в том, что разрабатываемые нами эксперименты должны быть *воспроизводимыми*, чтобы другие могли убедить себя в правильности гипотезы. Кроме того, формулируемые нами гипотезы должны быть *опровергаемы* — нужна возможность знать наверняка, что гипотеза стала неправильной (а следовательно, нуждается в пересмотре).

Наблюдение. Наша первая задача — количественное измерение продолжительности выполнения наших программ. Хотя точное измерение продолжительности выполнения программы затруднительно, обычно хватает и приблизительного. Существует множество инструментов, способных помочь в получении приблизительных значений. Самый простой, вероятно, — это физический секундомер или тип данных `Stopwatch` (см. программу 3.2.2). Можно просто запускать программу с разными входными данными и измерять отрезок времени, необходимый для каждого случая.

```
% python threesum.py < 1Kints.txt
```

тик тик тик

```
0
% python threesum.py < 2Kints.txt
```

тик тик тик тик тик тик
тик тик тик тик тик тик
тик тик тик тик тик тик
тик тик тик тик тик тик

```
2
391930676 -763182495 371251819
-326747290 802431422 -475684132
```

Наблюдение продолжительности выполнения программы

Наше первое качественное наблюдение для большинства программ — выявление *сложности задачи* (problem size), характеризующей трудность ее решения. Обычно сложность задачи — это либо размер ввода, либо значение аргумента командной строки. Интуитивно понятно, что продолжительность выполнения должна увеличиваться с увеличением сложности задачи, вопрос в том, *насколько* она увеличивается.

Следующее качественное наблюдение для многих программ — насколько продолжительность выполнения программы нечувствительна к вводу (прежде всего это зависит от сложности задачи). Если эта взаимозависимость не наблюдается, необходимо провести больше экспериментов и лучше понять происходящее, а возможно, нужен лучший контроль зависимости продолжительности выполнения от ввода. Поскольку это соотношение обычно действительно наблюдается, необходимо сосредоточиться на задаче лучшего определения соответствия между сложностью задачи и продолжительностью выполнения.

В качестве конкретного примера запустим программу 4.1.1 (`threesum.py`), вычисляющую количество триплетов в массиве из n чисел, сумма которых равна нулю. Это вычисление может показаться надуманным, но оно глубоко связано с многочисленными фундаментальными вычислительными задачами, особенно в вычислительной геометрии, поэтому данная задача достойна внимательного исследования. Так какова взаимосвязь между сложностью задачи n и продолжительностью выполнения программы `threesum.py`?

Гипотеза. На заре информатики Дональд Эрвин Кнут (Donald Knuth) доказал, что, несмотря на постоянное усложнение факторов, влияющих на продолжительность выполнения программ, *в принципе* вполне возможно создать точные модели, позволяющие точно прогнозировать продолжительность выполнения конкретных программ. Подобный анализ подразумевает следующее:

- детальное понимание программы;
- детальное понимание системы и компьютера;
- передовые инструментальные средства математического анализа.

Лучше оставим это экспертам. Но каждый программист должен знать, как быстро и легко оценить производительность. К счастью, эти навыки можно приобрести, используя комбинацию эмпирических наблюдений и небольшой набор математических инструментов.

Гипотеза удвоения. Для очень многих программ мы вполне можем быстро сформулировать гипотезу для следующего вопроса: *каково влияние на продолжительность выполнения удвоения размера ввода?* Это *гипотеза удвоения* (doubling hypothesis). Возможно, проще всего обратить внимание на стоимость — это задать себе этот вопрос во время разработки программы, а также при использовании их на практике. Далее мы опишем, как найти ответы, используя научный метод.

Эмпирический анализ. Конечно, мы можем начать с удвоения размера ввода и проверить результат влияния на продолжительность. Например, программа 4.1.2 (`doublingtest.py`) создает последовательность случайных входных массивов для программы `threesum.py`. На каждом этапе можно удваивать длину массива и выводить соотношение продолжительности выполнения метода `threesum.countTriples()` для каждого текущего ввода и предыдущего (имевшего половинный размер). Запустив такую программу, вы начнете цикл прогнозирования и проверки: она очень быстро выведет несколько строк, а затем начнет замедляться. Каждый раз, когда она выводит строку, вы задаете себе вопрос: какой длины она будет, пока не появится следующая строка? Проверяя работу программы с секундомером, можно заметить, что проходящее между выводом строк время увеличивается примерно кратно 8. Этот прогноз проверяется измерениями с использованием типа `Stopwatch`, создающего вывод программы и непосредственно подводящего к гипотезе, что при удвоении объема входных данных продолжительность выполнения увеличивается кратно 8. Мы могли бы также нарисовать

стандартный график продолжительности, демонстрирующий зависимость продолжительности от размера ввода, или логарифмический график такой же зависимости. В случае программы `threesum.py` логарифмический график — это прямая линия с наклоном 3, однозначно предлагающая гипотезу о том, что продолжительность подчиняется степенному закону в форме cn^3 (см. упр. 4.1.29).

Программа 4.1.1. Задача суммирования триплетов (`threesum.py`)

```
import stdarray
import stdio

def writeTriples(a):
    # См. упражнение 4.1.1.

def countTriples(a):
    n = len(a)
    count = 0
    for i in range(n):
        for j in range(i+1, n):
            for k in range(j+1, n):
                if (a[i]+a[j]+a[k]) == 0:
                    count += 1
    return count

def main():
    a = stdarray.readInt1D()
    count = countTriples(a)
    stdio.writeln(count)
    if count < 10:
        writeTriples(a)

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

`a[]`	Массив целых чисел
`n`	Длина `a[]`
`count`	Количество триплетов, сумма которых равна 0

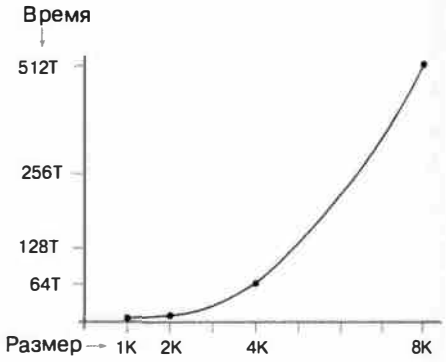
Эта программа читает со стандартного ввода массив целых чисел и выводит количество триплетов в массиве, сумма которых равна нулю. Если количество триплетов мало, то она выводит также и сами триплеты. Файл `1000ints.txt` содержит 1 000 случайных 32-разрядных целых чисел (от -2^{31} до $2^{31}-1$).

```
% more 8ints.txt
  8
 30
-30
-20
-10
 40
  0
 10
  5
```

```
% python threesum.py < 8ints.txt
4
 30  -30    0
 30  -20  -10
-30  -10   40
-10    0   10

% python threesum.py < 1000ints.txt
0
```

Математический анализ. Фундаментальный подход Кнута к построению математических моделей существенно упрощает описание продолжительности выполнения программы — полная продолжительность определяется двумя основными факторами.

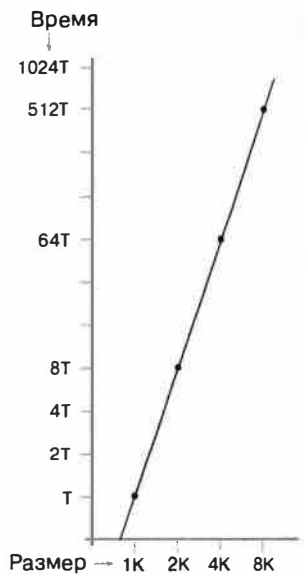
- Стоимость выполнения каждого оператора.
- Частота выполнения каждого оператора.



Стандартный график

Первый фактор — это свойство системы, а второй — свойство алгоритма. Если оба известны для всех инструкций в программе, их можно перемножить и суммировать для всех инструкций в программе, получив общую продолжительность.

Основная задача — определение частоты исполнения операторов. Некоторые операторы проанализировать просто; например, оператор, инициализирующий переменную `count` значением `0` в функции `threesum.countTriples()`, выполняется только однажды. Другие требуют более тщательного анализа; например, оператор `if` в функции `threesum.countTriples()` выполняется точно $n(n-1)(n-2)/6$ раз (количество способов выбора трех разных чисел из входного массива; см. упр. 4.1.5).



Логарифмический график

Анализ частот такого типа способен привести к сложным и длинным математическим выражениям. Для существенного упрощения математического анализа мы разработаем упрощенные *приблизительные* выражения двумя способами. Во-первых, мы работаем с *ведущими членами* (leading term) математических выражений, используя такие математические механизмы, как *запись с использованием тильды* (tilde notation). Мы пишем $\sim f(n)$, чтобы представить любое значение, которое, будучи разделено на $f(n)$, стремится к 1 при росте n. Мы также пишем $g(n) \sim f(n)$, когда необходимо указать, что $g(n)/f(n)$ стремится к 1 при росте n. При этой форме записи мы можем игнорировать сложные части выражения, не имеющие большого значения. Например, оператор `if` в программе `threesum.py` выполняется $\sim n^3/6$ раз, поскольку $(n-1)(n-2)/6 = n^3/6 - n^2/2 + n/3$, что будучи разделено на $n^3/6$, приближается к 1 при росте n. Эта форма записи полезна, когда члены после ведущего члена являются относительно незначительными (например, когда n = 1 000, его предполагаемое значение состав-

ляет $-n^2/2 + n/3 \approx -499\,667$, что относительно незначительно по сравнению с $n^3/6 \approx$ $\approx 166\,666\,667$). Во-вторых, мы сосредоточиваемся на инструкциях, выполняющихся наиболее часто, — речь идет о *внутреннем цикле* (inner loop) программы. В этой программе имеет смысл считать, что время, отведенное инструкциям вне внутреннего цикла, относительно незначительно.

Программа 4.1.2. Проверка гипотезы удвоения (doublingtest.py)

```
import sys
import stdarray
import stdio
import stdrandom
import threesum
from stopwatch import Stopwatch

# Время решения случайного экземпляра с суммой триплетов размером n.
def timeTrial(n):
    a = stdarray.create1D(n, 0)
    for i in range(n):
        a[i] = stdrandom.uniformInt(-1000000, 1000000)
    watch = Stopwatch()
    count = threesum.countTriples(a)
    return watch.elapsedTime()

n = int(sys.argv[1])
while True:
    previous = timeTrial(n // 2)
    current = timeTrial(n)
    ratio = current / previous
    stdio.writef('%7d %4.2f\n', n, ratio)
    n *= 2
```

n	Сложность задачи
a[ ]	Случайные целые числа
watch	Секундомер

n	Сложность задачи
previous	Продолжительность для n // 2
current	Продолжительность для n
ratio	Соотношение продолжительности

Эта программа выводит таблицу соотношений удвоения для задачи суммы триплетов. Таблица демонстрирует, как задача удвоения размера затрагивает продолжительность вызова функции `threesum.countTriples()` для сложностей задачи, удваивающихся для каждого ряда таблицы. Эти эксперименты приводят к гипотезе, что продолжительность увеличивается кратно 8 при удвоении объема входных данных. Запустив программу, обратите особое внимание на то, что прошедшее время увеличивается кратно 8 для каждой выведенной строки, подтверждая правильность гипотезы.

```
% python doublingtest.py 256
 256 7.52
 512 8.09
1024 8.07
2048 7.97
...
```

```
import stdarray
import stdio

def writeTriples(a):
    # See Exercise 4.1.1

def countTriples(a):
    n = len(a)                                          1
    count = 0
    for i in range(n):                                  n
        for j in range(i+1, n):                         ~n²/2
            for k in range(j+1, n):                     ~n³/6
                if (a[i] + a[j] + a[k]) == 0:
                    count += 1
    return count

def main():
    a = stdarray.readInt1D()
    count = countTriples(a)
    stdio.writeln(count)
    if count < 10:
        writeTriples(a)
if __name__ == '__main__': main()
```

Внутренний цикл

Зависит от входных данных

Анатомия частот выполнения операторов в программе

Ключевой пункт в анализе продолжительности выполнения программы таков: для очень многих программ продолжительность удовлетворяет соотношению

$$T(n) \sim cf(n)$$

где c — константа; $f(n)$ — функция, известная как *порядок роста* (order of growth) продолжительности. Для типичных программ $f(n)$ — это такая функция, как $\log n$, n, $n \log n$, n^2 или n^3, как вы вскоре увидите (обычно мы выражаем функции порядка роста без постоянного коэффициента). Когда функция f(n) является степенью n, как это часто бывает, данное предположение эквивалентно высказыванию, что продолжительность удовлетворяет экспоненциальному закону. В случае программы `threesum.py` эта гипотеза уже проверена нашими эмпирическими наблюдениями: *порядок роста продолжительности программы* `threesum.py` составляет n^3. Значение константы c зависит и от стоимости выполняющихся инструкций, и от деталей анализа частот, но мы обычно не обязаны учитывать это значение, как вы сейчас узнаете.

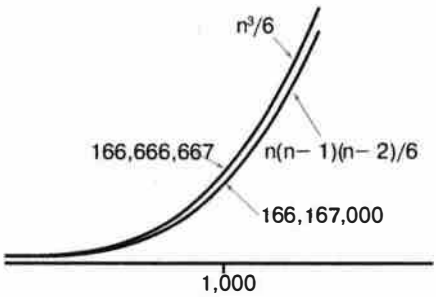
Порядок роста — простой, но мощная модель продолжительности. Например, знание порядка роста, как правило, непосредственно ведет к гипотезе удвоения. В случае программы `threesum.py`, зная, что порядок роста составляет n^3, можно ожидать, что продолжительность выполнения будет увеличиваться кратно 8 при удвоении сложности задачи, поскольку

$$T(2n) / T(n) \approx c(2n)^3 / (cn^3) = 8.$$

Это соответствует значению, полученному в результате эмпирического анализа, а следовательно, подтверждает правильность и модели, и эксперимента.

Тщательно изучите этот пример, поскольку этот метод можно использовать для выяснения производительности любой создаваемой вами программы.

Дональд Кнут установил, что вполне возможно разработать точную математическую модель продолжительности выполнения любой программы, и множество экспертов занимаются разработкой таких моделей. Но для выяснения производительности своих программ столь детальные модели вам не нужны: обычно стоимость инструкций вне внутреннего цикла вполне можно игнорировать (поскольку эта стоимость незначительна по сравнению со стоимостью инструкций во внутреннем цикле), в значении константы при приближенном вычислении продолжительности нет необходимости (поскольку при использовании гипотезы удвоения для прогноза оно отбрасывается).



Аппроксимация ведущего члена

Анализ продолжительности выполнения программы (пример)

Количество инструкций	Время инструкции в секундах	Частота	Полное время
6	2×10^{-9}	$n^3/6 - n^2/2 + n/3$	$(2n^3 - 6n^2 + 4n) \times 10^{-9}$
4	3×10^{-9}	$n^2/2 - n/2$	$(6n^2 - 6n) \times 10^{-9}$
4	3×10^{-9}	n	$(12n) \times 10^{-9}$
10	1×10^{-9}	1	10×10^{-9}
		Общий итог	$(2n^3 + 10n + 10) \times 10^{-9}$
		Тильда	$\sim 2n^3 \times 10^{-9}$
		Порядок роста	n^3

Эти приближения важны, поскольку они связывают абстрактный мир программы Python с реальным миром выполняющих их компьютеров. Приближения таковы, что характеристики конкретной используемой машины не играют существенной роли в моделях — анализ отделяет *алгоритм* от *системы*. Порядок роста продолжительности выполнения программы `threesum.py` составляет n^3, независимо от того, что она реализована на языке Python или выполняется на ноутбуке, мобильном телефоне или суперкомпьютере; он скорее зависит от того факта, что исследуются все триплеты. Свойства компьютера и *системы* суммируются с различными предположениями об отношениях между операторами программы и машинными инструкциями, а также фактически наблюдаемой продолжительностью и служат основанием для гипотезы удвоения. Порядок роста определяет используемый *алгоритм*. Это разделение — мощная концепция, поскольку она позволяет нам выработать знание о производительности алгоритмов, а затем применить это знание к любому компьютеру. Фактически бо́льшая

часть знаний о производительности классических алгоритмов была получена несколько десятилетий назад, но это знание все еще актуально для современных компьютеров.

Подобные эмпирические и математические исследования составляют модель (объяснение происходящего), которая может быть формализована в список всех упомянутых выше предположений (каждая инструкция занимает каждый раз тот же период времени, у продолжительности выполнения есть данная форма и т.д.). Не многие программы достойны детализированной модели, но необходимо представление об ожидаемой продолжительности каждой составляемой программы. *Обращайте внимание на стоимость.* Формулировка гипотезы удвоения (через эмпирические исследования, математический анализ или оба (что предпочтительно)) является хорошим началом. Эта информация о производительности чрезвычайно полезна, и вскоре вы будете формулировать и подтверждать правильность гипотез каждый раз, когда составляете программу. Действительно, на это стоит потратить то время, пока вы ожидаете завершения выполнения своей программы!

Классификация порядков роста. Для написания программ Python мы используем лишь несколько структурных примитивов (операторы, условные выражения, циклы и вызовы функций), поэтому порядок роста наших программ — зачастую лишь одна из нескольких функций от сложности задачи, приведенных в таблице ниже. Эти функции непосредственно приводят к гипотезе удвоения, которую мы можем проверить при запуске программ на выполнение. А вы уже *запускали* программы, демонстрирующие этот порядок роста, как обсуждается далее.

Постоянный. Программа с *постоянным* (constant) порядком роста продолжительности выполняет фиксированное количество операторов для решения своей задачи; следовательно, ее продолжительность выполнения не зависит от сложности задачи. К этой категории относится несколько наших первых программ из главы 1, таких как программа 1.1.1 (`helloworld.py`) и программа 1.2.5 (`leapyear.py`): каждая из них выполняет несколько операторов только один раз.

Все операции Python со стандартными числовыми типами требуют времени. Таким образом, применение операции к большому числу требует не больше времени, чем ее применение к малому числу. (За одним исключением: операции, задействующие целые числа с огромными количеством цифр, могут занять больше времени; см. подробности в разделе вопросов и ответов в конце этого раздела). Выполнение функций из модуля Python `math` также занимает время. Обратите внимание: мы не определяем размер константы. Например, константа для функции `math.atan2()` несколько больше константы для функции `math.hypot()`.

Логарифмический. Программа с *логарифмическим* (logarithmic) порядком роста продолжительности лишь ненамного медленнее программы с постоянным порядком роста. Классический пример программы с логарифмической

зависимостью продолжительности выполнения от сложности задачи — это поиск элемента в отсортированном массиве (см. в следующем разделе программу 4.2.3, `binarysearch.py`). Основание логарифма неважно для порядка роста (поскольку все логарифмы с постоянным основанием связаны с постоянным коэффициентом), поэтому, когда речь идет о порядке роста, мы используем log *n*. Иногда мы пишем более точные формулы, используя lg *n* (основание 2, или *двоичный логарифм*), или ln *n* (основание e, или *натуральный логарифм*), так как оба вполне естественны при изучении компьютерных программ. Например, округленный в бо́льшую сторону lg *n* — это количество битов в двоичном представлении *n*, а ln *n* возникает при анализе бинарных деревьев поиска (см. раздел 4.4).

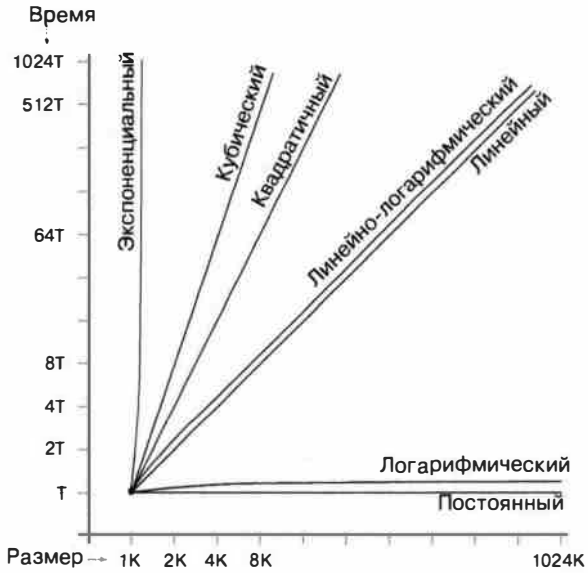
Общие функции роста

Порядок роста

Описание	Функция	Множитель для гипотезы удвоения
Постоянный	1	1
Логарифмический	log *n*	1
Линейный	*N*	2
Линейно-логарифмический	*N* log *n*	2
Квадратичный	N^2	4
Кубический	N^3	8
Экспоненциальный	2^n	2^n

Линейный. Программы, тратящие постоянные периоды времени на обработку каждой части входных данных (или основанные на одном цикле `for`), весьма распространены. Такая программа имеет *линейную* (linear) зависимость продолжительности от сложности задачи. Программа 1.5.3 (`average.py`), вычисляющая среднее для значений, поступающих со стандартного ввода, относится к этому типу, равно как и наш код перетасовки элементов массива в разделе 1.4. Фильтры, такие как программа 1.5.5 (`plotfilter.py`), также относятся к этой категории, как и рассматривавшиеся в разделе 3.2, различные фильтры обработки изображений, осуществляющие постоянное количество арифметических операции для каждого входного пикселя.

Линейно-логарифмический. Мы используем термин *линейно-логарифмический* (linearithmic) для описания программ, продолжительность выполнения которых составляет *n* log *n* для сложности задачи *n*. И опять-таки основание логарифма неважно. Например, программа `couponcollector.py` является линейно-логарифмической. Хороший пример — программа сортировки с объединением (см. программу 4.2.6). Некоторые важные задачи имеют вполне естественные квадратичные решения, но применяемые для их решения хитрые алгоритмы являются линейно-логарифмическими. Такие алгоритмы (включая сортировку с объединением) критически важны, поскольку они позволяют решать намного бо́льшие задачи, чем способны решать квадратичные подходы. В следующем разделе мы рассмотрим общую методику разработки линейно-логарифмических алгоритмов.



Порядки роста (логарифмический график)

Квадратичный. Типичная программа с порядком роста продолжительности выполнения n^2 имеет два вложенных цикла `for`, используемых для некоторого вычисления, задействующего все пары из n элементов. Парный цикл обновления силы есть в программе 3.4.2 (`universe.py`), к программам этого класса относится также программа 4.2.4, реализующая алгоритм сортировки.

Кубический. Наш пример этого раздела, `threesum.py`, имеет *кубический* (cubic) порядок роста продолжительности выполнения (n^3), поскольку он имеет три вложенных цикла `for` для обработки всех триплетов из n элементов. Продолжительность матричного умножения, реализованного в разделе 1.4 для умножения двух матриц `m` на `m`, имеет порядок роста m^3, поэтому простой алгоритм матричного умножения считается кубическим. Однако размер ввода (количество элементов в матрицах) пропорционален $n = m^2$, таким образом, этот алгоритм лучше всего классифицируется как $n^{3/2}$, а не кубический.

Экспоненциальный. Продолжительность программ 2.3.2 (`towersofhanoi.py`) и 2.3.3 (`beckett.py`) в разделе 2.3 пропорциональна 2^n, поскольку они обрабатывают все подмножества из n элементов. Обычно мы используем термин *экспоненциальный* (exponential) для описания алгоритмов с порядком роста b^n для всех постоянных $b>1$, даже при том, что различные значения b приводят к весьма разной продолжительности. Экспоненциальные алгоритмы чрезвычайно медлительны — их никогда не следует применять для больших задач. Однако они играют критически важную роль в теории алгоритмов, поскольку существует большой класс задач, для которых экспоненциальный алгоритм — наилучший возможный выбор.

Наиболее распространенные гипотезы о порядке роста

Описание	Порядок роста	Пример	Случай
Постоянный	1	`count += 1`	Оператор (инкремент целого числа)

Логарифмический	$\log n$	`while n > 0:`
		`  n = n // 2`
		`  count += 1`
			Деление на два (биты в двоичном представлении)

Линейный	n	`for i in range(n):`
		`  if a[i] == 0:`
		`  count += 1`
			Одиночный цикл

Линейно-логарифмический	$n \log n$	[см. сортировку с объединением (программа 4.2.6)]	Разделяй и властвуй (сортировка с объединением)

Квадратичный	n^2	`for i in range(n):`
		`  for j in range(i+1, n):`
		`  if (a[i]+a[j]) == 0:`
		`  count += 1`
			Двойной вложенный цикл (проверка всех пар)

Кубический	n^3	`for i in range(n):`
		`  for j in range(i+1, n):`
		`  for k in range(j+1, n):`
		`  if (a[i]+a[j]+a[k]) == 0:`
		`  count += 1`
			Тройной вложенный цикл (проверка всех триплетов)

Экспоненциальный	2^n	[см. код Грея (программа 2.3.3)]	Полный перебор (проверка всех подмножеств)

Конечно, в этой классификации приведены лишь наиболее распространенные порядки, а не полный набор. Действительно, подробный анализ алгоритмов может потребовать полной палитры математических инструментов, разработанных за столетия. Для выяснения продолжительности таких программ, как 1.3.9 (`factors.py`), 1.4.3 (`primesieve.py`) и 2.3.1 (`euclid.py`), требуются фундаментальные следствия теории чисел. Классические алгоритмы, как в программах 4.4.3 (`hashst.py`) и 4.4.4 (`bst.py`), требуют внимательного математического анализа. Программы 1.3.6 (`sqrt.py`) и 1.6.3 (`markov.py`) являются примерами числовых вычислений: их продолжительность зависит от коэффициента приближения вычисления к желаемому числовому результату. Модель Монте-Карло, программы 1.3.8 (`gambler.py`), 2.4.6 (`percolation.py`) и их варианты интересны тем, что подробные математические модели для них недоступны.

Однако у подавляющего большинства составляемых вами программ будут простые характеристики производительности, точно описываемые одним из порядков роста, рассмотренных и приведенных в таблице выше. Таким образом, обычно мы можем работать с простыми высокоуровневыми гипотезами, такими как *порядок роста продолжительности выполнения алгоритма сортировки*

объединением имеет линейно-логарифмическую зависимость от размера исходных данных. Для экономии мы сокращаем такое выражение до *сортировка объединением является линейно-логарифмической.* Большинство наших гипотез о стоимости имеет такую форму: *сортировка объединением быстрее сортировки вставкой.* Общеизвестное достоинство таких гипотез в том, что они относятся к алгоритмам, а не непосредственно к программам.

Прогноз. Вы всегда можете попытаться оценивать продолжительность выполнения программы, просто запустив ее, но при большой сложности задачи это может быть не лучшей идеей. Это будет похоже на попытку оценить, где упадет запущенная ракета, насколько разрушительной окажется ее боеголовка, и выдержит ли мост попадание.

Знание порядка роста продолжительности позволяет принимать решения о применимости к большим задачам, т.е. позволят ли имеющиеся ресурсы справиться с конкретной задачей, которую мы фактически должны решить. Как правило, мы используем результаты проверенных гипотез о порядке роста продолжительности выполнения программ одним из следующих способов.

Оценка возможности решения больших задач. Необходимо обращать внимание на стоимость, чтобы для каждой составляемой программы ответить на простой вопрос: сможет ли она обработать эти входные данные за разумный период времени? Например, кубический алгоритм, выполняющийся несколько секунд для задачи размером n, потребует нескольких недель для задачи размером $100n$, поскольку это будет в миллион раз (100^3) дольше, а несколько миллионов секунд — это несколько недель. Если необходимо решить задачу, размер которой именно таков, то нужен лучший метод. Знание порядка роста продолжительности алгоритма предоставляет точную информацию, необходимую для понимания ограничений, накладываемых на размер возможных для решения задач. Обретение такого понимания является важнейшей причиной для изучения производительности. Без этого вы, вероятно, не будете иметь никакого представления о времени, необходимом программе, а с ним вы можете быстро и легко оценить ожидаемую стоимость и продолжительность вычисления.

Влияние увеличения сложности задачи на программы, выполняющиеся несколько секунд

Порядок роста	Прогнозируемая продолжительность, если сложность задачи увеличивается в 100 раз
Линейный	Несколько минут
Линейно-логарифмический	Несколько минут
Квадратичный	Несколько часов
Кубический	Несколько недель
Экспоненциальный	Вечно

Оценка использования более быстрого компьютера. Обращая внимание на стоимость, вы также можете задаться таким вопросом: *насколько быстрее я смогу решить задачу, если использовать более быстрый компьютер?* И опять-таки, знание порядка роста продолжительности предоставит необходимую информацию. Известный эмпирический *закон Гордона Мура* (Moore's law) подразумевает, что вы можете заполучить компьютер примерно с вдвое высокой скоростью и вдвое большей памятью через 18 месяцев или компьютер с примерно в 10 раз более высокой скоростью и в 10 раз большей памятью приблизительно через 5 лет. Вполне естественно полагать, что если вы приобретете новый компьютер, в 10 раз более быстрый и имеющий в 10 раз больше памяти, чем прежний, то сможете решить задачу, в 10 раз большую, но это *не так* в случае квадратичных или кубических алгоритмов. Будь то инвестиционный банкир, ежедневно просчитывающий финансовые модели, или ученый, анализирующий экспериментальные данные, или инженер, проверяющий модель проекта, всем приходится регулярно запускать программы, выполнение которых занимает несколько часов. Предположим, вы используете программу с кубической продолжительностью, а затем покупаете новый компьютер, в 10 раз более быстрый и в 10 раз с бо́льшим объемом памяти, причем не только потому, что вам нравится новый компьютер, а потому, что сложность задачи увеличилась в 10 раз. Но вас ждет горькое разочарование — программа выполняется в 100 раз дольше, чем прежде! Это та самая ситуация, когда линейные и линейно-логарифмические алгоритмы столь ценны: при таком алгоритме новый, в 10 раз более быстрый компьютер и в 10 раз с бо́льшим количеством памяти, чем у прежнего, может решить в 10 раз бо́льшую задачу за то же время, что и прежнюю задачу на старом компьютере. Другими словами, вы не можете идти в ногу с законом Мура, если используете квадратичный или кубический алгоритм.

Влияние применения в 10 раз более быстрого компьютера для решения в 10 раз большей задачи

Порядок роста	Коэффициент увеличения продолжительности
Линейный	1
Линейно-логарифмический	1
Квадратичный	10
Кубический	100
Экспоненциальный	Вечно

Сравнение программ. Мы всегда ищем способы улучшить свои программы и зачастую можем улучшить или модифицировать свои гипотезы, а следовательно, нужно вычислять эффективность различных усовершенствований. Обладая способностью прогнозировать производительность, мы можем принимать проектные решения во время разработки, что может привести нас к лучшей, более эффективной реализации. Во многих случаях мы можем определить порядок

роста продолжительности и разработать точные гипотезы о сравнительной производительности. Порядок роста чрезвычайно полезен в этом процессе, поскольку он позволяет нам сравнить один конкретный алгоритм с целыми классами алгоритмов. Например, имея линейно-логарифмический алгоритм решения задачи, бессмысленно интересоваться квадратичными или кубическими алгоритмами для решения той же задачи.

Недостатки. Есть много причин, по которым вы могли бы получить противоречивые или неверные результаты при детальном анализе производительности программы. В основе их всех лежит ошибочность одного или нескольких принятых предположений. Мы можем разработать новые гипотезы на основании новых предположений, но чем больше учитываемых деталей, тем больше заботы следует проявить при анализе.

Время инструкции. Предположение о том, что каждая инструкция всегда занимает тот же период времени, не всегда правильна. Например, современные компьютерные системы используют для организации памяти *кеширование*, когда доступ к элементам в огромных массивах может занять намного больше времени, если они не находятся в массиве близко друг к другу. Результат кеширования можно наблюдать в программе `threesum.py`, если разрешить программе `doublingtest.py` выполняться достаточно долго. После кажущегося схождения к 8 соотношение продолжительности может перейти к большему значению для больших массивов из-за кеширования.

Не доминирующий внутренний цикл. Предположение, что внутренний цикл доминирует, может оказаться неправильным. Сложность задачи n может оказаться недостаточно большой, чтобы сделать ведущий член в математическом описании частоты исполнения инструкций во внутреннем цикле настолько большим, чтобы члены более низкого порядка можно было игнорировать. У некоторых программ есть существенный объем кода вне внутреннего цикла, и его также следует учитывать.

Системные соображения. Как правило, на компьютере происходит множество вещей. Python — это только одно из многих конкурирующих за ресурсы приложений, у самого Python также есть множество средств и элементов, существенно влияющих на производительность. Такие факторы способны налагаться на основополагающие принципы научного метода, согласно которому эксперименты должны быть воспроизводимы, но происходящее в данный момент на вашем компьютере никогда не будет воспроизведено снова. Таким образом, независимо от происходящего в вашей системе (и находящегося вне вашего контроля), влияние этих факторов *в принципе* должно быть незначительным.

Слишком близкая схожесть. Зачастую, когда мы сравниваем две разные программы для решения одной задачи, одна оказывается быстрее в некоторых ситуациях и медленнее в других. Значение может иметь лишь одно или несколько

упомянутых соображений. Среди программистов (и некоторых студентов) есть вполне естественная тенденция прилагать максимум энергии для поиска "лучшей" реализации, но такую работу лучше оставить экспертам.

Жесткая зависимость от входных значений. Одно из первых сделанных нами предположений при определении порядка роста продолжительности выполнения программы было в том, что продолжительность относительно нечувствительна к входным значениям. Когда это не так, мы можем получить противоречивые результаты или не сможем проверить свою гипотезу. У нашего примера `threesum.py` этой проблемы нет, но в данной главе приводится несколько примеров программ, продолжительность выполнения которых действительно зависит от входных значений. Зачастую главная задача проекта — устранить такую зависимость от входных значений. Если мы не можем этого сделать, необходимо тщательнее смоделировать вид входных данных, обрабатываемых при решении задачи, а это может оказаться весьма сложно. Например, если мы составляем программу для обработки генома, то как можно предугадать их разные наборы? Однако хорошая модель, точно описывающая реальные геномы, — это именно то, что нужно ученым, поэтому оценка продолжительности выполнения наших программ для данных из реального мира фактически вносит свой вклад в эту модель!

Несколько параметров. Мы сосредоточились на измерении производительности как функции *с одним* параметром *n*, обычно значением аргумента командной строки или размером ввода. Однако вполне обычно измерять производительность, используя два (или больше) параметра. Предположим, например, что массив `a[]` имеет длину `m`, а массив `b[]` — длину `n`. Рассмотрим следующий фрагмент кода, подсчитывающий количество пар `i` и `j`, для которых `a[i]+b[j]` равно `0`:

```
for i in range(m):
    for j in range(n):
        if a[i]+b[j] == 0:
            count += 1
```

В таких случаях мы рассматриваем параметры `m` и `n` по отдельности, с фиксированным одним при анализе другого. Например, порядок роста продолжительности этого фрагмента кода — `m*n`.

Несмотря на все эти недостатки, знание порядка роста продолжительности каждой программы весьма ценно для любого программиста, а описанные здесь методы мощны и широко применимы. Суть открытия Кнута в том, что мы можем применить эти методы до последней детали *в принципе* и создать подробные и точные прогнозы. Типичные компьютерные системы чрезвычайно сложны, и их подробный анализ лучше оставить экспертам, однако те же методы эффективны и для разработки приблизительных оценок продолжительности любой программы. Ученому-ракетостроителю необходимо иметь некоторое представление о том, закончится ли испытательный запуск в океане или в городе; при

медицинских исследованиях весьма желательно знать, убьет ли новый препарат бактерии или пациентов; у любого ученого или инженера, использующего компьютерную программу, должно быть некоторое представление о том, будет ли она выполняться в течение секунды или года.

Гарантии производительности. Для некоторых программ необходимо, чтобы продолжительность их выполнения была меньше определенного порога при *любом* размере входных данных. Для предоставления таких *гарантий производительности* теоретики предлагают чрезвычайно пессимистический подход: какова продолжительность в *самом плохом случае*?

Например, такой консервативный подход вполне подходит для программного обеспечения управления атомным реактором, системы авиадиспетчерской службы или тормозов автомобиля. Мы должны гарантировать, что такое программное обеспечение успеет закончить свою работу в пределах установленных границ, поскольку в противном случае результат может оказаться катастрофическим. При изучении естественного мира ученые обычно не рассматривают самый плохой случай: в биологии, например, самый плохой случай — исчезновение человеческого рода; в физике — конец мира. Но в компьютерных системах самый плохой случай может быть вполне реальной проблемой, когда ввод может быть создан другим (потенциально злонамеренным) пользователем, а не естественной природой. Например, веб-сайты, не использующие алгоритмы с гарантией производительности, уязвимы к атакам *отказа в обслуживании* (denial-of-service — DoS), когда хакеры переполняют их огромным количеством автоматически создаваемых запросов, обработка которых осуществляется медленнее, чем запланировано.

Используя научный метод, проверить гарантии производительности довольно трудно, поскольку мы не можем подтвердить такую гипотезу, как *сортировка с объединением гарантированно будет линейно-логарифмической*, не опробовав все возможные входные данные, а из-за слишком большого их количества мы не можем сделать это даже приблизительно. Мы могли бы опровергнуть такую гипотезу, предоставив входные данные, для которых сортировка с объединением окажется медленной, но как мы можем доказать, что это истина? Мы должны сделать это не экспериментально, а скорее на математических моделях.

Задача аналитика алгоритмов — обнаружить настолько много достоверной информации об алгоритме, насколько возможно, а задача разработчика приложений — применить это знание для составления программы, способной эффективно решить поставленную задачу. Например, если вы используете для решения задачи квадратичный алгоритм, но сможете найти для нее гарантировано линейно-логарифмический, то, вероятно, предпочтете последний. В редких случаях вы все еще могли бы предпочесть квадратичный алгоритм, поскольку он быстрее для некоторых видов входных данных или поскольку линейно-логарифмический алгоритм слишком сложен для реализации.

В идеале нужны алгоритмы, приводящие к четкому, ясному и компактному коду, дающему хорошую гарантию для самого плохого случая и обеспечивающего хорошую производительность для представляющих интерес входных данных. Большинство рассматриваемых в этой главе классических алгоритмов важно для широкого разнообразия приложений именно потому, что у них есть все эти свойства. Используя эти алгоритмы как модели, вы сами сможете разработать хорошие решения для своих задач.

Массивы и списки Python. Встроенный тип данных Python `list` представляет изменяемую последовательность объектов. Списки Python мы использовали в книге повсюду — напомним, что они используются как массивы, так как обеспечивают четыре основные операции над массивом: создание, индексированный доступ, индексированное присвоение и итерация. Но списки Python более общие, чем массивы, поскольку в них можно также вставлять и удалять элементы. Даже при том, что программисты Python обычно не различают списки и массивы, многие другие программисты учитывают такое различие. Например, во многих языках программирования массивы имеют фиксированную длину и не поддерживают вставку или удаление. Действительно, весь код обработки массивов, рассматривавшийся в этой книге до сих пор, использовал массивы фиксированной длины.

```
a = [3, 1, 4, 1, 5, 9]
b = [2, 7, 1]
```

Примеры операций со списком

Операция	Результат
`len(a)`	6
`a[4]`	5
`a[2:5]`	`[4, 1, 5, 9]`
`min(a)`	1
`max(a)`	9
`sum(a)`	23
`4 in a`	True
`b+[0]`	`[2, 7, 1, 0]`
`b += [6]`	**`[2, 7, 1, 6]`**
`del b[1]`	**`[2, 1, 6]`**
`b.insert(2, 9)`	**`[2, 1, 9, 6]`**
`b.reverse()`	**`[6, 1, 9, 2]`**
`b.sort()`	**`[1, 2, 6, 9]`**

В таблице ниже приведены наиболее используемые операции со списками Python. Обратите внимание, что некоторые операции (индексация, разделение, конкатенация, удаление, вхождение и итерация) обладают прямой языковой поддержкой в виде специального синтаксиса. Как было показано в таблице выше, некоторые из этих операций возвращают значение, а другие видоизменяют сам список.

Мы отложили этот API до данного раздела потому, что программисты, использующие списки Python, не следуя нашей мантре, *обращайте внимание на стоимость*, нарываются на неприятности. Рассмотрим, например, два фрагмента кода:

```
# квадратичное время
a = []
for i in range(n):
    a .insert(0, 'slow')
```

```
# линейное время
a = []
for i in range(n):
    a.insert(i, 'fast')
```

Код слева имеет квадратичное время, код справа — линейное. Чтобы понять, почему у операций со списками Python такие характеристики производительности, необходимо больше узнать о представлении *массива переменного размера* (resizing array), лежащего в основе списков Python.

Часть API для встроенного типа данных Python `list`

Операция	Описание
	Операции постоянного времени
`len(a)`	Длина `a`
`a[i]`	`i`-й элемент `a`
`a[i] = v`	Заменяет `i`-й элемент `a` элементом `v`
`a += [v]`	Добавляет элемент `v` в конец `a`
`a.pop()`	Удаляет элемент `a[len(a)-1]` из списка и возвращает его
	Операции линейного времени
`a+b`	Конкатенация `a` и `b`
`a[i:j]`	`[a[i], a[i+1], ..., a[j-1]]` (разделение)
`a[i:j] = b`	`a[i] = b[0], a[i+1] = b[1], ...` (присвоение части)
`v in a`	`True`, если `a` содержит `v`, и `False` в противном случае (вхождение)
`for v in a:`	Перебор элементов `a`
`del a[i]`	Удаляет элемент `a[i]` из списка (индексированное удаление)
`a.pop(i)`	Удаляет элемент `a[i]` из списка и возвращает его
`a.insert(i, v)`	Вставляет элемент `v` в `a` перед `a[i]`
`a.index(v)`	Индекс первого вхождения элемента `v` в списке `a`
`a.reverse()`	Меняет порядок элементов в `a` на обратный
`min(a)`	Минимальный элемент в `a`
`max(a)`	Максимальный элемент в `a`
`sum(a)`	Сумма элементов `a`
	Операции линейно-логарифмического времени
`a.sort()`	Меняет порядок элементов в `a` на возрастающий

Примечание 1. `a += [v]` и `a.pop()` являются "амортизированными" операциями постоянного времени.

Примечание 2. `del a[i]`, `pop(i)` и `insert(i, v)` являются "амортизированными" операциями постоянного времени, если `i` близко к `len(a)`.

Примечание 3. Время операции разделения линейно зависит от длины частей.

Примечание 4. Элементы `a` должны иметь типы, совместимые с операциями `min()`, `max()` и `sum()`.

Массив переменного размера (resizing array) — это структура данных, хранящая последовательность элементов (не обязательно фиксированную по длине), к которым можно обратиться по индексу. Для реализации массива переменного размера (на машинном уровне) Python использует массив фиксированной длины (размещенный в памяти как один непрерывный блок) для хранения ссылок

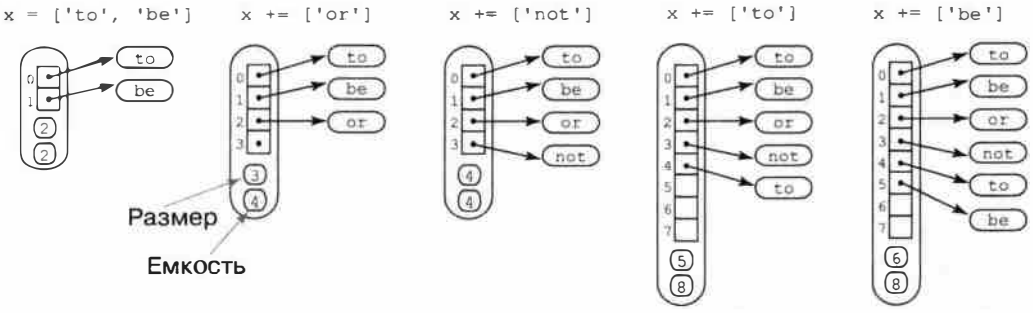
на элементы. Массив делится на две логические части: первая часть массива содержит последовательность элементов, вторая часть не используется и зарезервирована для последующих вставок. Таким образом, мы можем добавлять или удалять элементы с конца списка за постоянное время, используя зарезервированное пространство. Мы используем термин *размер* (size) для описания количества элементов в структуре данных, а термин *емкость* (capacity) для описания всей длины основного массива.

Основная сложность — гарантировать емкость структуры данных, достаточную для содержания всех элементов, но в то же время не столь большую, чтобы тратить впустую чрезмерный объем памяти. Решение этих двух задач оказывается удивительно простым.

Вначале, если необходимо добавить элемент в конец массива переменного размера, мы проверяем его емкость. Если место есть, мы просто вставляем новый элемент в конец. В противном случае мы *удваиваем* его емкость, создав новый массив вдвое большей длины, и копируем в него элементы из прежнего массива.

Аналогично при удалении элемента с конца массива переменного размера мы проверяем его емкость. Если он чрезмерно велик, мы *делим* его емкость на два, создавая новый массив половинной длины и копируя в него элементы. Обычно проверка выясняет, не меньше ли размер такого массива *одной четверти* его емкости. Таким образом, после уменьшения емкости массива переменного размера вдвое он окажется заполненным приблизительно наполовину и будет готов к вставке примерно такого же количества элементов, прежде чем снова возникнет необходимость в изменении его емкости.

Стратегия удвоения и деления гарантирует, что массив переменного размера остается заполненным от 25 до 100%, и занимаемое им пространство будет линейно зависеть от количества элементов. Но эта стратегия — не догма. Например, типичные реализации Python увеличивают емкость на 9/8 (а не вдвое), когда массив переменного размера полон. Это занимает впустую меньше пространства, но приводит к более частым увеличениям и сокращениям.



Изменение размера структуры данных, представляющей список Python

Амортизационный анализ. Стратегия удвоения и деления — разумный компромисс между напрасной тратой пространства (при слишком большой емкости в массиве остается много неиспользуемого пространства) и напрасной тратой времени (на создание нового массива или реорганизацию существующего). Однако важнее всего то, что стоимость операций удвоения и деления всегда незначительна (в пределах постоянного множителя) относительно стоимости других операций со списком Python.

Таким образом, начиная с пустого списка Python, любая последовательность из n операций, отнесенных в API на стр. 518 к операциям линейного времени, занимает время, линейно зависимое от n. Другими словами, *общая стоимость любой последовательности таких операций со списком Python, деленная на количество операций, ограничивается константой.* Это *амортизационный анализ* (amortized analysis). Данная гарантия не столь строга по сравнению с гарантией постоянного времени выполнения каждой операции, но во многих случаях у нее тот же смысл (например, когда нас в первую очередь интересует полная продолжительность). Подробности мы оставляем в качестве упражнения для любителей математики.

Для частного случая последовательности из n вставок в пустой массив переменного размера идея проста: каждая вставка занимает время добавления элемента; каждая вставка, приводящая к изменению размера (когда текущий размер — степень числа 2), занимает время, пропорциональное n, необходимое для копирования элементов из старого массива длиной n в новый массив длиной $2n$. Таким образом, если n является степенью числа 2 (для простоты), общая стоимость пропорциональна

$$(1 + 1 + 1 + \ldots + 1) + (1 + 2 + 4 + 8 + \ldots + n) \sim 3n.$$

Первый член (суммирующий до n) объясняет n операций вставки; второй член (суммирующий до $2n - 1$) объясняет $\lg n$ операций изменения размеров.

Понимание массивов переменного размера важно в программировании Python. Например, это объясняет, почему создание списка Python из n элементов при последовательном добавлении элементов в конец занимает время, пропорциональное n (и почему создание списка из n элементов последовательным добавлением элементов в начало занимает время, пропорциональное n^2). Подобные проблемы производительности — серьезная причина, по которой мы рекомендуем использовать более узкие интерфейсы, гарантирующие эффективность.

Строки. Тип данных Python `string` имеет много сходств со списком Python, за одним очень важным исключением: *строки неизменны.* Знакомя со строками, мы не подчеркивали этот факт, но именно он существенно различает строки и списки. Например, в то время вы могли не обратить внимание на невозможность изменить символ в строке. Вы могли бы подумать, что смогли бы в строке `s`, содержащей значение `'hello'`, легко преобразовать строчную букву в прописную оператором `s[0] = 'H'`, но это приведет к этой ошибке времени выполнения:

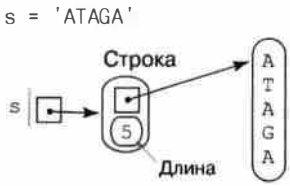
```
TypeError: 'str' object does not support item assignment
```

Если вам необходима строка `'Hello'`, создайте совершенно новую строку. Это различие укрепляет идею неизменности и имеет существенное значение для производительности, исследуемой нами.

Внутреннее представление. Внутренне Python использует намного более упрощенное представление строк, чем у списков и массивов, как показано на рисунке ниже. А именно: строковый объект содержит две части информации:

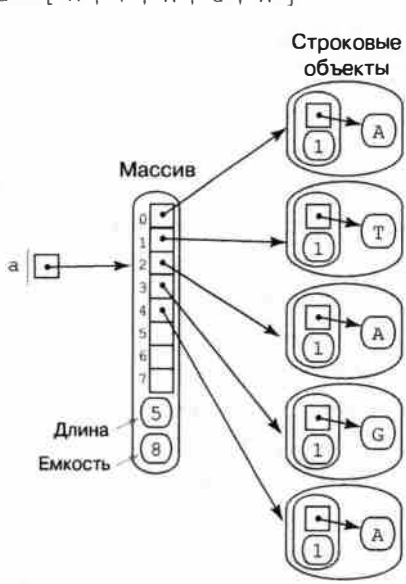
- ссылка на область в памяти, где последовательно хранятся символы строки;
- длина строки.

На другом рисунке представлен массив односимвольных строк. Более подробный анализ приведен далее в этом разделе, а пока заметим, что строковое представление, конечно, значительно упрощено. Оно использует намного меньше пространства для каждого символа и обеспечивает ускоренный доступ к каждому из них. Во многих случаях эти характеристики очень важны, поскольку строки могут быть очень длинными. Важно как то, что памяти используется не намного больше, чем необходимо для самих символов, так и то, что к символам можно быстро обратиться по индексу, как в массиве.



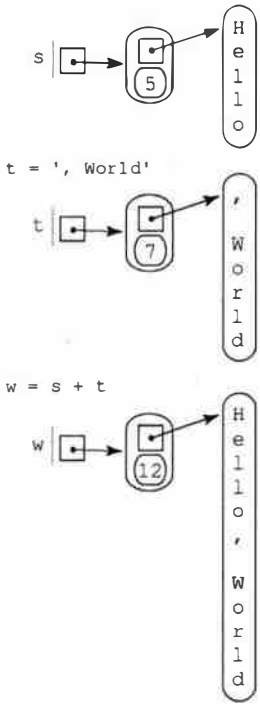
Строка Python

Производительность. Что касается массивов, то индексированный доступ и вычисление длины строк являются операциями постоянного времени. Из API в начале раздела 3.1 ясно, что большинство других операций занимает время, зависящее от длины входной строки или строк, поскольку речь идет о *копии* строки. В частности, *конкатенация символов в строку занимает время, а конкатенация двух строк занимает время, пропорциональное длине результата.* Пример приведен ниже. С точки зрения производительности — это наиболее значительное различие между строками и списками с массивами: в Python нет строк изменяемого размера, так как строки неизменны.



Массив односимвольных строк

Пример. Непонимание эффективности при конкатенации строк зачастую приводит к ошибкам производительности. Наиболее распространенная ошибка производительности — создание длинной строки, по одному символу за раз. Рассмотрим, например,



Конкатенация строк

следующий фрагмент кода, создающего новую строку с расположением символов в обратном порядке относительно символов в строке s:

```
n = len(s)
reverse = ''
for i in range(n):
    reverse = s[i] + reverse
```

Во время перебора i в цикле for оператор конкатенации строк создает строку длиной $i + 1$. Таким образом, общая продолжительность пропорциональна $1 + 2 + \ldots + n \sim n^2/2$ (см. упр. 4.1.4). Поэтому время выполнения данного фрагмента кода — *квадратичная* функция от длины строки n (см. упр. 4.1.13 для решения с линейным временем).

Понимание различий между такими типами данных, как строки и списки, критически важно при изучении любого языка программирования, и программистам следует быть бдительными и избегать ошибок производительности, подобных описанным только что. Это имело место с появления абстракции данных. Например, у строкового типа данных в языке C, разработанного в 1970-х годах, линейная функция зависимости времени от длины. Однако бесчисленные программисты, использующие для перебора строки длину в цикле for, сталкивались с квадратичной функцией там, где было простое линейное решение задачи. Это типичная ошибка производительности.

Память. Подобно продолжительности выполнения, использование программой памяти непосредственно связано с физическим миром: существенный объем микросхем вашего компьютера позволяет вашей программе сохранять значения, а впоследствии возвращать их. Чем больше значений необходимо хранить в любой конкретный момент, тем больше необходимо микросхем. Обращать внимание на стоимость необходимо, чтобы знать об использовании памяти. Вы, вероятно, знаете о пределах использования памяти на вашем компьютере (даже больше, чем о продолжительности выполнения), поскольку заплатили дополнительные деньги за больший объем памяти.

Кстати, вы должны знать, что гибкость, полученная при переходе от простейшего программирования к объектно-ориентированному (где все является объектами, даже логические значения), имеет свою стоимость, и одна из наиболее значительных ее составляющих — используемый объем памяти. Все станет понятней после рассмотрения нескольких конкретных примеров.

Использование памяти на вашем компьютере для Python точно определено (каждое значение требует точно такого же объема памяти каждый раз при

запуске программы), но поскольку Python реализован на очень широком диапазоне вычислительных устройств, объем используемой памяти зависит от конкретного случая. Различные версии Python вполне могут реализовать тот же тип данных по-разному. Для экономии, описывая значения, особенно зависящие от машины, мы используем термин *типичные*. Анализ использования памяти несколько отличается от анализа использования времени, прежде всего потому, что одним из наиболее значительных средств Python является собственная система распределения памяти, освобождающая нас, как предполагается, от необходимости волноваться о памяти. Конечно, вам правильно советуют использовать это средство в своих интересах. Но это ваша ответственность знать, по крайней мере приблизительно, когда требуемые программой объемы памяти смогут помешать решению заданной задачи.

Машинная память делится на *байты*, где каждый байт состоит из 8 битов и где каждый *бит* — это один двоичный знак. Для определения количества памяти, используемой программой Python, достаточно подсчитать количество используемых программой объектов и умножить их на количество байтов, требующихся для них согласно типу. Для применения этого подхода следует знать количество байтов, использованных объектом каждого конкретного типа. Для определения общего объема используемой объектом памяти к объему памяти, используемой каждой из его переменных экземпляра, следует добавить служебные затраты, связанные с каждым объектом.

Язык Python не определяет размеры встроенных типов данных (`int`, `float`, `bool`, `str` и `list`); размеры объектов этих типов зависят от конкретной системы. Соответственно размеры создаваемых вами типов данных также будут зависеть от конкретной системы, поскольку они основаны на этих встроенных типах данных. Вызов функции `sys.getsizeof(x)` возвращает количество байтов, занимаемых объектом `x` встроенного типа на вашей системе. Приводимые в этом разделе значения являются результатом наблюдений, сделанных с помощью этой функции в интерактивном режиме Python на одной из типичных систем. Можете сами сделать это на своем компьютере!

Предупреждение пользователям Python 3. Значения в этом разделе приведены для типичных систем Python 2. Python 3 использует более сложную модель памяти. Например, объем памяти, занимаемой объектом `0` типа `int`, не совпадает с таковым у объекта `1` типа `int`.

Целые числа. Для представления объекта типа `int` со значением в диапазоне (от -2^{63} до $2^{63}-1$) Python использует 16 байтов служебной информации и 8 байтов (т.е. 64 бита) для числового значения. Так, например, для представления объекта типа `int` с нулевым значением Python использует 24 байта, для представления объекта типа `int` со значением 1234 — 24 байта и т.д. В большинстве приложений

мы не имеем дела с огромными целыми числами вне этого диапазона, поэтому для каждого целого числа используется 24 байта. Для целых чисел вне этого диапазона Python переходит на иное внутреннее представление, использующее память, пропорциональную количеству цифр в целом числе, как в случае со строками (см. ниже).

Вещественные числа. Для представления объекта типа `float` Python использует 16 байтов служебной информации и 8 байтов для числового значения (т.е. мантисса, экспонента и знак), независимо от значения объекта. Таким образом, объект типа `float` всегда использует 24 байта.

Логические переменные. В принципе, Python мог бы представить логическое значение, используя один бит машинной памяти. На практике Python представляет логические значения как целые числа. А именно: Python использует 24 байта для представления объекта типа `bool` со значением `True`, равно как и со значением `False`. Это в 192 раза больше, чем минимально необходимо! Но эта расточительность частично компенсируется, так как Python "кеширует" два логических объекта (см. далее).

```
>>> import sys
>>> size = sys.getsizeof

>>> size(0)
24
>>> size(123456789)
24
>>> size(12345678901234567890)
36

>>> size(1.0)
24
>>> size(6.67e-11)
24

>>> size(True)
24
>>> size(False)
24

>>> size('abc')
43
>>> size('abcdefghijklmnopqr')
58
```

int

float

bool

str

Типичные требуемые объемы и конфигурации памяти встроенных типов

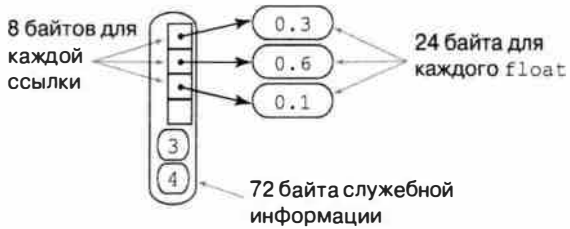
Кеширование. Для экономии памяти Python создает только одну копию объектов с определенными значениями. Например, Python создает только один объект типа `bool` со значением `True` и только один со значением `False`. Таким образом, любые логические переменные содержат лишь ссылку на один из этих двух объектов. Подобная методика *кеширования* возможна потому, что тип данных

`bool` неизменен. На типичных системах Python также кеширует малые значения типа `int` (от –5 до 256), поскольку они используются программистами чаще всего. Объекты типа `float` Python обычно не кеширует.

Кеширование объектов типа `bool`

Строки. Для представления объекта типа `str` Python использует 40 байтов служебной информации (включая длину строки) плюс один байт для каждого символа строки. Представляя строку `'abc'`, например, Python использует 40 + 3 = 43 байта, а представляя строку `'abcdefghijklmnopqr'` — 40 + 18 = 58 байтов. Как правило, Python кеширует только строковые литералы и односимвольные строки.

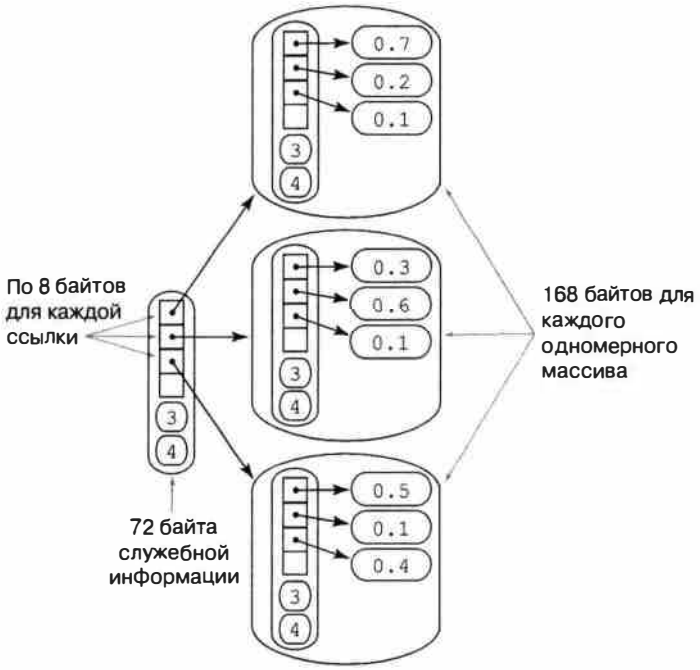
Массивы (списки Python). Для представления массива Python использует 72 байта служебной информации (включая длину массива) плюс 8 байтов для каждой ссылки на объект (по одной для каждого элемента в массиве). Так, например, представление Python массива `[0.3, 0.6, 0.1]` использует 72 + 8*3 = 96 байтов. Сюда не включена память для объектов, только ссылки массива; таким образом, общая используемая память для массива `[0.3, 0.6, 0.1]` составит 96 + 3 * 24 = 168 байтов. В общем, объем памяти для массива из n целых или вещественных чисел составляет $72 + 32n$ байтов. Это общее количество, вероятно, будет недооценено, поскольку структура данных массива переменного размера, то используемая Python для реализации массивов способна использовать дополнительные n байтов для резервирования пространства.



Использование памяти массивом `[0.3, 0.6, 0.1]`

Двумерные массивы и массивы объектов. Двумерный массив — это массив массивов, поэтому информации из предыдущего абзаца вполне достаточно для вычисления объема памяти, занимаемой двумерным массивом из m рядов и n столбцов. Поскольку каждый ряд — это массив, занимающий $72 + 32n$ байтов, общая сумма составит 72 байта служебной информации плюс $8m$ байт ссылок на ряды плюс $m(72 + 32n)$ байтов памяти для m рядов. В итоге получается $72 + 80m + 32mn$ байт. Та же логика работает с массивами объектов любого типа: если объект использует x байтов, массив из m таких объектов использует в общей сложности $72 + m(x + 8)$ байтов. Это значение также, вероятно, будет

несколько недооценено из-за структуры данных переменного размера, используемой Python для представления массива. *Примечание*: функция Python `sys.getsizeof(x)` не особенно поможет для этих целей, поскольку она не вычисляет память самих объектов, а возвращает $72 + 8m$ для любого массива длиной m (или любого двумерного массива на m рядов).



Использование памяти двумерным массивом

Объекты. Ключевой вопрос программирования Python таков: сколько памяти необходимо для представления пользовательского объекта? Ответ на этот вопрос может вас удивить, но знать его нужно: *сотни байт, по крайней мере.* А именно: Python использует 72 байта служебной информации *плюс* 280 байт для словаря, связывающего переменные экземпляра с объектами (словари мы обсудим в разделе 4.4), *плюс* по 24 байта для ссылок на каждую переменную экземпляра, *плюс* память для самих переменных экземпляра. Например, для представления объекта типа `Charge` Python использует по крайней мере $72 + 280 = 352$ байта для служебной информации, $8 * 3 = 24$ байта для хранения ссылок на значения трех его переменных экземпляра, 24 байта для хранения объекта типа `float`, на который ссылается переменная экземпляра `_rx`, 24 байта для хранения объекта типа `float`, на который ссылается переменная экземпляра `_ry`, и 24 байта для хранения объекта типа `float`, на который ссылается переменная экземпляра `_q`, что в общем итоге дает по крайней мере 448 байтов. В вашей системе

общее количество могло бы быть даже больше, поскольку некоторые реализации используют больше служебной информации.

Использование памяти объектом типа `Charge`

Эти фундаментальные механизмы эффективны для оценки использования памяти очень многими программами, но существует множество факторов, способных существенно усложнить эту задачу. Например, мы уже обращали внимание на потенциальный эффект кеширования. Кроме того, когда задействованы вызовы функций, использование памяти становится сложным динамическим процессом, поскольку более важную роль начинает играть системный механизм распределения памяти. Например, когда ваша программа вызывает функцию, система резервирует необходимую для функции память (для ее локальных переменных) в специальной области, называемой *стеком* (stack), а когда функция возвращает управление вызывающей стороне, эта память возвращается в стек. Поэтому создание массивов и других больших объектов в рекурсивных функциях опасно, так как каждый рекурсивный вызов подразумевает существенный расход памяти. Когда вы создаете объект, система резервирует необходимую для него память в специальной области — *распределяемой памяти* (heap). Каждый объект существует до тех пор, пока есть ссылки на него, а когда их не остается, запускается системный процесс *сбора мусора* (garbage collection), освобождающий занятую объектом область распределяемой памяти. Такая динамика способна существенно упростить задачу точной оценки используемой памяти.

Типичные объемы требуемой памяти

Тип	Байты
Логический	24
Целое число	24
Вещественное число	24
Ссылка	24
Строка длиной n	$40 + n$
Массив на n логических переменных	$72 + 8n$
Массив на n вещественных чисел	$72 + 32n$
Массив n на n вещественных чисел	$\sim 32mn$

Несмотря на все эти недостатки, каждый программист Python должен понимать, что *любой объект пользовательского типа использует большой объем памяти*. Таким образом, программа Python, определяющая большое количество объектов пользовательского типа, может использовать намного больше пространства (и времени), чем вы могли бы ожидать. Это особенно верно, если каждый объект содержит лишь несколько переменных экземпляра — в этом случае соотношение памяти, занятой служебной информацией и переменными экземпляра, очень высоко. Например, если вы сравните производительность нашего типа `Complex` и встроенного типа Python `complex` в программе `mandelbrot.py`, то разница, конечно, будет заметна. Множество объектно-ориентированных языков программирования появилось и исчезло с момента появления этой концепции несколько десятилетий назад, и большинство из их в конечном счете переходило к *облегченным* объектам для пользовательских типов. Python с этой целью предоставляет две дополнительные возможности (*именованные кортежи* и *слоты*), но в этой книге мы не будем их использовать.

Перспектива. Хорошая производительность важна. До невозможности медленная программа почти так же бесполезна, как неработающая, поэтому следует *обращать внимание на стоимость* вначале, чтобы получить некоторое представление об осуществимости решения задачи. В частности, всегда разумно получить некоторое представление о коде, составляющем внутренний цикл ваших программ.

Возможно, наиболее распространенной ошибкой программирования является слишком пристальное внимание к характеристикам производительности. Ваш главный приоритет — сделать свой код правильным и понятным. Изменение программы с единственной целью ускорения ее работы лучше оставить экспертам. На самом деле это зачастую оказывается не лучшей идеей, поскольку в результате получается сложный и трудный для понимания код. Чарльз Энтони Ричард Хоар (C. A. R. Hoare) (разработчик алгоритма быстрой сортировки и яркий сторонник концепции четкого, ясного и правильного кода) когда-то резюмировал эту идею так: "*преждевременная оптимизация — источник всех зол*", а Кнут добавил "*(или, по крайней мере, большинства из них) в программировании*". Кроме того, улучшение продолжительности выполнения мало чего стоит, если текущая продолжительность незначительна. Например, улучшение продолжительности выполнения программы в 10 раз несущественно, если раньше она выполнялась секунду. Даже когда выполнение программы занимает некоторое время, скажем, минуты, полное время, необходимое для реализации и отладки улучшенного алгоритма, может оказаться существенно большим, чем время, необходимое для работы немного более медленного, поэтому в данном случае можно позволить компьютеру делать свою работу. Хуже, когда вы тратите

значительные усилия и время на реализацию идеи, которая фактически никак не делает программу существенно быстрее.

Возможно, вторая наиболее распространенная ошибка в разработке алгоритма — это игнорирование характеристик производительности. Более быстрые алгоритмы зачастую куда сложней, чем простые решения "в лоб", поэтому вы могли бы применить более медленный алгоритм, чтобы избежать необходимости иметь дело с более сложным кодом. Но иногда лишь нескольких строк хорошего кода достаточно для получения огромной экономии. Пользователи на удивление большого количества компьютерных систем теряют существенное время на ожидание завершения простых квадратичных алгоритмов, хотя для решения этих задач существуют линейные или линейно-логарифмические алгоритмы, которые, будучи лишь ненамного сложнее, способны решить ту же задачу куда быстрее. Когда мы имеем дело с задачами огромных размеров, у нас зачастую нет другого выбора, кроме поиска лучших алгоритмов.

Улучшайте программы, делая их понятней, эффективней и изящней, пусть это будет вашей задачей при разработке каждой программы. Если вы постоянно *обращаете внимание на стоимость* программы при ее разработке, то будете получать выигрыш при каждом ее запуске.

Вопросы и ответы

В тексте упоминалось, что операции с очень большими целыми числами могут потребовать больше, чем постоянное время. Нельзя ли поточнее?

Нет, действительно. Определение "очень большой" зависит от конкретной системы. В большинстве практических случаев вы можете полагать, что постоянное время выполнения относится к операциям с 32- или 64-битовыми целыми числами. Современные криптографические приложения задействуют огромные числа с сотнями и даже тысячами цифр.

Как узнать, насколько долго выполняется сложение или умножение двух вещественных чисел на моем компьютере?

Проведите эксперимент! Программа `timeops.py`, выложенная на сайте книги, использует тип `Stopwatch` для проверки времени выполнения различных арифметических операций с целыми и вещественными числами. Эта методика измеряет фактическое прошедшее время, как будто по часам на стене. Если в вашей системе запущено не много других приложений, то она может дать довольно точные результаты. Для измерения продолжительности небольших фрагментов кода Python предоставляет также модуль `timeit`.

Есть ли способ провести измерение по процессорному времени, а не по системным часам?

В некоторых системах вызов функции `time.clock()` возвращает текущее процессорное время, выраженное в секундах, как значение типа `float`. В случае доступности, вы можете заменить `time.time()` на `time.clock()` при определении эффективности программ Python.

Сколько времени занимают такие функции, как `math.sqrt()`, `math.log()` и `math.sin()`?

Проведите эксперимент! Тип `Stopwatch` облегчает составление таких программ, как `timeops.py`, позволяющих непосредственно отвечать на подобные вопросы. Вы будете использовать свой компьютер намного эффективней, если возьмете себе в привычку делать это.

Почему резервирование массива (списка Python) размером *n* занимает время, пропорциональное *n*?

Python инициализирует все элементы массива любым определенным программистом значением. Таким образом, в Python нет способа зарезервировать память для массива без присвоения ссылки на объект каждому его элементу.

Присвоение объектной ссылки каждому элементу массива размером n занимает время, пропорциональное n.

Как узнать объем памяти, доступной для моих программ Python?

Поскольку Python передает сообщение `MemoryError` при выходе за пределы памяти, не трудно провести несколько экспериментов. Например, используйте эту программу (`bigarray.py`):

```
import sys
import stdarray
import stdio

n = int(sys.argv[1])
a = stdarray.create1D(n, 0)
stdio.writeln('finished')
```

и запустите ее так:

```
% python bigarray.py 100000000
finished
```

Это показывает, что есть место для 100 миллионов целых чисел. Но если запустить ее так:

```
% python bigarray.py 1000000000
```

Python зависнет, откажет или передаст сообщение об ошибке времени выполнения, и вы сможете заключить, что пространства для массива в 1 миллиард целых чисел нет.

Что означает, когда говорят, что продолжительность самого плохого случая алгоритма составляет $O(n2)$?

Это пример записи, известной как *нотация "большого O"* (big-O notation). Мы пишем: $f(n)$ равна $O(g(n))$, если существуют константы c и n_0, для которых $f(n) \leq c$ $g(n)$ для всех $n > n_0$. Другими словами, функция f(n) ограничивается сверху $g(n)$, до постоянных коэффициентов и для достаточно больших значений n. Например, функция $30n^2 + 10n + 7$ равна $O(n^2)$. Мы говорим, что продолжительность самого плохого случая алгоритма составляет $O(g(n))$, если продолжительность выполнения является функцией $O(g(n))$ от размера ввода n для всех возможных входных данных. Эта запись широко используется теоретическими программистами для доказательства теоремы об алгоритмах, поэтому вы гарантированно встретите ее, проходя курс алгоритмов и структур данных. Она представляет гарантию производительности самого плохого случая.

Таким образом, я могу использовать тот факт, что продолжительность самого плохого случая алгоритма составляет $O(n3)$ или $O(n2)$, чтобы прогнозировать производительность?

Нет, поскольку фактическая продолжительность могла бы быть намного меньше. Например, функция $30n^2 + 10n + 7$ составляет $O(n^2)$, но в то же время $O(n^3)$ и $O(n^{10})$, поскольку нотация большого O представляет только верхнюю границу продолжительности самого плохого случая. Кроме того, даже если есть некоторое семейство входных данных, для которых продолжительность пропорциональна данной функции, на практике эти входные данные могут и не встретиться. Следовательно, вы не можете использовать нотацию большого O, чтобы спрогнозировать производительность. Применяемые нами запись с использованием тильды и классификация порядка роста точнее, чем нотация большого O, поскольку они представляют верхние и нижние границы роста функции. Многие программисты неправильно используют нотацию большого O для обозначения соответственно верхних и нижних границ.

Сколько памяти Python обычно использует для хранения кортежа из n элементов?

$56 + 8n$ байт, а также вся память, необходимая для самих объектов. Это немного меньше, чем для массивов, поскольку Python может реализовать кортеж (на машинном уровне), используя обычный массив вместо массива переменного размера.

- **Почему Python использует так много памяти (280 байтов) для хранения словаря, сопоставляющего переменные экземпляра объекта с его значениями?**
- В принципе, у различных объектов того же типа данных могут быть разные переменные экземпляра. В данном случае Python нуждался в некотором способе сопоставления произвольного количества возможных переменных экземпляра для каждого объекта. Но большинству кода Python это не нужно (наш стиль в данной книге таков, что мы в этом не нуждаемся).

Упражнения

4.1.1. Реализуйте функцию `writeAllTriples()` для программы `threesum.py`, выводящую все триплеты, сумма которых равна нулю.

4.1.2. Модифицируйте программу `threesum.py` так, чтобы она получала в аргументе командной строки значение `x` и находила среди поступающих на стандартный ввод чисел триплет, сумма которого ближе всех к `x`.

4.1.3. Составьте программу `foursum.py`, читающую со стандартного ввода целое число `n`, затем `n` целых чисел и подсчитывающую количество отдельных кортежей по 4, сумма членов которых равна нулю. Используйте четыре вложенных цикла. Каков порядок роста продолжительности выполнения вашей программы? Оцените наибольшее `n`, которое ваша программа способна обработать за час. Затем запустите свою программу для подтверждения гипотезы.

4.1.4. Докажите, что $1 + 2 + \ldots + n = n(n + 1) / 2$.

Решение. Мы доказали это индукцией в начале раздела 2.3. Вот основание для другого доказательства:

```
   1  +  2  + ... + n-1 +  n
+  n  + n-1 + ... +  2  +  1
  ------------------------------
  n+1 + n+1 + ... + n+1 + n+1
```

4.1.5. Докажите индукцией, что количество отдельных триплетов целых чисел от 0 до $n - 1$ равно $n(n - 1)(n - 2) / 6$.

Решение. Формула верна для $n = 2$. Для $n > 2$ подсчет всех триплетов, не включая n, составляет $(n - 1)(n - 2)(n - 3) / 6$ в соответствии с индуктивной гипотезой, а все триплеты, включающие $n - 1$, составляющие $(n - 1)(n - 2) / 2$, дают общее количество

$$(n - 1)(n - 2)(n - 3) / 6 + (n - 1)(n - 2) / 2 = n(n - 1)(n - 2) / 6.$$

4.1.6. Покажите на приближении с интегралами, что количество отдельных триплетов целых чисел от 0 до $n - 1$ составляет $n^3 / 6$.

Решение. $\sum_0^n \sum_0^i \sum_0^j 1 \approx \int_0^n \int_0^i \int_0^j dk dj di = \int_0^n \int_0^i j dj di = \int_0^n (i^2 / 2) di = n^3 / 6.$

4.1.7. Каково значение `x` (как функции от n) после выполнения следующего фрагмента кода?

```
x = 0
for i in range(n):
    for j in range(i+1, n):
        for k in range(j+1, n):
            x += 1
```

Решение. $n (n - 1) (n - 2) / 6$.

4.1.8. Примените запись с использованием тильды для упрощения каждой из следующих формул, а также укажите порядок роста каждой:

a. $n\ (n-1)\ (n-2)\ (n-3)\ /\ 24$

b. $(n-2)\ (\lg n-2)(\lg n+2)$

c. $n(n+1)-n^2$

d. $n(n+1)\ /\ 2+n \lg n$

e. $\ln((n-1)(n-2)(n-3))^2$

4.1.9. Является ли следующий фрагмент кода линейным, квадратичным или кубическим (как функция от n)?

```
for i in range(n):
    for j in range(n):
        if i == j: c[i][j] = 1.0
        else: c[i][j] = 0.0
```

4.1.10. Предположим, продолжительность выполнения алгоритма для входных данных размером 1 000, 2 000, 3 000 и 4 000 составляет 5, 20, 45 и 80 секунд соответственно. Оцените время решения задачи размером 5 000. Является ли алгоритм линейным, линейно-логарифмическим, квадратичным, кубическим или экспоненциальным?

4.1.11. Какой алгоритм предпочесть: квадратичный, линейно-логарифмический или линейный?

Решение. Весьма заманчиво быстро принять решение на основании порядка роста, но так очень просто ошибиться. Необходимо иметь некоторое представление о сложности задачи и значении ведущих коэффициентов продолжительности. Предположим, например, что продолжительность выполнения составляет n^2 секунд, $100\ n \log_2 n$ секунд и $10\ 000n$ секунд. Квадратичный алгоритм будет самым быстрым для n примерно до 1 000, а линейный алгоритм никогда не будет быстрее линейно-логарифмического (n должно было быть больше 2^{100} — слишком много, чтобы имело смысл рассматривать).

4.1.12. Примените научный метод для разработки и проверки гипотезы о порядке роста продолжительности выполнения следующего фрагмента кода, как функции от аргумента n:

```
def f(n):
    if (n == 0): return 1
    return f(n-1)+f(n-1)
```

4.1.13. Примените научный метод для разработки и проверки гипотезы о порядке роста продолжительности выполнения каждого из следующих фрагментов кода как функции от `n`:

```
s = ''
for i in range(n):
    if stdrandom.bernoulli(0.5): s += '0'
    else:                        s += '1'

s = ''
for i in range(n):
    oldS = s
    if stdrandom.bernoulli(0.5): s += '0'
    else:                        s += '1'
```

Решение. На многих системах первое линейно; второе является квадратичным. У вас нет никакого способа узнать почему: в первом случае Python обнаруживает, что `s` — единственная переменная, ссылающаяся на строку, поэтому она добавляет каждый символ в строку, как в список (за амортизированное постоянное время), *даже при том, что строка неизменна*! Более безопасная альтернатива подразумевает создание списка, содержащего символы, и их конкатенацию при вызове метода `join()`.

```
a = []
for i in range(n):
    if stdrandom.bernoulli(0.5): a += ['0']
    else:                        a += ['1']
s = ''.join(a)
```

4.1.14. Каждая из четырех функций Python ниже возвращает строку длиной `n` со всеми символами `x`. Определите порядок роста продолжительности выполнения каждой функции. Помните, что конкатенация двух строк в Python занимает время, пропорциональное сумме их длин.

```
def f1(n):
    if (n == 0):
        return ''
    temp = f1(n // 2)
    if (n % 2 == 0): return temp + temp
    else:            return temp + temp + 'x'

def f2(n):
    s = ''
    for i in range(n):
        s += 'x'
```

```
    return s

def f3(n):
    if (n == 0): return ''
    if (n == 1): return 'x'
    return f3(n//2) + f3(n - n//2)

def f4(n):
    temp = stdarray.create1D(n, 'x')
    return ''.join(temp)

def f5(n):
    return 'x' * n
```

4.1.15. Каждая из трех функций Python, приведенных ниже, возвращает строку длиной n в обратном порядке. Каков порядок роста продолжительности выполнения каждой функции?

```
def reverse1(s):
    n = len(s)
    reverse = ''
    for i in range(n):
        reverse = s[i] + reverse
    return reverse

def reverse2(s):
    n = len(s)
    if (n <= 1):
        return s
    left = s[0 : n//2]
    right = s[n//2 : n]
    return reverse2(right) + reverse2(left)

def reverse3(s):
    return s[::-1]
```

Выражение `s[::-1]` использует необязательный третий аргумент для определения размера шага.

4.1.16. Следующий фрагмент кода (взятый из книги по программированию на Java) создает случайную перестановку целых чисел от 0 до $n-1$. Определите порядок роста продолжительности его выполнения как функции от n. Сравните его с порядком роста кода перетасовки в разделе 1.4.

```
a = stdarray.create1D(n, 0)
taken = stdarray.create1D(n, False)
```

```
count = 0
while (count < n):
    r = stdrandom.uniform(0, n)
    if not taken[r]:
        a[r] = count
        taken[r] = True
        count += 1
```

4.1.17. Как долго следующий фрагмент кода выполняет первый оператор `if` в трижды вложенном цикле?

```
for i in range(n):
    for j in range(n):
        for k in range(n):
            if (i < j) and (j < k):
                if a[i]+a[j]+a[k] == 0:
                    count += 1
```

Примените запись с использованием тильды для упрощения ответа.

4.1.18. Примените научный метод для разработки и проверки гипотезы о порядке роста продолжительности выполнения метода `collect()` в программе 2.1.3 (`coupon.py`) как функции от аргумента `n`. *Примечание*: удвоение не эффективно для сравнения линейных и линейно-логарифмических гипотез — попробуйте увеличить размер ввода вчетверо.

4.1.19. Примените научный метод для разработки и проверки гипотезы о порядке роста продолжительности программы 1.6.3 (`markov.py`) как функции от аргументов `moves` и `n`.

4.1.20. Составьте программу `mooreslaw.py`, получающую аргумент командной строки `n` и выводящую увеличение скорости процессора за декаду, если она удваивается каждые n месяцев. Насколько увеличится скорость процессора за следующее десятилетие, если скорости удваиваются каждые $n = 15$ месяцев или $n = 24$ месяца?

4.1.21. Используя модель памяти из текста, укажите требуемые объемы памяти для каждого объекта следующих типов данных из главы 3:

a. `Stopwatch`

b. `Turtle`

c. `Vector`

d. `Body`

e. `Universe`

4.1.22. Оцените как функцию от размера таблицы n объем памяти, используемый программой 2.4.4 (`visualizev.py`) и обнаружение вертикального просачивания (программа 2.4.2). *Дополнительное задание.* Ответьте на тот же вопрос для случая использования рекурсивного метода обнаружения просачивания в программе 2.4.6 (`percolation.py`).

4.1.23. Оцените размер наибольшего массива n на n целых чисел, который может содержать ваш компьютер, а затем попытайтесь создать его.

4.1.24. Оцените как функцию от количества документов n и размерности d объем пространства, используемого программой 3.3.5 (`comparedocuments.py`).

4.1.25. Составьте версию программы 1.4.3 (`primesieve.py`), использующую массив целых чисел вместо массива логических переменных и 32 бита для каждого целого числа. Определите наибольшее значение `n`, которое может быть обработано при коэффициенте 32.

4.1.26. В следующей таблице приведена продолжительность различных программ при разных значениях n. Заполните пробелы своими оценками, выработанными на основании предоставленной информации.

Программа	1 000	10 000	100 000	1 000 000
A	0,001 секунды	0,012 секунды	0,16 секунды	? секунды
B	1 минута	10 минут	1,7 часа	? часов
C	1 секунда	1,7 минуты	2,8 часа	? дней

Предоставьте гипотезы для порядка роста продолжительности каждой программы.

Практические упражнения

4.1.27. *Анализ с тремя суммами.* Вычислите вероятность того, что ни один из n триплетов случайных 32-разрядных целых чисел не даст в сумме 0, а также дайте приблизительную оценку для n, равных 1 000, 2 000 и 4 000. *Дополнительное задание.* Выведите приближенную формулу для ожидаемого количества таких триплетов (как функцию от n) и проведите эксперименты для проверки своей гипотезы.

4.1.28. *Поиск ближайшей пары.* Разработайте квадратичный алгоритм, находящий пару целых чисел, ближайших друг к другу. (В следующем разделе мы рассмотрим линейно-логарифмический алгоритм поиска.)

4.1.29. *Экспоненциальный закон.* Покажите на логарифмическом графике функции cn^b, что у нее есть наклон b и пересечение с x в $\log c$. Каков наклон и пересечение с x для $4\ n^3(\log n)^2$?

4.1.30. *Сумма, наиболее удаленная от нуля.* Разработайте алгоритм поиска пары целых чисел, сумма которых дальше всех от нуля. Возможен ли алгоритм линейного времени?

4.1.31. *Удаление символов.* Популярный веб-сервер предоставляет функцию `no2slash()`, сворачивающую повторяющиеся символы /. Например, строка `/d1///d2////d3/test.html` превращается в `/d1/d2/d3/test.html`. Исходный алгоритм последовательно ищет символ / и копирует остаток строки:

```
def no2slash(name):
    for x in range(1, len(name)):
        if x > 0:
            if (name[x-1] == '/') and (name[x] == '/'):
                for y in range(x+1, len(name)):
                    name[y-1] = name[y]
        else:
            x += 1
```

К сожалению, продолжительность выполнения этого кода имеет квадратичную зависимость от количества символов / во вводе. При одновременной посылке нескольких запросов с большими количествами символов / хакер может перегрузить сервер и замедлить другие процессы, отняв слишком много процессорного времени, проведя таким образом атаку отказа в обслуживании. Разработайте версию функции `no2slash()`, имеющую линейное время выполнения и нечувствительную к атакам такого типа.

4.1.32. *Диаграмма Юнга.* Предположим, имеется массив *n* на n целых чисел `a[][]`, где `a[i][j] < a[i+1][j]` и `a[i][j] < a[i][j+1]` для всех `i` и `j`, как в следующей таблице:

```
 5   23   54   67   89
 6   69   73   74   90
10   71   83   84   91
60   73   84   86   92
99   91   92   93   94
```

Разработайте алгоритм проверки принадлежности заданного целого числа *x* к данной диаграмме Юнга с линейным порядком роста от *n*.

4.1.33. *Сумма подмножества.* Составьте программу `anysum.py`, которая получает целое число *n* со стандартного ввода, затем читает со стандартного ввода *n* целых чисел и подсчитывает количество подмножеств, сумма которых

равна 0. Укажите порядок роста продолжительности выполнения вашей программы.

4.1.34. *Прокрутка массива.* Дан массив из n элементов предоставьте алгоритм линейного времени для прокрутки массива на k позиций. Таким образом, если массив содержит $a_0, a_1, \ldots, a_{n-1}$, то прокрученный — $a_k, a_{k+1}, \ldots, a_{n-1}, a_0, \ldots, a_{k-1}$. Используйте самый большой постоянный объем дополнительного пространства (индексы массива и значения массива). *Подсказка*: измените на обратные три подмассива.

4.1.35. *Поиск повторения целого числа.* (a) Дан массив из n целых чисел от 1 до n с одним значением, повторяющимся дважды, и одним отсутствующим. Предоставьте алгоритм поиска недостающего целого числа за линейное время и с постоянным дополнительным пространством. (b) Дан предназначенный только для чтения массив из n целых чисел со значениями от 1 до $n-1$, встречающихся только однажды, кроме одного двойного. Предоставьте алгоритм поиска повторяющегося значения за линейное время и при постоянном дополнительном пространстве. (c) Дан предназначенный только для чтения массив из n целых чисел со значениями от 1 до $n-1$, предоставьте алгоритм поиска повторяющихся значений за линейное время и при постоянном дополнительном пространстве.

4.1.36. *Факториал.* Разработайте быстрый алгоритм вычисления $n!$ для больших значений n. Используйте свою программу для вычисления самой длинной непрерывной последовательности девяток в 1000000! Разработайте и проверьте гипотезу для порядка роста продолжительности вашего алгоритма.

4.1.37. *Максимальная сумма.* Разработайте алгоритм линейного времени, который находит непрерывное подмножество не меньше m размером в последовательности из n целых чисел и возвращает самую большую сумму из всех таких подмножеств. Реализуйте свой алгоритм и подтвердите, что порядок роста его продолжительности линеен.

4.1.38. *Соответствие шаблону.* Дан массив n на n черных (1) и белых (0) пикселей, разработайте линейный алгоритм поиска наибольшего квадратного подмассива, полностью состоящего из черных пикселей. Например, следующий массив 8 на 8 содержит подмассив 3 на 3 полностью черных пикселей:

```
1 0 1 1 1 0 0 0
0 0 0 1 0 1 0 0
0 0 1 1 1 0 0 0
```

```
0 0 1 1 1 0 1 0
0 0 1 1 1 1 1 1
0 1 0 1 1 1 1 0
0 1 0 1 1 0 1 0
0 0 0 1 1 1 1 0
```

Реализуйте свой алгоритм и подтвердите, что порядок роста его продолжительности линеен от количества пикселей. *Дополнительное задание.* Разработайте алгоритм поиска наибольшего *прямоугольного* подмассива черных пикселей.

4.1.39. *Максимальное среднее.* Составьте программу поиска в массиве из n целых чисел непрерывного подмассива по крайней мере из m элементов, обладающего самым высоким средним значением из всех таких подмассивов. Используйте научный метод для подтверждения гипотезы о том, что порядок роста продолжительности вашей программы составляет mn^2. Затем составьте программу решения этой задачи предварительным вычислением `prefix[i] = a[0] + ... + a[i]` для каждого `i` и последующим вычислением среднего в интервале от `a[i]` до `a[j]` при помощи выражения `(prefix[j] - prefix[i]) / (j - i + 1)`. Используйте научный метод для подтверждения того, что этот метод сокращает порядок роста в n раз.

4.1.40. *Субэкспоненциальная функция.* Найдите функцию с порядком роста, большим, чем у любой полиноминальной функции, но меньшим, чем у любой экспоненциальной. *Дополнительное задание.* Составьте программу, у продолжительности выполнения которой тот же порядок роста.

4.1.41. *Массивы переменного размера.* Для каждой из следующих стратегий укажите, занимает ли любая операция с массивом переменного размера постоянное амортизируемое время, или найдите последовательность из n операций (начинающуюся с пустой структуры данных), которая занимает квадратичное время.

a. Удвоение емкости массива переменного размера, когда он заполнен, и уменьшение емкости вдвое, когда он наполовину пуст.

b. Удвоение емкости массива переменного размера, когда он заполнен, и уменьшение емкости вдвое, когда он заполнен на одну треть.

c. Увеличение емкости массива переменного размера в 9/8 раз, когда он полон, и уменьшение в 9/8 раз, когда он полон на 80%.

4.2. Сортировка и поиск

Программы этого раздела...

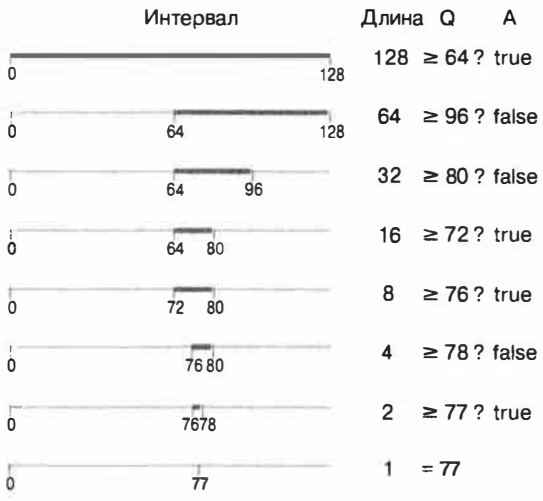
Программа 4.2.1. Бинарный поиск (20 вопросов) (questions.py) 544
Программа 4.2.2. Дихотомический поиск (bisection.py) 548
Программа 4.2.3. Бинарный поиск (в отсортированном массиве) (binarysearch.py) 551
Программа 4.2.4. Сортировка вставкой (insertion.py) 555
Программа 4.2.5. Проверка сортировки удвоением (timesort.py) 557
Программа 4.2.6. Сортировка с объединением (merge.py) 560
Программа 4.2.7. Подсчет частот (frequencycount.py) 563

Задача сортировки подразумевает перестройку элементов массива в порядке возрастания. Это известная и критически важная задача во многих вычислительных приложениях: музыка в вашей библиотеке расположена в алфавитном порядке, сообщения электронной почты имеют порядок, обратный времени получения, и т.д. Хранение вещей в некотором порядке является вполне естественным желанием. Это столь полезно потому, что *искать* намного проще в отсортированном списке, а не в не отсортированном. Эта потребность особенно остра в вычислениях с поиском в огромных списках, а эффективный поиск может быть важнейшим фактором при решении задачи.

Сортировка и поиск важны как для коммерческих приложений (фирмы хранят клиентские файлы упорядочено), так и для научных (для организации данных и вычислений), а также во многих областях, имеющих мало общего с хранением, включая сжатие данных, компьютерную графику, вычислительную биологию, числовые вычисления, комбинаторную оптимизацию, криптографию и многое другое.

Мы используем эти фундаментальные задачи для иллюстрации идеи, что *эффективные алгоритмы* — это ключ к эффективным решениям вычислительных задач. Действительно, в настоящее время предложено множество различных алгоритмов сортировки и поиска. Какой из них следует использовать для решения данной задачи? Этот вопрос важен потому, что у разных алгоритмов могут быть существенно разные характеристики производительности, определяющие различие между успехом в практической ситуации и полной невозможностью даже приблизиться к результату, несмотря на применение самого быстродействующего из доступных компьютеров.

В этом разделе мы подробно рассмотрим два классических алгоритма сортировки и поиска, а также несколько приложений, в которых их эффективность играет критически важную роль. На этих примерах вы убедитесь не только в удобстве этих алгоритмов, но и в необходимости *обращать внимание на стоимость* всякий раз при решении задачи, требующей существенного объема вычислений.



Бинарный поиск скрытого числа

Бинарный поиск. Игра "Двадцать вопросов" (см. программу 1.5.2, `twentyquestions.py`) дает важный и полезный урок с точки зрения разработки и использования эффективных алгоритмов для решения вычислительных задач. Постановка игры проста: ваша задача угадать значение скрытого номера (одного из n целых чисел от 0 до $n-1$). Каждый раз, когда вы называете предполагаемое число, вам говорят, угадали вы либо назвали число, слишком большое или слишком маленькое. Как мы обсуждали в разделе 1.5, эффективная стратегия такова: назвать значение в середине интервала, а затем использовать ответ для выбора половины интервала, содержащего секретное число. По причинам, которые станут понятны впоследствии, мы начнем с небольшого изменения игры: вопрос будет задаваться в форме "действительно ли мое число больше или равно m?", ответ будет только *да* или *нет*, а n будет степенями числа 2. Далее, основа эффективного алгоритма, позволяющего всегда получить скрытое число за минимум вопросов (в самом плохом случае), подразумевает получение интервала, содержащего секретное число и его сокращение наполовину при каждом этапе. Если говорить более точно, то мы используем *полуоткрытый интервал* (half-open interval), включающий левую конечную точку, но не правую. Для обозначения всех целых чисел, больших или равных lo и меньших (но не равных) hi, мы используем синтаксис $[lo, hi)$. Мы начинаем с $lo = 0$ и $hi = n$, а затем используем следующую рекурсивную стратегию.

- *Конечный случай.* Если $hi - lo$ равно 1, то секретное число — lo.
- *Рекурсивный этап.* В противном случае спросите, не больше и не равно ли секретное число $mid = (hi + lo) / 2$. Если да, то ищите число в интервале $[mid, hi)$, в противном случае — в интервале $[lo, mid)$.

Эта стратегия — пример общего алгоритма *бинарный поиск* (binary search), имеющего множество применений. Программа 4.2.1 (`questions.py`) является его реализацией.

Программа 4.2.1. Бинарный поиск (20 вопросов) (questions.py)

```
import sys
import stdio

def search(lo, hi):
    if (hi - lo) == 1:
        return lo
    mid = (hi+lo) // 2
    stdio.write('Greater than or equal to '+str(mid)+'? ')
    if stdio.readBool():
        return search(mid, hi)
    else:
        return search(lo, mid)

k = int(sys.argv[1])
n = 2 ** k
stdio.write('Think of a number ')
stdio.writeln('between 0 and '+str(n - 1))
guess = search(0, n)
stdio.writeln('Your number is '+str(guess))
```

lo	Наименьшее возможное целое число
hi - 1	Наибольшее возможное целое число
mid	Середина
n	Количество возможных целых чисел
k	Количество вопросов

Этот сценарий использует бинарный поиск для той же игры, что и программа 1.5.2, но роли теперь поменялись: вы выбираете секретное число, а программа предполагает свое значение. Она получает аргумент командной строки k, просит придумать число от 0 до 2^k-1 и всегда угадывает ответ за k вопросов.

```
% python questions.py 7
Think of a number between 0 and 127
Greater than or equal to 64?  True
Greater than or equal to 96?  False
Greater than or equal to 80?  False
Greater than or equal to 72?  True
Greater than or equal to 76?  True
Greater than or equal to 78?  False
Greater than or equal to 77?  True
Your number is 77
```

Доказательство правильности. Сначала необходимо убедиться в *правильности* стратегии — другими словами, что она всегда ведет к секретному числу. Для этого мы установим следующие факты.

- Интервал всегда содержит секретное число.
- Длины интервала — степени числа 2, уменьшающиеся от 2^k.

Первый из этих фактов подтверждается кодом; второй следует из наблюдения, что если $(hi - lo)$ степень числа 2, то $(hi - lo) / 2$ — следующая меньшая степень числа 2, а также длина обоих половинных интервалов. Эти факты — основа индукционного доказательства, что алгоритм работает как надо. В конечном счете длина интервала становится равной 1, что гарантирует нахождение числа.

Анализ продолжительности выполнения. Предположим, что n — количество возможных значений. В программе `questions.py` $nc = 2^k$, где $k = \lg n$. Теперь предположим, что $T(n)$ — это количество вопросов. Рекурсивная стратегия непосредственно подразумевает, что функция $T(n)$ должна удовлетворять следующему рекуррентному соотношению:

$$T(n) = T(n / 2) + 1.$$

При $T(1) = 0$. Подставив 2^k для n, при телескопической рекуренции (применение к себе самому) мы можем непосредственно получить выражение замкнутой формы:

$$T(2^k) = T(2^{k-1}) + 1 = T(2^{k-2}) + 2 = ... = T(1) + k = k.$$

Снова подставив n для 2^k (и $\lg n$ для k), получаем результат:

$$T(n) = \lg n.$$

Обычно мы используем это уравнение для подтверждения гипотезы о том, что использующая бинарный поиск программа имеет логарифмическую продолжительность. *Примечание*: бинарный поиск и функция `questions.search()` работают, даже когда n не степень числа 2 (см. упр. 4.2.1).

Линейно-логарифмическая пропасть. Альтернатива бинарному поиску — назвать 0, затем 1, 2, 3 и так далее, пока не встретится секретное число. Это алгоритм решения "в лоб": он решает задачу, но не обращает никакого внимания на стоимость (что фактически не позволит решить с его помощью достаточно большую задачу). В данном случае продолжительность выполнения алгоритма зависит от секретного числа и в самом плохом случае может занять столько же времени, сколько и n вопросов. Бинарный поиск, напротив, гарантирует нам не более $\lg n$ вопросов (точнее, $\lceil \lg n \rceil$, если n не степень числа 2). Как вы узнаете, различие между n и $\lg n$ имеет огромное значение в практических случаях. *Понимание чудовищности величины этого различия критически важно для понимания важности проектирования и анализа алгоритмов.* В данном контексте предположим, что обработка вопроса занимает 1 секунду. При бинарном поиске вы можете угадать

значение секретного числа из менее чем 1 миллиона возможных за 20 секунд; при алгоритме решения "в лоб" мог бы потребоваться 1 миллион секунд, что больше 1 недели. Мы увидим много примеров тому, что такое различие в стоимости — главный фактор при определении возможности решения практической задачи.

Двоичное представление. Если вернуться к программе 1.3.7 (`binary.py`), то можно сразу заметить, что бинарный поиск — почти то же преобразование числа в двоичный формат! Каждый вопрос определяет один бит ответа. В нашем примере информация — это, скажем, число от 0 до 127, количество битов которого в двоичном представлении равно 7. Ответ на первый вопрос (число больше или равно 64?) дает нам значение первого бита, ответ на второй вопрос — значение следующего бита и т.д. Например, если задумано число 77, последовательность ответов `True False False True True False True` немедленно дает 1001101 — двоичное представление числа 77.

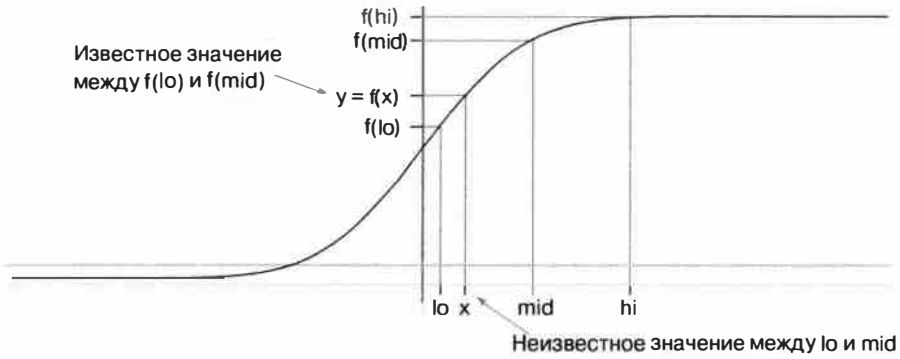
Размышление в терминах двоичного представления — это еще один способ понять суть линейно-логарифмической пропасти: когда имеется программа, продолжительность выполнения которой линейно зависит от параметра n, ее продолжительность выполнения пропорциональна значению n, тогда как логарифмическая продолжительность пропорциональна только количеству *цифр* в n. В контексте, вероятно, немного более знакомом вам, рассмотрим следующий вопрос, иллюстрирующий ту же точку зрения: вы хотели бы заработать 6 долларов или шестизначную сумму?

Инверсия функции. В качестве примера удобства бинарного поиска для научного вычисления вернемся к задаче, рассматривавшейся в разделе 2.1: инверсии возрастающей функции (см. упр. 2.1.26). Дана возрастающая функция f, значение y и открытый интервал (lo, hi), наша задача — найти значение x в пределах интервала таким образом, чтобы $f(x) = y$. Хотя в этой ситуации как конечные точки нашего интервала мы используем вещественные числа, а не целые; лежащий в основе подход остается тем же, что и при поиске скрытого целого числа в игре "Двадцать вопросов": на каждом этапе мы делим длину интервала на два, оставляя x в одной из половин интервала до тех пор, пока интервал не станет достаточно мал, чтобы мы могли сказать, что значение x находится в пределах желаемой точности d, которую мы передаем как аргумент функции. Первый этап показан на рисунке, приведенном ниже.

Эту стратегию реализует программа 4.2.2 (`bisection.py`). Мы начинаем с интервала (lo, hi), как известно, заранее содержащего x, а затем используем следующую рекурсивную процедуру.

- Вычислить $mid = (hi + lo) / 2$.
- *Конечный случай.* Если $hi - lo$ меньше δ, то возвратить mid как искомый x.

- *Рекурсивный этап.* В противном случае проверить, не больше ли $f(mid)$, чем y. Если да, то искать x в интервале (lo, mid); в противном случае — в интервале (mid, hi).



Бинарный поиск (деление пополам) при инверсии возрастающей функции (один этап)

Ключевой момент этого подхода в возрастании функции — для любых значений a и b известно, что если $f(a) < f(b)$, то $a < b$, и наоборот. Рекурсивный этап лишь применяет это знание: если $y = f(x) < f(mid)$, то $x < mid$, следовательно, x должен находиться в интервале (lo, mid), а если $y = f(x) > f(mid)$, то $x > mid$, а следовательно, x находится в интервале (mid, hi). Задачу можно свести к выяснению, какой из $n = (hi - lo) / \delta$ крошечных интервалов длиной δ внутри интервала (lo, hi) содержит x при логарифмической зависимости продолжительности от n. Подобно преобразованию целого числа в двоичное, определяем по одному биту x для каждой итерации. В этом контексте бинарный поиск зачастую называют *методом половинного деления* (bisection search), поскольку мы делим интервал пополам на каждом этапе.

Взвешивание. Бинарный поиск был известен издавна, возможно, частично из-за следующего применения. Предположим, необходимо взвесить некий объект, используя равноплечные балансирные весы. При бинарном поиске это можно сделать с помощью набора гирь с весами, равными степеням числа 2 (причем только по одной гире каждого веса). Поместите объект на правую чашу весов, а на левую — гири в порядке убывания. Если весы склоняются влево, уберите гирю, в противном случае оставьте ее. Этот процесс в точности похож на определение двоичного представления числа при вычитании уменьшающихся степеней числа 2, как в программе 1.3.7.

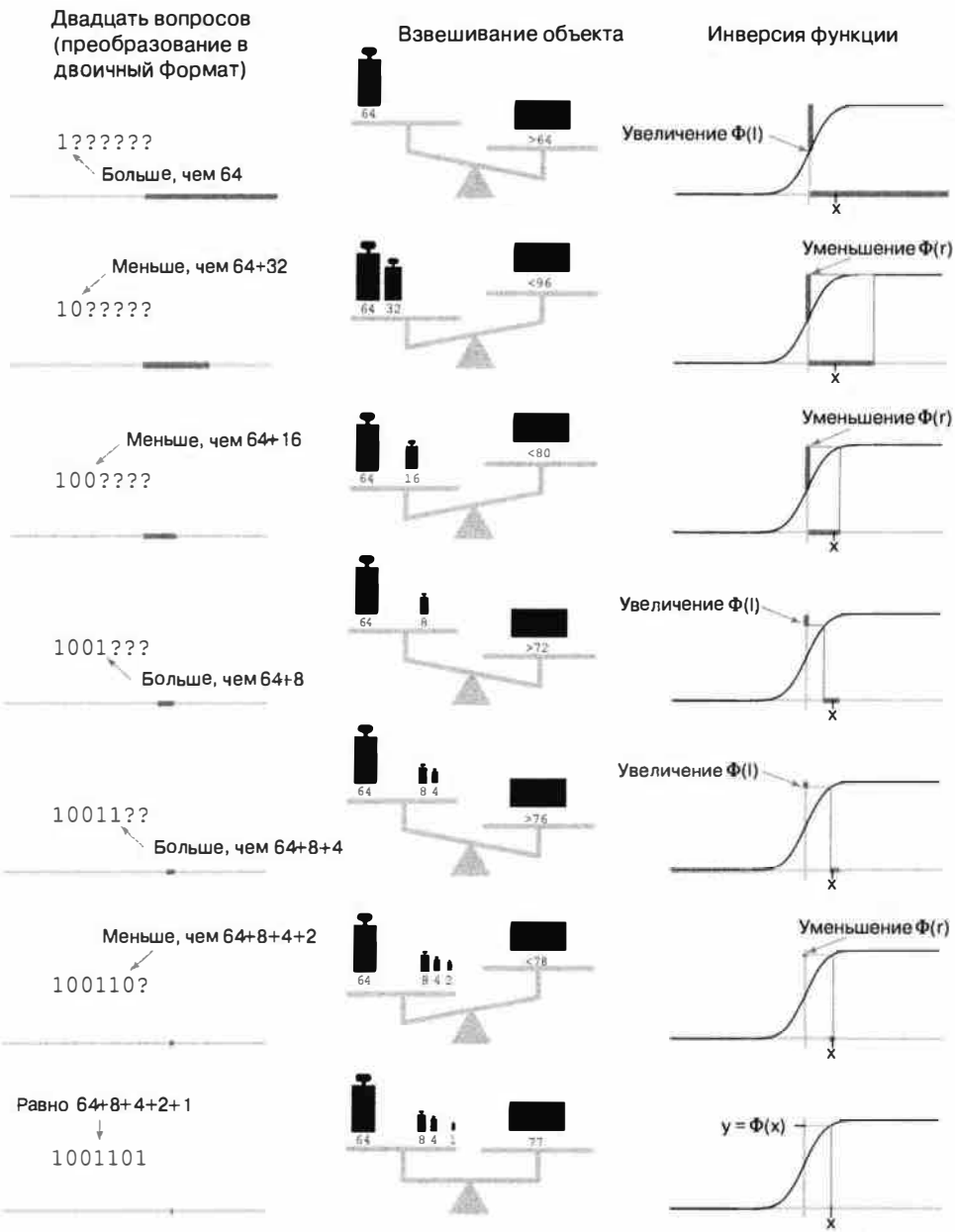
Программа 4.2.2. Дихотомический поиск (bisection.py)

```
import sys
import stdio
import gaussian

def invert(f, y, lo, hi, delta=0.00000001):
    mid = (lo + hi) / 2.0
    if (hi - lo) < delta:
        return mid
    if f(mid) > y:
        return invert(f, y, lo, mid, delta)
    else:
        return invert(f, y, mid, hi, delta)

def main():
    y = float(sys.argv[1])
    x = invert(gaussian.cdf, y, -8.0, 8.0)
    stdio.writef('%.3f\n', x)

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

f	Функция
y	Данное значение
delta	Точность
lo	Левая конечная точка
mid	Середина
hi	Правая конечная точка

Функция `invert()` в этой программе использует бинарный поиск для вычисления вещественного значения `x` в интервале `(lo, hi)`, для которого `f(x)` равна заданному значению `y`, с заданной точностью `delta` для любой функции f, возрастающей в этом интервале. Эта рекурсивная функция, делящая пополам интервал, содержащий заданное значение, вычисляет функцию в середине интервала и решает, принадлежит ли искомое значение `x` левой или правой половине. Деление продолжается, пока длина интервала не станет меньше заданной точности.

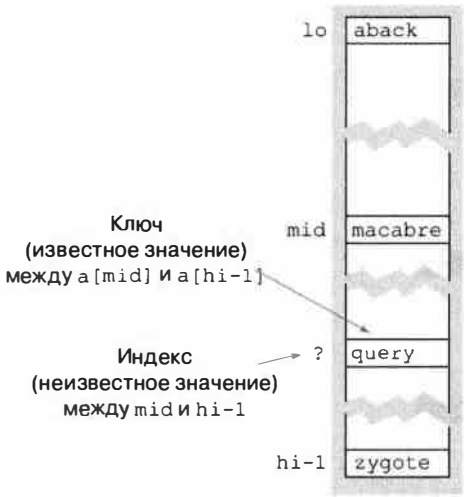
```
% python bisection.py .5
0.000

% python bisection.py .95
1.645

% python bisection.py .975
1.960
```



Три случая применения бинарного поиска

Бинарный поиск в отсортированном массиве. Один из важнейших случаев применения бинарного поиска — поиск информации с использованием ключа. В современных вычислениях он крайне популярен. Например, на протяжении нескольких прошлых столетий люди использовали такую вещь, как *словарь*, для поиска

определения слова и большую часть последнего столетия используют *телефонную книгу* для поиска номера телефона. В обоих случаях базовый механизм одинаков: записи отсортированы по идентифицирующему их ключу (слово в случае словаря и имя человека в случае телефонной книги, отсортированные в алфавитном порядке в обоих случаях). Вы, вероятно, нередко используете такую информацию на компьютере, но подумайте, как вы ищете слово в словаре. Решение "в лоб", известное как *последовательный поиск*, подразумевает исследование каждого ключа сначала по одному и продолжение, пока слово не будет найдено. Но этот подход никто не использует: вы открываете книгу на некой странице и ищете слово на ней. Если оно там есть, поиск закончен; в противном случае вы пропускаете часть книги перед текущей страницей или после, а затем повторяете поиск. Сейчас мы называем этот подход бинарным поиском. Смотрите ли вы точно посередине, несущественно; пока вы пропускаете каждый раз примерно постоянную долю ключей, ваш поиск будет логарифмическим.



Бинарный поиск в отсортированном массиве (один этап)

Исключающий фильтр. В разделе 4.3 мы рассмотрим подробности реализации типа компьютерной программы, которую вы используете вместо словаря или телефонной книги. Программа 4.2.3 (`binarysearch.py`) использует бинарный поиск для решения упрощенной *проблемы существования* (existence problem): есть ли в отсортированном массиве ключей заданный ключ? Например, при проверке правописания слова достаточно знать только то, что оно есть в словаре, и не важно, где именно. При компьютерном поиске мы храним информацию в массиве, отсортированном по ключу (для одних приложений информация поступает в отсортированном порядке; в других ее приходится предварительно сортировать, используя один из алгоритмов, обсуждаемых далее в этом разделе). Бинарный поиск в программе `binarysearch.py` отличается от других наших приложений в двух деталях. Во-первых, длина массива n не должна быть степенью числа 2. Во-вторых, следует учесть возможность отсутствия искомого ключа в массиве. При создании кода бинарного поиска с учетом этих деталей требуется некоторая осторожность, как обсуждается в разделе вопросов и ответов, а также в упражнениях.

Программа 4.2.3. Бинарный поиск (в отсортированном массиве) (binarysearch.py)

```
import sys
import stdio
from instream import InStream

def _search(key, a, lo, hi):
    if hi <= lo: return -1 # Не найдено.
    mid = (lo + hi) // 2
    if a[mid] > key:
        return _search(key, a, lo, mid)
    elif a[mid] < key:
        return _search(key, a, mid+1, hi)
    else:
        return mid

def search(key, a):
    return _search(key, a, 0, len(a))

def main():
    instream = InStream(sys.argv[1])
    a = instream.readAllStrings()
    while not stdio.isEmpty():
        key = stdio.readString()
        if search(key, a) < 0: stdio.writeln(key)

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

key	Искомый ключ
a[]	Отсортированный массив
Lo	Наименьший возможный индекс
mid	Середина
Hi	Наибольший возможный индекс

Метод `search()` использует для нахождения индекса ключа в отсортированном массиве бинарный поиск (если ключа в массиве нет, возвращается `-1`). Клиент проверки — исключающий фильтр, который читает отсортированный массив строк из файла белого списка, заданного в командной строке, и выводит слова, поступившие со стандартного ввода, отсутствующие в белом списке.

```
% more emails.txt
bob@office
carl@beach
marvin@spam
bob@office
bob@office
mallory@spam
dave@boat
eve@airport
5alice@home
```

```
% more white.txt
alice@home
bob@office
carl@beach
dave@boat

% python binarysearch.py white.txt < emails.txt
marvin@spam
mallory@spam
eve@airport
```

Клиент проверки в программе `binarysearch.py` известен как *исключающий фильтр*: он читает отсортированный массив строк из файла белого списка (whitelist) и произвольную последовательность строк со стандартного ввода, а затем выводит те строки последовательности, которых нет в белом списке. Исключающие фильтры применяются весьма широко. Например, если белый список содержит слова из словаря, а стандартный ввод — текстовый документ, то исключающий фильтр будет писать слова с орфографическими ошибками. Другой пример — применение в веб-приложениях: ваше приложение электронной почты могло бы использовать исключающий фильтр для отклонения любых сообщений, обратных адресов которых нет в белом списке ваших друзей, либо в вашей операционной системе может быть исключающий фильтр, отклоняющий все сетевые подключения к вашему компьютеру от любых устройств с IP-адресом, отсутствующим в заданном заранее белом списке.

Быстрые алгоритмы — важнейший элемент современного мира, а бинарный поиск — яркий пример, иллюстрирующий роль быстрых алгоритмов. Будь то огромные объемы экспериментальных данных или подробные представления небольшого количества аспектов физического мира, современным ученым приходится справляться с громадными объемами данных. Несколько несложных вычислений уверенно убеждают, что для таких задач, как поиск в документе всех слов с орфографической ошибкой или защита компьютера от злоумышленников с использованием исключающего фильтра, *требуют* такого быстрого алгоритма, как бинарный поиск. *Уделите этому время*, поскольку быстрый алгоритм может означать различие между возможностью решить задачу легко и просто либо трудно и сложно и с существенным расходом ресурсов (и потерпеть в результате неудачу). Поиск исключения в миллионе документов с миллионами слов и записей в белом списке может закончиться немедленно, тогда как та же задача могла бы занять дни или недели при использовании алгоритма решения "в лоб". В настоящее время веб-компании вполне рутинно оказывают услуги, подразумевающие выполнение бинарного поиска *миллиарды* раз в массивах с *миллиардами* элементов, — без столь быстрого алгоритма, как бинарный поиск, такие услуги были бы невозможны.

Сортировка вставкой. Бинарный поиск требует, чтобы массив был отсортирован. Сортировка имеет множество и других, прямых применений, поэтому мы теперь обратимся к алгоритмам сортировки. Сначала мы рассмотрим алгоритм решения "в лоб", а затем сложный алгоритм, применимый для огромных массивов.

Рассматриваемый нами алгоритм решения "в лоб" — это *сортировка вставкой* (insertion sort). Это самый простой подход, который люди зачастую используют для упорядочивания игральных карт — т.е. перебирают карты по одной и вставляют каждую в надлежащее место среди уже просмотренных и отсортированных.

Программа 4.2.4 (insertion.py) содержит реализацию функции sort(), подражающей этому процессу и сортирующую элементы в массиве a[] длиной n. Клиент проверки читает все строки со стандартного ввода, помещает их в массив и вызывает функцию sort() для их сортировки, а затем выводит отсортированный результат на стандартный вывод.

Внешний цикл for функции insertion.sort() сортирует первые i элементов массива. Внутренний цикл while завершает сортировку, помещая элемент a[i] в его надлежащую позицию в массиве, как в следующем примере, где i равно 6:

Вставка a[6] в позицию при обмене с большими элементами слева

i	*j*	*a*							
		0	1	2	3	4	5	6	7
6	6	**and**	**had**	**him**	**his**	**was**	**you**	*the*	but
6	5	and	had	him	his	was	*the*	**you**	but
6	4	and	had	him	his	*the*	**was**	you	but
		and	had	him	**his**	*the*	was	you	but

Элемент a[i] помещается на свое место среди сортированных элементов. Для этого он меняется местом с каждым большим элементом слева (используя вариант функции exchange(), с которой вы познакомились в разделе 2.1), двигаясь справа налево, пока не достигнет своей надлежащей позиции. Сравниваются и обмениваются (кроме последней) темные элементы в трех нижних рядах этой трассировки.

Только что описанный процесс вставки начинает выполняться с i, равного 1, затем 2, 3 и т.д., как показано в трассировке далее. Когда i достигает конца массива, весь массив оказывается отсортирован.

Вставка a[1] в позицию через a[n-1] (сортировка вставкой)

i	*j*	*a*							
		0	1	2	3	4	5	6	7
		was	**had**	**him**	**and**	**you**	**his**	**the**	**but**
1	0	*had*	**was**	him	and	you	his	the	but
2	1	had	*him*	**was**	and	you	his	the	but
3	0	*and*	**had**	**him**	**was**	you	his	the	but
4	4	and	had	him	was	*you*	his	the	but
5	3	and	had	him	*his*	**was**	**you**	the	but
6	4	and	had	him	his	*the*	**was**	**you**	but
7	1	and	*but*	**had**	**him**	**his**	**the**	**was**	**you**
		and	**but**	**had**	**him**	**his**	**the**	**was**	**you**

Эта трассировка отображает содержимое массива по завершении каждой итерации внешнего цикла for наряду с текущим значением j. Выделен тот элемент,

который в начале цикла был a[i], элементы, выделенные темным, — это другие элементы, задействованные в обмене и перемещающиеся вправо на одну позицию в пределах цикла. Элементы с a[0] до a[i-1] окажутся в отсортированном порядке по завершении цикла для каждого значения i. В частности, вся сортировка завершится, когда закончится цикл со значением i, равным len(a). Эти соображения снова иллюстрируют основную идею при изучении или разработке нового алгоритма: убедитесь в его корректности. Нужно понимать, что изучение производительности алгоритма позволяет использовать его эффективно.

Анализ продолжительности. Функция sort() содержит цикл while, вложенный в цикл for, что предполагает квадратичную продолжительность. Однако немедленно окончательный вывод по этому поводу мы сделать не можем, поскольку цикл while завершается, как только a[j] становится больше или равен a[j-1]. Например, в наилучшем случае, когда элементы массива уже отсортированы, цикл while оказывается не более чем сравнением (чтобы узнать, что a[j] больше или равно a[j-1] для каждого j от 1 до n-1), поэтому полная продолжительность линейна. С другой стороны, если массив отсортирован в обратном порядке, цикл while не завершится, пока j не будет равно 0. Следовательно, частота исполнения инструкций во внутреннем цикле такова:

$$1 + 2 + \ldots + n - 1 \sim n^2 / 2.$$

Таким образом, продолжительность является квадратичной (см. упр. 4.1.4). Чтобы выяснить производительность сортировки вставкой для *случайно* упорядоченных массивов, внимательно осмотрите трассировку: это массив n на n с одним темным элементом, соответствующим каждому обмену. Таким образом, количество темных элементов — это частота выполнения инструкций во внутреннем цикле. Мы ожидаем, что каждый вновь вставляемый элемент окажется в любой позиции с одинаковой вероятностью, так что в среднем элемент оказывается на полпути влево. Таким образом, мы ожидаем, что в среднем только половина элементов ниже диагонали будет темной (всего примерно $n^2 / 4$). Это наблюдение непосредственно ведет к гипотезе, что ожидаемая продолжительность сортировки вставкой для случайно упорядоченного массива является квадратичной.

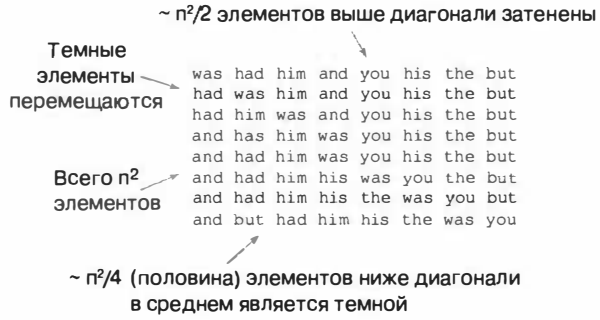
Программа 4.2.4. Сортировка вставкой (`insertion.py`)

```
import sys
import stdio

def exchange(a, i, j):
    a[i], a[j] = a[j], a[i]

def sort(a):
    n = len(a)
    for i in range(1, n):
        j = i
        while (j > 0) and (a[j] < a[j-1]):
            exchange(a, j, j-1)
            j -= 1

def main():
    a = stdio.readAllStrings()
    sort(a)
    for s in a:
        stdio.write(s + ' ')
    stdio.writeln()

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

`a[]`	Сортируемый массив
`n`	Количество элементов

Эта программа читает строки со стандартного ввода, сортирует их в порядке увеличения и выводит. Функция `sort()` является реализацией сортировки вставкой. Она сортирует массивы любого типа данных, поддерживающего оператор `<` (т.е. реализует метод `__lt__()`). Сортировка вставкой подходит для небольших и больших массивов в почти отсортированном состоянии, но она слишком медленна для больших неупорядоченных массивов.

```
% more tiny.txt
was had him and you his the but

% python insertion.py < tiny.txt
and but had him his the was you

% python insertion.py < TomSawyer.txt
tick tick tick tick tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick tick tick tick tick
```



Анализ сортировки вставкой

Эмпирический анализ. Программа 4.2.5 (`timesort.py`) предоставляет функции, необходимые для проверки нашей гипотезы о квадратичном характере сортировки вставкой для неупорядоченных массивов при проверке удвоением (см. раздел 4.1). Модуль включает две функции: `timeTrials()`, выполняющую эксперименты для задачи заданной сложности n, и `doublingTest()`, вычисляющую соотношения продолжительности между задачами сложностей $n/2$ и n. Мы можем также использовать эти функции для других алгоритмов сортировки. Для сортировки вставкой соотношение интерактивных сеансов Python стремится к 4, что подтверждает гипотезу о квадратичной продолжительности, высказанную в последнем разделе. Вы можете запустить приложение `timesort.py` на собственном компьютере. Как обычно, вы могли бы обратить внимание на результат кеширования или некую иную системную особенность при небольшом значении n, но квадратичная продолжительность должна быть вполне очевидной, и вы быстро убедитесь, что сортировка вставкой слишком медленна, чтобы ее можно было применять для больших входных данных.

Чувствительность к вводу. Обратите внимание, что каждая функция в программе `timesort.py` получает аргумент `trials` и выполняет `trials` экспериментов для каждой сложности задачи, а не только одной. Как мы недавно заметили, одна из причин этого в том, что *продолжительность сортировки вставкой чувствительна к входным значениям.* Это поведение весьма отличается, например, от программы `threesum.py` и означает необходимость тщательно интерпретировать результаты нашего анализа. Неверно будет категорически утверждать, что продолжительность сортировки вставкой будет квадратной, поскольку ваше приложение могло бы получать ввод, для которого продолжительность линейна. Когда производительность алгоритма чувствительна к вводимым значениям, вы не можете делать точные прогнозы, не принимая во внимание входные значения. Мы будем часто возвращаться к этой проблеме, поскольку она нередко возникает в реальных случаях. Например, находятся ли необходимые нам строки в неотсортированном порядке? Этот случай не так уж и редок. В частности, если в большом массиве несколько элементов находятся не на месте, то сортировка вставкой — удачный выбор.

Программа 4.2.5. Проверка сортировки удвоением (`timesort.py`)

```
import stdio
import stdrandom
import stdarray
from stopwatch import Stopwatch

def timeTrials(f, n, trials):
    total = 0.0
    a = stdarray.create1D(n, 0.0)
    for t in range(trials):
        for i in range(n):
            a[i] = stdrandom.uniformFloat(0.0, 1.0)
        watch = Stopwatch()
        f(a)
        total += watch.elapsedTime()
    return total

def doublingTest(f, n, trials):
    while True:
        prev = timeTrials(f, n // 2, trials)
        curr = timeTrials(f, n, trials)
        ratio = curr / prev
        stdio.writef('%7d %4.2f\n', n, ratio)
        n *= 2
```

`f()`	Проверяемая функция
`n`	Сложность задачи
`trials`	Количество испытаний
`total`	Полное прошедшее время
`a[]`	Сортируемый массив
`watch`	Секундомер

`n`	Сложность задачи
`prev`	Продолжительность для `n // 2`
`curr`	Продолжительность для `n`
`ratio`	Соотношение продолжительности

Функция `timeTrials()` запускает функцию `f()` для массивов из `n` случайных чисел типа `float`, выполняя эксперимент `trials` раз. Проведение нескольких экспериментов дает более точный результат, поскольку они снижают влияние системных эффектов и зависимости от ввода. Функция `doublingTest()` осуществляет проверку удвоением, начиная с `n`, удваивая `n` и выводя соотношение времени для текущего `n` и времени для предыдущего `n` на каждом цикле.

```
% python
>>> import insertion
>>> import timesort
>>> timesort.doublingTest(insertion.sort, 128, 100)
   128 3.90
   256 3.93
   512 3.98
  1024 4.12
  2048 4.13
```

Сравнимые ключи. Мы хотим быть в состоянии сортировать данные любых типов, обладающих естественным порядком. В научном приложении нам может понадобиться сортировать числовые результаты экспериментов; в коммерческом приложении может понадобиться сортировка по денежным суммам, времени или дате; в системном программном обеспечении это могли бы быть IP-адреса или номера учетных записей.

К счастью, функции сортировки вставкой и бинарного поиска работают не только со строками, но и с любыми *сравнимыми* типами данных. Как упоминалось в разделе 3.3, тип данных сравним, если он реализует шесть методов сравнения, а они *полный порядок*. Все встроенные типы Python, включая `int`, `float` и `str`, сравнимы.

Вы можете сделать пользовательский тип сравнимым, реализовав шесть специальных методов, соответствующих операторам <, <=, ==, !=, >= и >. Фактически наши функции сортировки вставкой и бинарного поиска полагаются только на оператор <, но хороший стиль подразумевает реализацию всех шести специальных методов.

Для пользовательских типов иногда лучше подходит более общий проект. Например, преподаватель мог бы пожелать сортировать файл записей студенческих оценок по именам, а в некоторых других случаях по оценкам. Одно из наиболее популярных решений в таких ситуациях подразумевает передачу функции, используемой для сравнения ключей, в качестве аргумента функции сортировки.

Тем не менее в реальном приложении продолжительность сортировки вставкой, вероятней всего, *окажется квадратичной*, поэтому необходимо рассмотреть более быстрые алгоритмы сортировки. Как упоминалось в разделе 4.1, простое и быстрое вычисление вполне способно продемонстрировать, что наличие более быстрого компьютера не окажет существенной помощи. Словарь, база научных или коммерческих данных вполне может содержать миллиарды элементов, как же тогда отсортировать такой большой массив? Поэтому мы обращаемся к классическому алгоритму решения этой задачи.

Сортировка с объединением. Для разработки более быстрого метода сортировки мы используем рекурсию (как для бинарного поиска) с применением к проекту алгоритма принципа *разделяй и властвуй* (divide-and-conquer), который должен понимать каждый программист. Этот принцип решения задачи подразумевает ее разделение на независимые части, независимое решение этих частей и последующее применение этих решений для общего решения всей задачи. Для сортировки

Ввод
`was had him and you his the but`
Сортировка влево
`and had him was you his the but`
Сортировка вправо
`and had him was but his the you`
Объединение
`and but had him his the was you`

Пример сортировки с объединением

массива сравнимых ключей с использованием этой стратегией мы делим его на две половины, сортируем эти две половины независимо, а затем *объединяем* результаты сортировки в полный массив. Это метод *сортировки с объединением* (mergesort).

Мы выражаем смежные подмассивы целого массива, используя запись [*lo*, *hi*), имея в виду *a*[*lo*], *a*[*lo* + 1], ..., *a*[*hi* – 1] (та же форма, что и при обозначении полуоткрытого интервала, исключающего *a*[*hi*] в бинарном поиске). Для сортировки *a*[*lo*, *hi*) мы используем следующую рекурсивную стратегию.

- *Конечный случай.* Если длина подмассива 0 или 1, он уже отсортирован.
- *Рекурсивный этап.* В противном случае вычислить *mid* = (*hi* + *lo*) / 2, отсортировать (рекурсивно) два подмассива *a*[*lo*, *mid*) и *a*[*mid*, *hi*), а затем объединить их.

Программа 4.2.6 (`merge.py`) реализует этот алгоритм. Элементы массива перестраивают рекурсивный код, *объединяющий* две половины рекурсивно отсортированного массива. Как обычно, проще всего изучить процесс объединения на примере трассировки содержимого массива. Код содержит по одному индексу для первой (`i`) и второй (`j`) половин массива, а также третий индекс `k` для вспомогательного массива `aux[]`, содержащего результат. Реализация объединения — это одиночный цикл, присваивающий `aux[k]` либо `a[i]`, либо `a[j]` (а затем увеличивает значения `k` и используемых индексов). Если `i` или `j` достигли конца своего подмассива, `aux[k]` присваиваются значения из другого; в противном случае присваивается меньшее из `a[i]` или `a[j]`. После того как все элементы из двух половин скопированы в массив `aux[]`, отсортированный результат копируется назад, в исходный массив. Уделите минуту изучению трассировки и убедитесь, что этот код всегда правильно объединяет два отсортированных подмассива в целый отсортированный массив.

Трассировка объединения отсортированной левой половины с отсортированной правой половиной

i	*j*	*k*	*aux[k]*	*a[]*							
				0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
				and	**had**	**him**	**was**	**but**	**his**	**the**	**you**
0	4	0	**And**	**and**	had	him	was	**but**	his	the	you
1	4	1	**But**	and	**had**	him	was	**but**	his	the	you
1	5	2	**Had**	and	**had**	him	was	but	**his**	the	you
2	5	3	**Him**	and	had	**him**	was	but	**his**	the	you
3	5	4	**His**	and	had	him	**was**	but	**his**	the	you
3	6	5	**The**	and	had	him	**was**	but	his	**the**	you
3	7	6	**Was**	and	had	him	**was**	but	his	the	**you**
4	7	7	**You**	and	had	him	was	but	his	the	**you**

Программа 4.2.6. Сортировка с объединением (merge.py)

```
import sys
import stdio
import stdarray

def _merge(a, lo, mid, hi, aux):
    n = hi - lo
    i = lo
    j = mid
    for k in range(n):
        if   i == mid:    aux[k] = a[j]; j += 1
        elif j == hi:     aux[k] = a[i]; i += 1
        elif a[j] < a[i]: aux[k] = a[j]; j += 1
        else:             aux[k] = a[i]; i += 1
    a[lo:hi] = aux[0:n]

def _sort(a, lo, hi, aux):
    n = hi - lo
    if n <= 1: return
    mid = (lo + hi) // 2
    _sort(a, lo, mid, aux)
    _sort(a, mid, hi, aux)
    _merge(a, lo, mid, hi, aux)

def sort(a):
    n = len(a)
    aux = stdarray.create1D(n)
    _sort(a, 0, n, aux)
```

a[lo, hi)	Сортируемый подмассив
n	Длина подмассива
mid	Середина
aux[]	Вспомогательный массив для объединения

Функция `sort()` в этом модуле — весьма быстрая и позволяет сортировать массивы любого сравнимого типа данных. Она основана на рекурсивной сортировке массива `a[lo, hi)` за счет рекурсивной сортировки двух его половин и их последующего слияния для формирования отсортированного результата. Вывод, приведенный ниже, — это трассировка сортируемого подмассива при каждом вызове функции `sort()` (см. упр. 4.2.8). В отличие от функций `insertion.sort()` и `merge.sort()`, она вполне подходит для сортировки огромных массивов.

```
% python merge.py < tiny.txt
 was had him and you his the but
 had was
         and him
 and had him was
                 his you
                         but the
                 but his the you
 and but had him his the was you
```

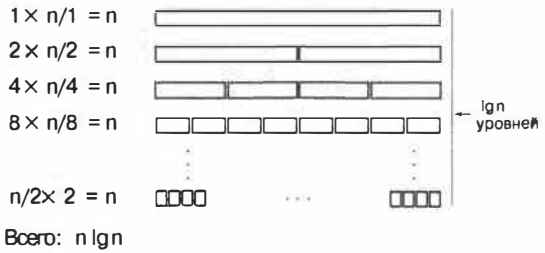
Рекурсивный метод гарантирует, что две половины массива окажутся перед объединением в отсортированном порядке. И опять-таки проще всего получить представление об этом процессе, изучая трассировку содержимого массива при каждом рекурсивном выходе из метода `sort()`. Такая трассировка для данного примера представлена ниже. Сначала объединяются `a[0]` и `a[1]`, создавая отсортированный подмассив `a[0, 2)`, затем `a[2]` и `a[3]` объединяются, создавая отсортированный подмассив `a[2, 4)`, затем эти два подмассива размером 2 объединяются, создавая отсортированный подмассив `a[0, 4)`, и т.д. Если вы убедились, что объединение работает правильно, остается убедиться в правильности деления кодом массив, что гарантирует корректную работу сортировки. Обратите внимание, что при нечетном количестве элементов левая половина будет иметь на один элемент меньше, чем правая.

Трассировка рекурсивных вызовов сортировки с объединением

	a[]							
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
	was	**had**	**him**	**and**	**you**	**his**	**the**	**but**
`_sort(a, 0, 8, aux)`								
`_sort(a, 0, 4, aux)`								
`_sort(a, 0, 2, aux)`								
return	**had**	**was**	him	and	you	his	the	but
`_sort(a, 2, 4, aux)`								
return	had	was	**and**	**him**	you	his	the	but
return	**and**	**had**	**him**	**was**	you	his	the	but
`_sort(a, 4, 8, aux)`								
`_sort(a, 4, 6, aux)`								
return	and	had	him	was	**his**	**you**	the	but
`_sort(a, 6, 8, aux)`								
return	and	had	him	was	his	you	**but**	**the**
return	and	had	him	was	**but**	**his**	**the**	**you**
return	**and**	**but**	**had**	**him**	**his**	**the**	**was**	**you**

Анализ продолжительности. Внутренний цикл сортировки с объединением сосредоточен на вспомогательном массиве. Цикл `for` задействует n итераций, следовательно, частота выполнения инструкций во внутреннем цикле пропорциональна сумме длин подмассивов во всех вызовах рекурсивной функции. Для получения этого значения следует упорядочить вызовы по уровням согласно их размеру. Для простоты предположим, что n — степень числа 2, тогда $n = 2^k$. На первом уровне имеется один вызов размером n; на втором уровне два вызова размером $n/2$; на третьем — четыре размером $n/4$ и т.д. до последнего уровня с $n/2$ вызовами размером 2. Всего есть $k = \lg n$ уровней, что в общем дает $n \lg n$ для частоты выполнения инструкций во внутреннем цикле сортировки

с объединением. Это уравнение подтверждает гипотезу о том, что продолжительность сортировки с объединением является линейно-логарифмической. *Примечание.* Когда *n* не степень числа 2, подмассивы на каждом уровне не обязательно имеют одинаковый размер, но количество уровней все еще остается логарифмическим, поэтому линейно-логарифмическая гипотеза подтверждается для всех *n* (см. упр. 4.2.14-4.2.15). Интерактивный сценарий Python, приведенный ниже, использует для проверки этой гипотезы функцию `timesort.doublingTest()` (см. программу 4.2.5).



Внутренний цикл сортировки с объединением (когда n — степень числа 2)

Выполните эти проверки на своем компьютере. Сделав это, вы, конечно, заметите, что функция `merge.sort()` намного быстрее для больших массивов, чем `insertion.sort()`, и что вы можете сортировать огромные массивы с относительной легкостью. Подтверждение гипотезы о линейно-логарифмической (а не линейной) продолжительности требует немного больше работы, но вы увидите, что сортировка с объединением позволяет решать масштабные задачи сортировки, никак не решаемые "в лоб" такими алгоритмами, как сортировка вставкой.

```
% python
...
>>> import merge
>>> import timesort
>>> timesort.doublingTest(
... merge.sort, 1024, 100)
   1024 1.92
   2048 2.19
   4096 2.07
   8192 2.13
  16384 2.13
  32768 2.11
  65536 2.31
 131072 2.14
 262144 2.29
 524288 2.13
1048576 2.17
```

Квадратно-линейно-логарифмическая пропасть. Различие между n^2 и $n \log n$ имеет огромное значение в практических случаях, оно подобно линейно-логарифмической пропасти, преодолеваемой бинарным поиском.

Параметры "тесного мира" для разных графов на 1 000 вершин. Для очень многих важных вычислительных задач ускорение, получаемое при переходе с квадратного на линейно-логарифмический алгоритм (такой, как переход с сортировки вставкой на сортировку с объединением), означает различие между способностью решить задачу для огромных объемов данных и неспособностью решить ее вообще.

Системная сортировка Python. Python включает две операции сортировки. Метод `sort()` во встроенном типе данных `list` перестраивает элементы списка в порядке возрастания, подобно функции `merge.sort()`. Встроенная функция `sorted()`, напротив, оставляет базовый список нетронутым, но возвращает новый список, содержащий элементы в порядке возрастания. Интерактивный сценарий Python, приведенный ниже, иллюстрирует оба способа. Системная сортировка Python использует версию `merge.sort()`. Она, вероятно, существенно быстрее (10 – 20×), чем `merge.py`, поскольку использует низкоуровневую реализацию (в Python такую не составить), а следовательно, без существенных дополнительных затрат, налагаемых Python на свои программы. Наряду с собственными реализациями сортировки вы вполне можете использовать системную сортировку с любым сравнимым типом данных, таким как встроенные типы данных Python `str`, `int` и `float`.

```
% python
...
>>> a=[3, 1, 4, 1, 5]
>>> b=sorted(a)
>>> a
[3, 1, 4, 1, 5]
>>> b
[1, 1, 3, 4, 5]

>>> a.sort()
>>> a
[1, 1, 3, 4, 5]
```

Сортировка с объединением восходит к *Джону фон Нейману* (John von Neumann), известному физику, который одним из первых заметил важность вычислений в научном исследовании. Фон Нейман внес существенный вклад в информатику, включая базовую концепцию компьютерной архитектуры, использующуюся с 1950-х годов. Что касается программирования, то фон Нейман открыл следующее:

- Сортировка — основной компонент многих приложений.
- На практике квадратичные алгоритмы слишком медленны.
- Подход "разделяй и властвуй" эффективен.
- Важны доказательство корректности программ и знание их стоимости.

С тех пор компьютеры стали на много порядков быстрее и обладают на много порядков большей памятью, но эти фундаментальные концепции остаются важными и сегодня. Те, кто использует компьютеры эффективно и успешно, знают, как и фон Нейман, что алгоритмы "в лоб" зачастую только начало. Python и другие современные системы все еще используют метод фон Неймана, доказывая обоснованность его идей.

Применение: подсчет частот. Программа 4.2.7 (`frequencycount.py`) читает последовательность строк со стандартного ввода и выводит таблицу отдельных найденных строк и количества их применения в порядке убывания частоты. Это вычисление полезно во множестве случаев: лингвист мог бы изучать шаблоны применения слов в длинных текстах, ученый — искать часто происходящие события в экспериментальных данных, коммерсант — искать клиентов, чаще всего встречающихся в длинном списке транзакций, а сетевой аналитик — искать наиболее активных пользователей. В каждом из этих случаев могли бы быть задействованы миллионы строк и даже больше, поэтому необходим линейно-логарифмический алгоритм (или даже лучший). Программа 4.2.7 достигает этого *двумя* сортировками.

Вычисление частот. Наш первый этап подразумевает чтение строк со стандартного ввода и их сортировку. В данном случае нам не столь важен факт сортировки строк по порядку, сколько факт *помещения равных строк рядом.* Если ввод такой:

```
to be or not to be to
```

то результат сортировки будет следующим:

```
be be not or to to to
```

То есть равные строки, такие как два вхождения `be` и три вхождения `to`, расположены в массиве рядом. Теперь, когда все равные строки собраны в массиве вместе, для подсчета всех частот достаточно одного прохода по массиву. Тип данных `Counter` (программа 3.3.2) является совершенным инструментом для решения этой задачи. Напомним, что у типа `Counter` есть строковая переменная экземпляра (инициализируемая аргументом конструктора), переменная экземпляра счетчика (инициализированная значением 0) и метод `increment()`, увеличивающий значение счетчика на 1. Мы создадим список Python `zipf[]` объектов типа `Counter`, а затем сделаем для каждой строки следующее:

Вычисление частот

i	M	a[i]	zipf[i].value()			
			0	1	2	3
	0					
0	1	be	1			
1	1	be	2			
2	2	not	2	1		
3	3	or	2	1	1	
4	4	to	2	1	1	1
5	4	to	2	1	1	2
6	4	to	2	1	1	3
			2	1	1	3

- если строка не равна предыдущей, создадим новый объект типа `Counter` и добавим в конец массива `zipf[]`;
- увеличим значение последнего созданного объекта типа `Counter`.

В конце этого процесса элемент `zipf[i]` содержит `i`-ю строку и ее частоту.

Программа 4.2.7. Подсчет частот (`frequencycount.py`)

```
import sys
import stdio
from counter import Counter

words = stdio.readAllStrings()
words.sort()      # или merge.sort(words)
zipf = []
for i in range(len(words)):
    if (i == 0) or (words[i] != words[i-1]):
        entry = Counter(words[i], len(words))
        zipf += [entry]
    zipf[len(zipf) - 1].increment()
zipf.sort()       # или merge.sort(zipf)
zipf.reverse()
for entry in zipf:
stdio.writeln(entry)
```

`words`	Строки во вводе
`zipf[]`	Массив счетчиков

Эта программа сортирует слова, поступающие со стандартного ввода, использует отсортированный массив для подсчета частоты вхождений каждого, а затем сортирует частоты. В использованном ниже файле проверки больше 20 миллионов слов. График сравнивает `i`-ю частоту (полосы) относительно линии `1/i`. Программа подразумевает, что тип данных `Counter` сравним (см. программу 3.3.2).

```
% python frequencycount.py < leipzig.txt
1160105 the
593492 of
560945 to
472819 a
435866 and
430484 in
205531 for
192296 The
188971 that
172225 is
148915 said
147024 on
...
```

Сортировка частот. Затем мы сортируем объекты `Counter` по частоте. Мы можем сделать это в клиентском коде, нарастив тип данных `Counter` так, чтобы он включал все шесть методов сравнения, чтобы его объекты можно было сравнивать по счету. Таким образом, мы просто отсортируем массив объектов `Counter`,

Сортировка частот

i		zipf[i]
До		
0	2	Be
1	1	Not
2	1	Or
3	3	To
После		
0	1	Not
1	1	Or
2	2	Be
3	3	To

чтобы перестроить его в порядке возрастания частот! После этого мы изменим порядок массива на обратный, чтобы элементы располагались в порядке убывания частот. И наконец, выводим каждый объект типа `Counter` на стандартный вывод. Как обычно, Python для этого автоматически вызывает встроенную функцию `str()`, в результате объекты `Counter` выводятся как число, сопровождаемое строкой.

Закон Ципфа. Программа `frequencycount.py` осуществляет элементарный лингвистический анализ: какие слова чаще всего встречаются в тексте? *Закон Ципфа* (Zipf's law) гласит, что частота i-го самого частого слова в тексте из m отдельных слов пропорциональна $1/i$ (его коэффициент пропорциональности является m-м гармоническим числом). Например, второе наиболее распространенное слово должно встречаться примерно вполовину реже первого. Эта эмпирическая гипотеза подтверждается на удивительном разнообразии ситуаций: от финансовых данных до статистики использования веб. Клиент проверки в программе 4.2.7 проверяет закон Ципфа для базы данных, содержащей 1 миллион взятых из веб выражений (см. сайт книги).

Вам самому, вероятно, приходилось (или придется в будущем) создавать программы для решения простой задачи, легко решаемой с использованием предварительной сортировки. При линейно-логарифмическом алгоритме сортировки, таком как сортировка с объединением, вы можете решать подобные задачи даже для очень больших наборов данных. Яркий пример — программа 4.2.7 (`frequencycount.py`), использующая две разные сортировки. Без хорошего алгоритма (и понимания его характеристик производительности) вас могла бы неприятно удивить неспособность вашего дорогого и быстрого компьютера решить задачу, кажущуюся простой. Несмотря на постоянно растущий набор задач, если вы знаете, как их решать эффективно, ваш компьютер может оказаться намного более эффективным инструментом, чем вы можете себе вообразить.

Уроки. Подавляющее большинство написанных вами программ подразумевает решение новой сложной практической задачи в результате разработки четкого, ясного и правильного решения, разделения программы на модули управляемого размера и использования простых доступных ресурсов. С самого начала в этой книге мы демонстрировали разработку программ согласно этим принципам. Но встретившись когда-нибудь с более сложными задачами, вы убедитесь, что четкого и правильного решения не всегда достаточно, поскольку ограничивающим фактором может оказаться стоимость вычисления. Примеры данного раздела — основная иллюстрация этого факта.

Учитывайте стоимость вычисления. Если вы можете быстро решить небольшую задачу простым алгоритмом — прекрасно. Но если необходимо решить задачу, подразумевающую работу с большим объемом данных или существенный объем вычислений, необходимо принять во внимание стоимость. Для сортировки вставкой мы сделали быстрый анализ и убедились, что метод решения "в лоб" неосуществим для больших массивов.

Алгоритмы "разделяй и властвуй". Стоит немного задуматься о мощи парадигмы "разделяй и властвуй", продемонстрированной при разработке логарифмического алгоритма поиска (бинарный поиск) и линейно-логарифмического алгоритма сортировки (сортировка с объединением). Тот же базовый подход эффективен для многих важных задач, как вы узнаете, пройдя курс разработки алгоритмов. (Имеет смысл выполнить упражнения в конце этого раздела, в которых представлены задачи, вполне решаемые алгоритмами "разделяй и властвуй", но никак не решаемые без них.)

Сведение к сортировке. Мы говорим, что задача A *сводится к* задаче B, если можем использовать решение B для решения A. Разработка нового алгоритма типа "разделяй и властвуй" с самого начала иногда родственна решению задачи, требующей наличия практического опыта и изобретательности, поэтому нельзя быть уверенным в том, что это получится сразу. Тем не менее зачастую эффективен самый простой подход: если дана новая задача, имеющая квадратичное решение "в лоб", спросите себя, как вы решили бы ее, будь данные отсортированы. Как правило, далее задачу решает относительно простой линейный проход по отсортированным данным. Таким образом, мы получаем линейно-логарифмический алгоритм с хитростью, скрытой в реализации сортировки с объединением. Рассмотрим, например, задачу выяснения уникальности значения каждого элемента в массиве. Эта задача сводится к сортировке, поскольку мы можем отсортировать массив, а затем перебрать его и проверить равенство значения текущего элемента следующему. Если равенство не обнаружено, значит, все элементы уникальны.

Знайте свои основные инструменты. Наша способность подсчитать частоту слов в огромном тексте, как в программе 4.2.7 (`frequencycount.py`), зависит от характеристик производительности специфических операций с массивами и списками. Во-первых, его эффективность зависит от того факта, что сортировка с объединением способна сортировать массив за линейно-логарифмическое время. Во-вторых, эффективность создания массива `zipf[]` зависит от того факта, что создание списка Python последовательным добавлением элементов по одному за раз занимает время и пространство, линейно зависящее от длины получаемого списка. На этом основана возможность создания такой структурой данных Python, как массив переменного размера (см. в разделе 4.1). Как разработчик приложений вы должны быть бдительны, поскольку не все языки программирования

предоставляют такую эффективную реализацию добавления элемента в список (и характеристики производительности редко указывают в API).

С появлением компьютеров люди стали разрабатывать такие алгоритмы, как бинарный поиск и сортировка с объединением, способные эффективно решить практические задачи. Такая научная область, как *проектирование и анализ алгоритмов*, охватывает исследование таких парадигм проектирования, как "разделяй и властвуй", методик выработки гипотез о производительности алгоритмов и самих алгоритмов для решения таких фундаментальных проблем, как сортировка и поиск, применимых в практических приложениях всех видов. Реализации большинства этих алгоритмов находятся в библиотеках Python и других специализированных библиотеках, но понимание основных вычислительных инструментов ведет к пониманию фундаментальных инструментов математики и науки. Вы можете использовать пакет `matrixprocessing` для поиска собственных значений (eigenvalue) матрицы, но вам все равно понадобится пройти курс линейной алгебры, чтобы применить концепции и интерпретировать результаты. Теперь, когда вы *знаете*, что быстрый алгоритм может означать быстрое и правильное решение практической задачи, вы можете сами найти ситуации, где такие эффективные алгоритмы, как бинарный поиск и сортировка с объединением, могут решить задачу или обеспечить возможности, когда проектирование и анализ алгоритма будут иметь значение.

Вопросы и ответы

Зачем такие сложности с доказательством корректности программы?

Сэкономишь — потеряешь, и бинарный поиск — известный тому пример. Например, вы теперь знаете о бинарном поиске; классическое упражнение по программированию — составить версию, использующую цикл `while` вместо рекурсии. Попробуйте выполнить упражнения 4.2.1–4.2.3, не прибегая к коду, приведенному в книге ранее. В известном эксперименте Джон Бентли попросил нескольких профессиональных программистов сделать это, и решения большинства из них оказались *некорректными*.

Зачем изучать алгоритм сортировки с объединением, когда Python предоставляет эффективный метод `sort()`, определенный в типе данных `list`?

Подобно многим изучаемым темам, вы будете эффективней использовать инструменты, если поймете причину их появления.

Какова продолжительность выполнения следующей версии сортировки вставкой для уже отсортированного массива?

```
def sort(a):
    n = len(a)
    for i in range(1, n):
        for j in range(i, 0, -1):
            if a[j] < a[j-1]: exchange(a, j, j-1)
            else:             break
```

В Python 2 время будет *квадратичным*; в Python 3 — *линейным*. Дело в том, что в Python 2 функция `range()` возвращает массив целых чисел, длина которого равна длине диапазона (что может оказаться расточительным, если цикл закончится преждевременно из-за оператора `break` или `return`). В Python 3 функция `range()` возвращает итератор, создающий столько целых чисел, сколько необходимо.

Что будет при попытке сортировать массив элементов не совпадающего типа?

Если элементы имеют совместимые типы (такие, как `int` и `float`), то все сработает прекрасно. Например, смешанные числовые типы сравниваются согласно их числовому значению, таким образом, значения `0` и `0.0` считаются равными. Если типы элементов несовместимы (например, `str` и `int`), то Python 3 передает во время выполнения сообщение об ошибке `TypeError`. Python 2 поддерживает сравнение некоторых смешанных типов, используя имя класса для определения меньшего объекта. Например, Python 2 считает, что любые целые числа меньше любых строк, поскольку имя `'int'` лексикографически меньше имени `'str'`.

Каков порядок при сравнении строк с использованием таких операторов, как == и <?

Неофициально для сравнения двух строк Python использует *лексикографический порядок*, как у слов в словаре. Например, слова `'hello'` и `'hello'` равны, `'hello'` и `'goodbye'` неравны, причем `'goodbye'` меньше, чем `'hello'`. Более формально Python сначала сравнивает первые символы каждой строки. Если они отличаются, то результат сравнения строк в целом определяется сравнением этих двух символов. В противном случае Python сравнивает вторые символы каждой строки. Если они отличаются, то строки в целом сравниваются, как эти два символа. Продолжая так далее, если Python достигает конца обеих строк одновременно (он считает их равными). В противном случае он считает меньшей более короткую строку. Для посимвольного сравнения Python использует Unicode. Вот некоторые из важнейших свойств:

- `'0'` меньше `'1'` и так далее.
- `'A'` меньше `'B'` и так далее.
- `'a'` меньше `'b'` и так далее.
- Десятичные цифры (от `'0'` до `'9'`) меньше прописных букв (от `'A'` до `'Z'`).
- Прописные буквы (от `'A'` до `'Z'`) меньше строчных букв (от `'a'` до `'z'`).

Упражнения

4.2.1. Разработайте реализацию программы 4.2.1 (`questions.py`), получающую как аргумент командной строки максимальное количество чисел `n` (это не обязательно должна быть степень числа 2). Докажите правильность вашей реализации.

4.2.2. Составьте не рекурсивную версию бинарного поиска (программа 4.2.3).

4.2.3. Модифицируйте программу `binarysearch.py` так, чтобы при нахождении искомого ключа поиска в массиве она возвращала наименьший индекс `i`, для которого `a[i]` равен `key`, а в противном случае возвращала наибольший индекс `i`, для которого `a[i]` меньше, чем `key` (или –1, если такого индекса не существует).

4.2.4. Опишите, что будет при применении бинарного поиска к неупорядоченному массиву. Почему вы не обязаны проверять, отсортирован ли массив, перед каждым бинарным поиском? Могли бы вы проверить, что бинарный поиск элементов проходит в порядке возрастания?

4.2.5. Опишите, почему при бинарном поиске желательно использовать неизменяемые ключи.

4.2.6. Пусть функция $f()$ монотонно возрастает при $f(a) < 0$ и $f(b) > 0$. Составьте программу, вычисляющую значение x, для которого $f(x) = 0$ (с точностью до заданной допустимой ошибки).

4.2.7. Добавьте в программу `insertion.py` код, выводящий приведенную в тексте трассировку.

4.2.8. Добавьте в программу `merge.py` код, выводящий приведенную в тексте трассировку.

4.2.9. Добавьте код вывода трассировки в сортировку вставкой и сортировку с объединением в стиле трассировок в тексте, чтобы получить ввод `it was the best of times it was`.

4.2.10. Составьте программу `dedup.py`, читающую строки со стандартного ввода и выводящую их без дубликатов (и в отсортированном порядке).

4.2.11. Составьте версию сортировки с объединением (программа 4.2.6), создающую вспомогательный массив при каждом рекурсивном вызове функции `_merge()`, вместо создания только одного вспомогательного массива в функции `sort()` и передачи его как аргумента. Какое влияние это изменение оказывает на производительность?

4.2.12. Составьте не рекурсивную версию сортировки с объединением (программа 4.2.6).

4.2.13. Найдите распределение частот (гистограмму) слов в вашей любимой книге. Подчиняется ли она закону Ципфа?

4.2.14. Проанализируйте сортировку с объединением математически, когда n — степень числа 2, как это было сделано для бинарного поиска.

Решение. Пусть $M(n)$ — это частота исполнения инструкций во внутреннем цикле. Таким образом, $M(n)$ должна удовлетворять следующему рекуррентному соотношению:

$$M(n) = 2M(n/2) + n$$

при $M(1) = 0$. Подстановка 2^k для n даст

$M(2^k) = 2M(^{2k-1}) + 2^k$. Это подобно, но немного сложнее, чем рекуренция, рассматриваемая для бинарного поиска. Но если поделить обе стороны на 2^k, то мы получим

$$M(2^k) / 2^k = M(2^{k-1}) / 2^{k-1} + 1,$$

что *точно* повторят рекуренцию, полученную для бинарного поиска. Таким образом, $M(2^k) / 2^k = T(2^k) = k$. Обратная подстановка n для 2^k (и $\lg n$ для k) дает результат $M(n) = n \lg n$.

4.2.15. Проанализируйте сортировку с объединением для случая, когда n не степень числа 2.

Частичное решение. Когда n — нечетное число, у одного из подмассивов должно быть на один элемент больше, чем у другого; когда n не является степенью числа 2, подмассивы на каждом уровне не обязательно все будут одинакового размера. Но каждый элемент окажется в некоем подмассиве, и количество уровней все еще останется логарифмическим, поэтому для всех n останется справедлива линейно-логарифмическая гипотеза.

Практические упражнения

Следующие упражнения призваны помочь приобрести практический опыт по разработке быстрых решений типичных проблем. Подумайте об использовании бинарного поиска, или сортировки с объединением, или другого алгоритма "разделяй и властвуй" собственного изобретения. Реализуйте и проверьте свой алгоритм.

4.2.16. *Медиана.* Реализуйте функцию `median()` в программе `stdstats.py` так, чтобы она вычисляла медиану за линейно-логарифмическое время. *Подсказка.* Сведите задачу к сортировке.

4.2.17. *Наиболее вероятное значение.* Добавьте в программу `stdstats.py` функцию `mode()`, вычисляющую за линейно-логарифмическое время наиболее вероятное значение (значение, встречающееся наиболее часто) в последовательности из *n* целых чисел. *Подсказка.* Сведите задачу к сортировке.

4.2.18. *Целочисленная сортировка.* Составьте фильтр *линейного времени*, читающий со стандартного ввода последовательность целых чисел (от 0 до 99) и выводящий целые числа в отсортированном порядке. Например, если дана следующая исходная последовательность

```
98 2 3 1 0 0 0 3 98 98 2 2 2 0 0 0 2:
```

ваша программа должна вывести такую последовательность:

```
0 0 0 0 0 0 1 2 2 2 2 2 3 3 98 98 98
```

4.2.19. *Пол и потолок.* Дан отсортированный массив из *n* сравнимых ключей. Составьте функции `floor()` и `ceiling()`, возвращающие за логарифмическое время индекс наибольшего (или наименьшего) ключа, не большего (или не меньшего), чем ключ аргумента.

4.2.20. *Битонический максимум.* Массив считается битоническим, если он состоит из возрастающей последовательности ключей, непосредственно сопровождаемой убывающей последовательностью ключей. Имея битонический массив, разработайте логарифмический алгоритм для поиска индекса максимального ключа.

4.2.21. *Поиск в битоническом массиве.* Дан битонический массив из *n* уникальных целых чисел. Разработайте алгоритм логарифмического времени для определения наличия в массиве заданного целого числа.

4.2.22. *Поиск ближайшей пары.* Дан массив из *n* вещественных чисел. Составьте функцию поиска за линейно-логарифмическое время пар ближайших по значению чисел.

4.2.23. *Поиск наиболее удаленных пар.* Дан массив из *n* вещественных чисел. Составьте функцию поиска за линейное время пар чисел с наиболее дальними значениями.

4.2.24. *Две суммы.* Составьте функцию, которая получает как аргумент массив из *n* целых чисел и определяет за линейно-логарифмическое время те из них, сумма *двух* из которых равна нулю.

4.2.25. *Три суммы.* Составьте функцию, которая получает как аргумент массив из *n* целых чисел и определяет те из них, сумма *трех* из которых равна нулю. Программа должна выполниться за время, пропорциональное

$n^2 \log n$. *Дополнительное задание.* Разработайте программу, решающую задачу за квадратичное время.

4.2.26. *Преобладание.* Дан массив из n элементов, элемент является *преобладающим* (majority), если встречается больше, чем $n / 2$ раз. Составьте функцию, которая получает как аргумент массив из n строк и выявляет преобладающий элемент (если он есть) за линейное время.

4.2.27. *Общий элемент.* Составьте функцию, которая получает как аргумент три массива строк и определяет наличие строки, общей для всех трех массивов, и, если таковая есть, возвращает такую строку. Продолжительность выполнения метода должна иметь линейно-логарифмическую зависимость от общего количества строк.

4.2.28. *Наибольший свободный интервал.* Дано n временных меток о запросах файла с веб-сервера. Найдите наибольший интервал времени, за который файл никто не запрашивал. Составьте программу решения этой задачи за линейно-логарифмическое время.

4.2.29. *Беспрефиксный код.* В сжатии данных набор строк является *беспрефиксным*, если ни одна из строк не является префиксом другой. Например, набор строк `01`, `10`, `0010` и `1111` является беспрефиксным, а набор строк `01`, `10`, `0010` и `1010` — нет, поскольку `10` является префиксом `1010`. Составьте программу, читающую со стандартного ввода набор строк и определяющую, является ли набор беспрефиксным.

4.2.30. *Разделение.* Составьте функцию сортировки массива, имеющего максимум два разных значения. *Подсказка.* Создайте два указателя: один из них движется вправо, начиная с левого конца, а второй — влево, начиная с правого конца. Создайте инвариант, где все элементы слева от левого указателя равны меньшему из этих двух значений, а все элементы справа от правого указателя равны большим из этих двух значений.

4.2.31. *Нидерландский национальный флаг.* Составьте функцию сортировки массива, имеющего максимум три разных значения. (Эдсгер Вибе Дейкстра (Edsgar Dijkstra) назвал это *задачей нидерландского флага*, поскольку в результате получаются три "полосы" значений, как на флаге).

4.2.32. *Быстрая сортировка.* Составьте рекурсивную программу, сортирующую массив случайно упорядоченных различных значений. *Подсказка.* Используйте метод, подобный описанному в упр. 4.2.31. Сначала разделите массив на левую часть со всеми элементами меньше v, само v и правую

часть со всеми элементами больше v. Затем рекурсивно отсортируйте эти две части. *Дополнительное задание.* Модифицируйте свой метод (в случае необходимости), чтобы он работал правильно, если значения не обязательно различны.

4.2.33. *Обратные домены.* Составьте программу для чтения списка имен доменов со стандартного ввода и вывода имен обратных доменов в отсортированном порядке. Например, обратный домен `cs.princeton.edu` — это `edu.princeton.cs`. Это вычисление полезно при анализе веб-журнала. Для этого создайте тип данных `Domain`, реализующий специальные методы сравнения и предоставляющий обратный порядок имени домена.

4.2.34. *Локальный минимум в массиве.* Дан массив из *n* элементов типа `float`. Составьте функцию для поиска за логарифмическое время *локального минимума* (индекс `i` при условии, что `a[i] < a[i-1]` и `a[i] < a[i+1]`).

4.2.35. *Дискретное распределение.* Разработайте быстрый алгоритм для последовательного создания чисел из дискретного распределения. Дан массив `p[]` неотрицательных вещественных чисел, сумма которых равна 1. Необходимо получить индекс `i` с вероятностью `p[i]`. Сформируйте массив `s[]` накопленных сумм таким образом, чтобы `s[i]` был суммой первых `i` элементов `p[]`. Затем создайте случайное значение `s` типа `float` от 0 до 1 и используйте бинарный поиск для возвращения индекса `i`, для которого `s[i] <= r < s[i+1]`.

4.2.36. *Рифмующиеся слова.* Составьте список, который можно использовать для поиска рифмующихся слов. Используйте следующий подход.

- Читайте слова из словаря в массив строк.
- Измените на обратный порядок букв в каждом слове (например, `confound` станет `dnuofnoc`).
- Отсортируйте полученный массив.
- Снова измените на обратный порядок букв, вернув все слова в исходное состояние.
- В полученном списке слово `confound`, например, окажется рядом со словами `astound` и `surround`.

4.3. Стеки и очереди

В этом разделе вы ознакомитесь с двумя близко связанными типами данных, позволяющими манипулировать произвольно большими коллекциями элементов: *стеком* (stack) и *очередью* (queue). Стеки и очереди — это частные случаи идеи *коллекции* (collection). Коллекция элементов характеризуется пятью операциями: *создание* коллекции, *вставка* элемента, *удаление* элемента и проверка, *не пуста ли* коллекция, а также определение ее *размера* или количества элементов.

Программы этого раздела...

Программа 4.3.1. Стек (массив переменного размера) (arraystack.py) 580
Программа 4.3.2 Стек (связанный список) (linkedstack.py) 584
Программа 4.3.3. Вычисление выражения (evaluate.py) 591
Программа 4.3.4. Очередь FIFO (связанный список) (linkedqueue.py) 595
Программа 4.3.5. Модель очереди M/M/1 (mm1queue.py) 601
Программа 4.3.6. Моделирование балансировки нагрузки (loadbalance.py) 604

Когда мы вставляем элемент, наше намерение ясно. Но когда мы удаляем элемент, необходимо решить, какой именно? Вы встретитесь с различными способами ответа на этот вопрос в различных реальных ситуациях, возможно, даже не думая об этом.

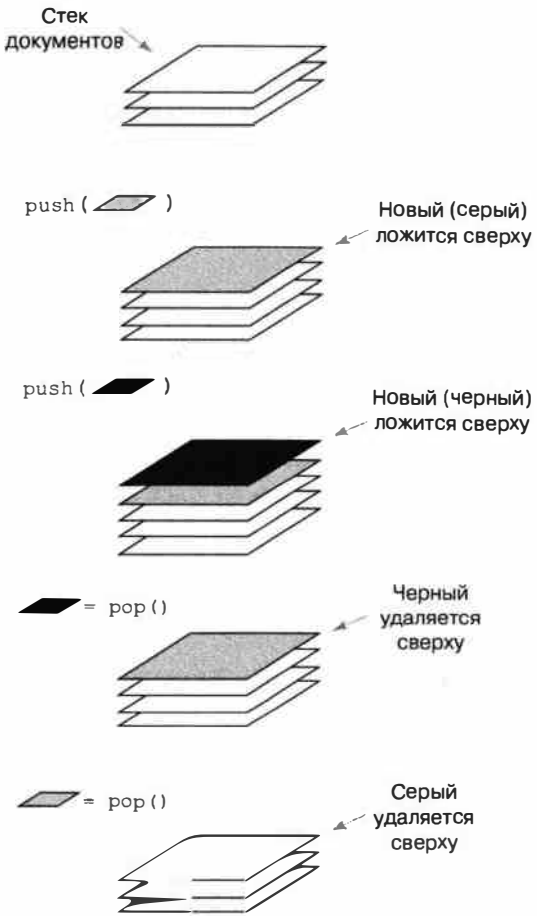
Каждый вид коллекции характеризуется правилами *удаления*. Кроме того, в зависимости от правила удаления каждый вид коллекции имеет различные реализации, отличающиеся характеристиками производительности. Например, правило *очереди* гласит, что всегда удаляется элемент, проведший в коллекции больший период времени. Эта политика известна как *первым пришел, первым вышел* (first-in first-out — FIFO), или *в порядке поступления*. Этому правилу следуют люди в очереди за билетами: они располагаются в порядке прибытия, поэтому обслуживается тот из них, кто провел в очереди больше всех времени.

Политика *стека* совершенно иная: удаляется элемент, проведший в коллекции *наименьший* период времени. Эта политика известна как *последним пришел, первым вышел* (last-in first-out — LIFO), или *в порядке обоймы*. Нечто подобное, например, наблюдается, когда вы входите и выходите из салона самолета: те, кто вошел в салон последними, покидают его перед теми, кто вошел ранее.

Стеки и очереди весьма популярны, поэтому важно ознакомиться с их фундаментальными свойствами и случаями применения. Это превосходные примеры фундаментальных типов данных, применимых для решения высокоуровневых задач программирования. Они широко используются при разработке систем и приложений, как будет продемонстрировано на нескольких примерах в этом разделе. Рассматриваемые реализации и структуры данных служат также моделями для других правил удаления, некоторые из которых мы исследуем в упражнениях в конце этого раздела.

Стеки — это коллекция на основании политики *последним пришел, первым вышел* (last-in first-out — LIFO).

Складывая корреспонденцию в стопку на столе, вы создаете стек. Новая почта оказывается сверху, и когда до нее дойдет время, прочитана она будет в первую очередь. Сейчас не так уж много бумажной почты, как в прошлом, но тот же принцип организации лежит в основе некоторых приложений, регулярно используемых на вашем компьютере. Например, многие организуют свою электронную почту как стек, когда сообщения поступают в начало списка и из начала извлекаются при чтении, предоставляя вначале самые последние поступившие. Преимущество этой стратегии в том, что мы просматриваем интересующую электронную почту как можно скорее; недостаток в том, что до некоторой старой электронной почты дело так никогда может и не дойти, если мы никогда не опустошим стек.



Операции со стеком

Вы, вероятно, встречались с другим распространенным примером стека в веб. Когда вы щелкаете на ссылке, ваш браузер отображает новую страницу (и вставляет ее в стек). Вы можете продолжить щелкать на ссылках, чтобы посетить новые страницы, но вы всегда можете вернуться к предыдущей странице, щелкнув на кнопке возврата (удалить страницу из стека). Предоставляемая стеком политика "последним пришел, первым вышел" обеспечивает именно такое поведение.

Такое использование стеков интуитивно понятно, но, возможно, не убедительно. Фактически важность стеков в компьютерных вычислениях фундаментальна, но мы отложим дальнейшие обсуждение ее применений. В настоящий момент наша задача — удостовериться в том, что вы понимаете работу и реализацию стеков.

Программисты использовали стеки с самых ранних дней компьютерных вычислений. По традиции мы называем операцию вставки в стек *помещением* (push), а операцию удаления из стека *извлечением* (pop), как указано в API далее.

API для типа данных `Stack`

Операции постоянного времени	Описание
`Stack()`	Новый стек
`s.isEmpty()`	Стек `s` пуст?
`len(s)`	Количество элементов в `s`
`s.push(item)`	Помещает `item` в `s`
`s.pop()`	Возвращает и удаляет последний добавленный элемент из `s`

Примечание. Используемое пространство должно быть пропорционально количеству элементов в стеке.

API включает базовые методы `push()` и `pop()` наряду с методом `isEmpty()`, позволяющим проверить, не пуст ли стек, и встроенную функцию `len()`, возвращающую количество элементов в стеке. При вызове для пустого стека метод `pop()` передает ошибку времени выполнения. Во избежание этого клиент должен вызвать предварительно функцию `isEmpty()`. Впоследствии мы рассматриваем две реализации этого API: `arraystack.py` и `linkedstack.py`.

Важное примечание. Когда мы включаем в API спецификации производительности, как в случае `Stack`, мы полагаем, что это *требования*. Не соответствующая им реализация могла бы быть реализацией класса `SlowStack` или `SpaceWastingStack`, но никак не `Stack`. Мы хотим, чтобы клиенты могли полагаться на гарантии производительности.

Список Python (массив переменного размера) — это реализация стека. Представление стека в виде списка Python — вполне естественная идея, но, прежде чем читать дальше, стоит самому задуматься о том, как сделали бы вы.

Использование списка Python для представления стека

Естественно, для хранения элементов стека в списке Python необходима переменная экземпляра `a[]`. Необходимо ли поддерживать элементы в порядке их вставки, где самый недавний элемент — `[0]`, второй — `[1]` и т.д.? Или элементы следует хранить в порядке, обратном их вставке? Для эффективности мы храним элементы в порядке их вставки, поскольку вставка и удаление с конца списка Python занимают постоянное время на операцию (тогда как вставка и удаление из начала требуют линейного времени на операцию).

Трудно было бы надеяться на более простую реализацию стека API, чем в программе 4.3.1 (stack.py), — все методы на одну строку! Переменная экземпляра — список Python _a[], содержит элементы стека в порядке их вставки. Для помещения элемента мы добавляем его в конец списка, используя оператор +=; для извлечения элемента используется метод pop(), возвращающий элемент с конца списка и удаляющий его; для определения размера стека используется встроенная функция len(). Эта операции обладают следующими свойствами.

- Стек содержит len(_a) элементов.
- Стек пуст, когда len(_a) возвращает 0.
- Список _a[] содержит элементы стека в порядке их вставки.
- Последний вставленный элемент стека, если он не пуст, — _a[len(_a) - 1].

Как обычно, размышление с точки зрения инвариантов такого тиа — самый простой способ удостовериться в корректности реализации.

Убедитесь, что полностью понимаете эту реализацию. Возможно, наилучший способ сделать это заключается в тщательном исследовании трассировки содержимого списка для последовательности операций pop() и push(). Клиент проверки в модуле stack.py допускает проверку с использованием произвольной последовательности операций: он вызывает метод push() для каждой введенной строки, кроме состоящей из знака "минус", для которой вызывается метод pop(). Ниже приведена трассировка для представленного ввода.

Трассировка клиента проверки arraystack.py

Ввод	Вывод	*a[]*					
		n	**0**	**1**	**2**	**3**	**4**
		0					
to		1	to				
be		2	to	be			
or		3	to	be	or		
not		4	to	be	or	not	
to		5	to	be	or	not	to
-	to	4	to	be	or	not	
be		5	to	be	or	not	be
-	be	4	to	be	or	not	
-	not	3	to	be	or		
that		4	to	be	or	that	
-	that	3	to	be	or		
-	or	2	to	be			
-	be	1	to				
is		2	to	is			

Основная черта этой реализации в том, что она использует пространство, линейно зависящее от количества элементов в стеке, и что *операции извлечения и помещения занимают амортизируемое константное время.* Эти свойства непосредственно следуют из соответствующих свойств списков Python, обсуждавшихся в конце раздела 3.1, которые в свою очередь зависят от реализации массива переменного размера Python и списков Python. Если ваш язык программирования предоставляет массивы фиксированного размера, как встроенный тип данных (но не массивы переменного размера), вы можете реализовать стек, реализовав собственный массив переменного размера (см. упр. 4.3.45).

Программа 4.3.1. Стек (массив переменного размера) (arraystack.py)

```
import sys
import stdio

class Stack:

    def __init__(self):
        self._a = []

    def isEmpty(self):
        return len(self._a) == 0

    def __len__(self):
        return len(self._a)

    def push(self, item):
        self._a += [item]

    def pop(self):
        return self._a.pop()

def main():
    stack = Stack()
    while not stdio.isEmpty():
        item = stdio.readString()
        if item != '-': stack.push(item)
        else:           stdio.write(stack.pop() + ' ')
    stdio.writeln()

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

Переменные экземпляра

`_a[]` Элементы стека

Эта программа определяет класс `Stack`, реализованный как список Python (который в свою очередь реализован с использованием массива переменного размера). Клиент проверки читает строки со стандартного ввода, встретив строку из знака "минус", извлекает и выводит строку, в противном случае помещает ее в стек.

```
% more tobe.txt
to be or not to - be - - that - - - is
% python arraystack.py < tobe.txt
to be not that or be
```

Реализация стека как связанного списка. Рассмотрим совершенно иной способ реализации стека, используя такую фундаментальную структуру данных, как

связанный список. Регулярное использование слова "список" здесь несколько сомнительно, но выбора нет — связанные списки появились намного раньше, чем Python.

Связанный список (linked list) — это рекурсивная структура данных, определенная следующим образом: *она либо пуста (null), либо ссылка на узел, имеющий ссылку на связанный список. Узел* (node) в этом определении — абстрактная сущность, способная содержать любой вид данных, в дополнение к ссылке узла, характеризующей ее роль в построении связанных списков[1]. Подобно концепции рекурсивных программ, рекурсивные структуры данных требуют поначалу усилий для понимания, но имеют огромное значение из-за простоты.

В объектно-ориентированном программировании реализация связанных списков не составляет труда. Начнем с класса для абстракции узла:

```
class Node:
    def __init__(self, item, next):
        self.item = item
        self.next = next
```

У объекта типа `Node` есть две переменные экземпляра: `item` (ссылка на элемент) и `next` (ссылка на другой объект `Node`). Переменная экземпляра `next` характеризует связную природу структуры данных. Чтобы подчеркнуть, что для структурирования данных мы используем только класс `Node`, мы не определяем никаких других методов, кроме конструктора. Мы также пропускаем начальные символы подчеркивания в именах переменных экземпляра, указывая таким образом, что они вполне доступны для кода, внешнего относительно типа данных (но все еще находящегося в пределах нашей реализации класса `Stack`).

Теперь, исходя из рекурсивного определения, мы можем представить связанный список со ссылкой на объект класса `Node`, содержащий ссылку на элемент и ссылку на другой объект класса `Node`, содержащий ссылку на элемент и ссылку на другой объект `Node`, и т.д. Последний объект `Node` в связанном списке должен указать, что он действительно последний. В языке Python для этого переменной экземпляра `next` объекта `Node` мы присваиваем значение `None`. Напомним, что `None` — это ключевое слово Python; переменная, имеющая такое значение, не ссылается ни на что.

Обратите внимание: наше определение класса `Node` соответствует нашему рекурсивному определению связанного списка. Имея класс `Node`, мы можем представить связанный список как переменную со значением либо (1) `None`, либо (2) ссылка на объект типа `Node`, поле `next` которого содержит ссылку на связанный список. Мы создаем объект типа `Node`, вызвав конструктор с двумя аргументами: ссылкой на элемент и ссылкой на следующий объект `Node` в связанном списке.

[1] Либо такое определение: связанный список — это набор узлов, состоящих из пар указателей, где первый содержит адрес хранимого объекта, а второй — адрес следующего узла или null, если такового нет. — *Примеч. ред.*

Например, чтобы создать связанный список, содержащий элементы 'to', 'be' и 'or', мы используем следующий код:

```
third =Node('or', None)
second=Node('be', third)
first =Node('to', second)
```

В результате этих операций фактически создаются три связанных списка.

- Связанный список third — это ссылка на объект Node, содержащий 'or' и None, т.е. ссылку на пустой связанный список.
- Связанный список second — это ссылка на объект Node, содержащий 'be' и ссылку на связанный список third.
- Связанный список first — это ссылка на объект Node, содержащий 'to' и ссылку на связанный список second.

Связанный список представляет собой последовательность элементов. В данном случае first представляет последовательность 'to', 'be', 'or'.

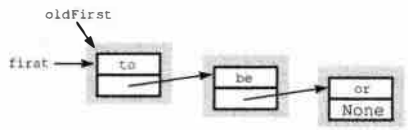
Для представления этой последовательности мы могли бы использовать и обычный массив (или список Python). Например, для представления той же последовательности строк мы могли бы использовать массив ['to', 'be', 'or']. Однако одно из главных преимуществ связанных списков — эффективная вставка (и удаление) элементов в *начало* и в *конец* последовательности. Далее мы рассмотрим код выполнения этих задач.

При трассировке кода, использующего связанные списки и другие связанные структуры, мы будем использовать визуальное представление, где:

- прямоугольник представляет каждый объект;
- в прямоугольнике указаны значения переменных экземпляра;
- для изображения ссылок используются стрелки, указывающие на объекты.

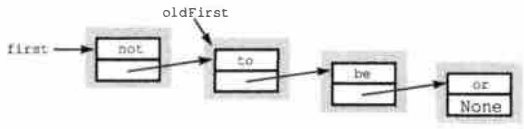
Сохранение в связанном списке ссылки на первый узел

```
Node oldFirst = first
```



Создание нового узла и установка его переменных экземпляра

```
first = Node('not', oldFirst)
```

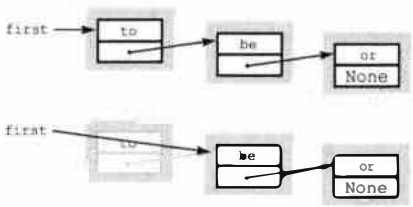


Вставка нового узла в начало связанного списка

Это визуальное представление отражает основную характеристику связанных списков. Для экономии мы используем термин *ссылка* (link) для описания ссылки (reference) на объект `Node`. Для простоты, когда элемент — строка (как в наших примерах), мы помещаем ее в прямоугольник узла (вместо более точной схемы, где узел содержит ссылку на строковый объект, хранимый вне узла). Такое визуальное представление позволяет нам сосредоточиться на ссылках.

Предположим, вы хотите *удалить* первый узел из связанного списка. Эта операция проста: достаточно присвоить `first` значение `first.next`. Обычно перед этим присвоением вы возвратили бы элемент (присвоив его некой переменной), поскольку после изменения переменной `first` любой доступ к узлу, на который она ранее ссылалась, может быть утрачен. Объект `Node`, как правило, становится "сиротой", и система управления памятью Python в конечном счете освобождает занимаемую им память.

```
first = first.next
```

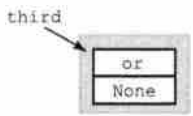


Удаление первого узла из связанного списка

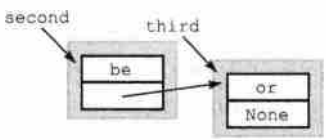
Теперь предположим, что в связанный список необходимо *вставить* новый узел. Проще всего сделать это в начале связанного списка. Например, чтобы вставить строку `'not'` в начало связанного списка с первым узлом `first`, мы сохраняем `first` в переменной `oldFirst`; создаем новый объект `Node`, переменная экземпляра `item` которого `'not'`, а переменная экземпляра `next` — `oldFirst`, и присваиваем `first` ссылку на этот новый объект `Node`.

Код вставки и удаления узла из начала связанного списка задействует лишь несколько операторов присвоения, а следовательно, занимает *константное* время (независимо от длины списка). Если вы сохраните ссылку на узел в произвольной позиции списка, то сможете использовать подобный (но чуть более сложный) код удаления узла после него или вставки перед ним, причем также постоянного времени, независимо от длины списка. Мы оставляем эти реализации как самостоятельные упражнения (см. упр. 4.3.22 и 4.3.24), поскольку вставка и удаление в начало — это единственные операции связанного списка, которые необходимо реализовать для стека.

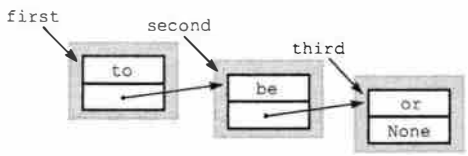
```
third = Node('or', None)
```




```
second = Node('be', third)
```




```
first = Node('to', second)
```



Соединение связанных списков

Программа 4.3.2 Стек (связанный список) (linkedstack.py)

```
import stdio

class Stack:
    def __init__(self):
        self._first = None

    def isEmpty(self):
        return self._first is None

    def push(self, item):
        self._first = _Node(item, self._first)

    def pop(self):
        item = self._first.item
        self._first = self._first.next
        return item
        class _Node:

    def __init__(self, item, next):
        self.item = item
        self.next = next

def main():
    stack = Stack()
    while not stdio.isEmpty():
        item = stdio.readString()
        if item != '-': stack.push(item)
        else:           stdio.write(stack.pop() + ' ')
        stdio.writeln()

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

Переменные экземпляра Stack

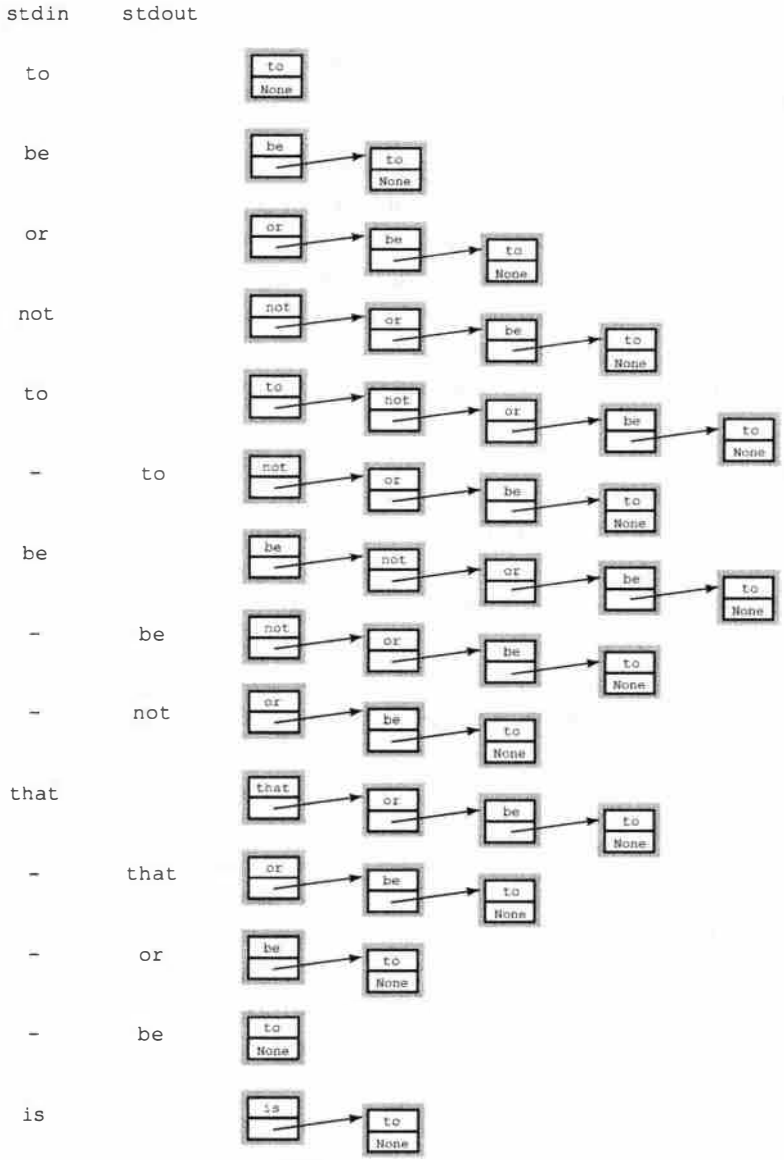
`_first`	Первый узел в списке

Переменные экземпляра Node

`item`	Элемент списка
`next`	Следующий узел в списке

Эта программа определяет класс `Stack`, реализованный как связанный список на основании закрытого класса `_Node`, чтобы представить стек как связанный список объектов `_Node`. Переменная экземпляра `first` ссылается на последний вставленный в связанный список объект `_Node`. Переменная экземпляра `next` в каждом `_Node` ссылается на следующий `_Node` (значением `next` в последнем узле является `None`). Клиент проверки тот же, что и в `arraystack.py`. Реализацию метода `__len__()` мы оставляем для упражнения 4.3.4.

```
% python linkedstack.py < tobe.tx
to be not that or be
```



Трассировка клиента проверки `linkedstack.py`

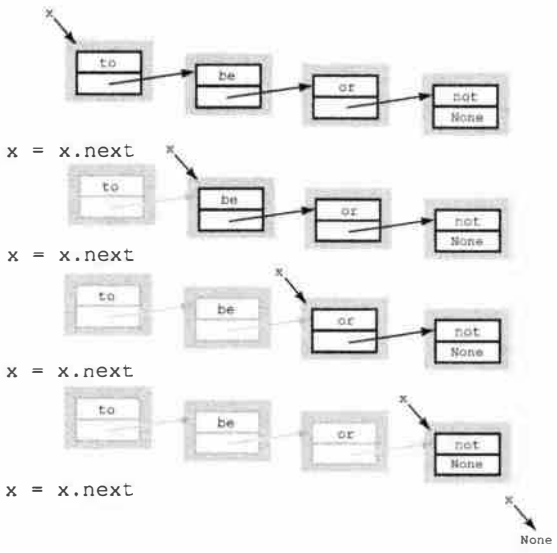
Реализация стека на базе связанного списка. Программа 4.3.2 (`linkedstack.py`) реализует стек на основании связанного списка, используя немного больше кода, чем элементарное решение на базе списка Python. Реализация будет на базе *закрытого* класса `_Node`, идентичного классу `Node`. Python позволяет определять в модуле несколько классов и использовать их вполне естественным способом. Мы делаем класс закрытым, поскольку *клиенты* типа данных `Stack` не должны

знать никаких деталей связанных списков. Как обычно, присваиваем классу имя, начинающееся с символа подчеркивания (т.е. `_Node` вместо `Node`), чтобы указать клиентам класса `Stack`, что они не должны обращаться к классу `_Node` непосредственно.

Сам класс `Stack`, определенный в программе `linkedstack.py`, имеет только одну переменную экземпляра: ссылку на связанный список, представляющий стек с элементом, вставленным в первый узел последним. Этой одной ссылки достаточно, чтобы непосредственно обратиться к элементу в вершине стека, а также предоставить доступ к остальной части элементов в стеке функциям `push()` и `pop()`.

Убедитесь, что понимаете эту реализацию, — она является прототипом нескольких реализаций на базе связанных структур, рассматриваемых далее в этой главе.

Перебор связанного списка. Хотя в программе `linkedstack.py` мы этого и не делаем, но в большинстве случаев применения связанного списка необходимо перебирать его элементы. Для этого мы сначала инициализируем индексную переменную цикла `x` ссылкой на первый узел связанного списка. Затем получаем связанный с `x` элемент, обратившись к `x.item`, а затем модифицируем `x` ссылкой на следующий узел в связанном списке, присвоив ему значением `x.next`. Процесс повторяется, пока `x` не станет `None` (что означает достижение конца связанного списка). Этот процесс называется *перебором* списка.



Перебор связанного списка

Обладая реализацией связанного списка, можно реализовать коллекции всех типов, особенно не заботясь об использовании пространства, поэтому связанные списки широко используются в программировании. Действительно, типичные реализации системы управления памятью Python основаны на поддержке связанных списков, соответствующих блокам памяти различных размеров. До широкого распространения таких высокоуровневых языков, как Python, детали управления памятью и создание связанных списков были критически важными элементами в арсенале любого программиста. В большинстве современных систем эти подробности инкапсулированы в реализациях нескольких типов данных, таких как стек, очередь, таблица

идентификаторов и набор, рассматриваемых далее в этой главе. Если вы пройдете курс по алгоритмам и структурам данных, то узнаете еще о нескольких, а также получите опыт по созданию и отладке программ, манипулирующих связанными списками. В противном случае вы можете сосредоточить свое внимание на понимании роли связанных списков в реализации этих фундаментальных типов данных.

Для стеков связанные списки существенны, поскольку они позволяют реализовать методы `push()` и `pop()` с постоянным временем выполнения в самом плохом случае, а также используют весьма немного дополнительного пространства (для ссылок). Если вы сомневаетесь в значении связанных структур, то подумайте, как вы смогли бы достичь тех же характеристик производительности стека для своего API, не используя их.

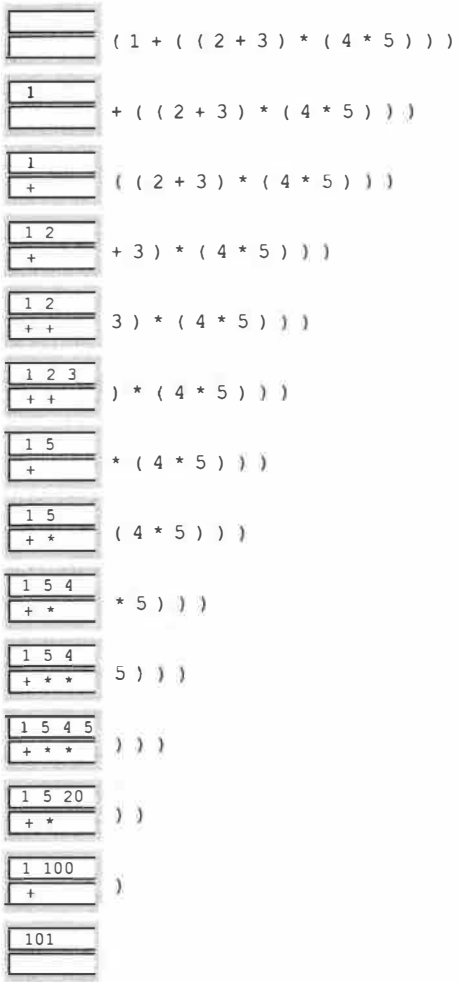
А именно: программа `arraystack.py` не реализует спецификации производительности в API, поскольку массивы переменного размера не гарантируют постоянное время выполнения каждой операции. Во многих практических ситуациях различие между амортизируемой и гарантируемой производительностью в самом плохом случае малосущественно, но подумайте о ситуации, когда некий огромный список обрабатывается в вашем телефоне, мешая позвонить, или во время подготовки самолета к взлету, или в системе управления тормозами вашего автомобиля.

Но программисты Python обычно предпочитают списки Python (массивы переменного размера), прежде всего из-за существенной поддержки пользовательских типов, таких как наш связанный список `Node`. Те же принципы относятся к коллекциям всех типов. Для реализации некоторых типов данных, более сложных, чем стеки и массивы переменного размера, также предпочтительны связанные списки, поскольку возможность получать доступ к любому элементу в массиве за постоянное время (за счет индексации) критически важна для реализации некоторых операций (см., например, `RandomQueue` в упражнении 4.3.40). Для некоторых других типов данных связанные структуры упрощают манипулирование, как будет продемонстрировано в разделе 4.4.

Применения стека. Стеки играют существенную роль в вычислениях. Если вы изучите операционные системы, языки программирования и другие дополнительные темы информатики, то узнаете, что стеки используются не только явно во многих приложениях, но и служат также основанием для выполнения программ, составленных на таких высокоуровневых языках, как Python.

Арифметические выражения. Некоторые из первых рассматривавшихся в главе 1 программ задействовали вычисление значения таких арифметических выражений, как

```
( 1+( ( 2+3 ) * ( 4 * 5 ) ) )
```



Трассировка вычисления выражения

Если вы умножите 4 на 5, добавите 3 к 2, умножите результат, а затем добавите 1, то получите значение 101. Но как Python осуществляет это вычисление? Не вдаваясь в подробности конструкции Python, мы можем обрисовать лишь основные идеи, написав программу Python, способную получить на вводе строку (выражение) и вывести значение, представленное выражением. Для простоты начнем со следующего явного рекурсивного определения: *арифметическое выражение* (arithmetic expression) — это либо число, либо левая круглая скобка, сопровождаемая арифметическим выражением, оператором, другим арифметическим выражением и правой круглой скобкой. Это упрощенное определение относится к арифметическим выражениям, *полностью заключенным в скобки*, где совершенно точно определено, какие операторы к каким операндам применять. При немного большем количестве работы можно также обработать такие выражения, как `1+2 * 3`, использующие вместо круглых скобок правила приоритета, но здесь мы избежим такого осложнения. Мы будем поддерживать знакомые парные операторы `*`, `+` и `-`, а также оператор квадратного корня `sqrt`, получающий только один аргумент. Мы легко могли бы позволить и большее количество операторов, а также большее количество видов знакомых математических выражений, включая тригонометрические, экспоненциальные и логарифмические функции, кроме других операторов по своему усмотрению. Но наше основное внимание сосредоточено здесь на понимании интерпретации набора круглых скобок, операторов и чисел, чтобы обеспечить выполнение низкоуровневых арифметических операций, доступных на любом компьютере, в надлежащем порядке.

Вычисление арифметического выражения. Так как же именно преобразовать арифметическое выражение (строку символов) в представляемые ей значения? Для решения этой задачи Эдсгер Дейкстра разработал в 1960-х годах

замечательно простой алгоритм, использующий два стека (один для операндов и один для операторов). Выражение состоит из круглых скобок, операторов и операндов (чисел). Двигаясь слева направо и извлекая сущности по одной, мы манипулируем стеками согласно следующим четырем возможным случаям.

- Поместить *операнд* в стек операндов.
- Поместить *оператор* в стек операторов.
- Проигнорировать *левую* круглую скобку.
- Встретив *правую* круглую скобку, извлечь оператор, извлечь необходимое количество операндов и поместить в стек операндов результат применения данного оператора к данным операндам.

По завершении обработки правой круглой скобки в стеке остается одно значение, являющееся значением выражения. Поначалу этот метод может показаться весьма таинственным, однако достаточно просто убедиться, что он вычисляет правильное значение: каждый раз, встретившись с подвыражением, состоящим из двух операндов, отделенных оператором и окруженных круглыми скобками, алгоритм оставляет результат выполнения этой операции с этими операндами в стеке операндов. Результат получится тот же, как будто во вводе было это значение, а не подвыражение. Таким образом, мы можем рассматривать замену подвыражения значением как получение выражения, возвращающего тот же результат. Мы можем применять этот аргумент снова и снова, пока не получим одно значение. Например, для всех этих выражений алгоритм вычисляет то же значение:

```
( 1+( ( 2+3 ) * ( 4 * 5 ) ) )
( 1+( 5 * ( 4 * 5 ) ) )
( 1+( 5 * 20 ) )
( 1+100 )
101
```

Программа 4.3.3 (`evaluate.py`) является реализацией данного алгоритма. Этот код — простой пример *интерпретатора*: программы, интерпретирующей вычисление, определенное заданной строкой, и выполняющей это вычисление для достижения результата. *Компилятор* — программа, преобразующая строку в машинный код низшего уровня, способный решить задачу. Это преобразование сложнее процесса пошагового преобразования, используемого интерпретатором, но базовый механизм тот же. Первоначально Python был основан на использовании интерпретатора, но теперь он включает компилятор, преобразующий арифметические выражения (точнее, программы Python) в код для *виртуальной машины Python* — программной системы, которую можно установить на реальном компьютере.

Языки программирования на базе стека. Алгоритм Дейкстры для двух стеков вычисляет то же значение этого выражения, что и в нашем примере:

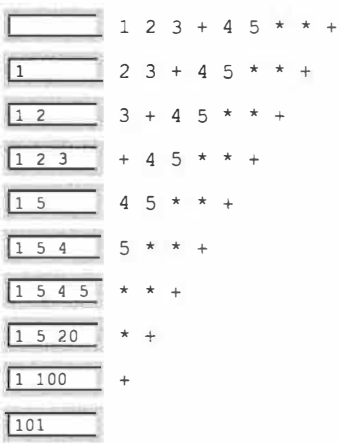
```
( 1 ( ( 2 3+) ( 4 5 * ) * )+)
```

Другими словами, мы можем поместить каждый оператор *после* его двух операндов, а не *между* ними. В таком выражении каждая правая круглая скобка непосредственно следует за оператором, поэтому мы можем игнорировать *оба* вида круглых скобок и записать выражение следующим образом:

```
1 2 3+4 5 * * +
```

Это *обратная польская нотация* (reverse Polish notation), или *постфиксная* нотация (postfix). Для вычисления постфиксных выражений мы используем один стек (см. упр. 4.3.13). Двигаясь слева направо и извлекая сущности по одной, мы манипулируем стеком согласно только двум возможным случаям.

- Операнды помещаются в стек операндов.
- Встретив оператор, извлекаем необходимое количество операндов и помещаем в стек операндов результат применения оператора к этим операндам.



Трассировка постфиксного вычисления

Этот процесс также оставляет в стеке одно значение, являющееся значением выражения. Это представление настолько просто, что некоторые языки программирования, такие как Forth (научный язык программирования) и PostScript (язык описания страниц, используемый в большинстве принтеров), используют явные стеки. Например, строка `1 2 3+4 5 * *+` является вполне допустимой программой для Forth и PostScript, оставляющей после выполнения в стеке значение `101`. Поклонники этих и подобных языков программирования на базе стека предпочитают именно их, поскольку они очень просты для многих типов вычисления. Действительно, даже виртуальная машина Python основана на стеке.

Программа 4.3.3. Вычисление выражения (evaluate.py)

```
import stdio
import math
from arraystack import Stack

ops = Stack()
values = Stack()

while not stdio.isEmpty():
    token = stdio.readString()
    if   token == '+': ops.push(token)
    elif token == '-': ops.push(token)
    elif token == '*': ops.push(token)
    elif token == 'sqrt': ops.push(token)
    elif token == ')':
        op = ops.pop()
        value = values.pop()
        if   op == '+':    value = values.pop() + value
        elif op == '-':    value = values.pop() - value
        elif op == '*':    value = values.pop() * value
        elif op == 'sqrt': value = math.sqrt(value)
        values.push(value)
    elif token != '(':
        values.push(float(token))
stdio.writeln(values.pop())
```

ops	Стек операторов
values	Стек операндов
token	Текущая лексема
value	Текущее значение

Этот клиент класса Stack читает со стандартного ввода полностью заключенное в скобки числовое выражение, использует алгоритм Дейкстры с двумя стеками для вычисления и выводит результат. Он иллюстрирует базовый вычислительный процесс: интерпретация строки как программы и выполнение этой программы для вычисления желаемого результата. Выполнение программы Python является не более чем усложненной версией того же процесса.

```
% python evaluate.py
( 1+( ( 2+3 ) * ( 4 * 5 ) ) )
101.0

% python evaluate.py
( ( 1+sqrt ( 5.0 ) ) * 0.5 )
1.618033988749895
```

Абстракция вызова функции. Вы, возможно, обратили внимание на шаблон в формальных трассировках повсюду в этой книге. Когда поток управления входит

Использование стека для поддержки вызова функций

в функцию, Python создает параметрические переменные функции поверх других переменных, которые могли бы уже существовать. По мере выполнения функции Python создает локальные переменные функции, и снова поверх других переменных, которые могли бы уже существовать. Когда управление потоком возвращается из функции, Python удаляет локальные и параметрические переменные этой функции. В этом смысле Python создает и удаляет параметрические и локальные переменные, как в стеке: переменные, созданные последними, удаляются первыми.

Тем не менее большинство программ использует стеки неявно, поскольку они обеспечивают естественный способ реализации вызовов функции следующим образом: в любой момент времени выполнения функции определяет ее состояние как значение всех ее переменных и указатель на следующую инструкцию для выполнения. Одной из фундаментальных характеристик компьютерных сред является то, что каждое вычисление полностью определяется его состоянием (и значением его входных данных). В частности, система может приостановить вычисление, сохранив его состояние, а затем возобновить его, восстановив состояние. Если вы пройдете курс по операционным системам, то изучите детали этого процесса, поскольку он критически важен для поведения большинства компьютеров, которое мы считаем само собой разумеющимся (например, переключение с одного приложения на другое — это просто вопрос сохранения и восстановления состояния). В настоящее время естественный способ реализации абстракции вызова функции, используемый почти всеми современными средами программирования, заключается в использовании стека. При вызове функции состояние просто помещается в стек. При выходе из вызова функции состояние извлекается из стека для восстановления значения всех переменных в их состоянии

перед вызовом функции, замены в выражении вызова функции ее возвращаемым значением (если оно есть) и возобновления выполнения со следующей инструкции (расположение которой было сохранено как часть состояния вычисления). Этот механизм срабатывает всякий раз, когда одна функция вызывает другую, даже рекурсивно. Действительно, если тщательно обдумать процесс, то можно заметить, что это, по существу, тот же процесс, который мы только что подробно исследовали на примере вычисления выражения. Программа — это просто сложное выражение.

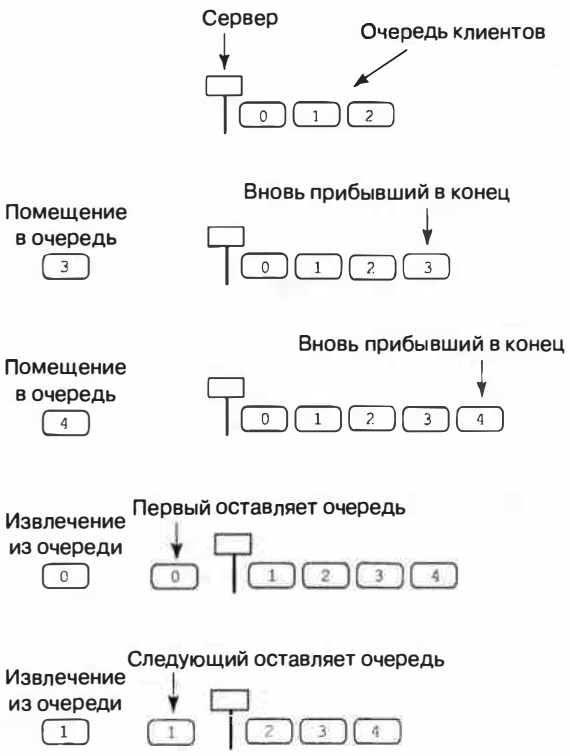
Стек — это фундаментальная компьютерная абстракция. Стеки используются для вычисления выражений, реализации абстракций вызова функций и других базовых задач с самого начала появления компьютеров. Другие задачи мы исследуем в разделе 4.4. Стеки интенсивно используются во многих областях информатики, включая проектирование алгоритмов, операционные системы, компиляторы и многие другие вычислительные приложения.

Очередь FIFO (FIFO queue), или просто *очередь* (queue), — это коллекция на основании политики "*первым пришел, первым вышел*" (first-in-first-out — FIFO).

Политика выполнения задач в порядке их поступления — это именно то, с чем мы зачастую сталкиваемся в повседневной жизни: от людей, ожидающих у входа в театр, и автомобилей, ждущих оплаты за проезд, до задач, ожидающих выполнения на вашем компьютере.

Основополагающий принцип любого обслуживания — понятие справедливости. Первая приходящая на ум мысль, когда большинство людей размышляют о справедливости, — сначала следует обслужить того, кто ждал дольше всех. Это в точности принцип FIFO, поэтому очередь играет центральную роль в многочисленных приложениях. Очередь — вполне естественная модель для очень многих повседневных явлений (их свойства были подробно изучены еще до появления компьютеров).

Как обычно, мы начинаем с озвучивания API и по традиции называем операцию вставки *помещением в очередь* (enqueue), а удаления — *извлечением из очереди* (dequeue), как указано в API ниже.



Типичное очереди FIFO

API для типа данных `Queue`

Операции постоянного времени	Описание
`Queue()`	Новая очередь
`q.isEmpty()`	Очередь `q` пуста?
`len(q)`	Количество элементов в `q`
`q.enqueue(item)`	Помещает `item` в `q`
`q.dequeue()`	Возвращает и удаляет первый добавленный элемент из `q`

Примечание. Используемое пространство должно быть пропорционально количеству элементов в очереди.

Для разработки реализации, где операции занимают постоянное время, а связанная с очередью память увеличивается и уменьшается согласно количеству элементов в очереди, применяя наше знание стеков, мы можем использовать либо списки Python (массивы переменного размера), либо связанные списки. Подобно стекам, каждая из этих реализаций представляет классическое упражнение по программированию. Вы можете самостоятельно обдумать решение и реализацию этих задач, прежде чем читать далее.

Реализация связанного списка. Для реализации очереди при помощи связанного списка мы будем сохранять элементы в порядке их поступления (в порядке, обратном использованному в программе `linkedstack.py`). Реализация функции `dequeue()` та же, что и реализация функции `pop()` в программе `linkedstack.py` (сохраняем элемент в первом узле, возвращаем из очереди первый сохраненный элемент и удаляем его узел). Реализация функции `enqueue()`, однако, немного сложней: как добавить узел в конец связанного списка? Для этого необходима ссылка последнего узла списка, поскольку ее следует изменить так, чтобы она ссылалась на новый узел, содержащий добавляемый элемент. В стеке единственная переменная экземпляра — это ссылка на *первый* узел в связанном списке; ее информации в принципе вполне достаточно для перебора всех узлов в связанном списке и достижения его конца. При длинных списках такое решение малопривлекательно. Вполне разумная альтернатива — хранить вторую переменную экземпляра, всегда ссылающуюся на *последний* узел связанного списка. Добавление дополнительной переменной экземпляра, требующей поддержки, — не такое простое дело, особенно в коде связанного списка, поскольку каждый метод, изменяющий связанный список, нуждается в коде, проверяющем, нужно ли изменять эту переменную (и вносящем необходимые изменения). Например, удаление первого узла в связанном списке могло бы повлечь изменение ссылки на последний узел, а когда в связанном списке есть только один узел, то это и первый, и последний! Точно так же необходима дополнительная проверка при добавлении нового узла в пустой связанный список. Подобные подробности весьма затрудняют отладку кода связанного списка.

Программа 4.3.4. Очередь FIFO (связанный список) (linkedqueue.py)

```
class Queue:
    def __init__(self):
        self._first = None
        self._last = None
        self._n = 0

    def isEmpty(self):
        return self._first is None

    def enqueue(self, item):
        oldLast = self._last
        self._last = _Node(item, None)
        if self.isEmpty(): self._first = self._last
        else:              oldLast.next = self._last
        self._n += 1

    def dequeue(self):
        item = self._first.item
        self._first = self._first.next
        if self.isEmpty(): self._last = None
        self._n -= 1
        return item

    def __len__(self):
        return self._n

class _Node:
    def __init__(self, item, next):
        self.item = item
        self.next = next
```

Переменные экземпляра `Queue`

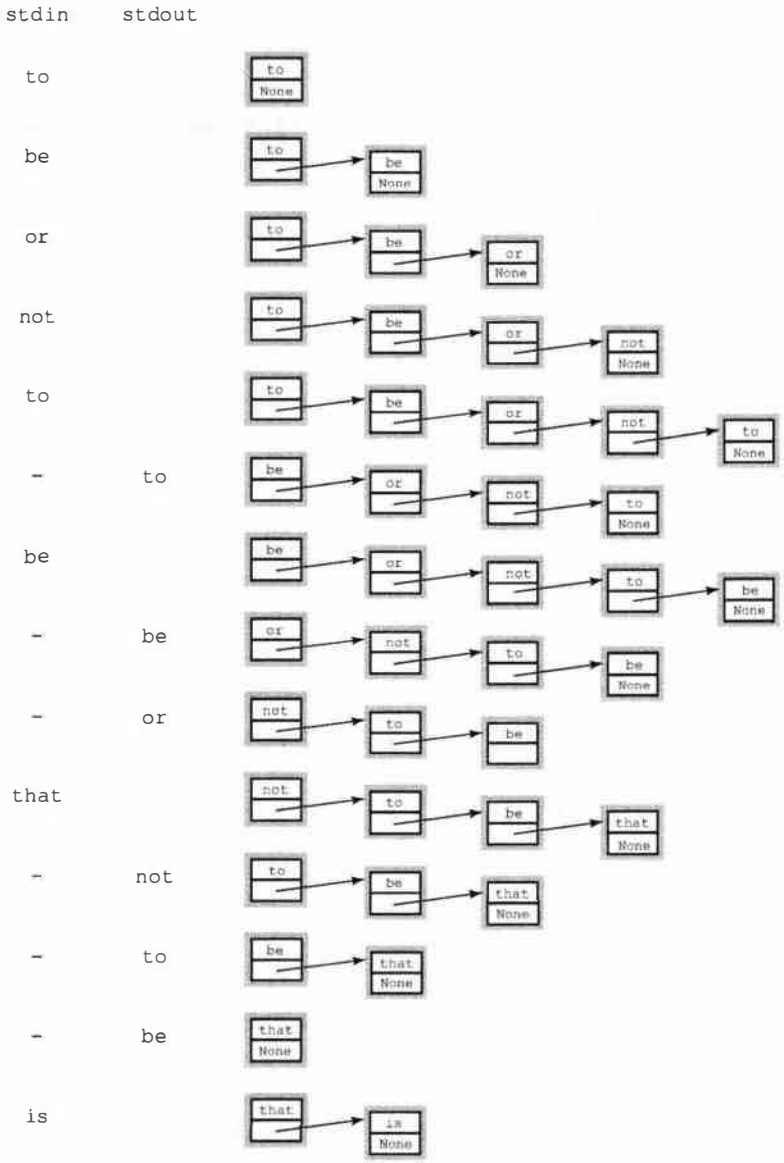
`_first`	Первый узел в списке
`_last`	Последний узел в списке
`_n`	Количество элементов

Переменные экземпляра `Node`

`item`	Элемент списка
`next`	Следующий узел в списке

Эта программа определяет класс `Queue`, реализованный на базе связанного списка. Реализация очень похожа на нашу реализацию стека на базе связанного списка (программа 4.3.2): функция `dequeue()` — почти та же, что и функция `pop()`, но функция `enqueue()` связывает новый элемент с концом списка, а не с началом, как функция `push()`. Для этого используется дополнительная переменная экземпляра, ссылающаяся на последний узел в списке. Клиент проверки подобен прежнему (читает строки со стандартного ввода, встретив строку из знака "минус", извлекает и выводит строку, а в противном случае помещает ее в стек) и здесь опущен.

```
% python linkedqueue.py < tobe.txt
to be or not to be
```

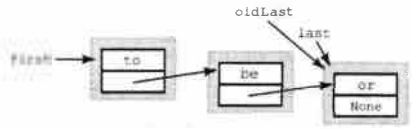


Трассировка клиента проверки `linkedqueue.py`

Программа 4.3.4 (`linkedqueue.py`) является реализацией класса `Queue` на базе связанного списка, обладающая теми же свойствами производительности, что и `Stack`: все методы — операции постоянного времени, а расход пространства линейно зависит от количества элементов в очереди.

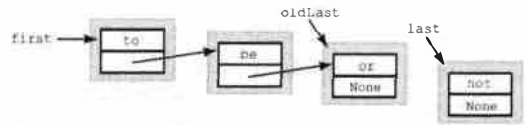
Сохраните ссылки на последний узел в связанном списке

```
oldLast = last
```



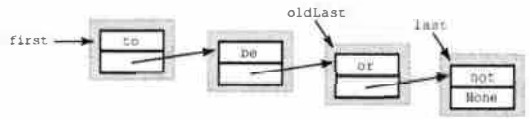
Создание нового узла

```
last = Node('not', None)
```



Связывание нового узла с концом списка

```
oldLast.next = last
```



Вставка нового узла в конец связанного списка

Реализация на базе массива переменного размера. Реализацию очереди FIFO вполне также можно разработать на основании массива переменного размера, причем с теми же характеристиками производительности, что и у стека в программе 4.3.1 (`arraystack.py`). Эта реализация — достойное и классическое упражнение по программированию, которое вы можете выполнить впоследствии (см. упр. 4.3.46). Было бы заманчиво использовать однострочные обращения к методам списка Python, как в программе 4.3.1. Однако методы вставки и удаления элементов в начало списка Python не будут соответствовать требованиям, поскольку они занимают линейное время (см. упр. 4.3.16-4.3.17). Например, если *a* — это список из *n* элементов, то такие операции, как `a.pop(0)` и `a.insert(0, item)`, занимают время, пропорциональное *n* (а не постоянное амортизируемое время). Напомним, что реализация, не гарантирующая заданной производительности, могла бы называться `SlowQueue`, но никак не `Queue`.

Случайная очередь. Несмотря на широкую популярность, в стратегиях FIFO и LIFO нет ничего священного. Иногда имеет смысл рассмотреть и другие правила удаления элементов. Один из важнейших — тип данных, метод `dequeue()` которого удаляет *случайный* (random) элемент (выборка без замены), а метод `sample()` возвращает случайный элемент, не удаляя его из очереди (выборка с заменой). Такие действия точно соответствуют многочисленным случаям, некоторые из которых уже рассматривались, начиная с программы 1.4.1 (`sample.py`). Используя список Python (массив переменного размера), реализовать метод `sample()` довольно

просто, для реализации метода `dequeue()` (обмен случайного элемента с последним элементом перед его удалением) мы можем использовать ту же идею, что и в программе 1.4.1. Мы назовем этот тип данных `RandomQueue` (см. упр. 4.3.40). Обратите внимание на то, что это решение зависит от использования представления массива переменного размера (список Python): невозможно получить доступ к случайному элементу в связанном списке за постоянное время, поскольку для этого придется начать с начала списка и перебирать его по одному до достижения необходимого. У списка Python (массива переменного размера) все операции занимают амортизируемое постоянное время.

API стека, очереди и случайной очереди чрезвычайно похожи и отличаются только именами классов и методов (выбранных произвольно). Размышление об этой ситуации позволяет улучшить свое понимание важных вопросов, связанных с типами данных, представленных в разделе 3.3. Истинные различия между этими типами данных кроются в семантике операции *удаления* — какой именно элемент должен быть удален? Различия между стеками и очередями воплощаются в англоязычных описаниях того, что они делают. Эти различия родственны различиям между методами `math.sin(x)` и `math.log(x)`, но мы могли бы усилить их, явно сформулировав в формальном описании стеков и очередей (таким же образом, как и математические описания функций синуса и логарифма). Однако описания точней, чем первым пришел, первым вышел, или последним пришел, первым вышел либо случайным вышел, не настолько просто. Для начала, какой язык вы использовали бы для такого описания? Английский? Python? Математической логики? Проблема описания поведения программы известна как *проблема спецификации*, и она непосредственно ведет к глубинным проблемам информатики. Одна из причин нашего акцента на четком и кратком коде в том, что код сам может служить спецификацией для простых типов данных, таких как стеки и очереди.

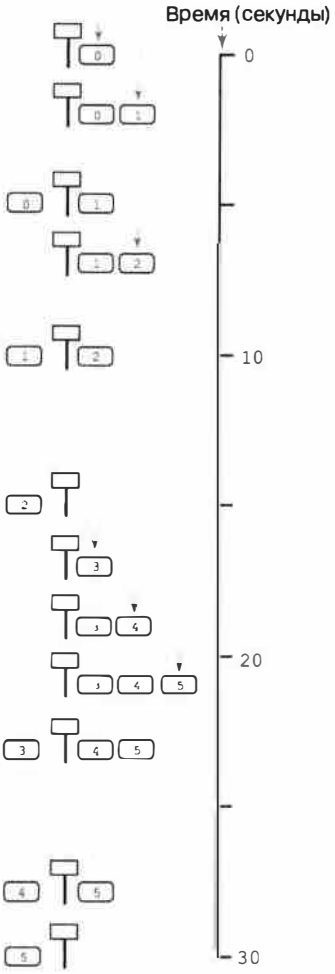
Применение очереди. В прошлом столетии очереди FIFO были точными и полезными моделями в широком разнообразии реальных случаев, от моделей производственных процессов до моделей трафика телефонных сетей. Такая область математики, как *теория очередей*, использовалась с большим успехом, помогая понимать и контролировать сложные системы всех видов. Очереди FIFO играют также важную роль в вычислениях. Вы часто встречаетесь с очередями, используя свой компьютер: очередь может содержать музыкальные произведения в списке, документы в очереди на печать или события в игре.

Вероятно, наиболее существенное применение очереди — это сам Интернет, в основе которого лежит огромное количество сообщений, двигающихся по огромному количеству очередей, имеющих различные свойства и взаимосвязанных сложными способами. Понимание и контроль такой сложной системы подразумевает надежные реализации абстракции очереди, применение

результатов математической теории очередей и исследования моделей. Далее мы рассмотрим классический пример этого процесса.

Очередь M / M / 1. Одной из важнейших моделей организации очередей является *очередь M / M / 1*, точно описывающая множество реальных ситуаций, таких как очередь автомобилей на оплату за проезд или пациентов перед кабинетом врача. *M* — это сокращение от *Марков* или *memoryless* (без запоминания) и означает, что и прибытие, и обслуживание являются *процессами Пуассона*: и интервал между прибытиями, и период обслуживания повинуются *экспоненциальному распределению* (см. упр. 2.2.12), а 1 означает, что есть только один обслуживающий. Параметрами очереди *M / M / 1* являются: *частота прибытий* (arrival rate) λ (например, количество автомобилей, прибывающих за минуту к пункту оплаты) и *частота обслуживания* (service rate) μ (например, количество автомобилей, оплачивающих проезд за минуту). Они характеризуются тремя свойствами.

- Есть один обслуживающий — очередь FIFO.
- Интервалы между прибытиями в очередь подчиняются экспоненциальному распределению с частотой λ в минуту.
- Периоды обслуживания при непустой очереди подчиняются экспоненциальному распределению с частотой μ в минуту.



	Прибытие	Отправление	Ожидание
0	0	5	5
1	2	10	8
2	7	15	8
3	17	23	6
4	19	28	9
5	21	30	9

Очередь M / M / 1

Среднее время между прибытиями — $1/\lambda$ минут, а среднее время между обслуживаниями (когда очередь не пуста) — $1/\mu$ минут. Таким образом, очередь не будет расти безгранично, если $\mu > \lambda$; в противном случае клиенты поступают и оставляют очередь в процессе весьма интересного динамического процесса.

Анализ. В практических случаях людей интересует результат влияния параметров λ и μ на три разных свойствах очереди. Если вы клиент, то можете захотеть

узнать период ожидания в системе; если разрабатываете систему, то могли бы захотеть узнать вероятное количество клиентов в системе, или нечто более сложное, например, когда размер очереди, вероятнее всего, превысит заданный максимальный размер. Для простых моделей теория вероятности возвращает к формулам, выражающим эти значения как функции от λ и μ. Для очереди $M/M/1$ известно, что

- среднее количество клиентов в системе L составляет $\lambda/(\mu-\lambda)$;
- среднее время, проводимое клиентом в системе W, составляет $1/(\mu-\lambda)$.

Например, если автомобили поступают с частотой $\lambda=10$ в минуту, а частота обслуживания $\mu=15$ в минуту, то средние количество автомобилей в системе составит 2, а среднее время, проводимое клиентом в системе, — 1/5 минуты, или 12 секунд. Эти формулы подтверждают, что время ожидания (и длина очереди) растет без границ, если λ приближается μ. Они также подчиняются общему правилу, известному как *закон Литтла* (Little's law): среднее количество клиентов в системе составляет λ раз, среднее время, проводимое клиентом в системе ($L=\lambda W$) для многих типов очередей.

Моделирование. Программа 4.3.5 (`mm1queue.py`) — это клиент класса `Queue`, который можно использовать для проверки различных математических результатов. Это простой пример *моделирования на основании событий* (event-based simulation): мы создаем *события* (event), имеющие место в определенное время, и корректируем наши структуры данных в соответствии с этими событиями, моделируя происходящее при их наступлении. В очереди $M/M/1$ есть два вида событий: *прибытие* клиента и *обслуживание* клиента. В свою очередь, мы поддерживаем две переменные:

- `nextService` — время следующего обслуживания;
- `nextArrival` — время следующего прибытия.

Для моделирования события прибытия мы помещаем в очередь переменную `nextArrival` типа `float` (время прибытия); для моделирования услуги извлекаем из очереди значение типа `float`, вычисляем время ожидания `wait` (время завершения обслуживания минус время, когда клиент поступил и встал в очередь) и добавляем время ожидания в гистограмму (см. программу 3.2.3).

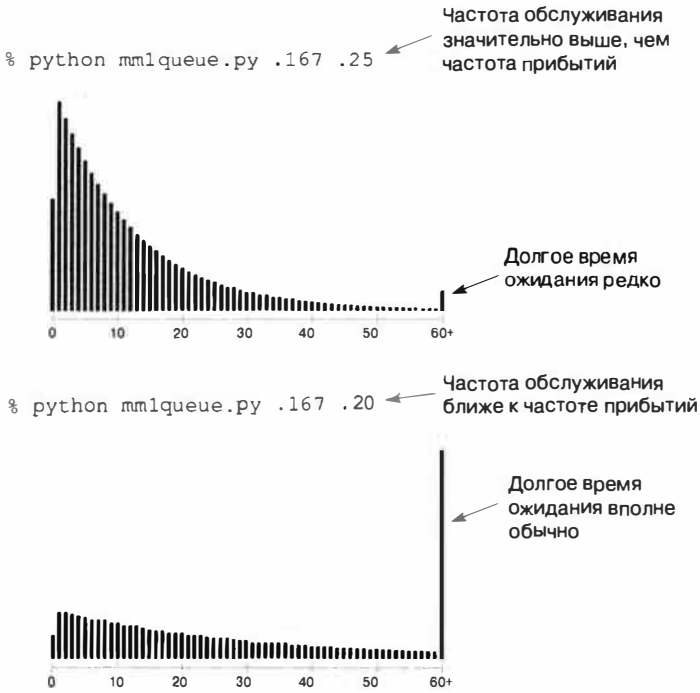
Программа 4.3.5. Модель очереди M / M / 1 (mm1queue.py)

```
import sys
import stddraw
import stdrandom
from linkedqueue import Queue
from histogram import Histogram

lambd = float(sys.argv[1])
mu = float(sys.argv[2])

histogram = Histogram(60 + 1)
queue = Queue()
stddraw.setCanvasSize(700, 500)

nextArrival = stdrandom.exp(lambd)
nextService = nextArrival + stdrandom.exp(mu)

while True:
    while nextArrival < nextService:
        queue.enqueue(nextArrival)
        nextArrival += stdrandom.exp(lambd)

    arrival = queue.dequeue()
    wait = nextService - arrival

    stddraw.clear()
    histogram.addDataPoint(min(60, int(round(wait))))
    histogram.draw()
    stddraw.show(20.0)
    if queue.isEmpty():
        nextService = nextArrival + stdrandom.exp(mu)
    else:
        nextService = nextService + stdrandom.exp(mu)
```

`lambd`	Частота прибытий λ
`mu`	Частота обслуживания μ
`histogram`	Гистограмма
`queue`	Очередь *M* / *M* / 1
`nextArrival`	Время следующего прибытия
`nextService`	Время следующего завершения обслуживания
`arrival`	Время прибытия следующего клиента на обслуживание
`wait`	Время в очереди

Эта программа получает аргументы командной строки `lambd` и `mu` типа `float` и моделирует очередь *M* / *M* / 1 с частотой прибытий `lambd` и частотой обслуживания `mu`, создавая динамическую гистограмму для времени ожидания. Время отслеживается по двум переменным: `nextArrival` и `nextService`, а также одиночной переменной `queue` типа `float`. Значение каждого элемента в очереди — (моделируемое) время постановки в очередь. Пример созданного программой графика представлен на следующей странице.

Полученный после большого количества испытаний график является характеристикой системы организации очередей $M/M/1$. С практической точки зрения одна из важнейших характеристик процесса, которую позволяет обнаружить запуск программы `mm1queue.py` для разных значений параметров λ и μ, заключается в том, что среднее время, проводимое клиентом в системе (и среднее количество клиентов в системе), может существенно увеличиться, когда частота обслуживания приближается к частоте прибытий. Когда частота обслуживания высока, у гистограммы есть видимый хвост, где частота клиентов, имеющих данное время ожидания, уменьшается до незначительной продолжительности. Но когда частота обслуживания близка к частоте прибытий, хвост увеличивается, и большинство значений по сути находится в хвосте гистограммы. Таким образом, доминирует частота клиентов, имеющих самое высокое время ожидания.



Пример запуска `mm1queue.py`

Как и во многих других рассмотренных ранее случаях, использование модели для проверки хорошо понятной математической модели является отправной точкой для изучения более сложных ситуаций. В практических случаях у нас может быть несколько очередей, несколько обслуживающих, многоступенчатые услуги, пределы на длину очереди и много других ограничений. Кроме того, распределения времени между двумя прибытиями и периодом обслуживания не могут быть выражены математически. В таких ситуациях у нас может не быть никакого

иного способа, кроме как прибегнуть к моделированию. Для системных разработчиков весьма характерно создавать компьютерные модели системы с очередями (такой, как `mm1queue.py`) и использовать их для корректировки параметров проекта (таких, как частота обслуживания), для правильной реакции на внешние условия (такие, как частота прибытий).

Распределение ресурсов. Теперь рассмотрим другое приложение, иллюстрирующее упомянутые структуры данных. Система с *совместными ресурсами* (resource-sharing) задействует большое количество независимых *серверов* (server), желающих совместно использовать ресурсы. Каждый сервер согласен поддерживать очередь совместно используемых элементов и централизовано распределять их между серверам (и сообщать пользователям, где они могут быть найдены). Например, элементы могли бы быть музыкой, фотографиями или видео, совместно используемыми большим количеством пользователей. Конкретно речь идет о миллионах элементов и тысячах серверов.

Мы рассмотрим программу централизованного распределения элементов, игнорируя динамику удаления элементов из системы, добавления и удаления серверов и т.д.

Если мы используем *циклическую* (round-robin) политику, то серверы перебираются по очереди и мы получаем сбалансированное распределение, но столь полный контроль над ситуацией редко возможен для распространителя. Например, вполне может быть множество независимых распространителей, поэтому ни у одного из них может не быть актуальной информации о серверах. Соответственно такие системы зачастую используют *случайную* (random) политику, где предоставление осуществляется на основании случайного выбора. Еще лучшая политика подразумевает случайную выборку серверов и предоставление нового элемента тому, у которого меньшие всего элементов. Для небольших очередей различие между этими политиками незначительно, но в системе с миллионами элементов на тысячах серверов различия могут оказаться весьма существенными, поскольку у каждого сервера есть фиксированный объем ресурсов, которые они могут посвятить этому процессу. Действительно, подобные системы используются в аппаратных средствах Интернета, где некоторые очереди могли бы быть реализованы на аппаратных средствах специального назначения, поэтому длина очереди непосредственно преобразуется в дополнительную стоимость оборудования. Но насколько большой должна быть выборка?

Программа 4.3.6 (`loadbalance.py`) моделирует политику выборки, и мы можем использовать ее для изучения этого вопроса. Программа использует тип данных `RandomQueue`, разработанный для облегчения понимания программы и экспериментов. Модель поддерживает случайную очередь очередей и осуществляет вычисления во внутреннем цикле, где каждый новый запрос к службе перенаправляется на наименьшую очередь из выборки. Для случайной выборки очередей

используется метод `sample()` класса `RandomQueue`. Конечный результат удивителен: выборка размером 2 приводит к почти совершенному балансу, поэтому в больших выборках нет никакого смысла.

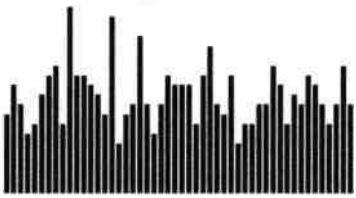
Программа 4.3.6. Моделирование балансировки нагрузки (`loadbalance.py`)

```
import sys
import stddraw
import stdstats
from linkedqueue import Queue
from randomqueue import RandomQueue

m = int(sys.argv[1])
n = int(sys.argv[2])
t = int(sys.argv[3])

servers = RandomQueue()
    for i in range(m):
servers.enqueue(Queue())

for j in range(n):
    best = servers.sample()
    for k in range(1, t):
        queue = servers.sample()
        if len(queue) < len(best):
            best = queue
    best.enqueue(j)

lengths = []
while not servers.isEmpty():
    lengths += [len(servers.dequeue())]

stddraw.setYscale(0, 2.0*n/m)
stdstats.plotBars(lengths)
stddraw.show()
```

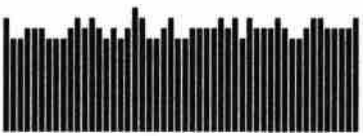
`m`	Количество серверов
`n`	Количество элементов
`t`	Размер выборки
`servers`	Очереди
`best`	Самая короткая в выборке
`lengths`	Длины очередей

Этот клиент классов `Queue` и `RandomQueue` моделирует процесс присвоения `n` элементов набору из `m` серверов. Запросы помещаются в самую короткую из выборок `t` очередей, выбранных наугад.

```
% python loadbalance.py 50 500 1
```



```
% python loadbalance.py 50 500 2
```



Мы подробно рассмотрели проблемы, связанные со временем и использованием памяти простых реализаций API стека и очереди, не только потому, что эти типы данных важны и полезны, но также и потому, что вы, вероятно, встретитесь с теми же проблемами в контексте реализаций собственных типов данных.

Нужно ли использовать стек, очередь FIFO или случайную очередь при разработке клиента, поддерживающего коллекции данных? Ответ на этот вопрос зависит от высокоуровневого анализа клиента, позволяющего определить, какая из политик (LIFO, FIFO или случайная) является наиболее подходящей.

Нужно ли использовать связанный список или массив переменного размера для структурирования данных? Ответ на этот вопрос зависит от низкоуровневого анализа характеристик производительности. У *связанного списка* есть преимущество — вы можете добавлять элементы с обоих концов, но есть и недостаток — вы не можете обращаться к произвольному элементу за постоянное время. У *массива переменного размера* есть преимущество — вы можете обратиться к любому элементу за постоянное время, но постоянная продолжительность операций вставки и удаления возможна только для одного конца (и это постоянное время только в амортизируемом смысле). Предпочтительность каждой структуры данных зависит от конкретной ситуации, и вы, вероятно, встретите их в большинстве сред программирования.

Пристальное внимание к производительности гарантирует, что заявленное в API наших типов `Stack` и `Queue` не должно считаться само собой разумеющимся. Они лишь первые в ряду типов данных, составляющих основу нашей вычислительной инфраструктуры. Следующий раздел посвящен более важной теме — таблице идентификаторов, а о других вы узнаете, когда пройдете курс по структурам данных и алгоритмам. К настоящему времени вы знаете, что учиться использовать новый тип данных — это как учиться кататься на велосипеде (или реализовать программу `helloworld.py`): сначала все кажется совершенно непонятным, пока не сделаешь это впервые, а потом это становится второй натурой. И изучение, и реализация нового типа данных при разработке новой структуры данных является сложным, творческим и интересным процессом. Как вы увидите в следующих упражнениях, стеки, очереди, связанные списки и массивы переменного размера достойны множества интересных вопросов, но и это только начало.

Вопросы и ответы

Когда я должен вызывать конструктор _Node?

Подобно любым другим классам, конструктор _Node следует вызывать при создании нового объекта _Node (нового узла в связанном списке). Вы не должны использовать его для создания новой ссылки на существующий объект _Node. Например, следующий код создает новый объект _Node, а затем немедленно теряет след единственной ссылки на него:

```
oldfirst = _Node(item, next)
oldfirst = first
```

Хотя этот код и не ошибочен, он несколько вреден, поскольку создает "сирот" без всякого смысла.

Почему бы не определять класс Node как автономный в отдельном файле node.py?

Определив класс _Node в таком файле, как linkedstack.py или linkedqueue.py, и начав его имя с символа подчеркивания, мы указываем клиентам классов Stack и Queue не использовать класс _Node непосредственно. Нам нужно, чтобы объекты _Node использовались только в реализациях linkedstack.py или linkedqueue.py, но не в других клиентах.

Нужно ли разрешить клиенту вставлять в стек или очередь элемент None?

Этот вопрос нередко возникает при реализации коллекций в Python. Наши реализации разрешают вставку любого объекта, включая None.

Есть ли стандартные модули Python для стеков и очередей?

Нет. Как уже упоминалось ранее в этом разделе, у встроенного типа данных Python list есть возможности, существенно облегчающие реализацию эффективного стека. Но тип данных list все же нуждается во многих дополнительных методах, обычно не ассоциируемых со стеком, например, индексированный доступ и удаление произвольного элемента. Главное преимущество ограничения набора операций только необходимыми (и только необходимыми) в том, что это облегчает реализацию и позволяет обеспечить наилучшие из возможных гарантий производительности. Python включает также тип данных collections.deque, реализующий изменяемую последовательность элементов с эффективной вставкой и удалением как в начало, так и в конец.

Почему бы не создать единый тип данных, реализующий методы вставки и удаления последнего вставленного элемента, удаления первого

вставленного и случайного элемента, перебора элементов, возвращения количества элементов в коллекции и другие операции по своему усмотрению? Это позволило бы получить все их реализации в едином классе, применимом многими клиентами.

Это пример *широкого интерфейса*, которого, как упоминалось в разделе 3.3, следует избегать. Один из недостатков широких интерфейсов в трудности разработки реализации, эффективной для всех операций. Однако важней всего то, что узкие интерфейсы привносят в программы определенный порядок, намного упрощающий понимание клиентского кода. Если один клиент использует класс `Stack`, а другой класс `Queue`, то для первого важен порядок LIFO, а для второго — FIFO. Для инкапсуляции операций, популярных для всех коллекций, можно попробовать использовать наследование. Но такие реализации лучше всего оставлять экспертам, а такие реализации, как `Stack` и `Queue`, доступны любому программисту.

Есть ли способ составить клиент, способный использовать и `arraystack.py`, и `linkedstack.py` в той же программе?

Да. Самый простой способ — добавить в оператор `import` директиву `as`, как показано ниже. В действительности этот вид оператора `import` создает псевдоним для имени класса, и ваш код затем может использовать этот псевдоним вместо имени класса.

```
from arraystack import Stack as ArrayStack
from linkedstack import Stack as LinkedStack

...
stack1 = ArrayStack()
stack2 = LinkedStack()
```

Упражнения

4.3.1. Каков вывод для вызова `python arraystack.py` при следующем вводе:

```
it was - the best - of times - - - it was - the - -
```

4.3.2. Каково содержимое и длина массива в программе `arraystack.py` после каждой операции для следующего ввода:

```
it was - the best - of times - - - it was - the - -
```

4.3.3. Предположим, клиент выполняет смешанную последовательность операций *помещения* и *извлечения* из объекта `Stack`. Операции помещения вносят в стек по порядку целые числа от 0 до 9, операции извлечения выводят возвращаемые значения. Какие из следующих последовательностей *не могли* быть получены?

a. 4 3 2 1 0 9 8 7 6 5
b. 4 6 8 7 5 3 2 9 0 1
c. 2 5 6 7 4 8 9 3 1 0
d. 4 3 2 1 0 5 6 7 8 9
e. 1 2 3 4 5 6 9 8 7 0
f. 0 4 6 5 3 8 1 7 2 9
g. 1 4 7 9 8 6 5 3 0 2
h. 2 1 4 3 6 5 8 7 9 0

4.3.4. Составьте клиент стека `reverse.py`, читающий строки со стандартного ввода и выводящий их в обратном порядке.

4.3.5. Составьте клиент стека `parentheses.py`, читающий текстовый поток со стандартного устройства ввода и использующий стек для определения правильности расстановки в нем скобок. Например, для `[()]{}{[()()]()}` ваша программа должна выводить `True`, а для `[(])` — `False`.

4.3.6. Добавьте в класс `Stack` из программы 4.3.2 (`linkedstack.py`) метод `__len__()`.

4.3.7. Добавьте в класс `Stack` из программы `arraystack.py` метод `peek()`, который возвращает последний вставленный элемент, не извлекая его.

4.3.8. Что выводит следующий фрагмент кода, когда `n` равно `50`? Дайте общее описание того, что делает этот фрагмент кода для заданного положительного целого числа `n`.

```
stack = Stack()
while n > 0:
    stack.push(n % 2)
```

```
    n /= 2
while not stack.isEmpty():
    stdio.write(stack.pop())
stdio.writeln()
```

Решение: выводит двоичное представление `n` (`110010`, когда `n` равно `50`).

4.3.9. Что делает следующий фрагмент кода? Что происходит с очередью `queue`?

```
stack = Stack()
while not queue.isEmpty(): stack.push(queue.dequeue())
while not stack.isEmpty(): queue.enqueue(stack.pop())
```

4.3.10. Составьте схему трассировки объектного уровня для примера с тремя узлами, используемого в этом разделе для ознакомления со связанными списками.

4.3.11. Составьте программу, получающую со стандартного ввода выражение без левых круглых скобок и выводящую эквивалентное инфиксное выражение со вставленными круглыми скобками. Например, при вводе

```
1+2 ) * 3 - 4 ) * 5 - 6 ) ) )
```

ваша программа должна вывести

```
( ( 1+2 ) * ( ( 3 - 4 ) * ( 5 - 6 ) )
```

4.3.12. Составьте фильтр `infixtopostfix.py`, преобразующий полностью заключенные в скобки арифметические выражения из инфиксной формы в постфиксную.

4.3.13. Составьте программу `evaluatepostfix.py`, которая читает со стандартного ввода постфиксное выражение, вычисляющее его и выводящее значение на стандартный вывод. (Пересылка вывода этой программы в программу из предыдущего упражнения дает поведение, эквивалентное программе `evaluate.py`.)

4.3.14. Предположим, клиент выполняет произвольную последовательность операций *помещения* и *извлечения* из очереди `Queue`. Операции помещения добавляют в очередь по порядку целые числа от 0 до 9, операции извлечения из очереди выводят возвращаемые значения. Какие из следующих последовательностей *не могли* быть получены?

a. `0 1 2 3 4 5 6 7 8 9`
b. `4 6 8 7 5 3 2 9 0 1`
c. `2 5 6 7 4 8 9 3 1 0`
d. `4 3 2 1 0 5 6 7 8 9`

4.3.15. Составьте клиент класса `Queue`, получающий аргумент командной строки `k` и выводящий *k*-ю от последней введенной строки.

4.3.16. Укажите продолжительность каждой операции в следующем классе `Queue`, где самый старый элемент находится в элементе `_a[0]`.

```
class Queue:
def __init__(self):          self._a=[]
def isEmpty(self):           return len(self._a) == 0
def __len__(self):           return len(self._a)
def enqueue(self, item):     self._a += [item]
def dequeue(self):           return self._a.pop(0)
```

4.3.17. Укажите продолжительность каждой операции в следующем классе `Queue`, где самый старый элемент находится в элементе `_a[0]`.

```
class Queue:
def __init__(self):          self._a=[]
def isEmpty(self):           return len(self._a) == 0
def __len__(self):           return len(self._a)
def enqueue(self, item):     self._a.insert(0, item)
def dequeue(self):           return self._a.pop()
```

4.3.18. Измените программу `mm1queue.py` так, чтобы она моделировала очередь, для которой периоды обслуживания фиксируются (детерминируются) к частоте μ. Проверьте экспериментально закон Литтла для этой модели.

Упражнения со связанным списком

Следующие упражнения призваны помочь вам приобрести практический опыт в работе со связанными списками. Простейший способ подразумевает рисование того, что описано в тексте.

4.3.19. Предположим, что `x` — это узел связанного списка. Что делает следующий фрагмент кода?

```
x.next = x.next.next
```

Решение: удаляет из списка узел непосредственно после `x`.

4.3.20. Составьте функцию `find()`, получающую как аргументы первый узел в связанном списке и объект `key`. Функция возвращает `True`, если в списке есть узел, содержащий `key` в поле своего элемента, и `False` в противном случае.

4.3.21. Составьте функцию `delete()`, получающую как аргументы первый узел в связанном списке и целое число `k`. Функция удаляет *k*-й элемент связанного списка, если он существует.

4.3.22. Предположим, что `x` — это узел связанного списка. Что делает следующий фрагмент кода?

```
t.next = x.next
x.next = t
```

Решение: вставка узла `t` непосредственно после узла `x`.

4.3.23. Почему следующий фрагмент кода не имеет того же действия, что и фрагмент кода в предыдущем вопросе?

```
x.next = t
t.next = x.next
```

Решение: когда приходит время модифицировать `t.next`, `x.next` больше не является первым узлом после `x`, теперь это сам `t`!

4.3.24. Составьте функцию `removeAfter()`, получающую как аргумент узел связанного списка и удаляющую узел после заданного (и ничего не делает, если аргумент или поле `next` в узле аргумента `None`).

4.3.25. Составьте функцию `copy()`, которая получает как аргумент узел связанного списка и создает новый связанный список с той же последовательностью элементов, не удаляя первоначальный связанный список.

4.3.26. Составьте функцию `remove()`, получающую как аргументы узел связанного списка и объект `item`. Функция удаляет каждый узел в списке с элементом `item`.

4.3.27. Составьте функцию `listmax()`, которая получает как аргумент первый узел в связанном списке и возвращает значение максимального элемента в списке. Подразумевается, что элементы сравнимы и возвращают `None`, если список пуст.

4.3.28. Найдите рекурсивное решение предыдущего вопроса.

4.3.29. Составьте функцию, которая получает как аргумент первый узел в связанном списке и изменяет порядок списка на обратный, возвращая в результате первый узел.

Итерационное решение. Для решения этой задачи мы храним ссылки на три последовательных узла в связанном списке: `reverse`, `first` и `second`. При каждой итерации мы извлекаем узел `first` из исходного связанного списка и вставляем его в начало списка с обратным порядком. Мы поддерживаем инвариант, что `first` — это первый узел, оставшийся в исходном списке, `second` — второй узел, оставшийся в исходном списке, и `reverse` — первый узел результирующего обратного списка.

```
def reverse(first):
    reverse = None
    while first is not None:
        second = first.next
        first.next = reverse
        reverse = first
        first = second
    return reverse
```

При создании кода, задействовавшего связанные списки, мы всегда должны делать все возможное для правильной обработки исключительных случаев (когда связанный список пуст, когда у списка есть только один или два узла) и граничных случаев (связанных с первыми или последними элементами). Обычно это намного сложнее ординарных случаев.

4.3.30. Составьте рекурсивную функцию, выводящую элементы связанного списка в обратном порядке. Не изменяйте ни одну из ссылок. *Просто*: используйте квадратичное время и постоянное дополнительное пространство. *Также просто*: используйте линейное время, линейное дополнительное пространство. *Не столь просто*: разработайте алгоритм "разделяй и властвуй", использующий линейно-логарифмическое время и логарифмическое дополнительное пространство.

Решение для квадратичного времени и постоянного пространства: мы рекурсивно изменяем на обратную часть списка, начинающуюся во втором узле, а затем добавляем первый элемент в конец.

```
def reverse(first):
    if first is None:
        return None
    if first.next is None:
        return first
    second = first.next
    rest = reverse(second)
    second.next = first
    first.next = None
    return rest
```

4.3.31. Составьте рекурсивную функцию, случайно перетасовывающую элементы связанного списка за счет изменения ссылок. *Просто*: используйте квадратичное время, постоянное дополнительное пространство. *Не столь просто*: разработайте алгоритм "разделяй и властвуй", занимающий линейно-логарифмическое время и использующий логарифмическую дополнительную память. Этап "слияния" см. в упр. 1.4.38.

Практические упражнения

4.3.32. *Двухсторонняя очередь.* Комбинацией стека и очереди является *двухсторонняя очередь* (double-ended queue, или deque). Составьте класс `Deque`, использующий связанный список для реализации следующего API.

API для двухсторонней очереди

Операции постоянного времени	Описание
`Deque()`	Новая двухсторонняя очередь
`d.isEmpty()`	Очередь `d` пуста?
`d.enqueue(item)`	Помещает `item` в конец `d`
`d.dequeue()`	Возвращает и удаляет элемент с конца `d`
`d.push(item)`	Помещает `item` в начало `d`
`d.pop(item)`	Возвращает и удаляет элемент из начала `d`

Примечание. Используемое пространство должно быть пропорционально количеству элементов.

4.3.33. *Задача Иосифа Флавия.* В исторической *задаче Иосифа Флавия* (Josephus problem) *n* окруженных в пещере повстанцев решили не сдаваться живьем. Они договорились встать в круг (в позиции от 0 до n) и убивать каждого m-го человека, пока в живых не останется только один. По легенде Иосиф быстро догадался, куда следует встать, чтобы остаться в живых. Составьте программу `josephus.py`, клиент класса `Queue`, получающий из командной строки значения *n* и m, а затем выводящий порядок устранения (подсказывая таким образом Иосифу правильную позицию в круге).

```
% python josephus.py 7 2
1 3 5 0 4 2 6
```

4.3.34. *Слияние двух отсортированных очередей.* Даны две очереди со строками в порядке возрастания. Переместите все строки в третью очередь так, чтобы они располагались в порядке возрастания.

4.3.35. *Не рекурсивная сортировка с объединением.* Дано *n* строк. Создайте *n* очередей, каждая из которых содержит одну из строк. Создайте очередь из *n* очередей. Затем последовательно примените операцию сортировки со слиянием к первым двум очередям и вставьте объединенную очередь в конец. Повторяйте, пока очередь очередей не будет содержать только одну очередь.

4.3.36. *Удалите i-й элемент.* Реализуйте класс, поддерживающий следующий API.

API для обобщенной очереди

Операция	Описание
`GeneralizedQueue()`	Новая очередь
`q.isEmpty()`	Очередь `q` пуста?
`q.insert(item)`	Добавляет `item` в `q`
`q.delete(i)`	Возвращает и удаляет `i`-й недавно вставленный элемент из `q`

Примечание. Используемое пространство должно быть пропорционально количеству элементов.

Сначала разработайте реализацию, использующую список Python (массив переменного размера), а затем разработайте реализацию, использующую связанный список. (Более эффективная реализация, использующая бинарное дерево поиска, приведена в упр. 4.4.69.)

4.3.37. *Очередь с двумя стеками.* Представьте реализацию очереди, использующую два стека (и только постоянный объем дополнительной памяти), чтобы каждая операция очереди задействовала постоянное амортизированное количество операций стека.

4.3.38. *Кольцевой буфер.* Кольцевой буфер, или циклическая очередь, — это структура данных FIFO фиксированной емкости n. Он очень полезен для передачи данных между асинхронными процессами или хранения файлов журнала. Когда буфер пуст, потребитель ожидает прибытия данных; когда буфер полон, производитель ждет, чтобы внести данные. Разработайте API для кольцевого буфера и реализуйте его, используя массив (с круговым переносом).

4.3.39. *Перенос в начало.* Прочитайте со стандартного ввода последовательность символов и сохраните их в связанном списке без дубликатов. Когда будете читать символ, не встречавшийся ранее, вставляйте его в начало списка. Когда будете читать уже имеющийся символ, удаляйте его из списка и снова вставляйте в начало. Назовите свою программу `MoveToFront`: она реализует известную стратегию *переноса в начало* (move-to-front), используемую для кеширования, сжатия данных и многого другого, где к элементам, к которым недавно обращались, вероятнее всего, обратятся снова.

4.3.40. *Случайная очередь.* Хранит коллекцию элементов согласно следующему API.

API для случайной очереди

Операции постоянного времени	Описание
`RandomQueue()`	Новая случайная очередь
`q.isEmpty()`	Очередь `q` пуста?
`q.enqueue(item)`	Добавляет `item` в очередь `q`
`q.dequeue()`	Возвращает и удаляет случайный элемент из `q` (выборка без замены)
`q.sample()`	Возвращает, но не удаляет случайный элемент из `q` (выборка с заменой)
`len(q)`	Количество элементов в `q`

Примечание. Используемое пространство должно быть пропорционально количеству элементов.

Составьте класс `RandomQueue`, реализующий этот API. *Подсказка.* Используйте список Python (массив переменного размера), как в программе 4.3.1. При удалении обменяйте элемент в случайной позиции (по индексу от `0` до `n-1`) с таковым в последней позиции (по индексу `n-1`). Затем удалите и возвратите последний объект. Составьте клиент, выводящий набор карт в случайном порядке, используя класс `RandomQueue`.

4.3.41. *Топологическая сортировка.* На сервере есть упорядоченная последовательность из `n` задач, пронумерованных от `0` до `n-1`. Некоторые из задач должны завершиться прежде, чем другие могут быть начаты. Составьте программу `topologicalsorter.py`, которая получает как аргумент командной строки `n` и читает со стандартного ввода последовательность упорядоченных пар задач `i j`, а затем выводит последовательность целых чисел таким образом, чтобы для каждой пары `i j` во вводе задача `i` присутствовала перед задачей `j`. Используйте следующий алгоритм: для каждой вводимой задачи создайте (1) очередь задач, которые должны следовать за данной, и (2) ее *полустепень захода* (indegree) (количество задач, которые должны поступить перед этой). Затем создайте очередь всех узлов с полустепенью захода 0 и последовательно удаляйте задачи с нулевой полустепнью захода, поддерживая все структуры данных.

4.3.42. *Буфер текстового редактора.* Разработайте тип данных для буфера в текстовом редакторе, реализующий следующий API.

API для текстового буфера

Операции постоянного времени	Описание
`Buffer()`	Новый буфер
`buf.insert(c)`	Вставляет в `buf` символ `c` непосредственно перед курсором
`buf.delete()`	Возвращает и удаляет из `buf` символ в позиции курсора
`buf.left(k)`	Перемещает позицию курсора `k` влево
`buf.right(k)`	Перемещает позицию курсора `k` вправо

Подсказка: используйте два стека.

4.3.43. *Скопируйте стек.* Создайте метод `copy()` для реализации стека на базе связанного списка так, чтобы код

```
stack2 = stack1.copy()
```

присваивал `stack2` ссылку на новую независимую копию стека `stack1`. Вы должны быть в состоянии выполнять операции вставки и извлечения из `stack1` и `stack2` независимо друг от друга.

4.3.44. *Скопируйте очередь.* Создайте метод `copy()` для реализации очереди на базе связанного списка так, чтобы код

```
queue2 = queue1.copy()
```

присваивал `queue2` ссылку на новую независимую копию очереди `queue1`. *Подсказка*: удалите все элементы из `queue1` и добавьте их в `queue1` и `queue2`.

4.3.45. *Стек с явным массивом переменного размера.* Реализуйте стек, используя явный массив переменного размера: инициализируйте пустой стек с массивом длиной 1 в качестве переменной экземпляра; удваивайте длину массива, когда он полон, и делите ее на два, когда он заполнен на четверть.

Решение:

```
class Stack:
    def __init__(self):
        self._a = [None]
        self._n = 0

    def isEmpty(self):
        return self._n == 0

    def __len__(self):
        return self._n
```

```
    def _resize(self, capacity):
        temp = stdarray.create1D(capacity)
        for i in range(self._n):
            temp[i] = self._a[i]
        self._a = temp

    def push(self, item):
        if self._n == len(self._a):
            self._resize(2 * self._n)
        self._a[self._n] = item
    self._n += 1

    def pop(self):
        self._n -= 1
        item = self._a[self._n]
        self._a[self._n] = None
        if (self._n > 0) and (self._n == len(self._a) // 4):
            self._resize(self._n // 2)
    return item
```

4.3.46. *Очередь с явным массивом переменного размера.* Реализуйте очередь, используя явный массив переменного размера так, чтобы все операции занимали постоянное амортизируемое время. *Подсказка*: проблема в том, что элементы "сползают" по массиву по мере добавления и удаления элементов из очереди. Для поддержки индексов массива элементов в передней и задней части очереди используйте арифметические операции над абсолютными значениями чисел.

Ввод	Вывод	*n*	*lo*	*hi*	a[]							
					0	1	2	3	4	5	6	7
		0	0	0	None							
to		1	0	1	**to**	None						
be		2	0	2	**to**	**be**						
or		3	0	3	**to**	**be**	**or**	None				
not		4	0	4	**to**	**be**	**or**	**not**				
to		5	0	5	**to**	**be**	**or**	**not**	**to**	None	None	None
-	to	4	1	4	None	**be**	**or**	**not**	**to**	None	None	None
be		5	1	6	None	**be**	**or**	**not**	**to**	**be**	None	None
-	be	4	2	6	None	None	**or**	**not**	**to**	**be**	None	None
-	or	3	3	6	None	None	None	**not**	**to**	**not**	None	None
that		4	3	7	None	None	None	**not**	**to**	**not**	**that**	None

4.3.47. *Модель очереди.* Что будет, когда вы модифицируете программу `mm1queue.py` так, чтобы использовать стек вместо очереди. Выполняется ли закон Литтла? Ответьте на тот же вопрос для случайной очереди. Выведите гистограммы и сравните среднеквадратичные отклонения времени ожидания.

4.3.48. *Модель балансировки нагрузки.* Пересмотрите программу `loadbalance.py` так, чтобы выводить среднюю и максимальную длину очереди, а не рисовать гистограммы. Используйте ее для моделирования ситуации с 1 миллионом элементов на 100 000 очередей. Выведите среднее значение максимальной длины очереди для 100 испытаний с размерами выборки 1, 2, 3 и 4 каждый. Подтверждают ли ваши эксперименты вывод, сделанный в тексте об использовании выборки размером 2?

4.3.49. *Список файлов.* Папка — это список файлов и папок. Составьте программу, получающую как аргумент командной строки имя папки и выводящую имена всех содержащихся в ней файлов и папок со всем их содержимым (рекурсивно). *Подсказка*: используйте очередь, а также просмотрите функцию `listdir()`, определенную в модуле Python `os`.

4.4. Таблицы идентификаторов

Программы этого раздела...

Программа 4.4.1. Поиск в словаре (lookup.py) 626
Программа 4.4.2. Индексация (index.py) 628
Программа 4.4.3. Хеш-таблица (hashst.py) 634
Программа 4.4.4. Бинарное дерево поиска (bst.py) 641

Таблица идентификаторов (symbol table) — это тип данных, используемый для связи *значений* (value) с *ключами* (key). Клиенты могут *поместить* (put) запись в таблицу идентификаторов, определив пару *"ключ–значение"* (key-value), а затем *получить* (get) значение, соответствующее указанному ключу, из таблицы идентификаторов. Например, университет мог бы ассоциировать такую информацию, как имя студента, домашний адрес и оценки (значения), с номером карточки социального страхования этого студента (ключ), чтобы к записи каждого студента можно было обратиться по его номеру карточки социального страхования. Тот же подход мог бы пригодиться ученому для организации данных, бизнесмену — для отслеживания транзакций клиента, поисковой системе Интернета — для ассоциации ключевых слов с веб-страницами и для многих других случаев.

Благодаря своей фундаментальной важности таблицы идентификаторов широко используются и хорошо изучены с ранних дней компьютеров. Разработка реализаций, гарантирующих высокую производительность, до сих пор остается темой активных исследований. Вполне естественно возникает необходимость в некоторых других операциях с таблицами идентификаторов, кроме помещения и возвращения, однако гарантия хорошей производительности для расширенных наборов операций может быть весьма сложной задачей.

В этой главе мы рассматриваем базовый API для типа данных таблицы идентификаторов. Наш API добавляет к операциям *помещения* и *получения* значений возможность проверять, связано ли некое значение с данным ключом (*содержание* (contain)), и *перебор* (iteration) по ключам. Мы также рассмотрим модификацию API для случая, где ключи сравнимы, что обеспечивает множество полезных операций.

Кроме того, речь пойдет о двух классических реализациях. Первая использует операцию *хеширования* (hashing), преобразующую ключи в индексы массива, используемые для доступа к значениям. Вторая основана на такой структуре данных, как *бинарное дерево поиска* (binary search tree — BST). Оба решения замечательно просты и применимы во многих практических ситуациях, а также служат основанием для реализаций промышленных таблиц идентификаторов, применяемых в современных средах программирования. Рассматриваемый код таблиц идентификаторов лишь ненамного сложнее, чем массив переменного размера

и связанного списка, использованных для реализации стеков и очередей, но это познакомит вас с новыми размерностями в структурировании данных.

API. Таблица идентификаторов — это коллекция пар "ключ–значение", где каждая запись ассоциирует значение с ключом следующим образом.

API для таблицы идентификаторов

Операции постоянного или линейно-логарифмического времени	Описание
`SymbolTable()`	Новая таблица идентификаторов
`st[key] = val`	Ассоциирует `key` с `val` в `st`
`st[key]`	Значение, связанное с `key` в `st`
`key in st`	Есть ли в `st` значение, связанное с `key`?
Операции линейного времени	
`for key in st:`	Перебор `st` по ключам

Примечание. Используемое пространство должно быть пропорционально количеству ключей в таблице идентификаторов.

В этом разделе мы обсудим данный API, клиентов, реализации и расширения. API совместим с API для встроенного типа данных Python `dict`, рассматриваемого далее в этом разделе. API уже отражает несколько проектных решений, перечисляемых далее.

Ассоциативные массивы. Для двух базовых операций помещения и возвращения мы перегружаем оператор `[]`. В клиентском коде это означает, что мы можем рассматривать таблицу идентификаторов как *ассоциативный массив* (associative array), где мы можем использовать стандартный синтаксис массива для любого типа данных в квадратных скобках вместо целого числа от нуля до длины, как у массива. Следовательно, мы можем ассоциировать кодон с названием аминокислоты с клиентским коде:

```
amino['TTA'] = 'Leucine'
```

Впоследствии в клиентском коде мы можем обращаться к названию, связанному с данным кодоном, так:

```
stdio.writeln(amino['TTA'])
```

Таким образом, ссылка на ассоциативный массив — это операция *получения*, если она не находится слева от оператора присвоения, тогда это операция *помещения*. Мы можем обеспечить эти операции, реализовав специальные методы `__getitem__()` и `__setitem__()`. Размышление в терминах ассоциативных массивов — хороший способ понять главные цели таблиц идентификаторов.

Политика замены старого значения. Если значение должно быть связано с ключом, у которого уже есть ассоциированное значение, согласно принятому соглашению, новое значение заменяет прежнее (как и оператор присвоения

массива), как и можно было бы ожидать от абстракции ассоциативного массива. Операция `key in st` поддерживается специальным методом `__contains__()` и обеспечивает клиенту гибкость, позволяя избежать нежелательных ситуаций.

Не найдено. Вызов `st[key]` передает сообщение об ошибке `KeyError`, если с ключом `key` в таблице не ассоциировано никакое значение. Альтернатива — возвращение в таких случаях `None`.

Ключи и значения `None`. Клиенты могут использовать `None` как ключ или значение, хотя обычно так не делают. Альтернатива — отбрасывать ключи и / или значения `None`.

Возможность итерации. Для поддержки конструкции `for key in st:`, согласно соглашению Python, необходимо реализовать специальный метод `__iter__()`, возвращающий *итератор* (iterator) — специальный тип данных, включающий методы, вызываемые в начале и при каждой итерации цикла `for`. Механизм Python для итераций рассматривается в конце этого раздела.

Удаление. Наш базовый API не включает метод для удаления ключей из таблицы идентификаторов. Некоторым приложениям действительно требуется такой метод, и Python предоставляет специальный синтаксис `del st[key]`, для поддержки которого следует реализовать специальный метод `__delitem__()`. Мы оставляем его реализацию в качестве упражнения или более расширенного курса по алгоритмам.

Неизменяемые ключи. Мы подразумеваем, что ключи не изменяют свои значения в таблице идентификаторов. Самые простые и наиболее часто используемые типы ключей (целые, вещественные числа и строки) являются неизменными. Если хорошо подумать, то это очень разумное свойство! Если клиент изменит ключ, то как реализация таблицы идентификаторов сможет отследить этот факт?

Вариации. Программисты выявили множество других полезных операций с таблицами идентификаторов, и API на основании многих из них были широко изучены. Некоторые из этих операций мы рассматриваем в этом разделе и в упражнениях.

Сравнимые ключи. Во многих приложениях ключи могут быть целыми, вещественными числами, строками или данными других типов, обладающих естественным порядком. В языке Python, как обсуждалось в разделе 3.3, такие ключи будут *сравнимы*. Таблицы идентификаторов со сравнимыми ключами важны по двум причинам. Во-первых, мы можем упорядочить ключи, что позволит разработать реализации операций помещения и возвращения, способные *гарантировать* спецификации производительности из API. Во-вторых, со сравнимыми ключами на ум приходит целый ворох новых операций. Клиент мог бы захотеть наименьший или наибольший ключ, или медиану, или осуществить перебор по ключам в отсортированном порядке. Полный обзор этой темы подходит скорее для книги по алгоритмам и структурам данных, но мы исследуем типичный

клиент и реализацию такого типа данных далее в этом разделе. Частичный API представлен ниже.

Таблицы идентификаторов — одна из наиболее тщательно изученных структур данных в информатике, поэтому воздействие их и многих альтернативных решений на проект было тщательно изучено, как вы узнаете, пройдя старшие курсы по информатике. В этом разделе мы познакомим вас с самым важным свойством таблиц идентификаторов на примере разработки двух клиентских программ и двух классических эффективных реализаций. Изучение характеристик производительности этих реализаций продемонстрирует, что они вполне соответствуют потребностям типичных клиентов, даже если таблицы идентификаторов огромны.

Частичный API для упорядоченной таблицы идентификаторов сравнимых ключей

Операции постоянного или линейно-логарифмического времени	Описание
`OrderedSymbolTable()`	Новая упорядоченная таблица идентификаторов
`st[key] = val`	Ассоциирует `key` с `val` в `st`
`st[key]`	Значение, ассоциированное с `key` в `st`
`key in st`	Ассоциировано ли значение с `key` в `st`?
`st.rank(key)`	Количество ключей меньше, чем `key` в `st`
`st.select(k)`	*k*-й наименьший ключ в `st` (ключ в `st` ранга `k`)
Операции линейного времени	
`for key in st:`	Перебор по ключам в отсортированном `st`

Примечание. Используемое пространство должно быть пропорционально количеству ключей в таблице идентификаторов.

Клиенты таблицы идентификаторов. Немного освоившись с идеей, вы поймете, что таблицы идентификаторов широко применимы. Чтобы убедить вас в этом факте, мы рассмотрим два типичных примера, каждый из которых имеет большое количество важных и общеизвестных практических применений.

Поиск в словаре. Проще всего создать таблицу идентификаторов последовательными операциями *помещения*, чтобы клиент мог осуществлять запросы на *получение*. Мы храним коллекцию данных, чтобы можно было быстро обращаться к необходимым значениям. Большинство приложений также использует ту идею, что таблица идентификаторов — это *динамический* словарь, в котором просто искать *и* модифицировать информацию. Ниже приведены общеизвестные примеры, иллюстрирующие удобство этого подхода.

Типичные применения словаря

	Ключ	Значение
Телефонная книга	Имя	Номер телефона
Словарь	Слово	Определение
Учетная запись	Номер счета	Баланс
Генетика	Кодон	Аминокислота
Данные	Дата / время	Результаты
Компилятор	Имя переменной	Расположение в памяти
Совместное использование файлов	Название музыкального произведения	Машина
DNS Интернета	Веб-сайт	IP-адрес

- *Телефонная книга.* Когда ключи — имена людей, а значения — номера их телефонов, таблица идентификаторов моделирует телефонную книгу. Весьма существенное отличие от печатной телефонной книги в том, что мы можем добавлять новые имена или изменять существующие номера телефонов. Мы также могли бы использовать номер телефона как ключ, а имя как значение. Если вы никогда так не делали, попробуйте ввести ваш номер телефона (с кодом города) в строке поиска в браузере.
- *Словарь.* Объединение слов с их определениями является общеизвестной концепцией толкового словаря. В течение многих лет люди хранили словари в домах и офисах, чтобы можно было проверить определения и правописание (значения) слов (ключи). Теперь благодаря хорошим реализациям таблиц идентификаторов пользователи ожидают наличия на своих компьютерах встроенных программ проверки орфографии и немедленного доступа к определениям слов.
- *Информация об учетной записи.* Владельцы акций регулярно проверяют их текущую цену в веб. Несколько веб-служб ассоциируют биржевой символ (ключ) с текущей ценой (значение), как правило, наряду с большим количеством другой информации. Коммерческие приложения такого типа имеются во множестве финансовых учреждений, они ассоциируют информацию учетной записи с именем или номером счета, а в образовательных учреждениях ассоциируют оценки с именами или идентификационными номерами студентов.
- *Генетика.* Символы играют центральную роль в современной генетике, как уже упоминалось выше (см. программу 3.1.1). Самый простой прием — использование символов `A`, `C`, `T` и `G` для представления нуклеотидов в ДНК живых организмов. Следующие примеры: соответствие между кодонами (триплетами нуклеотидов) и аминокислотами (`TTA` соответствует лейцину, `TCT` — цистину и т.д.), соответствие между последовательностями

аминокислот и белками и т.д. В генетических исследованиях для организации данных обычно используют различные типы таблиц идентификаторов.

- *Экспериментальные данные.* Во многих областях науки ученые накопили огромные объемы экспериментальных данных, а их эффективная организация и обработка жизненно важны для понимания их значения. Таблицы идентификаторов — это критически важная отправная точка, а передовые структуры данных и алгоритмы на базе таблиц идентификаторов являются важнейшей частью научного исследования.
- *Языки программирования.* Одним из самых первых применений таблиц идентификаторов была организация информации для программирования. Сначала программы были просто последовательностями чисел, но программисты очень быстро поняли, что использование символьных имен для операций и ячеек памяти (имена переменных) намного удобней. Ассоциация имен с числами требует таблиц идентификаторов. По мере увеличения размера программ стоимость операций с таблицами идентификаторов стала узким местом во время разработки программ, что привело к разработке структур данных и алгоритмов, подобных рассматриваемым в этом разделе.
- *Файлы.* Мы регулярно используем таблицы идентификаторов для организации данных компьютерных систем. Самый показательный пример, вероятно, — это *файловая система*, где имя файла (ключ) ассоциировано с расположением его содержимого (значение). Ваш проигрыватель использует ту же систему, ассоциируя заголовки музыкальных произведений (ключи) с расположением музыкальных файлов (значение).
- *DNS Интернета.* Система доменных имен (DNS), на основании которой организована информация в Интернете, ассоциирует понятные людям URL, такие как `www.princeton.edu` или `www.wikipedia.org` (ключи), с IP-адресами (значения), понятными компьютерам и сетевым маршрутизаторам, такими как `208.216.181.15` или `207.142.131.206`. Эта система — "телефонная книга следующего поколения". Таким образом, люди могут использовать легко запоминаемые имена, а машины могут эффективно обработать числа. Количество поисков в таблице идентификаторов, осуществляемых с этой целью на маршрутизаторах Интернета за каждую секунду во всем мире, просто огромна, поэтому производительность вполне очевидно имеет значение. Каждый год к Интернету подключаются миллионы новых компьютеров и других устройств, поэтому такие таблицы идентификаторов на маршрутизаторах Интернета должны быть динамическими.

Несмотря на широкое разнообразие, этот список далеко не исчерпывающий и должен лишь дать вам общее представление о разновидностях областей применения абстракции таблицы идентификаторов. Всякий раз, когда вы определяете

что-то по имени, срабатывает таблица идентификаторов. Файловая система вашего компьютера или веб могли бы самостоятельно выполнить эту работу, но таблица идентификаторов там где-нибудь все же есть.

Программа 4.4.1 (`lookup.py`) создает набор пар "ключ–значение" из указанных в командной строке файлов со значениями, разделяемыми запятыми (см. раздел 3.1), а затем выводит значения, соответствующие прочитанному ключу. Аргументы командной строки — имя файла и два целых числа: одно определяет поле, служащее ключом, а другое — значением. Примеры подобных, но немного более сложных клиентов проверки приведены в упражнениях. Например, мы могли бы сделать словарь динамическим, разрешив командам со стандартного ввода изменять значение, ассоциируемое с ключом (см. упр. 4.4.1).

Для того чтобы понимать таблицы идентификаторов, загрузите файлы `lookup.py` и `hashst.py` (их реализацию мы рассматриваем далее) с сайта книги, чтобы осуществлять поиск в таблице идентификаторов. Вы можете найти множество описываемых нами файлов `.csv`, связанных с разными приложениями, включая `amino.csv` (таблица кодонов и аминокислот), `djia.csv` (цена при открытии, объем и цена на момент закрытия биржи, для каждого дня) и `ip.csv` (выборка записей из базы данных DNS). Выбирая поле, используемое как ключ, помните, что *каждый ключ должен уникально определять значение*. Если будет несколько операций *помещения*, ассоциирующих значение с ключом, то в таблице сохранится лишь самый последний (вспомните об ассоциативных массивах). Далее мы рассмотрим случай, когда с ключом необходимо ассоциировать несколько значений.

```
% more amino.csv
TTT,Phe,F,Phenylalanine
TTC,Phe,F,Phenylalanine
TTA,Leu,L,Leucine
TTG,Leu,L,Leucine
TCT,Ser,S,Serine
TCC,Ser,S,Serine
TCA,Ser,S,Serine
TCG,Ser,S,Serine
TAT,Tyr,Y,Tyrosine
TAC,Tyr,Y,Tyrosine
TAA,Stop,Stop,Stop
...
GCA,Ala,A,Alanine
GCG,Ala,A,Alanine
GAT,Asp,D,Aspartic Acid
GAC,Asp,D,Aspartic Acid
GAA,Gly,G,Glutamic Acid
GAG,Gly,G,Glutamic Acid
GGT,Gly,G,Glycine
GGC,Gly,G,Glycine
GGA,Gly,G,Glycine
GGG,Gly,G,Glycine

% more djia.csv
...
20-Oct-87,1738.74,608099968,1841.01
19-Oct-87,2164.16,604300032,1738.74
16-Oct-87,2355.09,338500000,2246.73
15-Oct-87,2412.70,263200000,2355.09
...
30-Oct-29,230.98,10730000,258.47
29-Oct-29,252.38,16410000,230.07
28-Oct-29,295.18,9210000,260.64
25-Oct-29,299.47,5920000,301.22
...

% more ip.csv
...
www.ebay.com,66.135.192.87
www.princeton.edu,128.112.128.15
www.cs.princeton.edu,128.112.136.35
www.harvard.edu,128.103.60.24
www.yale.edu,130.132.51.8
www.cnn.com,64.236.16.20
www.google.com,216.239.41.99
www.nytimes.com,199.239.136.200
www.apple.com,17.112.152.32
www.slashdot.org,66.35.250.151
www.espn.com,199.181.135.201
www.weather.com,63.111.66.11
www.yahoo.com,216.109.118.65
...
```

Типичные файлы со значениями, разделяемыми запятыми (CSV)

Программа 4.4.1. Поиск в словаре (lookup.py)

```
import sys
import stdio
from instream import InStream
from hashst import SymbolTable

instream = InStream(sys.argv[1])
keyField = int(sys.argv[2])
valField = int(sys.argv[3])

database = instream.readAllLines()

st = SymbolTable()
for line in database:
    tokens = line.split(',')
    key = tokens[keyField]
    val = tokens[valField]
    st[key] = val

while not stdio.isEmpty():
    query = stdio.readString()
    if query in st: stdio.writeln(st[query])
    else:           stdio.writeln('Not found')
```

instream	Поток ввода (.csv)
keyField	Позиция ключа
valField	Позиция значения
database[]	Строки во вводе
st	Таблица идентификаторов
tokens	Значения в строке
key	Ключ
val	Значение
query	Строка запроса

Этот управляемый данными клиент таблицы идентификаторов читает пары "ключ–значение" из файла .csv, а затем выводит соответствующие ключам значения. И ключи, и значения — строки.

```
% python lookup.py amino.csv 0 3
TTA
Leucine
ABC
Not found
TCT
Serine

% python lookup.py amino.csv 3 0
Glycine
GGG
```

```
% python lookup.py ip.csv 0 1
www.google.com
216.239.41.99

% python lookup.py ip.csv 1 0
216.239.41.99
www.google.com

% python lookup.py djia.csv 0 1
29-Oct-29
252.38
```

Далее вы увидите, что стоимость ассоциативных ссылок массива в lookup.py может быть линейной или логарифмической. Этот факт подразумевает, что в получении ответа на ваш первый запрос (для всех операций *помещения* при построении таблицы) может быть маленькая задержка, но вы получите немедленный ответ на все другие запросы.

Типичные применения индексации

	Ключ	Значение
Книга	Термин	Номер страницы
Генетика	Подстрока ДНК	Расположение
Поиск в вебе	Ключевое слово	Веб-сайты
Бизнес	Имя клиента	Транзакции

Индексация. Программа 4.4.2 (`index.py`) является типичным примером клиента таблицы идентификаторов для сравнимых ключей. Она читает со стандартного ввода список строк и выводит отсортированную таблицу всех отличных строк наряду со списком целых чисел для каждой строки, определяющим ее позицию во вводе. В данном случае мы, казалось бы, ассоциируем с каждым ключом несколько значений, но фактически связываем только одно: список Python. У этой проблемы также есть вполне знакомые применения.

- *Книжный индекс.* У каждого учебника есть индекс, где вы ищете термины и получаете содержащие их номера страниц. Поскольку читатель не хочет видеть в индексе все слова из книги, такая программа, как `index.py`, может стать отправной точкой для создания хорошего индекса.
- *Языки программирования.* В большой программе, использующей множество идентификаторов, всегда полезно знать, где используется каждое имя. Программа `index.py` может оказаться весьма полезным инструментом, помогающим программисту отследить использование идентификаторов в их программах. Исторически печатная таблица идентификаторов была одними из важнейших инструментов, используемых программистами для управления большими программами. В современных системах таблицы идентификаторов — основа программных инструментальных средств для манипулирования именами модулей.
- *Генетика.* В типичном (весьма упрощенном) случае генетического исследования необходимо узнать позиции заданной генетической последовательности в существующем геноме или наборе геномов. Существование или близость определенных последовательностей может иметь серьезное научное значение. Отправная точка такого исследования — индекс, подобный создаваемому программой `index.py`, модифицированный так, чтобы учитывать тот факт, что геномы не разделяются на слова.
- *Поиск в веб.* Когда вы вводите ключевое слово и получаете список содержащих его веб-сайтов, вы используете индекс, созданный поисковой сетевой системой. С каждым ключом (запросом) ассоциировано одно значение (список страниц), хотя действительность немного динамичней и сложней, поскольку мы часто определяем несколько ключей, а страницы распространены по всей сети, а не хранятся в таблице на едином компьютере.
- *Информация об учетной записи.* Один из способов хранения компаниями учетных записей клиентов и отслеживания транзакций за день заключается

в хранении индекса для списка транзакций. Ключ — номер счета, значение — список вхождений этого номера в списке транзакций.

Программа 4.4.2. Индексация (index.py)

```
import sys
import stdio
from bst import OrderedSymbolTable

minLength = int(sys.argv[1])
minCount = int(sys.argv[2])

words = stdio.readAllStrings()

bst = OrderedSymbolTable()
for i in range(len(words)):
    word = words[i]
    if len(word) >= minLength:
        if word not in bst: bst[word] = []
        bst[word] += [i]

for word in bst:
    if len(bst[word]) >= minCount:
        stdio.write(word + ': ')
        for i in bst[word]:
            stdio.write(str(i) + ' ')
        stdio.writeln()
```

minLength	Минимальная длина
minCount	Пороговое значение счета
bst	Упорядоченная таблица идентификаторов
word	Текущее слово
bst[word]	Массив позиций для текущего слова

Эта программа получает как аргументы командной строки целые числа `minLength` и `minCount`, читает со стандартного ввода все слова и создает отсортированный индекс, указывающий, где встречается каждое слово в пределах стандартного ввода. Учитываются только слова, содержащие по крайней мере `minLength` символов, и выводятся только слова, встречающиеся по крайней мере `minCount` раз. Вычисление основано на таблице сравнимых идентификаторов, где каждый ключ — это слово, а каждое соответствующее значение — массив позиций, где слово встречается во вводе.

```
% python index.py 9 30 < tale.txt
confidence: 2794 23064 25031 34249 47907 48268 48577 ...
courtyard: 11885 12062 17303 17451 32404 32522 38663 ...
evremonde: 86211 90791 90798 90802 90814 90822 90856 ...
expression: 3777 5575 6574 7116 7195 8509 8928 15015 ...
gentleman: 2521 5290 5337 5698 6235 6301 6326 6338   ...
influence: 27809 36881 43141 43150 48308 54049 54067 ...
monseigneur: 85 90 36587 36590 36611 36636 36643     ...
...
```

Для сокращения объема вывода программа `index.py` получает три аргумента командной строки: имя файла и два целых числа. Первое целое число — минимальная длина строки, включаемой в таблицу идентификаторов, а второе — минимальное количество вхождений (среди слов, имеющихся в тексте) для включения в печатный индекс. В упражнениях также есть несколько подобных клиентов для различных практических задач. Например, один из популярных сценариев подразумевает создание нескольких индексов для тех же данных с использованием разных ключей. В нашем примере учетной записи один индекс мог бы использовать для ключей номер счета клиента, а другой — номер счета продавца.

Подобно программе `lookup.py`, вы можете загрузить с сайта книги файлы `index.py` и `bst.py` (реализации таблицы идентификаторов для сравнимых ключей, рассматриваемые позже) и запустить их для разных входных файлов, чтобы получить представление об удобстве таблиц идентификаторов для сравнимых ключей. Если вы сделаете так, то поймете, что программы могут создавать большие индексы для огромных файлов с небольшой задержкой, поскольку каждая операция с таблицей идентификаторов выполняется немедленно.

Одной из причин столь быстрого распространения алгоритмов и реализаций стало широкое разнообразие потребностей клиентов таблиц идентификаторов. С одной стороны, когда таблица идентификаторов или количество выполняемых операций невелики, сработает любая реализация. С другой стороны, таблицы идентификаторов для некоторых приложений настолько огромны, что они организуются как базы данных, располагаемые на внешнем запоминающем устройстве или в веб. В этом разделе мы рассмотрим огромный класс клиентов, подобных `index.py` и `lookup.py`, потребности которых расположены между этими экстремальными случаями, когда необходимо быть в состоянии использовать ассоциативные присвоения для построения и динамической поддержки больших таблиц при непосредственной поддержке большого количества ассоциативных поисков. Получение немедленного ответа для огромных динамических таблиц является одним из классических вкладов алгоритмических технологий.

Реализации элементарной таблицы идентификаторов. Все эти примеры — убедительное доказательство важности таблиц идентификаторов. Реализации таблицы идентификаторов хорошо изучены, с этой целью было разработано множество различных алгоритмов и структур данных, и современные среды программирования (включая Python) оказывают им прямую поддержку. Как обычно, знание работы простых реализаций поможет оценить, выбрать и эффективней использовать более сложные либо реализовать собственные версии для неких непредвиденных ситуаций, с которыми вы можете встретиться.

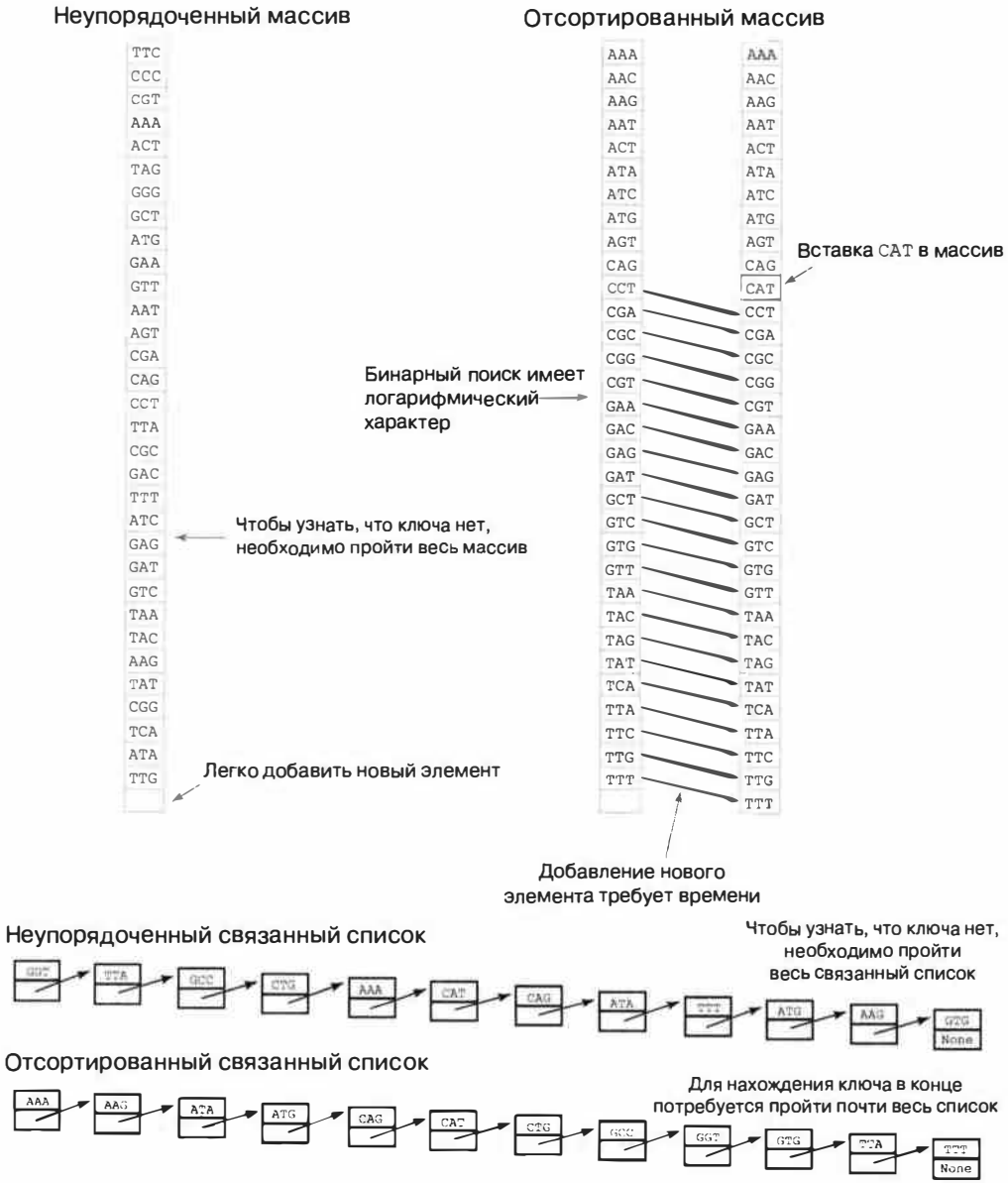
Для начала кратко рассмотрим четыре элементарных реализации, на основании двух уже знакомых структур исходных данных: массивов переменного размера и связанных списков. Наша цель в данном случае — убедиться

в необходимости более сложной структуры данных, поскольку ни одна из этих реализаций не удовлетворяет требованиям производительности из нашего API. Каждая реализация подразумевает линейное время выполнения некоторых операций, что делает их неподходящими для больших практических приложений.

Возможно, самая простая реализация подразумевает использование *последовательного поиска* (sequential search) с двумя *параллельными массивами переменного размера* (parallel (resizing) array), один для ключей и один для значений, следующим образом (реализации методов `__contains__()` и `__iter__()` см. в упр. 4.4.10 и 4.4.33 соответственно):

```
# реализация последовательного поиска в таблице идентификаторов
class SymbolTable:

    def __init__(self):
        self._keys = []
        self._vals = []

    def __getitem__(self, key):
        for i in range(len(self._keys)):
            if self._keys[i] == key:
                return self._vals[i]
        raise KeyError

    def __setitem__(self, key, val):
        for i in range(len(self._keys)):
            if self._keys[i] == key:
                self._vals[i] = val
                return
        self._keys += [key]
        self._vals += [val]
```

Для соответствия линейной стоимости поиска мы могли бы использовать для ключей *отсортированный массив переменного размера.* Фактически мы уже рассматривали идею словаря, когда описывали бинарный поиск в разделе 4.2. Создать реализацию таблицы идентификаторов на основании бинарного поиска не трудно (см. упр. 4.4.5), но такая реализация не подходит для использования с таким клиентом, как `index.py`, поскольку она зависит от поддержания массива переменного размера, отсортированного в порядке ключей. Каждый раз, когда добавляется новый ключ, множество ключей придется сдвинуть на одну позицию выше в массиве, что подразумевает *квадратичную* зависимость полного времени, необходимого для построения таблицы, от размера таблицы.



Реализации простых таблиц идентификаторов (показаны только ключи)

Массив ключей *неупорядочен* — мы просто добавляем новые ключи в конец. Как обычно для списков Python, это (амортизируемая) операция постоянного времени. Но поиск ключа в таблице идентификаторов обычно является операцией *линейного времени*. Например, если мы ищем ключ, отсутствующий в таблице, придется исследовать все ключи.

В качестве альтернативы мы могли бы рассмотреть реализации на базе *неупорядоченного связанного списка* (unordered linked list), так как он позволяет быстро добавлять новые пары "ключ–значение" в начало. Но такая реализация также не подходит для использования типичными клиентами, поскольку единственный способ поиска ключа в связанном списке подразумевает проход по ссылкам, поэтому необходимое полное время поиска является произведением количества поисков и размера таблицы, что также является препятствием. Ассоциация ключа со значением также является замедлением, поскольку сначала придется осуществлять поиски для предотвращения помещения в таблицу идентификаторов дублированных ключей. Даже хранение связанного списка в отсортированном порядке не особенно поможет — например, для добавления нового узла в конец связанного списка все еще потребуется перебор всех ссылок.

Для реализации таблицы идентификаторов, применимой для таких клиентов, как `lookup.py` и `index.py`, необходимы более гибкие и эффективные структуры данных, чем в этих элементарных примерах. Ниже мы рассмотрим два примера таких структур данных: хеш-таблицу и бинарное дерево поиска.

Хеш-таблица (hash table) — это структура данных, в которой ключи разделены на малые группы, где их можно быстро найти. Основная идея проста: мы выбираем параметр m и делим ключи на m групп примерно равного размера. Для каждой группы мы поддерживаем список ключей и используем последовательный поиск, как в только что рассмотренной элементарной реализации.

Для разделения ключей на малые группы мы используем *хеш-функцию* (hash function), сопоставляющую каждый возможный ключ со *значением хеш-функции* (hash value) — целым числом от 0 до $m - 1$. Она позволяет смоделировать таблицу идентификаторов как *массив фиксированной длины из списков* и использовать значение хеш-функции как индекс массива для доступа к желаемому списку. В Python мы можем реализовать и массив фиксированной длины, и списки, используя встроенный тип данных `list`.

Хеширование весьма популярно, и очень многие языки программирования имеют прямую поддержку для него. Как вы увидели в разделе 3.3, Python предоставляет встроенную функцию `hash()`, получающую как аргумент хешируемый объект и возвращающую целочисленный *хеш-код* (hash code). Для его преобразования в значение хеш-функции от 0 до $m - 1$ мы используем выражение

```
hash(x) % m
```

Помните, что объект считается *хешируемым*, если он удовлетворяет трем требованиям:

- объект может быть сравнен на равенство с другими объектами;
- два равных объекта имеют одинаковый хеш-код;
- на протяжении существования объекта его хеш-код не изменяется.

У не равных объектов может быть одинаковый хеш-код. Но для хорошей производительности мы ожидаем, что хеш-функция разделит наши ключи на m групп примерно равной длины.

В таблице ниже представлены хеш-коды и значения для двенадцати строковых ключей при $m = 5$. Хеш-коды и значения хеш-функции в вашей системе могут отличаться из-за различий в реализации функции `hash()`.

С такой подготовкой реализация эффективной таблицы идентификаторов с хешированием сводится к модификации уже рассмотренного кода последовательного поиска. Для ключей мы поддерживаем массив из m списков, где элемент `i` содержит список ключей Python, значения хеш-функции которых — `i`. Для значений мы поддерживаем параллельный массив из m списков, чтобы, найдя ключ, можно было обратиться к соответствующему значению, используя те же индексы. Программа 4.4.3 (`hashst.py`) является полной реализацией на базе фиксированного количества списков m (стандартно 1 024).

Ключ	Хеш-код	Значение
GGT	-6162965092945700575	0
TTA	-2354942681944301382	3
GCC	-6162965092941700414	1
CTG	-1658743042903269101	4
AAA	593367982085446532	2
CAT	-1658743042924269169	1
CAG	-1658743042924269156	4
ATA	593367982106446599	4
TTT	-2354942681944301393	2
ATG	593367982106446593	3
AAG	593367982085446530	0
GTG	-6162965092962700473	2

Эффективность программы `hashst.py` зависит от значения m и качества хеш-функции. С учетом, что хеш-функции разумно распределяет ключи, производительность будет в m раз выше, чем у последовательного поиска, за счет наличия m дополнительных ссылок и списков. Это классический пример *обмена пространства на время*: чем выше значение m, тем больше пространства используется, но тратится меньше времени.

На рисунке ниже представлена таблица идентификаторов, построенная из ключей из примера выше. (Для экономии места мы опускаем на этой схеме емкости массивов переменного размера в списках Python.) Сначала `GGT` вставляется в список 0, затем `TTA` вставляется в список 3, `GCC` вставляется в список 1 и т.д. После создания таблицы поиск `TTT` начинается с вычисления его значения хеш-функции и перебора `_keys[2]`. После нахождения ключа `TTT` в `_keys[2][1]` функция `__getitem__()` возвращает элемент `_vals[2][1]` или `Phenylalnine`.

Программа 4.4.3. Хеш-таблица (hashst.py)

```
import stdio
import stdarray

class SymbolTable:

    def __init__(self, m=1024):
        self._m = m
        self._keys = stdarray.create2D(m, 0)
        self._vals = stdarray.create2D(m, 0)

    def __getitem__(self, key):
        i = hash(key) % self._m
        for j in range(len(self._keys[i])):
            if self._keys[i][j] == key:
                return self._vals[i][j]
        raise KeyError

    def __setitem__(self, key, val):
        i = hash(key) % self._m
        for j in range(len(self._keys[i])):
            if self._keys[i][j] == key:
                self._vals[i][j] = val
                return
        self._keys[i] += [key]
        self._vals[i] += [val]
```

Переменные экземпляра

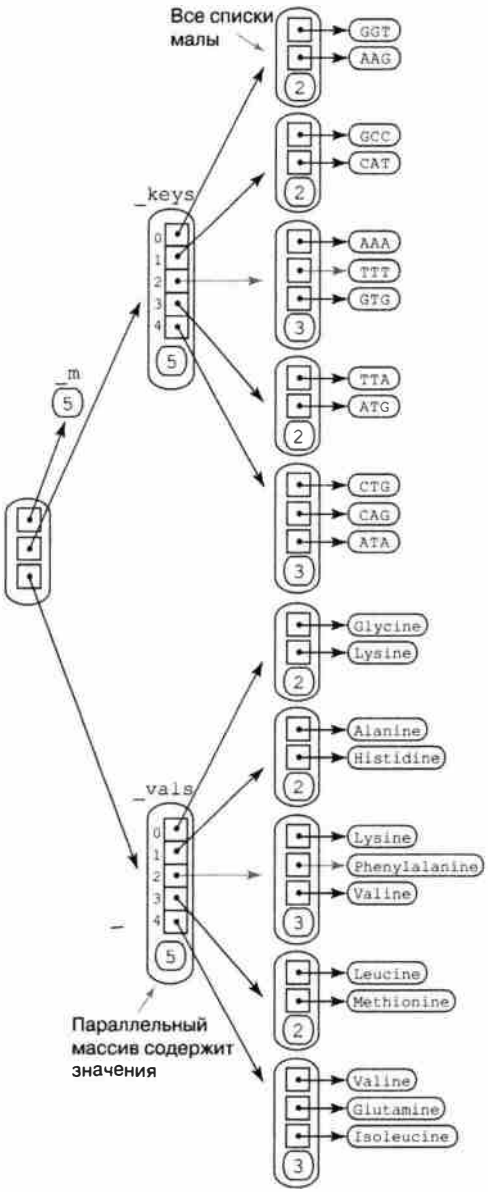
`_m`	Количество списков
`_keys`	Массив списков (ключи)
`_vals`	Массив списков (значения)

Для реализации хеш-таблицы эта программа использует два параллельных массива списков. Каждый список представлен как список Python переменной длины. Хеш-функция выбирает один из `m` списков. Когда в таблице есть `n` ключей, средняя стоимость операции помещения или получения составляет `n/m` для подходящей функции `hash()`. Эта стоимость операции является постоянной, если мы используем массив переменного размера для гарантии, что среднее количество ключей в списке будет от 1 до 8 (см. упр. 4.4.12). Мы оставляем реализации методов `__contains__()` и `__iter__()` для упражнений 4.4.11 и 4.4.34.

Как правило, программисты выбирают большие фиксированные значения m (стандартно выбирается значение 1 024) на основании примерной оценки количества обрабатываемых ключей. Используя массивы переменного размера для `_keys[]` и `_vals[]`, мы можем упорядочить все так, чтобы среднее количество ключей в списке было постоянным. Например, упражнение 4.4.12 демонстрирует,

как гарантировать, что среднее количество ключей в списке всегда останется между 1 и 8, что даст постоянное (амортизируемое) время выполнения операций *помещения* и *получения*. Можно, конечно, откорректировать эти параметры для наилучшего соответствия конкретной практической ситуации.

Основной недостаток хеш-таблиц в том, что они не используют преимуществ упорядочивания ключей, а потому не могут предоставить ключи в отсортированном порядке или обеспечить эффективные реализации таких операций, как поиск минимума или максимума. Например, ключи в программе `index.py` имеют произвольный порядок, а не отсортированный, как нужно. Далее мы рассмотрим реализацию таблицы идентификаторов, способную обеспечить такие операции, когда ключи сравнимы, не жертвуя особо производительностью.



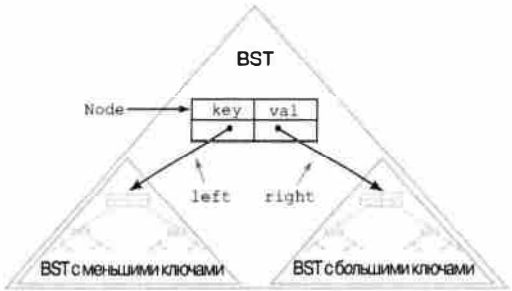
Хеш-таблица ($m = 5$)

Бинарное дерево поиска (binary tree) — это математическая абстракция, играющая центральную роль при эффективной организации информации. Подобно массивам, связанным спискам и хеш-таблицам, мы используем двоичные деревья для хранения коллекций данных. Бинарные деревья играют важную роль в программировании, поскольку они обеспечивают баланс между гибкостью и легкостью реализации.

Для реализации таблицы идентификаторов мы используем специальный тип бинарного дерева, позволяющий организовать данные и предоставить основание для эффективной реализаций операций *помещения* и *получения* из таблицы идентификаторов. *Бинарное дерево поиска* (Binary Search Tree — BST) ассоциирует сравнимые ключи со значениями в структуре, определенной рекурсивно. BST может быть одним из следующего:

- Пусто (None).
- Узел, имеющий пару "ключ–значение" и две ссылки: на BST *слева* (с меньшими ключами) и на BST *справа* (с большими ключами)

Ключи должны быть *сравнимы* оператором <. Тип значения не важен, поэтому узел BST способен содержать любой вид данных в дополнение к ключу и ссылкам на BST. Подобно нашему определению связанного списка в разделе 4.3, концепция рекурсивной структуры данных может потребовать небольшого размышления, однако все, что мы делаем — добавляем вторую ссылку в наше определение связанного списка (и вводим ограничение на упорядочивание).



Бинарное дерево поиска

Реализацию BST мы начинаем с класса для абстракции *узла*, у которой есть ссылки на ключ, значение, левый и правый BST:

```
class Node:
    def __init__(self, key, val):
        self.key   = key
        self.val   = val
        self.left  = None
        self.right = None
```

Это определение похоже на наше определение узлов для связанных списков, за исключением того, что здесь есть *две* ссылки, а не одна. Из рекурсивного определения BST мы можем представить BST переменной типа Node, гарантировав, что его значением будет либо None, либо ссылка на Node, чьи переменные экземпляра left и right являются ссылками на BST, а также гарантировав удовлетворение условию упорядочивания (ключи в левом BST меньше, чем key, а ключи в правом BST больше, чем key).

Результатом вызова Node(key, val) является ссылка на объект Node, переменные экземпляра key и val которого установлены в заданные значения, а переменные экземпляра left и right инициализированы значением None.

Подобно связанным спискам, при трассировке кода, использующего BST, мы можем применять визуальное представление изменений:

- Для представления всех объектов мы рисуем прямоугольники.
- Значения переменных экземпляра помещаем в прямоугольники.
- Ссылки изображаем как стрелки, указывающие на объект, на который ссылаются.

Как правило, мы используем довольно упрощенное абстрактное представление, где для представления узлов рисуем содержащие ключи прямоугольники (пропуская значения) и соединяем их стрелками, представляющими ссылки. Такое абстрактное представление позволяет сосредоточиться на структуре.

Например, для построения BST с одним узлом, ассоциирующим строковый ключ `'it'` с целочисленным значением `0`, мы создаем объект класса `Node`:

```
first = Node('it', 0)
```

Поскольку обе ссылки, левая и правая, `None`, ссылаются на BST, этот узел является BST. Чтобы добавить узел, ассоциирующий ключ `'was'` со значением `1`, создаем другой объект `Node`:

```
second = Node('was', 1)
```

(сам являющийся BST) и связываем с его полем `right` первый объект `Node`:

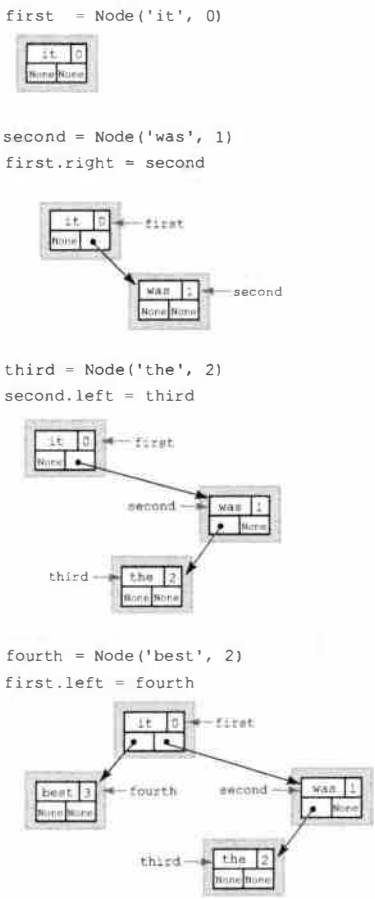
```
first.right = second
```

Второй узел должен располагаться справа от первого, поскольку `'it'` меньше по сравнению с `'was'` (либо мы, возможно, решили бы присвоить `second.left` объекту `first`). Теперь мы можем добавить третий узел, ассоциирующий ключ `'the'` со значением `2`:

```
third = Node('the', 2)
second.left = third
```

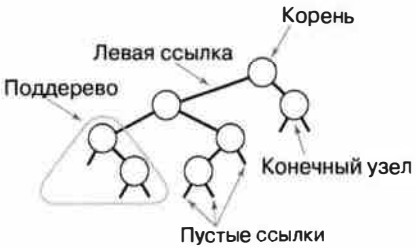
и четвертый узел, ассоциирующий ключ `'best'` со значением `3`:

```
fourth = new Node('best', 3)
first.left = fourth
```



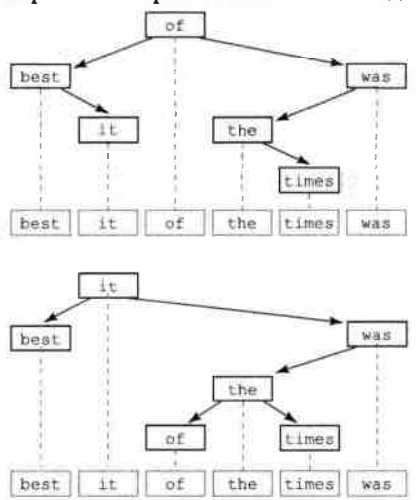
Соединение BST

Обратите внимание, что каждая из наших ссылок (`first`, `second`, `third` и `fourth`) является по определению BST (каждая либо `None`, либо ссылка на BST, и условие упорядочения удовлетворяется для каждого узла).



Анатомия бинарного дерева

При обсуждении BST мы зачастую используем терминологию деревьев. Узел вверху мы называем *корнем* (root) дерева, левую ссылку BST — *левым поддеревом* (left subtree), а его правую ссылку — *правым поддеревом* (right subtree). Традиционно программисты рисуют деревья наоборот, корнями вверх. Узлы с обеими пустыми (null) ссылками называют *конечными узлами*, или *листьями* (leaf). *Высота* (height) дерева — максимальное количество ссылок по любому пути от корневого узла до конечного. У деревьев есть много применений в науке, математике и компьютерных приложениях, поэтому вы наверняка встретитесь с этой моделью во многих случаях.



Два BST, представляющие ту же последовательность

В данном контексте нам следует позаботиться о гарантии, что мы всегда соединяем узлы таким образом, что *каждый* создаваемый объект `Node` будет корневым BST (имеет ключ, значение, ссылку на левый BST с меньшим значением и ссылку на правый BST с бо́льшим значением). С точки зрения структуры данных значение BST несущественно, поэтому мы часто игнорируем его в наших схемах. Мы также преднамеренно путаем свою спецификацию, используя `st` и для "таблиц идентификаторов" (symbol table), и для "дерева поиска" (search tree), поскольку деревья поиска играют важную роль в реализациях таблиц идентификаторов.

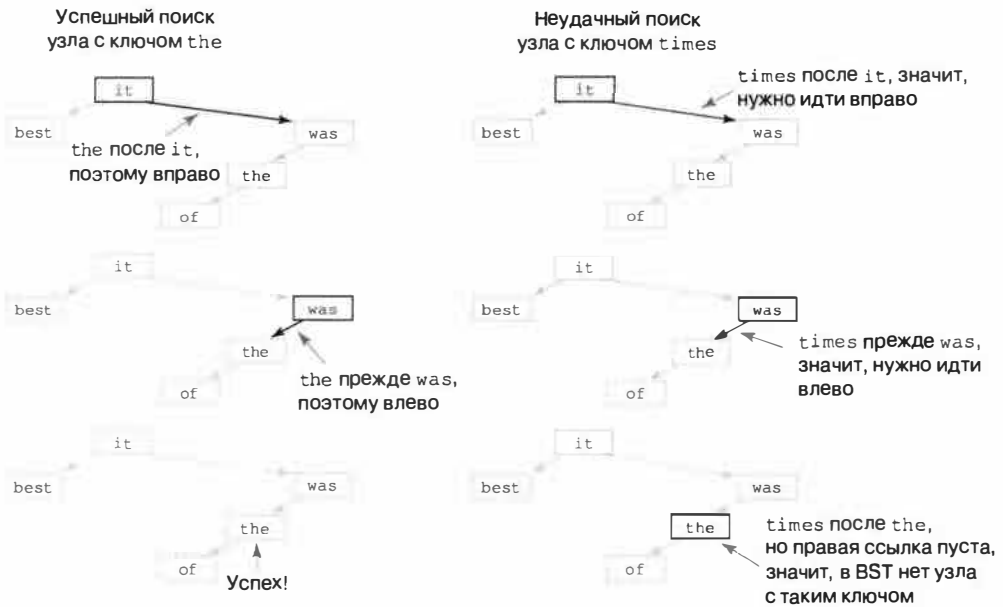
BST представляет *упорядоченную* последовательность элементов. В примере, рассматриваемом на предыдущей странице, `first` представляет последовательность `best it the was`. Как уже упоминалось, для представления упорядоченной последовательности мы могли бы также использовать массив. Например, мы могли использовать код

```
a=['best', 'it', 'the', 'was']
```

для представления той же упорядоченной последовательности строк. Представить набор уникальных ключей как упорядоченный массив можно только одним способом, но представить набор как BST можно многими способами

(см. упр. 4.4.14). Такая гибкость позволяет нам разрабатывать эффективные реализации таблицы идентификаторов. В нашем примере мы были в состоянии вставлять новые элементы, создав новый узел и изменив только одну ссылку. Кроме того, это можно сделать всегда. Не менее важным результатом является возможность легко находить в дереве заданный ключ и место, куда необходимо добавить ссылку на новый узел с данным ключом. Далее мы рассмотрим код таблицы идентификаторов, решающий эти задачи.

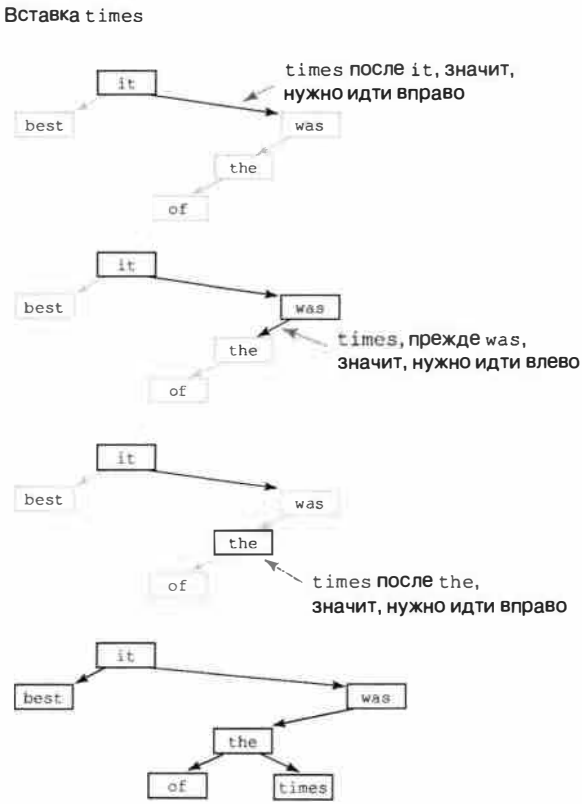
Предположим, вы хотите *искать* в BST узел с заданным ключом (или *получить* значение с заданным ключом из таблицы идентификаторов). Возможны два результата: поиск может быть *успешным* (мы находим ключ в BST; а в реализации таблицы идентификаторов возвращаем ассоциированное значение), а может быть *неудачным* (в BST нет заданного ключа; в реализации таблицы идентификаторов происходит ошибка времени выполнения).



Поиск в BST

Рекурсивный алгоритм поиска очень быстр: дан BST (ссылка на объект `Node`); сначала проверяется, не пусто ли дерево (ссылка не `None`). Если это так, то поиск завершается неудачей (в реализации на базе таблицы идентификаторов происходит ошибка времени выполнения). Если дерево не пусто, проверяется равенство ключа в узле искомому ключу. Если это так, то поиск закончен успешно (в реализации таблицы идентификаторов возвращение значения, ассоциированного с ключом). В противном случае — сравнение искомого ключа с ключом в узле. Если он меньше — рекурсивный поиск в левом поддереве, если больше — в правом.

Размышляя рекурсивно, не трудно убедиться, что этот метод ведет себя как надо (на основании инварианта, что ключ находится в BST), если и только если он находится в текущем поддереве. Ключевое свойство рекурсивного метода в том, что для исследования и принятия решения о последующих действиях у нас всегда есть только один узел. Кроме того, обычно исследуется лишь небольшое количество узлов в дереве: всякий раз, входя в одно из поддеревьев в узле, мы никогда не будем исследовать ни один из узлов в другом поддереве.



Вставка нового узла в BST

Предположим, вы хотите *вставить* новый узел в BST (в реализации на базе таблицы идентификаторов *поместить* новую пару "ключ–значение" в структуру данных). Логика подобна поиску ключа, но реализация сложней. Главное, понять, что для указания на новый узел следует изменить только одну ссылку, именно ту, которая была бы найдена как `None` при неудачном поиске ключа в этом узле.

Если BST пуст, мы создаем и возвращаем новый `Node`, содержащий пару "ключ–значение"; если искомый ключ меньше ключа в корне, то устанавливаем левую ссылку на результат вставки пары "ключ–значение" в левое поддерево; если искомый ключ больше, устанавливаем правую ссылку на результат вставки

пары "ключ–значение" в правое поддерево; в противном случае, если ключ равен искомому, переписываем существующее значение новым. Таким образом, переустановка левой или правой ссылки после такого рекурсивного вызова обычно не нужна, поскольку ссылки изменяются, только если поддерево пусто, но установить ссылку так же просто, как и проверить, чтобы избежать ее переустановки.

Программа 4.4.4. Бинарное дерево поиска (bst.py)

```
class OrderedSymbolTable:
    def __init__(self):
        self._root = None

    def _get(self, x, key):
        if x is None: raise KeyError
        if   key < x.key: return self._get(x.left, key)
        elif x.key < key: return self._get(x.right, key)
        else:             return x.val

    def __getitem__(self, key):
        return self._get(self._root, key)

    def _set(self, x, key, val):
        if x is None: return _Node(key, val)
        if   key < x.key: x.left  = self._set(x.left, key, val)
        elif x.key < key: x.right = self._set(x.right, key, val)
        else:             x.val   = val
        return x

    def __setitem__(self, key, val):
        self._root = self._set(self._root, key, val)

class _Node:
    def __init__(self, key, val):
        self.key   = key
        self.val   = val
        self.left  = None
        self.right = None
```

Переменные экземпляра BST

`_root`	Корень BST

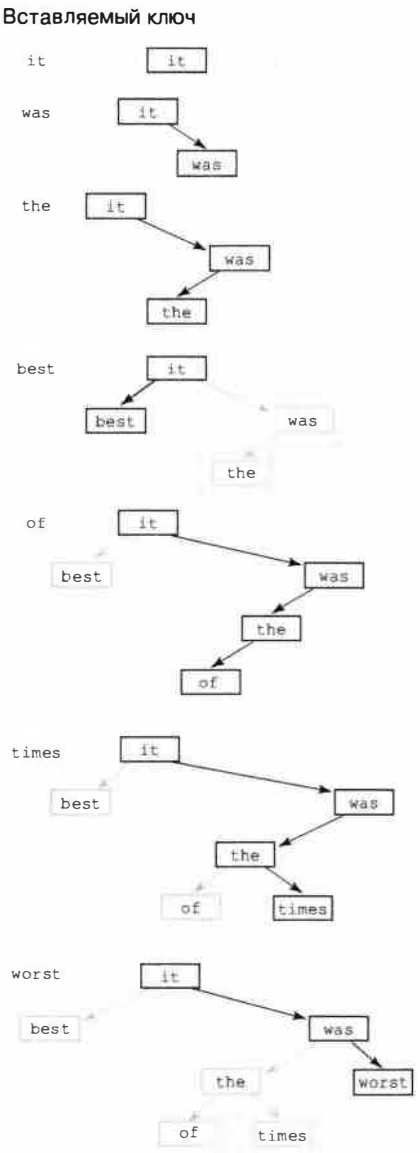
Переменные экземпляра Node

`key`	Ключ
`val`	Значение
`left`	Левое поддерево
`right`	Правое поддерево

Эта реализация типа данных таблицы идентификаторов сосредоточена на рекурсивной структуре данных BST и рекурсивных методах прохода по ней. Реализация метода `__contains__()` отложена для упражнения 4.4.13, а `__iter__()` будет рассмотрена позже.

Программа 4.4.4 (bst.py) является реализацией таблицы идентификаторов на основании этих двух рекурсивных алгоритмов. Подобно программам linkedstack.py и linkedqueue.py, мы используем закрытый класс _Node, чтобы подчеркнуть, что клиенты OrderedSymbolTable не обязаны знать никаких подробностей представления бинарного дерева поиска.

Если вы сравните программу bst.py с нашей реализацией бинарного поиска в программе 4.2.3 (binarysearch.py), а также нашей реализаций стека и очереди в программах 4.3.2 (linkedstack.py) и 4.3.4 (linkedqueue.py), то оцените элегантность и простоту этого кода. *Обдумайте этот код рекурсивно и убедитесь, что он ведет себя как нужно.* Возможно, простейший способ сделать это — проследить за созданием первоначально пустого BST из типичной последовательности ключей (см. схему). Если вы можете сделать это, значит, понимаете данную фундаментальную структуру данных.



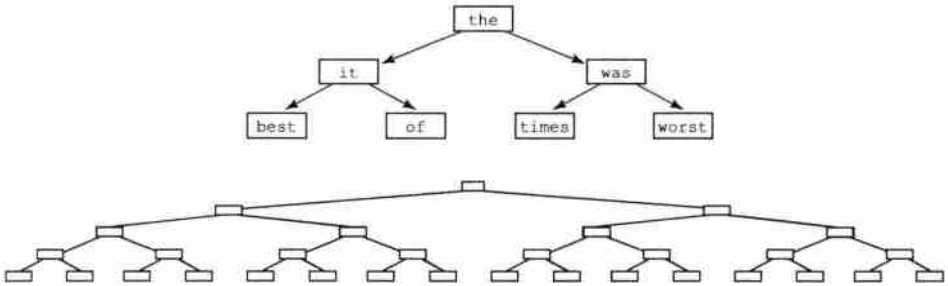
Построение BST

Реализации операций *помещения* и *получения* в BST необычайно эффективны; обычно каждая подразумевает обращение к небольшому количеству узлов в BST (по пути от корня до искомого узла или пустой ссылки, заменяемой ссылкой на новый узел). Далее мы продемонстрируем, что операции *помещения* и запросы на *получение* занимают логарифмическое время (при определенных условиях). Кроме того, чтобы добавить новую пару "ключ–значение" в таблицу идентификаторов, создается и связывается только один новый узел внизу дерева. Если вы нарисуете BST, созданный в результате вставки некоторых ключей в первоначально пустое дерево, то убедитесь, что можете нарисовать каждый новый узел в его уникальной позиции внизу дерева.

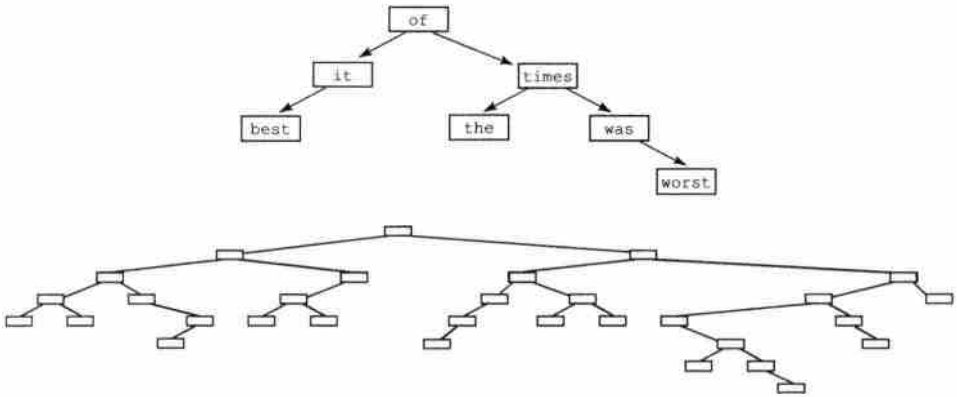
Характеристики производительности BST. Продолжительность алгоритмов BST в конечном счете зависит от формы деревьев, а форма деревьев зависит от порядка добавления ключей. Понимание этой

зависимости — критически важный фактор для способности эффективно использовать BST в практических ситуациях.

Лучший случай. В наилучшем случае дерево отлично сбалансировано (у каждого узла есть точно два не пустых потомка, кроме узлов в самом низу, у которых потомков нет вообще) при lg n узлов между корневым узлом и каждым конечным узлом. В таком дереве довольно просто заметить, что стоимость неудачного поиска является логарифмической, поскольку она удовлетворяет тому же рекуррентному соотношению, что и стоимость бинарного поиска (см. раздел 4.2), согласно которому стоимость каждой операции *помещения* и *получения* пропорциональна lg n или ниже. В действительности нужно быть фантастически удачливым, чтобы получить отлично сбалансированное дерево при вставке ключей один за другим, но наилучшие возможные характеристики производительности стоит знать.



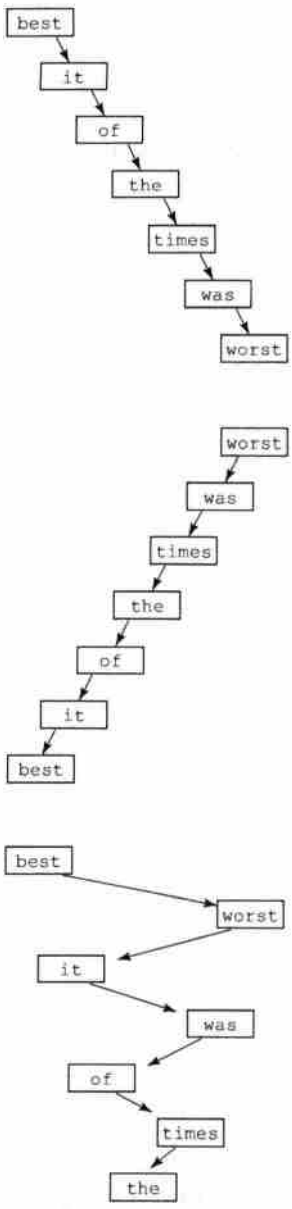
Лучший случай (отлично сбалансированный) BST



Типичный BST, построенный из ключей в случайном порядке

Средний случай. Если мы вставляем случайные ключи, то могли бы ожидать, что время поиска также будет логарифмическим, поскольку первый ключ становится корнем дерева и должен разделить ключи примерно пополам. Применяя тот же аргумент, мы ожидаем получить тот же результат, что и в наилучшем

случае. Это интуитивное предположение действительно подтверждается тщательным анализом: классическое математическое наследование демонстрирует, что время, необходимое для *помещения* и *получения* из дерева, построенного из случайных ключей, является логарифмическим (ссылки см. на сайте книги). Более точно: *ожидаемое количество сравнений ключей составляет ~2 ln n для случайного помещения или получения из дерева, построенного из n случайных ключей.* В практическом приложении, таком как `lookup.py`, когда можно явно перетасовать порядок ключей, этот результат достаточен для (вероятностной) гарантии логарифмической производительности. Действительно, начиная с 2 ln *n* и приблизительно до 1.39 lg *n*, *средний* случай лишь примерно на 39% хуже наилучшего. В таких приложениях, как `index.py`, где нет никакого контроля за порядком вставки, нет никакой гарантии, но типичные данные дают логарифмическую производительность (см. упр. 4.4.26). Как и с бинарным поиском, этот факт весьма существен из-за огромного размера логарифмически-линейной пропасти: с BST-ориентированной реализацией таблицы идентификаторов мы можем выполнять миллионы (и более) операций в секунду, даже в огромной таблице идентификаторов.



Худший случай BST

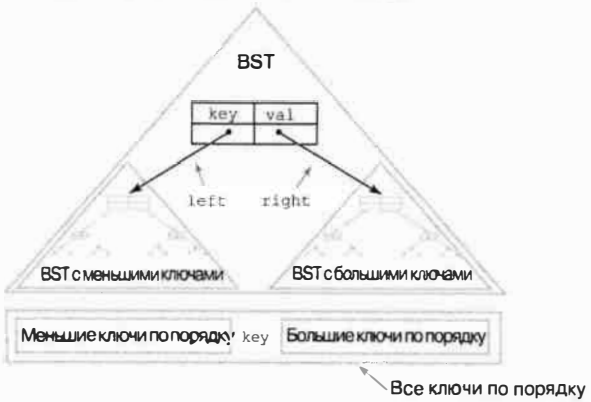
Худший случай. В самом плохом случае у каждого узла есть точно одна пустая ссылка, следовательно, BST походит на связанный список с одной бесполезной дополнительной ссылкой, где операции *помещения* и *получения* занимают линейное время. К сожалению, на практике этот самый плохой случай не редок, он возникает, например, когда вставляют упорядоченные ключи.

Таким образом, хорошая производительность простой реализации BST зависит от ключей, и при их достаточном подобии случайным дерево вряд ли будет содержать много длинных путей. Если вы не уверены, что это предположение близко к истине, *не используйте простой BST*. Единственным свидетельством его неправильности будет увеличение времени ответа при увеличении сложности задачи. (*Примечание.* Это весьма часто встречается в программном обеспечении подобного типа!) К счастью, есть варианты BST, устраняющие этот самый плохой случай и гарантирующие

логарифмическую производительность операций, обеспечивая создание всех деревьев почти отлично сбалансированными. Один из наиболее популярных вариантов — *красно-черное дерево* (red-black tree).

Обход BST. Вероятно, самая простая операция — это *обход дерева* (tree traversal): когда есть дерево (ссылка на него) и необходимо последовательно обработать каждую пару "ключ–значение" в нем. Для связанных списков мы выполняем эту задачу, следуя по единственной ссылке от одного узла к следующему. Но у бинарных деревьев приходится принимать решение, поскольку возможных для следования ссылок *две*. Рекурсия немедленно приходит на помощь. Для обработки каждого ключа в BST необходимо

- обработать каждый ключ в левом поддереве;
- обработать ключ в корне;
- обработать каждый ключ в правом поддереве.



Рекурсивный обход бинарного дерева поиска центрального порядка

Это способ обхода дерева в *центральном порядке* (inorder), отличном от *прямого порядка* (preorder) (корень вначале) и *обратного порядка* (postorder) (корень в конце), используемых в других приложениях. Математическая индукция позволяет довольно просто убедиться, что этот подход позволяет не только обработать каждый ключ в BST, но сделать это в *порядке сортировки ключей* (key-sorted order). Например, следующий метод выводит ключи из переданной в аргументе BST в порядке сортировки ключей:

```
def inorder(x):
    if x is None: return
    inorder(x.left)
    stdio.writeln(x.key)
    inorder(x.right)
```

Это замечательно, но самый простой метод достоин внимательного рассмотрения. Это является основанием реализации функции `str()` для BST

(см. упр. 4.4.23) и отправной точкой для разработки итератора, обеспечивающего клиентам возможность использовать цикл `for` для обработки ключей в порядке сортировки. Действительно, одной из фундаментальных операций с коллекцией является перебор (итерация) ее элементов. Рассматриваемая далее парадигма ведет к четкому, ясному и компактному коду, в значительной степени отдельному от деталей реализации коллекции. В случае BST для этого используется именно эта естественная процедура прохода по дереву.

Возможность итерации. Как уже упоминалось в разделах 1.3 и 1.4, вы можете использовать цикл `for` для перебора по целым числам в диапазоне или по элементам массива `a[]`.

```
for i in range(n):          for v in a:
    stdio.writeln(i)            stdio.writeln(v)
```

Цикл `for` подходит не только для целочисленных диапазонов и массивов — вы можете использовать его с любым *итерируемым* (iterable) объектом. Итерируемым является объект, способный возвращать свои *элементы* по одному. Все типы последовательностей Python (включая `list`, `tuple`, `dict`, `set` и `str`) являются итерируемыми, как и объект, возвращаемый встроенной функцией `range()`.

Теперь наша задача заключается в том, чтобы сделать тип `SymbolTable` итерируемым и позволить использовать цикл `for` для перебора его содержимого по ключам (и индексации для *получения* соответствующих значений):

```
st = SymbolTable()
...
for key in st:
    stdio.writeln(str(key) + ' ' + str(st[key]))
```

Чтобы сделать пользовательский тип данных итерируемым, необходимо реализовать специальный метод `__iter__()` для поддержки встроенной функции `iter()`. Функция `iter()` создает и возвращает *итератор* (iterator), типом данных которого является специальный метод `__next__()`, который Python вызывает в начале каждой итерации цикла `for`.

Хотя это и кажется сложным, но мы можем использовать сокращение на основании того факта, что списки Python являются итерируемыми: если `a` — это список Python, то `iter(a)` возвращает итератор по его элементам. С учетом этого мы можем сделать нашу таблицу идентификаторов для последовательного поиска итерируемой в одну строку:

```
def __iter__():
    return iter(_keys)
```

Точно так же мы можем сделать реализации своей хеш-таблицы и бинарного дерева поиска итерируемыми, собрав ключи в список Python и возвратив для него итератор. Например, чтобы сделать `bst.py` итерируемым, модифицируем рекурсивный метод `inorder()` с предыдущей страницы так, чтобы собрать

ключи в список Python вместо их вывода. Затем мы можем возвратить итератор для этого списка следующим образом:

```
def __iter__(self):
    a=[]
    self._inorder(self._root, a)
    return iter(a)

def _inorder(self, x, a):
    if x is None: return
    self._inorder(x.left, a)
    a += [x.key]
    self._inorder(x.right, a)
```

Python включает несколько встроенных функций, получающих итерируемые объекты как аргументы или возвращающие их, как указано в API ниже.

Предупреждение пользователям Python 2. В Python 3 функция `range()` возвращает итерируемое из целых чисел, а в Python 2 — массив целых чисел.

Большинство этих встроенных операций занимают линейное время, поскольку стандартная реализация просматривает все элементы в итерируемом, чтобы вычислить результат. Конечно, для приложений, где важна производительность при проверке наличия элемента в коллекции, мы использовали бы таблицу идентификаторов вместо встроенной конструкции `in`.

Далее мы увидим, что вполне возможно также разработать реализации для *минимума* и *максимума*, намного более эффективные, чем функции `min()` и `max()`, когда внутренняя структура данных BST. Мы также можем предоставить широкий диапазон других полезных операций.

Итерируемые операции и встроенные функции

Операция	Описание
`for v in a:`	Перебор элементов в `a`
`v in a`	Есть ли элемент `v` в `a`?
`range(i, j)`	Итерируемое, содержащее целые числа `i`, `i+1`, `i+2`, ..., `j-1` (стандартно `i` равно `0`)
`iter(a)`	Новый итератор по элементам в `a`
`list(a)`	Новый список, созданный из элементов в `a`
`tuple(a)`	Новый кортеж, созданный из элементов в `a`
`sum(a)`	Сумма элементов в `a`
`min(a)`	Минимальный элемент в `a`
`max(a)`	Максимальный элемент в `a`
`reversed(a)`	Итерируемое, содержащее элементы последовательности `a` в обратном порядке

Примечание. Итерируемые элементы должны быть числовыми для `sum()` и сравнимыми для `min()` и `max()`.

Операции упорядоченной таблицы идентификаторов. Гибкость BST и способность сравнивать ключи позволяет реализовать многие полезные операции, кроме тех, которые можно эффективно обеспечить в хеш-таблицах. Мы оставляем реализации этих операций для упражнений, а дальнейшее исследование их характеристик производительности и областей применения — для курса по структурам данных и алгоритмам.

Минимум и максимум. Для поиска наименьшего ключа в BST следуйте по левым ссылкам от корня до `None`. Последний встреченный ключ и является наименьшим в BST. Та же процедура для правых ссылок приводит к наибольшему ключу в BST (см. упр. 4.4.27).

Размер дерева и поддеревьев. Для отслеживания количества узлов в BST используйте в `OrderedSymbolTable` дополнительную переменную экземпляра `n`, подсчитывающую количество узлов в дереве. Инициализируйте ее значением 0 и увеличивайте его при создании каждого нового `_Node`. В качестве альтернативы можно хранить дополнительную переменную экземпляра `n` в каждом `_Node` и подсчитывать количество узлов в поддереве, подключенном к тому узлу (см. упр. 4.4.29).

Поиск в диапазоне и подсчет диапазонов. Обладая таким рекурсивным методом, как `_inorder()`, мы можем возвратить итератор для ключей, расположенных между двумя заданными значениями, за время, пропорциональное высоте BST, плюс количество ключей в диапазоне (см. упр. 4.4.30). Если мы поддерживаем в каждом узле переменную экземпляра, хранящую размер подключенного к нему поддерева, то можем *подсчитать* количество ключей, падающих между двумя заданными значениями за время, пропорциональное высоте BST (см. упр. 4.4.31).

Статистика рангов и ранги. Если мы поддерживаем в каждом узле переменную экземпляра, хранящую размер подключенного к нему поддерева, то можем реализовать рекурсивный метод, возвращающий k-й наименьший ключ за время, пропорциональное высоте BST (см. упр. 4.4.64). Точно так же мы можем вычислить *ранг* (rank) ключа, т.е. количество ключей в BST, строго меньших данного (см. упр. 4.4.65).

Удаление. Множеству приложений требуется способ удалить пару "ключ–значение" с заданным ключом. Вы можете найти код для удаления узла из BST на сайте книги или в книге по алгоритмам и структурам данных (см. упр. 4.4.32).

Этот список дает лишь общее представление; для BST было разработано множество других важных операций, широко используемых в приложениях.

Тип данных словаря. Теперь, понимая работу таблицы идентификаторов, вы готовы использовать промышленную версию Python. Встроенный тип данных `dict` следует тем же базовым API, что и `SymbolTable`, но с улучшенным набором операций, включая *удаление*; возвращающую стандартное значение версию функции *получения*, если ключа нет в словаре; и *перебора* пар "ключ–значение".

В основе реализации лежит хеш-таблица, поэтому операции упорядочивания не поддерживаются. Как обычно, поскольку Python написан на низкоуровневом языке и почти не имеет служебных затрат, налагаемых им самим на всех свих пользователей, эта реализация будет эффективней и предпочтительней, если операции упорядочивания не важны.

Рассмотрим простой пример. Клиент типа `dict` читает последовательность строк со стандартного ввода, подсчитывает количество вхождений каждой строки, а затем выводит строки и их частоты. Строки выводятся в *не* сортированном порядке:

```
import stdio
while not stdio.isEmpty():
    word=stdio.readString()
    st[word]=1+st.get(word, 0)
for word, frequency in st.iteritems():
    stdio.writef('%4d %s\n', word, frequency)
```

Упражнения в конце этого раздела содержат несколько примеров клиентов типа `dict`.

Часть API для встроенного типа данных Python `dict`

Операции постоянного времени	Описание
`dict()`	Новый пустой словарь
`st[key]=val`	Ассоциирует `key` с `val` в `st`
`st[key]`	Значение, ассоциированное с `key` в `st` (если такого ключа `key` в `st` нет, передается `KeyError`)
`st.get(key, x)`	`st[key]`, если `key` есть в `st`; в противном случае `x` (стандартно `x` равно `None`)
`key in st`	Есть ли `key` в `st`?
`len(st)`	Количество пар "ключ–значение" в `st`
`del st[key]`	Удаляет `key` (и ассоциированное с ним значение) из `st`
Операции линейного времени	
`for key in st:`	Перебор по ключам в `st`

Тип данных набора. В качестве заключительного примера рассмотрим тип данных, который хоть и проще, чем таблица идентификаторов, но не менее полезный, популярный и простой в реализации на базе хеширования или BST. *Набор* (set) — это коллекция уникальных ключей, как и таблица идентификаторов, но без значений. Например, мы могли реализовать набор, удалив ссылки на значения из `hashst.py` или `bst.py` (см. упр. 4.4.20–4.4.21). И снова Python предоставляет тип данных `set`, реализованный на низкоуровневом языке. Часть его API представлена ниже.

Рассмотрим, например, задачу чтения последовательности строк со стандартного ввода и вывода первого вхождения каждой строки (удаляя таким образом дубликаты). Мы могли бы использовать тип `set`, как в следующем клиентском коде:

```
import stdio
distinct = set()
while not stdio.isEmpty():
    key = stdio.readString()
    if key not in distinct:
        distinct.add(key)
        stdio.writeln(key)
```

Вы можете найти несколько других примеров клиентов типа `set` в упражнениях в конце этого раздела.

Часть API для встроенного типа данных Python `set`

Операции постоянного времени	Описание
`set()`	Новый пустой набор
`s.add(item)`	Добавляет `item` в `s` (если его нет в наборе)
`item in s`	Есть ли `item` в `s`?
`len(s)`	Количество элементов в `s`
`s.remove(item)`	Удаляет `item` из `s`
Операции линейного времени	
`for item in s:`	Перебор по элементам в `s`
`s.intersection(t)`	Пересечение `s` и `t`
`s.union(t)`	Объединение `s` и `t`
`s.issubset(t)`	Является ли `s` подмножеством `t`?

Перспектива. Реализации таблицы идентификаторов — главная тема подробного исследования в области структур данных и алгоритмов. Различные API и различные предположения о ключах требуют разных реализаций. При разных обстоятельствах доступны методы даже с лучшей производительностью, чем хеширование и BST (например, сбалансированные BST и деревья). Реализации большинства этих алгоритмов и структур данных представлены и в Python, и в большинстве других систем. Исследователи алгоритмов и структур данных все еще изучают реализации таблицы идентификаторов всех сортов.

Какая реализация таблицы идентификаторов лучше, хеширование или BST? Сначала необходимо удостовериться, что клиент имеет сравнимые ключи и нуждается в операциях с таблицей идентификаторов, задействующих такие упорядоченные операции, как выбор и ранг. Если да, то необходимо использовать BST. В противном случае большинство программистов, вероятно, воспользуются хешированием, хотя этот выбор потребует осторожности в двух отношениях: (1) необходима хорошая хеш-функция для типа данных ключа и (2) необходимо удостовериться в правильности установки размера хеш-таблицы либо

с использованием массивов переменного размера, либо иных соответствующих конкретному случаю средств.

Должны ли вы использовать встроенные типы данных Python `dict` и `set`? Конечно, если они поддерживают необходимые операции, поскольку они написаны на низкоуровневом языке и не имеют непроизводительных затрат, налагаемых Python на свой пользовательский код, а потому, вероятно, они будут быстрее, чем код, реализованный вами самостоятельно. Но если ваше приложение нуждается в таких основанных на порядке операциях, как поиск минимума или максимума, то рассмотрите возможность применения BST.

Люди используют словари, индексы и другие виды таблиц идентификаторов каждый день. Приложения на основании таблиц идентификаторов полностью заменили телефонные книги, энциклопедии и всякого рода физические устройства, верой и правдой служивших нам на протяжении последнего тысячелетия. Без реализаций таблицы идентификаторов на основании таких структур данных, как хеш-таблицы и BST, подобные приложения были бы невозможны; а с ними возникает чувство, что необходим только доступ к Интернету.

Вопросы и ответы

Могу ли я использовать массив (или список Python) как ключ в `dict` или `set`?

Нет, встроенный тип данных `list` изменяем, поэтому вы не должны использовать массивы как ключи в таблице идентификаторов или наборе. Фактически списки Python не хешируемы, поэтому вы не можете использовать их как ключи в `dict` или `set`. Встроенный тип данных `tuple` неизменен (и хешируем), поэтому вы можете использовать его вместо них.

Почему мой пользовательский тип данных не работает с `dict` или `set`?

Стандартно пользовательские типы хешируемы, `hash(x)` возвращает `id(x)` и `==`, проверяет ссылочное равенство. Хотя эти стандартные реализации удовлетворяют требованиям хеширования, они редко обеспечивают поведение, требуемое в реальном случае.

Почему я не могу возвратить список Python непосредственно в специальном методе `__iter__()`? Почему я должен вместо этого вызвать встроенную функцию `iter()` со списком Python в качестве аргумента?

Список Python — итерируемый объект (поскольку у него есть метод `__iter__()`, возвращающий итератор), но это не итератор.

Какую структуру данных Python использует для реализации `dict` и `set`?

Python использует *открыто адресуемую* (open-addressing) хеш-таблицу — "кузину" хеш-таблицы на базе связанного списка (separate chaining), рассматриваемой в этом разделе. Реализация Python чрезвычайно высоко оптимизирована и написана на низкоуровневом языке программирования.

Оказывает ли Python языковую поддержку для определения объектов типа `set` и `dict`?

Да, вы можете определить объект типа `set`, заключив в фигурные скобки разделяемый запятыми список его элементов. Вы можете определить объект типа `dict`, заключив в фигурные скобки разделяемый запятыми список его пар "ключ–значение", с двоеточием между каждым ключом и ассоциированным с ним значением.

```
stopwords = {'and', 'at', 'of', 'or', on', 'the', 'to'}
grades = {'A+':4.33, 'A':4.0, 'A-':3.67, 'B+':3.33, 'B':3.0}
```

Предоставляет ли Python встроенный тип данных для упорядоченной таблицы идентификаторов (или упорядоченного набора), обеспечивающий упорядоченную итерацию, статистику рангов и поиск диапазона?

Нет. Если необходима лишь упорядоченная итерация (по сравнимым ключам), то можно использовать тип данных Python `dict` и отсортировать ключи (и заплатить производительностью за сортировку). Например, если в программе `index.py` использовать тип `dict` вместо бинарного дерева поиска, то выводимые ключи можно упорядочить в порядке сортировки, используя такой код, как

```
for word in sorted(st):
```

Если для таблицы идентификаторов необходимы другие операции упорядочивания (такие, как поиск диапазона или статистика рангов), можете использовать нашу реализацию бинарного дерева поиска (и заплатить производительностью за использование типа данных, реализованного на языке Python).

Упражнения

4.4.1. Измените программу `lookup.py` так, чтобы обеспечить программе `lookupandput.py` возможность операций *помещения* со стандартного ввода. Используйте соглашение, согласно которому знак "плюс" означает, что следующие две вводимые строки являются парой "ключ–значение" для вставки.

4.4.2. Измените программу `lookup.py` так, чтобы программа `lookupmultiple.py`, обрабатывающая несколько значений с тем же ключом, собирала значения в массиве, как в `index.py`, а затем выводила их всех при запросе на *получение* следующим образом:

```
% python lookupmultiple.py amino.csv 3 0
Leucine
TTA TTG CTT CTC CTA CTG
```

4.4.3. Измените программу `index.py` так, чтобы программа `indexbykeyword.py` получала из командной строки имя файла и создавала индекс из стандартного ввода, используя только ключевые слова в этом файле. *Примечание.* Использование того же файла для индексации и ключевых слов должно дать тот же результат, что и `index.py`.

4.4.3. Измените программу `index.py` так, чтобы программа `indexlines.py` учитывала как ключи только последовательные наборы символов (никаких знаков пунктуации или чисел) и использовала номера строк вместо позиций слов в качестве значений. Эти функциональные возможности полезны для программ: когда на вводе дана программа Python, программа `indexlines.py` должна выводить индекс, представляющий каждое ключевое слово или идентификатор в программе (в отсортированном порядке), наряду с номерами строк, в которых они встречаются.

4.4.5. Разработайте реализацию API таблицы идентификаторов `OrderedSymbolTable`, поддерживающую параллельные массивы ключей и значений, храня их в отсортированном по ключу порядке. Используйте бинарный поиск для *получения* и перемещения больших элементов вправо на одну позицию при *помещении* (для сохранения линейной зависимости длины массива от количества пар "ключ–значение" в таблице используйте массивы переменного размера). Проверьте свою реализацию с программой `index.py` и подтвердите гипотезу, что использование такой реализации для `index.py` занимает время, пропорциональное произведению количества строк на количество уникальных строк во вводе.

4.4.6. Разработайте реализацию API таблицы идентификаторов `LinkedSymbolTable`, поддерживающую связанный список узлов, содержащих ключи и значения в произвольном порядке. Проверьте свою реализацию с программой `index.py` и проверяют гипотезы, что использование такой реализации для `index.py`, и подтвердите гипотезу, что использование такой реализации для `index.py` занимает время, пропорциональное произведению количества строк на количество уникальных строк во вводе.

4.4.7. Вычислите `hash(x) % 5` для ключей из одного символа

```
E A S Y Q U E S T I O N
```

Нарисуйте созданную хеш-таблицу, как в тексте, когда `i`-й ключ в этой последовательности ассоциируется со значением `i` для `i` от 0 до 11.

4.4.8. Что не так со следующей реализацией функции `__hash__()`?

```
def __hash__(self):
    return -17
```

Решение. Хотя технически это удовлетворяет условию для хешируемого типа данных (если два объекта равны, у них одинаковое значение хеш-функции), фактически она дает плохую производительность, поскольку мы ожидаем, что `hash(x) % m` разделит ключи на `m` групп примерно равного размера.

4.4.9. Дополните типы `Complex` (программа 3.2.6) и `Vector` (программа 3.3.3) так, чтобы они стали хешируемы, реализовав специальные методы `__hash__()` и `__eq__()`.

4.4.10. Реализуйте специальный метод `__contains__()` для таблицы идентификаторов последовательного поиска на странице 630.

4.4.11. Реализуйте специальный метод `__contains__()` для `hashst.py`.

4.4.12. Измените программу `hashst.py` так, чтобы, использовав массив переменного размера, обеспечить среднюю длину списка, ассоциируемого с каждым значением хеш-функции, между 1 и 8.

4.4.13. Реализуйте специальный метод `__contains__()` для `bst.py`.

4.4.14. Нарисуйте все BST, способные представить следующую последовательность ключей:

```
best of it the time was
```

4.4.15. Нарисуйте BST, полученное после вставки в первоначально пустое дерево элементов со следующими ключами:

```
E A S Y Q U E S T I O N
```

в этом порядке. Какова высота получившегося BST?

4.4.16. Предположим, в BST имеются целочисленные ключи от 1 до 1000, а найти необходимо 363. Какая из следующих последовательностей *невозможна* при исследовании ключей?

a. 2 252 401 398 330 363

b. 399 387 219 266 382 381 278 363

c. 3 923 220 911 244 898 258 362 363

d. 4 924 278 347 621 299 392 358 363

e. 5 925 202 910 245 363

4.4.17. Предположим, что 31 следующий ключ расположен (в том же порядке) в BST высотой 5:

```
10 15 18 21 23 24 30 30 38 41 42 45 50 55 59
60 61 63 71 77 78 83 84 85 86 88 91 92 93 94 98
```

Нарисуйте три верхних узла дерева (корневой и два его потомка).

4.4.18. Опишите воздействие на производительности замены `hashst` на `bst` в `lookup.py`. Для защиты от самого плохого случая вызовите `stdrandom.shuffle(database)`, прежде обработкой клиентских запросов.

4.4.19. Истина или ложь: дано BST, где x — конечный узел; p — его родитель. Затем либо (i) ключ p, — наименьший ключ в BST, больше, чем ключ x, либо (ii) ключ p является наибольшим ключом в BST, меньшим, чем ключ x.

4.4.20. Измените класс `SymbolTable` в программе `hashst.py` так, чтобы класс `Set` реализовал операции постоянного времени из приведенного в тексте частичного API для встроенного типа данных Python `set`.

4.4.21. Измените класс `OrderedSymbolTable` в программе `bst.py` так, чтобы класс `OrderedSet` реализовал операции постоянного времени из приведенного в тексте частичного API для встроенного типа данных Python `set`, подразумевая, что ключи сравнимы.

4.4.22. Измените программу `hashst.py` так, чтобы поддерживать клиентский код `del st[key]`, добавив метод `__delitem__()`, получающий аргумент `key` и удаляющий этот ключ (и соответствующее значение) из таблицы идентификаторов, если он существует. Как и в упражнении 4.4.12, используйте массив переменного размера для гарантии, что средняя длина списка, ассоциируемого с каждым значением хеш-функции, останется между 1 и 8.

4.4.23. Реализуйте метод `__str__()` для `bst.py`, используя рекурсивный вспомогательный метод `traverse()`. Как обычно, можете согласиться на квадратичную производительность из-за стоимости конкатенации строк. *Дополнительное*

задание. Составьте для `bst.py` линейный метод `__str__()`, использующий массив и метод `join()` встроенного типа данных Python `str`.

4.4.24. *Конкордация* — это алфавитный список всех слов в тексте со всеми их позициями. Таким образом, `python index.py 0 0` дает конкордацию. В известном случае одна группа исследователей попыталась установить вероятность достоверности свитков Мертвого моря, сравнив их конкордацию с таковыми у других. Составьте программу `invertconcordance.py`, получающую аргумент командной строки `n`, читающую конкордацию со стандартного ввода и выводящую `n` первых слов соответствующего текста.

4.4.25. Проведите эксперименты и проверьте заявления в тексте, что операции *помещения* и *получения* в `lookup.py` занимают постоянное время при использовании `hashst.py` с массивами переменного размера, как описано в упражнении 4.4.12. Разработайте клиенты проверки, создающие случайные ключи, а также проведите проверки для разных наборов данных, взятых с сайта книги, или по вашему собственному выбору.

4.4.26. Проведите эксперименты и проверьте заявления в тексте, что операции *помещения* и *получения* в `index.py` имеют логарифмическую зависимость от размера таблиц идентификаторов при использовании `bst.py`. Разработайте клиенты проверки, создающие случайные ключи, а также проверьте различные наборы данных, взятых с сайта книги, или по вашему собственному выбору.

4.4.27. Измените программу `bst.py`, добавив методы `min()` и `max()`, возвращающие наименьший (или наибольший) ключ в таблице (или `None`, если таблица пуста).

4.4.28. Измените программу `bst.py`, добавив методы `floor()` и `ceiling()`, получающие как аргумент ключ и возвращающие наибольший (наименьший) ключ в наборе, который не больше (не меньше) заданного ключа.

4.4.29. Измените программу `bst.py`, добавив специальную функцию `len()`, реализовав специальный метод `__len__()`, возвращающий количество пар "ключ–значение" в таблице идентификаторов. Используйте подход сохранения в пределах каждого `_Node` количества узлов в его поддереве.

4.4.30. Измените программу `bst.py`, добавив методы, получающие как аргументы два ключа, `lo` и `hi`, а возвращающие итератор по всем ключам, располагающимися между `lo` и `hi`. Продолжительность должна быть пропорциональна высоте плюс количество ключей в диапазоне.

4.4.31. Измените программу `bst.py`, добавив метод `rangeCount()`, получающий как аргументы ключи и возвращающий количество ключей в BST между этими двумя ключами. Ваш метод должен занимать время, пропорциональное высоте дерева. *Подсказка*: сначала закончите предыдущее упражнение.

4.4.32. Измените программу `bst.py`, чтобы она поддерживала такой клиентский код, как `del st[key]`, добавив метод `__delitem__()`, получающий как аргумент `key` и удаляющий этот ключ (и соответствующее значение) из таблицы идентификаторов, если он есть. *Подсказка*: эта операция труднее, чем могло бы показаться. Замените ключ и его ассоциированное значение следующим наибольшим ключом в BST и его ассоциированным значением, а затем удалите из BST узел, содержащий следующий наибольший ключ.

4.4.33. Реализуйте специальный метод `__iter__()` для таблицы идентификаторов последовательного поиска на стр. 630.

4.4.34. Реализуйте в программе `hashst.py` специальный метод `__iter__()` для поддержки итерации.

Решение. Соберите все ключи в список.

```
def __iter__(self):
    a=[]
    for i in range(self._m):
        a += self._keys[i]
    return iter(a)
```

4.4.35. Измените API таблицы идентификаторов так, чтобы обрабатывать значения с дублированными ключами при наличии метода `get()`, возвращающего *итератор* для значений, имеющих данный ключ. Повторно реализуйте программы `hashst.py` и `bst.py` согласно этому API. Обсудите преимущества и недостатки этого подхода по сравнению с представленным в тексте.

4.4.36. Предположим, что `a[]` — массив хешируемых объектов. Каково действие следующего оператора?

```
a=list(set(a))
```

4.4.37. Переделайте программы `lookup.py` и `index.py`, используя тип `dict` вместо `hashst.py` и `bst.py` соответственно. Сравните производительность.

4.4.38. Составьте клиент типа `dict`, который создает таблицу идентификаторов, сопоставляющую символы оценок с числовыми балами, как в таблице ниже, а затем читает со стандартного ввода список символов оценок и вычисляет их среднее (GPA).

A+	A	A-	B+	B	B-	C+	C	C-	D	F
4.33	4.00	3.67	3.33	3.00	2.67	2.33	2.00	1.67	1.00	0.00

4.4.39. Реализуйте методы `buy()` и методы `sell()` в программе 3.2.8 (`stockaccount.py`). Для хранения количества долей каждой акции используйте тип `dict`.

Упражнения на бинарное дерево

Следующие упражнения призваны помочь приобрести практический опыт работы с бинарными деревьями, не обязательно являющиеся BST. Все они подразумевают класс `Node` с тремя переменными экземпляра: положительное значение типа `double` и две ссылки на `Node`. Подобно связанным спискам, имеет смысл делать рисунки, используя визуальное представление описанного в тексте.

4.4.40. Реализуйте следующие функции, каждая из которых получает как аргумент `Node`, являющийся корнем бинарного дерева.

`size(node)`	Количество узлов в дереве с корнем в `node`
`leaves(node)`	Количество узлов в дереве с корнем в `node`, обе ссылки которого `None`
`total(node)`	Сумма значений ключей во всех узлах дерева с корнем в `node`

Все решения должны быть методами линейного времени.

4.4.41. Реализуйте функцию линейного времени `height()`, возвращающую максимальное количество узлов на любом пути от корневого до конечного узла (высота пустого дерева — 0; высота дерева с одним узлом — 1).

4.4.42. У бинарного дерева *в порядке кучи* (heap-ordered) корневой ключ больше ключей во всех его потомках. Реализуйте функцию `heapOrdered()` линейного времени, возвращающую значение `True`, если дерево упорядочено как куча, и `False` в противном случае.

4.4.43. Дано бинарное дерево из поддеревьев *уникальных значений* (single-value) — максимальных поддеревьев, содержащих то же значение. Разработайте алгоритм линейного времени, подсчитывающий количество поддеревьев уникальных значений в бинарном дереве.

4.4.44. Бинарное дерево *сбалансировано* (balanced), если сбалансированы его поддеревья, а высота его двух поддеревьев отличается максимум на 1. Реализуйте метод линейного времени `balanced()`, возвращающий значение `True`, если дерево сбалансировано, и `False` в противном случае.

4.4.45. Два бинарных дерева *изоморфны* (isomorphic), только если различаются значения их ключей (т.е. у них есть та же форма). Реализуйте функцию

линейного времени `isomorphic()`, получающую как аргументы две ссылки на деревья и возвращающую значение `True`, если они ссылаются на изоморфные деревья, и `False` в противном случае. Затем реализуйте функцию линейного времени `eq()`, которая получает как аргументы две ссылки на деревья и возвращает значение `True`, если они ссылаются на идентичные деревья (изоморфные с теми же значениями ключей), и `False` в противном случае.

4.4.46. Составьте функцию `levelOrder()`, выводящую ключи BST в *порядке уровней* (level order): сначала выведите корень, затем узлы на уровень ниже корня слева направо; потом узлы двумя уровнями ниже корня слева направо и т.д. *Подсказка*: используйте класс `Queue`.

4.4.47. Реализуйте функцию линейного времени `isBST()`, возвращающую значение `True`, если бинарное дерево — BST, и `False` в противном случае. *Решение.* Эта задача немного трудней, чем может показаться. Используйте рекурсивную вспомогательную функцию `_inRange()`, которая получает два дополнительных аргумента, `lo` и `hi`, а возвращает `True`, если бинарное дерево — BST и все его значения находятся между `lo` и `hi`, а для представления наименьшего и наибольшего возможный ключей используйте `None`.

```
def _inRange(node, lo, hi):
    if node is None: return True
    if (lo is not None) and (node.item <= lo): return False
    if (hi is not None) and (hi <= node.item): return False
    if not _inRange(node.left, lo, node.item): return False
    if not _inRange(node.right, node.item, hi): return False
    return True
def _isBST(node):
    return _inRange(node, None, None)
```

Обратите внимание, что эта реализация использует операторы < и <=, тогда как наш код бинарного дерева поиска использует только оператор <.

4.4.48. Вычислите значение, возвращенное функцией `mystery()` для некоторых типовых бинарных деревьях, а затем сформулируйте гипотезу о значении и докажите ее.

```
def mystery(node):
    if node is None: return 1
    return mystery(node.left) + mystery(node.right)
```

Практические упражнения

4.4.49. *Проверка правописания.* Составьте программу `spellchecker.py`, клиент типа `set`, который получает как аргумент командной строки имя содержащего словарь файла, а затем читает со стандартного ввода строки и выводит все строки, отсутствующие в словаре. Вы можете найти файл словаря на сайте книги. *Дополнительное задание*: усовершенствуйте свою программу, чтобы учитывать такие распространенные суффиксы, как *-ing* или *-ed*.

4.4.50. *Исправление правописания.* Составьте программу `spellcorrector.py`, клиент типа `dict`, которая работает как фильтр, заменяющий поступающие на стандартный ввод слова с орфографическими ошибками предложенной заменой, выводя результат на стандартный вывод. Файл, содержащий популярные неправильные написания и исправления, получайте как аргумент командной строки. Его пример можно найти на сайте книги.

4.4.51. *Фильтр веб.* Составьте программу `webblocker.py`, клиент типа `set`, которая получает как аргумент командной строки имя файла, содержащего список нежелательных веб-сайтов, а затем читает строки со стандартного ввода и выводит только те веб-сайты, которых нет в списке.

4.4.52. *Операции с набором.* Добавьте в `OrderedSet` (упр. 4.4.21) методы `union()` и `intersection()`, получающие как аргументы два набора и возвращающие для них объединение и пересечение соответственно.

4.4.53. *Частоты таблицы идентификаторов.* Разработайте тип данных `FrequencyTable`, поддерживающий операции `click()` и `count()`, оба получающие строковые аргументы. Значение типа данных — целое число, отслеживающее количество вызовов операции `click()` с данной строкой в качестве аргумента. Операция `click()` увеличивает счет на 1, а операция `count()` возвращает его значение, возможно, нуль. Клиентами этого типа данных могли бы быть анализаторы веб-трафика, аудиопроигрыватели, подсчитывающие количество использований каждой аудиозаписи, телефонное программное обеспечение, подсчитывающее вызовы, и т.д.

4.4.54. *Поиск диапазона 1D.* Разработайте тип данных, обеспечивающий операции вставки даты, поиска по дате и подсчета количества дат в структуре данных, лежащих в заданном интервале. Используйте тип данных Python `datetime.Date`.

4.4.55. *Поиск интервала без пересечения.* Дан список не пересекающихся интервалов целых чисел. Составьте функцию, которая получает целочисленный аргумент и определяет, в каком интервале, если таковой вообще есть, находится это значение. Например, если даны интервалы 1643–2033, 5532–7643, 8999–10332 и 5666653–5669321, то аргумент 9122 лежит в третьем интервале, а аргумент 8122 не принадлежит ни одному из них.

4.4.56. *Поиск стран по IP.* Составьте клиент типа `dict`, использующий файл данных `ip-tocountry.csv` с сайта книги, для определения страны, откуда поступил данный IP-адрес. В файле данных есть пять полей: начало диапазона IP-адресов, конец диапазона IP-адресов, код страны из двух символов, код страны из трех символов и название страны. Диапазоны IP-адресов не пересекаются. Такой инструмент применяется для обнаружения мошенничества с кредитной карточкой, фильтрации спама, автоматического выбора языка на веб-сайте и анализа журнала веб-сервера.

4.4.57. *Инвертированный индекс веб-страниц с запросами по одному слову.* Дан список веб-страниц. Создайте таблицу идентификаторов слов, содержащихся на веб-страницах. Сопоставьте с каждым словом список веб-страниц, в которых оно встречается. Составьте программу, которая читает список веб-страниц, создает таблицу идентификаторов и возвращает в ответ на запрос отдельного слова список веб-страниц, в которых оно встречается.

4.4.58. *Инвертированный индекс веб-страниц с запросами по несколько слов.* Дополните предыдущее упражнение так, чтобы оно поддерживало запросы из нескольких слов. В данном случае выведите список веб-страниц, содержащих по крайней мере одно вхождение каждого из запрошенных слов.

4.4.59. *Поиск нескольких слов (неупорядоченный).* Составьте программу, которая получает из командной строки `k` ключевых слов, читает со стандартного ввода последовательность слов и выявляет наименьший интервал текста, содержащий все `k` ключевых слов (не обязательно в том же порядке). Частичные слова учитывать не нужно.

4.4.60. *Поиск нескольких слов (упорядоченный).* Повторите предыдущее упражнение, но теперь ключевые слова должны быть в том порядке, как определено.

4.4.61. *Троекратное повторение той же позиции в шахматах.* В шахматной игре, если позиция на доске повторяется три раза подряд при совпадении

ходящей стороны, можно объявлять ничью. Опишите, как вы могли бы проверить это условие, используя компьютерную программу.

4.4.62. *Планировщик для диспетчера.* Диспетчер в известном Северо-Восточном университете недавно запланировал преподавателю лекцию в двух разных аудиториях одновременно. Помогите диспетчеру предотвратить подобные ошибки в будущем, создав метод проверки таких конфликтов. Для простоты считайте, что все аудитории занимают на 50 минут, начиная с 9, 10, 11, 1, 2 или 3 часов.

4.4.63. *Энтропия.* Мы определяем *относительную энтропию* (relative entropy) текстового корпуса из n слов, k из которых отличны, так:

$$E = 1 / (n \lg n)(p_0 \lg(k / p_0) + p_1 \lg(k / p_1) + \ldots + p_{k-1} \lg(k / p_{k-1})),$$

где p_i — доля раз, когда встречается слово i. Составьте программу, которая читает текстовый корпус и выводит относительную энтропию. Преобразуйте все символы в нижний регистр и рассматривайте знаки препинания как отступ.

4.4.64. *Статистика рангов.* Добавьте в программу `bst.py` метод `select()`, получающий целочисленный аргумент `k` и возвращающий k-й наименьший ключ в BST. Размеры поддерева храните в каждом узле (см. упр. 4.4.29). Продолжительность должна быть пропорциональной высоте дерева.

4.4.65. *Запрос ранга.* Добавьте в программу `bst.py` метод `rank()`, получающий как аргумент ключ и возвращающий количество ключей в BST, которые строго меньше заданного. Размеры поддерева храните в каждом узле (см. упр. 4.4.29). Продолжительность должна быть пропорциональна высоте дерева.

4.4.66. *Случайный элемент.* Добавьте в программу `bst.py` метод `random()`, возвращающий случайный ключ. Размеры поддерева храните в каждом узле (см. упр. 4.4.29). Продолжительность должна быть пропорциональна высоте дерева.

4.4.67. *Очередь без дубликатов.* Создайте тип данных, напоминающий очередь, но каждый элемент которой может встречаться только однажды. Игнорируйте запросы на вставку элемента, если он уже есть в очереди.

4.4.68. *Уникальные подстроки заданной длины.* Составьте программу, которая читает текст со стандартного ввода и вычисляет количество уникальных подстрок заданной длины `k` в нем. Например, если введено `CGCGGGCGCG`, то есть пять уникальных подстрок длиной 3: `CGC`, `CGG`, `GCG`, `GGC` и `GGG`. Это

вычисление полезно при сжатии данных. *Подсказка*: используйте строковую часть `s[i:i+k]` для извлечения `i`-й подстроки и вставки в таблицу идентификаторов. Проверьте свою программу на большом геноме с сайта книги и на первых 10 миллионах цифр числа π.

4.4.69. *Обобщенная очередь.* Реализуйте класс, который поддерживает следующий API.

API для обобщенной очереди

Операция	Описание
`GeneralizedQueue()`	Новая обобщенная очередь
`isEmpty()`	Не пуста ли очередь?
`q.enqueue(item)`	Элемент `item` помещается в очередь `q`
`q.dequeue(i)`	Возвращение и удаление `i`-го элемента из старейших элементов

Используйте BST, ассоциирующее k-й вставленный элемент с ключом `k`, а также храните в каждом узле полное количество узлов его поддерева. Для того чтобы найти `i`-й из старейших элементов, ищите `i`-й наименьший элемент в BST.

4.4.70. *Динамическое дискретное распределение.* Создайте тип данных, обладающий двумя операциями: `add()` и `random()`. Метод `add()` должен вставлять новый элемент в структуру данных, если его там не было; в противном случае он должен увеличить свой счет частот на 1. Метод `random()` должен возвратить случайный элемент, где вероятности взвешены по частоте каждого элемента. Используемое пространство должно быть пропорционально количеству элементов.

4.4.71. *Механизм проверки пароля.* Составьте программу, которая получает строку как аргумент командной строки и словарь слов со стандартного ввода, а затем проверяет пригодность строки для пароля. Под пригодностью подразумевается, что строка (1) по крайней мере восемь символов длиной, (2) не слово из словаря, (3) не слово из словаря, сопровождаемое цифрой 0–9 (например, `hello5`), (4) не два слова из словаря, связанные вместе (например, `helloworld`) и (5) ни одно из (2) – (4) на базе слов из словаря в обратном порядке.

4.4.72. *Случайные номера телефонов.* Составьте программу, которая получает как аргумент командной строки `n` и выводит `n` случайных номеров телефонов в формате `(xxx) xxx-xxxx`. Используйте тип `set` для предотвращения

выбора совпадающих номеров. Используйте только допустимые коды городов (файл таких кодов содержится на сайте книги).

4.4.73. *Разреженные векторы.* N-мерный вектор *разрежен* (sparse), если количество его значений, отличных от нуля, мало. Ваша задача — представить вектор с пространством, пропорциональным его не нулевым значениям, и обеспечить возможность суммирования двух разреженных векторов за время, пропорциональное полному количеству не нулевых значений. Реализуйте класс, поддерживающий следующий API.

API для разреженного вектора

Операция	Описание
`SparseVector()`	Новый разреженный вектор
`a[i]=v`	Присвоить `i`-му элементу `a` значение `v`
`a[i]`	`i`-й элемент в `a`
`a+b`	Векторная сумма `a` и `b`
`a.dot(b)`	Произведение векторов `a` и `b`

4.4.74. *Разреженные матрицы.* Матрица *n* на *n* разрежена, если количество ее не нулевых элементов пропорционально *n* (или меньше). Ваша задача — представить матрицу с пространством, пропорциональным *n*, и обеспечить возможность добавления и умножения двух разреженных матриц за время, пропорциональное общему количеству не нулевых значений (возможно, с дополнительным коэффициентом log *n*). Реализуйте класс, поддерживающий следующий API.

API для разреженной матрицы

Операция	Описание
`SparseMatrix()`	Новая разреженная матрица
`a[i][j]=v`	Присвоить элементу в ряду `i` и столбце `j` значение `v`
`a[i][j]`	Элемент в ряду `i` и столбце `j` матрицы `a`
`a+b`	Матричная сумма `a` и `b`
`a * b`	Матричное произведение `a` и `b`

4.4.75. *Изменяемая строка.* Создайте тип данных, поддерживающий такой API для строки. Используйте BST для реализации всех операций за логарифмическое время.

4.4.76. *Операторы присвоения.* Составьте программу для анализа и вычисления результата программ, состоящих из операторов присвоения при полностью заключенных в скобки арифметических выражениях (см. программу 4.3.3), а также вывода результата. Например, если дан следующий ввод:

```
A = 5
B = 10
C = A + B
D = C * C
write(D)
```

то ваша программа должна вывести значение 225. Подразумевается, что все переменные и значения имеют тип `float`. Для отслеживания имен переменных используйте таблицу идентификаторов.

4.4.77. *Таблица использования кодона.* Составьте программу, которая использует таблицу идентификаторов для вывода итоговой статистики для каждого кодона в геноме, полученном со стандартного ввода (частота в тысячах), в следующем формате:

```
UUU 13.2  UCU 19.6  UAU 16.5  UGU 12.4
UUC 23.5  UCC 10.6  UAC 14.7  UGC  8.0
UUA  5.8  UCA 16.1  UAA  0.7  UGA  0.3
UUG 17.6  UCG 11.8  UAG  0.2  UGG  9.5
CUU 21.2  CCU 10.4  CAU 13.3  CGU 10.5
CUC 13.5  CCC  4.9  CAC  8.2  CGC  4.2
CUA  6.5  CCA 41.0  CAA 24.9  CGA 10.7
CUG 10.7  CCG 10.1  CAG 11.4  CGG  3.7
AUU 27.1  ACU 25.6  AAU 27.2  AGU 11.9
AUC 23.3  ACC 13.3  AAC 21.0  AGC  6.8
AUA  5.9  ACA 17.1  AAA 32.7  AGA 14.2
AUG 22.3  ACG  9.2  AAG 23.9  AGG  2.8
GUU 25.7  GCU 24.2  GAU 49.4  GGU 11.8
GUC 15.3  GCC 12.6  GAC 22.1  GGC  7.0
GUA  8.7  GCA 16.8  GAA 39.8  GGA 47.2
```

4.5. Случай из практики: феномен "тесного мира"

Математическая модель, используемая для изучения характера попарных связей между сущностями, известна как *граф* (graph). Графы важны для изучения естественного мира, а также помогают лучше понять и детализировать создаваемые нами сети. От моделей нервной системы в нейробиологии и исследования распространения инфекционных болезней в медицине до разработки телефонных систем графы сыграли критически важную роль в прошлом столетии, включая разработку самого Интернета.

Программы этого раздела...

Программа 4.5.1. Тип данных графа (graph.py) 674
Программа 4.5.2. Использование графа для инверсии индекса (invert.py) 678
Программа 4.5.3. Клиент кратчайших путей (separation.py) 681
Программа 4.5.4. Реализация кратчайших путей (pathfinder.py) 687
Программа 4.5.5. Проверка "тесного мира" (smallworld.py) 692
Программа 4.5.6. Граф "исполнитель–исполнитель" (performer.py) 694

Некоторые графы демонстрируют специфическое свойство, известное как *феномен "тесного мира"* (small-world phenomenon). Возможно, вы знакомы с этим свойством, известным также как *теория шести рукопожатий* (six degrees of separation). Это фундаментальная идея о том, что, хотя у каждого из нас относительно немного знакомых, есть достаточно короткая цепь знакомств (шесть рукопожатий), отделяющая нас друг от друга. Эта гипотеза была экспериментально проверена *Стэнли Милгрэмом* (Stanley Milgram) в 1960-х годах и смоделирована математически Дунканом Ваттсом (Duncan Watts) и Стивеном Строгацом (Stephen Strogatz) в 1990-х. В последние годы оказалось, что принцип важен в огромном разнообразии случаев. Ученые интересуются графами "тесного мира", поскольку они моделируют естественные явления, а инженеров они интересуют при построении сетей, когда они используют в своих интересах естественные свойства графов "тесного мира".

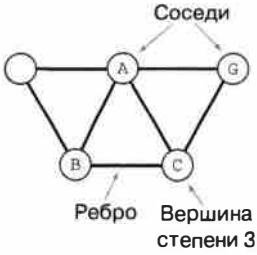
В этом разделе мы решаем простые вычислительные вопросы, связанные с исследованием графов "тесного мира". Действительно, простой вопрос

Демонстрирует ли данный граф феномен «тесного мира»?

может потребовать существенных вычислений. Для ответа на этот вопрос рассмотрим тип данных для обработки графа и несколько полезных клиентов обработки графов. В частности, исследуем клиент вычисления *кратчайшего пути* (shortest path), обладающий огромным количеством важных применений.

Изучаемые алгоритмы и структуры данных играют главную роль в обработке графа. Вы увидите, что некоторые из фундаментальных типов данных, представленных ранее в этой главе, помогут разработать изящный и эффективный код для изучения свойств графов.

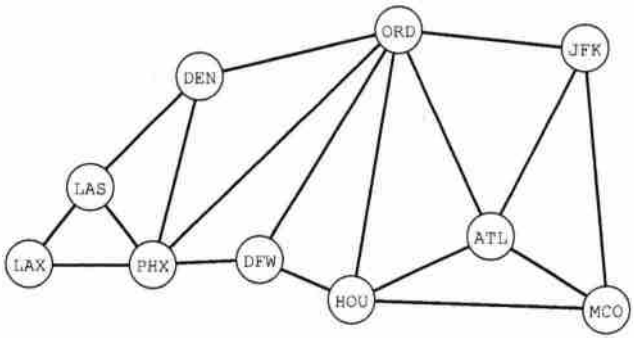
Графы. Начнем с некоторых базовых определений. *Граф* (graph) состоит из набора *вершин* (vertex) и ряда *ребер* (edge). Каждое ребро представляет соединение между двумя вершинами. Две вершины являются *соседями* (neighbor), если они соединяются ребром; *степень* (degree) вершины — это количество ее соседей. Обратите внимание: между концепциями графа, графика функции (графика значений функции) и графики (рисунка) нет ничего общего. Мы зачастую визуализируем графы, рисуя помеченные геометрические фигуры (вершины) и соединяя их линиями (ребра), но всегда важно помнить, что важны сами связи, а не способ их изображения.



Терминология графа

Ниже приведено широкое разнообразие систем, где графы являются отправной точкой для понимания структуры.

Транспортные системы. Железнодорожные рельсы соединяют станции, дороги соединяют перекрестки, а авиалинии соединяют аэропорты, поэтому все эти системы вполне естественно подчиняются простой модели графа. Без сомнения, вы использовали приложения на базе такой модели, ориентируясь по программе интерактивной карты или устройству GPS, либо используя сетевые службы бронирования перед путешествием. Как лучше всего добраться отсюда туда?



Ребра	Вершины
JFK	JFK MCO
MCO	ORD DEN
ATL	ORD HOU
ORD	DFW PHX
HOU	JFK ATL
DFW	ORD DFW
PHX	ORD PHX
DEN	ATL HOU
LAX	DEN PHX
LAS	PHX LAX
	JFK ORD
	DEN LAS
	DFW HOU
	ORD ATL
	LAS LAX
	ATL MCO
	HOU MCO
	LAS PHX

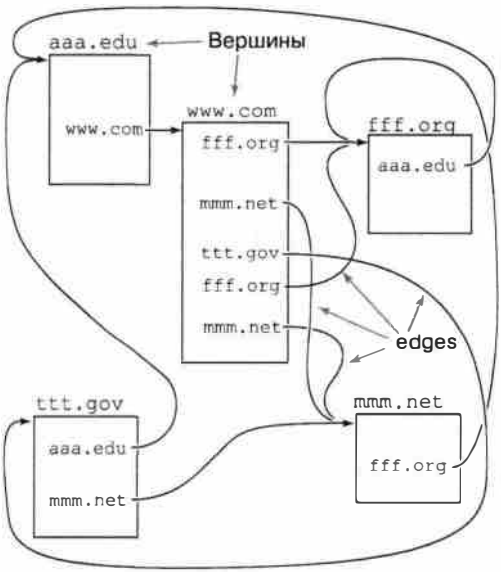
Модель графа транспортной системы

Биология человека. Артерии и вены соединяют органы, синапсы соединяют нейроны, а суставы соединяют кости, таким образом, понимание человеческой биологии зависит от понимания соответствующих моделей графа. Вероятно, наиболее важным и сложным в этой области является моделирование человеческого

мозга. Как локальные соединения между нейронами преобразуются в сознание, память и интеллект?

Социальные сети. Между людьми есть отношения. От исследования инфекционных болезней до исследования политических тенденций, модели графов этих отношений критически важны для понимания их значения. Как распространяется информация по сетям?

Физические системы. Взаимосвязанные атомы формируют молекулы; взаимосвязанные молекулы формируют вещество или кристалл; частицы соединяются взаимными силами, такими как гравитация или магнетизм. Модели графа подходят для изучения задачи просачивания (см. раздел 2.4), взаимодействия зарядов (см. раздел 3.1) и задачи *n*-тел (см. раздел 3.4). Как распространяются локальные взаимодействия таких систем при их развитии?



Вершины
aaa.edu
www.com
mmm.net
fff.org
ttt.gov

Ребра	
aaa.edu	www.com
www.com	fff.org
www.com	mmm.net
www.com	ttt.gov
www.com	fff.org
www.com	mmm.net
mmm.net	fff.org
fff.org	aaa.edu
ttt.gov	aaa.edu
ttt.gov	mmm.net

Модель графа веба

Системы коммуникаций. Все, от электрических цепей в телефонной системе до Интернета и беспроводных систем связи, основано на идее соединения устройств. Начиная с прошлого столетия модели графа играют критически важную роль в разработке таких систем. Каков наилучший путь для соединения устройств?

Распределение ресурсов. Линии электропередачи соединяют электростанции и домашние электрические системы, трубы соединяют резервуары и домашние краны, автодороги соединяют склады и розничные магазины. Исследование эффективных и надежных средств распределения ресурсов зависит от точных моделей графа. Где узкие места в системе распределения?

Механические системы. Стальные тросы и балки соединяют элементы мостов и зданий. Модели графа помогают разработать эти системы и понять их свойства. Какие силы должны выдерживать соединения или балки?

Программные системы. Методы в одном модуле программы вызывают методы из других модулей. Как упоминалось повсюду в этой книге, понимание отношений такого типа является залогом успеха программного проекта. На какие модули будет воздействовать изменение в API?

Финансовые системы. Транзакции соединяют учетные записи клиентов с финансовыми учреждениями. Здесь используется множество моделей графа для изучения сложных финансовых транзакций и получения прибыли. Какие транзакции рутинны, а какие являются показателем существенного события, способного принести прибыль?

Типичные модели графа

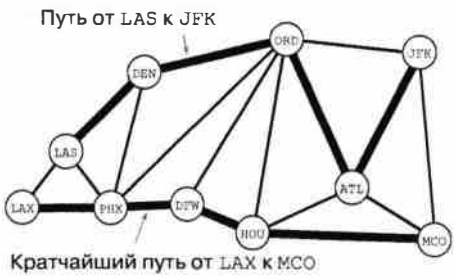
Система	Вершина	Ребро
Естественные явления		
Кровоснабжение	Орган	Кровеносный сосуд
Скелет	Сустав	Кость
Нервы	Нейрон	Синапс
Социум	Личность	Отношения
Эпидемиология	Человек	Инфекция
Химия	Молекула	Связь
n-тело	Частица	Сила
Генетика	Ген	Мутация
Биохимия	Белок	Взаимодействие
Инженерная система		
Транспорт	Аэропорт Перекресток	Рейс Дорога
Коммуникация	Телефон Компьютер Веб-страница	Провод Кабель Ссылка
Распределение	Электростанция Дом Резервуар Дом Склад Розничный магазин	Линия электропередачи Труба Маршрут грузовика
Механика	Соединение	Балка
Программное обеспечение	Модуль	Вызов
Финансы	Учетная запись	Транзакция

Некоторые из этих графов — модели естественных явлений, и наша задача — лучше понять естественный мир, разрабатывая простые модели, а затем используя их для формулирования и проверки гипотез. Другие модели графа относятся

к проектируемым нами сетям, где наша задача — построить лучшую сеть или лучше поддерживать сеть, уяснив ее фундаментальные характеристики.

Графы — весьма полезные модели, независимо от того, малы они или велики. Граф, имеющий лишь несколько дюжин вершин и ребер (например, моделирующий химический состав, где вершины — молекулы, а ребра — связи), уже является достаточно сложным комбинаторным объектом, поскольку возможно огромное количество подобных графов, а следовательно, важно понимание данной конкретной структуры. Граф, имеющий миллиарды или триллионы вершины и ребер (например, правительственная база данных, содержащая все телефонные сообщения, или модель графа человеческой нервной системы), значительно сложней и предполагает существенные вычислительные сложности.

Обработка графов обычно подразумевает их построение на основании информации из базы данных и последующие ответы на вопросы о графе. Кроме специфических для данного приложения вопросов, приведенных в примерах, зачастую приходится задавать вполне простые вопросы о графах. Сколько вершин и ребер имеет граф? Каковы соседи данной вершины? Некоторые вопросы зависят от понимания структуры графа. Например, *путь* (path) в графе — это последовательность вершин, соединенных ребрами. Есть ли путь, соединяющий две заданных вершины? Какой из путей самый короткий? Какова максимальная длина кратчайшего пути в графе (*диаметр* (diameter) графа)? В этой книге мы уже приводили несколько примеров вопросов из научных областей, намного более сложных, чем эти. Какова вероятность, что случайная навигация достигнет каждой вершины? Какова вероятность, что представленная данным графом система проницаема?



Пути в графе

На старших курсах вы встретите сложные системы и графы во многих разных контекстах. Вы также можете подробно изучить их свойства на старших курсах по математике, исследованию операций и информатике. Некоторые задачи по обработке графа выдвигают непреодолимые вычислительные проблемы; другие могут быть решены с относительной легкостью при реализации типов данных, подобных рассматриваемым здесь.

Тип данных графа. Алгоритмы обработки графа обычно сначала строят внутреннее представление графа, добавляя ребра, а затем обрабатывают его, перебирая вершины по соединяющим их ребрам. API ниже обеспечивают такую обработку. Как обычно, этот API отражает несколько возможных проектных альтернатив, и некоторые из них мы сейчас кратко обсудим.

Неориентированный граф. Ребра не имеют ориентации: ребро, соединяющее вершины `v` и `w`, — это то же ребро, соединяющее вершины `w` и `v`. Нас интересуют соединения, а не направления. Для направленных ребер (например, дорог с односторонним движением на карте) требуется немного иной тип данных (см. упр. 4.5.38).

Строковый тип вершины. Мы подразумевали, что вершина — это строка. Мы могли бы использовать более общий тип вершины, позволить клиентам строить графы с объектами любого сравнимого или хешируемого типа. Но мы оставим эти реализации для упражнения 4.5.10, поскольку для рассматриваемых здесь задач строковая вершина вполне подходит.

API для типа данных `Graph`

Операция	Описание
	Операции постоянного времени
`g.addEdge(v, w)`	Добавляет ребро `v-w` в `g`
`g.countV()`	Количество вершин в `g`
`g.countE()`	Количество ребер в `g`
`g.degree(v)`	Количество соседей `v` в `g`
`g.hasVertex(v)`	Принадлежит ли вершина `v` графу `g`?
`g.hasEdge(v, w)`	Принадлежит ли ребро `v-w` графу `g`?
	Операции линейного времени
`Graph(file, delimiter)`	Новый граф из файла `file`, использующего разделитель `delimiter` (стандартное значение `file` — `None`, пустой граф, а `delimiter` — пробел)
`g.vertices()`	Итерируемое для вершины `g`
`g.adjacentTo(v)`	Итерируемое для соседей вершины `v` в `g`
`str(g)`	Строковое представление `g`

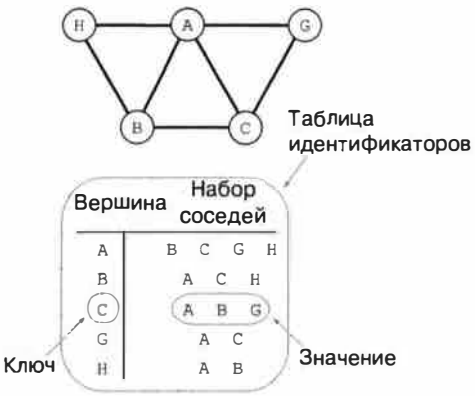
Примечание. Используемое пространство должно быть пропорционально количеству вершин плюс количество ребер.

Неявное создание вершин. Когда объект используется как аргумент метода `addEdge()`, мы подразумеваем, что это имя (строковое) вершины. Если ни одно из ребер еще не использует добавляемое имя, наша реализация создает вершину с этим именем. Альтернативный метод `addVertex()` требует большего количества клиентского кода (для создания вершины) и более громоздкого кода реализации (для проверки, что ребра соединяют ранее созданную вершину).

Петли и параллельные ребра. Хотя API не решает эту проблему явно, мы подразумеваем, что реализация *допускает* петли (ребра, соединяющие вершину саму с собой), но *не допускает* параллельные ребра (две копии того же ребра). Проверка на петли и параллельные ребра проста, поэтому не будем рассматривать их.

Методы клиентских запросов. Мы включаем также в наш API методы `countV()` и `countE()`, сообщающие клиенту количество вершин и ребер в графе. Точно так же методы `degree()`, `hasVertex()` и `hasEdge()` полезны в клиентском коде. Это все однострочные реализации постоянного времени.

Ни одно из этих проектных решений не канонично; они выбраны лишь для кода в этой книге. В различных ситуациях могли бы подойти другие варианты, а некоторые решения все еще требуют реализации. Следует обдуманно делать свой выбор проектного решения *и быть готовым защищать его.*



Представление графа таблицы идентификаторов наборов

Этот API реализует программа 4.5.1 (`graph.py`). Его внутреннее представление — *таблица идентификаторов наборов* (symbol table of sets): ключи — вершины; значения — наборы соседей (вершины, соседние с ключом). Небольшой пример приведен ниже. Для реализации этого представления мы используем два встроенных типа данных, `dict` и `set`, описанных в разделе 4.4. Этот выбор приводит к трем важным свойствам.

- Клиенты могут эффективно перебирать граф по вершинам.
- Клиенты могут эффективно перебирать вершины по соседям.
- Используемое пространство пропорционально количеству вершин плюс количество ребер.

Эти свойства следуют из базовых свойств типов `dict` и `set`. Как вы увидите, в основе обработки графа лежат эти два итератора.

Вполне естественный способ вывода типа `Graph` подразумевает помещение каждой вершины в отдельную строку в сопровождении списка ее непосредственных соседей. Соответственно, мы поддерживаем встроенную функцию `str()`, реализуя метод `__str__()` следующим образом:

```
def __str__(self):
    s = ''
    for v in self.vertices():
        s += v + ' '
        for w in self.adjacentTo(v):
            s += w + ' '
        s += '\n'
    return s
```

Программа 4.5.1. Тип данных графа (graph.py)

```
import sys
import stdio
from instream import InStream

class Graph:
    # См. текст для __str__() и __init__()

    def addEdge(self, v, w):
        if not self.hasVertex(v): self._adj[v]=set()
        if not self.hasVertex(w): self._adj[w]=set()
        if not self.hasEdge(v, w):
            self._e += 1
            self._adj[v].add(w)
            self._adj[w].add(v)

    def adjacentTo(self, v): return iter(self._adj[v])
    def vertices(self):      return iter(self._adj)
    def hasVertex(self, v):  return v in self._adj
    def hasEdge(self, v, w): return w in self._adj[v]
    def countV(self):        return len(self._adj)
    def countE(self):        return self._e
    def degree(self, v):     return len(self._adj[v])

def main():
    file=sys.argv[1]
    graph=Graph(file)
    stdio.writeln(graph)

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

Переменные экземпляра

_e	Количество ребер
_adj	Списки соседей

Эта реализация типа данных графа использует встроенные типы `dict` и `set` (см. раздел 4.4). Клиенты могут создавать графы, добавляя ребра по одному или читая их из файла; они могут обработать графы, перебирая все наборы вершин или набор вершин, соседних с данной вершиной. Клиент проверки строит граф из файла, указанного в командной строке.

```
% more tinygraph.txt
A B
A C
C G
A G
H A
B C
B H
```

```
% python graph.py
tinygraph.txt
A  B C G H
B  A C H
C  A B G
G  A C
H  A B
```

Полученная строка включает два представления каждого ребра: один для случая, когда обнаруживается, что w — сосед v, и один для случая, когда обнаруживается, что v — сосед w. На этой простой парадигме обработки каждого ребра в графе (дважды) основано множество алгоритмов графов. Эта реализация предназначена только для малых графов, поскольку в некоторых системах продолжительность имеет квадратичную зависимость от длины строк (см. упр. 4.5.3).

Выходной формат функции str() определяет также формат входного файла. Метод __init__() обеспечивает создание графа из файла в этом формате (каждая строка — имя вершины, сопровождаемое именами ее соседей, разделенными пробелами). Для гибкости мы позволяем использовать и иные разделители, кроме пробелов (чтобы, например, имена вершин могли содержать пробелы):

```
def __init__(self, filename=None, delimiter=None):
    self._e = 0
    self._adj = dict()
    if filename is not None:
        instream = InStream(filename)
        while instream.hasNextLine():
            line = instream.readLine()
            names = line.split(delimiter)
            for i in range(1, len(names)):
                self.addEdge(names[0], names[i])
```

Обратите внимание: конструктор (со стандартным разделителем в виде пробела) работает правильно, даже когда на вводе список ребер по одному в строке, как в клиенте проверки для программы 4.5.1. Кроме того, конструктор со стандартным именем файла и разделителем создает пустой граф. Добавление методов __init__() и __str__() в тип Graph обеспечивает завершенный тип данных, подходящий для широкого разнообразия приложений, как будет продемонстрировано далее.

Пример клиента графа. В качестве первого обрабатывающего граф клиента рассмотрим хорошо всем знакомый пример социальных отношений, для которого доступны обширные данные.

На сайте книги вы можете найти файл movies.txt (и много подобных файлов), содержащий список фильмов и участвующих в них исполнителей. В каждой строке приведено название фильма, сопровождаемое списком имен исполнителей. Поскольку в именах есть пробелы и запятые, как разделитель будем использовать символ '/'. Теперь вы можете убедиться, почему мы позволили клиентам графа определить разделитель.

Если вы изучите файл movies.txt, то обратите внимание на множество характеристик, заслуживающих некоторого внимания при работе с базой данных.

- Список исполнителей выводится не в алфавитном порядке.
- Названия фильмов и имена исполнителей — строки Unicode.

- После названия фильма в круглых скобках указан год.
- Совпадающие имена исполнителей различаются римскими цифрами в круглых скобках.

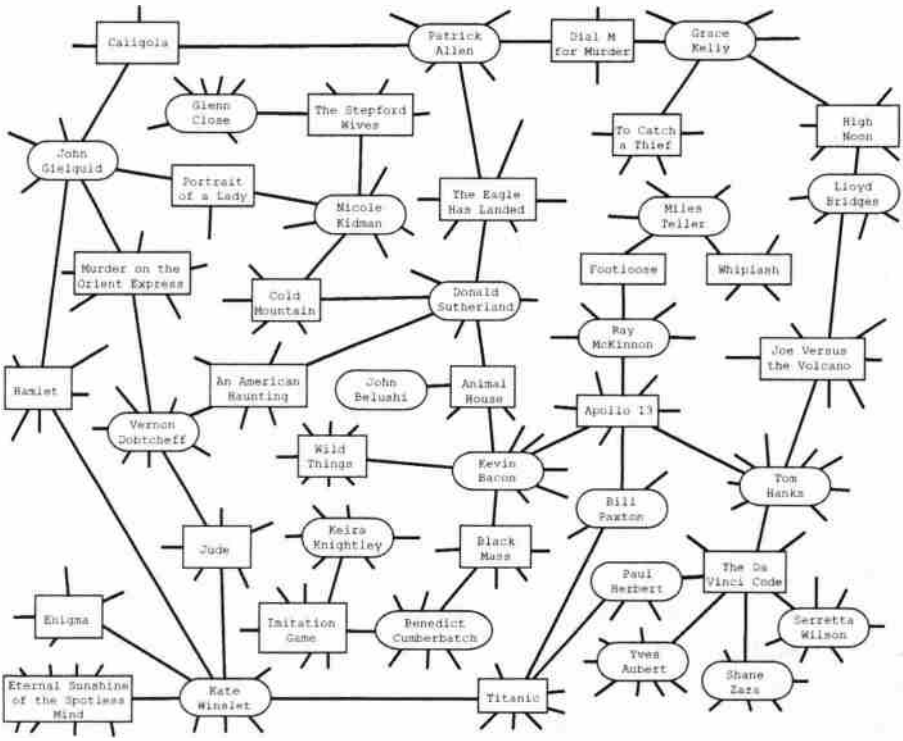
В зависимости от параметров вашего терминала и операционной системы, специальные символы могут быть заменены пробелами или вопросительными знаками. Подобные аномалии характерны для работы с большими объемами реальных данных. Если у вас возникли подобные проблемы, проконсультируйтесь на сайте книги, как настроить свою систему для правильной работы с символами Unicode.

```
% more movies.txt
...
Tin Men (1987)/DeBoy, David/Blumenfeld, Alan/... /Geppi, Cindy/Hershey, Barbara
Tirez sur le pianiste (1960)/Heymann, Claude/.../Berger, Nicole (I)
Titanic (1997)/Mazin, Stan/...DiCaprio, Leonardo/.../Winslet, Kate/...
Titus (1999)/Weisskopf, Hermann/Rhys, Matthew/.../McEwan, Geraldine
To Be or Not to Be (1942)/Verebes, Ernц (I)/.../Lombard, Carole (I)
To Be or Not to Be (1983)/.../Brooks, Mel (I)/.../Bancroft, Anne/...
To Catch a Thief (1955)/Parнs, Manuel/.../Grant, Cary/.../Kelly, Grace/...
To Die For (1995)/Smith, Kurtwood/.../Kidman, Nicole/.../ Tucci, Maria
...
```

Пример базы данных фильмов

Используя тип `Graph`, мы можем составить простой и удобный клиент для извлечения информации из файла `movies.txt`. Начнем с построения графа для лучшего структурирования информации. Что должно быть вершинами и ребрами модели? Вершинами должны быть фильмы, а ребра должны соединять два фильма, если исполнитель принимал участие в обоих? Вершинами должны быть исполнители, а ребра должны соединять двух исполнителей, если оба были заняты в том же фильме? Вероятны оба варианта, но какой из них использовать? Это решение затрагивает и код клиента, и реализации. Другой способ (который мы и выбираем, поскольку он приводит к простому коду реализации) подразумевает наличие вершин и для фильмов, и для исполнителей, с ребром, соединяющим каждый фильм с каждым его исполнителем. Как вы увидите, обрабатывающие этот граф программы могут ответить на великое множество интересных вопросов.

Программа 4.5.2 (`invert.py`) является первым примером. Это клиент типа `Graph`, получающий запрос, такой как имя фильма, и выводящий список его исполнителей.



Небольшая часть графа отношений между исполнителями и фильмами

Ввод имени фильма и получение его исполнителей не влечет за собой ничего, кроме вывода соответствующих строк файла `movies.txt`. (Обратите внимание, что список исполнителей выводится в произвольном порядке, поскольку для представления каждого списка соседних вершин мы используем тип `set`.) Куда интересней возможность программы `invert.py`: получив имя *исполнителя*, вывести список *фильмов*, в которых он участвовал. Почему это работает? Даже при том, что база данных, казалось бы, соединяет фильмы с исполнителями, а не наоборот, ребра в графе — это *связи*, соединяющие также исполнителей с фильмами.

В *двудольном графе* (bipartite graph) вершины одного вида соединяются с вершинами другого. Как иллюстрирует этот пример, у двудольных графов есть много естественных свойств, которые мы можем использовать интересными способами.

```
% more amino.csv
TTT,Phe,F,Phenylalanine
TTC,Phe,F,Phenylalanine
TTA,Leu,L,Leucine
TTG,Leu,L,Leucine
TCT,Ser,S,Serine
TCC,Ser,S,Serine
TCA,Ser,S,Serine
TCG,Ser,S,Serine
TAT,Tyr,Y,Tyrosine
...
GGA,Gly,G,Glycine
GGG,Gly,G,Glycine
% python invert.py amino.csv ","
TTA
  Lue
  L
  Leucine
Serine
  TCT
  TCC
  TCA
  TCG
```

Инверсия индекса

Программа 4.5.2. Использование графа для инверсии индекса (`invert.py`)

```
import sys
import stdio
from graph import Graph

file = sys.argv[1]
delimiter = sys.argv[2]
graph = Graph(file, delimiter)

while stdio.hasNextLine():
    v = stdio.readLine()
    if graph.hasVertex(v):
        for w in graph.adjacentTo(v):
            stdio.writeln('  ' + w)
```

`file`	Входной файл
`delimiter`	Разделитель вершин
`graph`	Граф
`v`	Запрос
`w`	Сосед `v`

Этот клиент типа `Graph` получает в аргументах командной строки имя файла графа и разделитель. Он строит граф из файла, а затем последовательно читает имена вершин со стандартного ввода и выводит соседей этой вершины. Когда файл представляет собой индекс исполнителей фильма (такой, как `movies.txt`), он создает двудольный граф и составляет интерактивный инвертированный индекс.

```
% python invert.py tinygraph.txt " "
C
 A
 B
 G
A
 B
 C
 G
 H
```

```
% python invert.py movies.txt "/"
Da Vinci Code, The (2006)
  Fosh, Christopher
  Sciarappa, Fausto Maria
  Zaza, Shane
  L'Abidine, Dhaffer
  Bettany, Paul
  ...
Bacon, Kevin
  Murder in the First (1995)
  JFK (1991)
  Novocaine (2001)
  In the Cut (2003)
  Where the Truth Lies (2005)
  ...
```

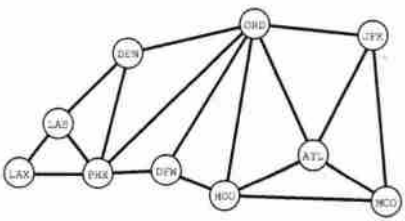
Как упоминалось в начале раздела 4.4, парадигма индексации является очень общей и знакомой. Стоит поразмыслить над фактом, что построение двудольного графа обеспечивает простой путь автоматического инвертирования *любого* индекса! База данных `movies.txt` индексируется по фильмам, но мы можем сделать запрос и по исполнителям. Мы могли бы использовать программу `invert.py`

точно таким же способом для вывода индекса слов, имеющихся на заданной странице, или кодонов, соответствующих заданной аминокислоте (см. пример ниже), или инвертировать любой из других индексов, обсуждаемых в начале раздела 4.2. Поскольку программа `invert.py` получает как аргумент командной строки разделитель, можно использовать это для создания интерактивного инвертированного индекса для файла `.csv` или файла с разделителями в виде пробелов.

Эти функциональные возможности инвертированного индекса — непосредственное преимущество *структуры данных* графа. Далее мы исследуем часть дополнительных преимуществ, предоставляемых *алгоритмами*, обрабатывающими структуру данных.

Кратчайшие пути в графах. Если даны две вершины в графе, то *путь* (path) — это последовательность соединенных ребрами вершин между ними. *Кратчайший путь* (shortest path) имеет минимальное количество ребер (возможно несколько кратчайших путей). Поиск кратчайшего пути, соединяющего две вершины в графе, является фундаментальной проблемой информатики. Кратчайшие пути успешно применяются для решения крупномасштабных проблем в широком разнообразии случаев: от маршрутизации Интернета и арбитража финансовых транзакций до изучения динамики нейронов мозга.

Начало	Конец	Дистанция	Кратчайший путь
JFK	LAX	3	JFK-ORD-PHX-LAX
LAS	MCO	4	LAS-PHX-DFW-HOU-MCO
HOU	JFK	2	HOU-ATL-JFK



Примеры кратчайших путей в графе

В качестве примера предположим, что вы клиент вымышленной авиалинии, обслуживающей весьма ограниченное количество городов с ограниченным количеством направлений. Считается, что наилучший способ добраться из одного места в другое — это минимизировать количество перелетов, поскольку время пересадки между рейсами, вероятно, окажется весьма продолжительным (или, возможно, имеет смысл заплатить больше за прямой рейс на другой авиалинии!). Алгоритм кратчайшего пути — это именно то, что нужно при планировании поездки. Такой случай интуитивно понятен и позволяет рассмотреть фундаментальную проблему и наш подход к ее решению. Затронув эти темы в контексте данного примера, мы рассмотрим приложение, где модель графа более абстрактна.

В зависимости от конкретного случая, у клиентов будут разные требования к кратчайшим путям. Мы хотим получить кратчайший путь, соединяющий две заданные вершины? Нас интересует только длина такого пути? Будет ли таких запросов много? Или представляет интерес некая конкретная вершина? Наш выбор стоит начать со следующего API.

API для типа данных `PathFinder`

Операция	Описание
`PathFinder(graph, s)`	Найти все кратчайшие пути от `s` в `graph`
`Pf.distanceTo(v)`	Дистанция между `s` и `v`
`Pf.hasPathTo(v)`	Есть ли путь между `s` и `v`?
`Pf.pathTo(v)`	Итерируемое для пути от `s` до `v`

В огромных графах или при огромном количестве запросов следует уделить особое внимание разработке API, поскольку стоимость вычисления путей может оказаться лимитирующим фактором. При таком проекте для заданного графа и заданной вершины клиенты могут создать объект `PathFinder`, а затем использовать его либо для поиска длины кратчайшего пути, либо для перебора вершин на кратчайшем пути к любой другой вершине в графе. Реализация этих методов известна как *алгоритм поиска кратчайшего пути от одной вершины до всех остальных* (single-source shortest-path algorithm). Классический алгоритм, известный как *поиск в ширину* (breadth-first search), обеспечивает прямое и изящное решение, где конструктор занимает линейное время; метод `distanceTo()` — постоянное время, а метод `pathTo()` — время, пропорциональное длине пути. Прежде чем исследовать нашу реализацию, рассмотрим несколько клиентов.

Клиент одной вершины. Предположим, вы имеете граф вершин и связей для карты маршрутов нашей вымышленной авиакомпании. Используя свой родной город как отправной пункт, составьте клиент, выводящий ваш маршрут перед каждым путешествием. Программа 4.5.3 (`separation.py`) является клиентом типа `PathFinder`, обеспечивающим эти функциональные возможности для любого графа. Клиент этого типа особенно полезен в приложениях, ожидающих многочисленных запросов из той же отправной точки. В этой ситуации стоимость построения `PathFinder` амортизируется стоимостями всех запросов. Вы можете исследовать свойства кратчайших путей, применив `PathFinder` к нашему файлу ввода `routes.txt` или любой входной модели по вашему выбору. Фактически очень многие довольно часто используют подобный алгоритм в приложении карт на своем телефоне.

Программа 4.5.3. Клиент кратчайших путей (separation.py)

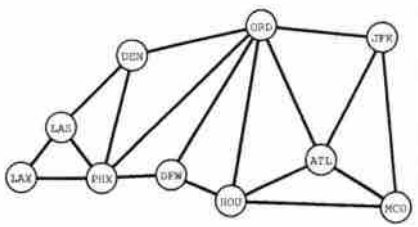
```
import sys
import stdio
from graph import Graph
from pathfinder import PathFinder

file = sys.argv[1]
delimiter = sys.argv[2]
graph = Graph(file, delimiter)

s = sys.argv[3]
pf = PathFinder(graph, s)

while stdio.hasNextLine():
    t = stdio.readLine()
    if pf.hasPathTo(t):
        distance = pf.distanceTo(t)
        for v in pf.pathTo(t):
            stdio.writeln('   ' + v)
        stdio.writeln('distance: ' + str(distance))
```

file	Имя файла графа
delimiter	Разделитель имен вершин
graph	Граф
pf	PathFinder из s
s	Исходная вершина
t	Конечная вершина
v	Вершина на пути от s до t

Этот клиент типа PathFinder получает как аргументы командной строки имя файла, разделитель и исходную вершину. Он строит граф из файла, подразумевая, что каждая его строка определяет вершину и список его соседних вершин, отделенных разделителем. Когда вы вводите вершину назначения t, вы получаете кратчайший путь от исходного пункта к пункту назначения.

```
% more routes.txt
JFK MCO
ORD DEN
ORD HOU
DFW PHX
JFK ATL
ORD DFW
ORD PHX
ATL HOU
DEN PHX
...

Hello, World
```

```
% python separation.py routes.txt " " JFK
LAX
   JFK
   ORD
   PHX
   LAX
distance 3
DFW
   JFK
   ORD
   DFW
distance 2
```



Степени разделения. Одно из классических приложений алгоритмов кратчайших путей — это поиск *степени разделения* (degree of separation) между личностями в социальной сети. Для закрепления идеи обсудим этот случай в терминах популярного не так давно времяпрепровождения, известного как *игра Кевина*

Бэйкона (Kevin Bacon game), использующая только что рассмотренный граф исполнителей и фильмов. Кевин Бэйкон — популярный актер, снявшийся во многих фильмах. Мы присваиваем каждому снявшемуся в фильме исполнителю *число Бэйкона* (Kevin Bacon number): сам Бэйкон имеет число 0, любой другой исполнитель в том же списке, что и Бэйкон, имеет число Бэйкона 1, а любого другой исполнитель (кроме Бэйкона) в том же списке, что и исполнитель с числом 1, имеет число Бэйкона 2 и т.д. Например, у Мерил Стрип число Бэйкона 1, поскольку она снялась в фильме *Дикая река* с Кевином Бэйконом. Число Николь Кидман 2: хотя она и не снималась ни в одном фильме с Кевином Бэйконом, она участвовала в фильме *Холодная гора* с Дональдом Сазерлендом, он снимался в фильме *Зверинец* с Кевином Бэйконом. Когда дано имя исполнителя, самая простая версия игры — это поиск некой последовательности фильмов и исполнителей, ведущей к Кевину Бэйкону. Например, Том Хэнкс снялся в *Джо против вулкана* с Ллойдом Бриджес, который снимался в *Жарком полдне* с Грейс Келли[2], которая была в фильме *В случае убийства набирайте «М»* с Патриком Алленом, который снялся в фильме *Орел приземлился* с Дональдом Сазерлендом, который, как известно, был в *Зверинце* с Кевином Бэйконом. Но этого знания недостаточно, чтобы установить число Бэйкона для Тома Хэнкса (это фактически 1, поскольку он снялся с Кевином Бэйконом в *Аполлоне 13*). Как можно заметить, число Бэйкона определяется при подсчете фильмов в *самой короткой* последовательности, поэтому трудно убедиться, выигрывает ли некто игру, не используя компьютер. Программа 4.5.3 (`separation.py`) как раз и необходима для поиска кратчайшего пути, устанавливающего число Бэйкона для любого исполнителя в файле `movies.txt`: это число — точно половина дистанции. Вы могли бы использовать эту программу или дополнить ее, чтобы отвечать на некоторые интересные вопросы о киноиндустрии или о многих других областях. Например, математики играют в ту же игру с графом о соавторстве публикаций с *Полом Эрдёшем* (Paul Erdös), известным математиком XX века. Точно так же у всех жителей Нью-Джерси, есть число Брюса Спрингстина 2, поскольку все в штате, кажется, знают кого-то, кто утверждает, что знал Брюса.

Другие клиенты. `PathFinder` — это универсальный тип данных, применимый во множестве практических случаев. Например, довольно просто разработать клиента, обрабатывающего парные запросы "откуда–куда" со стандартного ввода, при построении объекта `PathFinder` для каждой вершины (см. упр. 4.5.11). Бюро путешествий используют именно этот подход для обработки запросов с очень высокой скоростью. Поскольку клиент создает объект `PathFinder` для каждой вершины (каждый из которых способен занять пространство, пропорциональное количеству вершин), его применение для огромных разреженных графов может стать лимитирующим фактором. В качестве примера приложений еще более требовательных к производительности, концептуально такого же, рассмотрим маршрутизатор

[2] 10-я княгиня Монако, мать ныне правящего князя Альбера II. — *Примеч. ред.*

Интернета, имеющий граф соединений между доступными машинами и вынужденный принимать решения о наилучшем следующем транзитном участке при передаче пакетов заданному получателю. Для этого можно построить PathFinder с собой как с отправителем, а затем посылать пакет на первую вершину в pf.pathTo(w), т.е. на следующий транзитный узел в кратчайшем пути к w. Либо центральный сервер мог бы построить по одному объекту PathFinder для каждого из периферийных маршрутизаторов и использовать их для формирования инструкций маршрутизации. Способность обработать такие запросы с высокой скоростью является одной из главных обязанностей маршрутизаторов Интернета, а алгоритмы поиска кратчайших путей — критически важная часть этого процесса.

Дистанция кратчайшего пути. Мы определяем *дистанцию* (distance) между двумя вершинами как длину кратчайшего пути между ними. Первый этап в понимании поиска в ширину подразумевает рассмотрение проблемы вычисления дистанций между исходным пунктом и каждой вершиной (реализация distanceTo() в PathFinder). Наш подход подразумевает вычисление и сохранение всех дистанций в конструкторе и последующее их возвращение при вызове клиентом метода distanceTo(). Для ассоциации целочисленной дистанции с каждым именем вершины мы используем таблицу идентификаторов:

```
_distTo = dict()
```

Задача этой таблицы идентификаторов заключается в ассоциации с каждой вершиной длины кратчайшего пути (дистанция) между той вершиной и s. Начнем с присвоения s дистанции 0 оператором _distTo[s] = 0, а соседям s присваиваем дистанцию 1 следующим кодом:

```
for v in g.adjacentTo(s)
    self._distTo[v] = 1
```

Но что делать потом? Если бездумно установить дистанции для всех соседей в 2, то перед нами не только возникнет перспектива ненужной установки многих значений дважды (у соседей может быть много общих соседей), но мы установили бы также дистанцию 2 для s (ведь это сосед каждого из его соседей), а такой результат нам явно не нужен. Разрешить эти затруднения весьма просто.

- Рассматривать вершины в порядке их дистанции от s.
- Игнорировать вершину, если ее дистанция к s уже известна.

Для организации вычислений мы используем порядок FIFO. Начиная с s в очереди, мы осуществляем следующие операции, пока очередь не пуста.

- Извлекаем из очереди вершину v.
- Присваиваем всем соседям v с неизвестной дистанцией значение на 1 больше, чем дистанция v.
- Ставим в очередь всех неизвестных соседей.

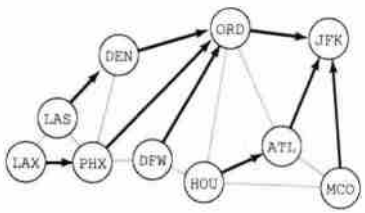
Согласно этому методу, вершина извлекается из очереди в неубывающем порядке их дистанции от исходного пункта s. Трассировка этого метода на типичном графе поможет убедиться в его правильности, продемонстрировав, что метод маркирует каждую вершину v ее дистанцией к s. Его математическая индукция — упражнение 4.5.13.

Дерево кратчайших путей. Необходима не только дистанция от исходного пункта, но также и длина пути. Для реализации метода `pathTo()` мы используем подграф, известный как *дерево кратчайших путей* (shortest-paths tree), определенное следующим образом.

- Исходная вершина s помещается в корень дерева.
- Сосед вершины v помещается в дерево, если он добавляются в очередь с ребром, соединяющим его с v.

Поскольку мы помещаем каждую вершину в очередь только однажды, эта структура — настоящее дерево: оно состоит из корня (исходный пункт), соединенного с одним поддеревом для каждого соседа. Изучая такое дерево, вы сразу заметите, что дистанция от каждой вершины до корня в дереве та же, что и дистанция кратчайшего пути к исходному пункту в графе. Однако важней всего то, что каждый путь в дереве — кратчайший путь в графе. Это наблюдение важно, поскольку оно предоставляет нам простой способ непосредственно обеспечить клиентов кратчайшими путями (реализация `pathTo()` в `PathFinder`).

Граф



Дерево кратчайших путей

Представление ссылок на родительские узлы

w	ATL	DEN	DFW	HOU	JFK	LAS	LAX	MCO	ORD	PHX
prev[w]	JFK	ORF	ORD	ATL		DEN	PHX	JFK	JFK	ORD

Дерево кратчайших путей

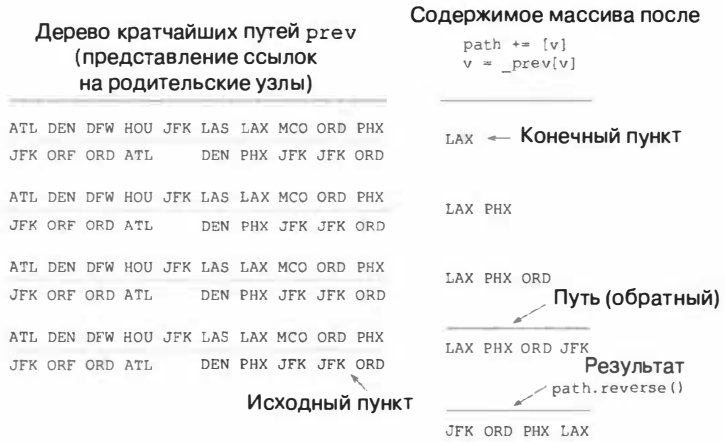
Сначала мы создаем таблицу идентификаторов, ассоциирующую каждую вершину с вершиной на один шаг ближе к исходной точке на кратчайшем пути:

```
_edgeTo = dict()
```

С каждой вершиной w мы хотим ассоциировать предыдущий пункт на кратчайшем пути от исходного пункта к w. Усовершенствовать метод кратчайших путей для вычисления этой информации также просто: когда мы помещаем w в очередь, впервые обнаружив его как сосед v, мы делаем именно это, поскольку v — это предыдущий пункт на кратчайшем пути от исходного пункта к w, мы можем присвоить `_edgeTo[w] = v`. Структура данных `_prev` — это не более чем представление дерева кратчайших путей: оно представляет ссылку от каждого узла на его родительский узел в дереве. Далее,

чтобы ответить на клиентский запрос о пути от исходного пункта к v (вызов pathTo(v) в PathFinder), мы следуем по этим ссылкам вверх по дереву до v, проходя путь в обратном порядке. Мы собираем вершины в массив по мере их прибытия, а затем изменяем массив на обратный. Таким образом, клиент получает путь от s до v, используя итератор, возвращенный методом pathTo().

Поиск в ширину. Программа 4.5.4 (pathfinder.py) является реализацией API поиска кратчайшего пути от одной вершины во все остальные на основании только что обсуждавшихся идей. Она поддерживает две таблицы идентификаторов. Одна хранит дистанцию между исходной вершиной и каждой другой вершиной; другая хранит предыдущий пункт на кратчайшем пути от исходной вершины до каждой другой. Конструктор использует очередь для отслеживания встретившихся вершин (соседей вершин, для которых были найдены кратчайшие пути, но чьи соседи еще не были исследованы). Этот процесс известен как *поиск в ширину* (Breadth-First Search — BFS), поскольку он осуществляет поиск в графе в ширину.

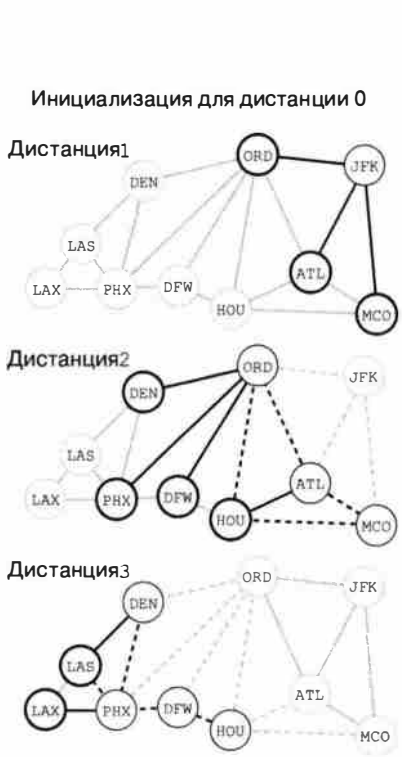


Восстановление пути из дерева

Другой важный метод поиска в графе, в отличие от него, известен как *поиск вглубь* (depth-first search). Он основан на рекурсивном методе, подобном использованному нами для задачи просачивания в программе 2.4.6. Поиск вглубь имеет смысл при поиске длинных путей, а поиск в ширину гарантирует нахождение кратчайших путей.

Производительность. Стоимость алгоритмов обработки графа обычно зависит от двух параметров графа: количества вершин V и количества ребер E. Для простоты мы подразумеваем, что есть путь от исходной вершины к любой другой вершине. Далее, как реализовано в PathFinder, *время поиска в ширину линейно зависит от размера ввода* (пропорциональный $E + V$) при подходящем

техническом предположении о хеш-функции. Чтобы убедиться в этом факте, обратите сначала внимание на то, что внешний цикл (`while`) осуществляет итерации ровно V раз, по одному для каждой вершины, поскольку мы позаботились о гарантии, что вершины не помещается в очередь больше одного раза. Затем обратите внимание, что внутренний цикл (`for`) осуществляет итерации 2E раз по всем итерациям, поскольку он исследует каждое ребро именно дважды: по одному разу для каждой из двух соединяемых вершин. Тело внутреннего цикла (`for`) требует по крайней мере одной операции *поиска* (чтобы определить, встречалась ли вершина ранее) и, возможно, еще одной операции *получения* и двух операций *помещения* (чтобы обновить информацию о дистанции и пути) для таблиц идентификаторов размером максимум V. Таким образом, общая продолжительность пропорциональна $E + V$ (при подходящем техническом предположении о хеш-функции). Если бы мы должны были использовать бинарные деревья поиска вместо хеш-таблиц, то общая продолжительность была бы линейно-логарифмической — пропорциональной $E \log V$.



v	Содержимое очереди	Дистанции от JFK									
		ATL	DEN	DFW	HOU	JFK	LAS	LAX	MCO	ORD	PHX
	JFK					0					
JFK						0					
JFK	ATL	1				0					
JFK	ATL MCO	1				0			1		
JFK	ATL MCO ORD	1				0			1	1	
ATL	MCO ORD	1				0			1	1	
ATL	MCO ORD HOU	1			2	0			1	1	
MCO	ORD HOU	1			2	0			1	1	
ORD	HOU	1			2	0			1	1	
ORD	HOU DEN	1	2		2	0			1	1	
ORD	HOU DEN DFW	1	2	2	2	0			1	1	
ORD	HOU DEN DFW PHX	1	2	2	2	0			1	1	2
HOU	DEN DFW PHX	1	2	2	2	0			1	1	2
DEN	DFW PHX	1	2	2	2	0			1	1	2
DEN	DFW PHX LAS	1	2	2	2	0	3		1	1	2
DFW	PHX LAS	1	2	2	2	0	3		1	1	2
PHX	LAS	1	2	2	2	0	3		1	1	2
PHX	LAS LAX	1	2	2	2	0	3	3	1	1	2
LAS	LAX	1	2	2	2	0	3	3	1	1	2
LAX		1	2	2	2	0	3	3	1	1	2

Использование поиска в ширину для вычисления кратчайшего пути в графе

Программа 4.5.4. Реализация кратчайших путей (pathfinder.py)

```
import stdio
import graph
from linkedqueue import Queue
class PathFinder:
    def __init__(self, graph, s):
        self._distTo = dict()
        self._edgeTo = dict()

        queue = Queue()
        queue.enqueue(s)
        self._distTo[s] = 0
        self._edgeTo[s] = None
        while not queue.isEmpty():
            v = queue.dequeue()
            for w in graph.adjacentTo(v):
                if w not in self._distTo:
                    queue.enqueue(w)
                    self._distTo[w] = 1 + self._distTo[v]
                    self._edgeTo[w] = v

    def distanceTo(self, v):
        return self._distTo[v]

    def hasPathTo(self, v):
        return v in self._distTo

    def pathTo(self, v):
        path = []
        while v is not None:
            path += [v]
            v = self._edgeTo[v]
        return reversed(path)
```

Переменные экземпляра

`_distTo`	Дистанция до `s`
`_prevTo`	Предыдущая вершина на кратчайшем пути от `s`

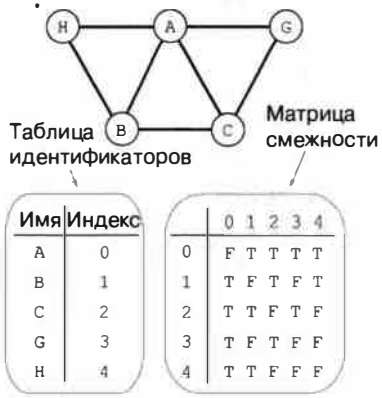
`graph`	Граф
`s`	Исходный пункт
`queue`	Очередь посещаемых вершин
`v`	Текущая вершина
`w`	Соседи `v`

Этот класс обеспечивает клиентам поиск (самых коротких) путей, соединяющих определенную вершину с другой вершиной в графе. Типичные клиенты см. в программе 4.5.3 и упражнении 4.5.17.

Представление матрицы смежности. Без надлежащих структур данных высокая производительность алгоритмов обработки графов иногда труднодостижима и не сама собой разумеется. Например, другое весьма часто реализуемое программистами представление графа — это *представление матрицы смежности* (adjacency-matrix representation), использующее таблицу идентификаторов для сопоставления имен вершин с целыми числам от 0 до $V-1$ и массив размером V на V логических

переменных со значением `True` в элементе ряда `i` и столбца `j` (и симметрично ряда `j` и столбца `i`), если есть ребро, соединяющее вершину, соответствующую `i`, с вершиной, соответствующей `j`, и значение `False`, если такого ребра нет. В этой книге подобные представления уже использовались в разделе 1.6 при изучении модели случайной навигации для ранжирования веб-страниц. Представление матрицы смежности просто, но неосуществимо для огромных графов — граф с миллионом вершин потребовал бы матрицы смежности с *триллионом* элементов. Понимание этого различия для задач обработки графов означает различие между решением задачи, возникшей в практической ситуации, и неспособностью решить ее вообще.

Поиск в ширину — фундаментальный алгоритм, используемый в приложениях на вашем мобильном устройстве для поиска маршрута на карте города, в метро, авиарейсов или в множестве подобных ситуаций. Как свидетельствует наш пример со степенями разделения, он применим также в других бесчисленных случаях: от просмотра веб и маршрутизации пакетов Интернета до изучения инфекционных болезней, моделей мозга и отношений между последовательностями генов. Большинство этих приложений задействует огромные графы, поэтому эффективный алгоритм абсолютно необходим.



Представление графа матрицы смежности

Обобщенная модель кратчайших путей должна ассоциировать положительный вес (способный представлять дистанцию или время) с каждым ребром и искать путь с минимальной суммой весов ребер. На старших курсах по алгоритмам и исследованию операций вы изучите обобщенный поиск в ширину, известный как *алгоритм Дейкстры* (Dijkstra's algorithm), решающий эту задачу за линейно-логарифмическое время. Когда вы получаете направление от устройства GPS или приложения карты на своем телефоне, в основе решения задачи нахождения кратчайшего пути лежит алгоритм Дейкстры. Эти важные и вездесущие приложения — лишь верхушка айсберга, поскольку модели графов — это намного больше, чем карты.

Графы "тесного мира". Ученые выявили особенно интересный класс графов, присутствующих во множестве случаев естественных и общественных наук. Графы "тесного мира" характеризуются тремя свойствами.

- Они *разрежены* (sparse), т.е. количество вершин намного меньше, чем количество ребер.
- У них есть *короткие средние длины путей* (short average path length). Если вы выбираете две случайные вершины, то длина кратчайшего пути между ними будет короткой.

- Они демонстрируют *локальную кластеризацию* (local clustering). Если две вершины являются соседями третьей, то эти две вершины, вероятно, также будут соседями.

Графы, обладающие всеми этими тремя свойствами, называют *феноменом "тесного мира"*. Термин *"тесный мир"* отражает идею, что у подавляющего большинства вершин есть и локальная кластеризация, и короткие пути к другой вершине, а термин *феномен* отражает тот неожиданный факт, что очень много встречающихся на практике графов разрежены, демонстрируют локальную кластеризацию и имеют короткие пути. Кроме только что рассмотренного случая социальных отношений, графы "тесного мира" используются для изучения рынка товаров и идей, формирования и распространения моды, анализа Интернета, создания защищенных одноранговых сетей, разработки алгоритмов маршрутизации и беспроводных сетей, проектирования силовых электрических сетей, моделирования обработки информации человеческим мозгом, исследования фазовых переходов в осцилляторах, распространения инфекционных вирусов (и в живых организмах, и в компьютерах), а также для многих других случаев. После первой публикации Дункана Ваттса и Стивена Строгаца в 1990-х годах был проведен огромный объем исследований по теме феномена "тесного мира".

Ключевой вопрос в таком исследовании следующий: *дан граф, как установить, что это граф "тесного мира"?* Чтобы ответить на этот вопрос, начнем с предположения, что граф не является малым (скажем, 1 000 вершин или более) и что он соединен (т.е. существует *некий* путь, соединяющий каждую пару вершин). Затем необходимо учесть определенные пороговые значения для каждого из свойств "тесного мира".

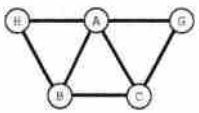
- Под *разреженным* мы подразумеваем граф со средней степенью вершин, меньшей $20 \lg V$.
- Под *короткой средней длиной пути* мы подразумеваем, что средняя длина кратчайшего пути между двумя вершинами меньше $10 \lg V$.
- Под *локальной кластеризацией* мы подразумеваем, что значение *коэффициента кластеризации* (clustering coefficient) больше 10%.

Определение локальной кластеризации немного сложней, чем определение разреженности и средней длины пути. Интуитивно коэффициент кластеризации вершины представляет вероятность, что два выбранных наугад соседа будут соединены ребром. Более точно, если у вершины есть t соседей, то есть $t(t-1)/2$ возможных ребер, соединяющих их; ее *локальный коэффициент кластеризации* — это доля находящихся в графе ребер (или 0, если степень вершины 0 или 1). *Коэффициент кластеризации графа* — это среднее локальных коэффициентов кластеризации его вершин. Если в среднем он более 10%, то мы говорим, что граф локально кластерный. На схеме, приведенной ниже, эти три значения вычисляются для крошечного графа.

Средняя степень вершин

Вершина	Степень
A	4
B	3
C	3
G	2
H	2
Всего	14

Средняя степень = 14 / 5 = 2.8



Средняя длина пути

Пара вершин	Кратчайший путь	Длина
A B	A-B	1
A C	A-C	1
A G	A-G	1
A H	A-H	1
B C	B-C	1
B G	B-A-G	2
B H	B-H	1
C G	C-G	1
C H	C-A-H	2
G H	G-A-H	2
	Всего	13

Коэффициент кластеризации

Вершина	Ребра соседей: Возможных	Ребра соседей: Фактических
A	6	3
B	3	2
C	3	2
G	1	1
H	1	1

$$\frac{3/6 + 2/3 + 2/3 + 1/1 + 1/1}{5} \approx .767$$

$$\frac{\text{Общее количество длин}}{\text{Количество пар}} = 13/10 = 1.3$$

Вычисление характеристик графа "тесного мира"

Чтобы лучше ознакомиться с этими определениями, определим несколько простых моделей графа и рассмотрим, подходят ли они под описание графов "тесного мира", проверив их соответствие трем необходимым свойствам.

Полные графы (complete graph). У полного графа с V вершинами есть $V(V-1)/2$ ребер, соединяющих каждую пару вершин. Полные графы — *не графы* "тесного мира". У них короткая средняя длина пути (у каждого кратчайшего пути длина 1), они демонстрируют локальную кластеризацию (коэффициент кластеризации 1), но они *не разрежены* (средняя степень вершины $V-1$, что намного больше, чем $20 \lg V$ для большого V).

Регулярные графы. Регулярный граф — это набор из V равноудаленных по окружности вершин, соединенных с соседями с обеих сторон. В *k-регулярном графе* (k-ring graph) каждая вершина соединена с ее k ближайшими соседями с обеих сторон.

На схеме ниже представлен 2-регулярный граф на 16 вершин. Регулярные графы — также *не* графы "тесного мира". Например, 2-регулярные графы разрежены (у каждой вершины степень 4) и демонстрируют локальную кластеризацию (коэффициент кластеризации 1/2), но их средняя длина пути не короткая (см. упр. 4.5.17).

Случайные графы. Модель Эрдёш-Реньи (Erdös-Rényi model) — хорошо изученная модель создания случайных графов. В этой модели мы создаем *случайный граф* (random graph) на V вершин при наличии каждого возможного ребра с вероятностью p. Случайные графы с достаточным количеством ребер, весьма вероятно, будут связными и будут иметь короткие средние длины пути, но и они *не* графы "тесного мира", поскольку не обеспечивают локальной кластеризации (см. упр. 4.5.43).

Эти примеры иллюстрируют разработку той модели графа, которая удовлетворяет всем трем свойствам одновременно. Уделим минуту попытке разработать

модель графа, который, по нашему мнению, мог бы подойти. Обдумав эту задачу, вы поймете, что для помощи в вычислениях, вероятно, понадобится программа. Кроме того, вы можете согласиться, что просто удивительно, насколько часто они находят практическое применение. Действительно, вы нередко могли бы задаться вопросом: является ли какой-нибудь граф графом "тесного мира"?

Выбор 10% для порогового значения кластеризации вместо некоторого другого фиксированного процента несколько произволен, как и выбор 20 lg V для порогового значения разреженности, и 10 lg V для порогового значения коротких путей, но мы зачастую даже близко не приближаемся к этим граничным значениям. Рассмотрим, например, *граф веб*, в котором есть вершина для каждой веб-страницы и ребра, соединяющие две веб-страницы, если они соединяются ссылкой. Ученые установили количество щелчков, необходимое для того, чтобы добраться с одной веб-страницы на другую, и оказалось, что оно редко превышает 30. Поскольку веб-страниц миллиарды, эта оценка свидетельствует о том, что средняя длина пути между двумя вершинами очень коротка, намного меньше, чем наше пороговое значение 10 lg V (составляющее примерно 300 для 1 миллиарда вершин).

Три модели графа

Модель	Разреженный?	Короткие пути?	Локальная кластеризация?
Полный	○	●	●
2-регулярный	●	○	●
Случайный	●	●	○

Свойства "тесного мира" моделей графа

Несмотря на знание определений, проверка, является ли граф графом "тесного мира", все еще может быть существенной вычислительной сложностью. Как вы, вероятно, и догадывались, рассмотренные типы данных обработки графа предоставляют именно те инструменты, которые необходимы. Программа 4.5.5 (`smallworld.py`) является клиентом типов `Graph` и `PathFinder`, реализующим эти

проверки. Без рассмотренных эффективных структур данных и алгоритмов стоимость этого вычисления была бы чрезмерной. Даже в этом случае для больших графов (таких, как `movies.txt`) нам придется прибегнуть к статистической выборке, чтобы оценить среднюю длину пути и коэффициент кластеризации за разумный период времени (см. упр. 4.5.41), поскольку функции `averagePathLength()` и `clusteringCoefficient()` занимают квадратичное время.

Программа 4.5.5. Проверка "тесного мира" (`smallworld.py`)

```
from pathfinder import PathFinder

def averageDegree(graph):
    return 2.0 * graph.countE() / graph.countV()

def averagePathLength(graph):
    total = 0
    for v in graph.vertices():
        pf = PathFinder(graph, v)
        for w in graph.vertices():
            total += pf.distanceTo(w)
    return 1.0 * total / (graph.countV() * (graph.countV() - 1))

def clusteringCoefficient(graph):
    total = 0
    for v in graph.vertices():
        possible = graph.degree(v) * (graph.degree(v) - 1)
        actual = 0
        for u in graph.adjacentTo(v):
            for w in graph.adjacentTo(v):
                if graph.hasEdge(u, w):
                    actual += 1
        if possible > 0:
            total += 1.0 * actual / possible
    return total / graph.countV()

# Клиент проверки см. в упражнении 4.5.21.
```

Этот клиент типов `Graph` и `PathFinder` вычисляет значения различных параметров графа, чтобы проверить, демонстрирует ли он феномен "тесного мира".

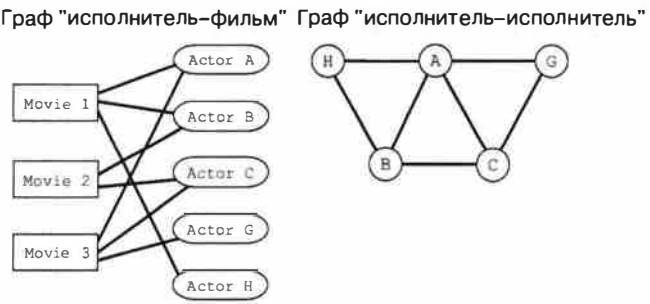
```
% python smallworld.py tinygraph.txt " "
5 vertices, 7 edges
average degree          = 2.800
average path length     = 1.300
clustering coefficient = 0.767
```

Классический граф "тесного мира". Наш граф исполнителей фильмов — не граф "тесного мира", так как он является двусторонним, а следовательно, имеет коэффициент кластеризации 0. Кроме того, некоторые пары исполнителей не соединяются друг с другом никакими путями. Однако упрощенный граф *"исполнитель–исполнитель"* (performer–performer), созданный при соединении двух исполнителей ребром, если они снимались в том же фильме, является классическим примером графа "тесного мира" (если отбросить всех исполнителей, никак не связанных с Кевином Бэйконом). На схеме, приведенной ниже, показаны графы "исполнитель–фильм" и "исполнитель–исполнитель", связанные небольшим файлом списка фильмов и исполнителей.

Программа 4.5.6 (`performer.py`) — это сценарий, создающий граф "исполнитель–исполнитель" из файла во входном формате списка фильмов и исполнителей. Помните, что каждая строка в приведенном файле состоит из названия фильма, сопровождаемого списком всех его исполнителей, разграниченных символом косой черты. Сценарий соединяет все пары исполнителей в этом фильме, добавляя ребра между каждой парой. Выполнение этого действия для каждого фильма во вводе создает граф, соединяющий исполнителей как нужно.

Файл фильмов и исполнителей

```
% more tinyMovies.txt
Movie 1/Actor A/Actor B/Actor H
Movie 2/Actor B/Actor C
Movie 3/Actor A/Actor C/Actor G
```



Два разных графа представления файла фильмов и исполнителей

Поскольку у графа "исполнитель–исполнитель" обычно имеется больше[3] ребер, чем у соответствующего графа "исполнитель–фильм", мы в настоящий момент будем работать с меньшим графом "исполнитель–исполнитель", полученным из файла `moviesg.txt`, содержащего 1 261 фильм и 19 044 исполнителя (все они связаны с Кевином Бэйконом). Программа `performer.py` свидетельствует, что у графа "исполнитель–исполнитель", связанного с файлом `moviesg.txt`, есть 19 044 вершины и 1 415 808 ребер, а следовательно, средняя степень вершины составляет 148,7 (примерно половину от $20 \lg V = 284{,}3$), что означает его разреженность; его средняя длина пути 3,494 (намного меньше, чем $10 \lg V = 142{,}2$), а значит, его пути коротки; его коэффициент кластеризации 0,911, а значит, есть локальная кластеризация. Таким образом, граф "тесного мира" обнаружен! Эти

[3] Вероятно, имелось в виду меньше. — *Примеч. ред.*

вычисления подтверждают гипотезу о том, что графы социальных отношений такого типа демонстрируют феномен "тесного мира". Вы можете найти другие реальные графы и проверить их с помощью программы `smallworld.py`. (Некоторые из них есть в упражнениях в конце этого раздела.)

Программа 4.5.6. Граф "исполнитель-исполнитель" (`performer.py`)

```
import sys
import stdio
import smallworld
from graph import Graph
from instream import InStream

file      = sys.argv[1]
delimiter = sys.argv[2]
graph = Graph()
instream = InStream(file)
while instream.hasNextLine():
    line = instream.readLine()
    names = line.split(delimiter)
    for i in range(1, len(names)):
        for j in range(i+1, len(names)):
            graph.addEdge(names[i], names[j])

degree = smallworld.averageDegree(graph)
length = smallworld.averagePathLength(graph)
cluster = smallworld.clusteringCoefficient(graph)
stdio.writef('number of vertices = %d\n', graph.countV())
stdio.writef('average degree = %7.3f\n', degree)
stdio.writef('average path length = %7.3f\n', length)
stdio.writef('clustering coefficient = %7.3f\n', cluster)
```

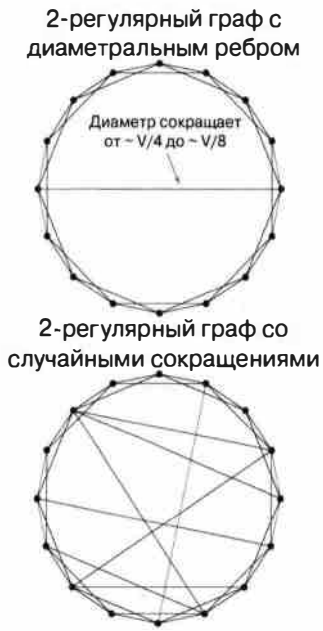
Этот сценарий — клиент `smallworld.py`, получает как аргументы командной строки имя файла фильмов и исполнителей, а также разделитель. Он создает ассоциированный граф "исполнитель-исполнитель" и выводит на стандартное устройство вывода количество вершин, их среднюю степень, среднюю длину пути и коэффициент кластеризации этого графа. Подразумевается, что граф "исполнитель-исполнитель" соединен (см. упр. 4.5.22), а следовательно, средняя длина страницы определена.

```
% python performer.py tinymovies.txt "/"
number of vertices     = 5
average degree         =  2.800
average path length    =  1.300
clustering coefficient =  0.767
```

```
% python performer.py moviesg.txt "/"
number of vertices     = 19044
average degree         = 148.688
average path length    =  3.494
clustering coefficient =  0.911
```

Один из подходов к пониманию феномена "тесного мира" подразумевает разработку математической модели, которую мы можем использовать для проверки гипотезы и прогнозов. В заключение возвратимся к проблеме разработки модели графа, способной помочь нам лучше понять феномен "тесного мира". Хитрость в разработке такой модели подразумевает комбинацию двух разреженных графов: 2-регулярного графа (с высоким коэффициентом кластеризации) и случайного графа (с малой средней длиной пути).

Регулярные графы со случайными сокращениями. Один из самых удивительных фактов, выявленных в работе Ваттса и Строгаца, в том, что добавление относительно небольшого количества случайных ребер в разреженный граф с локальной кластеризацией создает граф "тесного мира". Чтобы понять, почему дело обстоит именно так, рассмотрим 2-регулярный граф с диаметром (длина пути между самой дальней парой вершин) ~ $V/4$ (см. рис.). Добавление одного ребра, соединяющего диаметрально противоположные вершины, уменьшит диаметр до ~ $V/8$ (см. упр. 4.5.18). Добавление $V/2$ *случайных* ребер "сокращения" в 2-регулярный граф весьма вероятно значительно снизит среднюю длину пути, сделав ее логарифмически зависимой (см. упр. 4.5.25). Кроме того, это происходит при увеличении средней степени только на 1 и снижении коэффициента кластеризации менее, чем на 1/2. Таким образом, 2-регулярный граф с $V/2$ случайных ребер сокращения, чрезвычайно вероятно, будет графом "тесного мира"!



Новая модель графа

Генераторы, создающие графы из таких моделей, просты в разработке, и мы можем использовать программу `smallworld.py` для определения, демонстрируют ли графы феномен "тесного мира" (см. упр. 4.5.22). Мы можем также проверить аналитические результаты, полученные для таких простых графов, как `tinygraph.txt`, полных и регулярных графов. Как и в большинстве научных исследований, новые вопросы возникают так быстро, как быстро мы отвечаем на старые. Сколько случайных сокращений необходимо добавить, чтобы получить короткую среднюю длину пути? Каковы средняя длина пути и коэффициент кластеризации в случайном связном графе? Какие другие модели графа могли бы подойти для исследования? Сколько выборок необходимо для точной оценки коэффициента кластеризации или средней длины пути в огромном графе? Вы можете найти в упражнениях много рекомендаций для решения таких вопросов и для дальнейших исследований феномена "тесного мира". Обладая базовыми

инструментальными средствами и подходом к программированию, выработанным в этой книге, вы имеете все необходимое для решения этих и многих других научных вопросов.

Модель	Средняя степень	Средняя длина пути	Коэффициент кластеризации
Полный	999 ○	1 ●	1.0 ●
2-регулярный	4 ●	125.38 ○	0.5 ●
Случайный связный граф при p = 10/V	10 ●	3.26 ●	0.010 ○
2-регулярный с V/2 случайных сокращений	5 ●	5.71 ●	0.343 ●

Параметры "тесного мира" для разных графов на 1 000 вершин

Уроки. Этот случай иллюстрирует важность алгоритмов и структур данных в научном исследовании. Он также подтверждает некоторые из уроков, полученных в этой книге, и заслуживающих повторения.

Тщательно прорабатывайте свой тип данных — это одно из самых часто упоминаемых нами рекомендаций в этой книге. Эффективное программирование возможно только на основании точного понимания возможного набора значений типа данных и операций с ними. Использование такого современного объектно-ориентированного языка программирования, как Python, открывает путь к этому пониманию, поскольку мы можем разрабатывать, создавать и использовать собственные типы данных. Наш фундаментальный тип данных `Graph` — результат опыта и многих итераций проектирования, обсуждавшихся здесь. Простота и ясность нашего клиентского кода — свидетельство значения серьезного отношения к проектированию и реализации базовых типов данных в любой программе.

Разрабатывайте код последовательно. Подобно всем нашими другим случаям, мы создавали свое программное обеспечение модуль за модулем, тщательно проверяя и изучая каждый из них, прежде чем переходить к следующему.

Сначала решайте понятные задачи, затем малопонятные. Наш пример поиска кратчайшего набора авиарейсов между несколькими городами прост и интуитивно понятен. Он был достаточно сложен, чтобы поддержать интерес к отладке и трассировке, но не столь сложен, чтобы сделать эти задачи излишне трудоемкими.

Продолжайте проверять и проверять результаты. При работе со сложными программами, обрабатывающими огромные объемы данных, нельзя оказаться чересчур тщательным при проверке результатов. Используйте здравый смысл для вычисления каждого выводимого вашей программой бита. Начинающие

программисты мыслят оптимистически ("Если программа дает ответ, то она корректна"); мышление опытных программистов более пессимистично ("Если что то может пойти не так, то так и будет") и ближе к истине.

Используйте реальные данные. Файл `movies.txt` из *базы данных фильмов в Интернете* — только один из примеров файлов данных, вездесущих ныне в веб. В прошлом такие данные зачастую скрывались в закрытых или узко известных форматах, но теперь большинство людей понимают, что простые текстовые форматы предпочтительней. Различные методы типа данных Python `str` облегчают работу с реальными данными и являются наилучшим способом формулирования гипотез о реальных явлениях. Начните работу с небольших файлов в реальном формате, чтобы проверить производительность до того, как приступать к огромным файлам.

Многократное использование программного обеспечения. Еще один из наших советов, регулярно повторяемых в этой книге: эффективное программирование возможно только на основании точного понимания фундаментальных типов данных, доступных для использования, чтобы нам не пришлось переписывать код базовых функций. Наш главный пример — использование типов `dict` и `set` в типе `Graph`. Большинство программистов все еще предпочитают низкоуровневые представления и реализации, использующие вместо графов связанные списки или массивы, а следовательно, вынуждены повторно реализовать такие простые операции, как поддержка и пересечение связанных списков. Наш класс кратчайших путей `PathFinder` использует типы `dict`, `list`, `Graph` и `Queue` — фундаментальные структуры данных.

Вопросы производительности. Без хороших алгоритмов и структур данных большинство решенных в этой главе задач остались бы нерешенными, поскольку прямые методы требуют до невозможности большого периода времени или пространства. Необходимо понимание приблизительных потребностей в ресурсах наших программ.

Данный анализ вполне уместен в завершение книги, поскольку рассмотренные здесь программы являются лишь отправной точкой, а не полным исследованием. Эта книга — также лишь отправная точка для ваших дальнейших исследований в науке, математике или технике. Изученные здесь подходы к программированию и инструментальным средствам должны подготовить вас к решению любых вычислительных задач.

Вопросы и ответы

Сколько различных графов возможно, если дано V вершин?

Без петель или параллельных ребер возможно $V(V-1)/2$ ребер, каждое из которых может присутствовать или отсутствовать, таким образом, всего получается $2^{V(V-1)/2}$. Как демонстрируется в следующей таблице, это значение растет весьма быстро.

V	1	2	3	4	5	6	7	8	9
$2^{V(V-1)/2}$	1	2	8	64	1 024	32 768	2 097 152	268 435 456	68 719 476 736

Эти огромные числа дают немного для понимания сложностей социальных отношений. Например, если рассматривать только девять первых встречных человек, то можно обнаружить более 68 триллионов возможных взаимных знакомств!

Может ли у графа быть вершина, не соединенная ребром ни с какой другой вершиной?

Хороший вопрос. Такая вершина известна как *изолированная вершина* (isolated vertices). Наша реализация отбрасывает ее. Другая реализация могла бы допускать изолированные вершины, если включить явный метод `addVertex()` для операции *добавления вершины*.

Почему методы запроса `countV()` и `countE()` должны иметь реализации постоянного времени? Разве большинство клиентов не будут вызывать такие методы только однажды?

Так могло бы показаться, но код

```
while i < g.countE():
    ...
    i += 1
```

занял бы квадратичное время, если вы должны были бы использовать ленивую реализацию, подсчитывающую ребра, а не хранящую переменную экземпляра с количеством ребер.

Почему классы `Graph` и `PathFinder` находятся в отдельных классах? Разве не имело бы смысла включать методы класса `PathFinder` в API класса `Graph`?

Поиск кратчайших путей является лишь одним из многих алгоритмов обработки графа. Включение всех их в один интерфейс было бы плохим проектным решением. Перечитайте обсуждение широких интерфейсов в разделе 3.3.

Упражнения

4.5.1. Найдите в файле `movies.txt` исполнителя, участвовавшего в наибольшем количестве фильмов.

4.5.2. Измените метод `__str__()` в классе `Graph` так, чтобы он возвращал вершины в отсортированном порядке (подразумевается, что вершины сравнимы). *Подсказка*: используйте встроенную функцию `sorted()`.

4.5.3. Модифицируйте метод `__str__()` в классе `Graph` так, чтобы он выполнялся за время, линейно пропорциональное количеству вершин и количеству ребер в самом плохом случае. *Подсказка*: используйте метод `join()` типа данных `str` (см. упр. 4.1.13).

4.5.4. Добавьте в тип `Graph` метод `copy()`, создающий и возвращающий новую, независимую копию графа. Любые последующие изменения исходного графа не должны затрагивать вновь созданный граф (и наоборот!).

4.5.5. Составьте версию класса `Graph`, обеспечивающего явное создание вершин и допускающего петли, параллельные ребра и *изолированные вершины* (вершины степени 0).

4.5.6. Добавьте в класс `Graph` метод `removeEdge()`, получающий два строковых аргумента и удаляющий из графа заданное ребро, если оно там есть.

4.5.7. Добавьте в класс `Graph` метод `subgraph()`, получающий как аргумент набор строк и возвращающий *индуцированный подграф* (граф, состоящий только из тех вершин и только тех ребер исходного графа, которые соединяют любые два из них).

4.5.8. Опишите преимущества и недостатки использования для представления соседей вершины массива или связанного списка вместо типа `set`.

4.5.9. Составьте клиент класса `Graph`, который читает граф из файла, а затем выводит его ребра по одному в строку.

4.5.10. Модифицируйте класс `Graph` так, чтобы он поддерживал вершины любого хешируемого типа.

4.5.11. Реализуйте программу `allshortestpaths.py` как клиент класса `PathFinder`, который получает как аргументы командной строки имя файла графа и разделитель, а затем создает объект класса `PathFinder` для каждой вершины. Программа последовательно получает со стандартного ввода имена двух вершин (в одной строке, отделенные разделителем) и выводит соединяющий их кратчайший путь. *Примечание.* Для файла `movies.txt` именами вершин могут быть как исполнители, так и фильмы или исполнитель и фильм.

4.5.12. *Истина или ложь.* В некий момент поиска в ширину очередь может содержать две вершины, с дистанцией от отправного пункта 7 и с дистанцией 9. *Решение.* Ложь. Очередь может содержать вершину самое большее двух разных дистанций d и $d+1$. Поиск в ширину исследует вершины в порядке увеличения дистанции от отправного пункта. При исследовании вершины с дистанцией d в очередь может быть поставлена вершина только с дистанцией $d+1$.

4.5.13. Докажите индукцией на наборе посещенных вершин, что `PathFinder` находит кратчайшие пути от исходного пункта до каждой вершины.

4.5.14. Предположим, в классе `PathFinder` вы используете стек вместо очереди для поиска в ширину. Путь все еще находится? Кратчайшие пути все еще вычисляются правильно? В каждом случае докажите, что это работает, или приведите контрдовод.

4.5.15. Напишите программу, составляющую график зависимости средней длины пути от количества случайных ребер как случайных сокращений, добавленных к 2-регулярному графу на 1 000 вершин.

4.5.16. Добавьте необязательный аргумент `k` в метод `clusterCoefficient()` из модуля `smallworld.py`, чтобы он вычислял локальный коэффициент кластеризации для графа на основании общего количества имеющихся и возможных ребер из набора вершин в пределах дистанции `k` от каждой вершины. Используйте стандартное значение `k=1`, чтобы ваша функция давала результаты, идентичные функции с тем же именем в программе `smallworld.py`.

4.5.17. Докажите, что коэффициент кластеризации в k-регулярном графе составляет $(2k-2)/(2k-1)$. Представьте формулу для средней длины пути в k-регулярном графе на V вершин как функцию от V и k.

4.5.18. Докажите, что диаметр 2-регулярного графа на V вершин составляет $\sim V/4$. Докажите, что добавление одного ребра, соединяющего две диаметрально противоположные вершины, уменьшает диаметр до $\sim V/8$.

4.5.19. Проведите вычислительные эксперименты для подтверждения того, что средняя длина пути в регулярном графе на V вершин составляет $\sim 1/4\ V$. Повторимся, но добавьте одно случайное ребро в регулярный граф и подтвердите, что средняя длина пути уменьшается до $\sim 3/16\ V$.

4.5.20. Добавьте в программу `smallworld.py` функцию `isSmallWorld()`, которая получает как аргумент граф и возвращает `True`, если граф демонстрирует феномен "тесного мира" (как определено пороговыми значениями в тексте), и значение `False` в противном случае.

4.5.21. Реализуйте клиент проверки `main()` для программы 4.5.5 (`smallworld.py`), создающий вывод, как в примере запуска. Программа должна получать в аргументах командной строки имя файла графа и разделитель, а выводить количество вершин и ребер, среднюю степень, среднюю длину пути и коэффициент кластеризации для графа, а также указывать, являются ли значения слишком большими или слишком маленькими для графа, демонстрирующего феномен "тесного мира".

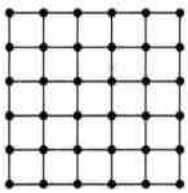
4.5.22. Составьте программу, создающую случайные связные графы и 2-регулярные графы со случайными сокращениями. Используя программу `smallworld.py`, создайте 500 случайных графов обеих моделей (на 1 000 вершин каждый) и вычислите их среднюю степень, среднюю длину пути и коэффициент кластеризации. Сравните свои результаты с соответствующими значениями из таблицы на странице 696.

4.5.23. Составьте программу `smallworld.py` и клиент типа `Graph`, создающий *k*-регулярные графы, и проверьте, демонстрируют ли они феномен "тесного мира" (сначала выполните упр. 4.5.20).

4.5.24. В *решетчатом графе* (grid graph) вершины упорядочены в таблицу *n* на n с ребрами, соединяющими каждую вершину с ее соседями выше, ниже, влево и вправо по решетке. Составьте программу `smallworld.py` и клиент типа `Graph`, который создает решетчатые графы и проверяет, демонстрируют ли они феномен "тесного мира" (сначала выполните упр. 4.5.20).

3-регулярный граф

4.5.25. Дополните свои решения из предыдущих двух упражнений, чтобы получить также из командной строки аргумент *m* и добавить в граф *m* случайных ребер. Поэкспериментируйте со своими программами для графов приблизительно на 1 000 вершин в поисках графа "тесного мира" с относительно немногими ребрами.



Решетчатый граф 6 на 6

4.5.26. Составьте клиент типов `Graph` и `PathFinder`, который получает как аргументы имя файла фильмов и исполнителей, а также разделитель, и выводит новый файл фильмов и исполнителей после удаления всех фильмов, никак не связанных с Кевином Бэйконом.

Практические упражнения

4.5.27. *Большие числа Бэйкона.* Найдите в файле `movies.txt` исполнителей с наибольшим, но конечным числом Бэйкона.

4.5.28. *Гистограмма.* Составьте программу `baconhistorgram.py`, которая выводит гистограмму чисел Бэйкона, демонстрируя количество исполнителей из файла `movies.txt`, обладающих числами Бэйкона 0, 1, 2, 3,.... Включите категорию для обладающих бесконечным числом (никак не связанных с Кевином Бэйконом вообще).

4.5.29. *Граф "исполнитель–исполнитель".* Как упоминалось в тексте, альтернативный способ вычисления чисел Бэйкона подразумевает построение графа, где для каждого исполнителя есть вершина (но не для каждого фильма), два исполнителя соединяются ребром, если они оба участвовали в фильме. Вычислите числа Бэйкона поиском в ширину на графе "исполнитель–исполнитель". Сравните его продолжительность с продолжительностью для `movies.txt`. Объясните, почему этот подход настолько медленнее. Объясните также, что нужно сделать, чтобы включить фильмы вдоль пути, как это делается автоматически в нашей реализации.

4.5.30. *Компонента связности графа.* В неориентированном графе *компонента связности графа* (connected component) — это максимальный набор взаимно достижимых вершин. Составьте тип данных `ConnectedComponents`, вычисляющий компоненты связности графа. Предоставьте конструктор, получающий как аргумент `Graph` и вычисляющий все компоненты связности, используя поиск в ширину. Включите метод `areConnected(v, w)`, возвращающий значение `True`, если `v` и `w` находятся в той же компоненте связности, и значение `False` в противном случае. Добавьте также метод `components()`, возвращающий количество компонентов связности.

4.5.31. *Заливка.* `Picture` (изображение) — это двумерный массив значений `Color` (см. раздел 3.1), представляющих цвета пикселей. `blob` — это коллекция соседних пикселей того же цвета. Составьте клиент типа `Graph`, конструктор которого создает решетчатый граф (см. упр. 4.5.24) из заданного изображения и обеспечивает операцию *заливки.* Получив координаты `col` и `row` пикселя, а также цвет `c`, он изменяет цвет этого и всех остальных пикселей в той же коллекции `blob` на `c`.

4.5.32 *Лестница слов.* Составьте программу `wordladder.py` (игру Word Ladder), получающую из командной строки две 5-буквенных строки и читающую со стандартного ввода список 5-буквенных слов. Программа выводит самую короткую последовательность слов, используя слова со стандартного ввода, соединяющую эти две строки (если она существует). Два слова могут быть соединены, если они отличаются только одним символом. Например, слова `green` и `brown` соединяет следующая последовательность:

```
green greet great groat groan grown brown
```

Составьте простой фильтр, извлекающий 5-буквенные слова из системного словаря для стандартного устройства ввода или загрузите список с сайта книги. (Эта игра, первоначально называвшаяся *doublet*, была изобретена Льюисом Кэрроллом.)

4.5.33. *Все пути.* Составьте класс `AllPaths` как клиент класса `Graph`, конструктор которого получает как аргумент объект класса `Graph` и обеспечивает операции подсчета и вывода всех простых путей между двумя заданными вершинами `s` и `t` в графе. *Простой* (simple) путь не проходит через вершину более чем один раз. В двумерных таблицах такие пути получаются в результате блуждания без самопересечений (см. раздел 1.4). Это фундаментальная проблема в статистической физике и теоретической химии, например, в модели пространственного расположения линейных молекул полимера. *Предупреждение*: количество путей может расти экспоненциально.

4.5.34. *Порог просачивания.* Разработайте модель графа для задачи просачивания и составьте клиент типа `Graph`, осуществляющий то же вычисление, что и программа 2.4.6 (`percolation.py`). Оцените порог просачивания для треугольных, квадратных и шестиугольных таблиц.

4.5.35. *Графы метро.* В системе метро Токио маршруты помечены символами, а остановки номерами, например `G-8` или `A-3`. Станции с переходами являются наборами остановок. Найдите карту метро Токио в веб, разработайте простой формат базы данных и составьте клиент класса `Graph`, читающий файл и способный отвечать на запросы о кратчайшем пути между станциями системы метро Токио. Если вы предпочитаете Парижскую систему метро, то ее маршруты являются последовательностями названий станций, а пересадки возможны на станциях с совпадающими названиями.

4.5.36. *Центр вселенной Голливуд.* Мы можем измерить, насколько Кевин Бэйкон хорош на роль центра вселенной Голливуд, вычислив *Голливудские числа* (Hollywood number) каждого исполнителя или среднюю длину пути. Голливудское число Кевина Бэйкона является средним значением чисел Бэйкона всех исполнителей (в его компоненте связности). Голливудское число другого исполнителя вычисляется так же, но только с этим исполнителем в качестве исходного пункта вместо Кевина Бэйкона. Вычислив Голливудское число Кевина Бэйкона, найдите исполнителя с лучшим Голливудским числом, чем у него. Найдите исполнителей (в том же компоненте связности, что и Кевин Бэйкон) с наилучшими и наихудшими Голливудскими числами.

4.5.37. *Диаметр. Эксцентриситет* (eccentricity) вершины — это самая большая дистанция между этой и любой другой вершиной. *Диаметр* (diameter) графа — это самая большая дистанция между любыми двумя вершинами (максимальный эксцентриситет любой вершины). Составьте клиент класса `Graph`, `diameter.py`, способный вычислить эксцентриситет вершины и диаметр графа. Используйте его для поиска диаметра графа, представленного в файле `movies.txt`.

4.5.38. *Ориентированные графы.* Реализуйте тип данных `Digraph`, представляющий *ориентированный граф* (directed graph), где направление ребер существенно: метод `addEdge(v, w)` означает добавление ребра от `v` до `w`, но *не* от `w` до `v`. Замените метод `adjacentTo()` двумя методами: `adjacentFrom()` — для добавления набора вершин, ребра которых направлены *от* вершины аргумента, и `adjacentTo()` — для добавления набора вершин, ребра которых направлены *к* вершине аргумента. Объясните, как изменить тип `PathFinder` для поиска кратчайших путей в ориентированных графах.

4.5.39. *Случайная навигация.* Измените свой класс `Digraph` из предыдущего упражнения так, чтобы получить класс `MultiDigraph`, допускающий параллельные ребра. Для клиента проверки запустите модель случайной навигации, как в программе 1.6.2 (`randomsurfer.py`).

4.5.40. *Транзитивное замыкание.* Составьте класс `TransitiveClosure`, клиент класса `Digraph`, конструктор которого получает как аргумент объект класса `Digraph` и чей метод `isReachable(v, w)` возвращает значение `True`, если вершина `w` достижима от вершины `v` по направленному пути в ориентированном графе, и значение `False` в противном случае. *Подсказка:*

проведите поиск в ширину от каждой вершины, как в упражнении 4.5.11 (`allshortestpaths.py`).

4.5.41. *Статистическая выборка.* Используйте статистическую выборку для оценки средней длины пути и коэффициента кластеризации графа. Например, для оценки коэффициента кластеризации выберите `t` случайных вершин и вычислите среднее из коэффициентов кластеризации этой вершины. Выполнение ваших функций должно быть на порядки быстрее, чем у соответствующих функций `smallworld.py`.

4.5.42. *Время покрытия.* Случайное блуждание в связанном неориентированном графе осуществляется перемещением от одной вершины к соседней, выбранной из других с равной вероятностью. (Для неориентированных графов процесс случайной навигации аналогичен.) Напишите программы для проведения экспериментов, обеспечивающих выработку гипотез о количестве этапов, необходимых для посещения каждой вершины в графе. Каково время покрытия для полного графа на V вершин? Регулярного графа? 2-регулярного графа? Можно ли найти семейство графов, где время покрытия растет пропорционально V^3 или 2^V?

4.5.43. *Модель случайного графа Эрдёша-Реньи.* В классической модели случайного графа мы создаем случайный граф на V вершин, включив каждое возможное ребро с вероятностью p, независимо от других ребер. Составьте клиент класса `Graph` для проверки следующих свойств.

- *Пороговое значение включения.* Если $p < 1 / V$, а V велико, то большинство компонентов связности будут малы, с наибольшим логарифмическим по размеру. Если $p > 1 / V$, т.е. почти наверняка гигантский компонент, содержащий почти все вершины. Если $p < \ln V / V$, то граф с высокой вероятностью разъединен; если $p > \ln V / V$, граф с высокой вероятностью будет связным.
- *Распределение степеней.* Распределение степеней следует биномиальному распределению, центрированному по среднему. Таким образом, у большинства вершин будут подобные степени. Вероятность, что вершина соединена с k другими вершинами, уменьшается экспоненциально от k.
- *Без концентраторов.* Максимальная степень вершины при постоянном p находится по крайней мере в логарифмической зависимости от V.

- *Без локальной кластеризации.* Коэффициент кластеризации близок к 0, если граф разрежен и связан. Случайные графы не являются графами "тесного мира".
- *Короткие длины пути.* Если $p > \ln V / V$, то диаметр графа (см. упр. 4.5.37) будет логарифмическим.

4.5.44. *Экспоненциальный закон веб-ссылок.* Полустепени захода и исхода страниц в веб подчиняются экспоненциальному закону, который может быть смоделирован процессом *предпочтительных связей* (preferred attachment). Предположим, у каждой веб-страницы есть только одна исходящая ссылка. Каждая страница создается по одной, начиная с одной страницы, указывающей на себя. С вероятностью $p < 1$ она связывается с одной из существующих страниц, выбранных однородно наугад. С вероятностью $1 - p$ она связывается с существующей страницей с вероятностью, пропорциональной количеству входящих ссылок данной страницы. Это правило отражает общую тенденцию у новых веб-страниц указывать на популярные страницы. Составьте программу, моделирующую этот процесс и рисующую гистограммы для количеств входящих ссылок. *Частичное решение.* Доля страниц с полустепенью захода k пропорциональна $k^{-1/(1-p)}$.

4.5.45. *Глобальный коэффициент кластеризации.* Добавьте в программу `small world.py` функцию, вычисляющую глобальный коэффициент кластеризации графа. *Глобальный коэффициент кластеризации* (global clustering coefficient) — это условная вероятность того, что две случайных вершины, являющиеся общими соседями некой вершины, являются соседями друг друга. Найдите графы, у которых локальные и глобальные коэффициенты кластеризации различаются.

4.5.46. *Модель графа Ваттса–Строгаца* (см. упр. 4.5.24-4.5.25). Ваттс и Строгац предложили гибридную модель, содержащую типичные связи вершин друг около друга (люди, знающие своих ближайших соседей), плюс некоторые случайные дальние связи. Нарисуйте график влияния добавления случайных ребер в решетчатый граф размером $n \times n$ на среднюю длину пути и на коэффициент кластеризации для $n = 100$. Сделайте то же для k-регулярных графов из V вершин, для $V = 10\,000$ и для различных значений k до $10 \log V$.4.5.47. *Модель графа Боллобаша–Чунга.* Боллобаш и Чунг предложили гибридную модель, комбинирующую 2-регулярный граф на V вершин (при четном V) и *случайное соответствие* (random

matching). *Соответствие* (matching) — это граф, каждая вершина которого имеет степень 1. Для создания случайного соответствия перетасуйте V вершин и добавьте ребра между вершинами i и $i + 1$ в случайном порядке. Определите степень каждой вершины для графов в этой модели. Используя программу `smallworld.py`, оцените среднюю длину пути и коэффициент кластеризации для случайных графов, созданных согласно этой модели для $V = 1\,000$.

4.5.48. *Модель графа Клейберга.* В модели Ваттса–Строгаца у участников не было никакого способа поиска коротких путей в децентрализованной сети. Но у эксперимента Милгрэма был также важный алгоритмический компонент — индивидуумы могут найти короткие пути! Джон Клейберг предложил подчинить распределение сокращений экспоненциальному закону с вероятностью, пропорциональной d-й степени дистанции (в размерностях d). У каждой вершины есть *один* дальний сосед. Составьте программу, которая создает графы согласно этой модели с клиентом проверки, использующим программу `smallworld.py` для проверки соответствия феномену "тесного мира". Нарисуйте гистограммы для демонстрации единообразия графов по всем масштабам дистанции (те же количества ссылок как на дистанциях 1–10, так и на дистанциях 10–100 или 100–1000). Составьте программу, которая вычисляет средние длины путей, полученных при выборке ребер, прокладывающих путь по возможности ближе к целевому объекту с точки зрения дистанций решетки, и проверьте гипотезу о том, что их среднее пропорционально $(\log V)^2$.

В этом заключительном разделе мы переносим ваши вновь приобретенные знания программирования в более широкий контекст, кратко описав некоторые из фундаментальных элементов мира вычисления, с которым вы, вероятно, встретитесь. Надеемся, что эта информация подтолкнет вас использовать полученные навыки программирования как отправную точку для дальнейшего изучения роли вычислений в окружающем мире.

Теперь вы умеете программировать. Подобно тому, как несложно научиться управлять джипом, умея управлять автомобилем, умея программировать на одном языке, несложно самостоятельно научиться программировать на другом. Многие ученые регулярно используют несколько разных языков в различных целях. Встроенные типы данных, условные выражения, циклы и функции, обсуждавшиеся в главах 1 и 2 (верой и правдой служащие программистам на протяжении двух десятилетий), и рассмотренный в главе 3 объектно-ориентированный подход программирования (используемый современными программистами) являются фундаментальными моделями, присущими многим языкам программирования. Полученный навык в использовании их и фундаментальных типов данных (глава 4) подготовил вас к работе с библиотеками, средами разработки программ и специализированными приложениями всех типов. Вы также готовы оценить важность абстракции при разработке сложных систем и понимания того, как они работают.

Изучение *информатики* — это значительно больше, чем умение программировать. Теперь, когда вы знакомы с программированием и сведущи в вычислениях, вы хорошо подготовлены к восприятию некоторых из выдающихся интеллектуальных достижений прошлого столетия, важнейших нерешенных проблем нашего времени и их роли в развитии окружающей нас вычислительной инфраструктуры. Возможно, вычисления играют даже более существенную, постоянно возрастающую роль в нашем понимании природы, от генетики до молекулярной динамики и астрофизики, о чем мы намекали не раз в этой книге. Дальнейшее изучение фундаментальных принципов информатики бесспорно принесет вам дивиденды.

Стандартные модули Python. Система Python предоставляет вам широкий диапазон ресурсов для использования. Мы интенсивно использовали некоторые из стандартных модулей Python, такие как `math`, но большинство из них игнорировали. В сети доступно множество информации о стандартных модулях. Если вы еще не просмотрели стандартные модули Python, то теперь самое время сделать это. Вы найдете, что большая часть этого кода предназначена для использования профессиональными разработчиками, но существует множество модулей, которые, вероятно, заинтересуют и вас. Возможно, важнейшая вещь, которую стоит иметь в виду, изучая модули, заключается в том, что вы не *обязаны* использовать их, а *можете* сделать это. Когда вы находите API, который, кажется, удовлетворяет вашим потребностям, обязательно используйте его в своих интересах.

Среды программирования. Конечно, в будущем вы будете использовать и другие среды программирования, а не только Python. Многие программисты, даже опытные профессионалы, застряли между прошлым из-за огромных объемов устаревшего кода на таких старых языках, как C, C++ и Fortran, и будущим, из-за наличия таких современных инструментальных средств, как JavaScript, Ruby, Java, Python и Scala. И опять-таки, возможно, самое важное, что следует иметь в виду при выборе языка программирования: вы не *обязаны* использовать их, а *можете* сделать это. Если для ваших потребностей лучше подходит некий другой язык, конечно, используйте его. Люди, настаивающие на использовании одной среды программирования, по любым причинам, просто упускают возможности.

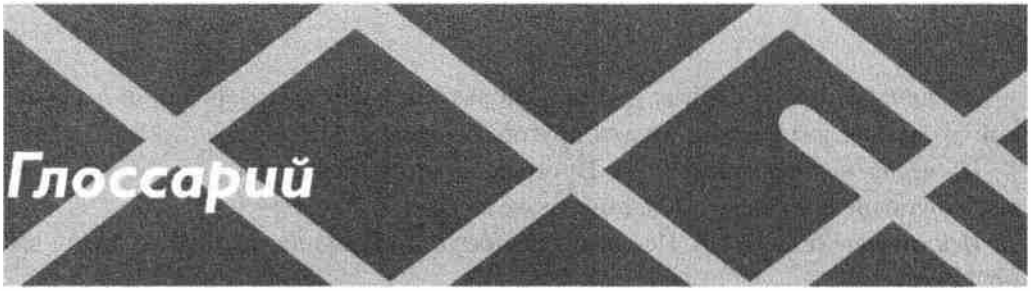
Научные вычисления. Иногда числовые вычисления могут быть очень сложными из-за проблем, связанных с точностью, поэтому использование библиотек математических функций безусловно оправдано. Многие ученые используют Fortran, старый научный язык; многие другие используют язык Matlab, разработанный специально для вычислений с матрицами. Комбинация хороших библиотек и встроенных матричных операций делает Matlab привлекательным выбором для решения многих задач. Но поскольку языку Matlab не хватает поддержки изменяемых типов и других современных средств, язык Python — лучший выбор для решения многих других задач. Вы *можете* использовать оба языка! В Python доступны те же математические библиотеки, используемые программистами Matlab и Fortran. Действительно, у сообщества Python есть обширные возможности благодаря библиотекам *NumPy* (числовой Python) и *SciPy* (научный Python). Если вы занимаетесь научными вычислениями, используйте эти библиотеки. Информацию о них можно найти на сайте книги.

Компьютерные системы. Ранее свойства конкретных компьютерных систем полностью определяли характер и уровень доступных для решения проблем, но теперь эти соображения едва ли имеют смысл. Вы все еще можете рассчитывать на появление более быстрой машины с намного большим объемом памяти в следующем году. Старайтесь разрабатывать код, независимый от конкретной

машины, чтобы облегчить его перенос. Однако еще важнее то, что веб играет все более и более важную роль в коммерческих и научных вычислениях, как вы видели во многих примерах, приведенных в этой книге. Вы можете составлять программы обработки данных, хранимых в другом месте; программы, взаимодействующие с программами, запущенными в другом месте, и использовать в своих интересах множество других свойств обширной и развивающейся вычислительной инфраструктуры. Смело делайте это. Люди, инвестирующие существенные усилия в написание программ для специфических машин, даже высокопроизводительных суперкомпьютеров, также упускают возможности.

Теоретическая информатика. Фундаментальные пределы вычислений были вполне очевидны с самого начала и продолжают играть важную роль в определении видов задач, доступных для решения. Вас это может удивить, но есть некоторые задачи, которые не сможет решить ни одна компьютерная программа, и много других задач, возникающих обычно на практике, считающихся слишком трудными для решения на любом мыслимом компьютере. Все, кто полагается на вычисления при решении задач, в творческой работе или исследованиях, должны учитывать эти факты.

Вы, конечно, проделали длинный путь от экспериментов по созданию, компиляции и запуску программы `helloworld.py`, но вам все еще предстоит многому научиться. Продолжайте программировать и узнавать о средах программирования, научном вычислении, компьютерных системах и теории информатики, и вы откроете для себя возможности, о которых другие даже не предполагают.



API (Application Programming Interface — интерфейс прикладных программ). Спецификация набора операций, характеризующая то, как клиент может использовать тип данных.

Алгоритм (algorithm). Последовательность процедур для решения задачи, например алгоритм Евклида.

Аргумент (argument). Передаваемая функции ссылка на объект.

Аргумент командной строки (command-line argument). Строка, передаваемая программе в командной строке.

Бит (bit). Одно из двух двоичных значений (0 или 1).

Возвращаемое значение (return value). Объект (ссылка), полученный как результат вызова функции.

Встроенная функция (built-in function). Функция, встроенная в сам язык Python, например `max()`, `abs()`, `int()`, `str()` или `hash()`.

Встроенный тип (built-in type). Тип данных, встроенный в сам язык Python, например `str`, `float`, `int`, `bool`, `list`, `tuple`, `dict` или `set`.

Вызов функции (function call). Выражение, выполняющее функцию и возвращающее значение.

Выражение (expression). Комбинация литералов, переменных, операторов и вызовов функций (возможно заключение в скобки), которая может быть упрощена до результирующего значения.

Вычисление выражения (evaluate an expression). Упрощение выражения до значения за счет применения операторов к операндам выражения согласно правилам приоритета.

Выявление ошибки (raise an error). Сообщение об ошибке во время компиляции или выполнения программы.

Глобальная переменная (global variable). Переменная, определенная вне какой-либо функции или класса.

Глобальный код (global code). Код, находящейся вне определения какой-либо функции или класса.

Закрытый (private). Код реализации, недоступный клиентам.

Идентификатор (identifier). Имя, используемое для идентификации переменной, функции, класса, модуля или другого объекта.

Интерпретатор (interpreter). Программа, выполняющая код, написанный на высокоуровневом языке по одной строке за раз.

Исключение (exception). Исключительное состояние или ошибка во время выполнения.

Исходный код (source code). Программа или фрагмент программы на высокоуровневом языке программирования.

Итератор (iterator). Тип данных, поддерживающий встроенную функцию `next()`, вызываемую языком Python в начале каждой итерации цикла `for`.

Итерируемый тип данных (iterable data type). Тип данных, возвращающий итератор по его элементам, например `list`, `tuple`, `str`, `dict` и `set`.

Класс (class). Конструкция языка Python, позволяющая реализовать пользовательский тип данных. Предоставляется шаблон для создания и манипулирования объектами, содержащими значения этого типа, как определено API.

Клиент (client). Программа, использующая реализацию через API.

Книжный модуль (booksite module). Модуль, созданный авторами для использования в книге.

Командная строка (command line). Рабочая строка терминального приложения; используется для ввода системных команд и запуска программ.

Комментарий (comment). Пояснительный текст (игнорируется компилятором), помогающий читателю понять цель кода.

Компилятор (compiler). Программа, преобразующая код высокоуровневого языка, такого как Python, в низкоуровневый код, такой как код виртуальной машины Python.

Конструктор (constructor). Специальный метод типа данных для создания и инициализации нового объекта.

Литерал (literal). Текстовое представление в исходном коде значений встроенных числовых и строковых типов данных, таких как `123`, `'Hello'` и `True`.

Локальная переменная (local variable). Переменная, определенная в пределах функции. Ее область видимости ограничена этой функцией.

Массив (array). Структура данных, хранящая набор элементов и обеспечивающая их создание, доступ по индексу, запись по индексу и перебор.

Метод экземпляра (instance method). Реализация операции для конкретного типа данных (функция, вызываемая для определенного объекта).

Модуль (module). Файл с расширением `.py`, структурированный так, чтобы его средства могли быть многократно использованы в других программах Python.

Модульное программирование (modular programming). Стиль программирования, подразумевающий использование отдельных, независимых модулей, ориентированных на решаемую задачу.

Модульное тестирование (unit testing). Практика включения в модуль проверочного кода.

Неизменяемый объект (immutable object). Объект, значение которого не может измениться.

Неизменяемый тип данных (immutable data type). Тип данных, значение экземпляра которого не может быть изменено.

Область видимости (scope). Область программы, где переменная или имя доступны непосредственно.

Объект (object). Представление в машинной памяти значения конкретного типа данных, характеризующееся идентификатором, типом и значением.

Объект `None`. Специальный объект, который не представляет никакой объект.

Объектная ссылка (object reference). Конкретное представление идентификатора объекта (адрес области памяти, где хранится объект).

Объектно-ориентированное программирование (object-oriented programming). Стиль программирования, подразумевающий моделирование реальной или абстрактной сущности с использованием типов данных и объектов.

Операнд (operand). Объект, на который воздействует оператор.

Оператор (operator). Специальный символ (или последовательность символов), представляющий операцию со встроенным типом данных, например `+`, `-`, `*` и `[]`.

Оператор (statement). Инструкция, выполняемая языком Python, например, оператор присвоения, операторы `if`, `while` и `return`.

Оператор `import`. Оператор Python, позволяющий обращаться к коду в другом модуле.

Оператор присвоения (assignment statement). Оператор языка Python, состоящий из имени переменной, символа `=` и выражения. Объект, содержащий полученное в результате вычисления выражения значение, привязывается к переменной.

Операционная система (operating system). Программа, управляющая ресурсами и обеспечивающая общее обслуживание программ и приложений на компьютере.

Определение и инициализация переменной (defining and initializing a variable). Привязка переменной к объекту впервые в программе.

Ошибка времени выполнения программы (run-time error). Ошибка, выявленная во время выполнения программы (исключение).

Ошибка времени компиляции (синтаксическая) (compile-time (syntax) error). Ошибка, обнаруживаемая компилятором.

Параметрическая переменная (parameter variable). Переменная, определенная в определении функции и инициализируемая значением соответствующего аргумента при вызове функции.

Параметрическая переменная `self`. Первая параметрическая переменная метода, связанная с объектом, который и вызывал метод. В соответствии с соглашением эта переменная называется `self`.

Перегрузка оператора (overloading an operator). Определение поведения такого оператора, как +, *, <= или [] для типа данных.

Перегрузка функции (overloading a function). Определение поведения такой встроенной функции, как `len()`, `max()` или `abs()` для типа данных.

Передача ссылки на объект (pass by object reference). Используемый языком Python способ передачи объекта функции (за счет передачи ссылки на объект).

Переменная (variable). Контейнер, хранящий ссылку на объект.

Переменная экземпляра (instance variable). Переменная, определенная в классе (но вне любого из методов) и представляющая значение типа данных (данные, связанные с каждым экземпляром класса).

Побочный эффект (side effect). Изменение состояния, например, запись вывода, чтение ввода, выявление ошибки или изменение значения некоего постоянного объекта (переменной экземпляра, параметрической или глобальной переменной).

Полиморфизм (polymorphism). Использование тех же API (или их части) для разных типов данных.

Последовательность (sequence). Итерируемый тип данных, обеспечивающий доступ по индексу `a[i]` и `len(a)`, например, `list`, `str` и `tuple` (но не `dict`).

Постоянная переменная (constant variable). Переменная, значение типа данных которой не изменяется во время выполнения программы (или между запусками программы).

Правила приоритета (precedence rules). Правила, определяющие порядок применения операторов в выражении.

Привязка (binding). Связь между переменной и объектом, содержащим значение типа данных.

Программа (program). Последовательность инструкций, выполняемых на компьютере.

Псевдоним (alias). Две (или более) переменных, ссылающихся на тот же объект.

Реализация (implementation). Программа, реализующая набор методов, определенных в API, и предназначенная для использования клиентом.

Сбор мусора (garbage collection). Процесс автоматического выявления и освобождения уже не используемых участков памяти.

Специальный метод (special method). Один из встроенных методов языка Python, вызываемых неявно при вызове операции соответствующего типа данных, например `__plus__()`, `__eq__()` или `__len__()`.

Сравнимый тип данных (comparable data type). Тип данных Python, допускающий определение порядка значений с использованием шести операторов сравнения <, <=, >, >=, == и !=, например `int`, `str`, `float` и `bool`.

Стандартное значение (default value). Объект, присваиваемый параметрической переменной, если функции при вызове не передан соответствующий аргумент.

Стандартные устройства ввода, вывода, графики и аудио (standard input, output, drawing, and audio). Модули ввода-вывода языка Python.

Структура данных (data structure). Способ организации компьютерных данных (обычно экономящий время или место), например, массив, массив переменного размера, связанный список или бинарное дерево поиска.

Сценарий (script). Короткая программа, обычно реализуемая как глобальный код и не предназначенная для многократного использования.

Терминал (terminal). Приложение операционной системы, позволяющее вводить команды.

Тип данных (data type). Набор значений и операций, определенных для этих значений.

Функция (function). Именованная последовательность операторов, осуществляющая вычисление.

Хешируемый тип данных (hashable data type). Тип данных с определенной встроенной функцией `hash()` и допустимый для использования с такими типами данных, как `dict` и `set`, например, `int`, `str`, `float`, `bool` и `tuple` (но не `list`).

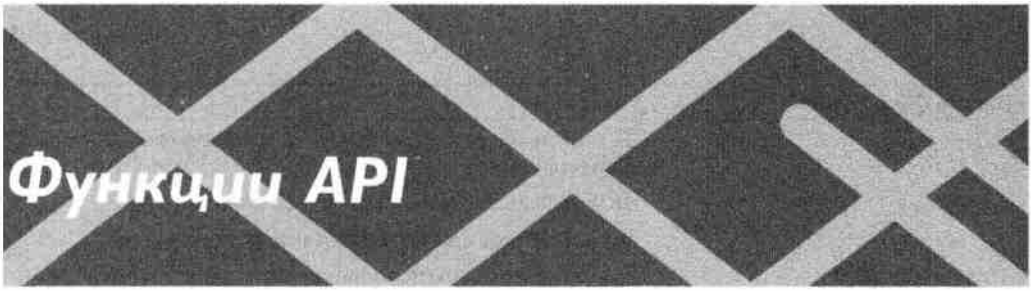
Часть (slice). Подмножество массива, строки или другой последовательности.

Чистая функция (pure function). Функция, всегда получающая те же аргументы и возвращающая то же значение, не производя заметного побочного эффекта.

Экземпляр (instance). Объект определенного класса.

Элемент (element). Один из объектов в массиве.

Элемент (item). Один из объектов в коллекции.



Модуль Python `math` (часть API)

Вызов функции	Описание
`math.sin(x)`	Синус x (выраженный в радианах)
`math.cos(x)`	Косинус x (выраженный в радианах)
`math.tan(x)`	Тангенс x (выраженный в радианах)
`math.atan2(y, x)`	Полярный угол точки (x, y)
`math.hypot(x, y)`	Евклидово расстояние между началом координат и точкой (x, y)
`math.radians(x)`	Преобразование x (в градусах) в радианы
`math.degrees(x)`	Преобразование x (в радианах) в градусы
`math.exp(x)`	Экспоненциальная функция x (ex)
`math.log(x, b)`	Логарифм x по основанию b (logbx) (e — стандартное значение основания b, т.е. натуральный логарифм)
`math.sqrt(x)`	Квадратный корень x
`math.erf(x)`	Функция ошибок x
`math.gamma(x)`	Гамма-функция x
`math.factorial(x)`	Факториал целого числа x

Примечание. Модуль `math` включает также обратные тригонометрические функции `asin()`, `acos()` и `atan()`; гиперболические функции `sinh()`, `cosh()`, `tanh()`, `asinh()`, `acosh()` и `atanh()`; а также константы `e` (`2.718281828459045`) и `pi` (`3.141592653589793`).

Наш модуль `stdio`

Вызов функции	Описание
	Функции чтения индивидуальных лексем со стандартного ввода
`stdio.isEmpty()`	Стандартный ввод действительно пуст (или это только отступ)?
`stdio.readInt()`	Читает лексему, преобразует ее в целое число и возвращает
`stdio.readFloat()`	Читает лексему, преобразует ее в число типа `float` и возвращает

Вызов функции	Описание
`stdio.readBool()`	Читает лексему, преобразует ее в логическое значение и возвращает
`stdio.readString()`	Читает лексему и возвращает ее как строку
	Функции чтения строк со стандартного ввода
`stdio.hasNextLine()`	Есть ли у стандартного ввода следующая строка?
`stdio.readLine()`	Читает следующую строку и возвращает ее как строку
	Функции чтения последовательности значений одинакового типа со стандартного ввода, пока он не пуст
`stdio.readAll()`	Читает весь оставшийся ввод и возвращает его как строку
`stdio.readAllInts()`	Читает все оставшиеся лексемы и возвращает их как массив целых чисел
`stdio.readAllFloats()`	Читает все оставшиеся лексемы и возвращает их как массив `float`
`stdio.readAllBools()`	Читает все оставшиеся лексемы и возвращает их как массив булевских переменных
`stdio.readAllStrings()`	Читает все оставшиеся лексемы и возвращает их как массив строк
`stdio.readAllLines()`	Читает все оставшиеся строки и возвращает их как массив строк
	Функции записи на стандартный вывод
`stdio.write(x)`	Записывает `x` на стандартный вывод
`stdio.writeln(x)`	Записывает `x` и новую строку на стандартный вывод (стандартно `x` — пустая строка)
`stdio.writef(fmt, arg1, ...)`	Записывает аргументы `arg1...` на стандартный вывод в соответствии со строкой формата `fmt`

Примечание 1. Лексема — это максимальная последовательность символов без отступов.
Примечание 2. Перед чтением лексемы все предваряющие отступы отбрасываются.
Примечание 3. Каждая читающая ввод функция передает ошибку времени выполнения, если не может прочитать следующее значение, или если последующий ввод отсутствует, или если ввод не соответствует ожидаемому типу.

Наш модуль `stddraw`

Вызов функции	Описание
	Базовые функции рисования
`stddraw.line(x0, y0, x1, y1)`	Рисует линию от `(x0, y0)` до `(x1, y1)`
`stddraw.point(x, y)`	Рисует точку в `(x, y)`
`stddraw.show()`	Отображает рисунок в окне стандартного графического устройства (и ожидает, пока его не закроет пользователь)

Вызов функции	Описание
	Функции установки параметров рисунка
`stddraw.setCanvasSize(w, h)`	Устанавливает размер холста `w` на `h` пикселей (стандартно `w` и `h` равны `512`)
`stddraw.setXscale(x0, x1)`	Устанавливает диапазон x холста в `(x0, x1)` (стандартно значение `x0` равно 0, а `x0` — 1)
`stddraw.setYscale(y0, y1)`	Устанавливает диапазон y холста в `(y0, y1)` (стандартно значение `y0` равно 0, а `y1` — 1)
`stddraw.setPenRadius(r)`	Устанавливает радиус пера `r` (стандартное значение `0.005`)

Примечание. При нулевом радиусе пера, точки и строки имеют минимально возможный размер.

	Функции рисования форм
`stddraw.circle(x, y, r)`	Рисует круг радиусом `r` с центром в `(x, y)`
`stddraw.square(x, y, r)`	Рисует квадрат 2`r` на 2`r` с центром в `(x, y)`
`stddraw.rectangle(x, y, w, h)`	Рисует прямоугольник `w` на `h` с левым нижним углом в `(x, y)`
`stddraw.polygon(x, y)`	Рисует многоугольник, соединяющий точки `(x[i], y[i])`

Примечание. Им соответствуют функции `filledCircle()`, `filledSquare()`, `filledRectangle()` и `filledPolygon()`, рисующие заполненные формы, а не контуры.

	Функции рисования текста
`stddraw.text(x, y, s)`	Рисует строку `s` с центром в `(x, y)`
`stddraw.setPenColor(color)`	Устанавливает цвет пера `color` (стандартно `stddraw.BLACK`)
`stddraw.setFontFamily(font)`	Устанавливает семейство шрифтов `font` (стандартно `'Helvetica'`)
`stddraw.setFontSize(size)`	Устанавливает размер шрифта `size` (стандартно `12`)
	Функции анимации
`stddraw.clear(color)`	Очищает фон холста (заполняя цветом `color`)
`stddraw.show(t)`	Отображает рисунок в окне стандартного графического устройства и ожидает `t` миллисекунд

Наш модуль `stdaudio`

Вызов функции	Описание
`stdaudio.playFile(filename)`	Проигрывает все звуковые выборки в файле `filename.wav`
`stdaudio.playSamples(a)`	Проигрывает все звуковые выборки в массиве `a[]` типа `float`

Вызов функции	Описание
`stdaudio.playSample(x)`	Проигрывает звуковую выборку в переменной `x` типа `float`
`stdaudio.save(filename, a)`	Сохраняет все звуковые выборки из массива `a[]` типа `float` в файле `filename.wav`
`stdaudio.read(filename)`	Читает все звуковые выборки из файла `filename.wav` и возвращает как массив типа `float`
`stdaudio.wait()`	Ожидает завершения проигрывания текущего звука (последний вызов в каждой программе `stdaudio`)

Наш модуль `stdarray`

Вызов функции	Описание
`create1D(n, val)`	Массив длиной `n`, каждый элемент которого инициализирован значением `val`
`create2D(m, n, val)`	Массив размером `m` на `n`, каждый элемент, которого инициализирован значением `val`
`readInt1D()`	Массив целых чисел, прочитанных со стандартного ввода
`readInt2D()`	Двумерный массив целых чисел, прочитанных со стандартного ввода
`readFloat1D()`	Массив вещественных чисел, прочитанных со стандартного ввода
`readFloat2D()`	Двумерный массив вещественных чисел, прочитанных со стандартного ввода
`readBool1D()`	Массив логических значений, прочитанных со стандартного ввода
`readBool2D()`	Двумерный массив логических значений, прочитанных со стандартного ввода
`write1D(a)`	Вывод массива `a[]` на стандартное устройство вывода
`write2D(a)`	Вывод двумерного массива `a[]` на стандартное устройство вывода

Примечание 1. Формат 1D — это целое число `n`, сопровождаемое `n` элементами. Формат 2D — два целых числа `m` и `n`, сопровождаемых `m` × `n` элементами в порядном порядке.

Примечание 2. Логические переменные выводятся как `0` и `1`, а не `False` и `True`.

Наш модуль `stdrandom`

Вызов функции	Описание
`uniformInt(lo, hi)`	Однородно случайное целое число в диапазоне [`lo`, `hi`)
`uniformFloat(lo, hi)`	Однородно случайное число типа `float` в диапазоне [`lo`, `hi`)
`bernoulli(p)`	`True` с вероятностью `p` (стандартное значение `p` — `0.5`)
`binomial(n, p)`	Количество орлов при `n` бросках монеты с вероятностью орлов `p` (стандартное значение `p` — `0.5`)
`gaussian(mu, sigma)`	Нормальное, среднее `mu`, среднеквадратичное отклонение `sigma` (стандартное значение `mu` — `0.0`, `sigma` — `1.0`)
`discrete(a)`	`i` с вероятностью, пропорциональной `a[i]`
`shuffle(a)`	Случайная перетасовка массива `a[]`

Наш модуль `stdstats`

Вызов функции	Описание
`mean(a)`	Среднее значение в числовом массиве `a[]`
`var(a)`	Выборочная дисперсия значений в числовом массиве `a[]`
`stddev(a)`	Выборочное среднеквадратичное отклонение значений в числовом массиве `a[]`
`median(a)`	Медиана значений в числовом массиве `a[]`
`plotPoints(a)`	Вывод графика значений в числовом массиве `a[]`
`plotLines(a)`	Вывод линейного графика значений в числовом массиве `a[]`
`plotBars(a)`	Вывод диаграммы значений в числовом массиве `a[]`

Встроенный тип данных Python `str` (часть API)

Операция	Описание
`len(s)`	Длина `s`
`s + t`	Новая строка, полученная в результате конкатенации `s` и `t`
`s += t`	Присвоение `s` новой строки, являющейся конкатенацией `s` и `t`
`s[i]`	`i`-й символ строки `s` (тоже строка)
`s[i:j]`	`i`-й символ, через `(j-1)` символов, строки `s` (стандартные значения `i` — `0`; `j` — `len(s)`)
`s < t`	`s` меньше, чем `t`?
`s <= t`	`s` меньше или равна `t`?
`s == t`	`s` равна `t`?
`s != t`	`s` не равна `t`?
`s >= t`	`s` больше или равна `t`?
`s > t`	`s` больше `t`?
`s in t`	`s` подстрока `t`?
`s not in t`	`s` не подстрока `t`?
`s.count(t)`	Количество вхождений `t` в `s`
`s.find(t, start)`	Первый индекс в строке `s`, где присутствует `t` (`-1`, если не найдено), начиная со `start` (стандартно `0`)
`s.upper()`	Замена строчных букв прописными
`s.lower()`	Замена прописных букв строчными
`s.startswith(t)`	`s` начинается с `t`?
`s.endswith(t)`	`s` заканчивается `t`?
`s.strip()`	Удаляет предваряющие и концевые пробелы
`s.replace(old, new)`	Замена вхождений `old` на `new`
`s.split(delimiter)`	Массив подстрок `s`, разделенных `delimiter` (стандартно отступ)
`delimiter.join(a)`	Конкатенация строк в `a[]`, разделенных `delimiter`

Наш тип данных `Color` (`color.py`)

Операция	Описание
`Color(r, g, b)`	Новый цвет с компонентами красного, зеленого и синего `r`, `g` и `b` (целые числа от 0 до 255)
`c.getRed()`	Красный компонент `c`
`c.getGreen()`	Зеленый компонент `c`
`c.getBlue()`	Синий компонент `c`
`str(c)`	`'(R, G, B)'` (строковое представление `c`)

Наш тип данных `Picture` (`picture.py`)

Операция	Описание
`Picture(w, h)`	Новый массив `w` на `h` пикселей, первоначально полностью черный
`Picture(filename)`	Новое изображение, инициализированное из файла `filename`
`pic.save(filename)`	Сохраняет `pic` в `filename`
`pic.width()`	Ширина `pic`
`pic.height()`	Высота `pic`
`pic.get(col, row)`	Цвет пикселя `(col, row)` в `pic`
`pic.set(col, row, c)`	Установка цвета `c` пикселя `(col, row)` в `pic`

Примечание. Имя файла должно заканчиваться на `.png` или `.jpg` (формат файла).

Отображение объекта `Picture`

Операция	Описание
`stddraw.picture(pic, x, y)`	Отображает `pic` на `stddraw`, с центром `(x, y)`

Примечание. Стандартно `x` и `y` в центре холста графического устройства.

Наш тип данных `InStream` (`instream.py`)

Операция	Описание
`InStream(filename)`	Новый поток ввода, инициализированный из файла `filename` (без аргумента ввода со стандартного устройства ввода)
	Методы чтения лексем со стандартного ввода
`s.isEmpty()`	`s` пуста? (она состоит исключительно из отступов?)
`s.readInt()`	Читает лексему из `s`, преобразует в целое число и возвращает
`s.readFloat()`	Читает лексему из `s`, преобразует в тип `float` и возвращает
`s.readBool()`	Читает лексему из `s`, преобразует в тип `bool` и возвращает
`s.readString()`	Читает лексему из `s`, преобразует в строку и возвращает
	Методы чтения строки со стандартного ввода
`s.hasNextLine()`	Есть ли в `s` другая строка?
`s.readLine()`	Читает следующую строку из `s` и возвращает ее как строку

Примечание 1. Лексема — это максимальная последовательность символов без отступов.
Примечание 2. Поведение, как у стандартного ввода; методы `readAll()` также поддерживаются.

Наш тип данных `OutStream` (`outstream.py`)

Операция	Описание
`OutStream(filename)`	Новый поток вывода, пишущий в файл `filename` (без аргумента вывод на стандартное устройство вывода)
`out.write(x)`	Пишет `x` в `out`
`out.writeln(x)`	Пишет в `out` значение `x` и новую строку (стандартно `x` пустая строка)
`out.writef(fmt, arg1, ...)`	Пишет аргументы `arg1`, ... в `out` как определено строкой формата `fmt`

Встроенный тип данных Python `int` (часть API)

Операция	Описание
`x + y`	Сумма `x` и `y`
`x - y`	Разница `x` и `y`
`x * y`	Произведение `x` и `y`
`x / y`	Частное `x` и `y`
`x // y`	Неполное частное `x` и `y`
`x % y`	Остаток от деления `x` на `y`
`x ** y`	`x` в степени `y`
`-x`	Инверсия `x`
`+x`	Неизменный `x`
`x < y`	`x` меньше `y`?
`x <= y`	`x` меньше или равен `y`?
`x == y`	`x` равен `y`?
`x != y`	`x` не равен `y`?
`x >= y`	`x` больше или равен `y`?
`x > y`	`x` больше `y`?
`abs(x)`	Абсолютное значение `x`
`min(x, y, ...)`	Минимум `x`, `y`, ...
`max(x, y, ...)`	Максимум `x`, `y`, ...

Примечание: В Python 3 оператор частного (`/`) осуществляет вещественное деление, а в Python 2 — целочисленное. В этой книге мы не используем оператор частного с двумя операндами типа `int`.

Встроенный тип данных Python `bool` (часть API)

Операция	Описание
`x and y`	`True` если `x` и `y` `True`, и `False` в противном случае
`x or y`	`True` если `x` или `y` `True`, и `False` в противном случае
`not x`	`True` если `x` `False`, и `False` в противном случае

Встроенный тип данных Python `float` (часть API)

Операция	Описание
`x + y`	Сумма `x` и `y`
`x - y`	Разница `x` и `y`
`x * y`	Произведение `x` и `y`
`x / y`	Частное `x` и `y`
`x // y`	Неполное частное `x` и `y`
`x % y`	Остаток от деления `x` на `y`
`x ** y`	`x` в степени `y`
`-x`	Инверсия `x`
`+x`	Неизменный `x`
`x < y`	`x` меньше `y`?
`x <= y`	`x` меньше или равен `y`?
`x == y`	`x` равен `y`?
`x != y`	`x` не равен `y`?
`x >= y`	`x` больше или равен `y`?
`x > y`	`x` больше `y`?
`abs(x)`	Абсолютное значение `x`
`min(x, y, ...)`	Минимум `x`, `y`, ...
`max(x, y, ...)`	Максимум `x`, `y`, ...

Примечание: В этой книге мы не используем операторы неполного частного (`//`) и остатка деления (`%`) с двумя операндами типа `float`.

Встроенный тип данных Python `tuple` (часть API)

Операция	Описание
`len(a)`	Длина `a`
`a[i]`	`i`-тый элемент `a`
`for v in a:`	Перебор элементов `a`
`v in a`	`True` если `a` содержится в `v`, и `False` в противном случае

Встроенный тип данных Python `list` (часть API)

Операция	Описание
	Операции постоянного времени
`len(a)`	Длина `a`
`a[i]`	`i`-тый элемент `a`
`a[i] = v`	Заменяет `i`-тый элемент `a` элементом `v`
`a += [v]`	Добавляет элемент `v` в конец `a`
`a.pop()`	Удаляет элемент `a[ len(a)-1]` из списка и возвращает его

Операция	Описание
	Операции линейного времени
`a + b`	Конкатенация `a` и `b`
`a[i:j]`	`[a[i], a[i+1], ..., a[j-1]]`
`a[i:j] = b`	`a[i] = b[0], a[i+1] = b[1], ...`
`v in a`	`True` если `a` содержит `v`, и `False` в противном случае
`v not in a`	`False` если `a` содержит `v`, и `False` в противном случае
`for v in a:`	Перебор элементов `a`
`del a[i]`	Удаляет элемент `a[i]` из списка
`a.pop(i)`	Удаляет элемент `a[i]` из списка и возвращает его
`a.insert(i, v)`	Вставляет элемент `v` в `a` перед `a[i]`
`a.index(v)`	Индекс первого вхождения элемента `v` в списке `a`
`a.reverse()`	Меняет порядок элементов в `a` на обратный
	Операции линейно-логарифмического времени
`a.sort()`	Меняет порядок элементов в `a` на возрастающий

Примечание 1: `a += [v]` и `a.pop()` являются "амортизированными" операциями постоянного времени.

Примечание 2: `del a[i]`, `pop(i)` и `insert(i, v)` являются "амортизированными" операциями постоянного времени если `i` близко к `len(a)`.

Встроенный тип данных Python `dict` (часть API)

Операция	Описание
	Операции постоянного времени
`dict()`	Новый пустой словарь
`st[key] = val`	Ассоциирует `key` с `val` в `st`
`st[key]`	Ассоциируемое с ключом `key` значение в наборе `st` (передает `KeyError` если в `st` такого ключа нет)
`st.get(key, x)`	`st[key]` если `key` находится в `st`; в противном случае `x` (стандартное значение `x` —`None`)
`key in st`	Находится ли `key` в `st`?
`len(st)`	Количество пар "ключ–значение" в `st`
`del st[key]`	Удаляет `key` (и ассоциированное с ним значение) из `st`
	Операции линейного времени
`for key in st:`	Перебирает `st` по ключам

Примечание: Ключи должны быть хешируемы.

Встроенный тип данных Python set (часть API)

Операция	Описание
	Операции постоянного времени
`set()`	Новый пустой набор
`s.add(item)`	Добавляет `item` в `s` (если еще нет в наборе)
`item in s`	Находится ли `item` в `s`?
`len(s)`	Количество элементов в `s`
`s.remove(item)`	Удаляет `item` из `s`
	Операции линейного времени
`for item in s:`	Перебор элементов в `s`
`s.intersection(t)`	Пересечение `s` и `t`
`s.union(t)`	Объединение `s` и `t`

Примечание: Элементы должны быть хешируемы.

Встроенные функции Python (часть API)

	Встроенные функции для объектов
`help(x)`	Документация объекта `x`
`type(x)`	Тип объекта `x`
`id(x)`	Идентификатор (адрес в памяти) объекта `x`
`hash(x)`	Значение хеш-функции объекта `x`
	Встроенные функции для преобразования типов
`int(x)`	Преобразование объекта `x` в целое число
`float(x)`	Преобразование объекта `x` в вещественное число
`str(x)`	Преобразование объекта `x` в строку
`round(x)`	Ближайшее целое число к числу `x`
	Встроенные функции, задействующие итераторы
`min(a)`	Минимальный элемент в итерируемом `a`
`max(a)`	Максимальный элемент в итерируемом `a`
`sum(a)`	Сумма элементов в итерируемом `a`
`sorted(a)`	Новое итерируемое, содержащее элементы `a` в отсортированном порядке
`reversed(a)`	Новое итерируемое, содержащее элементы `a` в обратном порядке
`tuple(a)`	Новый кортеж `tuple`, созданный из элементов `a`
`list(a)`	Новый список `list`, созданный из элементов `a`
`set(a)`	Новый набор `set`, созданный из элементов `a`
`range(i, j)`	Новое итерируемое, содержащее целые числа от `i` до `j-1` (стандартное значение `i` — `0`)
`iter(a)`	Новый итератор по элементам в итерируемом `a`

Примечание 1: Встроенные типы Python `str`, `list`, `tuple`, `dict` и `set` является итерируемыми.
Примечание 2: Функции `min()`, `max()` и `sorted()` подразумевают, что элементы сравнимы.
Примечание 3: Функция `set()` подразумевает, что элементы хешируемы.

A

Abstraction, 345
Adjacency-matrix, 687
Alias, 115; 228
Aliasing bug, 451
Amortized analysis, 520
API, 51; 150; 255
Application Programming Interface, 51; 255
Argument, 51; 213; 217
Arithmetic expression, 588
Array, 110
Assertion, 477
Assignment statement, 34
Associative array, 620
Automatic promotion, 55
Average
  magnitude, 188
  power, 188

B

Background canvas, 165
Barnsley fern, 264
Base
  case, 287
  class, 468
BFS, 685
Big-O notation, 531
Binary
  reflected Gray code, 297
  search, 543
  search tree, 619
  Search Tree, 635
Binding, 35
Bipartite graph, 677
Bisection search, 547
Block, 71
Booksite module, 29
Breadth-first search, 680
Breadth-First Search, 685
Brownian
  bridge, 302
  motion, 411
BST, 619; 635
Bytecode, 279

C

Call by
  object reference, 227
  value, 227
Capacity, 519
Character, 38
Checksum, 106; 240
Circular shift, 386
Class, 393; 394
Client, 248; 254
Coercion, 55
Collection, 138; 576
Column-major, 131
Comma-separated-value, 378
Compile-time error, 23
Complex number, 414
Conditional, 70
Constant variable, 33
Constructor, 354; 395
Control flow, 70
Conversion specification, 151
Cosine similarity, 474
Counter, 80
Coupon collector, 125

D

Data
  abstraction, 345
  mining, 469
  structure, 110; 499
  type, 31
Data-driven code, 160; 194
Default value, 51; 219
Defensive copy, 453
Degree, 668
  of separation, 681
Denial-of-service, 516
Dependency graph, 275
Depth-first search, 328; 685
Deque, 613
Derivative, 467
Derived class, 468
Directed percolation, 334
Discrete probability distribution, 195
DNS, 624
DoS, 516
Dot product, 111
Double buffering, 165
Doubling hypothesis, 502
Duck typing, 459

E

Edge, 668
Element, 110
Encapsulation, 444

Escape sequence, 38
Event, 600
Event-based simulation, 600
Exception, 476
Executing, 21
Existence problem, 550
Explicit type conversion, 54
Expression, 33
Extensible module, 468
Extension module, 29

F

Factorial, 286
Fade effect, 371
Fibonacci sequence, 306
FIFO, 576
FIFO queue, 593
Filter, 162
First-in first-out, 576
Fixed-length array, 117
Format string, 151
Fractal dimension, 302
Function, 50
  abstraction, 345
  body, 217
  name, 217
  overloading, 237
Function-call tree, 292

G

Garbage collection, 381; 527
Gcd, 105
Genomics, 352
Global variable, 219
Graph, 668
Gray codes, 296
Greatest common divisor, 105

H

Half-open interval, 543
Hamming distance, 313
Harmonic numbers, 85
Hash
  code, 464; 632
  function, 464; 632
  table, 632
Hashing, 464; 472; 619
Heap, 527
Histogram, 403
H-tree, 300
Hurst exponent, 302
H-дерево, 300

I

Identifier, 32
Identity, 34
Immutable, 228; 451
Implementation, 254
Indegree, 615
Indentation, 27
Informal trace, 34
Information content, 390
Initialization, 74; 130
Inner loop, 505
Input/output, 147
Insertion sort, 552
Instance variable, 395
Invariant, 477
I/O, 147
Iterable, 646
Iteration, 114
Iterator, 621; 646

J

Josephus problem, 613

K

Kevin Bacon game, 682
Keyword, 32
K-gram, 471
K-ring graph, 690
K-грамм, 471
K-регулярный граф, 690

L

Last-in first-out, 576
Leading term, 504
Leaf, 638
Lexicographic order, 58
Library, 255
LIFO, 576
Link, 193; 583
Linked
  list, 581
  structure, 499
Literal, 32
Little's law, 600
Local variable, 217
Logarithmic spiral, 410
Logical operator, 47
Loop, 70; 74
Loop and a half, 94

M

Mandelbrot set, 417
Mathematical induction, 285; 288
Matrix, 129
Memoization, 307
Memory leak, 381
Mergesort, 559
Method, 347; 397
Miraculous spiral, 411
Mixing, 198
Modular programming, 211; 273
Module, 211; 248
Monte Carlo simulation, 316
Moore's law, 513

Mutability, 114
Mutable, 227; 451

N

Neighbor, 668
Node, 581
Numeric tower, 469
N тел, 486

O

Object, 34; 345
  reference, 35
Object-level trace, 35
Object-oriented programming, 345
One-dimensional array, 110
Operand, 33
Operator, 32
  overloading, 461
  precedence, 33
Order of growth, 506
Orphan, 380

P

Parallel arrays, 424
Parameter variable, 217
Path, 671
Percolation, 317
  probability, 326
Permutation, 121
Phase transition, 333
Piping, 161
Plasma cloud, 304
Polymorphism, 221; 350
Pop, 578
Postcondition, 477
Postfix, 590
Precondition, 477
Prime, 92
Private
  function, 255
  method, 399
Problem size, 501
Pure function, 220
Push, 578
Python, 20

Q

Queue, 576; 593

R

Ragged array, 133
Random surfer, 193
Rank, 197; 648
Recurrence relation, 295
Recursion, 211; 285
Red-black tree, 645
Reduction step, 288
Reference, 358; 583
Resizing array, 518
Return
  statement, 217
  value, 53; 213; 217
Reverse Polish notation, 590
Root, 638
Row-major, 131
Run-time error, 23

S

Sample variance, 268
Sampling, 367
Scope, 218
Screen scraping, 376
Script, 251
Self-avoiding random walk, 134
Sequence, 110
Set, 649
Shortest path, 679
Shortest-paths tree, 684
Side effect, 220; 477
Sierpinski triangle, 264
Signature, 217
Six degrees of separation, 667
Size, 519
Sketch, 470
Small-world phenomenon, 667
Spatial vector, 454
Special method, 395
Specification problem, 442
Spider trap, 198
Spreadsheet, 131
Stack, 527; 576
State, 138; 405
Stress test, 261
String literal, 27
Stub, 319
Subclass, 468
Subtree, 638
Superclass, 468
Symbol table, 619
  of sets, 673
Syntax error, 27

T

Template, 70
Terminal window, 148
Tilde notation, 504
Token, 374
Total order, 465
Trace, 34
Transition matrix, 195
Tree node, 292
Tree traversal, 645
Truth-table, 47
Turtle graphics, 405
Two-dimensional array, 110; 129
Type, 34

U

Unit testing, 260

V

Value, 35
Variable, 32
Vector, 111; 453
Vertex, 668
Vertically percolates, 321
Vertical percolation, 319
Volatility, 302

W

Wide interface, 442

Z

Zipf's law, 566

А

Абстракция, 345
  данных, 393
  функциональная, 345
Автоматическое преобразование, 55
Адрес, 113
Алгоритм
  Дейкстры, 588; 688
  Евклида, 291
  поиска кратчайшего пути, 680
Амортизационный анализ, 520
Анализ экранных данных, 376
Аргумент, 51; 213; 217
Аргументы командной строки, 148
Арифметическое выражение, 588
Ассоциативный массив, 620

Б

Базовый класс, 468
Байт, 523
Библиотека, 255
Бинарное дерево поиска, 619; 635
Бинарный поиск, 543; 550
Биномиальное распределение, 146
Биномиальный коэффициент, 146
Бит, 59; 523
Блок, 71
Броуновский
  мост, 302
  остров, 315
Броуновское движение, 411
Булева логика, 48

В

Ввод и вывод, 147
Ведущий член, 504
Вектор, 111; 453
  единичный, 454
  пространственный, 454
  рядов, 133
  столбцов, 133
Вероятность
  просачивания, 326
Вершина, 668
Виртуальная машина Python, 442; 589
Внешний цикл, 84
Внутренний цикл, 84; 505
Возвращаемое значение, 53; 213; 217
Волатильность, 302
Выборка, 367
Выборочная дисперсия, 268
Выборочное среднеквадратичное
  отклонение, 268
Вызов
  метода, 347
  по значению, 227
  по ссылке на объект, 227
Выполнение, 21
Выравнивание, 27
Выражение, 33
Вычисление по сокращенной схеме, 49

Г

Гарантии производительности, 516
Гармонические числа, 85
Геномика, 352
Гипотеза удвоения, 502
Гистограмма, 403
Глобальная переменная, 219
Голливудское число, 704
Граф, 668
  Боллобаша–Чунга, 707
  Ваттса–Строгаца, 706
  зависимостей, 275
  Клейберга, 707
  \"тесного мира\", 688
График
  контурный, 438
  Тьюки, 283; 436

Д

Двойная буферизация, 165
Двудольный граф, 677
Двухсторонняя очередь, 613
Дерево
  вызова функции, 292
  кратчайших путей, 684
Дескриптор, 423
Диаметр графа, 671; 704
Динамическое программирование, 307
Директива else, 72
Дискретное распределение вероятности, 195
Длина кратчайшего пути, 683
Дополнение Уотсона–Крика, 386
Допустимая ошибка, 333
Допустимый промежуток, 333

Е

Евклидово расстояние, 140; 474
Емкость, 519

З

Заглушка, 319
Задача
  Иосифа Флавия, 613
  Коллатца, 314
  моделирования N тел, 486
  нидерландского флага, 574
  о разорении игрока, 90
Закон
  всемирного тяготения, 489
  Гордона Мура, 513
  Кулона, 353
  Литтла, 600
  Ципфа, 566
Закрытая функция, 255
Запись с использованием тильды, 504
Защитная копия, 453
Значение, 35
Значения, разделяемые запятыми, 378

И

Игра Кевина Бэйкона, 682
Идентификатор, 32; 34
Извлечение, 578
Изменчивость, 114; 227
Имя функции, 217
Инвариант, 477
Индексация, 130
Индуцированный подграф, 699
Инициализация, 74; 130
  переменной, 38
Инкапсуляция, 227; 444
Интеграл Римана, 467
Интерпретатор, 21; 589
Интерфейс прикладных программ, 51; 150; 255; 394
Информационное содержимое, 390
Исключающий фильтр, 552
Исключение, 476
Итератор, 621; 646
Итерация, 114
Итерируемый объект, 646

К

Кватернион, 435
Кеширование, 514; 524
Класс, 393; 394
Клиент, 248; 254
  проверки, 401
Ключевое слово, 32
  class, 394
Код, 20
  виртуальной машины, 279
  Грея, 296
    рефлексивный бинарный, 297
  преобразования, 151
  управляемый данными, 160
Кодон, 352
Код, управляемый данными, 194
Коллекционер купонов, 125
Коллекция, 138; 576
Комментарий, 22
Компилятор, 21; 589
Комплексное число, 414
Комплексно сопряженная величина, 460
Комплементарный палиндром Уотсона–Крика, 386
Компонента связности графа, 702
Конец файла, 156
Конечный
  случай, 287
  узел, 638
Конкатенация, 39
Конкордация, 657
Константа
  Кулона, 354
  Эйлера, 242
Константная переменная, 33
Конструктор, 354; 380; 395
  типа данных, 395
Контрольная сумма, 106; 240
Контроль потока, 70; 213
Корень, 638
Кортеж, 457
Косинусный коэффициент Отиаи, 474
Коэффициент кластеризации графа, 689
Красно-черное дерево, 645
Кратчайший путь, 679
Кривая
  дракона, 435
  Коха, 408
  Пеано, 435
Кусочно-линейная аппроксимация, 170

Л

Лексема, 374
Лексикографический порядок, 58; 570
Линейная алгебра, 454
Лист, 638
Литерал, 32
Логарифмическая спираль, 410
Логическая матрица, 318
Логическое значение, 47
Локальная
  кластеризация, 689
  переменная, 217

М

Массив, 110; 525
  двумерный, 110; 129; 525
  многомерный, 134
  объектов, 525
  одномерный, 110
  параллельный, 424
  переменного размера, 518; 578

с переменной длиной рядов, 133
фиксированной длины, 117
Масштабирование, 366
Математическая индукция, 285; 288
Матрица, 129
Адамара, 144; 339
переходов, 195
смежности, 687
Машина Тьюринга, 182
Мемоизация, 307
Метод, 347; 397
закрытый, 399
Марсальи, 105
Ньютона, 85
половинного деления, 547
прямоугольников, 467
смещения средней точки, 302
Множество
Жюлиа, 438
Мандельброта, 417
Моделирование на основании событий, 600
Модель
Монте-Карло, 316
Эрдёш-Реньи, 690
Модуль, 211; 248; 445
Python
numpy, 117
random, 121
stdarray, 118
stdaudio, 179
stddraw, 164
stdio, 22; 118; 153
книжный, 29
расширения, 29; 468
Модульное
программирование, 211; 247; 252; 273
тестирование, 260
Монохромная яркость, 361

Н

Набор, 649
Наибольший общий делитель, 105; 289
Наследование, 468
Неизменность, 228; 451
Необязательный аргумент, 219
Неполный цикл, 94
НОД, 289
Нотация "большого О", 531

О

Область видимости, 218
Обратная польская нотация, 590
Обход дерева, 645
Объект, 34; 345; 526
значение, 358
идентификатор, 358
поведение, 358
состояние, 358
тип, 358
Объектная ссылка, 35
Объектно-ориентированное программирование, 345
Окно терминала, 148
Операнд, 33
Оператор, 21; 32
[], 350
[\
], 350
/, 350
break, 94
for, 79
if, 70
import, 355
while, 74
выхода, 217
логический, 47
парный, 33
приоритет, 33
присвоения, 34
смешанного типа, 48
сравнения, 48
Определение
переменной, 38
функции, 217
Ориентированный граф, 704
Отказ в обслуживании, 516
Очередь, 576; 593
FIFO, 593
Очередь M/M/1, 599
Ошибка
времени выполнения, 23
времени компиляции, 23
псевдонимов, 451
синтаксическая, 27

П

Папоротник Барнсли, 264
Параметрическая переменная, 217
self, 397
Паучья ловушка, 198
Первым пришел, первым вышел, 576
Перебор связанного списка, 586
Перегрузка
оператора, 461
функций, 237
Передача исключения, 476
Перекрестное произведение, 481
Переменная, 32
глобальная, 427
класса, 428
константная, 33
экземпляра, 395
Перенаправление
в файл, 158
из файла, 160
Переполнение буфера, 113
Перестановка, 121
Пересылка, 161

Плазменные облака, 304
Побочный эффект, 220; 477
Поддерево, 638
Подмости, 319
Поиск
  вглубь, 328; 685
  в ширину, 680; 685
  данных, 469
Показатель Херста, 302
Полиморфизм, 221; 350; 458
Полный порядок, 465
Полуоткрытый интервал, 543
Полустепень захода, 615
Полярное представление, 445
Полярный угол, 407; 445
Помещение, 578
Порядок
  левосторонний, 33
  порядный, 131
  постолбцовый, 131
  правосторонний, 33
Порядок роста, 506
  квадратичный, 510
  кубический, 510
  линейно-логарифмический, 509
  линейный, 509
  логарифмический, 508
  постоянный, 508
  экспоненциальный, 510
Последним пришел, первым вышел, 576
Последовательность, 110
  Фибоначчи, 306
Последовательный поиск, 550
Постусловие, 477
Постфикс, 590
Поток, 153
Правило 90 на 10, 193
Предусловие, 477
Преобразование типов
  неявное, 55
  явное, 54
Приведение типа данных, 55
Привязка, 35
Принцип суперпозиции, 491
Приоритет, 33
Проблема
  спецификации, 442; 598
  существования, 550
  хрупкости базового класса, 469
Проектирование по контракту, 477
Производная, 467
Производный класс, 468
Проницаемость вертикальная, 321
Просачивание, 317
  вертикальное, 319
  в объеме, 342
  в треугольной решетке, 342
  направленное, 334
  по границе, 341
Простое число, 127
Простой множитель, 92
Псевдоним, 115; 228
Путь, 671

P

Разделение последовательности, 116
Размер, 519
Ранг, 197; 648
Распределяемая память, 527
Расстояние Хемминга, 313
Реализация, 254
Ребро, 668
Рекуррентное соотношение, 295
Рекурсия, 211; 285
Решето Эратосфена, 127

C

Сайт книги, 20
Сбор мусора, 381; 527
Связанная структура, 499
Связанный список, 581
Сигнатура, 217
Символ, 38
Сирота, 380
Система Python, 20
Система счисления
  двоичная, 59
  десятичная, 59
Скалярное произведение, 111
Скаффолдинг, 319
Словарь, 648
Сложность задачи, 501
Случайная навигация, 193
Случайное блуждание без самопересечений, 134
Случайный граф, 690
Смешение, 198
Событие, 600
Сортировка
  вставкой, 552
  с объединением, 559
Сосед, 668
Состояние, 138; 405
Специальное значение None, 220
Специальный метод, 395
Спецификация преобразования, 151
Список Python, 578
Средневзвешенное значение, 141
Среднее
  гармоническое, 186
  геометрическое, 186
  квадратов значений, 188
Среднеквадратичное отклонение, 186
Средняя точка, 188
Ссылка, 193; 358; 583
Стандартное
  аудиоустройство, 149
  графическое устройство, 149

значение аргумента, 219
устройство ввода, 149
устройство вывода, 148
Стек, 527; 576; 577; 593
Степень
вершины, 668
разделения, 681
Стресс-проверка, 261
Строка, 38; 525
Строка формата, 151
Строковый литерал, 27
Структура данных, 110; 499
Суперпозиция, 232
Сценарий, 251
Счетчик, 80

Т

Таблица, 131
идентификаторов, 619
идентификаторов наборов, 673
истинности, 47
Тело
метода, 397
функции, 217
цикла, 74
Теория
очередей, 598
шести рукопожатий, 667
Тип, 34
Тип NoneType, 237
Тип данных, 31
bool, 47; 524
float, 43; 524
int, 41; 523
list, 117; 517
str, 38; 525
string, 520
tuple, 457
изменяемый, 451
неизменяемый, 451
Типизация ввода, 154
Точечный оператор, 347
Точность, 151
Транспозиция, 141
Трассировка, 34
объектного уровня, 35
свободная, 34
Треугольник
Паскаля, 146
Серпинского, 264

У

Удивительная спираль, 411
Узел, 581
древа, 292
Умножение
векторно-матричное, 133
матрично-векторное, 133
Управление памятью, 380
Управляющая последовательность, 38
Условие продолжения цикла, 74
Условное выражение, 70
Утверждение, 477
Утечка памяти, 381
Утиная типизация, 459

Ф

Фазовый переход, 333
Файловая система, 624
Факториал, 286
Феномен "тесного мира", 667; 689
Фильтр, 162
Фоновый холст, 165
Форматированный вывод, 151
Фрактал, 300
Фрактальная размерность, 302
Фракционное броуновское движение, 302
Функция, 50; 212
высшего порядка, 467
стандартное значение, 51

Х

Ханойская башня, 291
Хеширование, 464; 472; 619
Хеш-код, 464; 632
Хеш-таблица, 632
Хеш-функция, 464; 471; 632

Ц

Целое число, 41
Цепь Маркова, 198
Цикл, 70; 74
Циклический сдвиг, 386

Ч

Частичный порядок, 480
Частота Найквиста, 185
Черепашья графика, 405
Число Бэйкона, 682
Числовая башня, 469
Чистая функция, 220

Ш

Шаблон, 70
Ширина поля, 151
Широкий интерфейс, 442
Шифр Камасутры, 388

Э

Эксцентриситет вершины, 704
Электрический потенциал, 353
Элемент, 110
Энтропия Шэннона, 389
Эскиз, 470
Этап
индукции, 288
редукции, 288
Эффективная яркость, 361
Эффект постепенного изменения, 371

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ PYTHON:

учебный курс

Программирование на языке Python

учебный курс

Любая научная или техническая дисциплина требует навыков программирования. Python — идеальный первый язык программирования, а эта книга — лучшее руководство по его изучению.

Преподаватели Принстонского университета Роберт Седжвик, Кевин Уэйн и Роберт Дондеро написали доступный междисциплинарный учебный курс по программированию на языке Python, рассматривающий важные и реальные случаи его применения, а не абстрактные примеры. Авторы демонстрируют инструментальные средства, необходимые студентам для изучения программирования естественным, нескучным и творческим способом.

Это руководство сосредоточивается на наиболее полезных средствах языка Python и знакомит с программированием на примерах, полезных для каждого студента научных, технических и информационных специальностей.

Особенности книги

- **Базовые элементы программирования:** переменные, операторы присвоения, встроенные типы данных, условные выражения, циклы, массивы, ввод и вывод, включая графику и звук.
- **Функции, модули и библиотеки:** организация программ в компоненты, обеспечивающие независимую отладку, поддержку и многократное использование.
- **Объектно-ориентированное программирование и абстракция данных:** объекты, модули, инкапсуляция и т.д.
- **Алгоритмы и структуры данных:** алгоритмы сортировки и поиска, стеки, очереди и таблицы символов.
- **Все примеры** из области прикладной математики, физики, химии, биологии и информатики совместимы с языком Python версий 2 и 3.

Опираясь на свою обширную преподавательскую практику, авторы завершают каждый раздел списками вопросов и ответов, упражнениями, а зачастую и практическими упражнениями. На сайте `introcs.cs.princeton.edu/python` доступно множество дополнительной информации и вспомогательных материалов, включая исходный код, библиотеки ввода и вывода, решения для некоторых упражнений и многое другое. Этот веб-сайт позволяет использовать собственные компьютеры для преподавания и изучения материала книги.

Роберт Седжвик — профессор информатики в Принстонском университете. Он занимал ведущие позиции в нескольких передовых научно-исследовательских лабораториях, а также в команде Adobe Systems. Роберт в соавторстве с Кевином Уэйном написал книги *Introduction to Programming in Java* (вышла в издательстве Addison-Wesley) и *Алгоритмы на Java, 4-е издание* (пер. с англ., ИД "Вильямс", 2012 г).

Кевин Уэйн — преподаватель информатики в Принстонском университете, где он работает с 1998 года. Он заслуженный педагог (ACM Distinguished Educator), доктор философии в области исследования операций и организации производства Корнелльского университета.

Роберт Дондеро — преподает информатику в Принстонском университете с 2001 года. Он лауреат восьми премий в области технического образования и имеет награды за долгую и безупречную работу. Он также доктор философии в области информатики и технологий Дрексельского университета.

Дизайн обложки: Чути Празерсит (Chuti Prasertsith)

Addison-Wesley

introcs.cs.princeton.edu/python

ДИАЛЕКТИКА

http://www.dialektika.com

ISBN 978-5-9908462-1-0