Студенты и исследователи всех уровней постепенно переходят на язык программирования Python как альтернативу коммерческим программным продуктам.

Этот интенсивный вводный курс позволяет пройти путь от основ до формирования продвинутых навыков, благодаря чему читатели смогут быстро повысить свой уровень профессиональной подготовки. Книга изобилует примерами и решениями, взятыми из реальной инженерной и научной практики.

Изучение начинается с общих концепций программирования, таких как циклы и функции в ядре Python 3, затем рассматриваются библиотеки NumPy, SciPy и Matplotlib для вычислительного программирования и визуализации данных. Обсуждается использование виртуального блокнота Jupyter Notebooks для создания мультимедийных совместно используемых документов для научного анализа. Отдельная глава посвящена анализу данных с использованием библиотеки pandas. В заключительной главе представлены более сложные темы, такие как точность вычислений с применением чисел с плавающей точкой и обеспечение стабильности алгоритмов.

Издание адресовано студентам, ученым, специалистам по работе с данными, которым требуется прочная основа для решения насущных задач с помощью Python.

Кристиан Хилл – физик и специалист в области физической химии, в настоящее время работающий в Интернациональном агентстве по использованию атомной энергии (International Atomic Energy Agency). Обладает более чем 25-летним опытом программирования в области физических наук и программирует на Python 15 лет. В своих исследованиях использует Python для создания, анализа, обработки, управления и визуализации крупных наборов данных в области спектроскопии, физики плазмы и материаловедения.

Интернет-магазин:
www.dmkpress.com

Оптовая продажа:
КТК "Галактика"
books@alians-kniga.ru

CAMBRIDGE UNIVERSITY PRESS

ДМК ИЗДАТЕЛЬСТВО
www.дмк.рф

ISBN 978-5-97060-914-9

Научное программирование на Python

Кристиан Хилл

Научное программирование на Python

ДМК ИЗДАТЕЛЬСТВО

Кристиан Хилл

Научное программирование на Python

Learning Scientific Programming with Python

Second Edition

Christian Hill

CAMBRIDGE
UNIVERSITY PRESS

Научное программирование на Python

Кристиан Хилл

Москва, 2021

УДК 004.94Phyton
ББК 32.972
Х45

Кристиан Хилл
Х45 Научное программирование на Python / пер. с анг. А. В. Снастина. – М.: ДМК Пресс, 2021. – 646 с.: ил.

ISBN 978-5-97060-914-9

Книга начинается с общих концепций программирования, таких как циклы и функции в ядре Python 3, затем рассматриваются библиотеки NumPy, SciPy и Matplotlib для вычислительного программирования и визуализации данных. Обсуждается использование виртуального блокнота Jupyter Notebooks для создания мультимедийных совместно используемых документов для научного анализа. Отдельная глава посвящена анализу данных с использованием библиотеки pandas. В заключительной части представлены более сложные темы, такие как точность вычислений с применением чисел с плавающей точкой и обеспечение стабильности алгоритмов.

Издание адресовано студентам, ученым, специалистам по работе с данными, которым требуется прочная основа для решения насущных задач с помощью Python

УДК 004.94Phyton
ББК 32.972

ISBN (анг.) 978-1108745918
ISBN (рус.) 978-5-97060-914-9

Научное программирование на Python

Изучение основных задач программирования на профессиональном уровне с нуля с реальными научными примерами и решениями, взятыми из научной и инженерной практики. Студенты и исследователи всех уровней постепенно переходят на мощный язык программирования Python как альтернативу коммерческим программным продуктам. Этот интенсивный вводный курс позволяет пройти путь от основ до продвинутых концепций в одной книге, дающий возможность читателям быстро поднимать свой уровень профессиональной подготовки.

Книга начинается с общих концепций программирования, таких как циклы и функции в ядре Python 3, затем рассматриваются библиотеки NumPy, SciPy и Matplotlib для вычислительного программирования и визуализации данных. Обсуждается использование виртуального блокнота Jupyter Notebooks для создания мультимедийных совместно используемых документов для научного анализа. Отдельная глава посвящена анализу данных с использованием библиотеки pandas. В заключительной части представлены более сложные темы, такие как точность вычислений с применением чисел с плавающей точкой и обеспечение стабильности алгоритмов.

Кристиан Хилл (Christian Hill) – физик и специалист в области физической химии, в настоящее время работающий в Интернациональном агентстве по использованию атомной энергии (International Atomic Energy Agency). Он обладает более чем 25-летним опытом программирования в области физических наук и программирует на Python 15 лет. В своих исследованиях Кристиан использует Python для создания, анализа, обработки, управления и визуализации крупных наборов данных в области спектроскопии, физики плазмы и материаловедения.

Оглавление

Предисловие от издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте **www.dmkpress.com**, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу **dmkpress@gmail.com**; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу **http://dmkpress.com/authors/publish_book/** или напишите в издательство по адресу **dmkpress@gmail.com**.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг – возможно, ошибку в основном тексте или программном коде, – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу **dmkpress@gmail.com**, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательство «ДМК Пресс» очень серьезно относится к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу **dmkpress@gmail.com**.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Благодарности

Эмме, Шарлотте и Лоренсу

Множество людей прямо или косвенно помогали написать эту книгу, в частности Джонатан Теннисон (Jonathan Tennyson) из UCL, Лоуренс Ротман (Laurence Rothman) и Айюли Гордон (Iouli Gordon) поддерживали меня во время годичного отпуска для научной работы в Центре астрофизики Гарварда и Смитсоновского института (Harvard-Smithsonian Center for Astrophysics).

Множество ошибок и промахов в первом издании этой книги было замечено и указано всего лишь несколькими людьми, которые всегда были готовы оказать мгновенную помощь: Стэффорд Бэйнс (Stafford Baines), Мэтью Гиллман (Matthew Gillman) и Стюарт Андерсон (Stuart Anderson). Те ошибки, которые все еще остаются в книге, – это, само собой, исключительно мои собственные ошибки.

Кроме того, особенно благодарен следующим людям: Хелен Рейнолдс (Helen Reynolds), Крис Пикар (Chris Pickard), Элисон Уайтли (Alison Whiteley), Джеймс Элиотт (James Elliott), Лианна Ишихара (Lianna Ishihara) и Мило Шаффер (Milo Shaffer). Как и любой человек, я очень нуждался в поддержке, поощрении и дружеском отношении Натали Хэйнс (Natalie Haynes).

Список листингов

Глава 1

Введение

1.1 Об этой книге

Эта книга предназначена для того, чтобы помочь ученым и инженерам освоить версию 3 языка программирования Python и связанных с ней библиотек: NumPy, SciPy, Matplotlib и pandas. Для чтения книги не требуется предварительный опыт программирования и научные знания в какой-либо конкретной области. Но знакомство с некоторыми математическими дисциплинами, такими как тригонометрия, комплексные числа и основы математического анализа, будет полезным при выполнении примеров и упражнений.

Python – мощный язык программирования со множеством расширенных функциональных возможностей и дополнительных пакетов поддержки. Основной синтаксис языка прост и понятен для изучения, но в полном объеме изучить его невозможно в книге такого размера. Таким образом, наша цель – сбалансированное, но достаточно подробное введение в самые главные функциональные возможности языка и самых важных библиотек. В текст включено множество примеров, связанных с научными исследованиями, в конце каждого раздела приведен список вопросов (коротких задач, предназначенных для проверки полученных знаний) и упражнений (более сложных задач, для решения которых обычно требуется написать небольшую компьютерную программу). Хотя нет необходимости выполнять абсолютно все упражнения, для читателей будет полезно попытаться выполнить хотя бы некоторые из них. Раздел, пример или упражнение, содержащие более сложный материал, который можно пропустить при первом чтении, обозначены символом ◊.

В главе 2 подробно рассматривается основной синтаксис, структуры данных и средства управления потоком выполнения в программе на языке Python. Глава 3 представляет собой небольшое отступление с описанием использования библиотеки pyplot для создания графических вариантов представления данных: это полезно для визуализации вывода программ в последующих главах. В главе 4 содержится более продвинутое описание ядра языка Python и краткое введение в объектно-ориентированное программирование. Далее следует еще одна короткая глава 5, представляющая широко известные виртуальные блокнотные среды IPython и Jupyter, далее – главы о научном программировании с использованием библиотек NumPy, SciPy, Matplotlib и pandas. В заключитель-

ной главе рассматриваются более общие темы научного программирования, включая арифметику с плавающей точкой, обеспечение стабильности алгоритмов и стиль программирования.

Читатели, уже знакомые с языком программирования Python, могут бегло просмотреть главы 2 и 4.

Примеры кода и решения упражнений можно загрузить с веб-сайта книги https://scipython.com/. Следует отметить, что хотя комментарии полностью включены в эти загружаемые программы, они не столь подробны в печатной версии данной книги: вместо этого исходный код описывается в тексте с помощью пронумерованных ссылок (например, ❶). Читатели, которые предпочитают вводить исходный код этих программ вручную, могут пожелать добавить собственные описательные комментарии в код.

1.2 Немного о Python

Python – это мощный язык программирования общего назначения, который разработал Гвидо ван Россум (Guido van Rossum) в 1989 году[1]. Python классифицируется как язык программирования высокого уровня, в котором автоматически обрабатывается большинство фундаментальных операций (таких как управление памятью), выполняемых на уровне процессора («машинный код»). Python считается языком более высокого уровня, чем, например, C, из-за его выразительного синтаксиса (который во многих случаях близок к естественному языку) и богатого разнообразия встроенных структур данных, таких как списки, кортежи, множества и словари. Например, рассмотрим следующую программу на Python, которая выводит список имен в отдельных строках.

Листинг 1.1. Вывод списка имен с использованием программы, написанной на Python

```
# eg1-names.py: вывод трех имен в консоли.

names = ['Isaac Newton', 'Marie Curie', 'Albert Einstein']
for name in names:
print(name)
```

Вывод:

```
Isaac Newton
Marie Curie
Albert Einstein
```

А теперь сравните исходный код из листинга 1.1 с кодом программы на C, которая делает то же самое.

[1] До настоящего момента Гвидо – «benevolent dictator for life (BDFL)» – «великодушный диктатор, управляющий жизнью» языка Python.

Листинг 1.2. Вывод списка имен с использованием программы, написанной на языке C

```
/* eg1-names.c: вывод списка имен в консоли. */
#include <stdio.h>
#include <stdlib.h>

const char *names[] = {"Isaac Newton", "Marie Curie", "Albert Einstein"};

int main(void)
{
  int i;

  for (i = 0; i < (sizeof(names) / sizeof(*names)); i++) {
    printf("%s\n", names[i]);
  }

  return EXIT_SUCCESS;
}
```

Даже если вы незнакомы с языком C, из листинга 1.2 можно понять, что написание кода на C даже для такой простой задачи связано с большими трудозатратами и издержками: две инструкции `include` для использования библиотек, не загружаемых по умолчанию, явные объявления переменных для хранения списка (в C это массив (array)) имен `names`, для счетчика `i`, а также явный проход по индексам этого массива в цикле `for`. Требуется даже вручную добавлять символы концов строк (`\n` – символ перехода на новую строку). Затем этот исходный код должен быть скомпилирован, т. е. преобразован в машинный код, который понимает процессор, прежде чем можно будет его выполнить. Более того, здесь огромное количество возможностей совершить ошибки (программные «баги» (bugs)): попытка вывода имени, хранящегося по адресу `name[10]`, вероятнее всего, приведет к выводу так называемого «мусора»: компилятор языка C не остановит вас при попытке доступа к несуществующему имени.

Та же самая программа, написанная в три строки на языке Python, проста и выразительна: не нужно явно объявлять, что `names` – список строк, для цикла не требуется счетчик, подобный `i`, и не приходится включать отдельные библиотеки (инструкцией `import` в Python). Для запуска Python-программы нужно просто ввести команду `python eg1-names.py`, которая автоматически вызовет интерпретатор Python для компиляции и выполнения полученного байт-кода (bytecode) (это особый тип промежуточного представления программы между исходным кодом и конечным машинным кодом, который Python перенаправляет в процессор).

Синтаксис Python призван гарантировать, что «существует один – и преимущественно единственный – очевидный способ сделать это». Это отличает Python от некоторых других широко известных языков высокого уровня, таких как Ruby и Perl, в которых используется противоположный подход, выражающийся в кратком принципе «существует более одного способа сделать это». Например, существует (как минимум) четыре очевидных способа вывода того же списка имен на языке Perl[2].

[2] Уточняю: очевидных для программистов на языке Perl.

Листинг 1.3. Различные способы вывода списка имен с использованием программы, написанной на Perl

```
@names = ("Isaac Newton", "Marie Curie", "Albert Einstein");
# Метод 1
print "$_\n" for @names;

# Метод 2
print join "\n", @names;
print "\n";

# Метод 3
print map { "$_\n" } @names;

# Метод 4
$" = "\n";
print "@names\n";
```

(Также обратите внимание на знаменитый лаконичный, но иногда непонятный синтаксис языка Perl.)

1.2.1 Преимущества и недостатки языка Python

Ниже перечислены некоторые из основных преимуществ языка программирования Python, а также причины, по котором вы, возможно, захотите его использовать:

- ясный и простой синтаксис позволяет быстро писать программы на Python и в общем сводит к минимуму возможности совершения скрытых ошибок. При правильном подходе результатом является высококачественное программное обеспечение, которое легко сопровождать и расширять;
- сама рабочая программная среда Python и связанные с ней библиотеки бесплатны, а кроме того, представляют собой программное обеспечение с открытым исходным кодом, в отличие от коммерческих предложений, таких как Mathematica и MATLAB;
- поддержка многих платформ: Python доступен для каждой общедоступной компьютерной системы, в том числе Windows, Unix, Linux и macOS. Несмотря на то что существуют расширения, зависящие от конкретной платформы, всегда есть возможность написания кода, который будет работать на любой платформе без каких-либо изменений;
- для Python существует большая библиотека модулей и пакетов, которая расширяет его функциональность. Многие из этих модулей и пакетов доступны как часть «стандартной библиотеки» (Standard Library), предоставляемой вместе с интерпретатором языка Python. Другие, в том числе библиотеки NumPy, SciPy, Matplotlib и pandas, используемые в научных вычислениях, можно абсолютно бесплатно загрузить и установить;
- язык Python относительно прост для изучения. Синтаксис и ключевые слова для основных операций применяются постоянно и согласованно в большинстве случаев более продвинутого использования языка. Сообщения об ошибках в основном представляют собой разумные пред-

положения о том, что пошло не так, в отличие от обобщенных «крахов», характерных для компилируемых языков высокого уровня, подобных C;

- Python – гибкий язык: его часто описывают как язык «многих парадигм», в котором имеются наилучшие функциональные возможности для процедурного, объектно-ориентированного и функционального программирования. Он требует совсем небольшой подготовительной работы, обязательной в других языках, когда задачу можно решить лишь с применением одного из этих подходов.

Так в чем же дело? А в том, что у Python имеются и некоторые недостатки, из-за которых он не подходит для абсолютно любого приложения:

- скорость выполнения программы на Python не так высока, как программ на других, полностью компилируемых языках, таких как C и Fortran. Для крупномасштабной вычислительной работы библиотеки NumPy и SciPy до некоторой степени способны облегчить ситуацию, используя «скрытый внутри» скомпилированный код C, но за счет некоторого снижения гибкости. Однако для множества приложений различия в скорости не так значимы, и снижение скорости выполнения почти полностью компенсируется гораздо более высокой скоростью разработки. Таким образом, процесс написания и отладки программы на Python занимает намного меньше времени, чем аналогичный процесс разработки на C, C++ или Java;
- трудно скрыть или замаскировать исходный код программы на Python, чтобы защитить ее от копирования и/или изменения. Но это не имеет особого значения, так как не существует успешных коммерческих программ на Python;
- на протяжении всей истории существования Python самыми частыми претензиями становились жалобы на излишне быстрое его развитие, приводящее к проблемам несовместимости между версиями. В действительности существуют два самых важных различия между версиями Python 2 и Python 3 (описанные в следующем разделе и в приложении Б), но претензии и жалобы основаны на том факте, что в группе версий Python 2 происходили основные усовершенствования и дополнения языка, из-за которых код, написанный в более поздней версии (например, в версии 2.7), не мог работать в более ранней версии (например, в версии 2.6), хотя код, написанный для более ранней версии Python, всегда будет работать в более поздней версии (в обеих ветвях – 2 и 3). Если вы используете самую последнюю версию Python (см. раздел 1.3), то, вероятнее всего, не столкнетесь с этой проблемой, но некоторые дистрибутивы операционных систем, в комплект которых входит Python, достаточно консервативны и устанавливают по умолчанию более старую версию.

1.2.2 Python 2 или Python 3

1 января 2020 года Python 2 завершил свой «жизненный путь»: он больше не будет обновляться и официально поддерживаться, поскольку более новая версия Python 3 уже активно развивается и сопровождается. Хотя различия между этими двумя версиями выглядят минимальными, код, написанный на Python 3, не будет работать в среде Python 2, и наоборот: Python 3 не обеспе-

чивает обратную совместимость со своим предшественником. В этой книге изучается Python 3.

Поскольку Python 3 появился в 2009 году, количество пользователей и поддержка библиотек для этой версии выросли до такого уровня, что новые пользователи не должны обнаружить особых преимуществ в изучении версии Python 2, за исключением необходимости поддержки старого кода.

Существует несколько обоснований внесения основных различий между версиями (с приведением в негодность пользовательского исходного кода не так-то легко примириться): Python 3 исправляет некоторые весьма неприятные особенности и нестыковки в языке, а также обеспечивает поддержку Юникода (Unicode) для всех строк (это полностью устраняет путаницу, возникавшую при обработке юникодных и неюникодных строк в Python 2). Unicode – это международный стандарт для представления текста в большинстве систем обработки и записи текстовых данных в мире.

Предвижу, что у большинства читателей этой книги не возникнет никаких проблем в процессе преобразования исходного кода между двумя версиями Python, если это будет необходимо. Список основных различий и дополнительную информацию см. в приложении Б.

1.3 Установка Python

Официальный сайт Python www.python.org содержит подробные и простые инструкции по загрузке (скачиванию) Python. Но существует несколько полноценных дистрибутивных комплектов, содержащих библиотеки NumPy, SciPy и Matplotlib (так называемый SciPy Stack). Эти дистрибутивы помогут избежать необходимости скачивания и установки библиотек по отдельности:

- пакет Anaconda доступен бесплатно (в том числе и для коммерческого использования) на сайте www.anaconda.com/distribution. Устанавливаются версии Python 2 и Python 3, но версию по умолчанию можно выбрать либо перед скачиванием, как указано на этой веб-странице, либо после скачивания с использованием команды `conda`;
- аналогичный дистрибутив Enthought Deployment Manager (EDM) существует в бесплатной версии и с различными компонентами для платных версий, включая техническую поддержку и программное обеспечение для разработки. Этот дистрибутив можно скачать здесь: https://assets.enthought.com/downloads/.

В большинстве случаев один из этих дистрибутивов – это все, что вам нужно. Ниже приведены некоторые замечания по отдельным платформам.

Исходный код (и бинарные файлы для некоторых платформ) для пакетов NumPy, SciPy, Matplotlib и IPython по отдельности доступен на следующих сайтах:

- NumPy: https://github.com/numpy/numpy;
- SciPy: https://github.com/scipy/scipy;
- Matplotlib: https://matplotlib.org/users/installing.html;
- IPython: https://github.com/ipython/ipython;
- Jupyter Notebook и JupyterLab: https://jupyter.org/.

Windows

Для пользователей Windows существует пара дополнительных возможностей установки полного стека SciPy: Python(x,y) (https://python-xy.github.io) и WinPython (https://winpython.github.io/). Оба дистрибутива бесплатные.

macOS

Операционная система macOS (бывшая Mac OS X), основанная на Unix, включает в комплект Python, но обычно более старой версии Python 2. Вы не должны удалять или обновлять этот вариант установки (он необходим операционной системе), но можно выполнить приведенные выше инструкции по установке Python 3 и стека SciPy. В macOS нет встроенного менеджера пакетов (приложения для установки и сопровождения программного обеспечения), но существуют два широко известных независимых менеджера пакетов: Homebrew (https://brew.sh/) и MacPorts (www.macports.org), которые поддерживают Python 3 и соответствующие пакеты, если вы предпочитаете данный вариант.

Linux

Почти все дистрибутивы Linux обычно содержат версию Python 2, а не Python 3, поэтому, возможно, потребуется установка новой версии по приведенным выше ссылкам: дистрибутивы Anaconda и Enthought предлагают специальные версии Linux. В большинстве дистрибутивов Linux имеется собственный менеджер программных пакетов (например, `apt` в Debian и `rpm` в RedHat). Менеджер пакетов можно использовать для установки Python 3 и всех необходимых библиотек, хотя при поиске репозиториев пакетов может потребоваться некоторое исследование ресурсов интернета. Будьте внимательны: не заменяйте и не обновляйте системный вариант установки, так как от него могут зависеть другие приложения.

1.4 Командная строка

Большинство примеров исходного кода в этой книге написаны как независимые программы, которые можно запускать из командной строки (command line) (или из интегрированной среды разработки (integrated development environment – IDE), если вы используете одну из таких сред: см. раздел 10.3.2). Для доступа к интерфейсу командной строки (также известного как консоль или терминал) на различных платформах выполните инструкции, приведенные ниже:

- Windows 7 и более ранние версии: **Пуск** > **Все программы** > **Командная строка**. Другой вариант: в окне ввода **Пуск** > **Выполнить** ввести команду `cmd`;
- Windows 8: **Preview** (нижний левый угол экрана) > **Windows System: All apps**. Другой вариант: ввод команды `cmd` в окне поиска, спускающегося из верхнего правого угла экрана;
- Windows 10: из меню **Пуск** (значок Windows в нижнем левом углу экрана) > **Служебные Windows** > **Командная строка**. Другой вариант: ввод команды `cmd` в окне поиска, вызываемого значком лупы в нижнем левом углу экрана рядом со значком Windows;

- Mac OS X и macOS: **Finder** > **Applications** > **Utilities** > **Terminal**;
- Linux: если вы не используете графический пользовательский интерфейс, то вы уже в командной строке. При использовании графического интерфейса найдите приложение Terminal (в разных дистрибутивах по-разному, но обычно терминал находится в разделе *System Utilities* или в разделе *System Tools*).

Команды, вводимые в командной строке, интерпретируются приложением, которое называется командной оболочкой (shell), позволяющей пользователю перемещаться по файловой системе и запускать разнообразные приложения. Например, команда

```
python myprog.py
```

сообщает командной оболочке о необходимости вызова интерпретатора языка Python с передачей в него файла *myprog.py* как скрипта для выполнения. Затем результат работы этой программы возвращается в командную оболочку и выводится в консоли (в терминале).

Глава 2

Ядро языка Python I

2.1 Командная оболочка Python

В этой главе рассматриваются основы синтаксиса, структуры и типы данных языка программирования Python. В нескольких первых разделах предполагается написание не более чем нескольких строк кода Python, которые могут быть выполнены с использованием командной оболочки (shell) языка Python. Это интерактивная рабочая среда: пользователь вводит инструкции на языке Python, которые выполняются немедленно сразу после нажатия клавиши **Enter**.

Действия, необходимые для получения доступа к встроенной командной оболочке языка Python, отличаются в различных операционных системах. Для запуска командной оболочки из командной строки сначала необходимо открыть окно терминала (консоли), воспользовавшись инструкциями из раздела 1.4, затем ввести команду `python`.

Для выхода из командной оболочки Python выполните команду `exit()`.

После запуска командной оболочки Python вы увидите приветственное сообщение (которое может меняться в зависимости от используемой операционной системы и версии Python). В моей системе это сообщение выглядит так:

```
Python 3.7.5 (default , Oct 25 2019, 10:52:18)
[Clang 4.0.1 (tags/RELEASE_401/final)] :: Anaconda , Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Три правые угловые скобки (`>>>`) – это приглашение, или промпт (prompt), после этих символов приглашения вы будете вводить команды языка Python. В данной книге используется версия Python 3, поэтому вы должны проверить номер версии в первой строке – `Python 3.X.Y`: здесь не особенно важны значения `X` и `Y`, представляющие минорный номер версии.

Многие дистрибутивные пакеты Python содержат немного более продвинутую командную оболочку, которая называется IDLE, с дополнительными функциями завершения по нажатии клавиши **Tab** и подсветкой синтаксиса (syntax highlighting) (ключевые слова выделяются различными цветами, когда вы вводите их). Но мы не будем рассматривать это приложение, а предпочтем ему более новую и более усовершенствованную рабочую программную среду IPython, обсуждаемую в главе 5.

Кроме того, для многих вариантов установки существует возможность (особенно в Windows) запуска командной оболочки Python непосредственно из приложения, установленного в процессе установки самого интерпретатора Python. Некоторые варианты установки даже добавляют ярлык со значком на рабочий стол, так что можно открыть командную оболочку Python щелчком (или двойным щелчком) по этому ярлыку.

2.2 Числа, переменные, операции сравнения и логические операции

2.2.1 Типы числовых значений

Одним из самых простых объектов языка Python являются числовые значения, для которых определены три типа: целые числа (тип: int), числа с плавающей точкой (тип: float) и комплексные числа (тип: complex).

Целые числа

Целые числа (integers) – это математические целые числа, такие как 1, 8, –72 и 3 847 298 893 721 407. В версии Python 3 нет ограничений по их величине (кроме ограничения по доступности оперативной памяти компьютера). Арифметика целых чисел является точной.

Для удобства разрешается разделять группы цифр (разрядов) целого числа символом подчеркивания «_». Например, запись `299_792_458` интерпретируется как целое число `299 792 458`.

Числа с плавающей точкой

Числа с плавающей точкой (floating-point numbers) являются представлением действительных чисел, таких как 1.2, –0.36 и $1\ 67263 \times 10^{-7}$. Вообще говоря, не существует точных значений действительных чисел в представлении Python, но эти числа хранятся в бинарном (двоичном) виде с определенной точностью (в большинстве систем точность составляет 15–16 разрядов)[3], как описано в разделе 10.1. Например, дробь 4/3 хранится как двоичное равнозначное представление 1.33333333333333325931846502..., которое приблизительно (но не точно) равно бесконечно повторяющемуся десятичному представлению дроби 4/3 = 1.3333.... Более того, даже числа, для которых существует точное десятичное представление, могут не иметь точного бинарного представления: например, дробь 1/10 представлена бинарным числом, равным значению 0.10000000000000000555111512.... Из-за такой ограниченной точности арифметика чисел с плавающей точкой не является абсолютно точной, но при аккуратном отношении она «достаточно точна» для большинства научных приложений.

Любое отдельное число, содержащее символ точки (.), рассматривается в Python как представление числа с плавающей точкой. Также поддерживается научный формат записи чисел с использованием буквы `e` или `E` для отделения

[3] Это соответствует реализации по международному стандарту представления чисел с плавающей точкой двойной точности IEEE-754.

значимой части числа (мантиссы) от показателя степени: например, `1.67263e-7` представляет число $1\ 67263 \times 10^{-7}$.

Как и для целых чисел, группы разрядов можно разделять символами подчеркивания. Например, `1.602_176_634e-34`.

Комплексные числа

Комплексные числа, такие как 4+3j, состоят из действительной и мнимой частей (в Python мнимая часть обозначается буквой `j`), каждая из которых сама по себе является числом с плавающей точкой (даже если записана без символа точки). Таким образом, арифметика комплексных чисел не является точной, но обеспечивает такую же конечную ограниченную точность, что и числа с плавающей точкой (`float`).

Комплексное число может быть определено «сложением» действительного числа с мнимым (обозначенным суффиксом `j`): `2.3 + 1.2j` – или посредством отдельной передачи действительной и мнимой частей при вызове `complex`, например `complex(2.3, 1.2)`.

Пример П2.1. Ввод числа после приглашения командной оболочки Python просто возвращает это же число

```
>>> 5
5
>>> 5.
5.0
>>> 0.10
0.1                                                    ❶
>>> 0.0001
0.0001
>>> 0.0000999
9.99e-05                                               ❷
```

Следует отметить, что интерпретатор Python выводит числа стандартным способом. Например:

❶ Внутреннее представление числа 0.1, описанное выше, округляется до `0.1`, т. е. до самого короткого числового значения в этом представлении.

❷ Числа, меньшие по величине, чем 0.0001, выводятся в научном формате.

Число любого типа может быть создано из числа другого типа с помощью соответствующего конструктора (constructior):

```
>>> float(5)
5.0
>>> int(5.2)
5
>>> int(5.9)
5                                                      ❶
>>> complex(3.)
(3+0j)                                                 ❷
>>> complex(0., 3.)                                    ❸
3j
```

❶ Обратите внимание: положительное число с плавающей точкой округляется с недостатком (в меньшую сторону) во время преобразования его в целое число. Более общее правило: метод `int` округляет в сторону нуля: `int(-1.4)` дает результат `-1`.
❷ Создание объекта типа `complex` из объекта типа `float` генерирует комплексное число с мнимой частью, равной нулю.
❸ Для генерации абсолютно мнимого числа необходимо явно передать два числа в метод `complex`, при этом первая, действительная часть должна быть равна нулю.

2.2.2 Использование командной оболочки Python в качестве калькулятора

Простые арифметические действия

С описанными в предыдущем разделе тремя основными типами чисел можно использовать командную оболочку Python как простой калькулятор, применяя операторы, описанные в табл. 2.1. Это бинарные операторы, поскольку они работают с двумя числами (операндами) для создания третьего числа (например, при вычислении `2**3` получается результат `8`).

Таблица 2.1. Основные арифметические операторы языка Python

+	Сложение
–	Вычитание
*	Умножение
/	Деление с плавающей точкой
//	Целочисленное деление
%	Деление по модулю (взятие остатка)
**	Возведение в степень

В версии Python 3 существует два типа операции деления: деление чисел с плавающей точкой (/) всегда возвращает как результат число с плавающей точкой (или комплексное число), даже если применяется к целым числам. Целочисленное деление (//) всегда округляет результат с недостатком (в меньшую сторону), т. е. к ближайшему меньшему целому числу (floor division). Типом результата является `int`, только если оба операнда имеют тип `int`, иначе возвращается значение типа `float`. Ниже приведены примеры, помогающие понять эти правила.

Обычное («истинное») деление с плавающей точкой с использованием оператора (/):

```
>>> 2.7 / 2
1.35
>>> 9 / 2
4.5
>>> 8 / 4
2.0
```

Последняя операция деления возвращает значения типа `float` даже притом, что оба операнда имеют тип `int`.

Целочисленное деление с использованием оператора (//):

```
>>> 8 // 4
2
>>> 9 // 2
4
>>> 2.7 // 2
1.0
```

Обратите внимание: оператор // может применять целочисленную арифметику (с округлением в меньшую сторону) к числам с плавающей точкой.

Оператор деления по модулю (%) возвращает остаток от целочисленного деления:

```
>>> 9 % 2
1
>>> 4.5 % 3
1.5
```

И в этом случае возвращается значение типа `int`, только если оба операнда имеют тип `int`.

Приоритет операторов

Арифметические операции могут быть объединены в последовательность, из-за чего естественным образом возникает вопрос приоритета выполнения операций: например, при вычислении выражения `2 + 4 * 3` должно получиться `14` (в результате `2 + 12`) или `18` (в результате `6 * 3`)? Из табл. 2.2 следует, что результат должен быть равен 14, так как операция умножения имеет более высокий приоритет, чем операция сложения, поэтому должна выполняться первой. Эти правила приоритета изменяются при использовании круглых скобок, например `(2 + 4) * 3 = 18`.

Таблица 2.2 Приоритеты арифметических операторов языка Python

**	Самый высокий приоритет
*, /, //, %	
+, –	Самый низкий приоритет

Операторы с одинаковым приоритетом выполняются слева направо, за исключением операции возведения в степень (**), которая выполняется справа налево (т. е. «сверху вниз» при записи в обычной математической форме показателей степени над строкой). Например:

```
>>> 6 / 2 / 4           # равнозначно 3 / 4
0.75
>>> 6 / (2 / 4)         # равнозначно 6 / 0.5
12.0
>>> 2**2**3             # равнозначно 2**(2**3) == 2**8
256
>>> (2**2)**3           # равнозначно 4**3
64
```

В показанных здесь примерах символ «решетка» (#) обозначает начало комментария, который игнорируется интерпретатором. Комментарии иногда будут использоваться для более подробного объяснения конкретной инструкции, но при вводе кода для выполнения комментарии вводить не обязательно.

Методы и атрибуты числовых значений

Числовые значения в языке Python являются объектами (objects) (в действительности в Python все является объектами) и имеют определенные атрибуты (attributes), доступные при использовании формата «точка» (dot notation): `<объект>.<атрибут>` (такое использование точки не имеет ничего общего с десятичной точкой в числах с плавающей точкой). Некоторые атрибуты являются простыми значениями, например объекты комплексных чисел имеют атрибуты `real` и `imag`, соответствующие действительной и мнимой частям (с плавающей точкой) комплексного числа:

```
>>> (4 + 5j).real
4.0
>>> (4 + 5j).imag
5.0
```

Другими атрибутами являются методы (methods): вызываемые функции, которые определенным образом работают со своим объектом[4]. Например, для комплексных чисел существует метод `conjugate`, который возвращает сопряженное комплексное число:

```
>>> (4 + 5j).conjugate()
(4-5j)
```

Здесь пара пустых круглых скобок обозначает свойство вызываемости метода, т. е. выполнения вычисления сопряженного комплексного числа для числа 4 + 5j. Если скобки не указаны, например `(4 + 5j).conjugate`, то мы получаем ссылку на сам метод (без его вызова) – ведь этот метод тоже является объектом.

В действительности целые числа и числа с плавающей точкой не обладают слишком многими атрибутами, которые имеет смысл использовать показанным выше способом, но если вам интересно узнать, сколько битов занимает целое число в оперативной памяти, то можно воспользоваться методом `bit_length`. Например:

```
>>> (3847298893721407). bit_length()
52
```

Следует отметить, что Python выделяет столько памяти, сколько необходимо для точного представления целого числа.

---

[4] В этой книге термины метод (method) и функция (function) используются как взаимозаменяемые. В языке Python все является объектами, поэтому различие между этими терминами не столь существенно, как в некоторых других языках программирования.

Математические функции

Две математические функции из множества, предоставляемые «по умолчанию» как встроенные (built-in) функции, – `abs` и `round`.

Функция `abs` возвращает абсолютное значение числа:

```
>>> abs(-5.2)
5.2
>>> abs(-2)
2
>>> abs(3 + 4j)
5.0
```

Это пример полиморфизма (polymorphism): одна и та же функция `abs` выполняет различные операции с различными объектами. Если в функцию передано действительное число x, то возвращается $|x|$, неотрицательная величина этого числа без учета знака. Если передается комплексное число $z = x + iy$, то возвращается его модуль $|z| = \sqrt{(x^2 + y^2)}$.

Функция `round` (с одним аргументом) округляет число с плавающей точкой до ближайшего целого числа, используя для этого метод округления банкира (Banker's rounding)[5]:

```
>>> round(-9.62)
-10
>>> round(7.5)
8
>>> round(4.5)
4
```

Можно также задать количество точных разрядов после десятичной точки как второй аргумент, передаваемый в функцию `round()`:

```
>>> round(3.141592653589793 , 3)
3.142
>>> round(96485.33289, -2)
96500.0
```

Python представляет собой в высшей степени модульный язык: функциональность доступна в пакетах и модулях, которые при необходимости импортируются, но по умолчанию не загружаются: это позволяет экономить память, требуемую для работы Python-программы, сохраняя ее размер на минимальном уровне, и улучшать производительность. Например, многие полезные математические функции расположены в модуле `math`, который импортируется следующей командой:

```
>>> import math
```

В модуль `math` включены операции для работы с числами с плавающей точкой и целыми числами (для функций, работающих с комплексными числами, существует другой модуль `cmath`). Функции вызываются с передачей одного

[5] При округлении по методу банкира числа «с 0.5» всегда округляются до ближайшего четного целого числа.

числа (иногда нескольких чисел) внутри круглых скобок (числа принимаются как аргументы вызываемой функции). Например:

```
>>> import math
>>> math.exp(-1.5)
0.22313016014842982
>>> math.cos(0)
1.0
>>> math.sqrt(16)
4.0
```

Полный список математических функций, предоставляемых модулем `math`, можно найти в онлайновой документации[6]. Некоторые наиболее часто используемые математические функции перечислены в табл. 2.3.

Таблица 2.3. Некоторые функции, предоставляемые модулем `math`. Предполагается, что аргументы, соответствующие значениям углов, задаются в радианах

`math.sqrt(x)`	$\sqrt{x}$
`math.exp(x)`	e^x
`math.log(x)`	ln x
`math.log(x, b)`	$\log_b x$
`math.log10(x)`	$\log_{10} x$
`math.sin(x)`	sin(x)
`math.cos(x)`	cos(x)
`math.tan(x)`	tan(x)
`math.asin(x)`	arcsin(x)
`math.acos(x)`	arccos(x)
`math.atan(x)`	arctan(x)
`math.sinh(x)`	sinh(x)
`math.cosh(x)`	cosh(x)
`math.tanh(x)`	tanh(x)
`math.asinh(x)`	arsinh(x)
`math.acosh(x)`	arcosh(x)
`math.atanh(x)`	artanh(x)
`math.hypot(x, y)`	Евклидово нормальное расстояние (гипотенуза) $\sqrt{(x^2 + y^2)}$
`math.factorial(x)`	Факториал x!
`math.erf(x)`	Функция ошибок по x
`math.gamma(x)`	Гамма-функция по x, $\Gamma(x)$
`math.degrees(x)`	Преобразование x из радианов в градусы
`math.radians(x)`	Преобразование x из градусов в радианы
`math.isclose(a, b)`	Проверка равенства a и b в пределах некоторого допустимого отклонения

Кроме того, модуль `math` предоставляет два весьма полезных нефункциональных атрибута: `math.pi` и `math.e` – это значения математических констант π и e (основание натуральных логарифмов).

[6] https://docs.python.org/3/library/math.html.

Существует возможность импорта модуля `math` с помощью команды `from math import *`, после чего становится доступным прямой доступ ко всем его функциям:

```
>>> from math import *
>>> cos(pi)
-1.0
```

Это может оказаться удобным для работы в командной оболочке Python, но не рекомендуется в программах на языке Python. Возникает опасность конфликтов имен (особенно если таким способом импортируются функции из многих модулей), и становится трудно узнать, из какого модуля импортирована та или иная функция. Импорт командой `import math` сохраняет связь функций с пространством имен (namespace) соответствующего модуля, поэтому, даже несмотря на то что ввод `math.cos` требует больше нажатий на клавиши, такой способ делает исходный код более удобным для понимания и сопровождения.

Пример П2.2. Вполне естественно, что математические функции можно объединять в одном выражении

```
>>> import math
>>> math.sin(math.pi/2)
1.0
>>> math.degrees(math.acos(math.sqrt(3)/2))
30.000000000000004
```

Обратите внимание на ограниченную (конечную) точность последнего выражения: точный результат $\arccos(\sqrt{3}/2) = 30°$.

Тем фактом, что функция `int` округляет в меньшую сторону при преобразовании положительного числа с плавающей точкой в целое число, можно воспользоваться для определения количества разрядов положительного целого числа:

```
>>> int(math.log10(9999)) + 1
4
>>> int(math.log10(10000)) + 1
5
```

2.2.3 Переменные

Что такое переменная

При создании объекта, например типа `float`, в программе на Python или при использовании командной оболочки Python для этого объекта выделяется память: в компьютерной архитектуре место расположения данного фрагмента памяти называется адресом памяти (address). Действительное значение адреса памяти объекта не очень полезно (и не очень удобно) в Python, но если вы захотите, то можете узнать адрес, вызвав встроенный метод `id`:

```
>>> id(20.1)
```

```
4297273888                    # пример
```

Это число является ссылкой на конкретное место в памяти, соответствующее фрагменту памяти, выделенному для хранения объекта типа `float` со значением `20.1`.

Кроме самых простых вариантов использования, существует необходимость в хранении объектов, участвующих в вычислении или в выполнении алгоритма, к которым можно было бы обращаться по некоторому более удобному и осмысленному имени (а не по числовому адресу в памяти). Для этого существуют переменные (variables)[7]. Имя переменной может быть присвоено любому объекту («связано» с любым объектом) и использоваться для идентификации этого объекта в дальнейших вычислениях. Например:

```
>>> a = 3
>>> b = -0.5
>>> a * b
-1.5
```

В этом фрагменте кода создается объект типа `int` со значением `3`, и ему присваивается имя `a`. Затем создается объект типа `float` со значением `-0.5`, и ему присваивается имя `b`. Выполняется вычисление `a * b`: значения `a` и `b` умножаются друг на друга, и возвращается результат. Этот результат не присваивается никакой переменной, сразу после вывода на экран он отбрасывается. Таким образом, для хранения результата память не требуется, а для значения `-1.5` типа `float` выделяется память только на то короткое время, которое необходимо для предъявления результата пользователю, затем память освобождается, а значение теряется[8]. Если результат необходим для дальнейших вычислений, то вы должны присвоить его другой переменной:

```
>>> c = a * b
>>> c
-1.5
```

Обратите внимание: не требуется предварительное объявление (declare) переменных до присваивания им значений (т. е. для сообщения Python о том, что имя переменной `a` является ссылкой на целое число, имя переменной `b` – ссылкой на число с плавающей точкой и т. д.), как это необходимо делать в некоторых языках программирования. Python – язык с динамической типизацией (dynamically typed), поэтому необходимый тип объекта логически выводится из его определения: при отсутствии десятичной точки число `3` определяется как `int`. Число `-0.5` выглядит как число с плавающей точкой, и Python определяет `b` как `float`[9].

---

[7] В Python, вероятнее всего, более правильно говорить об идентификаторах объектов (object identifiers) или об именах-идентификаторах (identifier names), нежели о переменных, но мы не будем вводить излишне строгую терминологию в этом случае.

[8] В действительности в интерактивном сеансе работы с Python результат самого последнего вычисления сохраняется в специальной переменной с именем `_` (символ подчеркивания), поэтому он не отбрасывается до тех пор, пока не будет перезаписан результатом следующего вычисления.

[9] Иногда такой способ называют «утиной типизацией» (duck typing) в соответствии с фразой, приписываемой Джеймсу Уиткомбу Райли (James Whitcomb Riley): «Когда

Имена переменных

Ниже приводится несколько правил, определяющих формирование допустимых («правильных») имен переменных:

- в именах переменных учитывается регистр символов (букв): `a` и `A` – это разные имена переменных;
- имена переменных могут содержать любую букву, символ подчеркивания (_) и любую цифру (0–9)…
- …но не должны начинаться с цифры;
- имя переменной не должно совпадать с одним из зарезервированных ключевых слов, приведенных в табл. 2.4;
- встроенные имена констант `True`, `False` и `None` запрещено использовать как имена переменных.

Таблица 2.4. Зарезервированные ключевые слова языка Python 3

and	as	assert	async	await	break
class	continue	def	del	elif	else
except	finally	for	from	global	if
import	in	is	lambda	nonlocal	not
or	pass	raise	return	try	while
with	yield	False	True	None	

Большинство зарезервированных ключевых слов вряд ли подходят для имен переменных, за исключением разве что слова `lambda`. Программисты на Python часто используют имя `lam` при необходимости. Текстовый редактор с расширенными функциями подсвечивает ключевые слова при вводе исходного кода программы, поэтому путаница с именами возникает редко.

Существует возможность присваивания переменной имени, совпадающего с именем встроенной функции (например, `abs` или `round`), но после такого присваивания встроенная функция становится недоступной, поэтому присваивания подобных имен лучше избегать – к счастью, большинство встроенных функций имеют имена, которые вряд ли будут выбираться в реальной практике[10].

В дополнение к перечисленным выше правилам ниже приводятся некоторые соглашения по стилю, определяющие общепринятые практические принципы именования переменных:

- имена переменных должны быть осмысленными (`area` лучше, чем `a`)…
- …но не слишком длинными (`the_area_of_the_triangle` – это слишком громоздкое имя);
- в общем случае лучше не использовать `I` (буква `i` в верхнем регистре), `l` (буква `L` в нижнем регистре) и букву `O` в верхнем регистре, так как они очень похожи на цифры 1 и 0;

я вижу птицу, которая ходит как утка, плавает как утка и крякает как утка, я называю эту птицу уткой».

[10] Полный список имен встроенных функций см. в документации: https://docs.python.org/3/library/functions.html.

- имена переменных `i`, `j` и `k`, как правило, используются для целочисленных счетчиков;
- рекомендуется использовать имена с буквами нижнего регистра с разделением слов символами подчеркивания вместо стиля именования «CamelCase»: например, `mean_height`, а не `MeanHeight`[11].

Эти и многие другие правила и соглашения определены в руководстве по стилю под названием PEP8, которое представляет собой одну из частей документации Python[12] (также см. раздел 10.3.1).

Нарушение этих правил хорошего стиля не приведет к нарушению правильной работы программы, но такую программу, возможно, труднее будет сопровождать и отлаживать – может быть, вы помогаете самому себе, соблюдая правила стиля.

Пример П2.3. Формула Герона для вычисления площади A треугольника со сторонами a, b, c:

$$A = \sqrt{s(s-a)(s-b)(s-c)}, \text{ где } s = \frac{1}{2}(a+b+c).$$

Например:

```
>>> a = 4.503
>>> b = 2.377
>>> c = 3.902
>>> s = (a + b + c) / 2
>>> area = math.sqrt(s * (s - a) * (s - b) * (s - c))                    ❶
>>> area
4.63511081571606
```

❶ Не забудьте выполнить команду импорта `import math`, если до этого вы пока еще не работали в сеансе Python.

Пример П2.4. Тип данных и адрес памяти объекта по имени соответствующей переменной можно получить с помощью встроенных функций `type` и `id`:

```
>>> type(a)
<class 'float'>
>>> id(area)
4298539728                  # пример
```

2.2.4 Операции сравнения и логические операции

Операторы

Основные операторы, используемые в Python для сравнения объектов (например, чисел), перечислены в табл. 2.5.

---

[11] Стиль имен CamelCase в Python обычно применяется для имен классов: см. раздел 4.6.2.

[12] https://legacy.python.org/dev/peps/pep-0008/.

Таблица 2.5. Операторы сравнения Python

==	Равно
!=	Не равно
>	Больше
<	Меньше
>=	Больше или равно
<=	Меньше или равно

Результатом сравнения является логический (boolean) объект (или объект типа `bool`), который может содержать одно и только одно из двух возможных значений: `True` или `False`. Это встроенные постоянные ключевые слова, которым не могут быть присвоены другие значения. Например:

```
>>> 7 == 8
False
>>> 4 >= 3.14
True
```

Python способен, если это возможно без двусмысленности, сравнивать объекты различных типов: целое число `4` приводится к типу `float` для сравнения с числом `3.14`.

Обратите особое внимание на различие между оператором `==` и оператором `=`. Один знак равенства – это присваивание (assignment), операция, которая не возвращает значение: инструкция `a = 7` присваивает имя переменной `a` целочисленному объекту `7`, и это все. Выражение `a == 7` – это проверка условия: возвращается значение `True` или `False`, в зависимости от значения переменной `a`[13].

Особое внимание требуется при сравнении чисел с плавающей точкой на равенство. Поскольку числа с плавающей точкой хранятся с некоторой неточностью, вычисления с ними часто приводят к потере точности, и можно получить неожиданные результаты по неосторожности. Например:

```
>>> a = 0.01
>>> b = 0.1**2
>>> a == b
False
```

В этом примере 0.01 не может быть представлено точно как число с плавающей точкой, а хранится (в моей системе) как бинарное число, равное значению 0.010000000000000000208, а с другой стороны, результат возведения в квадрат представления значения 0.1 в виде числа с плавающей точкой равен 0.010000000000000000194, и эти два числа не равны. Более подробно об этом см. раздел 10.1.

[13] В некоторых языках, например в C, операция присваивания возвращает значение, которое присваивается переменной, и это может приводить к некоторым опасным и трудно обнаруживаемым ошибкам, когда оператор = используется ошибочно, как оператор сравнения.

В версии Python 3.5 в библиотеке `math` появилась функция `isclose`, которая проверяет равенство двух чисел с плавающей точкой с учетом некоторого абсолютного или относительного интервала допустимого отклонения:

```
>>> math.isclose(0.1**2, 0.01)
True
```

Величина относительного допустимого отклонения может быть установлена с помощью аргумента `rel_tol`, по умолчанию равного `1.e-9`: это максимально допустимая разность между двумя числами относительно абсолютного значения большего из этих чисел. Например, для проверки равенства `a` и `b` в пределах 5 % от большего из них:

```
>>> a = 9.5
>>> b = 10
>>> math.isclose(a, b, rel_tol=0.05)
True
```

При этом типе относительного сравнения могут возникать проблемы, если одно из чисел равно нулю[14]. В этом случае может оказаться полезной проверка абсолютного допустимого отклонения, значение которого передается в аргументе `abs_tol` (по умолчанию 0):

```
>>> math.isclose(0, 1.e-12) # сравнение по относительному допустимому отклонению дает
                            # неверный результат, если rel_tol < 1
False
>>> math.isclose(0, 1.e-12, abs_tol=1.e-10)
True
```

Логические операторы

Операции сравнения можно изменять и объединять с помощью ключевых слов, обозначающих логические операторы: `and`, `not` и `or`. См. табл. 2.6, 2.7 и 2.8.

Таблица 2.6. Таблица истинности для оператора `not`

P	not P
True	False
False	True

Таблица 2.7. Таблица истинности для оператора and

P	Q	P and Q
True	True	True
False	True	False
True	False	False
False	False	False

[14] Относительная разность между любым числом a и 0 равна $(|a| - 0) / |a|$, и это значение определенно больше значения `rel_tol`, если для `rel_tol` установлено значение меньше 1.

Таблица 2.8. Таблица истинности для оператора or

P	Q	P or Q
True	True	True
False	True	True
True	False	True
False	False	False

Например:

```
>>> 7.0 > 4 and -1 >= 0         # равнозначно True and False
False
>>> 5 < 4 or 1 != 2             # равнозначно False or True
True
```

В составных выражениях, подобных приведенным выше, первыми выполняются операторы сравнения, а затем логические операторы в порядке их приоритетности: `not`, `and`, `or`. Приоритет можно изменить с помощью круглых скобок, как для арифметических выражений. Например:

```
>>> not 7.5 < 0.9 or 4 == 4
True
>>> not (7.5 < 0.9 or 4 == 4)
False
```

Таблицы 2.6, 2.7 и 2.8 – это таблицы истинности для логических операторов. Следует отметить, что, как и в большинстве языков программирования, в Python оператор `or` представляет собой вариант «включающее или» (inclusive or), при котором `A or B` равно `True`, если `A`, или `B`, или и `A`, и `B` истинны, в отличие от оператора «исключающее или» (exclusive or) (`A xor B` равно `True`, только если либо `A`, либо `B` истинно, но не оба операнда одновременно[15]).

◊ Логические равенства и условное присваивание

В выражении логической проверки не всегда необходимо выполнять явное сравнение для получения логического значения: Python попытается преобразовать объект в тип `bool`, если это необходимо. Для числовых объектов 0 преобразуется в `False`, а любое ненулевое значение – в `True`:

```
>>> a = 0
>>> a or 4 < 3           # равнозначно: False or 4 < 3
False
>>>
>>> not a + 1            # равнозначно: not True
False
```

[15] В Python нет встроенного оператора `xor`, но его можно импортировать как функцию с помощью инструкции `from operator import xor`. Вызов этой функции `xor(a, b)` возвращает `True` или `False`.

В этом примере операция сложения имеет более высокий приоритет, чем логический оператор `not`, поэтому сначала вычисляется `a+1` и дает результат `1`. Это соответствует логическому значению `True`, следовательно, все выражение равнозначно выражению `not True`. Для явного преобразования любого объекта в логический объект используется конструктор `bool`:

```
>>> a = 0
>>> a - 2 or a                                                    ❶
-2
>>> 4 > 3 and a - 2                                               ❷
-2
>>> 4 > 3 and a                                                   ❸
0
```

Логические выражения вычисляются слева направо, при этом предполагается, что вычисления `and` и `or` выполняются по укороченной схеме (shortcircuited): второе выражение вычисляется, только если необходимо определить истинное значение всего выражения в целом. Приведенные выше три примера можно проанализировать следующим образом:

❶ В первом примере сначала вычисляется `a-2`: результат равен –2, т. е. равнозначен значению `True`, поэтому условие `or` выполнено, и операнд, вычисленный как `True`, возвращается немедленно: –2.

❷ Проверка условия `4 > 3` дает результат `True`, поэтому должно быть вычислено второе выражение, чтобы определить истинность условия `and`. Выражение `a-2` равно –2, что также равнозначно `True`, следовательно, условие `and` выполнено, и возвращается –2 (как результат выражения, вычисленного самым последним).

❸ В последнем случае `a` равно 0, что равнозначно `False`, поэтому результатом вычисления условия `and` становится `False`, и возвращается значение 0.

None – специальное значение в языке Python

В Python определено отдельное специальное значение `None` с особенным типом `NoneType`. Оно используется для представления отсутствия определенного значения, например в случаях, когда не существует возможного значения, и других подобных случаях. Это особенно удобно, когда нужно избежать введения произвольных значений по умолчанию (таких как `0`, `-1` или `-99`) для испорченных или отсутствующих данных.

В выражениях логического сравнения `None` вычисляется как `False`, но для проверки равенства значения переменной `x` значению `None` необходимо использовать конструкцию

```
if x is None
```

и

```
if x is not None
```

вместо сокращенных вариантов `if x` и `if not x`[16].

[16] Обратите внимание: `not x` также вычисляется как `True`, если `x` является любым из следующих объектов: `0`, `False` или пустая структура данных, например пустой список,

Пример П2.5. Общеизвестный прием в Python – присваивание переменной значения, возвращаемого логическим выражением:

```
>>> a = 0
>>> b = a or -1
>>> b
-1
```

Это буквально означает (предполагается, что a должно быть целым числом): «установить b равным значению a, если не выполнено условие a == 0, а в случае выполнения этого условия установить значение b равным –1».

2.2.5 Неизменяемость и идентичность

Объекты, рассматриваемые до настоящего момента, такие как целые числа и логические значения, являются неизменяемыми (immutable). Неизменяемые объекты никогда не изменяются после создания, хотя имя переменной может быть переназначено для ссылки на другой объект, отличающийся от объекта, которому было изначально присвоено это имя. Например, рассмотрим следующие присваивания:

```
>>> a = 8
>>> b = a
```

В первой строке создается целочисленный объект со значением 8 в памяти, и ему присваивается имя a. Во второй строке тому же объекту присваивается имя b. Это можно увидеть, проверив адрес памяти объекта по каждому имени:

```
>>> id(a)
4297273504
>>> id(b)
4297273504
```

Таким образом, a и b – ссылки на один и тот же целочисленный объект. Теперь предположим, что имя a переназначается для нового числового объекта:

```
>>> a = 3.14
>>> a
3.14
>>> b
8
>>> id(a)
4298630152
>>> id(b)
4297273504
```

Обратите внимание: значение b не изменилось – эта переменная продолжает ссылаться на исходное значение 8. Переменная a теперь ссылается на новый

[ ] или пустая строка ''. Таким образом, это не самый надежный способ проверки неравенства значения x значению None.

объект типа `float` со значением `3.14`, расположенный по новому адресу памяти. Именно это подразумевается под неизменяемостью: это не «переменная», которая не может изменяться, а сам объект является неизменяемым – см. рис. 2.1.

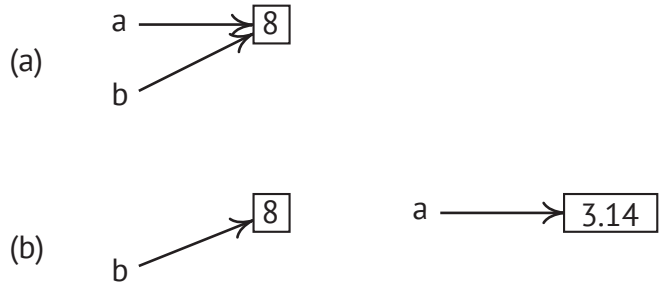


Рис. 2.1. (а) Две переменные ссылаются на одно и то же целое число; (б) после переназначения для переменной `a` другого значения

Более удобный способ установления того факта, что две переменные ссылаются на один и тот же объект, – использование оператора `is`, который определяет идентичность (identity) объектов:

```
>>> a = 2432
>>> b = a
>>> a is b
True
>>> c = 2432
>>> c is a
False
>>> c == a
True
```

Здесь присваивание `c = 2432` создает абсолютно новый объект, поэтому вычисление выражения `c is a` дает результат `False`, даже притом, что `a` и `c` содержат одинаковое значение. Таким образом, эти две переменные ссылаются на различные объекты с одинаковыми значениями.

Часто необходимо изменить значение переменной каким-либо способом, например:

```
>>> a = 800
>>> a = a + 1
>>> a
801
```

Целые числа 800 и 801 неизменяемы: строка *a* = *a* + 1 создает новый целочисленный объект со значением 801 (правая сторона выражения вычисляется в первую очередь) и присваивает его переменной с именем *a* (старое значение 800 «забывается»[17], если только на это значение не ссылается какая-либо

[17] То есть память, выделенная Python для этого значения, освобождается (механизмом сборки мусора – garbage collector) и становится доступной для общего использования.

другая переменная). Таким образом, a указывает на разные адреса памяти до и после выполнения этой инструкции.

Подобное переназначение переменной на результат выполнения арифметической операции выполняется настолько часто, что была введена удобная укороченная форма ее записи: комбинированное присваивание (augmented assignment) a += 5 равнозначно присваиванию a = a + 5. Точно так же работают операторы -=, *=, /=, //=, %=. Но в Python не поддерживаются операции инкремента и декремента в C-стиле, такие как a++ для операции a += 1[18].

Пример П2.6. Python предоставляет оператор is not: более естественно написать c is not a, чем not c is a

```
>>> a = 8
>>> b = a
>>> b is a
True
>>> b /= 2
>>> b is not a
True
```

◊ **Пример П2.7.** С учетом приведенного выше описания может показаться странным следующий факт:

```
>>> a = 256
>>> b = 256
>>> a is b
True
```

Причина в том, что Python сохраняет кеш (cache) общего пользования для небольших целочисленных объектов (в моей системе это числа от -5 до 256). Для повышения производительности присваивание a = 256 связывает имя переменной a с существующим целочисленным объектом без необходимости выделения новой памяти для него. Поскольку такое же присваивание выполняется и для переменной b, эти две переменные в данном конкретном случае в действительности указывают на один и тот же объект. Противоположный пример:

```
>>> a = 257
>>> b = 257
>>> a is b
False
```

---

[18] Присваивание и комбинированное присваивание в Python – это инструкции (команды), а не выражения, поэтому они не возвращают значение, следовательно, не могут объединяться в цепочку.

2.2.6 Упражнения

Вопросы

В2.2.1. Определить результаты вычислений следующих выражений и проверить их, используя командную оболочку Python.

а) `2.7 / 2`
б) `2 / 4 - 1`
в) `2 // 4 - 1`
г) `(2 + 5) % 3`
д) `2 + 5 % 3`
е) `3 * 4 // 6`
ж) `3 * (4 // 6)`
з) `3 * 2 ** 2`
и) `3 ** 2 * 2`

В2.2.2. Все операторы в табл. 2.1 являются бинарными: они обрабатывают два операнда (числа) и возвращают одно значение. Символ «минус» (–) также используется как унарный (unary) оператор, который возвращает отрицательное значение (точнее: изменяет знак на противоположный) одного обрабатываемого операнда. Например:

```
>>> a = 4
>>> b = -a
>>> b
-4
```

Обратите внимание: выражение `b = -a` (присваивающее переменной `b` значение `a` с противоположным знаком) отличается от выражения `b -= a` (которое вычитает `a` из `b` и сохраняет результат в `b`). Унарный оператор – обладает более высоким приоритетом, чем операторы `*`, `/` и `%`, но более низким приоритетом, чем оператор возведения в степень `**`, поэтому, например, `-2 ** 4` равно `-16` (так как вычисляется $-(2^4)$, а не $(-2)^4$).

Определить результаты вычислений следующих выражений и проверить их, используя командную оболочку Python.

а) `-2 ** 2`
б) `2 ** -2`
в) `-2 ** -2`
г) `2 ** 2 ** 3`
д) `2 ** 3 ** 2`
е) `-2 ** 3 ** 2`
ж) `(-2) ** 3 ** 2`
з) `(-2) ** 2 ** 3`

В2.2.3. Определить и объяснить результаты выполнения следующих инструкций.

а) `9 + 6j / 2`
б) `complex(4, 5).conjugate().imag`

в) `complex(0, 3j)`
г) `round(2.5)`
д) `round(-2.5)`
е) `abs(complex(5, -4)) == math.hypot(4,5)`

B2.2.4. Определить значение i^i как действительное число, где $i = \sqrt{(-1)}$.

B2.2.5. Объясните (необычное?) поведение следующего короткого фрагмента исходного кода:

```
>>> d = 8
>>> e = 2
>>> from math import *
>>> sqrt(d ** e)
16.88210319127114
```

B2.2.6. Формально операция целочисленного деления `a // b` определяется как округление в меньшую сторону (floor) результата деления a/b (иногда такое округление записывают как $\lfloor a/b \rfloor$), т. е. наибольшее целое число, меньшее или равное `a / b`. Тогда модуль или остаток от деления `a % b` (также записываемый как $a \bmod b$) равен:

$$a \bmod b = a - b\lfloor a/b \rfloor.$$

Использовать эти определения для получения результата следующих выражений. Проверить результаты в командной оболочке Python.

а) `7 // 4`
б) `7 % 4`
в) `-7 // 4`
г) `-7 % 4`
д) `7 // -4`
е) `7 % -4`
ж) `-7 // -4`
з) `-7 % -4`

B2.2.7. Если две смежные грани правильного шестигранного игрального кубика (кости) имеют значения a и b, если смотреть сбоку и читать слева направо, то значение на верхней грани кубика вычисляется по формуле $3(a^3b - ab^3) \bmod 7$.

Определить значение на верхней грани кубика, если:

а) $a = 2$, $b = 6$;
б) $a = 3$, $b = 5$.

B2.2.8. Сколько раз необходимо сложить пополам лист бумаги (толщина $t = 0.1$ мм, но можно принять и любой другой размер), чтобы достичь Луны (расстояние от Земли $d = 384\,400$ км)?

B2.2.9. Определить результаты вычислений следующих выражений и проверить их, используя командную оболочку Python.

а) `not 1 < 2 or 4 > 2`
б) `not (1 < 2 or 4 > 2)`
в) `1 < 2 or 4 > 2`
г) `4 > 2 or 10/0 == 0`
д) `not 0 < 1`
е) `1 and 2`
ж) `0 and 1`
з) `1 or 0`
и) `type(complex(2, 3).real) is int`

B2.2.10. Объясните, почему при вычислении следующего выражения не получается результат 100.

```
>>> 10^2
8
```

Подсказка: см. документацию Python по битовым операторам (bitwise operators).

Задачи

32.2.1. В Python нет встроенного оператора «исключающее или», но его можно сформировать из существующих операторов. Разработайте два различных способа реализации оператора «исключающее или». Таблица истинности для оператора xor приведена в табл. 2.9.

Таблица 2.9. Таблица истинности для оператора xor

P	Q	P xor Q
True	True	False
False	True	True
True	False	True
False	False	False

32.2.2. Некоторые интересные вычисления с использованием модуля `math`:
а) Какими особенностями обладают числа $\sin(2017 \times \sqrt[5]{2})$ и $(\pi + 20)^i$?
б) Что произойдет, если попытаться вычислить выражение, такое как e^{1000}, генерирующее число, большее, чем максимальное число с плавающей точкой, которое может быть представлено принятым по умолчанию значением с двойной точностью. Что произойдет, если при вычислении ограничиться целочисленной арифметикой (например, при вычислении 1000!)?
в) Что произойдет, если попытаться выполнить неопределенную математическую операцию, например деление на нуль?
г) Максимальное представление числа с плавающей точкой по стандарту IEEE-754 с двойной точностью приблизительно равно 1.8×10^{308}. Вычислить длину гипотенузы прямоугольного треугольника с катетами 1.5×10^{200} и 3.5×10^{201}:
   i) напрямую используя функцию `math.hypot()`;
   ii) без использования этой функции.

32.2.3. В некоторых языках есть функция `sign(a)`, которая возвращает –1, если аргумент `a` отрицательный, и 1, если аргумент `a` положительный. В Python такой функции нет, но в модуль `math` включена функция `math.copysign(x, y)`, которая возвращает абсолютное значение `x` со знаком `y`. Как можно было бы использовать эту функцию способом, аналогичным использованию отсутствующей функции `sign(a)`?

32.2.4. Всемирная геодезическая система (сеть) (World Geodetic System) – это комплект международных стандартов для описания формы Земли. В самой последней версии WGS-84 земной геоид приближенно определяется как эллипсоид, принимающий форму сжатого у полюсов сфероида с главной, или большой, полуосью эллипса a = 6 378 137.0 м и малой полуосью эллипса c = 6 356 752.314245 м.

Использовать формулу вычисления площади поверхности сжатого у полюсов сфероида

$$S_{obl} = 2\pi a^2(1 + ((1 - e^2)/e)\text{atanh}(e)), \text{ где } e^2 = 1 - (c^2/a^2),$$

для вычисления площади поверхности вышеописанного эллипсоида и сравнить полученный результат с площадью поверхности Земли при предположении, что Земля – сфера с радиусом 6371 км.

2.3 Объекты Python I: строки

2.3.1 Определение объекта, представляющего строку

В Python объект, представляющий строку (тип `str`), – это упорядоченная неизменяемая последовательность символов. Для определения переменной, содержащей постоянный текст (строковый литерал – string literal), необходимо взять этот текст в одиночные или двойные кавычки:

```
>>> greeting = "Hello , Sir!"
>>> bye = 'À bientôt'
```

Строки можно объединять (concatenate), используя оператор + или просто размещая их последовательно друг за другом в одной строке:

```
>>> 'abc' + 'def'
'abcdef'
>>> 'one ' 'two' ' three'
'one two three'
```

В Python нет ограничений на длину строки, поэтому строковый литерал можно определить в одном блоке текста, заключенном в кавычки. Но для удобства чтения, как правило, лучше создавать в программе строки, длина которых не превышает определенного максимума (рекомендуется максимальная длина 79 символов). Для размещения текстовой строки в двух и более сроках кода используется символ продолжения строки «обратный слеш» (\), но более удачным решением является заключение строкового литерала в круглые скобки:

```
>>> long_string = 'We hold these truths to be self -evident ,'\
...               ' that all men are created equal...'

>>> long_string = ('We hold these truths to be self -evident ,'
...                ' that all men are created equal...')
```

Здесь определяется переменная `long_string`, содержащая одну строку текста (без символов перехода на новую строку). Операция соединения не вставляет пробелы, поэтому необходимо явно включать их в строку, если это требуется. Пробелы в исходном коде для выравнивания строк по открывающим кавычкам в этом примере не обязательны, они лишь делают код более удобным для чтения.

Если строка состоит из повторяющихся символов (одного или нескольких), то можно использовать оператор `*` для объединения повторяющихся групп символов заданное число раз:

```
>>> 'a' * 4
'aaaa'
>>> '-o-' * 5
'-o--o--o--o--o-'
```

Пустая строка определяется просто как `s = ''` (две одиночные кавычки) или `s = «»`.

Встроенная функция `str` выполняет преобразование объекта, переданного как аргумент, в строку в соответствии с набором правил, определяемых самим исходным объектом:

```
>>> str(42)
'42'
>>> str(3.4e5)
'340000.0'
>>> str(3.4e20)
'3.4e+20'
```

Для получения более точного описания управления форматированием строк, представляющих числа, см. раздел 2.3.7.

Пример П2.8. Строки, объединяемые с помощью оператора `+`, могут повторяться с помощью оператора `*`, но только если они заключены в круглые скобки:

```
>>> ('a'*4 + 'B') * 3
'aaaaBaaaaBaaaaB'
```

2.3.2 Escape-последовательности

Выбор кавычек для определения строк позволяет включать сам символ кавычки в строковый литерал – просто при определении нужно использовать другой символ кавычки:

```
>>> verse = 'Quoth the Raven "Nevermore."'
```

А что, если необходимо включить в строку оба типа кавычек? Или если строковый литерал должен содержать несколько строк текста? В этом случае применяются особые escape-последовательности (или esc-последовательности), обозначаемые символом «обратный слеш» (\). Наиболее часто используемые esc-последовательности перечислены в табл. 2.10.

Таблица 2.10. Часто используемые в Python esc-последовательности

Esc-последовательности	Описание
\'	Одиночная кавычка (')
\"	Двойная кавычка (")
\n	Переход на новую строку (LF)
\r	Возврат каретки (к началу строки) (CR)
\t	Горизонтальная табуляция
\b	Возврат на одно знакоместо со стиранием символа («забой»)
\\	Сам символ «обратный слеш»
\u, \U, \N{}	Символ в кодировке Unicode (см. раздел 2.3.3)
\x	Байт в шестнадцатеричном формате

Примеры:

```
>>> sentence = "He said , \"This parrot's dead.\""
>>> sentence                                                    ❶
'He said , "This parrot\'s dead."'
>>> print(sentence)                                             ❷
He said , "This parrot's dead."
>>> subjects = 'Physics\nChemistry\nGeology\nBiology'
>>> subjects
'Physics\nChemistry\nGeology\nBiology'
>>> print(subjects)
Physics
Chemistry
Geology
Biology
```

❶ Обратите внимание: простой ввод только имени переменной после приглашения командной оболочки Python выводит литеральное значение этой переменной (в кавычках).

❷ Для получения требуемой строки, включающей правильную интерпретацию специальных символов, необходимо передать эту переменную во встроенную функцию `print` (см. раздел 2.3.6).

С другой стороны, если нужно определить строку с включением в нее esc-последовательностей, таких как `\n`, без их экранирования, то определяется необработанная (или неформатируемая) строка (raw string) с префиксом `r`:

```
>>> rawstring = r'The escape sequence for a new line is \n.'
>>> rawstring
'The escape sequence for a new line is \\n.'
>>> print(rawstring)
The escape sequence for a new line is \n.
```

При определении блока текста, включающего несколько символов конца строки (перехода на новую строку), зачастую неудобно все время использовать esc-последовательность `\n`. Этого можно избежать, воспользовавшись строкой в тройных кавычках (triple-quoted string): переходы на новую строку определяются непосредственно в строковом литерале, ограниченном символом `"""` или `'''`, и сохраняются в итоговой строке[19]:

```
a = """one
two
three"»»
>>> print(a)
one
two
three
```

Этот способ часто используется для создания так называемых docstrings, которые документируют блок кода в программе (см. раздел 2.7.1).

Пример П2.9. Esc-последовательность `\x` обозначает символ, закодированный как однобайтовое шестнадцатеричное значение, состоящее из последовательности двух символов. Например, заглавная (прописная) буква N кодируется значением 78 – в шестнадцатеричном формате это значение `4e`. Следовательно:

```
>>> '\x4e'
'N'
```

Невидимый (управляющий) символ backspace (возврат на одно знакоместо с удалением символа) кодируется как шестнадцатеричное значение `08`, поэтому esc-последовательность `\b` равнозначна коду `\x08`:

```
>>> 'hello\b\b\b\b\bgoodbye'
'hello\x08\x08\x08\x08\x08goodbye'
```

При передаче этой строки в функцию `print()` вывод формируется в соответствии с последовательностью символов в этом строковом литерале:

```
>>> print('hello\b\b\b\b\bgoodbye')
goodbye
```

2.3.3 Unicode

Строки в Python 3 состоят из символов в кодировке Unicode. Кодировка Unicode – это стандартное описание более 100 000 символов из почти всех извест-

[19] В общем случае для этой цели рекомендуется использовать три двойные кавычки `"""`.

ных человеческих языков, а также множества других специализированных символов, например научных символов. Каждому символу присвоено числовое значение (код – code point), и эти числовые значения, формирующие строку, затем кодируются как последовательность байтов[20]. В течение длительного времени не существовало официального соглашения об использовании стандарта кодировки, но кодировка UTF-8, применяемая в Python 3 по умолчанию, в настоящее время признана самой широко распространенной и наиболее часто применяемой[21]. Если ваш текстовый редактор не позволяет ввести какой-либо символ напрямую в строковый литерал, то вы можете воспользоваться соответствующим 16-битовым или 32-битовым шестнадцатеричным значением либо именем символа по стандарту Unicode в форме esc-последовательности:

```
>>> '\u00E9'               # 16-битовое шестнадцатеричное значение
'é'
>>> '\u000000E9'           # 32-битовое шестнадцатеричное значение
'é'
>>> '\N{LATIN SMALL LETTER E WITH ACUTE}'          # по имени
'é'
```

Пример П2.10. Предположим, что ваш текстовый редактор или терминал позволяет вводить символы Unicode, и вы можете набирать их на клавиатуре или копировать откуда-либо (например, из веб-браузера или из текстового процессора) и вставлять в строку. Тогда символы Unicode можно вставлять прямо в строковые литералы:

```
>>> creams = 'Crème fraîche, crème brûlée, crème pâtissière'
```

Python даже поддерживает имена переменных с использованием символов Unicode, поэтому идентификаторами могут быть символы, не содержащиеся в наборе ASCII:

```
>>> Σ = 4
>>> crème = 'anglaise'
```

Разумеется, из-за трудностей ввода символов, не содержащихся в наборе ASCII, с обычной клавиатуры, а также из-за внешнего сходства многих различных символов такой способ именования переменных не рекомендуется.

2.3.4 Индексирование и вырезание строк

Индексирование (indexing, или subscripting) строки возвращает один символ из заданной позиции. Как и все последовательности в Python, строки проиндексированы, и первому символу строки присвоен индекс 0. Это означает, что самый последний символ в строке, состоящей из n символов, имеет индекс $n - 1$. Например:

---

[20] Полный список кодов см. на официальном сайте Unicode в таблицах кодов символов: www.unicode.org/charts.

[21] В версию Unicode-кодировки UTF-8 включен «почтенный» набор символов ASCII в 8-битовой кодировке (в котором, например, A = 65).

```
>>> a = "Knight"
>>> a[0]
'K'
>>> a[3]
'g'
```

Символ возвращается в объекте `str` с длиной 1. Неотрицательный индекс отсчитывается в прямом направлении (вправо) от начала строки. Но существует и удобный способ представления индекса для движения по строке в обратном направлении (влево): отрицательный индекс, начиная с `-1` (для конечного символа):

```
>>> a = "Knight"
>>> a[-1]
't'
>>> a[-4]
'i'
```

При попытке индексирования строки за пределами ее длины возникает ошибка (в рассматриваемом здесь примере это индекс больше 5 и меньше –6): Python генерирует сообщение об ошибке `IndexError`:

```
>>> a[6]
Traceback (most recent call last):
File "<stdin >", line 1, in <module >
IndexError: string index out of range
```

Вырезание (slicing) (под)строки `s[i:j]` создает из исходной строки подстроку, расположенную между символами с двумя заданными индексами, включая первый (`i`) символ, но исключая последний (`j`) символ. Если первый индекс не задан, то подразумевается 0 (начало строки). Если второй индекс не задан, то подстрока вырезается до конца. Например:

```
>>> a = "Knight"
>>> a[1:3]
'ni'
>>> a[:3]
'Kni'
>>> a[3:]
'ght'
>>> a[:]
'Knight'
```

На первый взгляд, такой способ кажется немного непривычным, но он гарантирует, что длина подстроки, возвращаемой как `s[i:j]`, равна `j-i` (для положительных значений `i`, `j`), а также что `s[:i] + s[i:] == s`. В отличие от операции индексирования, вырезание строки за границами исходной строки не генерирует ошибку:

```
>>> a = "Knight"
>>> a[3:10]
'ght'
>>> a[10:]
''
```

Чтобы проверить, содержит ли исходная строка заданную подстроку, используется оператор `in`:

```
>>> 'Kni' in 'Knight':
True
>>> 'kni' in 'Knight':
False
```

Пример П2.11. Благодаря самой сущности операции вырезания подстроки `s[m:n]` разность `n-m` всегда представляет длину вырезаемой подстроки. Другими словами, чтобы вернуть `r` символов, начиная с индекса `m`, используйте `s[m:m+r]`. Например:

```
>>> s = 'whitechocolatespaceegg'
>>> s[:5]
'white'
>>> s[5:14]
'chocolate'
>>> s[14:19]
'space'
>>> s[19:]
'egg'
```

Пример П2.12. Третий необязательный параметр операции вырезания подстроки определяет шаг вырезания (stride). Если шаг вырезания не задан, то по умолчанию он равен 1: возвращается каждый символ в указанном диапазоне. Чтобы возвращался каждый *k*-й символ, необходимо задать шаг вырезания *k*. Отрицательные значения *k* изменяют порядок символов в вырезаемой подстроке на противоположный (реверсируют подстроку). Например:

```
>>> s = 'King Arthur'
>>> s[::2]
'Kn rhr'
>>> s[1::2]
'igAtu'
>>> s[-1:4:-1]
'ruhtrA'
```

Последнюю операцию вырезания можно описать как выбор символов, начиная с конечного (индекс `-1`) до символа с индексом `4` (но не включая сам этот символ) с шагом вырезания `-1` (выбирается каждый символ с размещением в обратном порядке).

Удобный способ реверсирования (изменения порядка символов на обратный) всей строки – вырезание между границами по умолчанию (т. е. не задаются первый и последний индексы) с указанием шага вырезания `-1`:

```
>>> s[::-1]
'ruhtrA gniK'
```

2.3.5 Методы обработки строк

В Python строки являются неизменяемыми объектами, поэтому после присваивания строку изменить невозможно. Попытка изменения приведет к ошибке, например:

```
>>> a = 'Knight'
>>> a[0] = 'k'
Traceback (most recent call last):
File "<stdin >", line 1, in <module >
TypeError: 'str' object does not support item assignment
```

Новые строки можно формировать из существующих строк, но только как новые объекты, например:

```
>>> a += ' Templar'
>>> print(a)
Knight Templar
>>> b = 'Black ' + a[:6]
>>> print(b)
Black Knight
```

Чтобы узнать количество символов, содержащихся в строке, используется встроенный метод `len()`:

```
>>> a = 'Earth'
>>> len(a)
5
```

Для строковых объектов существует множество методов обработки и преобразования. Эти методы доступны с помощью обычной точечной нотации, с которой мы уже встречались, – некоторые наиболее часто используемые методы перечислены в табл. 2.11. В этой и в других аналогичных таблицах текст, выделенный *курсивом*, подразумевает замену на конкретное значение, соответствующее применению конкретного метода. Курсивный текст в квадратных скобках обозначает необязательный аргумент.

Таблица 2.11. Некоторые часто применяемые методы

Метод	Описание
`center(`*`width`*`)`	Возвращает строку, отцентрированную в новую строку с общим количеством символов *`width`*
`endswith(`*`suffix`*`)`	Возвращает `True`, если строка заканчивается подстрокой *`suffix`*
`startswith(`*`prefix`*`)`	Возвращает `True`, если строка начинается подстрокой *`prefix`*
`index(`*`substring`*`)`	Возвращает наименьший индекс в строке, соответствующий содержащейся в ней подстроке *`substring`*
`lstrip([`*`chars`*`])`	Возвращает копию строки, в которой удалены все начальные символы, заданные необязательным аргументом [*`chars`*]. Если аргумент [*`chars`*] не задан, то удаляются все начальные пробелы
`rstrip([`*`chars`*`])`	Возвращает копию строки, в которой удалены все конечные (хвостовые) символы, заданные необязательным аргументом [*`chars`*]. Если аргумент [*`chars`*] не задан, то удаляются все конечные пробелы

Окончание табл. 2.11

Метод	Описание
`strip([chars])`	Возвращает копию строки, в которой удалены все начальные и конечные символы, заданные необязательным аргументом [*chars*]. Если аргумент [*chars*] не задан, то удаляются все начальные и конечные пробелы
`upper()`	Возвращает копию строки, в которой все символы переведены в верхний регистр
`lower()`	Возвращает копию строки, в которой все символы переведены в нижний регистр
`title()`	Возвращает копию строки, в которой все слова начинаются с заглавных букв (букв верхнего регистра), а все прочие символы переведены в нижний регистр
`replace(old, new)`	Возвращает копию строки, в которой каждая подстрока *old* заменена подстрокой *new*
`split([sep])`	Возвращает список (см. раздел 2.4.1) подстрок из исходной строки, которые разделены заданной строкой *sep*. Если строка *sep* не задана, то разделителем является любое количество пробельных символов
`join([list])`	Использует строку как разделитель при объединении списка *list* строк
`isalpha()`	Возвращает `True`, если все символы в строке являются алфавитными и строка не пустая, иначе возвращается `False`
`isdigit()`	Возвращает `True`, если все символы в строке являются цифровыми и строка не пустая, иначе возвращается `False`

Поскольку каждый из перечисленных в табл. 2.11 методов возвращает новую строку, методы можно объединять в цепочку вызовов:

```
>>> s = '-+-Python Wrangling for Beginners'
>>> s.lower().replace('wrangling', 'programming').lstrip('+-')
'python programming for beginners'
```

Пример П2.13. Ниже показаны некоторые возможные способы обработки строк с использованием этих методов:

```
>>> a = 'java python c++ fortran '
>>> a.isalpha()
False                                                    ❶
>>> b = a.title()
>>> b
'Java Python C++ Fortran '
>>> c = b.replace(' ', '!\n')
>>> c
'Java!\nPython!\nC++!\nFortran!'
>>> print(c)
Java!
Python!
C++!
Fortran!
>>> c.index('Python')
6                                                        ❷
>>> c[6:]. startswith('Py')
True
>>> c[6:12]. isalpha()
True
```

❶ Метод `a.isalpha()` возвращает `False`, потому что в строке содержатся пробелы и символы ++.

❷ Обратите внимание: `\n` – это один символ.

2.3.6 Функция `print`

В Python 3 `print` является встроенной функцией (как и многие другие, с которыми мы уже знакомы, например `len` и `round`). Она принимает список объектов для вывода и необязательные аргументы `end` и `sep`, определяющие, какие символы должны обозначать конец строки и какие символы должны использоваться для разделения выводимых объектов соответственно. Результатом отсутствия этих дополнительных аргументов будет вывод, в котором поля объектов разделяются одним символом пробела, а строка завершается символом newline (переход на новую строку)[22]. Например:

```
>>> ans = 6
>>> print('Solve:', 2, 'x =', ans, 'for x')
Solve: 2 x = 6 for x
>>> print('Solve: ', 2, 'x = ', ans, ' for x', sep='', end='!\n')
Solve: 2x = 6 for x!
>>> print()                                                          ❶

>>> print('Answer: x =', ans/2)
Answer: x = 3.0
```

❶ Обратите внимание: `print()` без аргументов просто выводит принятый по умолчанию символ перехода на новую строку `end`.

Для отмены перехода на новую строку в конце выводимой строки необходимо для аргумента `end` задать пустую строку: `end=''`:

```
>>> print('A line with no newline character', end='')
A line with no newline character >>>
```

Три символа `>>>` в конце выведенной строки – это приглашение (промпт) для ввода следующей команды Python.

Пример П2.14. Функцию print можно использовать для создания простых текстовых таблиц:

```
>>> heading = '| Index of Dutch Tulip Prices |'
>>> line = '+' + '-'*16 + '-'*13 + '+'
>>> print(line , heading , line ,
...       '|    Nov 23 1636 |        100 |',
...       '|    Nov 25 1636 |        673 |',
...       '|     Feb 1 1637 |       1366 |', line , sep='\n')
...
+----------------------------+
| Index of Dutch Tulip Prices |
+----------------------------+
|    Nov 23 1636 |        100 |
|    Nov 25 1636 |        673 |
|     Feb 1 1637 |       1366 |
+----------------------------+
```

---

[22] Использование специального символа перехода на новую строку newline зависит от операционной системы: например, в macOS это символ `\n` (linefeed – переход на новую строку), в Windows это два символа `\r\n` (carriage return + line feed).

2.3.7 Форматирование строк

Введение в форматирование строк в версии Python 3

Можно воспользоваться строковым методом `format` в его простейшей форме для вставки объектов в строку. Вот пример самого простого синтаксиса:

```
>>> '{} plus {} equals {}'.format(2, 3, 'five')
2 plus 3 equals five
```

Здесь метод `format` вызывается из строкового литерала с аргументами `2`, `3` и `'five'`, которые включаются в заданном порядке в места полей замены (replacement fields), обозначенные парами фигурных скобок `{}`. Поля замены также могут быть пронумерованы или поименованы, что удобно при работе с длинными строками, а еще позволяет несколько раз вставить одно и то же значение:

```
>>> '{1} plus {0} equals {2}'.format(2, 3, 'five')
'3 plus 2 equals five'
>>> '{num1} plus {num2} equals {answer}'.format(num1=2, num2=3, answer='five')
'2 plus 3 equals five'
>>> '{0} plus {0} equals {1}'.format(2, 2+2)
'2 plus 2 equals 4'
```

Обратите внимание: нумерованные поля индексируются, начиная с 0, и могут располагаться в строке в любом порядке.

Для полей замены можно задать минимальный размер в выводимой строке, указав целочисленное значение длины после двоеточия, как показано ниже:

```
>>> '=== {0:12} ==='.format('Python')
'=== Python       ==='
```

Если строка слишком длинна для заданного минимального размера, то будет вставлено столько символов, сколько необходимо (заданный размер поля замены замещается):

```
>>> 'A number: <{0:2}>'.format(-20)
'A number: <-20>' # Для -20 не хватает 2 символов: поэтому используется поле длиной 3
символа
```

По умолчанию вставляемая строка выравнивается по левому краю, но это можно изменить, определив выравнивание по правому краю или по центру. Символы <, > и ^ управляют выравниванием:

```
>>> '=== {0:<12} ==='.format('Python')
'=== Python       ==='
>>> '=== {0:>12} ==='.format('Python')
'===       Python ==='
>>> '=== {0:^12} ==='.format('Python')
'===    Python    ==='
```

В этих примерах поле замены заполняется пробелами, но символ заполнения также можно задать. Например, заполнение дефисами в самом последнем примере:

```
>>> '=== {0:-^12} ==='.format('Python')
'=== ---Python --- ==='
```

Существует даже возможность передачи минимального размера поля как параметра, который сам будет вставлен в подстроку. Просто замените числовой размер поля на ссылку в фигурных скобках, как показано ниже:

```
>>> a = 15
>>> 'This field has {0} characters: ==={1: >{2}}===.'.format(a, 'the field', a)
'This field has 15 characters: ===       the field===.'
```

Или в случае вставки именованных полей:

```
>>> 'This field has {w} characters: ==={0:>{w}}===.'.format('the field', w=a)
'This field has 15 characters: ===      the field===.'
```

В каждом случае второй параметр-определитель формата в этих примерах интерпретируется как `:>15`.

Для вставки самих символов фигурных скобок в форматируемую строку их необходимо продублировать, т. е. использовать {{ и }}.

Форматирование числовых значений

В Python 3 строковый метод format предоставляет разнообразные способы форматирования числовых значений.

Спецификаторы `d`, `b`, `o`, `x`/`X` обозначают десятичный, бинарный, восьмеричный и шестнадцатеричный в нижнем/верхнем регистре форматы целых чисел соответственно:

```
>>> a = 254
>>> 'a = {0:5d}'.format(a) # десятичный формат
'a =   254'
>>> 'a = {0:10b}'.format(a) # бинарный формат
'a =   11111110'
>>> 'a = {0:5o}'.format(a) # восьмеричный формат
'a =   364'
>>> 'a = {0:5x}'.format(a) # шестнадцатеричный формат (в нижнем регистре)
'a =    fe'
>>> 'a = {0:5X}'.format(a) # шестнадцатеричный формат (в верхнем регистре)
'a =    FE'
```

При выводе числа можно дополнять нулями для заполнения полей заданного размера, если перед минимальным размером поля вывода указать `0`:

```
>>> a = 254
>>> 'a = {a:05d}'.format(a=a)
'a = 00254'
```

По умолчанию знак выводится, только если число отрицательное. Это поведение также можно изменить, указывая перед значением минимальной ширины поля следующие символы:

- `+` – всегда выводить знак числа;
- `-` – выводить знак только отрицательного числа (это поведение по умолчанию);

- "" (пробел) – выводить префиксный пробел (вместо знака), только если число положительное.

Последний вариант позволяет правильно выравнивать столбцы положительных и отрицательных чисел:

```
>>> print('{0: 5d}\n{1: 5d}\n{2: 5d}'.format(-4510, 1001, -3026))
-4510
 1001
-3026
>>> a = -25
>>> b = 12
>>> s = '{0:+5d}\n{1:+5d}\n= {2:+3d}'.format(a, b, a+b)
>>> print(s)
  -25
  +12
= -13
```

Также существуют спецификаторы формата для чисел с плавающей точкой, которые можно выводить с выбором требуемой точности. Наиболее часто используются спецификаторы: `f` – обычный формат с плавающей точкой, `e/E` – экспоненциальный (или «научный») формат и `g/G` – общий формат, который применяет научную форму записи для очень больших и очень малых чисел[23]. Требуемая точность (количество десятичных знаков) определяется как `.p` после значения минимальной ширины поля вывода. Несколько примеров:

```
>>> a = 1.464e-10
>>> '{0:g}'.format(a)
'1.464e-10'
>>> '{0:10.2E}'.format(a)
'  1.46E-10'
>>> '{0:15.13f}'.format(a)
'0.0000000001464'
>>> '{0:10f}'.format(a)
'  0.000000'                                                    (1)
```

(1) Обратите внимание: Python не обеспечивает защиту от такого типа округления до нуля, если не предоставлено достаточно места для вывода числа в обычном формате с плавающей точкой.

Форматированные строковые литералы (f-строки)

С версии 3.6 Python поддерживает более развитый способ включения значений в строки: строковый литерал, обозначенный буквой `f` перед открывающей кавычкой, может вычислять выражения, помещенные в фигурные скобки, включая ссылки на переменные, вызовы функций и операции сравнения. Это обеспечивает выразительный и компактный способ определения строковых объектов, например, используя переменные

[23] Более точно: спецификатор `g/G` работает как `f/F` для чисел в диапазоне между 10^{-4} и 10^{p}, где p – требуемая точность (по умолчанию равная 6), а во всех остальных случаях работает как `e/E`.

```
>>> h = 6.62607015e-34
>>> h_units = 'J.s'
```

Вместо применения функции `format`:

```
>>> 'h = {h:.3e} {h_units}'.format(h=h, h_units=h_units)
'h = 6.626e-34 J.s'
```

можно просто написать:

```
>>> f'h = {h:.3e} {h_units}'
'h = 6.626e-34 J.s'
```

Это означает избавление от необходимости неудобного повторения вызова функции `format` (`h=h`, `h_units=h_units`), а длинные строки с многократным включением значений проще читать при использовании описанного выше способа. В общем случае это еще и более быстрый способ выполнения, поскольку такой синтаксис является частью базовой грамматики Python и не требует явных вызовов функций.

Вообще говоря, не самым лучшим решением является размещение сложных выражений в полях замены *f*-строки, тем не менее в *f*-строку достаточно часто включают вызовы функций и операции сравнения:

```
>>> name = 'Elizabeth'
>>> f'The name {name} has {len(name)} letters and {name.lower().count("e")} "e"s.'
'The name Elizabeth has 9 letters and 2 "e"s.'
```

или даже:

```
>>> letter = 'k'
>>> f'{name} has {len(name)} letters and {name.lower().count(letter)} "{letter}"s.'
'Elizabeth has 9 letters and 0 "k"s.'
```

Необходимо помнить о некоторых небольших ограничениях и недостатках: кавычки, используемые внутри выражения *f*-строки, не должны конфликтовать с кавычками, используемыми для ограничения самого строкового литерала (в приведенном выше примере обратите внимание на использование `"`, чтобы избежать конфликта с внешними одиночными кавычками `f'...'`. Кроме того, поскольку *f*-строки вычисляются только один раз во время выполнения, не существует возможности определения многократно используемого «шаблона»:

```
>>> radius = 2.5
>>> s = f'The radius is {radius} m.'
>>> print(s)

The radius is 2.5 m.

>>> radius = 10.3
>>> print(s)

The radius is 2.5 m.
```

Для такого варианта использования лучше применить привычный способ включения значений в строку с помощью функции `format`:

```
>>> radius = 2.5
>>> t = 'The radius is {} m.'
>>> print(t.format(radius))

The radius is 2.5 m.

>>> radius = 10.3
>>> print(t.format(radius))

The radius is 10.3 m.
```

В этой книге будут использоваться оба способа: обычное включение значений в строку с помощью функции `format` и *f*-строки.

Старый способ форматирования в стиле языка C

Python 3 также поддерживает менее удобные спецификаторы формата в стиле языка C, которые все еще широко применяются. В этом варианте форматирования поля замены определяются с помощью спецификаторов минимальной ширины поля и точности, следующих за знаком процента `%`. Объекты, значения которых должны включаться в строку, перечисляются после конца этой строки, и перед ними также должен быть указан знак процента `%`. Если объектов несколько, то они обязательно должны быть взяты в круглые скобки. Для указания различных выводимых типов используются буквы, описанные выше, в предыдущих подразделах. Строки обязательно должны быть явно определены как `"%s"`. Например:

```
>>> kB = 1.380649e-23
>>> 'Here\'s a number: %10.2e' % kB
"Here's a number: 1.38e-23"
>>> 'The same number formatted differently: %7.1e and %12.6e' % (kB, kB)
'The same number formatted differently: 1.4e-23 and 1.380649e-23'
>>> '%s is %g J/K' % ("Boltzmann's constant", kB)
"Boltzmann's constant is 1.38065e-23 J/K"
```

Пример П2.15. Python может обеспечить строковое представление числовых значений, в которых разряды тысяч разделены запятыми:

```
>>> '{:11,d}'.format (1000000)
' 1,000,000'
>>> '{:11,.1f}'.format (1000000.)
'1,000,000.0'
```

Ниже показан пример еще одной таблицы, сформированной с использованием нескольких различных методов обработки строк:

```
title = '|' + '{:^51}'.format('Cereal Yields (kg/ha)') + '|'
line = '+' + '-'*15 + '+' + ('-'*8 + '+')*4
row = '| {:<13} |' + ' {:6,d} |'*4
header = '| {:^13s} |'.format('Country') + (' {:^6d} |'*4).format(1980, 1990, 2000, 2010)
print('+' + '-'*(len(title)-2) + '+',
      title ,
      line ,
      header ,
      line ,
      row.format('China', 2937, 4321, 4752, 5527),
      row.format('Germany', 4225, 5411, 6453, 6718),
      row.format('United States', 3772, 4755, 5854, 6988),
      line ,
      sep='\n')
```

```
+-------------------------------------------------+
|               Cereal Yields (kg/ha)             |
+---------------+--------+--------+--------+------+
| Country       | 1980   | 1990   | 2000   | 2010 |
+---------------+--------+--------+--------+------+
| China         | 2,937  | 4,321  | 4,752  | 5,527  |
| Germany       | 4,225  | 5,411  | 6,453  | 6,718  |
| United States | 3,772  | 4,755  | 5,854  | 6,988  |
+---------------+--------+--------+--------+------+
```

2.3.8 Упражнения

Вопросы

B2.3.1. Выполнить вырезание из строки `s = 'seehemewe'`, чтобы получить следующие подстроки:

а) `'see'`
б) `'he'`
в) `'me'`
г) `'we'`
д) `'hem'`
е) `'meh'`
ж) `'wee'`

B2.3.2. Написать однострочное выражение, определяющее, является ли строка палиндромом (т. е. читается одинаково в обоих направлениях).

B2.3.3. Определить результаты вычислений следующих выражений и проверить их, используя командную оболочку Python.

```
>>> days = 'Sun Mon Tues Weds Thurs Fri Sat'

а) print(days[days.index('M'):])
б) print(days[days.index('M'):days.index('Sa')].rstrip())
в) print(days[6:3:-1].lower()*3)
г) print(days.replace('rs', '').replace('s ', ' ')[::4])
д) print(' -*- '.join(days.split()))
```

B.2.3.4. Что выводит следующий фрагмент кода? Объяснить, как это работает.

```
>>> suff = 'thstndrdthththththththth'
>>> n = 1
>>> print('{:d}{:s}'.format(n, suff[n*2:n*2+2]))
>>> n = 3
>>> print('{:d}{:s}'.format(n, suff[n*2:n*2+2]))
>>> n = 5
>>> print('{:d}{:s}'.format(n, suff[n*2:n*2+2]))
```

B.2.3.5. Рассмотреть следующие (неправильные) проверки, пытающиеся определить, имеет ли строка `s` одно из двух значений. Описать, как эти инструкции интерпретируются Python, и предложить правильный вариант проверки.

```
>>> s = 'eggs'
>>> s == ('eggs' or 'ham')
True

>>> s == ('ham' or 'eggs')
False
```

Задачи

32.3.1.

а) Имеется строка, представляющая последовательность пар оснований (т. е. содержащие только буквы A, G, C, T). Определить доли (проценты) оснований G и C в этой последовательности.
(Совет: для строк существует метод `count`, возвращающий количество найденных вхождений заданных подстрок.)

б) Используя лишь методы обработки строк, разработать способ, позволяющий определить, является ли нуклеотидная последовательность палиндромом в том смысле, что она равнозначна собственной комплементарной последовательности, читаемой в обратном порядке. Например, последовательность TGGATCCA является палиндромом, так как соответствующая ей комплементарная последовательность ACCTAGGT при прочтении в обратном порядке совпадает с исходной последовательностью. Комплементарными парами оснований являются (A, T) и (C, G).

32.3.2. В приведенной ниже табл. 2.12 перечислены названия, символические обозначения, числовые значения, погрешности и единицы измерения некоторых физических констант.

Таблица 2.12

Название	Символ	Значение	Погрешность	Единицы измерения
Постоянная Больцмана	k_B	1.380649×10^{-23}	(def)	$J\ K^{-1}$
Скорость света	c	2.99792458×10^{8}	(def)	$m\ s^{-1}$
Постоянная Планка	h	6.62607015×10^{-34}	(def)	$J\ s$
Число Авогадро	N_A	6.02214076×10^{23}	(def)	mol^{-1}
Магнитный момент электрона	μ_e	$-9.28476377\times10^{-24}$	2.3×10^{-31}	$J\ T^{-1}$
Гравитационная постоянная	G	6.67430×10^{-11}	1.5×10^{-15}	$N\ m^2\ kg^{-2}$

Определяя переменные в форме

```
G = 6.6743e-11        # J/K
G_unc = 1.5e-15       # погрешность
G_units = 'Nm^2/kg^2'
```

использовать метод строкового объекта format для получения следующих вариантов вывода:

а) kB = 1.381e-23 J/K;
б) G = 0.0000000000667430 Nm^2/kg^2;
в) использовать одинаковый спецификатор формата для каждой строки:

```
kB = 1.3807e-23 J/K
mu_e = -9.2848e-24 J/T
N_A = 6.0221e+23 mol -1
c = 2.9979e+08 m/s
```

г) и еще раз использовать одинаковый спецификатор формата для каждой строки:

```
=== G = +6.67E-11 [Nm^2/kg^2] ===
=== e = -9.28E-24 [ J/T] ===
```

Подсказка: код Unicode для греческой буквы мю в нижнем регистре `U+03BC`.

д) (Более сложная задача.) Получить показанный ниже вывод, в котором погрешность (стандартное отклонение) в значении каждой константы выражается как число в круглых скобках после предшествующих числовых разрядов, т. е. $6.67430(15)\times10^{-11}$ означает $6.67430\times10^{-11} \pm 1.5\times10^{-15}$.

```
G = 6.67430(15)e-11 Nm^2/kg^2
mu_e = -9.28476377(23)e-24 J/T
```

32.3.3. Обозначив элементы матрицы 3×3 как девять переменных `a11`, `a12`, ..., `a33`, получить строковое представление этой матрицы, используя методы форматирования: а) предполагая, что элементы матрицы являются действительными (возможно, отрицательными) числами с одним знаком после десятичной точки; б) предполагая, что это матрица перестановок с целочисленными элементами, принимающими только значения 0 или 1. Например:

```
>>> print(s_a)
[ 0.0 3.4 -1.2 ]
[ -1.1 0.5 -0.2 ]
[ 2.3 -1.4 -0.7 ]
>>> print(s_b)
[ 0 0 1 ]
[ 0 1 0 ]
[ 1 0 0 ]
```

32.3.4. Найти коды Unicode для обозначения символов планет, перечисленных на сайте НАСА (https://solarsystem.nasa.gov/resources/680/solar-system-symbols/). Эти символы в основном расположены в шестнадцатеричном диапазоне 2600–26FF: Miscellaneous Symbols (Различные символы) (https://www.unicode.org/charts/PDF/U2600.pdf). Вывести список названий планет с соответствующими символами.

2.4 Объекты Python II: списки, кортежи и циклы

2.4.1 Списки

Инициализация и индексация списков

Python предоставляет структуры данных для хранения упорядоченного списка объектов. В некоторых других языках (например, C и Fortran) такая структура данных называется массивом (array) и может содержать только данные одного типа (например, массив целых чисел). Но в Python базовые структуры массивов могут хранить данные различных типов.

В Python список (list) – это упорядоченный изменяемый (mutable) массив объектов. Список создается из заданных объектов, разделенных запятыми, и помещается между квадратными скобками `[ ]`. Например:

```
>>> list1 = [1, 'two', 3.14, 0]
>>> list1
[1, 'two', 3.14, 0]
>>> a = 4
>>> list2 = [2, a, -0.1, list1 , True]
>>> list2
[2, 4, -0.1, [1, 'two', 3.14, 0], True]
```

Следует отметить, что в Python список может содержать ссылки на объект любого типа: строки, разнообразные типы чисел, встроенные константы, такие как логическое значение `True`, и даже ссылки на другие списки. Нет необходимости предварительно объявлять размер списка перед его использованием. Можно создать пустой список: `list0 = []` или `list0 = list()`.

Любой элемент можно извлечь из списка по индексу (напомню: в Python индексация начинается с 0):

```
>>> list1[2]
3.14
>>> list2[-1]
True
>>> list2[3][1]
'two'
```

В последнем примере извлекается второй (индекс: `1`) элемент из четвертого (индекс: `3`) элемента списка `list2`. Это допустимая операция, поскольку элемент `list2[3]` сам является списком (его можно также идентифицировать по

имени переменной list1), а элемент list1[1] – это строка 'two'. А поскольку строки тоже можно индексировать, то:

```
>>> list2[3][1][1]
'w'
```

Для проверки наличия элемента в списке используется оператор in, как и для строк:

```
>>> 1 in list1
True
>>> 'two' in list2:
False
```

Для последнего выражения вычисляется результат False, потому что список list2 не содержит строковый литерал 'two', хотя в этом списке содержится список list1, в котором есть этот литерал: оператор in не выполняет рекурсию по спискам списков при проверке наличия элемента в списке.

Списки и свойство изменяемости

В Python список – первый изменяемый (mutable) объект, который встретился нам. В отличие от строк, которые нельзя изменить после определения, списку можно присваивать другие элементы:

```
>>> list1
[1, 'two', 3.14, 0]
>>> list1[2] = 2.72
>>> list1
[1, 'two', 2.72, 0]
>>> list2
[2, 4, -0.1, [1, 'two', 2.72, 0], True]
```

Обратите внимание: изменился не только список list1, но и список list2 (который содержит list1 как элемент)[24]. Это поведение многие разработчики обнаруживают, когда начинают работать со списками особенно, если требуется копирование списка в другую переменную.

```
>>> q1 = [1, 2, 3]
>>> q2 = q1
>>> q1[2] = 'oops'
>>> q1
[1, 2, 'oops']
>>> q2
[1, 2, 'oops']
```

Здесь переменные q1 и q2 ссылаются на один и тот же список, хранящийся в одной локации памяти, а поскольку списки являются изменяемыми, инструкция q1[2] = 'oops' действительно изменяет одно из значений, хранящих-

[24] В действительности этот список не изменился: в нем содержится только последовательность ссылок на объекты: ссылка на list1 остается той же самой, даже если были изменены ссылки внутри списка list1.

ся в этой локации. Список q2 продолжает указывать на ту же локацию, поэтому также отображает все изменения в ней. В действительности здесь существует только один список (на который ссылаются две переменные с различными именами), и он изменяется один раз. В противоположность списку целые числа являются неизменяемыми (immutable), поэтому показанный ниже фрагмент кода не изменяет значение в списке q[2]:

```
>>> a = 3
>>> q = [1, 2, a]
>>> a = 4
>>> q
[1, 2, 3]
```

Операция присваивания a = 4 создает абсолютно новый целочисленный объект, полностью независимый от исходного значения 3, находящегося в конце списка q. Этот исходный целочисленный объект не изменяется в результате присваивания (целые числа неизменяемы), поэтому и список остается неизменным. Это различие показано на рис. 2.2, 2.3 и 2.4.

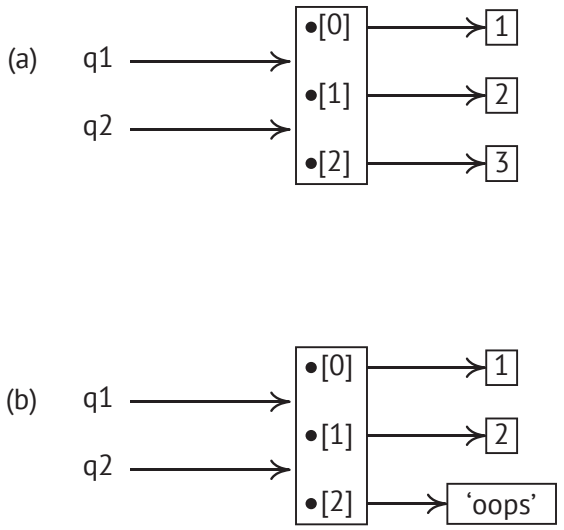


Рис. 2.2. Две переменные, ссылающиеся на один и тот же список: а) при инициализации; б) после присваивания q1[2] = 'oops'

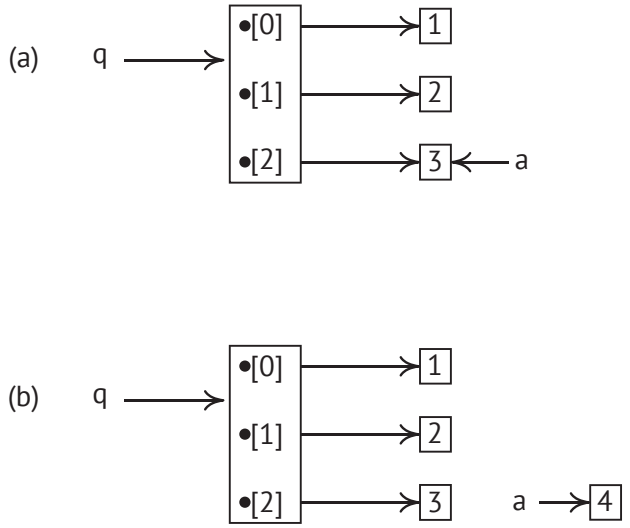


Рис. 2.3. Список, определенный как `q = [1, 2, a]`, где `a = 3`: а) при инициализации; б) после изменения значения посредством присваивания `a = 4`

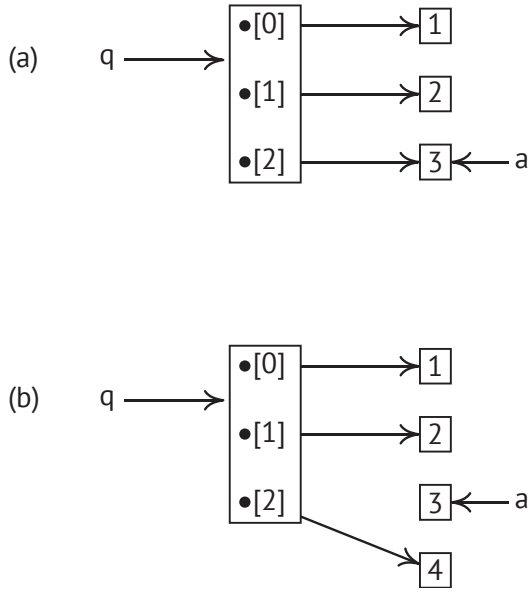


Рис. 2.4. Список, определенный как `q = [1, 2, a]`, где `a = 3`: а) при инициализации; б) после изменения значения в списке посредством присваивания `q[2] = 4`

Из списков можно вырезать (slice) группу элементов тем же способом, как это выполнялось в строковых последовательностях:

```
>>> q1 = [0., 0.1, 0.2, 0.3, 0.4, 0.5]
>>> q1[1:4]
[0.1, 0.2, 0.3]
>>> q1[::-1] # возвращает реверсированную копию списка (элементы списка в обратном порядке)
[0.5, 0.4, 0.3, 0.2, 0.1, 0.0]
>>> q1[1::2] # выборочное копирование с заданным шагом: возвращаются элементы
с индексами 1, 3, 5
[0.1, 0.3, 0.5]
```

Операция вырезания элементов копирует данные в новый список. Таким образом:

```
>>> q2 = q1[1:4]
>>> q2[1] = 99       # воздействует только на список q2
>>> q2
[0.1, 99, 0.3]
>>> q1
[0.0, 0.1, 0.2, 0.3, 0.4, 0.5]
```

Методы списка

Как и для строк, для списков Python существует множество полезных методов, перечисленных в табл. 2.13. Так как объекты типа `list` являются изменяемыми, они могут увеличивать или уменьшать свой размер без необходимости копирования содержимого в новый объект, как это происходило со строками.

Таблица 2.13. Некоторые часто используемые методы списков

Метод	Описание
`append(`*`element`*`)`	Добавляет *`element`* в конец списка
`extend(`*`list2`*`)`	Расширяет список с использованием элементов из списка *`list2`*
`index(`*`element`*`)`	Возвращает наименьший индекс списка, содержащего *`element`*
`insert(`*`index, element`*`)`	Вставляет в список *`element`* по индексу *`index`*
`pop()`	Удаляет и возвращает самый последний элемент из списка
`reverse()`	Изменяет порядок элементов списка на обратный (реверсирует список)
`remove(`*`element`*`)`	Удаляет первое вхождение *`element`* из списка
`sort()`	Сортирует список
`copy()`	Возвращает копию списка
`count(`*`element`*`)`	Возвращает количество элементов, равных *`element`*, в списке

Ниже перечислены наиболее важные методы списков:

- `append` – добавляет элемент в конец списка;
- `extend` – добавляет один или несколько объектов, копируя их из другого списка[25];

[25] В действительности любой объект Python, образующий последовательность, по которой можно организовать итеративный проход (например, строка), может использоваться как аргумент для метода `extend`.

- `insert` – вставляет элемент по заданному индексу;
- `remove` – удаляет заданный элемент из списка.

```
>>> q = []
>>> q.append(4)
>>> q
[4]
>>> q.extend([6, 7, 8])
>>> q
[4, 6, 7, 8]
>>> q.insert(1, 5)     # вставка 5 по индексу 1
>>> q
[4, 5, 6, 7, 8]
>>> q.remove(7)
>>> q
[4, 5, 6, 8]
>>> q.index(8)
3   # элемент 8 расположен по индексу 3
```

Два полезных метода списков – `sort()` и `reverse()`: первый сортирует элементы списка, второй размещает элементы в обратном порядке (реверсирует список). Таким образом, оба этих метода изменяют сам объект списка, но не возвращают значение:

```
>>> q = [2, 0, 4, 3, 1]
>>> q.sort()
>>> q
[0, 1, 2, 3, 4]
>>> q.reverse()
>>> q
[4, 3, 2, 1, 0]
```

Если необходима отсортированная копия списка с сохранением без изменений исходного списка, то можно воспользоваться встроенной функцией `sorted()`:

```
>>> q = ['a', 'e', 'A', 'c', 'b']
>>> sorted(q)
['A', 'a', 'b', 'c', 'e'] # возвращает новый список
>>> q
['a', 'e', 'A', 'c', 'b'] # исходный список остается неизменным
```

По умолчанию `sort()` и `sorted()` располагают элементы в любом массиве в возрастающем порядке. Для размещения элементов в убывающем порядке определяется необязательный аргумент `reverse=True`:

```
>>> q = [10, 5, 5, 2, 6, 1, 67]
>>> sorted(q, reverse=True)
[67, 10, 6, 5, 5, 2, 1]
```

В Python не разрешается напрямую сравнивать между собой строки и числовые значения, поэтому при попытке сортировки списка, содержащего различные типы данных (числа и строки), возникает ошибка:

```
>>> q = [5, '4', 2, 8]
>>> q.sort()
TypeError: unorderable types: str() < int()
```

Пример П2.16. Методы `append()` и `pop()` существенно упрощают использование списка для реализации структуры данных, известной как стек (stack):

```
>>> stack = []
>>> stack.append(1)
>>> stack.append(2)
>>> stack.append(3)
>>> stack.append(4)
>>> print(stack)
[1, 2, 3, 4]
>>> stack.pop()
4
>>> print(stack)
[1, 2, 3]
```

Конец списка представляет собой вершину стека, в которую можно добавлять и из которой можно удалять элементы (принцип «последним вошел, первым вышел» – LIFO: можно интерпретировать стек как стопку тарелок).

Пример П2.17. Строковый метод `split()` генерирует список подстрок из заданной строки, разделяя их по заданному символу-разделителю:

```
>>> s = 'Jan Feb Mar Apr May Jun'
>>> s.split()        # По умолчанию разделение по любым пробельным символам
['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun']
>>> s = "J. M. Brown AND B. Mencken AND R. P. van't Rooden"
>>> s.split(' AND ')
['J. M. Brown', 'B. Mencken', "R. P. van't Rooden"]
```

2.4.2 Кортежи

Объект *tuple*

Кортеж `tuple` можно воспринимать как неизменяемый список. Кортеж создается посредством размещения элементов внутри круглых скобок:

```
>>> t = (1, 'two', 3.)
>>> t
(1, 'two', 3.0)
```

Кортежи можно индексировать и вырезать из них элементы, как в списках, но поскольку кортежи неизменяемы, в них нельзя добавлять и удалять элементы, а также расширять кортежи:

```
>>> t = (1, 'two', 3.)
>>> t[1]
'two'
>>> t[2] = 4
Traceback (most recent call last):
File "<stdin >", line 1, in <module >
TypeError: 'tuple' object does not support item assignment
```

Хотя сами кортежи неизменяемы, они могут содержать ссылки на изменяемые объекты, такие как списки. Таким образом:

```
>>> t = (1, ['a', 'b', 'd'], 0)
>>> t[1][2] = 'c'    # Внутри кортежа можно изменять элементы содержащегося в нем списка.
>>> t
(1, ['a', 'b', 'c'], 0)
```

Пустой кортеж создается с помощью пары круглых скобок: `t0 = ()`. Но для создания кортежа, содержащего только один элемент (синглтон – singleton), недостаточно просто взять этот элемент в скобки (так как это можно перепутать с другими вариантами использования синтаксиса круглых скобок). Вместо этого после данного одного элемента обязательно должна быть указана завершающая запятая: `t = ('one' ,)`.

Использование кортежей

При некоторых условиях, в частности при простых присваиваниях, подобных показанным в предыдущем подразделе, круглые скобки, окружающие элементы кортежа, не обязательны:

```
>>> t = 1, 2, 3
>>> t
(1, 2, 3)
```

Такое использование представляет собой пример упаковки кортежа (tuple packing). Противоположное действие – распаковка кортежа (tuple unpacking) – это общеизвестный способ присваивания значений нескольким переменным в одной строке:

```
>>> a, b, c = 97, 98, 99
>>> b
98
```

Этот метод присваивания значений нескольким переменным часто используется вместо раздельных инструкций присваивания в отдельных строках или в одной строке с разделением символами точки с запятой (это в высшей степени анти-Python-стиль присваивания):

```
a = 97; b = 98; c = 99            # Не делайте так.
```

Кортежи удобны, когда последовательность элементов не должна изменяться. В предыдущем примере объект `tuple` существует только с тем порядком элементов, который задан во время присваивания значений элементам `a`, `b` и `c`.

Присваиваемые значения 97, 98 и 99 упаковываются в кортеж для достижения цели этой операции присваивания (для распаковки в указанные переменные), но сразу после выполнения присваивания объект кортежа уничтожается. Другой пример: функция (см. раздел 2.7) может возвращать более одного объекта: эти возвращаемые объекты упаковываются в кортеж. Если необходимо более существенное обоснование, то можно добавить, что кортежи во многих вариантах использования быстрее, чем списки.

Пример П2.18. При выполнении операции присваивания выражение справа от оператора = вычисляется в первую очередь. Это обеспечивает удобный способ обмена значениями между двумя переменными с использованием кортежей:

```
a, b = b, a
```

Здесь правая часть упаковывается в объект кортежа, который затем распаковывается в переменные, указанные в левой части. Это более удобно, чем использование временной переменной:

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

2.4.3 Итерируемые объекты

Примеры итерируемых объектов

Строки, списки и кортежи – все это примеры структур данных, представляющие собой объекты, в которых существует возможность организации итеративного прохода: это упорядоченные последовательности элементов (символов в строках или произвольных объектов в списках и кортежах), которые можно брать по одному за одну итерацию. Одним из способов наблюдения за этим процессом является использование альтернативного метода инициализации списка (или кортежа) с применением встроенных методов-конструкторов `list()` и `tuple()`. Эти методы принимают любой итерируемый объект и генерируют список или кортеж соответственно из предоставленной последовательности элементов. Например:

```
>>> list('hello')
['h', 'e', 'l', 'l', 'o']
>>> tuple([1, 'two', 3])
(1, 'two', 3)
```

Поскольку элементы данных копируются при создании нового объекта с использованием этих методов-конструкторов, `list()` является еще одним способом создания независимого объекта списка из другого объекта:

```
>>> a = [5, 4, 3, 2, 1]
>>> b = a         # b и a ссылаются на один и тот же объект списка
>>> b is a
True
>>> b = list(a)   # создание совершенно нового объекта списка с тем же содержимым,
                  # что и список a
>>> b is a
False
```

Поскольку операция вырезания также возвращает копию ссылок на объекты из последовательности, особый прием `b = a[:]` применяется чаще, чем `b = list(a)`.

any и *all*

Встроенная функция `any()` проверяет каждый элемент в итерируемом объекте на равенство логическому значению `True`, а встроенная функция `all()` проверяет все элементы на равенство `True`. Например:

```
>>> a = [1, 0, 0, 2, 3]
>>> any(a), all(a)
(True , False) # некоторые (но не все) элементы списка a равны True
>>> b = [[], False , 0.]
>>> any(b), all(b)
(False , False) # ни один из элементов списка b не равен True
```

◊ Синтаксис *

Иногда необходимо вызвать функцию с аргументами, выбираемыми из списка или любой другой последовательности. Синтаксис *, используемый при вызове функции, распаковывает такую последовательность в позиционные аргументы, передаваемые в функцию (также см. раздел 2.7). Например, функция `math.hypot` принимает два аргумента `a` и `b` и возвращает числовое значение $\sqrt{a^2 + b^2}$. Если аргументы, которые необходимо использовать, размещены в списке или в кортеже, то возникает следующая ошибка:

```
>>> t = [3, 4]
>>> math.hypot(t)
Traceback (most recent call last):
  File "<stdin >", line 1, in <module >
TypeError: hypot expected 2 arguments , got 1
```

Здесь выполнена попытка вызова функции `math.hypot()` с одним аргументом (объект списка `t`), что привело к ошибке. Можно было бы явно использовать индексы списка для извлечения двух требуемых значений:

```
>>> t = [3, 4]
>>> math.hypot(t[0], t[1])
5.0
```

Но есть более изящный способ – распаковка списка в аргументы этой функции с применением синтаксиса `*t`:

```
>>> math.hypot(*t)
5.0
```

Циклы *for*

Часто возникает необходимость в последовательном переборе элементов (по одному) итерируемого объекта и выполнении каких-либо действий с каждым отдельным элементом по очереди. В других языках, таких как C, этот тип цикла (loop) служит для обращения к каждому элементу по отдельности по соответствующему целочисленному индексу. В Python это также возможно, но существует и более естественный и удобный способ с использованием специального приема:

```
for item in iterable_object:
```

Здесь каждый элемент итерируемого объекта поочередно передается для обработки в последующий блок кода. Например:

```
>>> fruit_list = ['apple', 'melon', 'banana', 'orange']
>>> for fruit in fruit_list:
...     print(fruit)
...
apple
melon
banana
orange
```

Каждый элемент в объекте списка `fruit_list` поочередно выбирается и присваивается переменной `fruit` для передачи в блок инструкций, следующих за символом двоеточия (`:`) – каждая инструкция в этом блоке кода обязательно должна быть смещена вправо на одинаковое количество позиций (с помощью пробелов). Можно использовать любое количество пробелов или символов табуляции, но настоятельно рекомендуется использовать четыре пробела для форматирования строк кода[26].

Циклы могут быть вложенными – блок кода внутреннего цикла должен сдвигаться на то же количество пробелов относительно блока кода внешнего цикла (т. е. на восемь пробелов):

```
>>> fruit_list = ['apple', 'melon', 'banana', 'orange']
>>> for fruit in fruit_list:
...     for letter in fruit:
...         print(letter , end='.')
...     print()
...
a.p.p.l.e.
m.e.l.o.n.
b.a.n.a.n.a.
o.r.a.n.g.e.
```

[26] Использование пробелов (пробельных символов) как части синтаксиса Python – одна из наиболее спорных особенностей этого языка. Некоторые программисты, пользующиеся такими языками, как C и Java, в которых блоки кода выделяются с помощью фигурных скобок {...}, считают отступы «проклятием». Другие программисты придерживаются более уравновешенной точки зрения и отмечают, что код почти всегда форматируется логически согласованно, чтобы сделать его более удобным для чтения, даже если это не является обязательным требованием грамматики языка.

В этом примере выполняется итеративный проход по строковым элементам списка `fruit_list` с поочередным перебором их по одному, и для каждой строки (название фрукта) выполняется итеративный проход по отдельным буквам. Каждая буква выводится со следующей за ней точкой, обозначающей конец строки без перехода на новую строку (это код в теле внутреннего цикла). Самая последняя инструкция внешнего цикла `print()` выполняет принудительный переход на новую строку после вывода всех букв из очередного названия фрукта.

Пример П2.19. Выше уже кратко рассматривался строковый метод `join()`, принимающий последовательность строковых объектов и объединяющий их в одну строку:

```
>>> ', '.join( ('one', 'two', 'three') )
'one, two, three'
>>> print('\n'.join(reversed(['one', 'two', 'three'])))
three
two
one
```

Встроенная функция `reversed()` выполняет итерационный проход по последовательности в обратном порядке при том преимуществе (для длинных последовательностей), что эта функция не создает новый объект и не изменяет исходный объект.

Напомню, что сами по себе строки являются итерируемыми последовательностями, поэтому могут передаваться в метод `join()`. Например, для соединения букв из строки `'hello'` с помощью одиночных пробелов:

```
>>> ' '.join('hello')
'h e l l o'
```

Тип range

Python предоставляет эффективный метод ссылки на последовательность числовых значений, которые образуют простую арифметическую прогрессию: $a_n = a_0 + nd$ при $n = 0, 1, 2, \ldots$. В такой последовательности каждый ее член отличается от предыдущего на определенное постоянное значение – шаг (stride) d. В самом простом случае необходим лишь целочисленный счетчик, который работает с единичным шагом, а начальным значением является ноль: $0, 1, 2, \ldots, N - 1$. Существует возможность создания списка, содержащего каждое из этих значений, но для большинства целей потребуется слишком большой объем памяти, поэтому проще генерировать следующее число в последовательности без необходимости одновременного хранения всех чисел.

Представление таких арифметических прогрессий для итеративных проходов – предназначение типа `range`. Объект типа `range` можно создать посредством передачи от одного до трех аргументов, определяющих начальное целое число, конечное целое число и шаг (который может быть отрицательным).

```
range([a0=0], n, [stride=1])
```

Приведенный здесь формат вызова конструктора `range` означает, что если не задано начальное значение `a0`, то по умолчанию принимается значение `0`. Шаг `stride` также не является обязательным аргументом, и при его отсутствии по умолчанию определен шаг `1`. Несколько примеров:

```
>>> a = range(5)            # 0, 1, 2, 3, 4
>>> b = range(1, 6)         # 1, 2, 3, 4, 5
>>> c = range(0, 6, 2)      # 0, 2, 4
>>> d = range(10, 0, -2)    # 10, 8, 6, 4, 2
```

В Python 3 объект, создаваемый конструктором `range`, не является списком. Это итерируемый объект, который может создавать целые числа по запросу: объекты типа `range` можно индексировать, преобразовывать в списки и кортежи и выполнять по ним итерационный проход:

```
>>> c[1]        # т. е. второй элемент последовательности 0, 2, 4
2
>>> c[0]
0
>>> list(d)     # создание списка из объекта типа range
[10, 8, 6, 4, 2]
>>> for x in range(5):
... print(x)
0
1
2
3
4
```

Пример П2.20. Ряд Фибоначчи – это последовательность чисел, генерируемых с применением следующих правил:

$$a_1 = a_2 = 1, a_i = a_{i-1} + a_{i-2}.$$

Таким образом, *i*-е число Фибоначчи является суммой двух предыдущих чисел этой последовательности: 1, 1, 2, 3, 5, 8, 13, ...

Ниже представлены два способа генерации последовательности Фибоначчи. В первом способе применяется список (см. листинг 2.1).

Листинг 2.1. Вычисление последовательности Фибоначчи с помощью списка

```
# eg2-i-fibonacci.py
# Вычисляет и сохраняет первые n чисел Фибоначчи.

n = 100
fib = [1, 1]
for i in range(2, n+1):
    fib.append(fib[i-1] + fib[i-2])
print(fib)
```

В другом способе можно генерировать ту же последовательность без необходимости одновременного хранения более чем двух чисел, как показано в листинге 2.2.

Листинг 2.2. Вычисление последовательности Фибоначчи без сохранения всех чисел

```
# eg2-ii-fibonacci.py
# Вычисление первых n чисел Фибоначчи.

n = 100
# Постоянное отслеживание двух самых последних чисел Фибоначчи.
a, b = 1, 1
print(a, b, end='')
for i in range(2, n+1):
    # Следующее число (b) равно a+b; затем a становится предыдущим значением b.
    a, b = b, a+b
    print(' ', b, end='')
```

Метод *enumerate*

Поскольку объекты типа `range` можно использовать для генерации последовательности целых чисел, возникает соблазн воспользоваться ими для создания индексов списков и кортежей при итеративном проходе в цикле `for`:

```
>>> mammals = ['kangaroo', 'wombat', 'platypus']
>>> for i in range(len(mammals)):
        print(i, ':', mammals[i])
0 : kangaroo
1 : wombat
2 : platypus
```

Разумеется, этот способ будет работать, но существует более правильное решение: исключить явное создание объекта типа `range` (и вызов встроенной функции `len`) и воспользоваться методом `enumerate`. Метод `enumerate` принимает любой итерируемый объект и создает поочередно для каждого элемента кортеж `(count, item)`, состоящий из счетчика индекса и самого элемента:

```
>>> mammals = ['kangaroo', 'wombat', 'platypus']
>>> for i, mammal in enumerate(mammals):
        print(i, ':', mammal)
0 : kangaroo
1 : wombat
2 : platypus
```

Обратите внимание: каждый кортеж `(count, item)` распаковывается в цикле `for` в переменные `i` и `mammal`. Кроме того, есть возможность установить начальное значение счетчика `count`, отличающееся от 0 (хотя в дальнейшем это не повлияет на значение индекса элемента в исходном списке, разумеется):

```
>>> list(enumerate(mammals , 4))
[(4, 'kangaroo'), (5, 'wombat'), (6, 'platypus')]
```

◊ Встроенная функция *zip*

Что, если необходимо выполнить итеративный проход одновременно по двум (и более) последовательностям? Именно для этого предназначена встроенная функция `zip`: она создает объект-итератор, в котором каждый элемент представляет собой кортеж из элементов, поочередно выбираемых из переданных в функцию последовательностей:

```
>>> a = [1, 2, 3, 4]
>>> b = ['a', 'b', 'c', 'd']
>>> zip(a, b)
<builtins.zip at 0x104476998 >
>>> for pair in zip(a, b):
...     print(pair)
...
(1, 'a')
(2, 'b')
(3, 'c')
(4, 'd')
>>> list(zip(a, b))      # преобразование в список
[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd')]
```

Замечательным свойством функции `zip` является возможность ее использования еще и для разделения (unzip) последовательности кортежей элементов, как показано ниже:

```
>>> z = zip(a, b)       # соединение (zip)
>>> A, B = zip(*z)      # разделение (unzip)
>>> print(A, B)
(1, 2, 3, 4) ('a', 'b', 'c', 'd')
>>> list(A) == a, list(B) == b
(True , True)
```

Функция `zip` не копирует элементы в новый объект, поэтому она эффективно использует память и работает быстро, но это означает, что вам предоставляется только лишь возможность однократного итеративного прохода по объединенным элементам, однако вы не можете обращаться к ним по индексам:

```
>>> z = zip(a, b):
>>> z[0]
TypeError: 'zip' object is not subscriptable

>>> for pair in z:
...     x = 0   # Это всего лишь некоторая фиктивная операция, выполняемая на каждой итерации.
...
>>> for pair in z:
...     print(pair)
...
        # (Ничего не происходит: действие итератора z уже закончилось.)
>>>
```

2.4.4 Упражнения

Вопросы

B2.4.1. Определить и объяснить вывод результатов выполнения следующих инструкций с использованием переменных `s = 'hello'` и `a = [4, 10, 2]`.

```
а) print(s, sep='-')
б) print(*s, sep='-')
в) print(a)
г) print(*a, sep='')
д) list(range(*a))
```

B2.4.2. Список `list` можно использовать как простое представление многочлена $P(x)$ с элементами, являющимися коэффициентами для последовательных степеней x, и индексами, являющимися самими показателями степеней. Таким образом, многочлен $P(x) = 4 + 5x + 2x^3$ должен быть представлен списком `[4, 5, 0, 2]`. Почему показанная ниже попытка дифференцирования многочлена не привела к вычислению правильного ответа?

```
>>> P = [4, 5, 0, 2]
>>> dPdx = []
>>> for i, c in enumerate(P[1:]):
...     dPdx.append(i*c)
>>> dPdx
[0, 0, 4]            # Ответ неправильный.
```

Как можно исправить этот код?

B2.4.3. Имеется упорядоченный список экзаменационных оценок. Создать список, связывающий каждую оценку с определенной категорией (rank) (начиная с 1 для самой высокой оценки). Равные оценки должны соответствовать одной категории. Например, для исходного списка `[87, 75, 75, 50, 32, 32]` должен быть создан следующий список категорий `[1,2,2,4,5,5]`.

B2.4.4. Использовать цикл `for` для вычисления числа π по первым 20 членам ряда Мадхавы–Лейбница:

$$\pi = \sqrt{12}\left(1 - \frac{1}{3 \cdot 3} + \frac{1}{5 \cdot 3^2} + \frac{1}{7 \cdot 3^2} + \ldots\right).$$

B2.4.5. Для какой итерируемой последовательности x при вычислении выражения `any(x) and not all(x)` получается результат `True`?

B2.4.6. Объяснить, почему `zip(*z)` является действием, обратным `z = zip(a, b)`, т. е. если `z` соединяет элементы в пары `(a0, b0), (a1, b1), (a2, b2), ...`, то `zip(*z)` снова разделяет их `(a0, a1, a2, ...), (b0, b1, b2, ...)`.

B2.4.7. Отсортировать список кортежей, расположив кортежи в порядке, определяемом в первую очередь первым элементом каждого кортежа. Если два и более кортежей содержат одинаковый первый элемент, то они упорядочиваются по второму элементу, и т. д.:

```
>>> sorted([(3, 1), (1, 4), (3, 0), (2, 2), (1, -1)])
[(1, -1), (1, 4), (2, 2), (3, 0), (3, 1)]
```

Предполагается один из способов применения функции `zip` для сортировки одного списка с использованием элементов другого списка. Реализовать этот метод для приведенных ниже данных, чтобы получить упорядоченный список среднего количества часов солнечной погоды в Лондоне по месяцам. Месяц с наибольшим количеством часов солнечной погоды должен выводиться первым.

```
Jan  Feb  Mar   Apr   May   Jun   Jul   Aug   Sep   Oct   Nov  Dec
44.7 65.4 101.7 148.3 170.9 171.4 176.7 186.1 133.9 105.4 59.6 45.8
```

Задачи

32.4.1. Написать небольшую программу на Python, которая по заданному массиву `a` вычисляет массив того же размера `p`, в котором каждый элемент `p[i]` является произведением всех целых чисел в массиве `a`, за исключением элемента `a[i]`. Например, если `a = [1, 2, 3]`, то `p = [6, 3, 2]`.

32.4.2. Расстояние Хэмминга (Hamming distance) между двумя строками равной длины – это количество позиций, в которых символы различны. Написать небольшую программу на Python для вычисления расстояния Хэмминга между двумя строками `s1` и `s2`.

32.4.3. Используя кортеж строк, являющихся числительными, соответствующими цифрам 0–9, написать программу на Python, которая выводит представление числа π с восемью знаками после десятичной точки в форме последовательности числительных:

```
three point one four one five nine two six five
```

32.4.4. Написать программу для вывода правильно отформатированного представления первых восьми строк треугольника Паскаля.

32.4.5. Последовательность ДНК кодирует каждую аминокислоту, формирующую протеин, как последовательность из трех нуклеотидов – триплет (или кодон – codon). Например, фрагмент последовательности AGTCTTATATCT содержит триплеты (AGT, CTT, ATA, TCT), если читать с первой позиции («frame»). Если читать во втором фрейме (со второй позиции), то получатся триплеты (GTC, TTA, TAT), в третьем фрейме – (TCT, TAT, ATC).

Написать код Python для извлечения триплетов в список из трехбуквенных строк из заданной последовательности и определить `frame` как целочисленное значение (0, 1 или 2).

32.4.6. Функция факториал $n! = 1 \times 2 \times 3 \times \ldots \times (n-1) \times n$ – произведение первых n положительных чисел представлена в модуле `math` методом `factorial`. Функция двойной факториал $n!!$ – это произведение положительных нечетных чисел, включая n (которое также обязательно должно быть нечетным числом):

$$n!! = \prod_{i=1}^{(n+1)/2} (2i-1) = 1 \times 3 \times 5 \times \ldots \times (n-2) \times n$$

Написать программу вычисления $n!!$.

Дополнительное задание: расширить формулу для вычисления с четными числами, включая четное число n:

$$n!! = \prod_{i=1}^{n/2} (2i) = 2 \times 4 \times 6 \times \ldots \times (n-2) \times n.$$

32.4.7. Закон Бенфорда (закон первой цифры) – это наблюдение по распределению частот появления первой значащей определенной цифры в числовых значениях в множестве разнообразных наборов данных. Достаточно часто обнаруживается, что распределение вероятностей появления первых цифр неравномерно, оно соответствует логарифмическому распределению:

$$P(d) = \log_{10}((d + 1)/d).$$

Таким образом, числа, начинающиеся с 1, встречаются чаще, чем числа, начинающиеся с 2, и т. д., а числа, начинающиеся с 9, встречаются реже всего. Вероятности приведены ниже:

1 0.301
2 0.176
3 0.125
4 0.097
5 0.079
6 0.067
7 0.058
8 0.051
9 0.046

Закон Бенфорда наиболее точен для наборов данных, расположенных в диапазоне нескольких порядков величины, а его точность может быть подтверждена для некоторых бесконечных числовых последовательностей.

а) Показать, что первые цифры первых 500 чисел Фибоначчи (см. пример П2.20) достаточно точно соответствуют закону Бенфорда.
б) Длины последовательностей аминокислот для 500 случайно выбранных протеинов представлены в файле *protein_lengths.py*, который можно скачать здесь: https://scipython.com/ex/bba. Этот файл содержит список `naa`, который можно импортировать в начале программы инструкцией

```
from protein_lengths import naa
```

В какой степени это распределение длин цепочек протеинов соответствует закону Бенфорда?

2.5 Управление потоком выполнения

Лишь немногие компьютерные программы выполняются исключительно линейно, т. е. инструкции выполняются друг за другом в том порядке, в котором они записаны в исходном коде. Гораздо более вероятно, что при выполнении программы выбираются объекты данных и блоки кода выполняются в зависимости от конкретных условий на основе некоторых оперативных проверок. Поэтому все языки программирования, применяемые в реальной практике, содержат различные варианты конструкции if-then-(else). В этом разделе описывается синтаксис Python-версии данной конструкции, а также рассматривается еще один тип цикла – `while`.

2.5.1 Конструкция `if…elif…else`

Конструкция `if…elif…else` позволяет выполнять инструкции в зависимости от выполнения заданных условий, т. е. в зависимости от результата одной или нескольких проверок логических выражений (которые при вычислении выдают логические значения `True` или `False`):

```
if <логическое выражение 1>:
    <блок инструкций 1>
elif <логическое выражение 2>:
    <блок инструкций 2>
...
else:
    <блок инструкций>
```

Таким образом, если при вычислении *<логическое выражение 1>* получен результат `True`, то выполняется *<блок инструкций 1>*, иначе если при вычислении *<логическое выражение 2>* получен результат `True`, то выполняется *<блок инструкций 2>* и т. д. Если ни одно из заданных явно логических выражений не дало результат `True`, то выполняется блок инструкций, следующих за ключевым словом `else:`. Все эти блоки инструкций сдвигаются вправо с помощью пробелов, как блок кода для цикла `for`. Например:

```
for x in range(10):
    if x <= 3:
        print(x, 'is less than or equal to three')
    elif x > 5:
        print(x, 'is greater than five')
    else:
        print(x, 'must be four or five , then')
```

При выполнении этого фрагмента кода выводится следующий результат:

```
0 is less than or equal to three
1 is less than or equal to three
2 is less than or equal to three
3 is less than or equal to three
4 must be four or five , then
5 must be four or five , then
6 is greater than five
7 is greater than five
8 is greater than five
9 is greater than five
```

Не обязательно заключать в круглые скобки выражение проверки условия, например `x <= 3`, как это требуется, скажем, в C, но символ двоеточия после этого выражения обязателен. В действительности выражение проверки условия при вычислении не обязательно должно выдавать только логические значения `True` или `False`: как мы видели ранее, другие типы данных считаются равнозначными `True`, если они не равны `0` (`int`) или `0.` (`float`), пустой строке `''`, пустому списку `[ ]`, пустому кортежу `()` и т. д. или специальному значению `None` (см. раздел 2.2.4). Рассмотрим следующий фрагмент кода:

```
for x in range(10):
    if x % 2:
        print(x, 'is odd!')
    else:
        print(x, 'is even!')
```

Этот код работает, потому что `x % 2 = 1` для нечетных целых чисел, что равнозначно `True`, и `x % 2 = 0` для четных целых чисел, что равнозначно `False`.

В Python нет конструкции выбора варианта `switch...case...finally` – равнозначную версию конструкции управления потоком выполнения можно реализовать с помощью конструкции `if...elif...else` или с использованием словарей (см. раздел 4.2).

Пример П2.21. В григорианском календаре год считается високосным, если представляющее его число делится без остатка на 4, за исключением тех годов, которые делятся без остатка на 100. Эти годы не считаются високосными, если только они также не делятся без остатка на 400. В листинге 2.3 приведена программа, определяющая, является ли год `year` високосным.

Листинг 2.3. Определение високосного года

```
year = 1900

if not year % 400:
    is_leap_year = True
elif not year % 100:
    is_leap_year = False
elif not year % 4:
    is_leap_year = True
else:
    is_leap_year = False

s_ly = 'is a' if is_leap_year else 'is not a'
print('{:4d} {:s} leap year'.format(year , s_ly))
```

При выполнении выводится следующий результат:

```
1900 is not a leap year
```

2.5.2 Цикл `while`

В цикле `for` устанавливается конкретное фиксированное количество итераций, тогда как инструкции в блоке кода цикла `while` выполняются пока и только тогда, когда выполняется некоторое заданное условие:

```
>>> i = 0
>>> while i < 10:
...     i += 1
...     print(i, end='.')
...
>>> print()
1.2.3.4.5.6.7.8.9.10.
```

Счетчик i инициализируется значением 0, которое меньше 10, поэтому начинается выполнение цикла while. На каждой итерации i увеличивается на единицу, и это значение выводится. Когда значение i становится равным 10 на следующей итерации, проверка условия i < 10 дает результат False: цикл завершается, и выполнение продолжается за пределами тела цикла, где команда print() выводит символ перехода на новую строку.

Пример П2.22. Более интересный пример использования цикла while – приведенная ниже реализация алгоритма Евклида для поиска наибольшего общего делителя двух чисел – gcd(a,b):

```
>>> a, b = 1071, 462
>>> while b:
...     a, b = b, a % b
...
>>> print(a)
21
```

Цикл продолжается, пока при делении a на b получается остаток, на каждой итерации b присваивается остаток от целочисленного деления a//b, затем a присваивается старое значение b. Напомню, что целое значение 0 интерпретируется как логическое значение False, поэтому условие цикла while b: в данном случае равнозначно while b != 0:.

2.5.3 Дополнительные средства управления потоком выполнения: break, continue, pass и else

Команда *break*

Python предоставляет три дополнительные команды для управления потоком выполнения программы. Команда break, выполненная в теле цикла, немедленно завершает выполнение этого цикла и передает управление инструкциям, следующим за циклом:

```
x = 0
while True:
    x += 1
    if not (x % 15 or x % 25):
        break
print(x, 'is divisible by both 15 and 25')
```

В этом примере условие цикла while всегда равно (литеральному значению) True, поэтому выход из цикла выполняется лишь при переходе к инструкции break. Это происходит только в том случае, когда счетчик x делится без остатка и на 15, и на 25. Следовательно, будет выведен результат:

```
75 is divisible by both 15 and 25
```

Аналогично, для поиска индекса самого первого вхождения отрицательного числа в списке:

```
alist = [0, 4, 5, -2, 5, 10]
for i, a in enumerate(alist):
    if a < 0:
        break
print(a, 'occurs at index', i)
```

Вывод:

```
-2 occurs at index 3
```

Обратите внимание: после экстренного выхода из цикла переменные `i` и `a` сохраняют значения, которые они получили в цикле к моменту выполнения команды `break`.

Команда *continue*

Команда `continue` действует подобно команде `break`, но вместо немедленного выхода из содержащего ее цикла она немедленно начинает новую итерацию этого цикла без завершения блока инструкций для текущей итерации. Например:

```
for i in range(1, 11):
    if i % 2:
        continue
    print(i, 'is even!')
```

Здесь выводятся только четные числа 2, 4, 6, 8, 10. Если `i` не делится на 2 без остатка (следовательно, `i % 2` равно `1`, что равнозначно `True`), то выполнение текущей итерации цикла отменяется, и цикл возобновляется со следующим значением `i` (выполнение инструкции `print` отменяется).

Команда *pass*

Команда `pass` ничего не делает. Она полезна как «заглушка» для кода, который пока еще не написан, но в этом месте синтаксически обязательно требуется команда Python по соглашению для блока кода с отступами, сделанными с помощью пробелов.

```
>>> for i in range(1, 11):
...     if i == 6:
...         pass     # Здесь необходимо сделать что-то особенное, если i равно 6
...     if not i % 3:
...         print(i, 'is divisible by 3')
...

3 is divisible by 3
6 is divisible by 3
9 is divisible by 3
```

Если бы вместо команды `pass` выполнялась команда `continue`, то строка `6 is divisible by 3` не была бы выведена: управление было бы передано на начало цикла с присваиванием `i = 7` вместо перехода ко второй инструкции `if`.

◊ Команда *else*

За телом цикла `for` или `while` может следовать блок инструкций команды `else`, который будет выполнен только в том случае, если цикл завершился «нормально» (т. е. без вмешательства команды `break`). Для циклов `for` это означает, что инструкции блока `else` будут выполнены после того, как цикл пройдет до конца всю итерируемую последовательность. Для циклов `while` инструкции блока `else` будут выполнены после того, как условие цикла `while` становится равным `False`. Например, рассмотрим еще раз программу для поиска первого вхождения отрицательного числа в списке. При отсутствии отрицательного числа в списке исходная версия кода ведет себя достаточно странно:

```
>>> alist = [0, 4, 5, 2, 5, 10]
>>> for i, a in enumerate(alist):
...     if a < 0:
...         break
...
>>> print(a, 'occurs at index', i)
10 occurs at index 5
```

Здесь выводится индекс и самый последний числовой элемент списка (вне зависимости от того, является это число отрицательным или нет). Это можно исправить, если наблюдать за тем, как завершился цикл `for`: прошел ли он по всем элементам без обнаружения отрицательного числа (следовательно, без выполнения команды `break`), и вывести соответствующее сообщение:

```
>>> alist = [0, 4, 5, 2, 5, 10]
... for i, a in enumerate(alist):
...     if a < 0:
...         print(a, 'occurs at index', i)
...         break
... else:
...     print('no negative numbers in the list')
...

no negative numbers in the list
```

Другой пример: рассмотрим следующую (не слишком изящную) программу поиска наибольшего делителя числа `a > 2`:

```
a = 1013
b = a - 1
while b != 1:
    if not a % b:
        print('the largest factor of', a, 'is', b)
        break
    b -= 1
else:
    print(a, 'is prime!')
```

Число `b` является наибольшим делителем, не равным числу `a`. Цикл `while` продолжает выполняться, пока `b` не станет равным `1` (в том случае, если `a` – простое число), и на каждой итерации уменьшает `b` на единицу после проверки

деления a на b без остатка. Если проверка выполнена успешно, то b является наибольшим делителем a, и выполняется команда break для немедленного выхода из цикла while.

Пример П2.23. Простой виртуальный робот «черепашка» живет в бесконечной двумерной плоскости, а его местоположение всегда определяется парой целочисленных координат (*x*,*y*). Робот может быть ориентирован (по направлению взгляда) только в направлениях, параллельных осям *x* и *y* (т. е. «север», «восток», «юг» или «запад»), и понимает четыре команды:

- F – перемещение вперед на одну единицу (клетку);
- L – поворот влево (против часовой стрелки) на 90°;
- R – поворот вправо (по часовой стрелке) на 90°;
- S – остановиться и выйти.

Программа на Python, приведенная в листинге 2.4, принимает список этих команд как строку и отслеживает местоположение робота-черепашки. Черепашка начинает движение с позиции (0,0) и с ориентации в направлении (1,0) (на восток). Программа игнорирует неправильные команды (но предупреждает о том, что они некорректны) и сообщает о том, что черепашка пересекает свой собственный путь.

Листинг 2.4. Виртуальный робот-черепашка

```
# eg2-turtle.py
commands = 'FFFFFLFFFLFFFFRRRFXFFFFFFS'

# Текущая локация, текущее направление движения.
x, y = 0, 0
dx, dy = 1, 0
# Отслеживание локации черепашки в списке кортежей locs.
locs = [(0, 0)]

for cmd in commands:                                                    ❶
    if cmd == 'S':
        # Команда остановки.
        break
    if cmd == 'F':
        # Перемещение вперед в текущем направлении движения.
        x += dx
        y += dy
        if (x, y) in locs:
            print('Path crosses itself at: ({}, {})'.format(x, y))
        locs.append((x, y))
        continue
    if cmd == 'L':
        # Поворот влево (против часовой стрелки).
        # L => (dx, dy): (1, 0) -> (0, 1) -> (-1, 0) -> (0, -1) -> (1, 0).
        dx, dy = -dy, dx
        continue
    if cmd == 'R':
        # Поворот вправо (по часовой стрелке).
        # R => (dx, dy): (1, 0) -> (0, -1) -> (-1, 0) -> (0, 1) -> (1, 0).
        dx, dy = dy, -dx
        continue
```

```
    # Если мы попали в эту часть программы, значит, команда не определена:
    # выводится предупреждение.
    print('Unknown command:', cmd)
else:                                                                    ❷
    # Весь список команд выполнен, но не обнаружена команда S для остановки (STOP).
    print('Instructions ended without a STOP')

# Изображать пройденный путь звездочками.
# Сначала определить общий диапазон обнаруженных значений x и y.
x, y = zip(*locs)                                                        ❸
xmin , xmax = min(x), max(x)
ymin , ymax = min(y), max(y)
# Размер сетки координат, необходимый для вывода изображения, - (nx, ny).
nx = xmax - xmin + 1
ny = ymax - ymin + 1
# Изменить направление оси y на обратное, чтобы ордината уменьшалась *вниз* по экрану.
for iy in reversed(range(ny)):
    for ix in range(nx):
        if (ix + xmin , iy + ymin) in locs:
            print('*', end='')
        else:
            print(' ', end='')
    print()
```

❶ Можно выполнить итеративный проход по строке команд `commands`, чтобы выбирать по одному символу за один раз.

❷ Обратите внимание: команда `else:` для цикла `for` выполняется только в том случае, если не была выполнена команда `break` для немедленного выхода из цикла при обнаружении команды STOP.

❸ Разделение (unzip) списка кортежей `locs` на отдельные последовательности координат *x* и *y* с помощью вызова метода `zip(*locs)`.

При заданном списке команд перемещения черепашки эта программа выводит следующий результат:

```
Unknown command: X
Path crosses itself at: (1, 0)
 *****
 *   *
 *   *
******
 *
 *
 *
 *
```

2.5.4 Упражнения

Вопросы

B2.5.1. Написать программу на Python для нормализации списка чисел *a*, чтобы значения размещались в интервале от 0 до 1. Например, список чисел

`a = [2, 4, 10, 6, 8, 4]` должен быть преобразован в список `[0.0, 0.25, 1.0, 0.5, 0.75, 0.25]`.
Совет: воспользуйтесь встроенными функциями `min` и `max`, возвращающими минимальное и максимальное значения в последовательности соответственно, например `min(a)` возвращает 2 из приведенного выше списка чисел.

B2.5.2. Написать цикл `while` для вычисления арифметико-геометрического среднего (АГС) двух положительных действительных чисел *x* и *y*, определяемого как предел последовательностей:

$$a_{n+1} = 1/2(a_n + b_n)$$
$$b_{n+1} = \sqrt{(a_n b_n)}$$

при начальных значениях $a_0 = x$, $b_0 = y$. Обе последовательности сходятся к одному и тому же числу, обозначаемому как agm(*x*,*y*). Используйте этот же цикл для определения константы Гаусса $G = 1/\text{agm}(1, \sqrt{2})$.

B2.5.3. В игре Fizzbuzz используется числовой счет, но с заменой чисел, делящихся нацело на 3, словом Fizz, чисел, делящихся нацело на 5, – словом Buzz, а чисел, делящихся нацело и на 3, и на 5, – словом FizzBuzz. Написать программу, имитирующую эту игру при числовом счете до 100.

B2.5.4. Неразветвленные молекулы алканов – это насыщенные углеводороды с общей стехиометрической формулой C_nH_{2n+2}, в которой атомы углерода образуют простую цепочку, например бутан C_4H_{10} описывается структурной формулой, которую можно представить в виде $H_3CCH_2CH_2CH_3$. Написать программу, выводящую структурную формулу таких алканов с заданной стехиометрией (предполагается, что $n > 1$). Например, при заданной стехиометрии `stoich = 'C8H18'` должен выводиться результат `H3C-CH2-CH2-CH2-CH2-CH2-CH2-CH3`.

Задачи

32.5.1. Изменить решение задачи 32.4.4 для вывода первых 50 строк треугольника Паскаля, но вместо чисел выводить звездочку, если число нечетное, и пробел, если число четное.

32.5.2. Приближенное представление многозвенной слабой кислоты определяет концентрацию ионов водорода $[H^+]$ в растворе с учетом константы диссоциации кислоты K_a и концентрации кислоты *c*, последовательно применяя формулу

$$[H^+]_{n+1} = \sqrt{(K_a(c - [H^+]_n))}$$

при начальном значении $[H^+]_0 = 0$. Итерации продолжаются до тех пор, пока изменение значения $[H^+]$ не станет меньшим некоторой предварительно заданной малой пороговой величины допустимого отклонения.

Использовать этот метод для определения концентрации ионов водорода и соответственно pH ($= -\log_{10}[H^+]$) для раствора с концентрацией c = 0.01 M уксусной кислоты ($K_a = 1.78\times10^{-5}$). Принять величину допустимого отклонения равной `TOL = 1.e-10`.

32.5.3. Алгоритм Луна (Luhn algorithm) – это простая формула вычисления контрольной суммы для проверки подлинности кредитных карт и номеров банковских счетов. Он предназначен для предотвращения часто встречающихся ошибок при непреднамеренном искажении данных (например, при ручном вводе номера карты) и выявляет все ошибки в одной цифре и почти все случаи перестановок двух рядом стоящих цифр. Этот алгоритм можно записать в виде следующей последовательности шагов:

a. реверсирование (изменение порядка цифр на обратный) числа;
b. интерпретация числа как массива цифр; затем выбираются цифры с четными индексами (нумерация индексов начинается с 1; цифры с нечетными индексами остаются неизменными) и их значения удваиваются. Если после удваивания получилось число больше 9, то оно заменяется суммой его цифр (например, цифра 6 после удваивания дает результат 12, следовательно, заменяется на 1 + 2 = 3);
c. все элементы полученного массива суммируются;
d. если полученная сумма кратна 10 (делится без остатка на 10), то исходное число (номер кредитной карты) верно.

Написать программу на Python, принимающую номер кредитной карты как строку из цифр (возможно, в форме групп, разделенных пробелами) и определяющую подлинность этой карты. Например, строка `'4799 2739 8713 6272'` является корректным номером кредитной карты, но при изменении любой цифры в этой строке номер становится некорректным.

32.5.4. Ниже описан метод Герона для вычисления квадратного корня числа S: сначала принимается исходное предполагаемое значение x_0, затем последовательно вычисляются значения $x_{n+1} = 1/2(x_n + S/x_n)$, позволяющие получать все более точные приближения к $\sqrt{S}$. Реализовать этот алгоритм для приближенного вычисления квадратного корня из 2 117 519.73 до двух значащих цифр после десятичной точки и сравнить с «точным» значением, полученным с помощью метода `math.sqrt`. Для выполнения этого задания принять исходное предполагаемое значение $x_0 = 2000$.

32.5.5. Написать программу определения завтрашней даты по заданной строке, представляющей сегодняшнюю дату в формате «`Д/М/Г`» или «`М/Д/Г`». Необходимо учитывать как британский (европейский), так и американский формат дат при синтаксическом разборе строки `today` в соответствии со значением логической переменной `us_date_style`. Например, если значением переменной `us_date_style` является `False`, а значением `today` – строка `'3/4/2014'`, то завтрашняя дата должна быть представлена как `'4/4/2014'`[27]. (Совет: используйте алгоритм определения високосного года, который представлен в примере из раздела 2.5.1.)

32.5.6. Написать программу на Python для определения $f(n)$, числа конечных нулей в значении факториала $n!$, используя для этого особый случай формулы Полиньяка (Polignac's formula):

[27] В реальной практике лучше пользоваться библиотекой Python `datetime` (описанной в разделе 4.5.3), но в этом упражнении ее применение запрещено.

$$f(n) = \sum_{i=1} \lfloor (n/5^i) \rfloor,$$

где $\lfloor x \rfloor$ обозначает округление в меньшую сторону (с недостатком) числа x, т. е. наибольшее целое число, меньшее или равное x.

32.5.7. Последовательность чисел-градин (hailstone sequence; гипотеза Коллатца – Collatz conjecture), начинающаяся с целого числа $n > 0$, генерируется с помощью многократно повторяющегося применения следующих трех правил:

- если $n = 1$, то последовательность завершается;
- если n четное, то следующее число последовательности равно $n/2$;
- если n нечетное, то следующее число последовательности равно $3n + 1$.

а) Написать программу вычисления последовательности чисел-градин, начиная с 27.

б) Установить время останова процесса как количество чисел в заданной последовательности чисел-градин. Изменить программу вычисления последовательности чисел-градин, чтобы она возвращала время останова процесса (stopping time) вместо самих чисел-градин. Дополнить программу возможностью демонстрации того факта, что последовательности чисел-градин, начинающиеся с числа $1 \leqslant n \leqslant 100$, согласуются с гипотезой Коллатца (утверждающей, что все последовательности чисел-градин в конце концов завершаются, т. е. при вычислении очередного числа-градины получается единица).

32.5.8. Алгоритм, известный как решето Эратосфена, находит простые числа в списке 2, 3, …, n. Его можно описать в виде последовательно выполняемых шагов, как показано ниже, при начальном значении $n = 2$ (первое простое число):

- шаг 1: пометить все числа, кратные p, в предложенном списке как не простые (т. е. числа mp, где $m = 2, 3, 4, \ldots$, – это составные числа);
- шаг 2: найти в списке первое непомеченное число, большее p. Если таких чисел нет, то остановить выполнение алгоритма;
- шаг 3: присвоить p это найденное непомеченное число и вернуться к шагу 1.

После останова выполнения алгоритма все непомеченные числа являются простыми.

Реализовать алгоритм решето Эратосфена в программе на Python и найти все простые числа, меньшие 10 000.

32.5.9. Функция (мультипликативная) Эйлера (Euler's totient function) $\varphi(n)$ подсчитывает количество положительных целых чисел, меньших или равных n, которые являются взаимно простыми с n. (Два числа a и b являются взаимно простыми, если единственным положительным целым числом, на которые делятся оба этих числа, является 1, т. е. если наибольший общий делитель НОД(a,b) = 1.)

Написать программу на Python для вычисления функции $\varphi(n)$ при $1 \leqslant n < 100$.

(*Совет:* можно воспользоваться алгоритмом Евклида для вычисления наибольшего общего делителя, описанным в примере из раздела 2.5.2.)

32.5.10. Значение числа π можно приблизительно вычислить методами Монте-Карло. Рассмотрим область плоскости *xy*, ограниченную значениями $0 \leqslant x \leqslant 1$ и $0 \leqslant y \leqslant 1$. Выбирая большое количество случайно выбранных точек в этой области и подсчитывая пропорциональное отношение точек, расположенных в области, ограниченной функцией $y = \sqrt{(1 - x^2)}$, описывающей четверть окружности, можно приблизительно вычислить значение π/4, – это область, ограниченная осями координат и графиком функции $y(x)$. Написать программу приблизительного вычисления значения числа π по этому методу.

Совет: воспользуйтесь модулем Python `random`. Метод `random.random()` генерирует (псевдо)случайное число в интервале от 0 до 1. Более подробно см. раздел 4.5.1.

32.5.11. Написать программу, принимающую строку текста (слова, возможно, со знаками препинания, разделенные пробелами), и вывести тот же текст с переставленными в случайном порядке буквами в середине слов. Необходимо сохранить знаки пунктуации в конце слов. Например, строка

```
Four score and seven years ago our fathers brought forth on this continent a new nation,
conceived
in liberty, and dedicated to the proposition that all men are created equal.
```

может быть преобразована так:

```
Four sorce and seevn yeras ago our fhtaers bhrogut ftroh on this cnnoientt a new noitan,
cvieecond
in lbrteiy, and ddicetead to the ptosoiporin that all men are cetaerd euaql.
```

Совет: метод `random.shuffle` перемешивает в случайном порядке элементы непосредственно в списке. См. раздел 4.5.1.

32.5.12. Конфигурация электронов в атоме – это спецификация распределения электронов по атомным орбиталям. Атомная орбиталь определяется главным квантовым числом $n = 1, 2, 3, \ldots$, определяющим оболочку, состоящую из одной или нескольких подоболочек, определяемых азимутальным (орбитальным) квантовым числом $l = 0, 1, 2, \ldots, n - 1$. Значения $l = 0, 1, 2, 3$ обозначаются буквами s, p, d и f соответственно. Таким образом, первые несколько орбиталей – это 1s ($n = 1, l = 0$), 2s ($n = 2, l = 0$), 2p ($n = 2, l = 1$), 3s ($n = 3, l = 0$), и каждая оболочка содержит n подоболочек. Данную оболочку могут занимать максимум $2(2l + 1)$ электронов.

В соответствии с правилом Маделунга (Madelung rule) N электронов атома заполняют орбитали в порядке возрастания суммы главного и орбитального квантовых чисел $n + l$, и если две орбитали имеют одинаковое значение $n + l$, то они заполняются в порядке возрастания n. Например, устойчивое состояние титана ($N = 22$) прогнозируется (и определяется) как $1s^2 2s^2 2p^6 3s^2 3p^6 4s^2 3d^2$.

Написать программу для прогнозирования конфигурации электронов для всех химических элементов до резерфордия ($N = 104$). Для титана должна выводиться следующая строка:

```
Ti: 1s2.2s2.2p6.3s2.3p6.4s2.3d2
```

Список Python, содержащий упорядоченный список химических элементов, можно загрузить отсюда: https://scipython.com/ex/bbb.

Дополнительное задание: изменить программу, чтобы обеспечить вывод конфигураций с использованием соглашения о том, что часть конфигурации, соответствующая самой удаленной от центра замкнутой оболочке, т. е. конфигурации инертного газа, заменяется на соответствующий символ инертного газа в квадратных скобках, т. е.

```
Ti: [Ar].4s2.3d2
```

Конфигурация аргона: `1s2.2s2.2p6.3s2.3p6`.

2.6 Файловый ввод/вывод

До настоящего момента данные кодировались непосредственно в программах Python, а вывод выполнялся в консоль (терминал). Разумеется, часто будет возникать необходимость ввода данных из внешнего файла, а также необходимость записи данных в файл вывода. Для этой цели в Python существуют объекты типа `file`.

2.6.1 Открытие и закрытие файла

Объект типа `file` создается при открытии файла с заданным именем (filename) и режимом доступа (mode). Имя файла может быть задано как абсолютное путевое имя или как путевое имя, определяемое по отношению к каталогу, в котором должна выполняться программа. Режим доступа `mode` – это строка, содержащая одно из значений, перечисленных в табл. 2.13.

Таблица 2.13. Режимы доступа к файлу

Аргумент `mode`	Режим доступа при открытии файла
`r`	Текстовый, только для чтения (по умолчанию)
`w`	Текстовый, запись (существующий файл с тем же именем будет перезаписан)
`a`	Текстовый, добавление в существующий файл
`r+`	Текстовый, чтение и запись
`rb`	Бинарный, только для чтения
`wb`	Бинарный, запись (существующий файл с тем же именем будет перезаписан)
`ab`	Бинарный, добавление в существующий файл
`rb+`	Бинарный, чтение и запись

Например, чтобы открыть файл для записи в текстовом режиме, выполняется следующая команда:

```
>>> f = open('myfile.txt', 'w')
```

Объекты `file` закрываются с помощью метода `close`, например `f.close()`. Python автоматически закрывает все открытые объекты `file` при завершении работы программы.

2.6.2 Запись в файл

Метод `write` объекта `file` записывает строку в файл и возвращает количество записанных символов:

```
>>> f.write('Hello World!')
12
```

Более полезным способом является использование встроенной функции `print` с передачей в нее аргумента `file`, определяющего, куда перенаправляется вывод:

```
>>> print(35, 'Cl', 2, sep='', file=f)
```

Здесь выполняется запись строки `'35Cl2'` не в консоль, а в файл, открытый как объект `f` типа `file`.

Пример П2.24. В приведенной ниже программе выполняется запись первых четырех степеней чисел в диапазоне от 1 до 1000 в поля, разделенные запятыми, в файл *powers.txt*:

```
f = open('powers.txt', 'w')
for i in range(1,1001):
    print(i, i**2, i**3, i**4, sep=', ', file=f)
f.close()
```

Содержимое записанного файла:

```
1, 1, 1, 1
2, 4, 8, 16
3, 9, 27, 81
...
999, 998001, 997002999, 996005996001
1000, 1000000, 1000000000, 1000000000000
```

2.6.3 Чтение из файла

Чтобы прочитать `n` байт из файла, необходимо вызвать метод `f.read(n)`. Если аргумент `n` не указан, то считывается весь файл[28].

Метод `readline()` считывает одну строку из файла, включая символ перехода на новую строку. При следующем вызове `readline()` считывается очередная строка и т. д. Методы `read()` и `readline()` возвращают пустую строку при достижении конца файла.

Чтобы прочитать все строки файла в список строк за одну операцию, необходимо воспользоваться методом `f.readlines()`.

Объекты `file` являются итерируемыми, и в цикле прохода по (текстовому) объекту `file` поочередно возвращается одна строка файла на каждой итерации:

---

[28] Цитата из официальной документации (перевод): «если размер файла в два раза больше размера памяти вашего компьютера, то это ваша проблема».

```
>>> for line in f:
...     print(line , end='')                                    ❶
...
First line
Second line
...
```

❶ Поскольку при считывании в строке `line` сохраняется символ перехода на новую строку, здесь используется аргумент `end=''`, чтобы функция `print` не добавляла еще один такой символ, выводящий лишнюю пустую строку.

Возможно, в реальной практике потребуется весьма частое применение этого метода, если только вы действительно не захотите сохранять каждую строку в оперативной памяти. В разделе 4.3.4 рассматривается использование ключевого слова Python `with` для более правильной организации обработки файлов.

Пример П2.25. Для чтения чисел из файла *powers.txt*, созданного в предыдущем примере, нужно обязательно преобразовать столбцы в списки целых чисел. Для этого каждую строку необходимо разделить на поля и каждое поле явно преобразовать в значение типа `int`:

```
f = open('powers.txt', 'r')
squares , cubes , fourths = [], [], []
for line in f.readlines():
    fields = line.split(',')
    squares.append(int(fields[1]))
    cubes.append(int(fields[2]))
    fourths.append(int(fields[3]))
f.close()
n = 500
print(n, 'cubed is', cubes[n-1])
```

При выполнении данная программа выводит следующий результат:

```
500 cubed is 125000000
```

Но в реальной практической деятельности лучше использовать библиотеку NumPy (см. главу 6) для считывания данных из файлов вместо способа, продемонстрированного в этом примере.

2.6.4 Упражнения

Задачи

32.6.1. Монотипный род деревьев тихоокеанского побережья «Секвойя красная» (Sequoia sempervirens) включает один из самых древних и самых высоких живых организмов на Земле. Подробная информация об отдельных деревьях содержится в текстовом файле *redwood-data.txt* (поля данных разделены символами табуляции), доступном здесь: https://scipython.com/ex/bbd. (Эти данные опубликованы с разрешения БД Gymnosperm (голосеменные растения) www.conifers.org/cu/Sequoia.php.)

Написать программу на Python для считывания этих данных и генерации сообщения о самом высоком дереве и о дереве с наибольшим диаметром.

32.6.2. Написать программу для считывания содержимого текстового файла и цензурного редактирования слов, содержащихся в списке запрещенных слов, посредством замены их букв на равное количество звездочек. В программе запрещенные слова должны храниться в нижнем регистре, но подлежащие цензурному редактированию образцы могут быть записаны в любом регистре. Предполагается отсутствие знаков пунктуации.

Дополнительное задание: обработка текста, содержащего знаки пунктуации. Например, при заданном списке запрещенных слов `['C', 'Perl', 'Fortran']` предложение

```
'Some alternative programming languages to Python are C, C++, Perl , Fortran and Java.'
```

должно быть преобразовано в следующее предложение:

```
'Some alternative programming languages to Python are *, C++, ****, ******* and Java.'
```

32.6.3. Индекс подобия Земле (Earth Similarity Index – ESI) – попытка вычисления физического сходства какого-либо астрономического тела (обычно планеты или луны (спутника)) с Землей. Индекс подобия Земле определяется по формуле

$$\mathrm{ESI}_j = \prod_{i=1}^{n} \left(1 - \left| \frac{x_{i,j} - x_{i,\oplus}}{x_{i,j} + x_{i,\oplus}} \right| \right)^{w_i/n}.$$

Параметры $x_{i,j}$, их значения для Земли $x_{i,\oplus}$ и весовые коэффициенты w_i приведены в табл. 2.15. Значения радиуса, плотности и космической скорости приняты относительно аналогичных значений для Земли. Значение ESI находится в интервале от 0 до 1. Значения, более близкие к 1, означают большее сходство с Землей (которая обладает индексом ESI, в точности равным 1: Земля идентична самой себе).

Таблица 2.15. Параметры, используемые для вычисления ESI

i	Параметр	Значение для Земли, $x_{i,\oplus}$	Весовой коэффициент, w_i
1	Радиус	1.0	0.57
2	Плотность	1.0	1.07
3	Космическая скорость, v_{esc}	1.0	0.7
4	Температура поверхности	288 K	5.58

Файл *ex2-6-g-esi-data.txt* доступен здесь: https://scipython.com/ex/bbc. В нем содержатся вышеупомянутые параметры для некоторой группы астрономических тел. Использовать эти данные для вычисления индекса ESI для каждого из этих тел. Какое из оцениваемых астрономических тел обладает свойствами, наиболее схожими со свойствами Земли?

32.4.6. Написать программу для считывания двумерного массива строк в список списков из файла, в котором элементы строк разделены одним или несколькими пробелами. При открытии файла может быть заранее неизвестно количество строк *m* и количество столбцов *n*.

Например, для текстового файла

```
A B C D
E F G H
I J K L
```

должен быть создан объект `grid` в следующем виде:

```
[['A', 'B', 'C', 'D'], ['E', 'F', 'G', 'H'], ['I', 'J', 'K', 'L']]
```

При таком способе считывания `grid` содержит список строк исходного массива. После завершения считывания массива написать циклы для вывода массива по столбцам:

```
[['A', 'E', 'I'], ['B', 'F', 'J'], ['C', 'G', 'K'], ['D', 'H', 'L']]
```

Более сложная задача: вывести также все диагонали, считываемые в одном направлении:

```
[['A'], ['B', 'E'], ['C', 'F', 'I'], ['D', 'G', 'J'], ['H', 'K'], ['L']]
```

и в другом направлении:

```
[['D'], ['C', 'H'], ['B', 'G', 'L'], ['A', 'F', 'K'], ['E', 'J'], ['I']]
```

2.7 Функции

В Python функция (function) – это набор инструкций, сгруппированных в единый блок, которому присвоено некоторое имя, благодаря чему можно многократно выполнять этот блок в программе. Использование функций дает два основных преимущества. Во-первых, функции позволяют многократно использовать фрагмент кода без копирования его в различные части программы, а во-вторых – функции позволяют разделять сложные задачи на отдельные процедуры, каждая из которых реализована собственной функцией: зачастую намного проще (в том числе и с точки зрения сопровождения) отдельно писать код для каждой процедуры, нежели кодировать всю задачу в целом.

2.7.1 Определение и вызов функций

Ключевое слово `def` определяет функцию, присваивает ей имя и задает список аргументов (если это необходимо), который эта функция предположительно принимает при вызове. Инструкции в теле функции записываются в блоке со смещением вправо после строки с определением `def`. Если в какой-то момент выполнения этого блока инструкций встречается команда `return`, то вызывающей стороне возвращается определенное значение. Например:

```
>>> def square(x):                                                    ❶
...     x_squared = x**2
...     return x_squared
...
>>> number = 2
>>> number_squared = square(number)                                   ❷
>>> print(number , 'squared is', number_squared)
2 squared is 4
>>> print('8 squared is', square(8))                                  ❸
8 squared is 64
```

❶ Простая функция с именем `square` принимает один аргумент `x`. Функция вычисляет выражение `x**2` и возвращает вычисленное значение вызывающей стороне. Функция определяется один раз, но может вызываться многократно.

❷ В первом примере возвращаемое значение присваивается переменной `number_squared`.

❸ Во втором примере возвращаемое значение передается прямо в метод `print` для вывода в консоли.

Для возвращения двух и более значений из функции необходимо упаковать их в кортеж `tuple`. Например, в следующей программе определяется функция, возвращающая оба корня квадратного уравнения $ax^2 + bx + c$ (предполагается, что уравнение имеет два действительных корня):

```
import math

def roots(a, b, c):
    d = b**2 - 4*a*c
    r1 = (-b + math.sqrt(d)) / 2 / a
    r2 = (-b - math.sqrt(d)) / 2 / a
    return r1, r2

print(roots(1., -1., -6.))
```

При выполнении эта программа выводит вполне ожидаемый результат:

```
(3.0, -2.0)
```

Но функция не обязательно должна явно возвращать какой-либо объект: функции, которые приходят к концу своего внутреннего блока инструкций без обнаружения команды `return`, возвращают специальное значение Python `None`.

Определения функций могут располагаться в любом месте Python-программы, но функцию нельзя вызывать до ее определения. Функции могут даже быть вложенными, но функция, определенная внутри другой функции, недоступна (напрямую) за пределами содержащей ее (внешней) функции.

Строки документирования docstrings

Функция docstring – это строковый литерал, который представляет собой самую первую инструкцию в определении функции. Этот литерал должен быть записан как текст в тройных кавычках на одной строке, если функция простая,

или на нескольких строках с начальной однострочной аннотацией и дальнейшим более подробным описанием более сложных функций. Например:

```
def roots(a, b, c):
    """Return the roots of ax^2 + bx + c."""                    ❶
    d = b**2 - 4*a*c
    ...
```

❶ Перевод docstring: `"""Возвращает корни квадратного уравнения ax^2 + bx + c."""`. Профессиональные программисты в основном соблюдают соглашение о том, что комментарии, docstrings и встроенная документация должны быть написаны на английском языке. – *Прим. перев.*

Эта строка документации становится специальным атрибутом __doc__ соответствующей функции:

```
>>> roots.__doc__
'Return the roots of ax^2 + bx + c.'
```

Строка документации docstring должна содержать подробную информацию о том, как использовать данную функцию: какие аргументы в нее передаются и какие объекты она возвращает[29], но в общем случае не должна включать подробности об особенной реализации алгоритмов, используемых этой функцией (лучше всего разместить такое описание в обычных комментариях, начинающихся с символа #).

Строки docstrings также используются для создания документации по классам и модулям (см. разделы 4.5 и 4.6.2).

Пример П2.26. В Python функции являются объектами первого класса (first class objects): функции могут иметь связанные с ними идентификаторы переменных, функции могут передаваться как аргументы в другие функции и даже могут возвращаться из других функций. Функции присваивается имя, по которому она определяется, но это имя может быть переприсвоено для ссылки на другой объект, если такое необходимо. (Подобное переприсваивание не рекомендуется, кроме тех случаев, когда вы точно знаете, что делаете.)

```
>>> def cosec(x):
...     """Return the cosecant of x, cosec(x) = 1/sin(x)."""    ❶
...     return 1./math.sin(x)
...
>>> cosec
<function cosec at 0x100375170 >
>>> cosec(math.pi/4)
1.4142135623730951
>>> csc = cosec                                                ❷
>>> csc
<function cosec at 0x100375170 >
>>> csc(math.pi/4)
1.4142135623730951
```

[29] В крупных проектах строки документации docstrings описывают прикладной программный интерфейс (API) проекта.

❶ Перевод docstring: `"""Возврат косеканса x, cosec(x) = 1/sin(x)."""` – *Прим. перев.*
❷ Операция присваивания `csc = cosec` присваивает идентификатор (имя переменной) `csc` тому же объекту функции, на который ссылается идентификатор `cosec`: теперь эту функцию можно вызывать и как `csc()`, и как `cosec()`.

2.7.2 Аргументы по умолчанию и именованные аргументы

Именованные аргументы

В предыдущем примере аргументы передавались в функцию в том порядке, в котором они были заданы в определении этой функции (в этом случае они называются позиционными аргументами). Также возможна передача аргументов в произвольном порядке, если передавать их явно как именованные аргументы (keyword arguments):

```
roots(a=1., c=-6., b=-1.)
roots(b=-1., a=1., c=-6.)
```

При совместной передаче неименованных (позиционных) и именованных аргументов позиционные аргументы непременно должны записываться первыми, иначе Python не узнает, какой переменной соответствует позиционный аргумент:

```
>>> roots(1., c=6., b=-1.) # Это правильная передача аргументов.
(3.0, -2.0)
>>> roots(b=-1., 1., -6.) # Ошибка: не понятно, какой аргумент предназначен для a, а какой
                          # для c.
  File "<stdin >", line 1
SyntaxError: non-keyword arg after keyword arg
```

Аргументы по умолчанию

Иногда необходимо определить функцию, принимающую необязательный аргумент: если вызывающая сторона не определяет значение такого аргумента, то используется значение по умолчанию. Значение по умолчанию для подобного аргумента устанавливается в определении функции:

```
>>> def report_length(value , units='m'):
...     return 'The length is {:.2f} {}'.format(value , units)
>>> report_length(33.136, 'ft')
'The length is 33.14 ft'
>>> report_length (10.1)
'The length is 10.10 m'
```

Значение по умолчанию присваивается аргументам, когда интерпретатор Python в первый раз встречает соответствующее определение функции. Это может приводить к некоторым непредсказуемым результатам, особенно для изменяемых аргументов. Например:

```
>>> def func(alist = []):
...     alist.append(7)
...     return alist
...
>>> func()
[7]
>>> func()
[7, 7]
>>> func()
[7, 7, 7]
```

Здесь значением аргумента по умолчанию для функции `func` является пустой список, но особенность заключается в том, что пустой список присваивается только один раз при определении этой функции. Поэтому при каждом вызове `func` размер этого списка увеличивается на один элемент.

Пример П2.27. Значения аргументов по умолчанию присваиваются при определении функции. Таким образом, если функция определяется с передачей аргумента, которым по умолчанию является некоторый неизменяемый объект, то последующее изменение значения этой переменной не будет изменять значение по умолчанию:

```
>>> default_units = 'm'
>>> def report_length(value , units=default_units):
...     return 'The length is {:.2f} {}'.format(value , units)
...
>>> report_length (10.1)
'The length is 10.10 m'
>>> default_units = 'cubits'
>>> report_length (10.1)
'The length is 10.10 m'
```

Заданные по умолчанию единицы измерения, используемые функцией `report_length`, не изменяются после следующего присваивания значения переменной с именем `default_units`: значение по умолчанию установлено как строковый объект, связанный с именем `default_units`, когда компилятор Python встретил ключевое слово `def` (и присвоил этому объекту значение `'m'`), поэтому в дальнейшем изменить значение объекта `default_units` невозможно.

Это также означает, что если в качестве значения аргумента по умолчанию присваивается изменяемый объект, то это всегда один и тот же объект, который используется при любом вызове функции без указания альтернативного значения: см. вопрос В2.7.4 из раздела упражнений.

2.7.3 Область видимости

Функция может определять и использовать собственные переменные. Эти переменные являются локальными (local) относительно своей функции: они недоступны за пределами функции. А переменные, созданные вне пределов всех определений функций `def`, являются глобальными (global) и доступны в любом месте файла программы. Например:

```
>>> def func():
...     a = 5
...     print(a, b)
...
>>> b = 6
>>> func()
5 6
```

Функция `func` определяет локальную переменную `a`, но выводит обе переменные `a` и `b`. Поскольку переменная `b` не определена в локальной области видимости (local scope) этой функции, Python ищет ее в глобальной области видимости (global scope), находит `b = 6`, следовательно, выводится именно это значение. Не важно, что переменная `b` не была определена при определении функции, но, разумеется, переменная `b` должна быть обязательно определена перед вызовом функции.

Что происходит, когда функция определяет переменную с тем же именем, что и имя глобальной переменной? В этом случае сначала выполняется поиск в локальной области видимости функции для разрешения конфликта имен переменных, поэтому извлекается объект, на который указывает локальное имя переменной. Например:

```
>>> def func():
...     a = 5
...     print(a)
...
>>> a = 6
>>> func()
5
>>> print(a)
6
```

Обратите внимание: локальная переменная `a` существует только в теле функции, после этого обнаруживается, что ее имя полностью совпадает с именем глобальной переменной `a`. Локальная переменная исчезает после выхода из функции и уже не замещает глобальную переменную `a`.

В Python правила разрешения конфликтов областей видимости можно описать аббревиатурой LEGB: сначала локальная (local – L) область видимости, затем внешняя включающая (enclosing – E) область видимости (для вложенных функций), потом глобальная (global – G) область видимости, наконец, встроенные (built-ins – B) объекты, если так случилось, что вы дали переменной имя, совпадающее с именем встроенной функции (например, `range` или `len`), то конфликт имен разрешается в пользу вашей переменной (в локальной или глобальной области видимости), а не в пользу встроенного объекта. Следовательно, далеко не лучшим решением является назначение переменным имен, совпадающих с именами встроенных объектов.

◊ Ключевые слова *global* и *nonlocal*

В предыдущем разделе описана возможность доступа к переменным, определенным в областях видимости, отличающихся от локальной области видимости функции. Но можно ли изменять эти переменные (связывать их с новыми объектами)? Рассмотрим различие в поведении следующих функций:

```
>>> def func1():
...     print(x) # Допустимо: используемая переменная x определяется в глобальной или
                 # внутренней области видимости.
...
>>> def func2():
...     x += 1 # Недопустимо: запрещено изменять переменную x, если она не является локальной.
...
>>> x = 4
>>> func1()
4
>>> func2()
UnboundLocalError: local variable 'x' referenced before assignment
```

Если действительно требуется изменять переменные, определенные за пределами локальной области видимости, то сначала необходимо обязательно объявить в теле функции об этом своем намерении с помощью ключевых слов `global` (для переменных в глобальной области видимости) и `nonlocal` (для переменных во внутренней области видимости, например когда одна функция определена внутри другой). Для приведенного выше примера:

```
>>> def func2():
...     global x
...     x += 1            # Теперь допустимо: Python знает, что подразумевается x
                          # в глобальной области видимости.
...
>>> x = 4
>>> func2()               # Ошибки нет.
>>> x
5
```

Функция `func2` действительно изменила значение переменной *x* в глобальной области видимости.

Вы должны тщательно обдумать, действительно ли необходима такая методика (не лучше ли передать `x` как аргумент и вернуть с помощью `return` обновленное значение из функции?). Особенно в больших программах использование имен переменных в одной области видимости, которые изменяют значение (или даже тип) внутри функций, приводит к запутыванию кода, к трудно предсказуемому поведению и к глубоко скрытым ошибкам.

Пример П2.28. Внимательно изучите исходный код в листинге 2.5 и попробуйте предсказать результат до его выполнения.

Листинг 2.5. Правила определения областей видимости Python

```
# eg2-scope.py

def outer_func():
    def inner_func():
        a = 9
        print('inside inner_func , a is {:d} (id={:d})'.format(a, id(a)))
        print('inside inner_func , b is {:d} (id={:d})'.format(b, id(b)))
        print('inside inner_func , len is {:d} (id={:d})'.format(len,id(len)))

    len = 2
    print('inside outer_func , a is {:d} (id={:d})'.format(a, id(a)))
    print('inside outer_func , b is {:d} (id={:d})'.format(b, id(b)))
    print('inside outer_func , len is {:d} (id={:d})'.format(len,id(len)))
    inner_func()

a, b = 6, 7
outer_func()
print('in global scope , a is {:d} (id={:d})'.format(a, id(a)))
print('in global scope , b is {:d} (id={:d})'.format(b, id(b)))
print('in global scope , len is', len, '(id={:d})'.format(id(len)))
```

В этой программе определяется функция `inner_func`, вложенная в другую функцию `outer_func`. После определения этих функций выполнение программы продолжается следующим образом:

1) инициализируются глобальные переменные `a = 6` и `b = 7`;
2) вызывается внешняя функция `outer_func`:
   a. функция `outer_func` определяет локальную переменную `len = 2`;
   b. выводятся значения `a` и `b` – они не существуют в локальной области видимости и в какой-либо внутренней области видимости, поэтому Python находит их в глобальной области видимости: выводятся значения этих переменных (6 и 7);
   c. выводится значение локальной переменной `len` (равное 2);
   d. вызывается внутренняя функция `inner_func`:
      i. определяется локальная переменная `a = 9`;
      ii. выводится значение этой локальной переменной;
      iii. выводится значение переменной `b`, которая не существует в локальной области видимости, поэтому Python ищет ее во внешней включающей области видимости (функции `outer_func`). Здесь она не найдена, поэтому Python продолжает поиск в глобальной области видимости и находит ее: выводится значение `b = 7`;
      iv. выводится значение `len` – эта переменная не существует в локальной области видимости, но во внешней включающей области видимости определена переменная `len = 2` (в функции `outer_func`): выводится ее значение;

3) после завершения выполнения функции outer_func выводятся значения переменных a и b в глобальной области видимости;
4) выводится значение len. Эта переменная не определена в глобальной области видимости, поэтому Python выполняет поиск в собственных встроенных именах: len – встроенная функция для определения длины любой последовательности. Сама по себе эта функция также является объектом и предоставляет короткую строку собственного описания при выводе.

```
inside outer_func, a is 6 (id=232)
inside outer_func, b is 7 (id=264)
inside outer_func, len is 2 (id=104)
inside inner_func, a is 9 (id=328)
inside inner_func, b is 7 (id=264)
inside inner_func, len is 2 (id=104)
in global scope, a is 6 (id=232)
in global scope, b is 7 (id=264)
in global scope, len is <built-in function len> (id=977)
```

Обратите внимание: в этом примере функция outer_func переопределяет (возможно, неразумно) или заново связывает имя len с целочисленным объектом 2. Это означает, что исходная встроенная функция len становится недоступной в теле функции outer_func (а кроме того, недоступной и в теле вложенной функции inner_func).

2.7.4 ◊ Передача аргументов в функции

Новые пользователи Python, обладающие знаниями о других компьютерных языках, обычно задают вопрос: аргументы передаются в функции «по значению» или «по ссылке»? Другими словами, создает ли функция собственную копию аргумента, оставляя без изменений его копию на вызывающей стороне, или принимает «указатель» на область памяти аргумента, содержимое которой функция может изменить? Это различие важно для языков, подобных C, но не вполне соответствует модели Python «имя-объект». Аргументы функций Python иногда (не очень удобно) обозначаются как «ссылки, передаваемые по значению». Напомню, что в Python все является объектом, и один и тот же объект может иметь несколько идентификаторов (которые мы до сих пор называли просто «переменными»). Когда имя передается в функцию, передается именно «значение», на которое указывает соответствующий объект. Возможность изменения функцией этого объекта (с точки зрения вызывающей стороны) зависит от того, является объект изменяемым или неизменяемым.

Пара примеров должна прояснить все окончательно. Простая функция func1, принимающая целочисленный аргумент, получает ссылку на соответствующий целочисленный объект, с которым связывает локальное имя (которое может совпадать или не совпадать с именем глобальной переменной). Функция не может изменить этот целочисленный объект (поскольку он является неизменяемым), поэтому любое переприсваивание локального имени просто указывает на новый объект: глобальное имя продолжает указывать на исходный целочисленный объект.

```
>>> def func1(a):
...     print('func1: a = {}, id = {}'.format(a, id(a)))
...     a = 7         # Новое присваивание локальной переменной a целочисленного значения 7.
...     print('func1: a = {}, id = {}'.format(a, id(a)))
...
>>> a = 3
>>> print('global: a = {}, id = {}'.format(a, id(a)))

global: a = 3, id = 4297242592

>>> func1(a)
func1: a = 3, id = 4297242592
func1: a = 7, id = 4297242720

>>> print('global: a = {}, id = {}'.format(a, id(a)))

global: a = 3, id = 4297242592
```

Таким образом, функция `func1` выводит `3` (внутри функции `a` изначально является локальным именем для переданного исходного целочисленного объекта). Затем выводится значение 7 (то же локальное имя теперь указывает на новый целочисленный объект с новым идентификатором), см. рис. 2.5. После возврата из функции глобальное имя продолжает указывать на исходное значение 3.

Теперь рассмотрим передачу изменяемого объекта, например списка (`list`) в функцию `func2`. В этом случае присваивание списку изменяет исходный объект, и все подобные изменения сохраняются после завершения вызова функции.

```
>>> def func2(b):
...     print('func2: b = {}, id = {}'.format(b, id(b)))
...     b.append(7)               # Добавление элемента в список.
...     print('func2: b = {}, id = {}'.format(b, id(b)))
...
>>> c = [1, 2, 3]
>>> print('global: c = {}, id = {}'.format(c, id(c)))

global: c = [1, 2, 3], id = 4361122448

>>> func2(c)
func2: b = [1, 2, 3], id = 4361122448
func2: b = [1, 2, 3, 7], id = 4361122448

>>> print('global: c = {}, id = {}'.format(c, id(c)))

global: c = [1, 2, 3, 7], id = 4361122448
```

Обратите внимание: не имеет значения, какое имя назначается списку внутри функции: это имя указывает на тот же объект, как можно убедиться, если вывести его идентификатор. Отношения между именами переменных и объектами показаны на рис. 2.6.

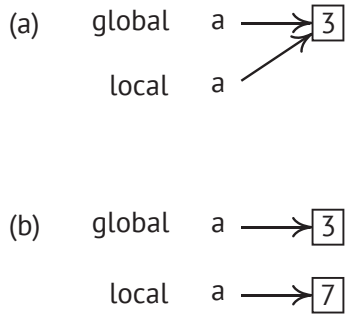


Рис. 2.5. Неизменяемые объекты. Внутри функции `func1`: а) перед изменением присваивания локальной переменной `a`; б) после изменения присваивания значения локальной переменной `a`

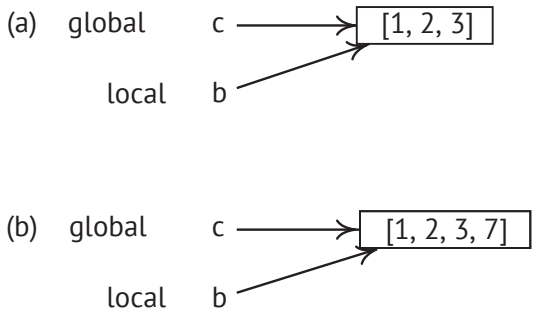


Рис. 2.6. Изменяемые объекты. Внутри функции `func2`: а) перед добавлением элемента в список, на который указывают и глобальная переменная `c`, и локальная переменная `b`; б) после добавления элемента в список командой `b.append(7)`

И все же: как передаются аргументы в Python – по значению или по ссылке? Вероятно, самый правильный ответ таков: аргументы передаются по значению, но это значение является ссылкой на объект (который может быть изменяемым или неизменяемым).

Пример П2.29. Центральные многоугольные числа (Lazy caterer's sequence) $f(n)$ – последовательность, описывающая максимальное количество кусков, на которые можно разрезать круглую пиццу или пирог прямыми линиями при возрастающем числе разрезов n. Очевидно, что $f(0) = 1$, $f(1) = 2$ и $f(2) = 4$. Для $n = 3$ $f(3) = 7$ (максимальное количество кусков получается, если линии разрезов не пересекаются в одной общей точке). Можно показать, что применима общая рекурсивная формула

$$f(n) = f(n - 1) + n.$$

Несмотря на то что существует замкнутая форма для этой последовательности $f(n) = 1/2(n^2 + n + 2)$, можно также определить функцию увеличения списка последовательных значений таких чисел:

```
>>> def f(seq):
...     seq.append(seq[-1] + n)
...
>>> seq = [1]             # f(0) = 1
>>> for n in range(1,16):
...     f(seq)
...
>>> print(seq)
[1, 2, 4, 7, 11, 16, 22, 29, 37, 46, 56, 67, 79, 92, 106, 121]
```

Список `seq` является изменяемым объектом, поэтому увеличивает собственный размер при каждом вызове функции `f()`. Переменная `n`, используемая в теле функции, представляет собой имя, найденное в глобальной области видимости (это счетчик цикла `for`).

2.7.5 Рекурсивные функции

Функция, которая может вызывать сама себя, называется рекурсивной функцией (recursive function). Рекурсия не всегда необходима, но в некоторых ситуациях позволяет создавать изящные алгоритмы[30]. Например, одним из способов вычисления факториала целого числа $n \geqslant 1$ является определение следующей рекурсивной функции:

```
>>> def factorial(n):
...     if n == 1:
...         return 1
...     return n * factorial(n - 1)
...
>>> factorial(5)
120
```

Здесь при вызове функции `factorial(n)` выполняется возврат `n` раз после вызова `factorial(n-1)`, при котором выполняется возврат `n-1` раз после вызова `factorial(n-2)`, и т. д. до тех пор, пока вызов `factorial(1)` не вернет `1` по определению. Таким образом, этот алгоритм в действительности использует рекурсивную формулу $n! = n \times (n - 1)!$. При реализации подобных рекурсивных алгоритмов требуется особое внимание, чтобы непременно обеспечить их останов при выполнении некоторого конкретного условия[31].

[30] В действительности из-за накладных расходов при выполнении вызова функции рекурсивный алгоритм может предсказуемо выполняться медленнее, чем правильно спроектированный итеративный алгоритм.

[31] На практике бесконечный цикл невозможен, потому что при каждом вызове функции выделяется некоторый объем памяти, а кроме того, Python устанавливает максимальный предел рекурсивных вызовов.

Пример П2.30. В знаменитой задаче «Ханойская башня» рассматриваются три стержня, на одном из которых (стержень A) размещено *n* дисков с различными диаметрами. Диск с наибольшим диаметром – самый нижний, над ним располагаются диски в порядке уменьшения диаметров. Задача состоит в том, чтобы переместить все диски на третий стержень (стержень C) (расположив их в итоге в том же порядке), перемещая по одному диску за один ход так, чтобы диск большего диаметра никогда не располагался над диском меньшего диаметра. Необходимо использовать второй стержень (стержень B) как промежуточное временное место расположения дисков.

Эту задачу можно решить, используя следующий рекурсивный алгоритм. Пометим диски как D_i, при этом D_1 – диск наименьшего размера, а D_n – диск наибольшего размера. Тогда алгоритм выглядит так:

- переместить диски $D_1, D_2, \ldots, D_{n-1}$ с A на B;
- переместить диск D_n с A на C;
- переместить диски $D_1, D_2, \ldots, D_{n-1}$ с B на C.

Второй шаг представляет собой одно перемещение, но первый и третий шаги требуют комплексного перемещения стопки из $n - 1$ дисков с одного стержня на другой, а это именно та задача, которую, собственно, алгоритм и должен решить.

В коде из листинга 2.6 диски идентифицируются целыми числами 1, 2, 3, ..., хранящимися в одном из трех списков A, B и C. Начальное состояние системы: все диски находятся на стержне A, что обозначается, например, `A = [5, 4, 3, 2, 1]`, где первый по индексу элемент представляет «нижний диск» на стержне, а последний по индексу элемент – «верхний диск». По условиям задачи требуется, чтобы все три списка всегда являлись исключительно убывающими последовательностями.

Листинг 2.6. Решение задачи «Ханойская башня»

```
# eg2-hanoi.py

def hanoi(n, P1, P2, P3):
    """ Перемещение n дисков со стержня P1 на стержень P3. """
    if n == 0:
        # На этом шаге больше нет дисков для перемещения.
        return

global count
count += 1

# Перемещение n - 1 дисков с P1 на P2.
hanoi(n - 1, P1, P3, P2)

if P1:
    # Перемещение диска с P1 на P3.
    P3.append(P1.pop())
    print(A, B, C)
    # Перемещение n - 1 дисков с P2 на P3.
    hanoi(n - 1, P2, P1, P3)

# Инициализация состояния стержней: все n дисков находятся на стержне A.
n = 3
```

```
A = list(range(n, 0, -1))
B, C = [], []

print(A, B, C)
count = 0
hanoi(n, A, B, C)
print(count)
```

Обратите внимание: функция `hanoi` просто перемещает стопку дисков с одного стержня на другой: списки (представляющие стержни) передаются в нее в определенном порядке, и функция перекладывает диски со стержня, представленного первым списком, локально обозначенным как `P1`, на стержень, представленный третьим списком `P3`. При этом даже нет необходимости знать, какой список обрабатывается, `A`, `B` или `C`.

2.7.6 Упражнения

Вопросы

B2.7.1. Каждая из приведенных ниже небольших программ пытается вывести простую операцию суммирования двух чисел:

```
   56
  +44
-----
  100
-----
```

Какие две программы работают правильно? Подробно объясните, что сделано неправильно в каждой из других программ.

```
а) def line():
       '-----'

   my_sum = '\n'.join([' 56', ' +44', line(), ' 100', line()])
   print(my_sum)

б) def line():
       return '-----'

   my_sum = '\n'.join([' 56', ' +44', line(), ' 100', line()])
   print(my_sum)

в) def line():
       return '-----'

   my_sum = '\n'.join([' 56', ' +44', line , ' 100', line])
   print(my_sum)

г) def line():
       print('-----')
       print('  56')
       print(' +44')
```

```
       print(line)
       print(' 100')
       print(line)

д) def line():
       print('-----')

   print('  56')
   print(' +44')
   print(line())
   print(' 100')
   print(line())

е) def line():
       print('-----')

   print('  56')
   print(' +44')
   line()
   print(' 100')
   line()
```

В2.7.2. В следующем фрагменте кода выполняется попытка вычисления баланса накопительного счета с годовой процентной ставкой 5 % после 4 лет при начальном балансе 100 долларов.

```
>>> balance = 100
>>> def add_interest(balance , rate):
...     balance += balance * rate / 100
...
>>> for year in range(4):
...     add_interest(balance , 5)
...     print('Balance after year {}: ${:.2f}'.format(year + 1, balance))
...
Balance after year 1: $100.00
Balance after year 2: $100.00
Balance after year 3: $100.00
Balance after year 4: $100.00
```

Объяснить, почему этот код не работает, затем предложить правильное решение.

В2.7.3. Число харшад (число Нивена) – это целое число, которое делится нацело на сумму своих цифр (например, 21 делится нацело на 2 + 1 = 3, следовательно, является числом харшад). Исправить следующий код, который должен возвращать `True`, если `n` является числом харшад, или `False`, если `n` не является числом харшад:

```
def digit_sum(n):
    """ Вычисление суммы цифр целого числа n. """

    s_digits = list(str(n))
    dsum = 0
    for s_digit in s_digits:
        dsum += int(s_digit)
```

```
def is_harshad(n):
    return not n % digit_sum(n)
```

При выполнении функция `is_harshad` генерирует ошибку:

```
>>> is_harshad(21)
TypeError: unsupported operand type(s) for %: 'int' and 'NoneType'
```

B2.7.4. Определить и объяснить вывод при выполнении следующего кода:

```
def grow_list(a, lst=[]):
    lst.append(a)
    return lst

lst1 = grow_list(1)
lst1 = grow_list(2, lst1)

lst2 = grow_list('a')

print(lst1)
print(lst2)
```

Задачи

32.7.1. В настольную игру «Скрэббл» (Scrabble; есть русская версия «Эрудит») играют на поле размером 15×15 клеток, строки которого обозначаются буквами (A–O), а столбцы – числами (1–15). Написать функцию, определяющую, умещается ли слово на игровом поле, если задана позиция его первой буквы как строка (например, `'G7'`), переменная, указывающая расположение слова по горизонтали или по вертикали, и само слово.

32.7.2. Написать программу поиска наименьшего положительного числа n, факториал которого не делится нацело на сумму цифр самого факториала. Например, 6 не является таким числом, потому что 6! = 720, а 720 нацело делится на 7 + 2 + 0 = 9.

32.7.3. Написать две функции, которые получают два списка длиной 3, представляющих трехмерные векторы **a** и **b**, вычисляют скалярное произведение $\mathbf{a} \cdot \mathbf{b}$ и векторное произведение $\mathbf{a} \times \mathbf{b}$.

Написать еще две функции, возвращающие скалярное смешанное произведение $\mathbf{a} \cdot (\mathbf{b} \cdot \mathbf{c})$ и векторное смешанное произведение $\mathbf{a} \times (\mathbf{b} \times \mathbf{c})$.

32.7.4. Правильная пирамида с высотой h и основанием, представленным правильным n-угольником с длиной стороны s, имеет объем $V = {}^1/_3 Ah$ и общую площадь поверхности $S = A + ½nsl$, где A – площадь основания, l – высота боковой грани, которую можно вычислить по апофеме многоугольника основания $a = ½s \operatorname{ctg}(\pi/n)$ как $A = ½nsa$, и $l = \sqrt{h^2 + a^2}$.

Использовать эти формулы для определения функции `pyramid_AV`, возвращающей объем V и общую площадь поверхности S при передаче в нее значений n, s и h.

32.7.5. Дальность полета снаряда, выпущенного под углом α со скоростью v, на ровной земной поверхности определяется по формуле

$$R = (v^2 \sin 2\alpha) / g,$$

где g – ускорение свободного падения, значение которого можно принять равным 9.81 м/с^2 для Земли. Максимальная высота подъема снаряда определяется по формуле

$$H = (v^2 \sin^2 \alpha) / 2g.$$

(Мы пренебрегаем сопротивлением воздуха, кривизной поверхности и скоростью вращения Земли.) Написать функцию для вычисления и вывода дальности полета и максимальной высоты подъема снаряда, принимающую как аргументы значения α и v. Протестировать функцию на значениях v = 10 м/с и $\alpha = 30°$.

32.7.6. Написать функцию `sinm_cosn`, которая возвращает значение следующего определенного интеграла для целых чисел m, $n > 1$.

$$\int_0^{\pi/2} \sin^n\theta \cos^m\theta d\theta = \begin{cases} \dfrac{(m-1)!!(n-1)!!}{(m+n)!!}\dfrac{\pi}{2} \\ \dfrac{(m-1)!!(n-1)!!}{(m+n)!!} \end{cases}$$

в другом случае оба числа m, n четные

Совет: для вычисления двойного факториала см. задачу 32.4.6.

32.7.7. Написать функцию, определяющую, является ли строка палиндромом (т. е. одинаково читается в обоих направлениях), с использованием рекурсии.

32.7.8. Тетрация (tetration) определяется как следующий гипероператор (гипероператор-4) после возведения в степень. Таким образом, операцию $x \times n$ можно записать в виде суммы $x + x + x + \ldots + x$ с n слагаемыми, операцию x^n – в виде произведения n сомножителей $x + x + x + \ldots + x$, а выражение, записанное в форме ${}^n x$, равнозначно повторяющемуся возведению в степень, включающему n вхождений x:

$${}^n x = x^{x^{\cdot^{\cdot}}}$$

Например, ${}^4 2 = 2^{2^{2^2}} = 2^{2^4} = 2^{16} = 65\,536$. Следует отметить, что степенная башня вычисляется сверху вниз.

Написать рекурсивную функцию для вычисления ${}^n x$ и протестировать ее (для небольших положительных действительных значений x и неотрицательных целых n, потому что тетрация генерирует очень большие числа).

Из скольких цифр состоят числа ${}^3 5$ и ${}^5 2$?

Глава 3

Небольшое отступление: простые схемы и диаграммы

Python становится все более распространенным языком, поэтому растет количество доступных библиотек пакетов и модулей, расширяющих функциональность этого языка. Matplotlib – одна из таких библиотек. Библиотека Matplotlib предоставляет средства создания графических схем, которые могут включаться в приложения, отображены на экране или выведены в форме файлов изображений высокого качества для публикации.

Библиотека Matplotlib предлагает полнофункциональный объектно-ориентированный интерфейс, который более подробно описан в главе 7, но для создания простых схем в интерактивном сеансе командной оболочки упрощенный процедурный интерфейс `pyplot` предоставляет удобный способ визуализации данных. В этой короткой главе рассматривается использование `pyplot` вместе с некоторыми основными функциями NumPy (библиотека NumPy более подробно описана в главе 6).

В системе с установленными библиотеками Matplotlib и NumPy рекомендуется выполнять следующие инструкции импорта:

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
```

Это означает вызовы методов библиотек с префиксами `plt.` и `np.`.

Замечание: в более старом модуле Python `pylab` была объединена функциональность модулей `pyplot` и `numpy` с возможностью импортирования всех их функций в общее пространство имен для имитации коммерческого программного пакета MATLAB. Использование старого модуля `pylab` не рекомендуется, и здесь он не рассматривается.

3.1 Создание простых схем

3.1.1 Линейные графики и точечные диаграммы

Самый простой линейный (*x*,*y*) график[32] создается с помощью вызова метода `plt.plot` с передачей в него двух итерируемых объектов одинаковой длины (обычно это списки числовых значений или массивы NumPy). Например:

```
>>> ax = [0., 0.5, 1.0, 1.5, 2.0, 2.5, 3.0]
>>> ay = [0.0, 0.25, 1.0, 2.25, 4.0, 6.25, 9.0]
>>> plt.plot(ax,ay)
>>> plt.show()
```

Метод `plt.plot` создает объект Matplot (в приведенном выше примере объект `Line2D`), а метод `plt.show()` выводит этот объект на экран. На рис. 3.1 показан результат. По умолчанию линия окрашена синим цветом.

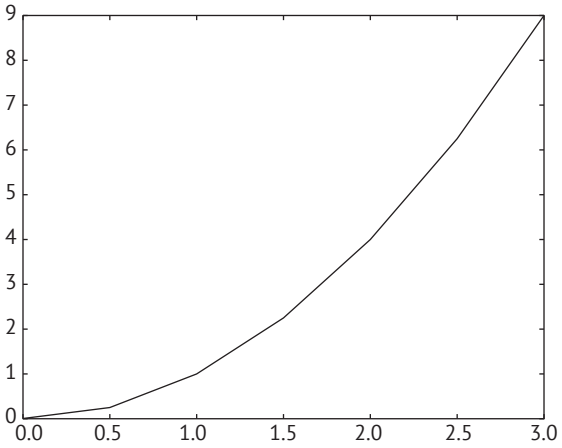


Рис. 3.1. Простой линейный (*x*,*y*) график

Для отображения точек (*x*,*y*) в виде точечной диаграммы (диаграммы рассеяния), а не линейного графика вызывается метод `plt.scatter()`:

```
>>> import random
>>> ax, ay = [], []
>>> for i in range(100):
...     ax.append(random.random())
...     ay.append(random.random())
...
>>> plt.scatter(ax,ay)
>>> plt.show()
```

Полученная в результате диаграмма показана на рис. 3.2.

---

[32] Здесь автор включает в понятие «линейный график» (line plot) не только графики в виде сплошной прямой линии, но и графики, сформированные в виде отдельных точек, соединенных прямыми линиями (как в приведенном ниже примере). – *Прим. перев.*

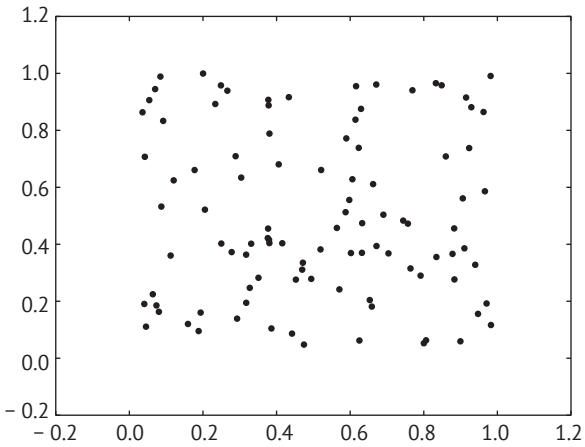


Рис. 3.2. Простая точечная диаграмма

Диаграмму можно сохранить как изображение, вызвав метод `plt.savefig(`*имя_файла*`)`. Требуемый формат изображения логически выводится из расширения файла. Например:

```
plt.savefig('plot.png')      # Сохранить как изображение в формате PNG.
plt.savefig('plot.pdf')      # Сохранить как файл формата PDF.
plt.savefig('plot.eps')      # Сохранить в формате Encapsulated PostScript (EPS).
```

Пример П3.1. Рассмотрим построение графика функции $y = \sin^2 x$ при $-2\pi \leqslant x \leqslant 2\pi$. При использовании только тех средств Python, которые рассматривались в предыдущей главе, применяется описанный ниже подход.

Вычисляется и отображается 1000 точек (x,y), координаты которых сохраняются в списках `ax` и `ay`. Для формирования списка `ax` как абсцисс нельзя использовать `range` напрямую, потому что этот метод генерирует только целочисленные последовательности, так что сначала устанавливается точность шага между каждым значением x по формуле

$$\Delta x = (x_{\max} - x_{\min}) / (n - 1)$$

(если рассматриваемые здесь n значений включают $x_{\min}$ и $x_{\max}$, то существует $n - 1$ интервалов шириной Δx), тогда точки абсцисс вычисляются по формуле

$$x_i = x_{\min} + i\Delta x \text{ при } i = 0, 1, 2, \ldots, n - 1.$$

Соответствующие координаты y вычисляются по формуле

$$y_i = \sin^2(x_i).$$

Программа в листинге 3.1 реализует описанный выше подход и выводит точки в форме простого линейного графика (см. рис. 3.3).

Листинг 3.1. Вывод графика функции $y = \sin^2 x$

```
# eg3-sin2x.py

import math
import matplotlib.pyplot as plt
xmin , xmax = -2. * math.pi, 2. * math.pi
n = 1000
x = [0.] * n
y = [0.] * n
dx = (xmax - xmin)/(n-1)
for i in range(n):
    xpt = xmin + i * dx
    x[i] = xpt
    y[i] = math.sin(xpt)**2

plt.plot(x,y)
plt.show()
```

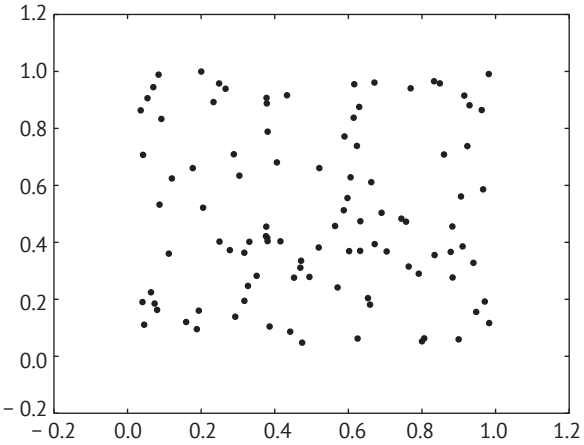


Рис. 3.3. График функции $y = \sin^2 x$

3.1.2 Метод linspace и векторизация

Для построения и вывода графика функции $y = \sin^2 x$ в примере из предыдущего раздела потребовалось достаточно много работы, самая большая часть которой пришлась на формирование списков x и y. Библиотека NumPy, более подробно описанная в главе 6, может существенно упростить жизнь разработчика.

Во-первых, последовательность координат x с постоянным шагом (список x) можно создать с использованием метода `linspace`. Метод очень похож на версию встроенной функции `range` для чисел с плавающей точкой: он принимает начальное и конечное значения, а также количество значений в последовательности и генерирует массив значений, представляющий арифме-

тическую прогрессию между двумя заданными числами (включая сами эти граничные числа). Например, `x = np.linspace(-5, 5, 1001)` создает последовательность: –5.0, –4.99, –4.98, ..., 4.99, 5.0.

Во-вторых, аналоги методов из модуля `math` в библиотеке NumPy могут работать с итерируемыми объектами (такими как списки или массивы NumPy). Поэтому выражение `y = np.sin(x)` создает последовательность значений (в действительности это массив NumPy `ndarray`), равных $\sin(x_i)$ для каждого значения x_i из массива `x`:

```
import numpy as np
import matplotlib.pyplot as plt
n = 1000
xmin , xmax = -2*np.pi, 2*np.pi
x = np.linspace(xmin , xmax , n)
y = np.sin(x)**2
plt.plot(x,y)
plt.show()
```

Эта операция называется векторизацией (vectorization) и описывается более подробно в разделе 6.1.3. Списки и кортежи можно преобразовать в объекты-массивы, поддерживающие операцию векторизации, с помощью метода-конструктора `array`:

```
>>> w = [1.0, 2.0, 3.0, 4.0]
>>> w = np.array(w)
>>> w * 100           # Каждый элемент умножается на 100.
array([ 100., 200., 300., 400.])
```

Для добавления второй линии в график нужно просто вызвать еще раз метод `plt.plot`:

```
...
x = np.linspace(xmin , xmax , n)
y1 = np.sin(x)**2
y2 = np.cos(x)**2
plt.plot(x,y1)
plt.plot(x,y2)
plt.show()
```

Следует отметить, что после завершения вывода графика с помощью метода `show` или после сохранения с помощью метода `savefig` график становится недоступным для повторного вывода – для этого необходимо еще раз вызвать метод `plt.plot`. Причина в процедурной сущности интерфейса `pyplot`: каждый вызов метода `pyplot` изменяет внутреннее состояние объекта графика. Объект графика создается последовательными вызовами таких методов (добавление линий, подписей и меток, установка граничных значений осей и т. д.), затем сформированный объект графика выводится на экран или сохраняется в файле.

Пример П3.2. Функция sinc описывается такой формулой:

$$f(x) = \sin x / x.$$

Для построения ее графика в пределах $-20 \leqslant x \leqslant 20$ выполняется следующий код:

```
>>> x = np.linspace(-20, 20, 1001)
>>> y = np.sin(x)/x

__main__:1: RuntimeWarning: invalid value encountered in true_divide
>>> plt.plot(x,y)
>>> plt.show()
```

Обратите внимание: несмотря на то что Python предупреждает о делении на нуль при $x = 0$, график функции создается и выводится правильно: для этой особой точки устанавливается специальное значение `nan` (сокращение выражения «Not a Number» – «не число»), и на графике эта точка не отображается (см. рис. 3.4).

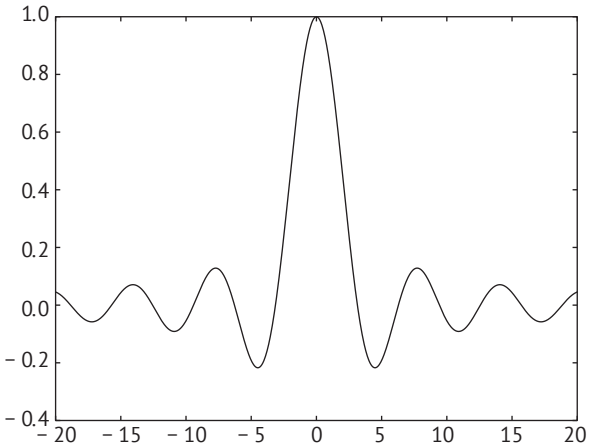


Рис. 3.4. График функции $y = \text{sinc}(x)$

```
>>> y[498:503]
array([ 0.99893367, 0.99973335,        nan, 0.99973335, 0.99893367])
```

3.1.3 Упражнения

Задачи

33.1.1. Вывести графики функций

$$f_1(x) = \ln(1 / \cos^2 x)$$

и

$$f_2(x) = \ln(1 / \sin^2 x)$$

в 1000 точках в диапазоне $-20 \leqslant x \leqslant 20$. Что происходит с этими функциями в точках $x = n\pi/2$ ($n = 0, \pm1, \pm2, \ldots$)? Что отображается на графике в этих точках?

33.1.2. Уравнение Михаэлиса–Ментен – наиболее известная модель ферментативных кинетических реакций:

$$v = \mathrm{d}[\mathrm{P}] / \mathrm{d}t = V_{\max}[\mathrm{S}] / (K_m + [\mathrm{S}]),$$

где v – скорость реакции, преобразующей субстрат S в продукт P и катализируемой ферментом. $V_{\max}$ – максимальная скорость реакции (когда все ферменты связаны с субстратом S), а константа Михаэлиса K_m – это концентрация субстрата, при которой скорость реакции составляет половину от ее максимального значения.

Построить график зависимости v от [S] для реакции при $K_m = 0.04$ М и $V_{\max} = 0.1$ М/с. Если потребуются метки для осей координат, то загляните в следующий раздел.

33.1.3. Нормализованная гауссова функция, центрированная в точке $x = 0$, описывается формулой

$$g(x) = 1 / (\sigma\sqrt{(2\pi)}) \exp(-x^2 / 2\sigma^2).$$

Вывести формы графика этой функции и сравнить их при значениях стандартного отклонения $\sigma = 1$, 1.5 и 2.

3.2 Метки, надписи и настройка параметров графиков

3.2.1 Метки и надписи

Описание графика

Для каждой линии на графике можно создать метку, передавая строковый объект в аргументе `label`. Но эта метка не появится на графике, если не будет вызван метод `plt.legend` для добавления описания графика:

```
plt.plot(ax, ay1, label='sin^2(x)')
plt.legend()
plt.show()
```

Место расположения описания по умолчанию – верхний правый угол графика, но его можно изменить, если передать в метод `legend` аргумент `loc` с одним из строковых или целочисленных значений, приведенных в табл. 3.1.

Таблица 3.1. Спецификаторы места расположения описания графика

Строка	Целое число
'best'	0
'upper right'	1
'upper left'	2
'lower left'	3
'lower right'	4
'right'	5
'center left'	6
'center right'	7
'lower center'	8
'upper center'	9
'center'	10

Название графика и метки осей

Графику можно присвоить название, расположенное выше осей координат, с помощью вызова метода `plt.title` и передачи в него строки названия. Методы `plt.xlabel` и `plt.ylabel` управляют метками осей *x* и *y*: нужно просто передать метку как строку в эти методы. Необязательный дополнительный атрибут `fontsize` устанавливает размер шрифта в пунктах. Например, приведенный ниже код генерирует график, показанный на рис. 3.5.

```
t = np.linspace(0., 0.1, 1000)
Vp_uk , Vp_us = 230 * np.sqrt(2), 120 * np.sqrt(2)
f_uk , f_us = 50, 60
V_uk = Vp_uk * np.sin(2 * np.pi * f_uk * t)                    ❶
V_us = Vp_us * np.sin(2 * np.pi * f_us * t)
plt.plot(t*1000, V_uk , label='UK')                            ❷
plt.plot(t*1000, V_us , label='US')
plt.title('A comparison of AC voltages in the UK and US')
plt.xlabel('Time /ms', fontsize=16.)
plt.ylabel('Voltage /V', fontsize=16.)
plt.legend()
plt.show()
```

❶ Напряжение вычисляется как функция от времени (*t* в с) в Великобритании и в США, где действующее (эффективное) напряжение различно (230 В и 120 В соответственно; кроме того, эти значения умножаются на $\sqrt{2}$, чтобы получить двойные амплитудные значения напряжения). Также различна частота переменного тока (50 Гц и 60 Гц).

❷ Время отображается по оси *x* в мс (`t*1000`).

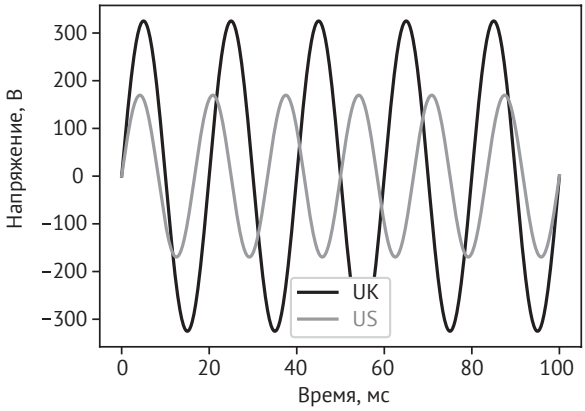


Рис. 3.5. Сравнение напряжений переменного тока в Великобритании и США

Использование LaTeX в *pyplot*

В графиках `pyplot` можно использовать язык разметки LaTeX, но для этого необходимо разрешить применение этой возможности в rc-настройках Matplotlib, как показано ниже:

```
plt.rc('text', usetex=True)
```

Затем нужно просто передать команду языка разметки LaTeX как строку в любую метку, выводимую этим способом. Рекомендуется использовать неформатируемые строки (`r'xxx'`), чтобы Python не экранировал символы с использованием обратных слешей, характерные для LaTeX (см. раздел 2.3.2).

Пример П3.3. Для вывода графиков функций $f_n(x) = x^n \sin x$ при $n = 1, 2, 3, 4$ выполняется следующий код:

```
import matplotlib.pyplot as plt
import numpy as np
plt.rc('text', usetex=True)

x = np.linspace(-10,10,1001)
for n in range(1,5):
    y = x**n * np.sin(x)
    y /= max(y)                                                    ❶
    plt.plot(x,y, label=r'$x^{}\sin x$'.format(n))
plt.legend(loc='lower center')
plt.show()
```

❶ Для упрощения сравнения графиков они промасштабированы до максимального значения 1 в рассматриваемой области значений.

Полученные графики показаны на рис. 3.6.

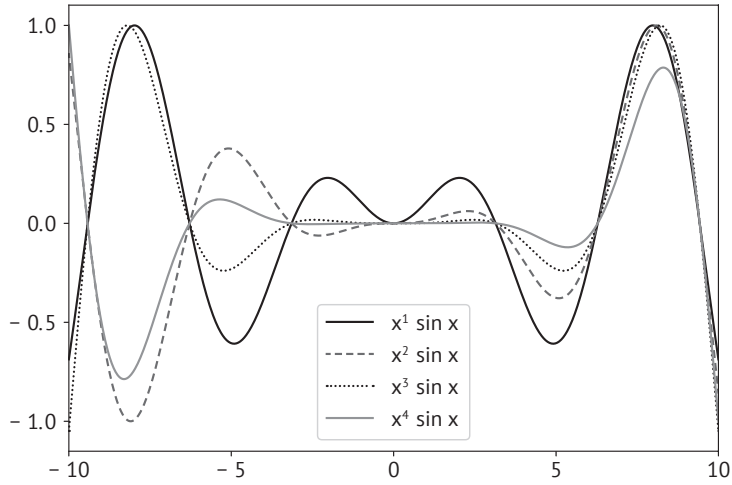


Рис. 3.6. Графики функций $f_n(x) = x^n \sin x$ при $n = 1, 2, 3, 4$

3.2.2 Графики со специализированными настройками параметров

Маркеры

По умолчанию метод `plot` генерирует линейный график без маркеров в отображаемых точках. Для добавления маркера в каждой точке данных на графике используется аргумент `marker`. Также можно определить несколько различных маркеров – это описано в онлайновой документации (https://matplotlib.org/api/markers_api.html). Некоторые из наиболее полезных маркеров перечислены в табл. 3.2.

Таблица 3.2. Некоторые стили маркеров библиотеки Matplotlib

Код	Маркер	Описание
`.`	·	Точка
`o`	○	Кружок
`+`	+	Знак плюс
`x`	×	Крестик
`D`	⋄	Ромбик
`v`	▽	Треугольник с вершиной вниз
`^`	△	Треугольник с вершиной вверх
`s`	□	Квадратик
`*`	★	Звездочка

Цвета

Цвет выводимой линии и/или ее маркеров можно определить с помощью аргумента `color`. Поддерживается несколько форматов определения цвета. Во-первых, однобуквенные коды для некоторых основных цветов приведены

в табл. 3.3. Например, `color='r'` определяет красную линию и маркеры. Эти цвета слишком яркие, поэтому (с версии Matplotlib 2.0) для последовательности линий на одном графике по умолчанию установлены более мягкие группы цветов Tableau (живой цвет), строки идентификаторов которых также приведены в табл. 3.3.

Таблица 3.3. Буквенные и строковые обозначения цветов в Matplotlib

Коды основных цветов	Живые (Tableau) цвета
`b` = синий	`tab:blue`
`g` = зеленый	`tab:orange`
`r` = красный	`tab:green`
`c` = бирюзовый	`tab:red`
`m` = фиолетовый	`tab:purple`
`y` = желтый	`tab:brown`
`k` = черный	`tab:pink`
`w` = белый	`tab:gray`
	tab:olive
	tab:cyan

Кроме того, можно определять оттенки серого цвета как строку, представляющую число с плавающей точкой `float` в диапазоне 0–1 (`0.` соответствует черному цвету, `1.` – белому). Шестнадцатеричные строки HTML, определяющие красную, зеленую и синюю (RGB) компоненты цвета в диапазоне `00-ff`, также могут передаваться в аргументе `color` (например, `color='#ff00ff'` определяет фиолетовый (magenta) цвет). Наконец, компоненты RGB можно передавать как кортеж (`tuple`) из трех значений в диапазоне 0–1 (например, `color=(0.5, 0., 0.)` – это темно-красный цвет).

Стили и ширина линий

По умолчанию принят стиль для графика: сплошная линия шириной 1.5 пункта. Для настройки этого параметра определяется аргумент `linestyle` (это тоже строка). Некоторые возможные значения, определяющие стиль линии, приведены в табл. 3.4.

Таблица 3.4. Стили линий Matplotlib

Код	Стиль линии
`-`	Сплошная
`--`	Штриховая
`:`	Пунктирная
`-.`	Штрихпунктирная

Чтобы линии вообще не отображались, устанавливается значение `linestyle=''` (пустая строка). Толщину линии можно задать в пунктах, передавая значение типа `float` в атрибуте `linewidth`. Например:

```
x = np.linspace(0.1, 1., 100)
yi = 1. / x
ye = 10. * np.exp(-2 * x)
plt.plot(x, yi, color='r', linestyle=':', linewidth=4.)
plt.plot(x, ye, color='m', linestyle='--', linewidth=2.)
plt.show()
```

Результат выполнения этого кода показан на рис. 3.7.

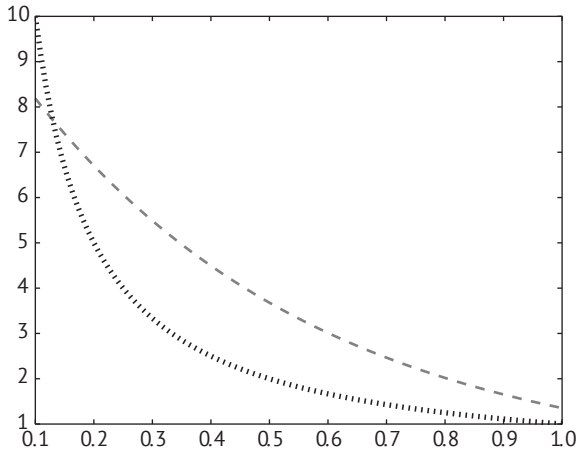


Рис. 3.7. Два различных стиля линий на одном графике

Кроме того, допустимыми являются следующие сокращенные обозначения свойств линий графика:

- `c` – `color` – цвет;
- `ls` – `linestyle` – стиль линии;
- `lw` – `linewidth` – ширина линии.

Например:

```
plt.plot(x, y, c='g', ls='--', lw=2)         # Утолщенная зеленая штриховая линия.
```

Также возможно определение цвета, стиля линии и маркера в одной строке:

```
plt.plot(x, y, 'r:^')         # Красная пунктирная линия с треугольными маркерами.
```

Наконец, можно изобразить несколько линий, используя последовательность аргументов `x`, `y`, `format`:

```
plt.plot(x, y1, 'r--', x, y2, 'k-.')
```

Выводится красная штриховая линия для `(x, y1)` и черная штрихпунктирная линия для `(x, y2)`.

Границы графика

Методы `plt.xlim` и `plt.ylim` определяют границы графика по осям *x* и *y* соответственно. Они должны вызываться только после всех инструкций `plt.plot`, но перед выводом или сохранением изображения. Например, следующий код создает график по предоставленным последовательностям данных между выбранными границами (см. рис. 3.8):

```
t = np.linspace(0, 2, 1000)
f = t * np.exp(t + np.sin(20*t))
plt.plot(t, f)
plt.xlim(1.5,1.8)
plt.ylim(0,30)
plt.show()
```

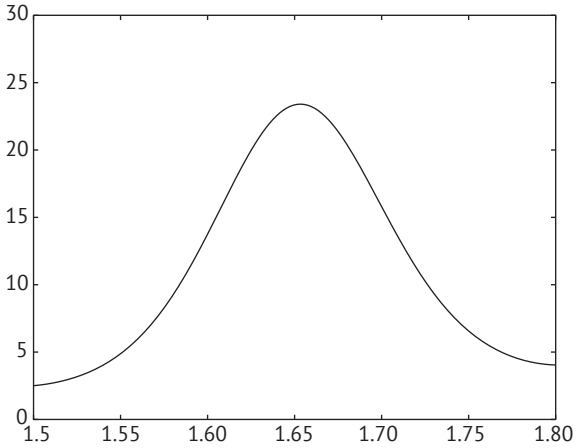


Рис. 3.8. График, созданный с явно определенными границами данных

Пример П3.4. Закон Мура основан на следующем наблюдении: количество транзисторов в микросхемах центральных процессорных устройств (CPU) приблизительно удваивается через каждые 2 года. Программа в листинге 3.2 демонстрирует действие этого закона с помощью сравнения между реальным количеством транзисторов в ЦПУ ведущих производителей с 1972 по 2012 г., затем выполняет прогнозирование по закону Мура, который можно записать в математической форме:

$$n_i = n_0\, 2^{(yi-y0)/T2},$$

где n_0 – количество транзисторов в некотором эталонном году y_0, а $T_2 = 2$ – число лет, необходимых для удваивания этого количества. Поскольку рассматриваются данные за 40 лет, значения n_i охватывают несколько порядков величины, поэтому удобнее применить закон Мура к логарифмам этих значений, что позволит показать линейную зависимость от *y*:

$$\log_{10} n_i = \log_{10} n_0 + ((y_i - y_0) / T_2) \log_{10} 2.$$

Листинг 3.2. Демонстрация действия закона Мура

```
# eg3-moore.py
import numpy as np
import matplotlib.pyplot as plt

# Данные - список лет:
year = [1972, 1974, 1978, 1982, 1985, 1989, 1993, 1997, 1999, 2000, 2003,
        2004, 2007, 2008, 2012]
# Количество транзисторов (ntrans) в ЦПУ в млн:
ntrans = [0.0025, 0.005, 0.029, 0.12, 0.275, 1.18, 3.1, 7.5, 24.0, 42.0,
          220.0, 592.0, 1720.0, 2046.0, 3100.0]
# Преобразование списка ntrans в массив NumPy и умножение каждого элемента на 1 млн.
ntrans = np.array(ntrans) * 1.e6

y0, n0 = year[0], ntrans[0]
# Линейный массив лет, охватывающий данные по отдельным годам.
y = np.linspace(y0, year[-1], year[-1] - y0 + 1)
# Время в годах, необходимое для удваивания количества транзисторов.
T2 = 2.
moore = np.log10(n0) + (y - y0) / T2 * np.log10(2)

plt.plot(year , np.log10(ntrans), '*', markersize=12, color='r',
         markeredgecolor='r', label='observed')
plt.plot(y, moore , linewidth=2, color='k', linestyle='--', label='predicted')
plt.legend(fontsize=16, loc='upper left')
plt.xlabel('Year')
plt.ylabel('log(ntrans)')
plt.title("Moore's law")
plt.show()
```

В этом примере данные содержатся в двух списках равной длины, представляющих год и характерное (репрезентативное) количество транзисторов в ЦПУ в этом году. Приведенная выше формула закона Мура реализуется в логарифмической форме с использованием массива лет, охватывающего предоставленные данные. (В действительности, поскольку на логарифмической шкале это будет прямая линия, вполне достаточно всего лишь двух точек данных.)

Для построения графика, показанного на рис. 3.9, точки данных выводятся в виде увеличенных звездочек, а прогноз по закону Мура изображается штриховой черной линией.

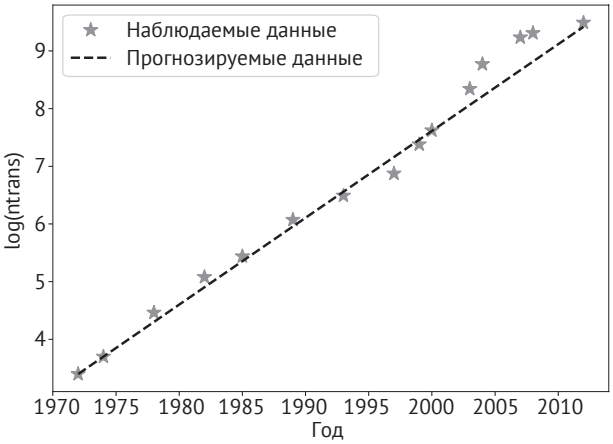


Рис. 3.9. Закон Мура, моделирующий экспоненциальный рост количества транзисторов в ЦПУ

3.2.3 Упражнения

Задачи

33.2.1. Молекула A вступает в реакцию для образования молекулы B или C с константами скорости реакции первого порядка k_1 и k_2 соответственно. Таким образом:

$$\mathrm{d[A]} / \mathrm{d}t = -(k_1 + k_2)[\mathrm{A}],$$

следовательно:

$$[\mathrm{A}] = [\mathrm{A}]_0\, e - (k_1 + k_2)t,$$

где $[\mathrm{A}]_0$ – начальная концентрация A. Значения концентрации продуктов (начиная с 0) увеличиваются в соответствии с отношением $[\mathrm{B}]/[\mathrm{C}] = k_1/k_2$, а закон сохранения вещества требует соблюдения условия $[\mathrm{B}] + [\mathrm{C}] = [\mathrm{A}]_0 - [\mathrm{A}]$. Следовательно:

$$[\mathrm{B}] = k_1 / (k_1 + k_2)\, [\mathrm{A}]_0\, (1 - e^{-(k_1+k_2)t});$$

$$[\mathrm{C}] = k_2 / (k_1 + k_2)\, [\mathrm{A}]_0\, (1 - e^{-(k_1+k_2)t}).$$

Для реакции с коэффициентами $k_1 = 300$ с$^{-1}$ и $k_2 = 100$ с$^{-1}$ вывести график значений концентрации A, B и C в зависимости от времени с учетом начальной концентрации реагента $[\mathrm{A}]_0 = 2.0$ моль/дм^3.

33.2.2. Гауссово целое число – это комплексное число, действительная и мнимая части которого являются целыми числами. Гауссово простое число – это гауссово целое комплексное число $x + iy$, такое, что для него выполняется одно из следующих условий:

- одно из чисел x или y равно нулю, а второе является простым числом в форме $4n + 3$ или $-(4n + 3)$ при некотором целом значении $n \geqslant 0$; или
- оба числа x и y не равны нулю и $x^2 + y^2$ – простое число.

Рассмотреть последовательность гауссовых целых чисел, получаемых при наблюдении за воображаемой частицей, изначально находящейся в состоянии c_0 и перемещающейся в комплексной плоскости по следующему закону: выполняется целочисленное количество шагов в текущем направлении (± 1 в действительном или мнимом направлении), но если встречается гауссово простое число, то выполняется поворот влево. Изначально частица ориентирована в положительном действительном направлении ($\Delta c = 1 + 0i \Rightarrow \Delta x = 1$, $\Delta y = 0$). Путь, который наблюдается при движении такой частицы, называется спиралью гауссовых простых чисел (Gaussian prime spiral).

Написать программу вывода спирали гауссовых простых чисел, начиная с $c_0 = 5 + 23i$.

33.2.3. Риск смерти, оцениваемый за год (принимаемый как «1 к N»), для мужчин и женщин в Великобритании в 2005 г. для различных возрастных групп приведен в табл. 3.5. Использовать метод `pyplot` для вывода этих данных на одном графике.

Таблица 3.5

Возрастные группы	Женщины	Мужчины
< 1	227	177
1–4	5376	4386
5–14	10 417	8333
15–24	4132	1908
25–34	2488	1215
35–44	1106	663
45–54	421	279
55–64	178	112
65–74	65	42
75–84	21	15
> 84	7	6

3.3 Построение более сложных графиков

3.3.1 График в полярных координатах

Метод `pyplot.plot` создает график в декартовой системе координат (x,y). Для получения графика в полярных координатах (r,θ) используется метод `pyplot.polar`, в который передаются аргументы `theta` (обычно это независимая переменная) и `r`.

Пример П3.5. Кардиоида – это плоская фигура, описываемая в полярных координатах уравнением $r = 2a(1 + \cos\theta)$ при $0 \leqslant \theta \leqslant 2\pi$:

```
theta = np.linspace(0, 2.*np.pi, 1000)
a = 1.
r = 2 * a * (1. + np.cos(theta))
plt.polar(theta , r)
plt.show()
```

Созданный этим фрагментом кода график в полярных координатах показан на рис. 3.10.

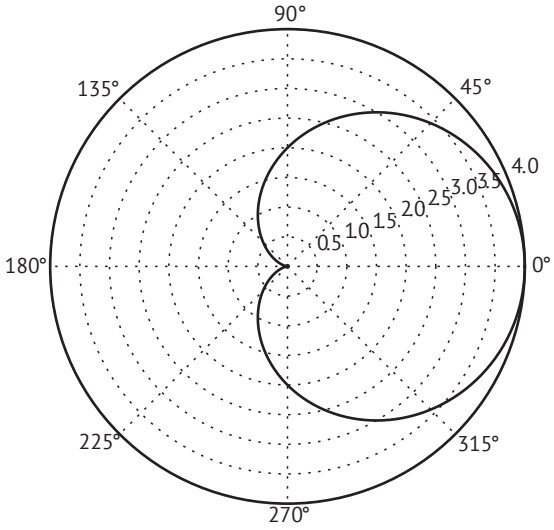


Рис. 3.10. Кардиоида при $a = 1$

3.3.2 Гистограммы

Гистограмма представляет распределение данных в виде последовательных (обычно вертикальных) полос, длина которых пропорциональна числовым значениям элементов данных, входящим в предварительно определенные диапазоны (известные как интервалы гистограммы (bins)). Таким образом, диапазон значений данных делится на интервалы, и гистограмма формируется посредством подсчета количества значений данных в каждом интервале.

В библиотеке `pyplot` функция `hist` создает гистограмму из последовательности значений данных. Количество интервалов можно передать как дополнитель-

ный необязательный аргумент `bins`, по умолчанию его значение равно 10. Кроме того, по умолчанию высоты полос гистограммы равны абсолютным величинам данных в соответствующем интервале, но установка атрибута `density=True` нормализует гистограмму так, что ее область (произведение высоты на ширину каждой полосы, просуммированное по всем полосам) становится единообразной.

Например, для 5000 случайных значений из нормального распределения со средним значением 0 и стандартным отклонением 2 (см. раздел 4.5.1):

```
>>> import matplotlib.pyplot as plt
>>> import random
>>> data = []
>>> for i in range(5000):
...     data.append(random.normalvariate(0, 2))
>>> plt.hist(data , bins=20, density=True)
>>> plt.show()
```

Полученная гистограмма изображена на рис. 3.11.

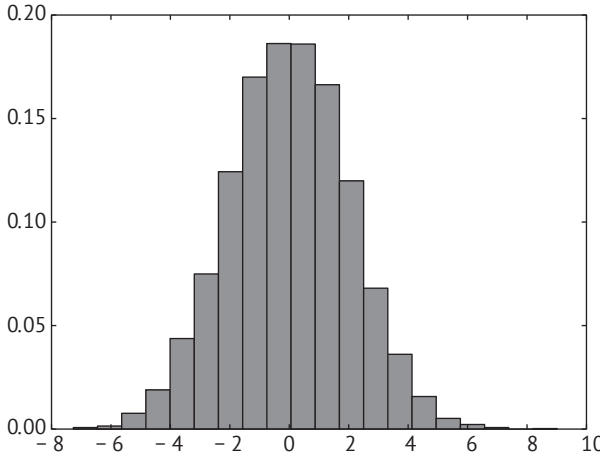


Рис. 3.11. Гистограмма случайных данных с нормальным распределением

3.3.3 Дополнительные оси

Команда `pyplot.twinx()` инициализирует новый набор осей с осью *x*, совпадающей с первоначальной, и новой осью *y*. Это удобно для вывода двух и более последовательностей данных, которые совместно используют ось абсцисс (ось *x*), но значения по оси *y* существенно отличаются по величине или представлены в других единицах измерения. Рассмотрим следующий пример.

Пример П3.6. По данным https://tylervigen.com/ существует странная и совершенно нелепая зависимость, наблюдаемая во времени, между количеством разводов в штате Мэн (США) и потреблением маргарина на душу населения во всей стране. Приведенные здесь две временны́е последовательности представлены в различных единицах измерения и имеют разный смысл, поэтому они должны отображаться по отдельным осям *y*, но совместно использовать общую ось *x* (где отмечены годы).

Листинг 3.3. Зависимость между потреблением маргарина в США и количеством разводов в штате Мэн

```
# eg3-margarine -divorce.py
import matplotlib.pyplot as plt

years = range(2000, 2010)
divorce_rate = [5.0, 4.7, 4.6, 4.4, 4.3, 4.1, 4.2, 4.2, 4.2, 4.1]
margarine_consumption = [8.2, 7, 6.5, 5.3, 5.2, 4, 4.6, 4.5, 4.2, 3.7]

line1 = plt.plot(years , divorce_rate , 'b-o', label='Divorce rate in Maine')          ❶
plt.ylabel('Divorces per 1000 people')
plt.legend()

plt.twinx()
line2 = plt.plot(years , margarine_consumption , 'r-o', label='Margarine consumption')
plt.ylabel('lb of Margarine (per capita)')

# Мы преодолели все препятствия, чтобы создать метки в том же самом блоке описания:
lines = line1 + line2                                                                  ❷
labels = []
for line in lines:
    labels.append(line.get_label())                                                    ❸

plt.legend(lines , labels)
plt.show()
```

Пришлось проделать некоторую дополнительную работу, чтобы создать описание, в которое включены обе строки надписей для этого графика:

❶ Метод `pyplot.plot` возвращает список объектов, представляющих выводимые строки, поэтому они сохраняются как `line1` и `line2`.

❷ Эти строки объединяются.

❸ Выполняется проход в цикле по объединенным строкам, чтобы извлечь требуемые надписи. Далее список строк и надписей можно передать непосредственно в метод `pyplot.legend`.

Результатом выполнения кода из листинга 3.3 является график, показанный на рис. 3.12.

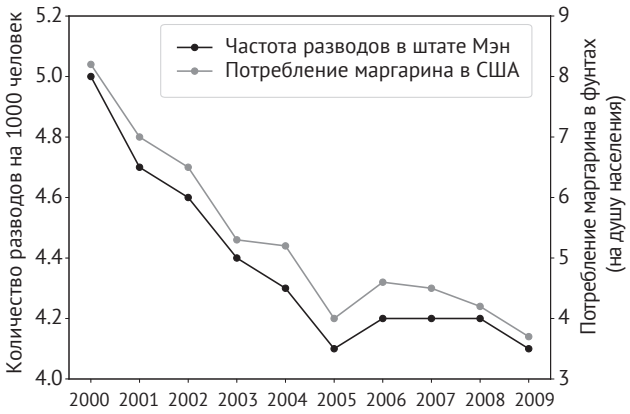


Рис. 3.12. Зависимость между количеством разводов в штате Мэн и потреблением маргарина на душу населения в США

3.3.4 Упражнения

Задачи

33.3.1. Спираль можно рассматривать как фигуру, описывающую траекторию движения точки по воображаемой прямой при одновременном вращении этой прямой относительно начала координат с постоянной угловой скоростью. Если точка закреплена на прямой, то описывающей фигурой будет окружность.

а) Если точка на вращающейся прямой перемещается от начала координат с постоянной скоростью, то ее траектория соответствует архимедовой спирали. В полярных координатах уравнение архимедовой спирали записывается так: $r = a + b\theta$. Использовать библиотеку `pyplot` для построения графика спирали, определяемой параметрами $a = 0$, $b = 2$ при $0 \leqslant \theta \leqslant 8\pi$.

б) Если точка перемещается по вращающейся прямой со скоростью, увеличивающейся пропорционально расстоянию от начала координат, то результатом является логарифмическая спираль, уравнение которой имеет следующий вид: $r = a^{\theta}$. Построить график логарифмической спирали, определяемой параметром $a = 0.8$, при $0 \leqslant \theta \leqslant 8\pi$. Логарифмическая спираль обладает свойством самоподобия: на каждом 2π витке спираль увеличивает размер, но сохраняет форму[33]. Логарифмические спирали часто встречаются в природе: от форм раковин моллюсков-наутилусов до форм космических галактик.

33.3.2. Простая модель взаимодействия потенциалов двух атомов как функция расстояния между ними r описывается формулой Леннарда–Джонса (Lennard-Jones) (потенциал Леннарда–Джонса):

$$U(r) = B/r^{12} - A/r^{6},$$

где A и B – положительные константы[34].

Для атомов аргона эти константы можно принять равными $A = 1.024 \times 10^{-23}$ Дж×нм^6 и $B = 1.582 \times 10^{-26}$ Дж×нм^{12}.

а) Построить график $U(r)$. На второй оси у того же изображения построить график сил межатомного взаимодействия:

$$F(r) = -dU/dr = 12B/r^{13} - 6A/r^{7}.$$

График должен показывать «самую интересную» часть этих кривых, поскольку вычисляемые значения чрезвычайно быстро увеличиваются при малых значениях r.

---

[33] Швейцарский математик Якоб Бернулли (Jakob Bernoulli) был так впечатлен этим свойством, что назвал логарифмическую спираль Spira mirabilis (чудесной спиралью) и пожелал, чтобы на его надгробном камне была выгравирована логарифмическая спираль с надписью «Eadem mutata resurgo» («Даже изменяясь, я остаюсь собой»). К сожалению, на надгробный камень Бернулли по ошибке была помещена архимедова спираль.

[34] Такая форма записи была широко распространена во времена зарождения вычислительной техники, потому что r^{-12} легко вычисляется как квадрат r^{-6}.

Совет: вычисления можно упростить, если разделить A и B на постоянную Больцмана 1.381×10^{-23} Дж/К, тогда $U(r)$ будет измеряться в К (кельвинах). Какова глубина потенциальной ямы ε и расстояние минимума потенциала r_0 для этой системы?

б) При малых отклонениях от уравновешенного межатомного расстояния (на котором $F = 0$) потенциал можно приблизительно представить с помощью функции гармонического осциллятора:

$$V(r) = \tfrac{1}{2}k(r - r_0)^2 + \varepsilon,$$

где $k = |d^2U/dr^2|_{r0} = 156B/r_0^{14} - 42A/r_0^{8}$.

Построить графики $U(r)$ и $V(r)$ на одной диаграмме.

33.3.3. Семенную шапку подсолнечника можно смоделировать следующим образом. Определяется число n семян как $s = 1, 2, \ldots, n$, и каждое семя помещается на расстоянии $r = \sqrt{s}$ от центра с поворотом на угол $\theta = 2\pi s/\phi$ относительно оси x, где ϕ – некоторая константа. Природа выбрала для ϕ золотое сечение $\phi = (1 + \sqrt{5})/2$, обеспечивающее максимально плотную упаковку семян при росте семенной шапки подсолнечника.

Написать программу на Python для построения модели семенной шапки подсолнечника. (Совет: использовать полярные координаты.)

Глава 4

Ядро языка Python II

В этой главе продолжается введение в основные функциональные возможности языка Python, начатое в главе 2. Здесь рассматриваются обработка ошибок с помощью исключений, структуры данных, известные как словари и множества, некоторые удобные и эффективные приемы решения часто встречающихся задач, а также приводится краткий обзор некоторых модулей из стандартной библиотеки Python Standard Library. В конце главы представлено краткое введение в объектно-ориентированное программирование с точки зрения языка Python.

4.1 Ошибки и исключения

Python различает два типа ошибок: синтаксические ошибки и прочие исключения (exceptions). Синтаксические ошибки – это ошибки в грамматике языка, которые выявляются перед выполнением программы. Исключения – это ошибки времени выполнения (runtime errors): обычно они возникают при попытках выполнения недопустимой операции с некоторым элементом данных. Различие заключается в том, что синтаксические ошибки всегда являются критическими: компилятор Python не в силах что-либо сделать, если программа не соответствует грамматике языка. Исключения – это условия, возникающие во время выполнения Python-программы (например, при попытке деления на ноль), поэтому существует механизм «перехвата» и аккуратной обработки этих условий без прекращения выполнения программы.

4.1.1 Синтаксические ошибки

Синтаксические ошибки обнаруживаются компилятором Python, при этом выводится сообщение с указанием места обнаружения ошибки. Например:

```
>>> for lambda in range(8):
  File "<stdin>", line 1
    for lambda in range(8):
              ^
SyntaxError: invalid syntax
```

Так как `lambda` – зарезервированное ключевое слово, его нельзя использовать как имя переменной. Компилятор обнаружил его там, где ожидалось имя переменной, поэтому возникла синтаксическая ошибка. Другой случай:

```
>>> for f in range(8:
  File "<stdin>", line 1
    for f in range(8:
                   ^
SyntaxError: invalid syntax
```

Здесь синтаксическая ошибка возникла, потому что во встроенную функцию `range` должен быть передан аргумент как целое число в круглых скобках: двоеточие нарушает синтаксис вызова функций, поэтому Python диагностирует синтаксическую ошибку.

Поскольку строки кода Python могут быть разделены внутри скобок (`"()"`, `"[]"`, `"{}"`), инструкция, разделенная на несколько строк, иногда может приводить к выводу сообщения `SynatxError` в некотором месте, отличающемся от настоящего места расположения ошибки, например:

```
>>> a = [1, 2, 3, 4,
... b = 5
  File "<stdin >", line 4
    b = 5
    ^
SyntaxError: invalid syntax
```

Здесь инструкция `b = 5` синтаксически правильная: ошибка возникает из-за отсутствия закрывающей квадратной скобки в предыдущем определении списка – командная оболочка Python считает, что строка является продолжением предыдущей строки, и обозначает это многоточием в начале строки (...).

Существуют два типа синтаксических ошибок `SyntaxError`, заслуживающих особого внимания: ошибка `IndentationError` возникает при неправильном форматировании (со сдвигом вправо) блока кода, а ошибка `TabError` возникает, когда символы табуляции и пробелы используются совместно, но несогласованно для сдвига блока кода[35].

Пример П4.1. Самая частая синтаксическая ошибка, с которой встречаются начинающие программисты на Python, – использование оператора присваивания `=` вместо оператора сравнения на равенство `==` в условном выражении:

```
>>> if a = 5:
  File "<stdin >", line 1
    if a = 5:
         ^
SyntaxError: invalid syntax
```

Это присваивание `a = 5` не возвращает значение (данная операция просто присваивает целочисленный объект `5` имени переменной `a`), следовательно, здесь отсутствует объект, соответствующий логическому значению `True` или `False`, который может использовать оператор `if`, следовательно, возникает синтаксическая ошибка `SyntaxError`. Это полная противоположность концепции, принятой в языке C, в котором операция присваивания возвращает

[35] Этой ошибки можно избежать, если для форматирования кода использовать только пробелы.

значение, присваиваемое переменной (т. е. инструкция присваивания `a = 5` вычисляет значение 5, отличное от нуля, следовательно, равнозначное логическому значению `True`[36]). Такое поведение является источником многих ошибок, которые очень трудно обнаружить, и уязвимостей с точки зрения безопасности, поэтому оно преднамеренно исключено из языка Python на этапе проектирования.

4.1.2 Исключения

Исключение (exception) возникает при выполнении синтаксически правильного выражения, в котором встречается ошибка времени выполнения (runtime error). Существуют различные типы встроенных исключений, а кроме того, возможны специализированные исключения, определяемые программистом, если это необходимо. Если исключение не выполняет «перехват» с использованием конструкции `try…except`, описанной ниже, то Python выводит сообщение об ошибке (обычно осмысленное и полезное). Если исключение возникает в теле функции (которая, возможно, в свою очередь, вызвана другой функцией и т. д.), то предъявляемое сообщение принимает форму трассировки стека в обратном направлении (stack traceback): выводится хронология вызовов функций, которые привели к ошибке, так что можно определить место ее возникновения при выполнении программы.

С некоторыми встроенными исключениями вы уже знакомы – они встречались при использовании Python в предыдущих главах.

Исключение *NameError*

```
>>> print('4z = ', 4*z)

Traceback (most recent call last):
  File "<stdin >", line 1, in <module >
NameError: name 'z' is not defined
```

Исключение `NameError` возникает, когда используется имя переменной, которое не было ранее определено: в приведенном примере команда `print` правильная, но Python не знает, на что указывает идентификатор `z`.

Исключение *ZeroDivisionError*

```
>>> a, b = 0, 5
>>> b / a

Traceback (most recent call last):
  File "<stdin >", line 1, in <module >
ZeroDivisionError: float division by zero
```

Деление на ноль не является определенной математической операцией.

[36] В языке C нет встроенных логических значений `True` и `False`. Любое отличающееся от нуля значение считается истинным, а нулевое значение – ложным. – *Прим. перев.*

Исключения *TypeError* и *ValueError*

Исключение `TypeError` генерируется, если в выражении или в функции используется некорректный тип. Например:

```
>>> '00' + 7

Traceback (most recent call last):
  File "<stdin >", line 1, in <module >
TypeError: Can't convert 'int' object to str implicitly
```

Python – (достаточно) строго типизированный язык, поэтому он не позволяет добавлять строку к целому числу[37].

С другой стороны, исключение `ValueError` возникает, когда используемый объект имеет правильный тип, но недопустимое значение:

```
>>> float('hello')

Traceback (most recent call last):
  File "<stdin >", line 1, in <module >
ValueError: could not convert string to float: 'hello'
```

Встроенная функция `float` принимает строку как аргумент, поэтому выражение `float('hello')` не приводит к ошибке `TypeError`: исключение возникает, потому что конкретная строка `'hello'` не может быть преобразована в осмысленное число с плавающей точкой. Более сложный случай:

```
>>> int('7.0')

Traceback (most recent call last):
  File "<stdin >", line 1, in <module >
ValueError: invalid literal for int() with base 10: '7.0'
```

Строка, выглядящая как число типа `float`, не может быть напрямую преобразована в тип `int`: для получения требуемого результата необходимо использовать выражение `int(float('7.0'))`.

В табл. 4.1 приведен список наиболее часто встречающихся встроенных исключений с краткими описаниями.

Таблица 4.1. Часто встречающиеся исключения Python

Исключение	Причина и описание
`FileNotFoundError`	Попытка открыть файл или каталог, который не существует, – это исключение представляет конкретный тип ошибки `OSError`
`IndexError`	Индексирование последовательности (например, списка или строки) с использованием индекса, выходящего за пределы диапазона
`KeyError`	Обращение по индексу к словарю с использованием значения ключа, которое не существует в этом словаре (см. раздел 4.2.2)
`NameError`	Ссылка на локальное или глобальное имя переменной, которое не определено предварительно
`TypeError`	Попытка использования объекта несоответствующего типа как аргумента встроенной операции или функции

[37] В отличие, например, от языков JavaScript и PHP, где такое добавление допускается.

Окончание табл. 4.1

Исключение	Причина и описание
`ValueError`	Попытка использования объекта корректного типа, но имеющего недопустимое значение, как аргумента встроенной операции или функции
`ZeroDivisionError`	Попытка деления на ноль (либо явно (с использованием оператора / или //), либо как части операции получения остатка %)
`SystemExit`	Генерируется функцией `sys.exit` (см. раздел 4.4.1) – если это исключение не обрабатывается, то функция `sys.exit` выполняет выход из интерпретатора Python

Пример П4.2. Если исключение сгенерировано, но не обработано (см. раздел 4.1.3), то Python выводит отчет с обратной трассировкой (traceback report), показывающий, где именно в потоке выполнения программы возникла данная ошибка. Это особенно удобно, когда ошибка возникает во вложенных функциях или внутри импортированных модулей. Например, рассмотрим следующую короткую программу[38]:

```
# exception-test.py
import math

def func(x):
    def trig(x):
        for f in (math.sin, math.cos, math.tan):
            print('{f}({x}) = {res}'.format(f=f.__name__ , x=x, res=f(x)))
    def invtrig(x):
        for f in (math.asin , math.acos , math.atan):
            print('{f}({x}) = {res}'.format(f=f.__name__ , x=x, res=f(x)))      ❶
    trig(x)
    invtrig(x)                                                                  ❷

func(1.2)                                                                       ❸
```

Функция `func` передает свой аргумент `x` в две вложенные в нее функции. Первая вложенная функция `trig` выполняется без проблем, но во второй вложенной функции `invtrig` очевидно, что значение `x` находится вне домена (диапазона допустимых значений) для обратной тригонометрической функции арксинус `asin`:

```
sin(1.2) = 0.9320390859672263
cos(1.2) = 0.3623577544766736
tan(1.2) = 2.5721516221263183
Traceback (most recent call last):
  File "exception-test.py", line 14, in <module>
    func(1.2)
  File "exception-test.py", line 12, in func
    invtrig(x)
  File "exception-test.py", line 10, in invtrig
    print('{f}({x}) = {res}'.format(f=f.__name__, x=x, res=f(x)))
ValueError: math domain error
```

[38] Обратите внимание на использование конструкции `f.__name__` для возврата строки, представляющей имя функции в этой программе, например `math.sin.__name__` соответствует имени функции `sin`.

Последовательное изучение трассировки в обратном порядке показывает, что исключение `ValueError` было сгенерировано внутри вложенной функции `invtrig` (строка 10, ❶), которая была вызвана из функции `func` (строка 12, ❷), в свою очередь вызванной из модуля `exception-test.py` (т. е. из программы) в строке 14, ❸.

4.1.3 Обработка и генерация исключений

Обработка исключений

Часто программа должна обрабатывать данные таким способом, который может стать причиной возникновения исключения. Предположим, что существует такое условие, которое не приводит к аварийному завершению программы с ошибкой, но требует «аккуратной» обработки в определенном смысле (некорректные точки данных игнорируются, результат деления на ноль отбрасывается и т. д.). В подобной ситуации возможно применение двух методик: проверка значения объекта данных перед его использованием или обработка любого сгенерированного исключения перед возобновлением выполнения программы. В Python применяется вторая методика, краткой характеристикой которой является выражение EAFP: «It is Easier to Ask Forgiveness than to seek Permission» (Проще попросить прощения, чем пытаться получить разрешение).

Для перехвата исключения в блоке кода необходимо поместить этот блок кода в конструкцию `try:` , а код обработки любых сгенерированных исключений – в конструкцию `except:`. Например:

```
try:
    y = 1 / x
    print('1 /', x, ' = ',y)
except ZeroDivisionError:
    print('1 / 0 is not defined.')
# ... Другие инструкции.
```

Не требуется никаких проверок: просто продолжается выполнение и вычисляется выражение `1/x`, а ошибка, возникающая при делении на ноль, обрабатывается при необходимости. Выполнение программы продолжается после блока `except` вне зависимости от того, было сгенерировано исключение `ZeroDivisionError` или нет. Если возникает другое исключение (например, `NameError`, если переменная `x` не определена), то оно не будет перехвачено – это необработанное исключение (unhandled exception), поэтому выводится сообщение об ошибке.

Для обработки более одного исключения в одном блоке except необходимо перечислить требуемые исключения в кортеже (обязательно в скобках).

```
try:
    y = 1. / x
    print('1 /', x, ' = ',y)
except (ZeroDivisionError , NameError):
    print('x is zero or undefined!)
# ... Другие инструкции.
```

Для отдельной обработки каждого исключения требуется несколько блоков except:

```
try:
    y = 1. / x
    print('1 /', x, ' = ',y)
except ZeroDivisionError:
    print('1 / 0 is not defined.')
except NameError:
    print('x is not defined')
# ... Другие инструкции.
```

Предупреждение: может встречаться следующий тип конструкции:

```
try:
    [выполняются какие-то инструкции]
except:               # Никогда так не делайте!
    pass
```

Здесь выполняются инструкции в блоке try, но игнорируются любые сгенерированные исключения – вообще говоря, это чрезвычайно неразумное решение, так как подобный код очень трудно сопровождать и отлаживать (если в таком коде возникают ошибки, то вы не получите никакой информации о них). Главная цель – перехват конкретных исключений и их правильная обработка, позволяющая «проявляться» любым другим исключениям, чтобы их также можно было обработать (или не обрабатывать) другими блоками except.

В конструкции try…except имеются два дополнительных необязательных ключевых слова (которые при необходимости должны следовать за всеми существующими блоками except). Инструкции в блоке ключевого слова finally выполняются всегда вне зависимости от того, было сгенерировано исключение или нет. Инструкции в блоке ключевого слова else выполняются, только если исключение не было сгенерировано (см. пример П4.5).

◊ Генерация исключений

Обычно исключение генерируется интерпретатором Python как результат некоторого (предусмотренного или непредвиденного) поведения программы. Но иногда требуется, чтобы программа сама сгенерировала конкретное исключение при выполнении определенного условия. Ключевое слово raise позволяет программе принудительно сгенерировать специальное исключение и определить особое сообщение или другие данные, связанные с этим исключением. Например:

```
if n % 2:
    raise ValueError('n must be even!')
# Здесь можно продолжать выполнение инструкций, точно зная, что n - четное число.
```

Связанное с raise ключевое слово assert вычисляет условное выражение и генерирует исключение AssertionError, если при вычислении условного выражения не получен результат, равнозначный True. Инструкции assert могут

оказаться полезными для проверки некоторого весьма важного условия в конкретный момент выполнения программы, что часто удобно при отладке.

```
>>> assert 2 == 2  # [Нет исключения]: 2 == 2 - результат True, поэтому ничего не происходит.
>>>
>>> assert 1 == 2  # Будет сгенерировано исключение AssertionError.

Traceback (most recent call last):
  File "<stdin >", line 1, in <module >
AssertionError
```

Синтаксическая конструкция `assert` *`expr1`*, *`expr2`* передает выражение *`expr2`* (обычно сообщение об ошибке) в исключение `AssertionError`:

```
>>> assert 1 == 2, 'One does not equal two'

Traceback (most recent call last):
  File "<stdin >", line 1, in <module >
AssertionError: One does not equal two
```

Python – язык с динамической типизацией, поэтому допустима передача аргументов любого типа в функцию, даже если эта функция ожидает аргумент конкретного типа. Иногда необходимо проверить корректность типа объекта аргумента перед его использованием, и для этого также можно использовать конструкцию `assert`.

Пример П4.3. Приведенная ниже функция возвращает представление в виде строки двумерного (2D) или трехмерного (3D) вектора, который обязательно должен быть представлен как список (list) или кортеж (tuple), содержащий два или три элемента.

```
>>> def str_vector(v):
...     assert type(v) is list or type(v) is tuple ,\
...                 'argument to str_vector must be a list or tuple'
...     assert len(v) in (2, 3),\
...                 'vector must be 2D or 3D in str_vector'
...     unit_vectors = ['i', 'j', 'k']
...     s = []
...     for i, component in enumerate(v):
...         s.append('{}{}'.format(component , unit_vectors[i]))
...     return '+'.join(s).replace('+-', '-')                          ❶
```

❶ Здесь метод `replace('+-', '-')` выполняет преобразование, например `'4i+-3j'` в `'4i-3j'`.

Пример П4.4. Еще один пример: предположим, что имеется функция, вычисляющая векторное произведение двух векторов, представленных как объекты типа `list`. Это произведение определено только для трехмерных векторов, поэтому его вызов с передачей списков любой другой длины приводит к ошибке.

```
>>> def cross_product(a, b):
...     assert len(a) == len(b) == 3, 'Vectors a, b must be three-dimensional'
...     return [a[1]*b[2] - a[2]*b[1],
...             a[2]*b[0] - a[0]*b[2],
...             a[0]*b[1] - a[1]*b[0]]
...
>>> cross_product([1, 2, -1], [2, 0, -1, 3]) # Ошибка.

Traceback (most recent call last):
  File "<stdin >", line 1, in <module >
  File "<stdin >", line 2, in cross_product
AssertionError: Vectors a, b must be three-dimensional
                [Векторы a, b обязательно должны быть трехмерными]

>>> cross_product([1, 2, -1], [2, 0, -1])
[-2, -1, -4]
```

Пример П4.5. Ниже приведен пример использования полной конструкции `try…except…else…finally`:

```
# try-except-else-finally.py

def process_file(filename):
    try:
        fi = open(filename , 'r')
    except IOError:
        print('Oops: couldn\'t open {} for reading'.format(filename))
        return
    else:
        lines = fi.readlines()                                          ❶
        print('{} has {} lines.'.format(filename , len(lines)))
        fi.close()
    finally:
        print(' Done with file {}'.format(filename))                    ❷

    print('The first line of {} is:\n{}'.format(filename , lines[0]))
    # Дальнейшая обработка строк.
    return

process_file('sonnet0.txt')
process_file('sonnet18.txt')
```

❶ Внутри блока `else` содержимое файла считывается, только если файл был успешно открыт.

❷ Внутри блока `finally` сообщение `'Done with file` *`filename`*`'` (`Обработан файл filename`) выводится вне зависимости от того, был файл успешно открыт или нет.

Предположим, что файл *sonnet0.txt* не существует, а файл *sonnet18.txt* существует, тогда при выполнении этой программы выводится следующий результат:

```
Oops: couldn't open sonnet0.txt for reading
   Done with file sonnet0.txt
sonnet18.txt has 14 lines.
   Done with file sonnet18.txt
The first line of sonnet18.txt is:
Shall I compare thee to a summer's day?
```

4.1.4 Упражнения

Вопросы

B4.1.1. Какой смысл имеет ключевое слово `else`? Почему бы не поместить инструкции блока `else` в начальный блок инструкций `try`?

B4.1.2. Какой смысл имеет ключевое слово `finally`? Почему бы не поместить инструкции, которые нужно обязательно выполнить, после блока `try` (вне зависимости от того, было сгенерировано исключение или нет) после всей конструкции `try…except`?

Совет: необходимо рассмотреть, что происходит, если изменить код примера П4.5, поместив инструкции из блока `finally` после блока `try`.

Задачи

34.1.1. Написать программу для считывания данных из файла *swallow-speeds.txt* (файл доступен здесь: https://scipython.com/ex/bda) и использования этих данных для вычисления средней скорости (свободного) полета африканской ласточки. Использовать обработку исключений при работе со строками, содержащими некорректные точки данных.

34.1.2. Изменить функцию из примера П4.3, которая возвращает вектор в показанной ниже форме:

```
>>> print(str_vector([-2, 3.5]))
-2i + 3.5j
>>> print(str_vector((4, 0.5, -2)))
4i + 0.5j - 2k
```

чтобы генерировалось исключение, если какой-либо элемент в массиве вектора не является действительным числом.

34.1.3. Python соблюдает соглашение, принятое во многих языках программирования при выборе определения значения $0^0 = 1$. Написать функцию `powr(a,b)`, поведение которой почти полностью совпадает с поведением выражения `a**b` (или для рассматриваемого здесь случая – `math.pow(a,b)`), но генерирует исключение `ValueError`, если `a` и `b` равны нулю.

4.2 Объекты Python III: словари и множества

В Python словарь (dictionary) – это тип «ассоциативного массива» (в некоторых языках этот тип обозначается термином «хеш» (hash)). Словарь может содержать любые объекты как значения (values), но, в отличие от таких последовательностей, как списки и кортежи, в которых элементы индексируются целыми числами, начиная с 0, каждый элемент в словаре индексируется неповторяющимся ключом (key), который может быть любым неизменяемым объектом[39]. Таким образом, словарь существует как набор пар ключ-значение (key-value). Сами по себе словари являются изменяемыми объектами.

4.2.1 Определение и индексирование словаря

Словарь можно определить с помощью пар `key: value`, записанных в фигурных скобках:

```
>>> height = {'Burj Khalifa': 828., 'One World Trade Center': 541.3,
              'Mercury City Tower': -1., 'Q1': 323.,
              'Carlton Centre': 223., 'Gran Torre Santiago': 300.,
              'Mercury City Tower': 339.}
>>> height
{'Burj Khalifa': 828.0,
 'One World Trade Center': 541.3,
 'Mercury City Tower': 339.0,
 'Q1': 323.0,
 'Carlton Centre': 223.0,
 'Gran Torre Santiago': 300.0}
```

Команда `print(height)` возвращает словарь в том же формате (в фигурных скобках). Если один и тот же ключ связан с различными значениями (как `'Mercury City Tower'` в приведенном выше примере), то сохраняется только самое последнее значение: ключи в словаре не должны повторяться.

До версии Python 3.6 не обеспечивалось размещение элементов в словаре в каком-либо определенном порядке. В версии 3.6 (и более поздних) сохраняется порядок вставки элементов. Следует отметить, что переопределение значения, связанного с ключом, как в показанном выше примере, не изменяет порядок вставки ключа: `'Mercury City Tower'` – ключ, определенный третьим, когда ему было присвоено значение `-1.`, позже с этим ключом было связано другое значение `339.`, но он остался в третьей позиции при использовании словаря.

Отдельный элемент можно извлечь по его ключу, применяемому в качестве индекса, или по литеральному значению (`'Q1'`), или с помощью переменной, значение которой равно ключу:

---

[39] В действительности ключом словаря может быть любой хешируемый объект: в Python хешируемый объект – это объект с особым методом для генерации конкретного целого числа из любого экземпляра этого объекта. Смысл в том, что экземпляры такого объекта (объект может быть большим и сложным), которые сопоставляются как равные, должны иметь числовые хеш-значения, также сопоставляемые как равные, поэтому их можно быстро найти в хеш-таблице. Это важно для некоторых структур данных и для оптимизации скорости алгоритмов, работающих с подобными объектами.

```
>>> height['One World Trade Center']
541.3
>>> building = 'Carlton Centre'
>>> height[building]
223.0
```

Элементы (значения) в словаре также можно присваивать по индексу, как показано ниже:

```
height['Empire State Building'] = 381.
height['The Shard'] = 306.
```

Другой способ определения словаря – передача последовательности пар (*ключ, значение*) в конструктор `dict`. Если ключами являются простые строки (которые могут использоваться как имена переменных), то пары можно также определять как именованные аргументы для конструктора `dict`:

```
>>> ordinal = dict([(1, 'First'), (2, 'Second'), (3, 'Third')])
>>> mass = dict(Mercury=3.301e23, Venus=4.867e24, Earth=5.972e24)
>>> ordinal[2]              # Обратите внимание: здесь 2 - ключ, а не индекс.
'Second'
>>> mass['Earth']
5.972e+24
```

Итеративный проход в цикле `for` по словарю возвращает ключи словаря (в порядке вставки ключей):

```
>>> for c in ordinal:
...     print(c, ordinal[c])
...
1 First
2 Second
3 Third
```

Пример П4.6. Простой словарь римских цифр:

```
>>> numerals = {'one':'I', 'two':'II', 'three':'III', 'four':'IV', 'five':'V',
                'six':'VI', 'seven':'VII', 'eight':'VIII',
                1: 'I', 2: 'II', 3: 'III', 4:'IV', 5: 'V', 6:'VI', 7:'VII',
                8:'VIII'}
>>> for i in ['three', 'four', 'five', 'six']:
...     print(numerals[i], end=' ')
...
III IV V VI
>>> for i in range(8,0,-1):
...     print(numerals[i], end=' ')
VIII VII VI V IV III II I
```

Обратите внимание: независимо от порядка, в котором сохраняются ключи, словарь можно проиндексировать в любом порядке. Следует также отметить, что хотя ключи словаря обязательно должны быть неповторяющимися, на значения это ограничение не распространяется.

4.2.2 Методы словаря

Метод *get()*

Обращение к словарю по индексу с несуществующим ключом приводит к ошибке:

```
>>> mass['Pluto']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Pluto'
```

Но можно использовать удобный метод `get()` для извлечения значения, задавая ключ, если он существует, или некоторое значение по умолчанию, если ключ не существует. Если значение по умолчанию не задано, то возвращается специальное значение `None`. Например:

```
>>> print(mass.get('Pluto'))
None
>>> mass.get('Pluto', -1)
-1
```

Методы *keys*, *values* и *items*

Три метода `keys`, `values` и `items` возвращают соответственно ключи, значения и пары ключ-значение (в виде кортежей) словаря. В предыдущих версиях Python эти объекты возвращались в списке, но для большинства целей это был напрасный расход памяти: например, вызов метода `keys` требовал, чтобы все ключи словаря копировались как список, по которому в большинстве случаев просто выполнялся итеративный проход. Поэтому обычно нет необходимости в сохранении полной новой копии ключей словаря. Версия Python 3 решает эту проблему, возвращая итерируемый объект, который обеспечивает последовательный доступ к каждому ключу словаря без копирования их в список. Это работает быстрее и позволяет экономить память (что важно для весьма больших словарей). Например:

```
>>> planets = mass.keys()
>>> print(planets)
dict_keys(['Mercury', 'Venus', 'Earth'])
>>> for planet in planets:
...     print(planet , mass[planet])
...
Mercury 3.301e+23
Venus 4.867e+24
Earth 5.972e+24
```

По объекту `dict_keys` можно выполнять итеративный проход любое число раз, но это не список, поэтому его невозможно индексировать и нельзя выполнять операции присваивания:

```
>>> planets = mass.keys()
>>> planets[0]

Traceback (most recent call last):
  File "<stdin >", line 1, in <module >
TypeError: 'dict_keys' object is not subscriptable
```

Если действительно необходим список ключей словаря, то нужно просто передать объект `dict_keys` в конструктор списка `list` (который принимает любой тип последовательности и создает из нее список):

```
>>> planet_list = list(mass.keys())
>>> planet_list
['Mercury', 'Venus', 'Earth']
>>> planet_list[0]
'Mercury'
>>> planet_list[1] = 'Jupiter'                                    ❶
>>> planet_list
['Mercury', 'Jupiter', 'Earth']
```

❶ Это последнее присваивание изменяет только список `planet_list`, но ключи исходного словаря не изменяются.

Существуют аналогичные методы для извлечения значений и элементов (пар ключ-значение) словаря: возвращаются объекты `dict_values` и `dict_items`. Например:

```
>>> mass.items()
dict_items([('Mercury', 3.301e+23), ('Venus', 4.867e+24), ('Earth', 5.972e+24)])
>>> mass.values()
dict_values ([3.301e+23, 4.867e+24, 5.972e+24])
>>> for planet_data in mass.items():
...     print(planet_data)
...
('Mercury', 3.301e+23)
('Venus', 4.867e+24)
('Earth', 5.972e+24)
```

Пример П4.7. Словарь Python можно использовать как простую базу данных. В листинге 4.1 показан код, сохраняющий информацию о некоторых астрономических объектах в словаре, состоящем из кортежей, где ключами являются имена объектов. При обработке создается список значений плотности планет.

Листинг 4.1. Обработка астрономических данных

```
# eg4-astrodict.py
import math

# Масса (в кг) и радиус (в км) для некоторых астрономических тел.
body = {'Sun': (1.988e30, 6.955e5),
        'Mercury': (3.301e23, 2440.),
        'Venus': (4.867e+24, 6052.),
        'Earth': (5.972e24, 6371.),
        'Mars': (6.417e23, 3390.),
        'Jupiter': (1.899e27, 69911.),
        'Saturn': (5.685e26, 58232.),
        'Uranus': (8.682e25, 25362.),
        'Neptune': (1.024e26, 24622.)
       }
```

```
planets = list(body.keys())
# Солнце - это не планета.
planets.remove('Sun')

def calc_density(m, r):
    """ Returns the density of a sphere with mass m and radius r. """
    # """ Возвращает плотность шара с массой m и радиусом r. """
    return m / (4/3 * math.pi * r**3)

rho = {}
for planet in planets:
    m, r = body[planet]
    # Вычисление плотности планеты в г/см3.
    rho[planet] = calc_density(m*1000, r*1.e5)

for planet , density in sorted(rho.items()):                    ❶
    print('The density of {0} is {1:3.2f} g/cm3'.format(planet , density))
```

❶ Метод `sorted(rho.items())` возвращает список пар ключ-значение словаря `rho`, отсортированный по ключу. Ключи – это строки, поэтому в рассматриваемом здесь примере сортировка создает список ключей в алфавитном порядке.

При выполнении программа выводит следующий результат:

```
The density of Earth is 5.51 g/cm3
The density of Jupiter is 1.33 g/cm3
The density of Mars is 3.93 g/cm3
The density of Mercury is 5.42 g/cm3
The density of Neptune is 1.64 g/cm3
The density of Saturn is 0.69 g/cm3
The density of Uranus is 1.27 g/cm3
The density of Venus is 5.24 g/cm3
```

◊ Именованные аргументы

В разделе 2.7 рассматривался синтаксис передачи аргументов в функции. При этом предполагалось, что функция всегда должна знать, какие аргументы могут передаваться, и эти аргументы указывались в определении функции. Например:

```
def func(a, b, c):
```

Python предоставляет несколько удобных функциональных возможностей для обработки случаев, когда не обязательно знать, какие аргументы будет принимать функция. При включении `*args` (после всех «формально определенных» аргументов) любой дополнительный позиционный аргумент помещается в кортеж `args`, как показано в следующем коде:

```
>>> def func(a, b, *args):
...     print(args)
...
>>> func(1, 2, 3, 4, 'msg')
(3, 4, 'msg')
```

Таким образом, в теле функции `func` в дополнение к формальным аргументам `a=1` и `b=2` доступны также аргументы `3`, `4` и `'msg'` как элементы кортежа `args`. Этот кортеж может иметь произвольную длину. В Python встроенная функция `print` работает аналогичным образом: она принимает произвольное количество аргументов для вывода их как строки, а за этим кортежем аргументов следуют некоторые дополнительные именованные аргументы:

```
def print(*args , sep=' ', end='\n', file=None):
```

Кроме того, можно собрать произвольные именованные аргументы (см. раздел 2.7.2), передаваемые в функцию, в словаре, используя для этого синтаксис `**kwargs` в определении функции. Python собирает все именованные аргументы, не указанные в определении функции, и упаковывает их в словарь `kwargs`. Например:

```
>>> def func(a, b, **kwargs):
...     for k in kwargs:
...         print(k, '=', kwargs[k])
...
>>> func(1, b=2, c=3, d=4, s='msg')
d = 4
s = msg
c = 3
```

Можно также использовать `*args` и `**kwargs` при вызове функции, что может оказаться удобным, например, для функций, принимающих большое количество аргументов:

```
>>> def func(a, b, c, x, y, z):
...     print(a, b, c)
...     print(x, y, z)
...
>>> args = [1, 2, 3]
>>> kwargs = {'x': 4, 'y': 5, 'z': 'msg'}
>>> func(*args , **kwargs)
1 2 3
4 5 msg
```

◊ Объект *defaultdict*

В обычных словарях Python попытка извлечения значения с использованием несуществующего ключа генерирует исключение `KeyError`. Существует полезный контейнер `defaultdict`, который создает подкласс встроенного объекта `dict`, позволяющий определить `default_factory`, функцию, возвращающую значение по умолчанию, присваиваемое отсутствующему ключу.

Пример П4.8. Для анализа длины слов в первой строке Геттисбергской речи (Авраама Линкольна) с помощью обычного словаря требуется код для перехвата исключения `KeyError` и установки значения по умолчанию:

```
text = 'Four score and seven years ago our fathers brought forth on this
continent, a new nation, conceived in Liberty, and dedicated to the proposition
that all men are created equal'

text = text.replace(',', '').lower()        # Удаление знаков препинания.

word_lengths = {}
for word in text.split():
    try:
        word_lengths[len(word)] += 1
    except KeyError:
        word_lengths[len(word)] = 1
print(word_lengths)
```

Использование `defaultdict` в этом случае позволяет написать более компактный и удобочитаемый код:

```
from collections import defaultdict                                    ❶
word_lengths = defaultdict(int)                                        ❷
for word in text.split():
    word_lengths[len(word)] += 1
print(word_lengths)
```

❶ Обратите внимание: `defaultdict` не является встроенным объектом – он обязательно должен импортироваться из модуля `collections`.

❷ Здесь для функции `default_factory` определяется возврат значения типа `int`: если ключ отсутствует, то он будет вставлен в словарь и инициализирован с помощью вызова `int()`, который возвращает `0`.

При выполнении приведенного выше кода выводится следующий результат:

```
defaultdict(<class 'int'>, {4: 3, 5: 5, 3: 9, 7: 4, 2: 3, 9: 3, 1: 1, 6: 1, 11: 1})
```

4.2.3 Множества

Множество `set` – это неупорядоченный набор неповторяющихся элементов. Как и ключи словарей, элементы множества обязательно должны быть хешируемыми объектами. Множество удобно использовать для удаления повторяющихся элементов из последовательности и для определения объединения (union), пересечения и разности между двумя наборами элементов. Поскольку элементы не упорядочены, объекты типа `set` нельзя индексировать и выполнять в них операцию вырезания (slice), но можно производить итеративный проход по множеству, проверять наличие элемента. Множества поддерживают встроенную функцию `len`. Объект `set` создается с помощью перечисления его элементов в фигурных скобках ({…}) или при передаче итерируемого объекта в конструктор `set()`:

```
>>> s = set([1, 1, 4, 3, 2, 2, 3, 4, 1, 3, 'surprise!'])
>>> s
{1, 2, 'surprise!', 3, 4}
>>> len(s)                  # Мощность множества.
5
>>> 2 in s, 6 not in s      # Проверка на наличие и на отсутствие элемента в множестве.
(True , True)
>>> for item in s:
...     print(item)
...
1
2
surprise!
3
4
```

Метод множества `add` используется для добавления элементов. Для удаления элементов существует несколько методов: `remove` удаляет заданный элемент, но генерирует исключение `KeyError`, если элемент отсутствует в множестве. Метод `discard()` делает то же самое, но не генерирует исключение. Оба метода принимают (как единственный аргумент) удаляемый элемент. Метод `pop` (без аргументов) удаляет (и возвращает) произвольный элемент из множества, а метод `clear` удаляет все элементы:

```
>>> s = {2,-2,0}
>>> s.add(1)
>>> s.add(-1)
>>> s.add(1.0)                                                    ❶
>>> s
{0, 1, 2, -1, -2}
>>> s.remove(1)
>>> s
{0, 2, -1, -2}
>>> s.discard(3)          # OK - ничего не делает.
>>> s
{0, 2, -1, -2}
>>> s.pop()
0                         # (например)
>>> s
{2, -1, -2}
>>> s.clear()
set()                     # Пустое множество.
```

❶ Эта инструкция не добавляет новый элемент в множество, даже если существующий элемент `1` является целым числом, а добавляемый элемент имеет тип `float`. Проверка `1 == 1.0` дает результат `True`, поэтому `1.0` считается уже существующим в этом множестве.

Объекты `set` имеют обширный набор методов, соответствующих свойствам математических множеств. Наиболее полезные методы перечислены в табл. 4.2, и при их описании используются следующие термины теории множеств:

- мощность (cardinality) множества |A| – это количество элементов, содержащихся в множестве;
- два множества равны (equal), если они содержат одинаковые элементы;
- множество A является подмножеством (subset) множества B ($A \subseteq B$), если все элементы A также являются элементами B, тогда B называется надмножеством, или супермножеством (superset) множества A;
- множество A является истинным, или строгим, подмножеством (proper subset) множества B ($A \subset B$), если A является подмножеством B, но не равно ему, тогда B называется истинным, или строгим, надмножеством (proper superset) множества A;
- объединение (union) двух множеств ($A \cup B$) – это множество всех элементов из множеств A и B;
- пересечение (intersection) двух множеств ($A \cap B$) – это множество всех элементов, которые содержатся и в A, и в B;
- разность (difference) множества A и множества B ($A \setminus B$) – это множество элементов A, которых нет в B;
- симметрическая разность (symmetric difference) двух множеств $A \Delta B$ – это множество элементов, содержащихся в одном из множеств A или B, но не в обоих множествах;
- два множества называются непересекающимися (disjoint), если они не имеют общих элементов.

Таблица 4.2. Методы множества set

Метод	Описание
`isdisjoint(other)`	Множество *`set`* не пересекается с множеством *`other`*?
`issubset(other)`, `set <= other`	Множество *`set`* является подмножеством *`other`*?
`set < other`	Множество *`set`* является истинным (строгим) подмножеством *`other`*?
`issuperset(other)`, `set >= other`	Множество *`set`* является надмножеством *`other`*?
`set > other`	Множество *`set`* является истинным (строгим) надмножеством *`other`*?
`union(other)`, `set | other | …`	Объединение множества *`set`* и множества *`other`* (возможно, нескольких множеств)
`intersection(other)`, `set & other & …`	Пересечение множества *`set`* и множества *`other`* (возможно, нескольких множеств)
`difference(other)`, `set - other - …`	Разность множества *`set`* и множества *`other`* (возможно, нескольких множеств)
`symmetric_difference(other)`, `set ^ other ^ …`	Симметрическая разность множества *`set`* и множества *`other`* (возможно, нескольких множеств)

Существуют две формы для большинства выражений с использованием множеств `set`: синтаксис операторов требует, чтобы все аргументы (операнды) были объектами типа `set`, тогда как явные вызовы методов выполняют преобразование любого итерируемого аргумента в множество `set`.

```
>>> A = set((1, 2, 3))
>>> B = set((1, 2, 3, 4))
>>> A <= B
True
>>> A.issubset((1, 2, 3, 4))       # OK: (1, 2, 3, 4) преобразуется в множество.
True
```

Еще несколько примеров:

```
>>> C, D = set((3, 4, 5, 6)), set((7, 8, 9))
>>> B | C                    # Объединение.
{1, 2, 3, 4, 5, 6}
>>> A | C | D                # Объединение трех множеств.
{1, 2, 3, 4, 5, 6, 7, 8, 9}
>>> A & C                    # Пересечение.
{3}
>>> C & D
set()                       # Пустое множество.
>>> C.isdisjoint(D)
True
>>> B - C                    # Разность.
{1, 2}
>>> B ^ C                    # Симметрическая разность.
{1, 2, 5, 6}
```

◊ Объекты *frozenset*

Множества `set` – изменяемые объекты (можно добавлять и удалять элементы в множестве), поэтому их невозможно хешировать, следовательно, нельзя использовать как ключи словарей или как члены других множеств.

```
>>> a = set((1, 2, 3))
>>> b = set(('q', (1, 2), a))

Traceback (most recent call last):
  File "<stdin >", line 1, in <module >
TypeError: unhashable type: 'set'
>>>
```

(По той же причине списки `list` не могут быть ключами словарей и элементами множеств.) Но существует объект `frozenset`, являющийся типом неизменяемого (и хешируемого) множества[40]. Объекты `frozenset` – это неизменяемые неупорядоченные наборы неповторяющихся объектов, и они могут использоваться как ключи словарей и элементы других множеств.

```
>>> a = frozenset((1, 2, 3))
>>> b = set(('q', (1, 2), a))     # OK: frozenset - хешируемый объект.
>>> b.add(4)                      # OK: b - обычное множество.
>>> a.add(4)                      # Недопустимо: frozenset - неизменяемый объект.

Traceback (most recent call last):
  File "<stdin >", line 1, in <module >
AttributeError: 'frozenset' object has no attribute 'add'
```

[40] В том смысле, что `frozenset` можно сравнивать с множествами `set` так же, как кортежи можно сравнивать со списками.

Пример П4.9. Простое число Мерсенна (Mercenne prime) M_i – это простое число вида $M_i = 2^i - 1$. Множество чисел Мерсенна, меньших некоторого числа *n*, можно представить как пересечение множества всех простых чисел, меньших *n*, P_n с множеством A_n целых чисел, соответствующих условию $2^i - 1 < n$.

Программа в листинге 4.2 возвращает список простых чисел Мерсенна, меньших 1 000 000.

Листинг 4.2. Простые числа Мерсенна

```
import math

def primes(n):
    """ Return a list of the prime numbers <= n. """
    # """ Возвращает список простых чисел <= n. """

    sieve = [True] * (n // 2)
    for i in range(3, int(math.sqrt(n)) + 1, 2):
        if sieve[i//2]:
            sieve[i*i//2::i] = [False] * ((n - i*i - 1) // (2*i) + 1)
    return [2] + [2*i+1 for i in range(1, n // 2) if sieve[i]]

n = 1000000

P = set(primes(n))                                                    ❶

# Список целых чисел вида 2^i - 1 <= n.
A = []
for i in range(2, int(math.log(n+1, 2)) + 1):                         ❷
    A.append(2**i - 1)

# Множество чисел Мерсенна как пересечение множеств P и A.
M = P.intersection(A)

# Вывод полученного множества M как отсортированного списка.
print(sorted(list(M)))                                                ❸
```

Простые числа генерируются в виде списка функцией primes, реализованной как оптимизированная версия алгоритма решета Эратосфена (см. пример П2.5.8), и полученный список преобразуется в множество `P` (❶). Можно выполнить операцию пересечения этого множества с любым итерируемым объектом, используя метод `intersection`, поэтому здесь не требуется явное преобразование второго списка целых чисел A (❷) в множество.

❸ Наконец, создается множество простых чисел Мерсенна `M` – неупорядоченный набор элементов, поэтому при выводе это множество преобразуется в отсортированный список (`list`).

При *n* = 1 000 000 выводится следующий результат:

```
[3, 7, 31, 127, 8191, 131071, 524287]
```

4.2.4 Упражнения

Вопросы

B4.2.1. Написать однострочную программу на Python, определяющую, является ли строка панграммой (т. е. строкой, содержащей каждую букву алфавита, как минимум, один раз).

B4.2.2. Написать функцию, использующую объекты типа `set`, для удаления повторяющихся элементов из упорядоченного списка (`list`). Например:

```
>>> remove_dupes([1, 1, 2, 3, 4, 4, 4, 5, 7, 8, 8, 9])
[1, 2, 3, 4, 5, 7, 8, 9]
```

B4.2.3. Определить и объяснить результаты выполнения следующих инструкций:

```
>>> set('hellohellohello')
>>> set(['hellohellohello'])
>>> set(('hellohellohello'))
>>> set(('hellohellohello',))
>>> set(('hello', 'hello', 'hello'))
>>> set(('hello', ('hello', 'hello')))
>>> set(('hello', ['hello', 'hello']))
```

B4.2.4. Если известно, что объекты `frozenset` неизменяемы, то почему оказался возможным показанный ниже результат?

```
>>> a = frozenset((1, 2, 3))
>>> a |= {2, 3, 4, 5}
>>> print(a)
frozenset([1, 2, 3, 4, 5])
```

B4.2.5. Изменить пример П4.8 с использованием объекта `defaultdict` для получения списка слов, ключами которых являются их длины, из текста первой строки Геттисбергской речи.

Задачи

34.2.1. Величины и допуски старых сопротивлений (резисторов) определяются по четырем цветовым полоскам: первые две обозначают первые две значимые цифры электрического сопротивления в омах, третья полоска обозначает десятичный множитель (число нулей), а четвертая – допустимое отклонение от номинала. Цвета и их значения для каждой полоски перечислены в табл. 4.3.

Таблица 4.3. Цветовые коды сопротивлений (резисторов)

Цвет	Сокращенное обозначение	Значимые цифры	Множитель	Допустимое отклонение
Черный	`bk`	0	1	-
Коричневый	`br`	1	10	±1 %
Красный	`rd`	2	10^2	±2 %

Окончание табл. 4.3

Цвет	Сокращенное обозначение	Значимые цифры	Множитель	Допустимое отклонение
Оранжевый	`or`	3	10^3	–
Желтый	`yl`	4	10^4	±5 %
Зеленый	`gr`	5	10^5	±0.5 %
Синий	`bl`	6	10^6	±0.25 %
Фиолетовый	`vi`	7	10^7	±0.1 %
Серый	`gy`	8	10^8	±0.05 %
Белый	`wh`	9	10^9	–
Золотой	`au`	–	–	±5 %
Серебряный	`ag`	–	–	±10 %
Нет	`--`	–	–	±20 %

Например, для резистора с полосками следующих цветов: фиолетовая, желтая, красная, зеленая – значение сопротивления равно $74 \times 10^2 = 7400\ \Omega$, а допустимое отклонение от номинала составляет ±0.5 %.

Написать программу, в которой определяется функция преобразования списка из четырех сокращенных обозначений цветов (полосок) в значение сопротивления и допустимое отклонение. Например:

```
In [x]: print(get_resistor_value(['vi', 'yl', 'rd', 'gr']))
Out[x]: (7400, 0.5)
```

34.2.2. Для романа «Моби Дик» (автор Герман Мелвилл) истек срок защиты авторского права, и его можно загрузить в виде текстового файла с веб-сайта Project Gutenberg: www.gutenberg.org/2/7/0/2701/. Написать программу вывода 100 слов, наиболее часто используемых в этой книге, с сохранением счетчика каждого встреченного слова в словаре.

Совет: рекомендуется воспользоваться строковыми методами для удаления всех знаков пунктуации. Достаточно заменить на пустую строку все обнаруженные экземпляры следующих символов: `!?»:;,()'.*[]`. После создания словаря со словами в качестве ключей и соответствующими счетчиками слов в качестве значений нужно сформировать список из кортежей `(счетчик, слово)` и отсортировать его.

Дополнительное задание: сравнить частоты 2000 слов, которые чаще всего встречаются в романе «Моби Дик», с прогнозом распределения частоты слов по закону Ципфа (Zipf's law):

$$\log f(w) = \log C - a \log r(w),$$

где $f(w)$ – количество вхождений слова w, $r(w)$ – соответствующий ранг (1 = наиболее часто встречающееся слово, 2 = второе слово по частоте вхождений и т. д.), а C и a – некоторые константы. В стандартной формулировке закона Ципфа $C = \log f(w_1)$ и $a = 1$, где w_1 – наиболее часто встречающееся слово, так что $r(w_1) = 1$.

34.2.3. Обратная польская запись (reverse Polish notation – RPN), или постфиксная запись (postfix notation), – это форма записи математических выражений, в которой каждый оператор располагается после своих операндов (в противоположность более привычной инфиксной (infix) форме записи, в которой оператор располагается между обрабатываемыми операндами). Например, инфиксное выражение `5 + 6` в обратной польской (постфиксной) нотации записывается как `5 6 +`. Преимущество такого способа записи состоит в том, что в этом случае не нужны скобки: для вычисления выражения `(3 + 7) / 2` его можно записать как `3 7 + 2 /`. Выражение в постфиксной нотации вычисляется слева направо с помещением промежуточных результатов в стек – список значений, работающий по принципу «последним вошел, первым вышел» (LIFO) – и извлечением (выталкиванием) из стека значений, когда они необходимы для выполнения очередного оператора (см. также пример П2.16). Таким образом, при обработке выражения `3 7 + 2 /` значение `3`, затем значение `7` помещаются в стек (при этом `7` оказывается на вершине стека). Далее следует символ `+`, поэтому значения извлекаются из стека, суммируются, и результат `10` снова помещается в стек (который теперь пуст). Потом `2` помещается в стек. Завершающий символ `/` извлекает два значения `10` и `2` из стека и выполняет деление с результатом `5`.

Написать программу вычисления выражения в форме обратной польской (постфиксной) записи, состоящего из символов, разделенных пробелами (операторов `+ - * / **` и чисел).

Совет: для парсинга необходимо преобразовать выражение в список строковых элементов и выполнить итеративный проход по этому списку с преобразованием и помещением чисел в стек (который можно реализовать как список (`list`) с добавлением элементов). Определить функции для выполнения операций над значениями, извлекаемыми из стека с помощью метода `pop`. Следует отметить, что Python не предоставляет синтаксическую конструкцию `switch…case`, но объекты функций операций могут быть значениями в словаре, ключами которого являются символы операторов.

34.2.4. Использовать словарь кодов азбуки Морзе из файла *morse.py*, размещенного здесь: https://scipython.com/ex/bdb, чтобы написать программу преобразования текстового сообщения в код Морзе и обратно. Использовать пробелы для разделения отдельных «букв» кода Морзе, а символ слеш (/) – для разделения слов. Например, `'PYTHON 3'` преобразовывается в `'.--. -.-- - .... - --. / ...--'`.

34.2.5. Файл *shark-species.txt*, размещенный по адресу https://scipython.com/ex/bdc, содержит список существующих видов акул, организованный в иерархической форме: семейство, род и вид (вид представлен в форме «биологическое название: общее название»). Выполнить считывание содержимого этого файла в структуру данных, состоящую из вложенных словарей, доступ к которым можно осуществлять следующим образом:

```
>>> sharks['Lamniformes']['Lamnidae']['Carcharodon']['C. carcharias']
Great white shark
```

4.3 Идиоматические выражения Python: синтаксический сахар

Многие языки программирования предоставляют синтаксис, позволяющий проще выполнять часто встречающиеся небольшие действия и писать более ясный код. Такой синтаксический сахар (syntactic sugar) состоит из конструкций, которые можно убрать из языка без ущерба его функциональности. Мы уже рассматривали один пример так называемого комбинированного присваивания: `a += 1` равнозначно `a = a + 1`. Другой пример – отрицательный индекс в последовательности: `b[-1]` равнозначно `b[len(b)-1]` и более удобно.

4.3.1 Рациональные операции сравнения и присваивания

Если один объект необходимо присвоить нескольким переменным, то можно воспользоваться рациональной операцией присваивания:

```
x = y = z = -1
```

Обратите внимание: если таким способом присваиваются изменяемые объекты, то все имена переменных указывают на один и тот же объект, а не на различные его копии (см. раздел 2.4.1).

Как было показано в разделе 2.4.2, несколько присваиваний различных объектов можно выполнить в одной строке с помощью распаковки кортежа:

```
a, b, c = x + 1, 'hello', -4.5
```

Кортеж в правой части этого выражения (в этом случае скобки не обязательны) распаковывается для присваивания его элементов именам переменных в левой части. Эта строка равнозначна следующим трем строкам:

```
a = x + 1
b = 'hello'
c = -4.5
```

В подобных выражениях сначала вычисляется правая часть, затем выполняется присваивание в левой части. Как уже было показано ранее, такой подход предоставляет очень удобный способ обмена значениями двух переменных без использования временной переменной:

```
a, b = b, a
```

Операции сравнения также можно объединять в цепочку вполне естественным способом:

```
if a == b == 3:
    print('a and b both equal 3')
if -1 < x < 1:
    print('x is between -1 and 1')
```

Python поддерживает операцию условного присваивания: имени переменной может быть присвоено одно или другое значение в зависимости от резуль-

тата вычисления выражения `if…else` непосредственно в строке присваивания. Например:

```
y = math.sin(x)/x if x else 1
```

Короткие примеры, подобные приведенному выше, в которых показано, как можно избежать потенциального деления на ноль (напомню, что `0` вычисляется как `False`), весьма просты. Поэтому не рекомендуется применять эту идиоматическую конструкцию в более сложных случаях, а лучше заменить ее более явной конструкцией, например:

```
try:
    y = math.sin(x)/x
except ZeroDivisionError:
    y = 1
```

4.3.2 Генерация списка

Генератор списков в Python – это конструкция для создания списка на основе другого итерируемого объекта в одной строке кода. Например, если задан список чисел `xlist`, то список квадратов этих чисел можно сгенерировать следующим образом:

```
>>> xlist = [1, 2, 3, 4, 5, 6]
>>> x2list = [x**2 for x in xlist]
>>> x2list
[1, 4, 9, 16, 25, 36]
```

Это более быстрый и синтаксически более удобный и понятный способ создания списка, по сравнению с созданием того же списка в блоке кода цикла `for`:

```
>>> x2list = []
>>> for x in xlist:
...     x2list.append(x**2)
```

Инструкция генерации списка также может содержать условные выражения:

```
>>> x2list = [x**2 for x in xlist if x % 2]
>>> x2list
[1, 9, 25]
```

Здесь `x` передается в выражение `x**2` для включения в формируемый список `x2list`, только если выражение `x % 2` дает результат `True` (т. е. если `x` – нечетное число). Это пример фильтра (одиночного условного выражения `if`). Если требуется более сложное отображение значений из исходной последовательности в значения создаваемого списка, то необходимо поместить выражение `if…else` перед циклом `for`:

```
>>> [x**2 if x % 2 else x**3 for x in xlist]
[1, 8, 9, 64, 25, 216]
```

Этот генератор возводит в квадрат нечетные целые числа или в куб четные целые числа из списка `xlist`.

Разумеется, последовательность, используемая для генерации списка, не должна содержать другой список. Например, строки, кортежи и объекты `range` являются итерируемыми объектами, поэтому могут использоваться для генерации списков:

```
>>> [x**3 for x in range(1, 10)]
[1, 8, 27, 64, 125, 216, 343, 512, 729]
>>> [w.upper() for w in 'abc xyz']
['A', 'B', 'C', ' ', 'X', 'Y', 'Z']
```

Наконец, генераторы списков могут быть вложенными. Например, в следующем фрагменте кода демонстрируется преобразование списка, состоящего из вложенных списков, в простой «одномерный» список:

```
>>> vlist = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> [c for v in vlist for c in v]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Здесь первый цикл `for` обрабатывает внутренние списки по очереди как `v`, и по каждому внутреннему списку `v` выполняется итеративный проход с переменной `c` для добавления элементов в создаваемый список.

Пример П4.10. Рассмотрим матрицу 3×3, представленную как список списков:

```
M = [[1, 2, 3],
     [4, 5, 6],
     [7, 8, 9]]
```

Без использования генератора списков операцию транспонирования этой матрицы можно было бы реализовать с помощью циклов с проходом по строкам и столбцам:

```
MT = [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
for ir in range(3):
    for ic in range(3):
        MT[ic][ir] = M[ir][ic]
```

Применяя генератор списков, операцию транспонирования матрицы можно реализовать следующим образом:

```
MT = []
for i in range(3):
    MT.append([row[i] for row in M])
```

Здесь строки транспонированной матрицы формируются из столбцов (проиндексированных как `i=0, 1, 2`) из каждой строки, взятой из исходной матрицы `M`. Внешний цикл можно представить сам по себе как генератор собственного списка:

```
MT = [[row[i] for row in M] for i in range(3)]
```

Но следует отметить, что библиотека NumPy предоставляет гораздо более простые и удобные средства для работы с матрицами.

4.3.3 Лямбда-функции

Лямбда-функция (`lambda`) в Python – это тип простой анонимной функции. Выполняемое тело лямбда-функции обязательно должно быть выражением (expression), а не инструкцией (statement), т. е. тело лямбда-функции не может содержать, например, блоки циклов, проверки условий или инструкции `print`. Лямбда-функции обеспечивают ограниченную поддержку парадигмы программирования, известной как «функциональное программирование» (functional programming)[41]. Простейший пример применения лямбда-функции немного отличается от обычного способа определения функции `def`, показан ниже:

```
>>> f = lambda x: x**2 - 3*x + 2
>>> print(f(4.))
6.0
```

Аргумент передается в `x`, а результат, обусловленный определением лямбда-функции после двоеточия, возвращается вызывающей стороне. Для передачи нескольких переменных в лямбда-функцию используется кортеж (без скобок):

```
>>> f = lambda x,y: x**2 + 2*x*y + y**2
>>> f(2., 3.)
25.0
```

В этих примерах не наблюдается какая-либо особая польза от лямбда-функций, да и определенные здесь функции не вполне анонимны (поскольку они были связаны с именем переменной `f`). Более полезное применение – создание списка функций, как показано в примере П4.11.

Пример П4.11. Функции – это объекты (как и все в Python), поэтому их можно сохранять в списках. Без использования лямбда-функций пришлось бы определять именованные функции (с помощью ключевого слова `def`) перед созданием списка:

```
def const(x):
    return 1.
def lin(x):
    return x
def square(x):
    return x**2
def cube(x):
    return x**3
flist = [const , lin, square , cube]
```

После этого `flist[3](5)` возвращает `125`, так как `flist[3]` – это функция `cube`, и она вызывается с аргументом `5`.

Преимущество использования лямбда-выражений как анонимных функций заключается в том, что для этих функций не требуются имена, поэтому они могут определяться как подстановочные или встраиваемые (inline) элементы при формировании списка:

---

[41] Функциональное программирование – это стиль программирования, в котором вычисления реализованы через выполнение математических функций с минимальными ссылками на переменные, определяющие состояние (state) программы.

```
>>> flist = [lambda x: 1,
...          lambda x: x,
...          lambda x: x**2,
...          lambda x: x**3]
>>> flist[3](5) # flist[3] is x**3
125
>>> flist[2](4) # flist[2] is x**2
16
```

Пример П4.12. Встроенный метод `sorted` и метод списка `sort` могут упорядочивать списки на основе значения, возвращаемого функцией, вызываемой предварительно для каждого элемента для выполнения сравнений. Эта функция передается как аргумент `key`. Например, при сортировке списка строк по умолчанию учитывается регистр символов (букв):

```
>>> sorted('Nobody expects the Spanish Inquisition'.split())
['Inquisition', 'Nobody', 'Spanish', 'expects', 'the']
```

Но можно не учитывать регистр букв при сортировке, передавая каждое слово в метод `str.lower`:

```
>>> sorted('Nobody expects the Spanish Inquisition'.split(), key=str.lower)
['expects', 'Inquisition', 'Nobody', 'Spanish', 'the']
```

(Разумеется, точно так же будет работать и вариант `key=str.upper`.) Следует отметить, что сами элементы списка не изменяются: они упорядочиваются на основе собственной версии написания в нижнем регистре. Здесь не используются скобки, как в инструкции `str.lower()`, потому что в аргументе `key` передается сама функция (как объект), а не выполняется ее непосредственный вызов.

Для этой цели обычно применяются лямбда-выражения, чтобы предоставить простые анонимные функции. Например, для сортировки списка атомов (символ элемента, атомное число) как кортежей в порядке атомных чисел (второго элемента каждого кортежа):

```
>>> halogens = [('At', 85), ('Br', 35), ('Cl', 17), ('F', 9), ('I', 53)]
>>> sorted(halogens , key=lambda e: e[1])
[('F', 9), ('Cl', 17), ('Br', 35), ('I', 53), ('At', 85)]
```

Здесь алгоритм сортировки вызывает функцию, определяемую по аргументу `key`, для каждого элемента-кортежа, чтобы решить, какое место должен занять в итоговом списке этот элемент. Анонимная функция просто возвращает второй элемент каждого кортежа, обеспечивая сортировку по атомному числу.

4.3.4 Ключевое слово `with`

Ключевое слово `with` создает блок кода, который выполняется в определенном контексте. Контекст определяется диспетчером контекста (context manager), который предоставляет пару методов, описывающих, как войти и выйти из контекста. Контексты, определяемые пользователем, в основном применяются

в профессиональном (продвинутом) коде и могут быть весьма сложными, но в обобщенном простом примере применения встроенного диспетчера контекста рассматривается файловый ввод/вывод. Здесь вход в контекст выполняется при открытии файла. В блоке контекста файл считывается или записывается, после чего файл закрывается с выходом из контекста. Объект `file` является диспетчером контекста, который возвращается методом `open()`. Диспетчер контекста определяет метод выхода, который просто закрывает файл (если до этого он был успешно открыт), поэтому не требуется явное выполнение этой операции. Для открытия файла в контексте используется следующий код:

```
with open('filename') as f:
    # Обработка файла каким-либо способом, например:
    lines = f.readlines()
```

Обоснование такого метода работы с файлом: вы можете быть абсолютно уверены в том, что файл будет закрыт после выхода из блока `with`, даже если внутри блока что-то пошло не так, как нужно, – диспетчер контекста выполнит код, который вам пришлось бы писать для перехвата ошибок времени выполнения такого типа.

4.3.5 Генераторы

Генераторы – мощный инструмент языка Python, они позволяют объявить функцию, поведение которой похоже на итерируемый объект. Такую функцию можно использовать в цикле `for`, так как она будет последовательно генерировать по одному значению по запросу. Часто подобный метод более эффективен, чем вычисление и сохранение всех значений, для которых должны выполняться итерации (особенно если количество таких значений весьма велико). Функция-генератор выглядит почти как обычная функция Python, но вместо выхода с возвращаемым (`return`) значением она содержит ключевое слово `yield`, которое возвращает значение каждый раз, когда оно требуется на очередной итерации.

Самый простой пример поможет лучше понять работу генератора. Определим генератор `count` для реализации счетчика до значения переменной `n`:

```
>>> def count(n):
...     i = 0
...     while i < n:
...         i += 1
...         yield i
...
>>> for j in count(5):
...     print(j)
...
1
2
3
4
5
```

Обратите внимание: мы не можем просто вызвать этот генератор как обычную функцию:

```
>>> count(5)
<generator object count at 0x102d8e6e0 >
```

Предполагается, что генератор `count` должен вызываться как часть цикла (здесь это цикл `for`), и на каждой итерации генератор выдает (`yield`) результат и сохраняет свое состояние (текущее значение `i`) до следующего вызова из цикла.

В действительности мы уже использовали генераторы, поскольку знакомая нам встроенная функция `range` в Python 3 представляет собой объект типа генератор.

Существует еще и синтаксис генерации генераторов (generator comprehension), похожий на синтаксис генерации списков (вместо квадратных скобок используются круглые):

```
>>> squares = (x**2 for x in range(5))
>>> for square in squares:
...     print(square)
...
0
1
4
9
16
```

Но после того, как определенный таким способом генератор генератора будет «исчерпан», мы не сможем еще раз выполнить итеративный проход по нему без переопределения. При попытке:

```
>>> for square in squares:
...     print(square)
...
>>>
```

мы не получим никакого результата, так как ранее уже был достигнут конец генератора `squares`.

Для получения списка или кортежа значений генератора нужно просто передавать их в список или в кортеж, как показано в примере П4.13.

Пример П4.13. Эта функция определяет генератор треугольных чисел $T_n = \sum_{k=1}^{n} k = 1 + 2 + 3 + \ldots + n$ при $n = 0, 1, 2, \ldots$. То есть $T_n = 0, 1, 3, 6, 10, \ldots$.

```
>>> def triangular_numbers(n):
...     i, t = 1, 0
...     while i <= n:
...         yield t
...         t += i
...         i += 1
...
>>> list(triangular_numbers (15))
[0, 1, 3, 6, 10, 15, 21, 28, 36, 45, 55, 66, 78, 91, 105]
```

Обратите внимание: инструкции после строки с ключевым словом `yield` выполняются каждый раз при возобновлении выполнения генератора `triangular_numbers`. Вызов `triangular_numbers(15)` возвращает итератор, который передает эти числа в метод `list` для формирования списка передаваемых значений.

4.3.6 ◊ Встроенная функция map

Встроенная функция `map` возвращает итератор, который применяет заданную функцию к каждому элементу обрабатываемой последовательности, выдавая результаты так, как это сделал бы генератор[42]. Например, один из способов суммирования списка списков – определение итеративного применения (map) функции `sum` к элементам списка:

```
>>> mylists = [[1, 2, 3], [10, 20, 30], [25, 75, 100]]
>>> list(map(sum, mylists))
[6, 60, 200]
```

(Здесь необходимо явное обратное приведение к типу списка `list`, потому что функция `map` возвращает объект, похожий на генератор.) Эта инструкция равнозначна операции генерации списка:

```
>>> [sum(l) for l in mylists]
[6, 60, 200]
```

Функция `map` иногда полезна, но становится потенциальной причиной создания весьма запутанного и непонятного кода, поэтому в основном рекомендуется отдавать предпочтение спискам или методу генерации генераторов. Это замечание относится и к встроенной функции `filter`, создающей итератор из элементов заданной последовательности, для которых предоставленная функция возвращает значение `True`. В следующем примере генерируются нечетные целые числа меньше 10: эта функция возвращает `x % 2`, и это выражение вычисляет результат 0, равнозначный False, если число `x` четное:

```
>>> list(filter(lambda x: x%2, range(10)))
[1, 3, 5, 7, 9]
```

Но и в подобном случае генератор списка более выразителен и понятен:

```
>>> [x for x in range(10) if x % 2]
[1, 3, 5, 7, 9]
```

4.3.7 ◊ Выражения присваивания: морж-оператор

В Python 3.8 введен новый элемент синтаксиса, позволяющий выполнять присваивание переменной внутри выражения. Привычное выражение языка Python, такое как `2 + 2` или `x == 'a'`, возвращает значение (которым может быть

[42] Конструкции, подобные `map`, часто используются в функциональном программировании.

None). Инструкции Python формируются из выражений и в общем случае оказывают некоторое воздействие на состояние программы (например, они присваивают значение переменной или проверяют некоторое условие). Возможность присваивания переменной внутри выражения может привести к более компактному коду с меньшими повторениями. Например, следующий код, проверяющий, содержится ли в строке меньше 10 символов, и выводящий информативное сообщение об ошибке:

```
>>> s = 'A string with too many characters'
>>> if len(s) > 10:
...     print(f's has {len(s)} characters. The maximum is 10.')
...

s has 33 characters. The maximum is 10.
```

Проблема в этом коде заключается в том, что длина строки вычисляется дважды (первый раз при проверке, второй – при выводе сообщения). Чтобы избежать двойного вычисления, можно выполнить присваивание:

```
>>> slen = len(s)
>>> if slen > 10:
...     print(f's has {slen} characters. The maximum is 10.')
...
```

Но существует и более компактный способ, который позволяет «сэкономить» одну строку кода, – выражение присваивания (assignment expression). Синтаксическую конструкцию `a := b` можно использовать для присваивания `a` значения `b` (точнее: для связывания `a` со значением `b`) в контексте какого-либо выражения (например, в условном выражении) вместо выполнения присваивания в отдельной инструкции. То есть такой способ позволяет присваивать значение, а затем возвращает это значение в противоположность поведению обычной операции присваивания Python (которая ничего не возвращает). Таким образом:

```
>>> if (slen := len(s)) > 10:
...     print(f's has {slen} characters. The maximum is 10.')
...

s has 33 characters. The maximum is 10.
```

Символ `:=` по общему мнению напоминает глаза и клыки моржа, поэтому стал широко известен как «морж-оператор» (walrus operator). Следует особо отметить, что такие выражения присваивания в общем случае должны быть заключены в круглые скобки.

Пример П4.14. Правильный вариант применения выражения присваивания – повторное использование значения, вычисление которого может оказаться весьма сложным и затратным, например при генерации списка:

```
filtered_values = [f(x) for x in values if f(x) >= 0]
```

Здесь можно воспользоваться оператором := для присваивания значения, возвращаемого функцией f(x), и одновременной проверки на положительность этого же значения:

```
filtered_values = [val for x in values if (val := f(x)) >= 0]
```

Еще один пример – рассмотрим следующий блок кода, считывающий и обрабатывающий содержимое большого файла фрагментами по 4 Кб:

```
CHUNK_SIZE = 4096
chunk = fi.read(CHUNK_SIZE)
while chunk:
    process_chunk(chunk)
    chunk = fi.read(CHUNK_SIZE)
```

Этот блок кода можно записать более понятно и просто:

```
while chunk := fi.read(CHUNK_SIZE):
    process_chunk(chunk)
```

(Обратите внимание: в этом случае нет необходимости заключать выражение присваивания в круглые скобки.)

Выражения присваивания являются дополнением языка Python, вызывающим споры, и не всегда делают код более понятным и удобным для чтения. В этой книге выражения присваивания используются не слишком часто, поскольку всегда существует другой способ, который надежно работает в версиях Python 3, предшествующих версии 3.8.

4.3.8. Упражнения

Вопросы

В4.3.1. Переписать список лямбда-функций, созданный в примере П4.11, с использованием одного генератора списка.

В4.3.2. Что делает приведенный ниже код, и как он работает?

```
>>> nmax = 5
>>> x = [1]
>>> for n in range(1,nmax+2):
...     print(x)
...     x = [([0] + x)[i] + (x + [0])[i] for i in range(n+1)]
...
```

В4.3.3. Заданы списки

```
>>> a = ['A', 'B', 'C', 'D', 'E', 'F', 'G']
>>> b = [4, 2, 6, 1, 5, 0, 3]
```

Определить и объяснить вывод результатов выполнения следующих инструкций:

```
а) [a[x] for x in b]
б) [a[x] for x in sorted(b)]
в) [a[b[x]] for x in b]
г) [x for (y, x) in sorted(zip(b, a))]
```

В4.3.4. Словари – это структуры данных, в которых (с версии Python 3.6) пары ключ-значение сохраняются в порядке их вставки. Написать однострочную инструкцию Python, возвращающую список пар `(ключ, значение)`, отсортированный по ключам. Предполагается, что все ключи имеют один и тот же тип данных (объяснить, почему это важно). Выполнить упражнение повторно для получения списка, упорядоченного по значениям словаря.

В4.3.5. В полицейском телесериале «Прослушка» (The Wire) торговцы наркотиками шифруют номера телефонов с помощью простого подстановочного шифра, основанного на стандартной раскладке клавиатуры телефона. Каждая цифра телефонного номера, за исключением 5 и 0, заменяется на соответствующую цифру, расположенную по другую сторону от клавиши 5 («прыжок через пятерку»), а 5 и 0 меняются местами. Таким образом, номер 555-867-5309 становится номером 000-243-0751. Напишите однострочную инструкцию для зашифрования и расшифрования номеров телефонов этим способом.

В4.3.6. Встроенная функция `sorted` и метод последовательностей `sort` требуют, чтобы элементы последовательности имели сравнимые друг с другом типы: при использовании этих методов возникает критическая ошибка, например, если список содержит строки и числа. Но часто список содержит числа и специальное значение `None` (возможно, обозначающее отсутствие данных). Разработать способ сортировки такого списка с передачей в аргументе `key` лямбда-функции. Значения `None` должны размещаться в конце отсортированного списка.

В4.3.7. Использовать выражение присваивания (морж-оператор):

а) в цикле `while` для определения наименьшего числа Фибоначчи, большего 5000;
б) в цикле `while` для эхо-отображения ввода пользователя в версии с использованием букв нижнего регистра (применить встроенную функцию `input`) до тех пор, пока пользователь не введет слово `exit`.

Задачи

З4.3.1. Использовать генератор списка для вычисления следа матрицы M (т. е. суммы ее диагональных элементов). Совет: встроенная функция `sum` принимает итерируемый объект и возвращает сумму его значений.

З4.3.2. Шифр с заменой ROT13 зашифровывает строку, заменяя каждую букву другой буквой, расположенной на 13 позиций дальше в алфавите (при необходимости перемещение по алфавиту зацикливается). Например, `a` → `n` и `p` → `c`.

а) Для заданного слова, записанного в виде строки, состоящей только из символов (букв) нижнего регистра, использовать генератор списка для созда-

ния версии этой строки в кодировке ROT-13. Совет: в Python есть встроенная функция `ord`, позволяющая преобразовать символ в соответствующий числовой код Unicode (например, `ord('a')` возвращает `97`), а другая встроенная функция `chr` выполняет противоположное преобразование (например, `chr(122)` возвращает символ `'z'`).

б) Расширить возможности генератора списка для зашифрования предложений, состоящих из слов (в нижнем регистре), разделенных пробелами, в форме предложения ROT-13 (в котором зашифрованные слова также разделены пробелами).

34.3.3. В книге «Наука нового типа»[43] Стивен Вольфрам (Stephen Wolfram) описывает множество простых одномерных клеточных автоматов, в которых каждая ячейка может содержать только одно из двух возможных значений: «on» или «off». Ряд («строка») из ячеек инициализируется в некотором состоянии (например, одна из ячеек строки содержит значение «on»), затем переходит в новое состояние в соответствии с правилом, определяющим последующее состояние ячейки («on» или «off») по ее значению и по значениям двух ближайших соседей. Существует $2^3 = 8$ различных состояний для этих трех «родительских» ячеек, рассматриваемых совместно, следовательно, $2^8 = 256$ различных правил клеточного автомата, т. е. состояние ячейки i в следующем поколении определяется состояниями ячеек $i - 1$, i и $i + 1$ в текущем поколении.

Эти правила нумеруются от 0 до 255 в соответствии с бинарной нумерацией, характеризуемой восемью различными исходами (результатами), каждый из которых определяется восемью возможными родительскими состояниями. Например, правило 30 порождает результат (off, off, off, on, on, on, on, off) или (00011110) из родительских состояний, заданных в порядке, показанном на рис. 4.1. Эволюцию ячеек можно изобразить с помощью вывода строки, соответствующей каждому поколению, размещенной под своей родительской строкой, как показано на рис. 4.1.

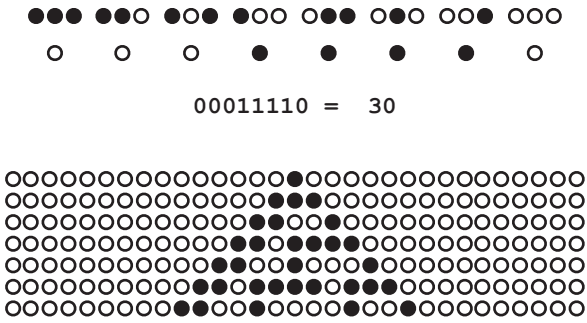


Рис. 4.1. Правило 30 для одномерного клеточного автомата Вольфрама с двумя возможными состояниями и первые семь поколений

Написать программу для вывода нескольких первых строк, сгенерированных по правилу 30 в командной строке, начиная с единственной ячейки со

[43] *Wolfram S.* (2002). A New Kind of Science, Wolfram Media.

значением «on» в середине строки из 80 ячеек. Использовать символ звездочки для обозначения «on» и символ пробела для представления значения «off» в ячейке.

34.3.4. Файл *iban_lengths.txt*, доступный по адресу https://scipython.com/ex/bdd, содержит два столбца данных: двухбуквенный код страны и длину международного номера банковского счета IBAN (International Bank Account Number):

```
AL 28
AD 24
...
GB 22
```

Приведенный ниже фрагмент кода выполняет парсинг этого файла для преобразования его содержимого в словарь длин, а ключом является код страны:

```
iban_lengths = {}
with open('iban_lengths.txt') as fi:
    for line in fi.readlines():
        fields = line.split()
        iban_lengths[fields[0]] = int(fields[1])
```

Использовать лямбда-функцию и генератор списка для получения того же результата: а) в двух строках кода; б) в одной строке кода.

34.3.5. Степенное множество множества S, обозначаемое как P(S), – это множество всех подмножеств S, включая пустое множество и само множество S. Например:

$$P(\{1, 2, 3\}) = \{\{\}, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}.$$

Написать программу, которая использует генератор для возврата степенного множества (множества всех подмножеств) для заданного множества.

Совет: необходимо преобразовать исходное множество в упорядоченную последовательность, например в кортеж. Для каждого элемента этой последовательности возвращается степенное множество, сформированное из всех последующих элементов, включающее и исключающее выбранный элемент. Не забудьте преобразовать кортежи обратно в множества после завершения обработки.

34.3.6. Корпус (языка) Brown Corpus – это набор из 500 образцов (американского) англоязычного текста, сформированный в 1960-х гг. для использования в области компьютерной лингвистики. Файл с этими образцами Brown Corpus можно загрузить из репозитория https://nltk.github.com/nltk_data/packages/corpora/brown.zip.

Каждый образец в этом корпусе состоит из слов, помеченных тегами, обозначающими части речи и помещенными после символа слеша. Например:

```
    The/at football/nn opponent/nn on/in homecoming/nn is/bez ,/, of/in
course/nn ,/, selected/vbn with/in the/at view/nn that/cs
```

Здесь слово The помечено как артикль (тег /at), слово football – как существительное (тег /nn) и т. д. Полный список тегов приведен в сопровождающем руководстве[44].

Написать программу, анализирующую Brown Corpus и возвращающую список восьмибуквенных слов, характерным свойством каждого из которых являются все возможные двухбуквенные комбинации, встречающиеся в точности дважды. Например, двухбуквенная комбинация pc присутствует только в словах topcoats и upcoming; комбинация mt встречается только в словах boomtown и undreamt.

4.4 Сервисы операционной системы

4.4.1 Модуль sys

Модуль sys предоставляет конкретные, зависимые от операционной системы параметры и функции. Многие из них представляют интерес только для достаточно продвинутых пользователей менее распространенных реализаций Python (например, подробности реализации арифметики с плавающей точкой могут отличаться в разных системах, но, вероятнее всего, одинаковы на всех самых распространенных платформах – см. раздел 10.1). Однако в модуле sys представлены также некоторые полезные и важные параметры и функции, которые описаны в следующих подразделах.

Список *sys.argv*

Список sys.argv содержит аргументы командной строки, передаваемые в Python-программу при ее выполнении. Это список строк. Первый элемент sys.argv[0] – имя самой программы. Это обеспечивает некоторую степень интерактивности без необходимости чтения из файлов конфигурации и обязательности прямого ввода пользователя, а также означает, что другие программы и/или скрипты командной оболочки могут вызывать Python-программу и передавать ей конкретные входные значения или параметры настройки. Например, рассмотрим простой скрипт возведения в квадрат заданного числа:

```
# square.py
import sys

n = int(sys.argv[1])
print(n, 'squared is', n**2)
```

(Обратите внимание: необходимо преобразовать входное значение в число типа int, потому что оно хранится в списке sys.argv как строка.) При запуске этой программы в командной строке с передачей аргумента

```
python square.py 3
```

выводится ожидаемый результат:

[44] Руководство доступно здесь: http://clu.uni.no/icame/manuals/BROWN/INDEX.HTM, хотя сами теги лучше описаны в статье «Википедии»: https://en.wikipedia.org/wiki/Brown_Corpus.

```
3 squared is 9
```

Но поскольку значение n не было жестко закодировано в исходном коде, ту же программу можно выполнить с другим аргументом:

```
python square.py 4
```

и получить правильный результат:

```
4 squared is 16
```

Метод *sys.exit*

Вызов sys.exit приводит к завершению работы программы и к выходу из программной среды Python. Это происходит «аккуратно», поэтому сначала выполняются все команды в блоке finally конструкции try и закрываются все открытые файлы. Необязательным аргументом для sys.exit может быть любой объект: если это целое число, то оно передается в командную оболочку, которая, как предполагается, знает, что с ним делать[45]. Например, 0 обычно означает «успешное» завершение программы, а ненулевое значение сообщает об ошибке некоторого типа. Отсутствие аргумента или передача значения None равнозначны 0. Если в качестве аргумента для sys.exit задан любой другой объект, то он передается в поток stderr, реализацию Python стандартного потока ошибок. Например, строка выводится в консоли как сообщение об ошибке (если в командной оболочке не было выполнено перенаправление потоков вывода).

Пример П4.15. Распространенный способ помощи пользователям при работе со скриптами, которые принимают аргументы из командной строки, – вывод сообщения-подсказки, если скрипт используется неправильно, как показано в листинге 4.3.

Листинг 4.3. Вывод сообщения-подсказки для скрипта, принимающего аргументы из командной строки

```
# square.py
import sys

try:
    n = int(sys.argv[1])
except (IndexError , ValueError):
    sys.exit('Please enter an integer , <n>, on the command line.\nUsage: '
             'python {:s} <n>'.format(sys.argv[0]))
print(n, 'squared is', n**2)
```

Здесь выводится понятное сообщение об ошибке и завершается работа программы, если не был задан аргумент командной строки (т. е. при обращении по индексу sys.argv[1] генерируется исключение IndexError) или строку аргумента командной строки невозможно преобразовать в целое число (в этом случае попытка преобразования с помощью метода int приводит к генерации исключения ValueError). Ниже показан пример вывода этой программы:

---

[45] По крайней мере, если это число в диапазоне 0–127. Для значений вне этого диапазона могут быть получены неопределенные результаты.

```
$ python square.py hello
Please enter an integer, <n>, on the command line.
Usage: python square.py <n>

$ python square.py 5
5 squared is 25
```

4.4.2 Модуль os

Модуль os предоставляет разнообразные интерфейсы операционной системы способом, не зависящим от какой-либо конкретной платформы. Многочисленные функции и параметры этого модуля подробно описаны в официальной документации[46], но некоторые наиболее важные из них рассматриваются в этом подразделе.

Информация о процессе

В программной среде Python процесс (process) – это конкретный экземпляр приложения на языке Python, которое выполняется определенной программой (или предоставляет командную оболочку Python для интерактивного использования). Модуль os предлагает ряд функций для извлечения информации о контексте, в котором работает процесс Python. Например, функция os.uname() возвращает информацию об операционной системе, в которой функционирует Python, и о сетевом имени компьютера, выполняющего данный процесс.

Одна из функций имеет специфическое предназначение: os.getenv(*key*) возвращает значение переменной среды *key*, если такая переменная существует (или значение None, если переменной не существует). Многие переменные среды зависят от конкретной системы, но к общим для большинства систем относятся следующие:

- HOME – путь к домашнему каталогу пользователя;
- PWD – текущий рабочий каталог;
- USER – текущее имя пользователя;
- PATH – переменная, содержащая системный путь поиска (может содержать несколько путей поиска).

Например, в моей системе:

```
>>> os.getenv('HOME')
'/Users/christian'
```

Команды файловой системы

Часто возникает необходимость в перемещении по дереву системных каталогов и в работе с файлами и каталогами непосредственно в Python-программе. Модуль os предоставляет для таких операций функции, перечисленные в табл. 4.4. Разумеется, при этом существует определенная степень опасности: Python-программа может сделать все, что может сделать пользователь, в том числе переименовывать и удалять файлы.

[46] https://docs.python.org/3/library/os.htm.

Таблица 4.4. Модуль os: некоторые команды файловой системы

Функция	Описание
`os.listdir(path='.')`	Выводит список элементов каталога, заданного аргументом *path* (если каталог не задан, то по умолчанию выводится содержимое текущего рабочего каталога)
`os.remove(path)`	Удаляет файл, заданный аргументом *path* (если *path* – каталог, то генерируется исключение `OSError`; необходимо использовать функцию `os.rmdir`)
`os.rename(old_name, new_name)`	Выполняет переименование файла или каталога *old_name* в *new_name*. Если файл с именем *new_name* уже существует, то он будет замещен (с учетом пользовательских прав доступа)
`os.rmdir(path)`	Удаляет каталог, заданный аргументом *path*. Если каталог не пуст, то генерируется исключение `OSError`
`os.mkdir(path)`	Создает каталог, заданный аргументом *path*
`os.system(command)`	Выполняет команду *command* в порожденном экземпляре командной оболочки. Если команда генерирует какой-либо вывод, то он перенаправляется в стандартный поток вывода интерпретатора `stdout`

Операции с путевым именем[47]

Модуль `os.path` предоставляет группу полезных функций для работы с путевыми именами. Версия этой библиотеки, установленная вместе с Python, будет именно той, которая подходит для операционной системы, на которой она работает (например, на компьютере с ОС Windows компоненты путевого имени разделяются символом обратного слеша \, тогда как в системах Unix и Linux используется символ (обычного) слеша /.

Наиболее часто функции модуля `os.path` используются для поиска (выделения) имени файла в путевом имени (`basename`), для проверки существования файла или каталога (`exists`), для объединения строк, чтобы сформировать полное путевое имя (`join`), для разделения имени файла на основу («root») и расширение («extension») (`splitext`), а также для определения времени последнего изменения файла (`getmtime`). Все эти часто применяемые методы кратко описаны в табл. 4.5.

Таблица 4.5. Модуль os.path: часто применяемые операции с путевым именем

Функция	Описание
`os.path.basename(path)`	Возвращает имя файла из путевого имени *path*, определяющего относительный или абсолютный путь к файлу: обычно это означает собственно имя файла
`os.path.dirname(path)`	Возвращает имя каталога из путевого имени *path*
`os.path.exists(path)`	Возвращает `True`, если каталог или файл, заданный в *path*, существует, иначе возвращает `False`
`os.path.getmtime(path)`	Возвращает время последнего изменения файла `path`

[47] В этом подразделе рассматриваются низкоуровневые компоненты модуля `os.path`. С версии Python 3.4 модуль стандартной библиотеки `pathlib` стал доступным: в нем предоставляется высокоуровневая объектно-ориентированная методика работы с путевыми именами файловой системы, которая может оказаться более внятной. Более подробно см. здесь: https://docs.python.org/3/library/pathlib.html.

Окончание табл. 4.5

Функция	Описание
`os.path.getsize(path)`	Возвращает размер файла `path` в байтах
`os.path.join(path1, path2,...)`	Возвращает путевое имя, сформированное объединением путевых компонентов *path1*, *path2* и т. д. с применением разделителя каталогов, соответствующего используемой операционной системе
`os.path.split(path)`	Разделяет путевое имя *path* на каталог и имя файла, возвращаемые как кортеж (равнозначно вызову функций `dirname` и `basename` соответственно)
`os.path.splitext(path)`	Разделяет путевое имя *path* на основу («root») и расширение («extension») (возвращаемые как пара кортежа)

Ниже показаны примеры некоторых операций, выполняемых с файлом */home/brian/test.py*:

```
>>> os.path.basename('/home/brian/test.py')
'test.py'                          # Только имя файла.

>>> os.path.dirname('/home/brian/test.py')
'/home/brian'                      # Только каталог.

>>> os.path.split('/home/brian/test.py')
('/home/brian', 'test.py')         # Каталог и имя файла в кортеже.

>>> os.path.splitext('/home/brian/test.py')
('/home/brian/test', '.py')        # Путь к файлу с основой и расширение файла в кортеже.

>>> os.path.join(os.getenv('HOME'), 'test.py')
'/home/brian/test.py'              # Объединение каталогов и/или имени файла.

>>> os.path.exists('/home/brian/test.py')
False                              # Файл не существует.
```

Пример П4.16. Предположим, что существует каталог с файлами данных, идентифицируемых по именам в формате `data-DD-Mon-YY.txt`, где `DD` – число месяца из двух цифр, `Mon` – сокращенное обозначение месяца из трех букв, `YY` – две последние цифры года, например `'02-Feb-10'`. Программа в листинге 4.4 преобразовывает имена файлов в формат `data-YYYY-MM-DD.txt`, чтобы упорядочение по алфавиту имен файлов размещало их в правильном хронологическом порядке.

Листинг 4.4. Переименование файлов данных для упорядочения по дате

```
# eg4-osmodule.py
import os
import sys

months = ['jan', 'feb', 'mar', 'apr', 'may', 'jun',
          'jul', 'aug', 'sep', 'oct', 'nov', 'dec']
```

```
dir_name = sys.argv[1]
for filename in os.listdir(dir_name):
    # Имя файла должно быть представлено в формате 'data-DD-MMM-YY.txt'
    d, month , y = int(filename[5:7]), filename[8:11], int(filename[12:14])
    m = months.index(month.lower())+1                                    ❶

    newname = 'data-20{:02d}-{:02d}-{:02d}.txt'.format(y, m, d)
    newpath = os.path.join(dir_name, newname)
    oldpath = os.path.join(dir_name, filename)
    print(oldpath , '->', newpath)
    os.rename(oldpath , newpath)
```

❶ Номер месяца вычисляется по индексу соответствующего сокращенного обозначения месяца в списке `months` с добавлением 1, так как в Python индексы списка нумеруются, начиная с 0.

Например, рассмотрим каталог *testdir*, содержащий следующие файлы:

```
data-02-Feb-10.txt
data-10-Oct-14.txt
data-22-Jun-04.txt
data-31-Dec-06.txt
```

При выполнении команды `python eg4-osmodule.py testdir` выводится следующий результат:

```
testdir/data-02-Feb-10.txt -> testdir/data-2010-02-02.txt
testdir/data-10-Oct-14.txt -> testdir/data-2014-10-10.txt
testdir/data-22-Jun-04.txt -> testdir/data-2004-06-22.txt
testdir/data-31-Dec-06.txt -> testdir/data-2006-12-31.txt
```

Также см. задачу 34.4.4 и модуль `datetime` (раздел 4.5.3).

4.4.3 Упражнения

Задачи

34.4.1. Изменить генератор последовательности чисел-градин из задачи 32.5.7 для генерации последовательности чисел-градин, начиная с произвольного положительного целого числа, которое вводит пользователь в командной строке (использовать `sys.argv`). Необходимо правильно («аккуратно») обрабатывать случай, когда пользователь забывает задать исходное число `n` или указывает некорректное значение для `n`.

34.4.2. Формула гаверсинуса (haversine) позволяет вычислять кратчайшее расстояние (по дуге большой окружности, или по ортодромии) d между двумя точками на сфере радиуса R по значениям долготы (λ_1, λ_2) и широты (ϕ_1, ϕ_2) этих точек:

$$d = 2R \arcsin\left(\sqrt{\text{haversin}(\phi_2 - \phi_1) + \cos\phi_1 \cos\phi_2 \,\text{haversin}(\lambda_2 - \lambda_1)}\right),$$

где функция haversine для угла α определяется формулой

$$\text{haversin}(\alpha) = \sin^2(\alpha/2).$$

Написать программу для вычисления кратчайшего расстояния в километрах между двумя точками на поверхности Земли (рассматривая Землю как сферу с радиусом 6378.1 км), заданными как два аргумента командной строки, каждый из которых представлен в форме пары значений `широта`, `долгота` в градусах. Например, расстояние между Парижем и Римом вычисляется так:

```
python greatcircle.py 48.9,2.4 41.9,12.5
1107 km
```

34.4.3. Написать программу для создания каталога *test* в домашнем каталоге пользователя и поместить в этот каталог 20 файлов в формате SVG (Scalable Vector Graphics), изображающих маленький закрашенный красным цветом круг внутри большой незакрашенной черной окружности. Например:

```
<?xml version="1.0" encoding="utf -8"?>
                 <svg xmlns="http://www.w3.org/2000/svg"
                      xmlns:xlink="http://www.w3.org/1999/xlink"
                      width="500" height="500" style="background: #ffffff">
<circle cx="250.0" cy="250.0" r="200" style="stroke: black; stroke -width: 2px; fill:
none;"/>
<circle cx="430.0" cy="250.0" r="20" style="stroke: red; fill: red;"/>
</svg>
```

В каждом файле маленький красный круг должен сдвигаться вдоль внутренней границы большой окружности так, чтобы эти 20 файлов в совокупности могли создать эффект анимации.

Один из способов достижения такого результата – использование бесплатной программы ImageMagick (www.imagemagick.org/). Проследить за тем, чтобы все SVG-файлы имели имена *fig00.svg*, *fig01.svg* и т. д. Затем выполнить в командной строке операционной системы следующую команду:

```
convert -delay 5 -loop 0 fig*.svg animation.gif
```

для создания анимированного изображения в формате GIF.

34.4.4. Изменить программу из примера П4.16 (листинг 4.4) так, чтобы она перехватывала следующие ошибки и аккуратно обрабатывала их:

- пользователь не указал имя каталога в командной строке (необходимо вывести сообщение-подсказку по использованию программы);
- заданный каталог не существует;
- имя файла в заданном каталоге не соответствует корректному формату;
- имя файла записано в правильном формате, но сокращенное обозначение месяца не определено (не найдено в списке допустимых обозначений месяцев).

В первых двух случаях программа должна завершать свое выполнение. В третьем и четвертом случаях некорректное имя файла должно быть пропущено.

4.5 Модули и пакеты

Как мы уже видели, Python является вполне модульным языком, и его функциональность не ограничивается основными принципами программирования (т. е. встроенными методами и структурами данных, с которыми мы имели дело до сих пор). В любой программе доступны также расширенные функциональные свойства, включаемые с помощью команды `import`. Эта команда создает ссылку на модули, представляющие собой обычные Python-файлы, содержащие определения и инструкции. Встретив строку

```
import <module>
```

интерпретатор Python выполняет инструкции в файле *<module>.py* и включает имя модуля `<module>` в текущее пространство имен, после чего атрибуты, определяемые этим модулем, становятся доступными с применением синтаксиса точки: `<module>.<attribute>`.

Определить собственный модуль так же просто, как поместить исходный код в файл *<module>.py*, расположенный в локации, в которой интерпретатор Python может найти его (для небольших проектов это обычно тот же каталог, в котором расположена программа, импортирующая модуль). С учетом синтаксиса команды `import` необходимо избегать такого именования модуля, которое не соответствует корректному идентификатору Python (см. раздел 2.2.3). Например, имя файла *<module>.py* не должно содержать символ дефиса и не должно начинаться с цифры. Не рекомендуется давать модулю имя, совпадающее с именем какого-либо встроенного модуля (такого как `math` или `random`), поскольку встроенные модули имеют более высокий приоритет при импорте Python.

Пакет (package) языка Python – это просто структурированный комплект модулей, размещенный в некотором каталоге файловой системы. Пакеты представляют собой естественный способ организации и распространения крупных проектов на языке Python. Для создания пакета файлы модулей помещаются в один каталог вместе с файлом *__init__.py*. Этот файл запускается при импорте пакета и может выполнять некоторые операции инициализации и собственные команды импорта. Если особенная инициализация не требуется, то этот файл может быть пустым (длиной ноль байтов), но он обязательно должен существовать, чтобы Python распознавал такой каталог как пакет.

Например, пакет NumPy (см. главу 6) существует как показанный ниже каталог (некоторые файлы и подкаталоги не показаны для экономии места):

```
numpy/
    __init__.py
    core/
    fft/
        __init__.py
        fftpack.py
        info.py
        ...
    linalg/
        __init__.py
        linalg.py
        info.py
        ...
    polynomial/
        __init__.py
        chebyshev.py
        hermite.py
        legendre.py
        ...
    random/
    version.py
    ...
```

Здесь, например, `polynomial` – это вложенный пакет в структуре пакета `numpy`, содержащий несколько модулей, в том числе модуль `legendre`, который можно импортировать:

```
import numpy.polynomial.legendre
```

Чтобы избежать использования развернутого синтаксиса с точкой при обращении к атрибутам этого модуля, удобнее воспользоваться такой командой:

```
from numpy.polynomial import legendre
```

В табл. 4.6 перечислены некоторые основные бесплатные модули и пакеты Python для программных приложений общего назначения, а также для вычислительной и научной работы. Некоторые из них устанавливаются вместе с основным дистрибутивом Python (стандартная библиотека Standard Library)[48], другие можно загрузить и установить отдельно. Перед тем как приступить к реализации какого-либо собственного алгоритма, проверьте, не включена ли такая реализация в один из существующих пакетов Python.

Таблица 4.6. Модули и пакеты Python. Помеченные звездочкой (*) компоненты не являются частью Python Standard Library, поэтому должны устанавливаться отдельно, например с помощью утилиты pip

Модуль/пакет	Описание
`os, sys`	Сервисы операционной системы, описанные в разделе 4.4
`math, cmath`	Математические функции, представленные в разделе 2.2.2

[48] Полный список компонентов стандартной библиотеки можно найти здесь: https://docs.python.org/3/library/index.html.

Окончание табл. 4.6

Модуль/пакет	Описание
`random`	Генератор случайных чисел (см. раздел 4.5.1)
`collections`	Типы данных для контейнеров, расширяющие функциональность словарей, кортежей и т. п.
`itertools`	Инструменты для эффективных итераторов, расширяющие функциональность простых циклов Python
`glob`	Расширение шаблона путевого имени в стиле Unix
`datetime`	Парсинг и обработка дат и времени (см. раздел 4.5.3)
`fractions`	Арифметика рациональных чисел (правильных дробей)
`re`	Регулярные выражения
`argparse`	Парсер для ключей и аргументов командной строки
`urllib`	Открытие, чтение и парсинг URL (включая веб-страницы) (см. раздел 4.5.2)
`* Django (django)`	Широко известная рабочая среда для веб-приложений
`* pyparsing`	Лексический парсер для простых грамматик
`pdb`	Отладчик языка Python
`logging`	Встроенный в Python модуль ведения журналов
`xml, lxml`	Парсеры языка разметки XML
`* VPython (visual)`	Трехмерная визуализация
`unittest`	Рабочая среда модульного тестирования для систематического проведения тестирования и валидации отдельных единиц (модулей) кода (см. раздел 10.3.4)
`* NumPy (numpy)`	Научные вычисления и численные методы (пакет подробно описан в главе 6)
`* SciPy (scipy)`	Научные вычислительные алгоритмы (пакет подробно описан в главе 8)
`* Matplotlib (matplotlib)`	Графическое отображение данных (см. главы 3 и 7)
`* SymPy (sympy)`	Символические вычисления (компьютерная алгебра)
`* pandas`	Обработка и анализ данных с использованием табличных структур данных
`* scikit-learn`	Машинное обучение
`* Beautiful Soup 4 (beautifulsoup4)`	Парсер языка разметки HTML с возможностями обработки некорректно сформированных документов

Несмотря на существование других менеджеров пакетов[49], приложение `pip`[50] стало фактическим стандартом. Приложение `pip` обычно устанавливается по умолчанию вместе с большинством дистрибутивов Python и великолепно выполняет работу по управлению версиями и зависимостями пакетов. Для установки пакета *package* из командной строки используется показанный ниже синтаксис:

[49] Например, `conda` из дистрибутива Anaconda – см. раздел 1.3.

[50] Полную документацию см. здесь: https://pip.pypa.io/en/stable/.

```
pip install package              # Установка самой последней версии пакета.
pip install package==X.Y.Z       # Установка версии X.Y.Z.
pip install 'package>=X.Y.Z'     # Установка версии не ниже X.Y.Z.
```

Для корректного удаления (деинсталляции) пакета применяется следующая команда:

```
pip uninstall package
```

4.5.1 Модуль random

Для имитаций, моделирования и некоторых вычислительных алгоритмов часто требуется генерация случайных чисел из некоторого распределения. Тема генерации случайных чисел весьма сложна и интересна, но в данном случае для нас важен тот факт, что в Python, как и в большинстве других языков программирования, имеется реализация генератора псевдослучайных чисел (ГПСЧ). Этот алгоритм генерирует последовательность чисел, свойства которых приближенно соответствуют свойствам «истинных» случайных чисел. Такие последовательности определяются с помощью исходного состояния seed (семя, посев) и при одном и том же значении seed всегда одинаковы: в этом смысле последовательности являются детерминированными. Это может быть положительным свойством (можно повторно воспроизвести вычисление с использованием определенной последовательности случайных чисел) или отрицательным свойством (например, в криптографии, когда последовательность случайных чисел должна храниться в секрете). Любой ГПСЧ будет генерировать последовательность, которая в конце концов повторяется, но у качественного генератора весьма длительный период неповторяющихся чисел. Реализованный в Python ГПСЧ – это вихрь Мерсенна (Mersenne Twister), надежный, хорошо изученный алгоритм с периодом $2^{19937} - 1$ (это число, содержащее более 6000 знаков по основанию 10).

Генерация случайных чисел

Исходный посев (seed) для генератора случайных чисел можно выполнить с помощью любого хешируемого объекта (например, неизменяемого объекта, такого как целое число). Сразу после импорта модуля выполняется посев с использованием представления текущего системного времени (если операционная система не предоставляет более эффективный источник случайного посева). Посев для ГПСЧ можно в любой момент изменить с помощью вызова `random.seed`.

Основным методом генерации случайных чисел является `random.random`. Этот метод генерирует случайное число, выбранное из равномерного распределения в полуоткрытом интервале [0,1), т. е. включающем 0, но не включающем 1.

```
>>> import random
>>> random.random()       # "Случайный" посев для ГПСЧ.
0.5204514767709216
>>> random.seed(42)       # Посев для ГПСЧ с использованием конкретного числового значения.
>>> random.random()
0.6394267984578837
>>> random.random()
0.025010755222666936
...
```

```
>>> random.seed(42)         # Повторный посев с тем же значением, что и ранее.
>>> random.random()
0.6394267984578837          # Поэтому последовательность случайных чисел повторяется.
>>> random.random()
0.025010755222666936
```

При вызове `random.seed()` без аргумента выполняется повторный посев для ГПСЧ со «случайным» значением, которое использовалось модулем `random` сразу после импорта.

Для выбора случайного числа с плавающей точкой N из заданного интервала $a \leqslant N \leqslant b$ используется метод `random.uniform(a,b)`:

```
>>> random.uniform(-2., 2.)
-0.899882726523523
>>> random.uniform(-2., 2.)
-1.107157047404709
```

В модуле `random` есть несколько методов для произвольного выбора случайных чисел из неравномерных распределений (см. документацию здесь: https://docs.python.org/3/library/random.html) – это очень важно для случаев, описанных ниже.

Чтобы вернуть число из нормального распределения со средним значением `mu` и стандартным отклонением `sigma`, используется метод `random.normalvariate(mu, sigma)`:

```
>>> random.normalvariate(100, 15)
118.82178896586194
>>> random.normalvariate(100, 15)
97.92911405885782
```

Для выбора случайного целого числа N из заданного интервала $a \leqslant N \leqslant b$ применяется метод `random.randint(a,b)`:

```
>>> random.randint(5, 10)
7
>>> random.randint(5, 10)
10
```

Последовательности случайных чисел

Иногда может потребоваться выбор случайного элемента из некоторой последовательности, например из списка. Это можно сделать с помощью метода `random.choice`:

```
>>> seq = [10, 5, 2, 'ni', -3.4]
>>> random.choice(seq)
-3.4
>>> random.choice(seq)
'ni'
```

Другой метод `random.shuffle` в случайном порядке перемешивает (переставляет) элементы в последовательности (изменяя саму последовательность):

```
>>> random.shuffle(seq)
>>> seq
[10, -3.4, 2, 'ni', 5]
```

Обратите внимание: поскольку случайная перестановка выполняется в самой последовательности, эта последовательность непременно должна быть изменяемой: например, нельзя перемешивать содержимое кортежей.

Наконец, для произвольного выбора списка с *k* неповторяющимися элементами из последовательности или множества (без замены) `population` существует метод `random.sample(population, k)`:

```
>>> raffle_numbers = range(1, 100001)
>>> winners = random.sample(raffle_numbers , 5)
>>> winners
[89734, 42505, 7332, 30022, 4208]
```

Итоговый список содержит элементы в порядке их выбора (первый по индексу элемент – выбранный первым), поэтому можно, например, без предвзятости объявить билет с номером 89 734 выигравшим джекпот, а остальные четыре номера – «победителями второй категории».

Пример П4.17. Парадокс (задача) Монти Холла (Monty Hall problem) – широко известная задача теории вероятностей, излагаемая в форме воображаемого игрового шоу. Участнику предлагается выбрать одну из трех дверей: за одной находится автомобиль, за двумя другими – козы. Участник выбирает дверь, после чего ведущий открывает другую дверь, за которой обнаруживается коза. Ведущий заранее знает, за какой дверью скрыт автомобиль. Затем участнику предлагается изменить выбор, чтобы открыть другую дверь, или оставить в силе свой первоначальный выбор.

Вопреки здравому смыслу наилучшей стратегией для выигрыша машины является изменение выбора, как показано в приведенной ниже имитации в листинге 4.5.

Листинг 4.5. Задача Монти Холла

```
# eg4-montyhall.py
import random

def run_trial(switch_doors, ndoors=3):
    """
    Run a single trial of the Monty Hall problem, with or without switching
    after the game show host reveals a goat behind one of the unchosen doors.
    (switch_doors is True or False). The car is behind door number 1 and the
    game show host knows that. Returns True for a win, otherwise returns False.
    """
    # """
    # Запуск одиночного испытания парадокса (задачи) Монти Холла с изменением или без
    # изменения варианта выбора, после того как ведущий предъявил козу, открыв одну из
    # невыбранных дверей. (Смена выбора двери обозначена как True или False.) Автомобиль
    # находится за дверью номер 1, и ведущий знает это. При выигрыше возвращается значение
    # True, иначе возвращается False.
    # """
```

```
    # Выбор случайной двери из доступных ndoors.
    chosen_door = random.randint(1, ndoors)
    if switch_doors:
        # Обнаружена коза.
        revealed_door = 3 if chosen_door==2 else 2
        # Изменение варианта выбора на любую другую дверь, отличную от выбранной
        # изначально, при одной открытой двери, за которой обнаружилась коза.
        available_doors = [dnum for dnum in range(1, ndoors+1)
                                if dnum not in (chosen_door, revealed_door)]
        chosen_door = random.choice(available_doors)

    # Выигрыш, если выбрана дверь номер 1.
    return chosen_door == 1                                                    ❶

def run_trials(ntrials , switch_doors , ndoors=3):
    """
    Run ntrials iterations of the Monty Hall problem with ndoors doors, with
    and without switching (switch_doors = True or False). Returns the number
    of trials which resulted in winning the car by picking door number 1.
    """
    # """
    # Запуск ntrials итераций задачи Монти Холла с ndoors дверьми с изменением или без
    # изменения варианта выбора (switch_doors = True или False). Возвращает количество
    # испытаний, завершившихся выигрышем автомобиля при выборе двери номер 1.
    # """

    nwins = 0
    for i in range(ntrials):
        if run_trial(switch_doors, ndoors):
            nwins += 1
    return nwins

ndoors, ntrials = 3, 10000
nwins_without_switch = run_trials(ntrials, False, ndoors)
nwins_with_switch = run_trials(ntrials, True, ndoors)

print('Monty Hall Problem with {} doors'.format(ndoors))
print('Proportion of wins without switching: {:.4f}'
       .format(nwins_without_switch/ntrials))
print('Proportion of wins with switching: {:.4f}'
       .format(nwins_with_switch/ntrials))
```

❶ Без нарушения общности условий задачи можно поместить автомобиль за дверью номер 1, оставив для участника возможность выбора любой двери случайным образом.

Чтобы сделать код чуть более интересным, было введено переменное количество дверей в имитации этой задачи (но при этом остается только один автомобиль).

```
Monty Hall Problem with 3 doors
Proportion of wins without switching: 0.3334
Proportion of wins with switching: 0.6737
```

4.5.2 ◊ Пакет `urllib`

Пакет `urllib` в Python 3 – это набор модулей для открытия и извлечения содержимого (контента), на который указывают URL (Uniform Resource Locators), обычно в форме веб-адресов, доступных по протоколу HTTP (HyperText Transfer Protocol), HTTPS или FTP (File Transfer Protocol). В этом разделе предлагается краткая вводная инструкция по использованию пакета `urllib`.

Открытие и чтение URL

Для получения содержимого по URL с использованием протокола HTTP сначала необходимо подготовить HTTP-запрос (HTTP request), создав объект `Request`. Например:

```
import urllib.request
req = urllib.request.Request('https://www.wikipedia.org')
```

Объект `Request` позволяет передавать данные (используя команду GET или POST) и другую информацию о запросе (метаданные, передаваемые в заголовках HTTP, – см. ниже). Но для простого запроса можно просто открыть URL напрямую как объект, подобный файлу, с помощью метода `urlopen()`:

```
response = urllib.request.urlopen(req)
```

Правильной практической методикой является перехват двух основных типов исключений, которые генерируются при выполнении этой инструкции. Первый тип `URLError` генерируется, если сервер не существует или если отсутствует сетевое соединение. Второй тип `HTTPError` генерируется, если сервер возвращает код ошибки (например, `404: Page Not Found`). Эти исключения определяются в модуле `urllib.error`.

```
from urllib.error import URLError , HTTPError
try:
    response = urllib.request.urlopen(req)
except HTTPError as e:
    print('The server returned error code', e.code)
except URLError as e:
    print('Failed to reach server at {} for the following reason:\n{}'
          .format(url, e.reason))
else:
    # Получен ответ об успешном выполнении запроса: OK
```

Предполагая, что метод `urlopen()` отработал успешно, часто не требуются какие-либо дополнительные действия, кроме простого чтения контента из объекта ответа:

```
content = response.read()
```

Контент будет возвращен как строка байтов (bytestring). Для перекодировки ее в строку Python (Unicode) необходимо знать, как именно закодирована исходная строка. Правильный ресурс включает определение используемого на-

бора символов в атрибут `Content-Type` HTTP-заголовка. Его можно применять следующим образом:

```
charset = response.headers.get_content_charset()
html = content.decode(charset)
```

Здесь `html` становится декодированной Unicode-строкой Python. Если в возвращаемых заголовках не указан используемый набор символов, то можно попытаться сделать вероятное предположение (например, установить `charset='utf-8'`).

Запросы GET и POST

Часто необходимо вместе с URL передавать данные для извлечения контента с сервера. Например, при заполнении HTML-формы на веб-странице значения для соответствующих полей должны быть переведены в требуемую кодировку и переданы на сервер в соответствии с протоколами GET или POST.

Модуль `urllib.parse` позволяет кодировать данные из словаря Python в форму, пригодную для передачи на веб-сервер. Ниже приведен пример из «Википедии» для API с использованием запроса GET:

```
>>> url = 'https://wikipedia.org/w/api.php'
>>> data = {'page': 'Monty_Python', 'prop': 'text', 'action': 'parse', 'section': 0}
>>> encoded_data = urllib.parse.urlencode(data)
>>> full_url = url + '?' + encoded_data
>>> full_url
'https://wikipedia.org/w/api.php?page=Monty_Python&prop=text&action=parse&section=0'
>>> req = urllib.request.Request(full_url)
>>> response = urllib.request.urlopen(req)
>>> html = response.read().decode('utf-8')
```

Для выполнения запроса POST вместо добавления перекодированных данных в строку `<url>?` эти данные напрямую передаются в конструктор `Request`:

```
req = urllib.request.Request(url, encoded_data)
```

4.5.3 Модуль `datetime`

Модуль `datetime` предоставляет классы для работы с датами и временем. Существует множество тонкостей, связанных с обработкой таких данных (временные пояса и зоны, различные календари, переход на летнее и зимнее время и т. д.), поэтому полная документация по модулю `datetime` доступна в онлайн-режиме (https://docs.python.org/3/library/datetime.html). В этом разделе представлен краткий обзор наиболее часто встречающихся вариантов использования.

Даты

Объект `datetime.date` представляет конкретный день, месяц и год в идеализированном календаре (предполагается, что используемый в настоящее время григорианский календарь применяется для всех дат в прошлом и в будущем). Для создания объекта `date` необходимо передать в явной форме числовые значения, соответствующие году, месяцу и дню, или вызвать конструктор `date.today`:

```
>>> from datetime import date
>>> birthday = date(2004, 11, 5)      # OK

>>> notadate = date(2005, 2, 29)      # Ошибка: 2005 год не был високосным.

Traceback (most recent call last):
  File "<stdin >", line 1, in <module >
ValueError: day is out of range for month

>>> today = date.today()
>>> today
datetime.date(2014, 12, 6)           # Например.
```

Допустимыми являются даты в интервале от 1/1/1 до 31/12/9999. Парсинг дат с преобразованием в строки и наоборот также поддерживается (см. `strptime` и `strftime`).

Ниже показано применение некоторых полезных методов объекта `date`:

```
>>> birthday.isoformat()      # Формат даты по стандарту ISO 8601: YYYY -MM-DD.
'2004-11-05'

>>> birthday.weekday()        # Понедельник = 0, Вторник = 1, ..., Воскресенье = 6.
4   # (Пятница).

>>> birthday.isoweekday()     # Понедельник = 1, Вторник = 2, ..., Воскресенье = 7.
5

>>> birthday.ctime()          # Вывод времени по стандарту C.
'Fri Nov 5 00:00:00 2004'
```

Кроме того, объекты `date` можно сравнивать (в хронологическом порядке):

```
>>> birthday < today
True

>>> today == birthday
False
```

Время

Объект `datetime.time` представляет (местное) время суток с округлением до ближайшей микросекунды. Для создания объекта `time` необходимо передать числовые значения часов, минут, секунд и микросекунд (в указанном здесь порядке, для пропущенных значений по умолчанию устанавливается нулевое значение).

```
>>> from datetime import time
>>> lunchtime = time(hour=13, minute=30)
>>> lunchtime
datetime.time(13, 30)

>>> lunchtime.isoformat()      # Формат времени по стандарту ISO 8601: HH:MM:SS, если нет
мс.
'13:30:00'
```

```
>>> precise_time = time(4,46,36,501982)
>>> precise_time.isoformat()   # Формат времени по стандарту ISO 8601: HH:MM:SS.мммммм
'04:46:36.501982'

>>> witching_hour = time(24)  # Ошибка: значение часа должно находиться в интервале 0 <=
hour < 24

Traceback (most recent call last):
  File "<stdin >", line 1, in <module >
ValueError: hour must be in 0..23
```

Объект *datetime*

Объект `datetime.datetime` содержит информацию из обоих объектов `date` и `time`: `year`, `month`, `day`, `hour`, `minute`, `second`, `microsecond`. Для этого объекта возможна передача значений для всех перечисленных выше атрибутов в конструктор `datetime`, а также доступны методы `today` (возвращает текущую дату) и `now` (возвращает текущую дату и время):

```
>>> from datetime import datetime           # Весьма опасная операция импорта.
>>> now = datetime.now()
>>> now
datetime.datetime(2020, 1, 27, 10, 27, 35, 762464)

>>> now.isoformat()
'2020-01-27T10:27:35.762464'

>>> now.ctime()
'Mon Jan 27 10:27:35 2020'
```

Форматирование даты и времени

Объекты `date`, `time` и `datetime` поддерживают метод `strftime` для вывода своих значений в виде строки, отформатированной в соответствии с настройками синтаксиса с использованием спецификаторов формата, перечисленных в табл. 4.7.

Таблица 4.7. Спецификаторы формата для методов strftime и strptime. Обратите внимание: многие спецификаторы зависят от локали (например, в системе с немецким языком %A будет генерировать дни недели Sonntag, Montag и т. д.)

Спецификатор	Описание
`%a`	Сокращенное название дня недели (`Sun`, `Mon` и т. д.)
`%A`	Полное название дня недели (Sunday, Monday и т. д.)
`%w`	Номер дня недели (0 = воскресенье, 1 = понедельник, …, 6 = суббота)
`%d`	Дополненное нулем число месяца: 01, 02, 03, …, 31
`%b`	Сокращенное название месяца (Jan, Feb и т. д.)
`%B`	Полное название месяца (January, February и т. д.)
`%m`	Дополненный нулем номер месяца: 01, 02, …, 12
`%y`	Год без века (из двух цифр, с дополнением нулем): 01, 02, …, 99

Окончание табл. 4.7

Спецификатор	Описание
%Y	Год с веком (из четырех цифр, с дополнением нулями): 0001, 0002, ..., 9999
%H	Часы в 24-часовом формате с дополнением нулем: 00, 01, ..., 23
%I	Часы в 12-часовом формате с дополнением нулем: 00, 01, ..., 12
%p	AM или PM (или равнозначные обозначения в конкретной локали)
%M	Минуты (из двух цифр, с дополнением нулем): 00, 01, ..., 59
%S	Секунды (из двух цифр, с дополнением нулем): 00, 01, ..., 59
%f	Микросекунды (из шести цифр, с дополнением нулем): 000000, 000001, ..., 999999
%%	Знак % как литерал

```
>>> birthday.strftime('%A, %d %B %Y')
'Friday , 05 November 2004'

>>> now.strftime('%I:%M:%S on %d/%m/%y')
'10:27:35 on 27/01/20'
```

Для выполнения обратного процесса, т. е. для преобразования строки в объект `datetime`, применяется метод `strptime`:

```
>>> launch_time = datetime.strptime('09:32:00 July 16, 1969', '%H:%M:%S %B %d, %Y')
>>> print(launch_time)
1969-07-16 09:32:00

>>> print(launch_time.strftime('%I:%M %p on %A, %d %b %Y'))
09:32 AM on Wednesday , 16 Jul 1969
```

4.6 ◊ Введение в объектно-ориентированное программирование

4.6.1 Основы объектно-ориентированного программирования

Стили структурного программирования в обобщенном смысле можно разделить на две категории: процедурное и объектно-ориентированное. Программы, которые рассматривались в этой книге до настоящего момента, по своей сущности были процедурными: мы писали функции (которые в других языках обычно называют процедурами или подпрограммами), эти функции вызывались, в них передавались данные, они возвращали значения, полученные в результате внутренних вычислений. Функции, которые мы определяли, не хранят собственные данные и не запоминают свое состояние между вызовами. После определения функции не изменялись.

Другой парадигмой программирования, которая стала широко распространенной благодаря использованию таких языков, как C++ и Java, является объектно-ориентированное программирование (object-oriented programming). В таком контексте объект представляет концепцию некоторого рода: это может

быть физическая сущность, но также возможно представление в виде абстрактного набора компонентов, связанных друг с другом семантически согласованным способом. Объект хранит данные о самом себе (атрибуты) и определяет функции (методы) для обработки данных. Обработка данных может приводить к изменению состояния объекта (т. е. могут измениться некоторые атрибуты объекта). Объект создается (точнее, создается экземпляр класса) из «черновика», называемого классом (class), который обусловливает поведение объекта, определяя его атрибуты и методы.

В действительности мы уже имели дело с объектно-ориентированным программированием, потому что в Python все является объектами. Например, строка Python – это экземпляр класса `str`. Объект `str` содержит собственные данные (последовательность символов, составляющих строку) и предоставляет («предъявляет») группу методов для обработки этих данных. Например, метод `capitalize` возвращает новый строковый объект, созданный из исходной строки посредством перевода ее первой буквы в верхний регистр (преобразование в заглавную, или прописную, букву). Метод `split` возвращает список строк, полученных в результате разделения исходной строки:

```
>>> a = 'hello , aloha , goodbye , aloha'
>>> a.capitalize()
'Hello , aloha , goodbye , aloha'
>>> a.split(',')
['hello', ' aloha', ' goodbye', ' aloha']
```

Даже простое обращение по индексу на самом деле представляет собой вызов метода `__getitem__()`:

```
>>> b = [10, 20, 30, 40, 50]
>>> b.__getitem__(4)
50
```

Таким образом, выражение `a[4]` равнозначно выражению `a.__getitem__(4)`[51].

Широкое распространение объектно-ориентированного программирования, по крайней мере для крупных проектов, в определенной степени обусловлено способом, помогающим разработать концепцию задачи, которую должна решить программа. Часто имеется возможность разделить общую глобальную задачу на компоненты данных и операции, которые должны выполняться с этими данными. Например, банк, обслуживающий физических лиц, работает с людьми, имеющими счета в этом банке. Естественный объектно-ориентированный подход к управлению банком должен заключаться в определении класса `BankAccount` с такими атрибутами, как номер счета, баланс, владелец, и второго класса `Customer` с такими атрибутами, как имя, адрес и дата рождения. Класс `BankAccount` может иметь методы для разрешения (или запрещения) транзакций в зависимости от баланса клиента, а класс `Customer` может, например, содержать методы для вычисления возраста клиента по дате его рождения (см. рис. 4.2).

[51] Синтаксис с использованием двойных символов подчеркивания обычно обозначает имя с некоторым особым смыслом в языке Python.

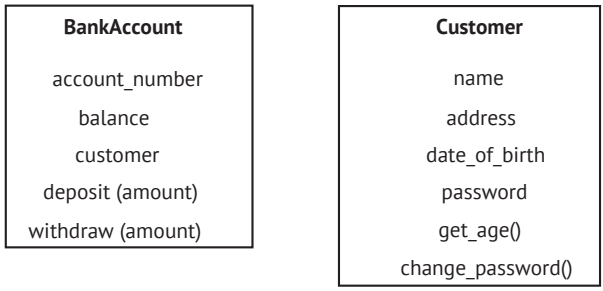


Рис. 4.2. Простые классы, представляющие счет в банке и клиента

Важным аспектом объектно-ориентированного программирования является наследование (inheritance). Часто между объектами существуют взаимоотношения в форме иерархии. Обычно обобщенный тип объекта определяется базовым классом, затем от него производятся (наследуются) специализированные классы с более специфической функциональностью. В рассматриваемом здесь примере с банком могут существовать различные типы банковских счетов: накопительные, текущие (чековые) и т. д. Каждый такой счет является производным от обобщенного основного банковского счета, для которого можно просто определить главные атрибуты, такие как баланс и номер счета. Более специализированные классы банковского счета наследуют (inherit) свойства базового класса, но могут также изменять их, замещая (переопределяя) один или несколько методов, а также добавлять собственные атрибуты и методы. Подобный подход помогает структурировать программу и поощряет многократное использование кода (code reuse) – нет необходимости отдельно объявлять атрибут «номер счета» для накопительных и текущих счетов, поскольку оба класса автоматически унаследовали этот атрибут от базового класса. Если не создается экземпляр самого базового класса, но он служит только в качестве шаблона для производных классов, то такой базовый класс называется абстрактным классом (abstract class).

На рис. 4.3 показано отношение между базовым классом и двумя производными подклассами. Базовый класс `BankAccount` определяет некоторые атрибуты (`account_number`, `balance` и `customer`) и методы (такие как `deposit` и `withdraw`), общие для всех типов счетов. Эти атрибуты и методы наследуются подклассами. Подкласс `SavingsAccount` добавляет атрибут и метод для обработки выплат по процентной ставке для этого типа счета. Подкласс `CurrentAccount`, в свою очередь, добавляет два атрибута, описывающих ежегодную оплату счета и лимит по транзакции снятия денег, а также замещает (переопределяет) метод базового класса `withdraw`, возможно, для проверки превышения лимита перед разрешением операции снятия денег со счета.

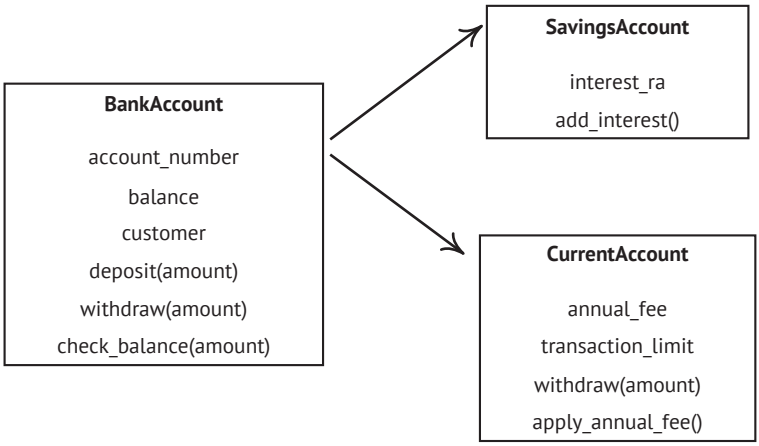


Рис. 4.3. Классы, производные от абстрактного базового класса: `SavingsAccount` и `CurrentAccount` наследуют методы и атрибуты от класса `BankAccount`, но вместе с тем специализируют и расширяют свою функциональность

4.6.2 Определение и использование классов в Python

Класс определяется с использованием ключевого слова `class` и со сдвигом вправо в теле класса инструкций (атрибутов и методов) в блоке, следующем за строкой объявления класса. Существует соглашение, по которому имена классов записываются в стиле CamelCase. Правильный практический прием: за инструкцией объявления `class` следует строка документации docstring, описывающая, что делает (для чего предназначен) этот класс (см. раздел 2.7.1). Методы класса определяются с помощью уже знакомого ключевого слова `def`, но первым аргументом каждого метода должна быть переменная с именем `self`[52] – это имя используется для ссылки на сам объект, когда необходимо вызывать собственные методы, или для обращения к атрибутам, как мы увидим в дальнейшем.

В рассматриваемом здесь примере с банковским счетом базовый класс можно определить, как показано в листинге 4.6.

Листинг 4.6. Определение абстрактного базового класса BankAccount

```
# bank_account.py

class BankAccount:
    """ An abstract base class representing a bank account."""
    # """ Абстрактный базовый класс, представляющий банковский счет."""
    currency = '$'

    def __ init __(self, customer, account_number, balance=0):
    """
    Initialize the BankAccount class with a customer, account number
    and opening balance (which defaults to 0.)
    """
```

[52] В действительности имя переменной может быть любым, но практически всегда используется имя `self`.

```
    # """
    # Инициализация класса BankAccount значениями имени клиента, номера счета и
    # баланса при открытии счета (по умолчанию 0).
    # """

    self.customer = customer
    self.account_number = account_number
    self.balance = balance

def deposit(self, amount):
    """ Deposit amount into the bank account."""
    # """Размер вклада на банковский счет."""
    if amount > 0:
        self.balance += amount
    else:
        print('Invalid deposit amount:', amount)

def withdraw(self, amount):
    """
    Withdraw amount from the bank account, ensuring there are sufficient
    funds.
    """
    # """
    # Сумма средств, снимаемых с банковского счета, при условии достаточной суммы
    # на этом счете.
    # """

    if amount > 0:
        if amount > self.balance:
            print('Insufficient funds')
        else:
            self.balance -= amount
    else:
        print('Invalid withdrawal amount:', amount)
```

Для практического использования этого простого класса можно сохранить исходный код его определения в файле *bank_account.py* и импортировать этот файл в новую программу или в интерактивную командную оболочку Python следующей командой:

```
from bank_account import BankAccount
```

Теперь новая программа может создавать объекты `BankAccount` и работать с ними, вызывая методы, описанные в листинге 4.6.

Создание экземпляра класса

Экземпляр (instance) класса создается с помощью синтаксической конструкции *`object = ClassName(args)`*. Может потребоваться создание экземпляра класса, который должен инициализировать сам себя некоторым способом (возможно, установкой соответствующих значений для атрибутов). Такая инициализация выполняется специализированным методом `__init__()`, который принимает любые аргументы `args`, заданные в инструкции инициализации.

В рассматриваемом здесь примере счет открывается при создании объекта BankAccount с передачей в него имени владельца счета (клиента), номера счета и (необязательно) баланса при открытии счета (если это значение не задано, то по умолчанию присваивается 0):

```
my_account = BankAccount('Joe Bloggs', 21457288)
```

Мы заменим строку customer на объект Customer в примере П4.18.

Методы и атрибуты

Рассматриваемый здесь класс определяет два метода: один для вклада (положительной) денежной суммы, второй для снятия денег (если снимаемая сумма задана как положительное число и не превышает баланс счета).

Класс BankAccount содержит два различных типа атрибутов: self.customer, self.account_number и self.balance – переменные экземпляра класса (instance variables) – они могут содержать различные значения в различных объектах, созданных из класса BankAccount. К другому типу атрибутов относится переменная currency – это переменная класса (class variable): она определена внутри класса, но вне какого-либо из его методов, поэтому совместно используется всеми экземплярами этого класса.

Доступ к атрибутам и методам осуществляется с использованием нотации *object.attr*. Например:

```
>>> my_account.account_number         # Доступ к атрибуту класса my_account.
21457288
>>> my_account.deposit(64)            # Вызов метода класса my_account.
>>> my_account.balance
64
```

Добавим третий метод для вывода баланса счета. Этот метод обязательно должен быть определен внутри блока кода класса:

```
    def check_balance(self):
        """ Print a statement of the account balance. """
        # """ Вывод состояния баланса счета. """
        print('The balance of account number {:d} is {:s}{:.2f}'
              .format(self.account_number , self.currency , self.balance))
```

Пример П4.18. Теперь определим класс Customer, описанный в диаграмме классов на рис. 4.2: экземпляр этого класса будет атрибутом customer в классе BankAccount. Обратите внимание: была возможность создать экземпляр класса BankAccount, передавая в атрибут customer строковый литерал. Это следствие динамической типизации языка Python: проверка типа не выполняется автоматически при передаче объекта как аргумента в конструктор класса, поэтому можно передавать объект любого корректного типа.

Следующий код определяет класс Customer, его нужно сохранить в файле с именем *customer.py*:

```
from datetime import datetime
class Customer:
    """ A class representing a bank customer. """
    # """ Класс, представляющий клиента банка. """
```

```
    def __ init __(self, name, address, date_of_birth):
        self.name = name
        self.address = address
        self.date_of_birth = datetime.strptime(date_of_birth, '%Y-%m-%d')
        self.password = '1234'

    def get_age(self):
        """ Calculates and returns the customer' s age. """
        # """ Вычисляет и возвращает возраст клиента. """
        today = datetime.today()
        try:
            birthday = self.date_of_birth.replace(year=today.year)
        except ValueError:
            # День рождения 29 февраля, но текущий год не является високосным.
            birthday = self.date_of_birth.replace(year=today.year, day=self.date_of_birth.
day - 1)
        if birthday > today:
            return today.year - self.date_of_birth.year - 1
        return today.year - self.date_of_birth.year
```

Теперь можно передавать объекты `Customer` в конструктор `BankAccount`:

```
>>> from bank_account import BankAccount
>>> from customer import Customer
>>>
>>> customer1 = Customer('Helen Smith', '76 The Warren , Blandings , Sussex', '1976-02-29')
>>> account1 = BankAccount(customer1 , 21457288, 1000)
>>> account1.customer.get_age()
39
>>> print(account1.customer.address)
76 The Warren , Blandings , Sussex
```

4.6.3 Наследование класса в языке Python

Производный класс можно создать наследованием от одного или нескольких базовых классов с использованием следующей синтаксической конструкции:

```
class SubClass(BaseClass1, BaseClass2, ...):
```

Определим два производных класса (или подкласса), показанных на рис. 4.3, от базового класса `BankAccount`. Их можно определить в том же файле, в котором определен класс `BankAccount`, или в другом Python-файле, где импортируется `BankAccount`.

```
class SavingsAccount(BankAccount):
    """ A class representing a savings account. """
    # """ Класс, представляющий накопительный счет. """

    def __ init __(self, customer, account_number, interest_rate, balance=0):
        """ Initialize the savings account. """
        # """ Инициализация накопительного счета. """
        self.interest_rate = interest_rate                                    ❶
        super().__ init __(customer, account_number, balance)                 ❷
```

```
    def add_interest(self):
        """ Add interest to the account at the rate self.interest_rate. """
        # """ Добавление процентной ставки к сумме счета с коэффициентом self.interest_rate. """

        self.balance *= (1. + self.interest_rate / 100)
```

❶ Класс `SavingsAccount` добавляет новый атрибут `interest_rate` и новый метод `add_interest` к своему базовому классу, а также замещает метод `__init__`, чтобы обеспечить инициализацию атрибута `interest_rate` при создании экземпляра класса.

❷ Обратите внимание: новый метод `__init__` вызывает метод `__init___` базового класса, чтобы инициализировать другие атрибуты: встроенная функция `super()` позволяет ссылаться на родительский базовый класс[53]. Новый класс `SavingsAccount` можно использовать следующим образом:

```
>>> my_savings = SavingsAccount('Matthew Walsh', 41522887, 5.5, 1000)
>>> my_savings.check_balance()
The balance of account number 41522887 is $1000
>>> my_savings.add_interest()
>>> my_savings.check_balance()
The balance of account number 41522887 is $1055.00
```

Второй подкласс `CurrentAccount` имеет похожую структуру:

```
class CurrentAccount(BankAccount):
    """ A class representing a current (checking) account. """
    # """ Класс, представляющий текущий (чековый) счет. """
    def __ init __(self, customer, account_number, annual_fee, transaction_limit,
balance=0):
        """ Initialize the current account. """
        # """ Инициализация текущего счета. """
        self.annual_fee = annual_fee
        self.transaction_limit = transaction_limit
        super().__ init __(customer, account_number, balance)

    def withdraw(self, amount):
        """
        Withdraw amount if sufficient funds exist in the account and amount
        is less than the single transaction limit.
        """
        # """
        # Снятие денежной суммы, если на счете имеется достаточно средств и снимаемая сумма
        # меньше лимита, установленного для одной транзакции.
        # """
        if amount <= 0:
            print('Invalid withdrawal amount:', amount)
            return

        if amount > self.balance:
            print('Insufficient funds')
            return
```

[53] Встроенная функция `super()`, вызываемая таким способом, создает промежуточный (proxy) объект (объект-посредник), который делегирует вызовы методов в родительский базовый класс (в данном случае в класс `BankAccount`).

```
        if amount > self.transaction_limit:
            print('{0:s}{1:.2f} exceeds the single transaction limit of'
                  ' {0:s}{2:.2f}'.format(self.currency, amount, self.transaction_limit))
            return

        self.balance -= amount

    def apply_annual_fee(self):
        """ Deduct the annual fee from the account balance. """
        # """ Удержание ежегодной оплаты с баланса счета. """

        self.balance = max(0., self.balance - self.annual_fee)
```

Обратите внимание на то, что происходит, если вызвать метод `withdraw` в конкретном созданном объекте `CurrentAccount`:

```
>>> my_current = CurrentAccount('Alison Wicks', 78300991, 20., 200.)
>>> my_current.withdraw(220)
Insufficient Funds

>>> my_current.deposit(750)
>>> my_current.check_balance()
The balance of account number 78300991 is $750.00

>>> my_current.withdraw(220)
$220.00 exceeds the transaction limit of $200.00
```

Вызывается именно тот метод `withdraw`, который определен в классе `CurrentAccount`, так как он замещает метод с тем же именем из базового класса `BankAccount`.

Пример П4.19. Простая модель полимера в растворе интерпретирует его как последовательность случайно ориентированных сегментов, т. е. не существует связи между ориентацией какого-либо сегмента и любого другого сегмента (это так называемая модель случайного перемещения (случайного блуждания) в пространстве).

Определим класс `Polymer` для описания такого полимера. В этом классе позиции сегментов хранятся в списке кортежей `(x, y, z)`. Объект `Polymer` будет инициализироваться значениями `N` и `a` – количеством сегментов и длиной сегмента соответственно. Метод инициализации вызывает метод `make_polymer` для заполнения списка позиций сегментов.

Объект `Polymer` также вычисляет общее расстояние между противоположными концами полимера и реализует метод `calc_Rg` для вычисления и возврата радиуса инерции полимера, определяемый по формуле

$$R_g = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(r_i - r_{CM})^2}.$$

Листинг 4.7. Класс Polymer

```
# polymer.py

import math
import random

class Polymer:
    """ A class representing a random-flight polymer in solution. """
    # """ Класс, представляющий модель случайного перемещения полимера в растворе. """

    def __ init __(self , N, a):
        """
        Initialize a Polymer object with N segments, each of length a.
        """
        # """
        # Инициализация объекта Polymer с N сегментами и длиной каждого сегмента a.
        # """

        self.N, self.a = N, a
        # Список self.xyz содержит векторы позиций сегментов как кортежи.
        self.xyz = [(None , None , None)] * N
        # Вектор полного полимера (от одного конца до другого).
        self.R = None
        # Создание полимера посредством присваивания позиций сегментов.
        self.make_polymer()

    def make_polymer(self):
        """
        Calculate the segment positions, center of mass and end-to-end
        distance for a random-flight polymer.
        """
        # """
        # Вычисление позиций сегментов, центра масс и общего расстояния между противоположными
        # концами модели случайного перемещения полимера в растворе.
        # """

        # Определение исходной позиции полимера в начале координат (0, 0, 0).
        self.xyz[0] = x, y, z = cx, cy, cz = 0, 0, 0
        for i in range(1, self.N):
            # Выбор случайной ориентации для следующего сегмента.                    ❶
            theta = math.acos(2 * random.random() - 1)
            phi = random.random() * 2. * math.pi
            # Добавление соответствующего вектора смещения для этого сегмента.
            x += self.a * math.sin(theta) * math.cos(phi)
            y += self.a * math.sin(theta) * math.sin(phi)
            z += self.a * math.cos(theta)
            # Сохранение позиции сегмента и обновление суммарного значения центра масс.
            self.xyz[i] = x, y, z
            cx, cy, cz = cx + x, cy + y, cz + z
        # Вычисление положения центра масс.                                          ❷
        cx, cy, cz = cx / self.N, cy / self.N, cz / self.N
        # Вектор полного полимера - это позиция последнего сегмента,
        # так как начало полимера совпадает с началом координат.
        self.R = x, y, z
```

```
        # Итоговая корректировка центра полимера по новому центру масс.
        for i in range(self.N):
           self.xyz[i] = (self.xyz[i][0] - cx,
                          self.xyz[i][1] - cy,
                          self.xyz[i][2] - cz)

    def calc_Rg(self):
        """
        Calculates and returns the radius of gyration, Rg. The polymer
        segment positions are already given relative to the center of
        mass, so this is just the rms position of the segments.
        """
        # """
        # Вычисляет и возвращает радиус инерции полимера Rg.
        # Позиции сегментов полимера уже заданы относительно центра масс,
        # поэтому это просто среднеквадратичное значение позиций сегментов.
        # """

        self.Rg = 0.
        for x, y, z in self.xyz:
            self.Rg += x**2 + y**2 + z**2
        self.Rg = math.sqrt(self.Rg / self.N)
        return self.Rg
```

❶ Один из способов выбора положения следующего сегмента заключается в выборе случайной точки на поверхности единичной сферы и использовании соответствующей пары углов в системе сферических полярных координат θ и ϕ (где $0 \leqslant \theta < \pi$ и $0 \leqslant \phi < 2\pi$) для вычисления смещения от позиции предыдущего сегмента по следующим формулам:

- $\Delta x = a \sin\theta \cos\phi$;
- $\Delta y = a \sin\theta \sin\phi$;
- $\Delta z = a \cos\theta$.

❷ Вычисляется позиция центра масс полимера $\mathbf{r}_{CM}$, затем сдвигается начало координат сегмента полимера так, чтобы расстояния измерялись относительно этой точки (т. е. координаты сегмента определяются по отношению к центру масс полимера).

Теперь можно протестировать класс `Polymer`, импортировав его в командную оболочку Python:

```
>>> from polymer import Polymer
>>> polymer = Polymer(1000, 0.5)        # Полимер с 1000 сегментов длиной 0.5.
>>> polymer.R                           # Общий вектор полимера.
(5.631332375722011, 9.408046667059947, -1.3047608473668109)
>>> polymer.calc_Rg()                   # Радиус инерции полимера.
5.183761585363432
```

А теперь сравним распределение расстояний между противоположными концами полимера с теоретически прогнозируемой функцией плотности вероятности:

$$P(R) = 4\pi R^2 \left(\frac{3}{2\pi \left\langle r^2 \right\rangle} \right)^{3/2} \exp\left(-\frac{3R^2}{2\left\langle r^2 \right\rangle} \right),$$

где среднеквадратичная позиция сегментов равна $\langle r^2 \rangle = Na^2$.

Листинг 4.8. Распределение полимеров, созданных по модели случайного перемещения

```
# eg4-c-ii-polymer -a.py
# Сравнение наблюдаемого распределения расстояний между противоположными концами
# полимера для Np-модели случайного перемещения с прогнозируемой функцией распределения
# вероятностей.

import matplotlib.pyplot as plt
from polymer import Polymer
pi = plt.pi

# Вычисление расстояния R для Np-модели полимеров.
Np = 3000
# Каждый полимер состоит из N сегментов длиной a.
N, a = 1000, 1.
R = [None] * Np
for i in range(Np):
    polymer = Polymer(N, a)
    Rx, Ry, Rz = polymer.R
    R[i] = plt.sqrt(Rx**2 + Ry**2 + Rz**2)
    # Вывод индикатора выполнения процесса для каждых 100 полимеров.
    if not (i+1) % 100:
        print(i+1, '/', Np)

# Вывод распределения расстояний Rx как нормализованной гистограммы
# с использованием 50 полос.
plt.hist(R, 50, normed=1)

# Вывод теоретического распределения вероятностей Pr как функции от r.
r = plt.linspace(0,200,1000)
msr = N * a**2
Pr = 4.*pi*r**2 * (2 * pi * msr / 3)**-1.5 * plt.exp(-3*r**2 / 2 / msr)
plt.plot(r, Pr, lw=2, c='r')
plt.xlabel('R')
plt.ylabel('P(R)')
plt.show()
```

Эта программа выводит график, который обычно выглядит так, как показано на рис. 4.4, и демонстрирует соответствие теории.

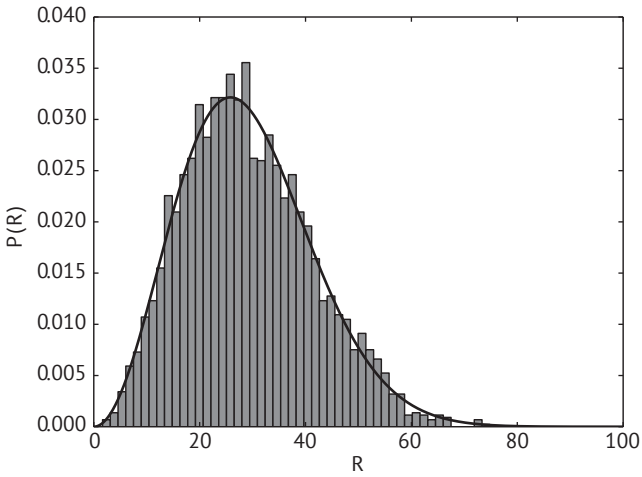


Рис. 4.4. Распределение расстояний *R* между противоположными концами полимера для модели случайных перемещений при $N = 1000$, $a = 1$

4.6.4 Классы и операторы

Операторы (такие как +, * и <=) и встроенные функции, такие как `len` и `abs`, работают с объектами Python, вызывая специализированные методы этих объектов. Имена таких специализированных методов начинаются и заканчиваются двумя символами подчеркивания __ (поэтому подобные методы часто называют dunder-методами – от **d**ouble **under**score – двойное подчеркивание). Для реализации («перегрузки») их функциональности в пользовательских классах просто определяются методы с такими же именами. Полный список специализированных методов можно найти в документации по языку Python (https://docs.python.org/3/reference/datamodel.html), но в табл. 4.8 представлен список наиболее часто используемых специализированных методов. Например, для вычисления выражения `x + y` вызывается специализированный метод `x.__add__(y)`.

Таблица 4.8. Часто используемые специализированные методы Python

Метод	Описание	Пример
`__add__`	+, сложение	`x + y`
`__sub__`	-, вычитание	`x - y`
`__mul__`	*, умножение	`x * y`
`__truediv__`	/, «натуральное» деление	`x / y`
`__floordiv__`	//, деление по модулю	`x // y`
`__mod__`	%, получение остатка (от деления)	`x % y`
`__pow__`	**, возведение в степень	`x ** y`
`__neg__`	Смена знака (унарный минус)	`-x`
`_matmul__`	@, умножение матриц	`x @ y`
`__abs__`	Абсолютное значение	`abs(x)`

Окончание табл. 4.8

Метод	Описание	Пример
`__contains__`	Принадлежность	`y in x`
`__lt__`	Меньше	`y < x`
`__le__`	Меньше или равно	`y <= x`
`__eq__`	Равно	`y == x`
`__ne__`	Не равно[54]	`y != x`
`__gt__`	Больше	`y > x`
`__ge__`	Больше или равно	`y >= x`
`__str__`	Представление строки, удобной для чтения человеком	`str(x)`
`__repr__`	Недвусмысленное (однозначное) представление строки	`repr(x)`

Python поддерживает полиморфизм, и могут возникать условия, при которых `x` и `y` являются значениями различных типов. Если для объекта `x` не реализован требуемый метод, то Python будет искать «рефлексивную» (отраженную) версию в объекте `y`. Таким образом, выражение `'a' * 4` вызывает специализированный метод `'a'.__mul__(4)` в строковом объекте `'a'`; выражение `4 * 'a'` сначала пытается вызвать `4.__mul__('a')`, а когда попытка завершается неудачно (объекты `int` не знают, как умножать число на строку `str`), затем пытается вызвать рефлексивную (отраженную) версию `'a'.__rmul__(4)`, которая возвращает строку `'aaaa'`: объекты типа `str` знают, как выполнить умножение на целое число (`int`).

Специализированные методы `__str__` и `__repr__` заслуживают отдельного описания. Оба возвращают представление объекта в виде строки, но если от `__str__` ожидается возврат строки, удобной для чтения человеком, то целью `__repr__` является по возможности недвусмысленное (однозначное) представление объекта. В зависимости от класса возможен естественный выбор возвращаемого значения `__str__`, которое передает самые важные свойства экземпляра, тогда как возвращаемое значение `__repr__` должно быть достаточно полным, чтобы содержащуюся в нем информацию можно было бы использовать для отладки или для создания идентичного экземпляра. Следует отметить, что если для объекта `obj` определен метод `__repr__`, но не определен метод `__str__`, то вызов `str(obj)` возвращает `obj.__repr__()`. Класс всегда должен определять метод `__repr__()` и дополнительно (но не обязательно) определять метод `__str__()`, если требуется еще и представление строки, легко воспринимаемое человеком.

Пример П4.20. Несмотря на то что библиотека NumPy (см. главу 6) предлагает более эффективный вариант, написание исходного кода класса для векторов только средствами Python остается хорошим обучающим примером. Код в листинге 4.9 определяет класс Vector2D и тестирует его на различных операциях.

---

[54] Если метод не реализован явно, то `__ne__` вызывает метод `__eq__` и инвертирует результат.

Листинг 4.9. Простой класс, представляющий двумерный вектор в декартовых координатах

```
import math

class Vector2D:
    """A two-dimensional vector with Cartesian coordinates."""
    # """Двумерный вектор в декартовых координатах. """

    def __init__(self, x, y):
        self.x, self.y = x, y

    def __str__(self):
        """Human -readable string representation of the vector."""
        # """Строка представления вектора, удобная для чтения человеком. """
        return '{:g}i + {:g}j'.format(self.x, self.y)

    def __repr__(self):
        """Unambiguous string representation of the vector."""
        # """Строка недвусмысленного представления вектора. """
        return repr((self.x, self.y))

    def dot(self, other):
        """The scalar (dot) product of self and other. Both must be vectors."""
        # """Скалярное произведение самого объекта и другого объекта.
        #    Оба должны быть векторами. """
        if not isinstance(other, Vector2D):                                    ❶
            raise TypeError('Can only take dot product of two Vector2D objects')
        return self.x * other.x + self.y * other.y

    # Псевдоним для метода __matmul__ - dot, чтобы можно было использовать a @ b,
    # а также a.dot(b).
    __matmul__ = dot

    def __sub__(self, other):
        """Vector subtraction."""
        # """Вычитание векторов. """
        return Vector2D(self.x - other.x, self.y - other.y)

    def __add__(self, other):
        """Vector addition."""
        # """Сложение векторов. """
        return Vector2D(self.x + other.x, self.y + other.y)

    def __mul__(self, scalar):
        """Multiplication of a vector by a scalar."""
        # """Умножение вектора на скаляр. """
        if isinstance(scalar, int) or isinstance(scalar, float):               ❷
            return Vector2D(self.x*scalar, self.y*scalar)
        raise NotImplementedError('Can only multiply Vector2D by a scalar')

    def __rmul__(self, scalar):
        """Reflected multiplication so vector * scalar also works."""
        # """Рефлексивное умножение, чтобы можно было вычислить выражение vector * scalar."""
        return self.__mul__(scalar)

    def __neg__(self):
        """Negation of the vector (invert through origin.)"""
        # """Изменение знака вектора (инвертирование относительно начала координат). """
        return Vector2D(-self.x, -self.y)
```

```
    def __truediv__(self , scalar):
        """True division of the vector by a scalar."""
        # """Обычное деление вектора на скаляр. """
        return Vector2D(self.x / scalar, self.y / scalar)

    def __mod__(self, scalar):
        """One way to implement modulus operation: for each component."""
        # """Один из способов реализации операции получения остатка от деления: для каждого
        #    компонента. """
        return Vector2D(self.x % scalar, self.y % scalar)

    def __abs__(self):
        """Absolute value (magnitude) of the vector."""
        # """Абсолютное значение (величина) вектора. """
        return math.sqrt(self.x**2 + self.y**2)

    def distance_to(self , other):
        """The distance between vectors self and other."""
        # """Расстояние между этим и другим векторами. """
        return abs(self - other)

    def to_polar(self):
        """Return the vector's components in polar coordinates."""
        # """Возврат компонентов вектора в полярных координатах. """
        return self.__abs__(), math.atan2(self.y, self.x)

if __name__ == '__main__':                                              ❸
    v1 = Vector2D(2, 5/3)
    v2 = Vector2D(3, -1.5)
    print('v1 = ', v1)
    print('repr(v2) = ', repr(v2))
    print('v1 + v2 = ', v1 + v2)
    print('v1 - v2 = ', v1 - v2)
    print('abs(v2 - v1) = ', abs(v2 - v1))
    print('-v2 = ', -v2)
    print('v1 * 3 = ', v1 * 3)
    print('7 * v2 = ', 7 * v1)
    print('v2 / 2.5 = ', v2 / 2.5)
    print('v1 % 1 = ', v1 % 1)
    print('v1.dot(v2) = v1 @ v2 = ', v1 @ v2)
    print('v1.distance_to(v2) = ',v1.distance_to(v2))
    print('v1 as polar vector , (r, theta) =', v1.to_polar())
```

❶ Генерируется исключение, если оба операнда для скалярного произведения не являются векторами.

❷ Разрешается только произведение вектора на скалярное значение, но не поддерживаются оба варианта умножения: a**v** и **v**a.

❸ Код в этом блоке выполняется только в том случае, если выполняется как главная (main) программа. В этом случае Python должен установить для переменной `__name__` жестко закодированную строку `'__main__'`. Если файл интерпретируется как модуль и импортируется (например, командой `from vector2d import Vector2D`), то этот блок игнорируется.

При выполнении программы выводится следующий результат:

```
v1 = 2i + 1.66667j
repr(v2) = (3, -1.5)
v1 + v2 = 5i + 0.166667j
v1 - v2 = -1i + 3.16667j
abs(v2 - v1) = 3.3208098075285464
-v2 = -3i + 1.5j
v1 * 3 = 6i + 5j
7 * v2 = 14i + 11.6667j
v2 / 2.5 = 1.2i + -0.6j
v1 % 1 = 0i + 0.666667j
v1.dot(v2) = v1 @ v2 = 3.5
v1.distance_to(v2) = 3.3208098075285464
v1 as polar vector, (r, theta) = (2.6034165586355518, 0.6947382761967033)
```

Пример П4.21. Код в листинге 4.10 использует определенный выше (в листинге 4.9) класс `Vector2D` для реализации простой имитации молекулярной динамики сферических частиц с равными массами, перемещающихся в двух измерениях. Изначально все частицы имеют одинаковую скорость, столкновения регулируют скорости в соответствии с распределением Максвелла–Больцмана, как показано на рис. 4.5. На веб-сайте книги находится расширенный код для анимации этой имитации (https://scipython.com/eg/baa). Примечание: несмотря на изящество, принятый здесь объектно-ориентированный подход не является самым быстрым: этому препятствуют издержки на создание экземпляров многих объектов. Издержки становятся значительными, когда необходимо рассматривать множество частиц и столкновений в каждом отрезке времени. Самым быстрым является подход с использованием библиотеки NumPy, см. ссылки на указанной веб-странице.

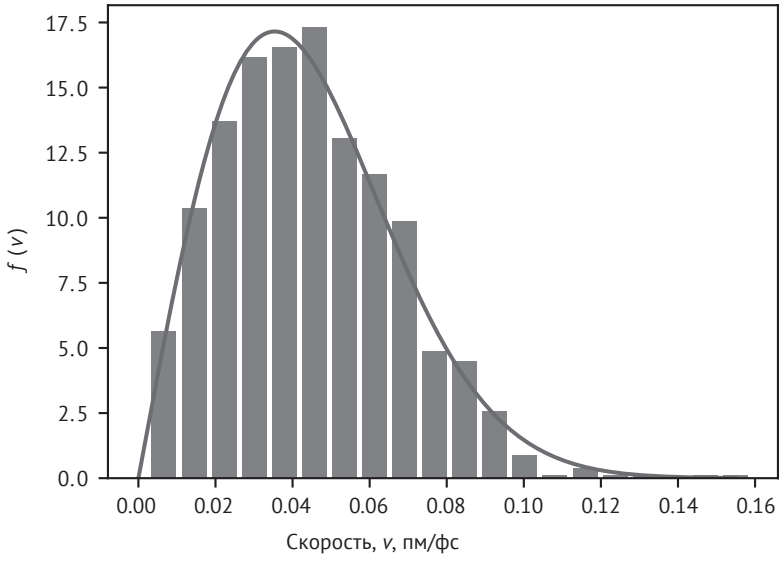


Рис. 4.5. Распределение скоростей частиц после выравнивания по статистике Максвелла–Больцмана при многочисленных столкновениях

Листинг 4.10. Простая двумерная имитация молекулярной динамики

```
import math
import random
import matplotlib.pyplot as plt
from vector2d import Vector2D

class Particle:
    """A circular particle of unit mass with position and velocity."""
    # """Сферическая частица единичной массы с заданным положением и скоростью."""

    def __init__(self, x, y, vx, vy, radius=0.01):
        self.pos = Vector2D(x, y)
        self.vel = Vector2D(vx, vy)
        self.radius = radius

    def advance(self, dt):
        """Advance the particle's position according to its velocity."""
        # """Изменение положения частицы в соответствии с ее скоростью."""

        # Используются циклические граничные условия: частица, пересекающая границу
        # области 0 <= x < 1, 0 <= y < 1, появляется на противоположной границе.
        self.pos = (self.pos + self.vel * dt) % 1

    def distance_to(self, other):
        """Return the distance from this Particle to other Particle."""
        # """Возвращает расстояние от частицы до другой частицы."""
        return self.pos.distance_to(other.pos)

    def get_speed(self):
        """Return the speed of the Particle from its velocity."""
        # """Возвращает (абсолютную) величину скорости частицы по текущей скорости движения."""
        return abs(self.vel)

class Simulation:
    """A simple simulation of circular particles in motion."""
    # """Простая имитация движения сферических частиц."""

    def __init__(self, nparticles=100, radius=0.01, v0=0.05):
        self.nparticles = nparticles
        self.radius = radius
        # Случайная инициализация положений и направлений скоростей частиц.
        self.particles = [self.init_particle(v0) for i in range(nparticles)]
        self.t = 0

    def init_particle(self, v0=0.05):
        """Return a new Particle object with random position and velocity.
           The position is chosen uniformly from 0 <= x < 1, 0 <= y < 1;
           The velocity has fixed magnitude, v0, but random direction.
        """
        # """Возвращает новый объект Particle со случайным положением и скоростью.
        #    Положение выбирается единообразно из диапазона значений 0 <= x < 1, 0 <= y < 1.
        #    Скорость имеет фиксированную величину v0, но случайно выбранное направление.
        # """

        x, y = random.random(), random.random()
        theta = 2*math.pi * random.random()
        self.v0 = v0
        vx, vy = self.v0 * math.cos(theta), self.v0 * math.sin(theta)
        return Particle(x, y, vx, vy, self.radius)
```

```
def advance(self, dt):
    """Advance the Simulation by dt in time , handling collisions."""
    # """Изменение объекта Simulation на dt по времени с обработкой столкновений."""

    self.t += dt
    for particle in self.particles:
        particle.advance(dt)

   # Найти все различные пары частиц, находящиеся в состоянии столкновения в текущий момент.
    colliding_pair = []
    for i in range(self.nparticles):
        pi = self.particles[i]
        for j in range(i+1, self.nparticles):
            pj = self.particles[j]
           # Частица pi сталкивается с частицей pj, если разделяющее их расстояние меньше,
            # чем их удвоенный радиус.
            if pi.distance_to(pj) < 2 * self.radius:
                colliding_pair.append((i, j))

    print('ncollisions =', len(colliding_pair))
    # Для каждой пары частиц скорости изменяются в соответствии с законами динамики
    # упругого столкновения частиц.
    for i,j in colliding_pair:
        p1, p2 = self.particles[i], self.particles[j]
        r1, r2 = p1.pos, p2.pos
        v1, v2 = p1.vel, p2.vel
        dr, dv = r2 - r1, v2 - v1
        dv_dot_dr = dv.dot(dr)
        d = r1.distance_to(r2)**2
        p1.vel = v1 - dv_dot_dr / d * (r1 - r2)
        p2.vel = v2 - dv_dot_dr / d * (r2 - r1)

if __name__ == '__main__':
    import numpy as np

    sim = Simulation(nparticles=1000, radius=0.005, v0=0.05)
    dt = 0.02

    nit = 500
    dnit = nit // 10
    for i in range(nit):
        if not i % dnit:
            print(f'{i}/{nit}')
        sim.advance(dt)

    # Создание гистограммы скоростей частиц.
    nbins = sim.nparticles // 50
    hist, bins, _ = plt.hist([p.get_speed() for p in sim.particles], nbins, density=True)
    v = (bins[1:] + bins[:-1])/2

    # Средняя кинетическая энергия каждой частицы.
    KE = sim.v0**2 / 2

    # Распределение Максвелла-Больцмана для скоростей в условиях термодинамического
      равновесия.
    a = 1 / 2 / KE
    f = 2*a * v * np.exp(-a*v**2)
    plt.plot(v, f)

    plt.show()
```

4.6.5 Упражнения

Задачи

34.6.1.

а) Изменить базовый класс `BankAccount` для проверки номера счета, передаваемого в конструктор класса `__init__` на соответствие алгоритму Луна, описанному в задаче 32.5.3.

б) Изменить класс `CurrentAccount` для реализации свободного овердрафта (перерасхода средств). Лимит должен быть установлен в конструкторе `__init__`, снятие денег разрешается только в пределах установленного лимита.

34.6.2. Добавить метод `save_svg` в класс `Polymer` из примера П4.19 для сохранения изображения полученного полимера в файле формата SVG. Шаблон для работы с SVG-файлом см. в задаче 34.4.3.

34.6.3. Написать программу для создания изображения созвездия с использованием данных из каталога звезд Yale Bright Star Catalog (http://tdc-www.harvard.edu/catalogs/bsc5.html).

Создать класс `Star` для представления звезды с атрибутами: название, величина и положение на небе. Для получения этих атрибутов выполнить парсинг файла *bsc5.dat*, являющегося частью каталога. В классе `Star` реализовать метод, преобразующий положение звезды на небесной сфере, представленное в форме (прямое восхождение: α, склонение: δ), в точку на плоскости (x, y), например, с использованием ортографической (прямоугольной) проекции относительно центральной точки (α_0, δ_0):

$$\Delta\alpha = \alpha - \alpha_0$$

$$x = \cos\delta \sin\Delta\alpha$$

$$y = \sin\delta \cos\delta_0 - \cos\delta \cos\Delta\alpha \sin\delta_0.$$

Спроецированные и промасштабированные соответствующим образом положения звезд можно вывести в изображении в формате SVG как объекты `circle` (чем ярче звезда, тем больше радиус). Например, строка

```
<circle cx="200" cy="150" r="5" stroke="none" fill="#ffffff"/>
```

представляет белый кружок радиусом 5 пикселов и с координатами центра на холсте (canvas) (200, 150).

Совет: необходимо преобразовать прямое восхождение из формата (ч, мин, с) и склонение из формата (град, мин, с) в радианы. Использовать данные, соответствующие «equinox J2000, epoch 2000.0» («точка равноденствия J2000, эпоха равноденствия каталога 2000.0») в каждой строке файла *bsc5.dat*. Позволить пользователю выбирать созвездие из командной строки с использованием трехбуквенной аббревиатуры (например, Ori для Orion), которая ука-

зана как часть названия звезды в каталоге. Не следует забывать, что звездные величины меньше для более ярких звезд. При использовании предложенной выше ортографической проекции следует выбрать (α_0, δ_0) как среднее значение (α, δ) для звезд в созвездии.

34.6.4. Спроектировать и реализовать класс `Experiment` для считывания и сохранения простых последовательностей данных (x, y) как массивов NumPy из текстового файла. Включить в класс методы для преобразования этих последовательностей данных с помощью некоторой простой функции (например, $x' = \ln x$, $y' = 1/y$) и для выполнения линейной регрессии по методу наименьших квадратов с преобразованными данными (с возвратом коэффициента угла наклона и точки пересечения с осью координат линии самой точной подгонки $y'_{\text{fit}} = mx' + c$). Библиотека NumPy предоставляет методы для выполнения линейной регрессии (см. раздел 6.5.3), но для решения этой задачи можно напрямую реализовать следующие формулы:

$$m = \frac{\overline{xy} - \overline{x}\,\overline{y}}{\overline{x^2} - \overline{x}^2},$$
$$c = \overline{y} - m\overline{x},$$

где форма записи с надчеркиванием $\bar{\cdot}$ обозначает среднее арифметическое значение переменной. (Совет: необходимо использовать метод `np.mean(arr)` для возврата среднего значения массива `arr`.)

Хлоруксусная кислота – важный компонент для промышленного синтеза в фармакологии, производстве пестицидов и топлива. При высокой концентрации и при сильнощелочных условиях гидролиз этой кислоты можно рассматривать как следующую реакцию:

$$ClCH_2COO^- + OH^- \rightleftharpoons HOCH_2COO^- + Cl^-.$$

Данные о концентрации $ClCH_2COO^-$ c (в М) как функции от времени t (в с) для этой реакции представлены как избыток щелочи при пяти различных температурах в файлах данных *caa-T.txt* (`T = 40`, `50`, `60`, `70`, `80` в °C): эти данные можно получить здесь: https://scipython.com/ex/bde. Описанная реакция известна как реакция второго порядка, поэтому подчиняется закону комплексного нормирования:

$$1/c = 1/c_0 + kt,$$

где k – константа фактической скорости, c_0 – начальная (при $t = 0$) концентрация хлоруксусной кислоты.

Использовать созданный класс `Experiment` для интерпретации описанных выше данных с помощью линейной регрессии $1/c$ относительно t, определяя $m(\equiv k)$ для каждой температуры. Затем для каждого значения k определить энергию активизации реакции с помощью второй линейной регрессии по $\ln k$ относительно $1/T$ в соответствии с законом Аррениуса:

$$k = Ae^{-Ea/RT} \Rightarrow \ln k = \ln A - E_a/RT,$$

где R = 8.314 Дж/К·моль – (универсальная) газовая постоянная.

Примечание: температура должна быть задана в кельвинах (К).

34.6.5. Создать новый класс, производный от класса объекта списка `list`. Новый класс должен реализовать индексацию, начинающуюся с единицы, вместо индексации с нуля. Выполнить перегрузку всех необходимых специализированных методов, перечисленных в документации https://docs.python.org/3/reference/datamodel.html, и написать тесты для проверки корректности созданного кода класса.

Глава 5

Командная оболочка IPython и блокнотная среда Jupyter Notebook

Командная оболочка IPython и связанная с ней виртуальная блокнотная среда на основе браузера Jupyter Notebook представляют собой взаимосвязанные мощные интерфейсы для языка Python. IPython обладает некоторыми преимуществами по сравнению с собственной командной оболочкой языка Python, включая упрощенное взаимодействие с операционной системой, средством интроспекции и автозавершением ключевых слов с помощью клавиши **Tab**. Виртуальная блокнотная среда Jupyter Notebook (ранее IPython Notebook) постоянно адаптируется для использования научными работниками для совместного применения данных и исходного кода, создаваемого для научного анализа в стандартизованном стиле, обеспечивающем повышение воспроизводимости (многократного использования) и степени визуализации. Используемой по умолчанию средой выполнения («ядром») является Python, но среду выполнения можно сконфигурировать для работы с любым из нескольких десятков поддерживаемых языков программирования.

5.1 Командная оболочка IPython

5.1.1 Установка IPython

Подробная информация об установке командной оболочки IPython доступна на веб-сайте IPython: https://ipython.org/install.html, а здесь приводится краткая инструкция.

Командная оболочка IPython включена в дистрибутивный комплект Continuum Anaconda Python. Для обновления до текущей версии внутри Anaconda используйте менеджер пакетов `conda`:

```
conda update conda
conda update ipython
```

Если в системе уже установлен Python, то существует несколько других возможных вариантов установки. Если установлен менеджер пакетов `pip`:

```
pip install ipython
```

Кроме того, возможна загрузка вручную самой последней версии IPython из репозитория GitHub: https://github.com/ipython/ipython/releases – с последующей компиляцией и установкой из каталога исходного кода самого верхнего уровня командой

```
python setup.py install
```

5.1.2 Использование командной оболочки IPython

Для запуска сеанса интерактивной командной оболочки IPython из командной строки просто введите `ipython`. Вы должны увидеть приветственное сообщение, похожее на приведенное ниже:

```
Python 3.7.3 (default , Mar 27 2019, 16:54:48)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.6.1 -- An enhanced Interactive Python. Type '?' for help.

In [1]:
```

(Подробное содержание этого сообщения зависит от типа установки в системе.) Промпт (приглашение) `In [1]:` указывает, что здесь вы вводите команды Python, и заменяет стандартный промпт >>> собственной командной оболочки Python. Счетчик в квадратных скобках увеличивается после ввода каждой команды или блока кода Python. Например:

```
In [1]: 4 + 5
Out[1]: 9
In [2]: print(1)
1
In [3]: for i in range(4):
    ...: print(i, end='')
    ...:
0123
In [4]:
```

Для выхода из командной оболочки IPython введите команду `quit` или `exit`. В отличие от собственной командной оболочки Python, здесь не требуются круглые скобки[55].

Команды вывода справочной информации

Как указано в приветственном сообщении, существует несколько полезных команд для получения информации об использовании командной оболочки IPython:

[55] Кто-то может посчитать это еще одной причиной для использования IPython.

- при вводе одного символа '?' выводится общий обзор использования основных функциональных возможностей IPython (пробел или f – переход к следующей странице; b – возврат на предыдущую страницу; q – выход из страницы справки);
- %quickref предоставляет краткий справочник по каждой основной команде IPython и о «магических функциях» (см. раздел 5.1.3);
- help() или help(*object*) вызывает встроенную справочную систему языка Python (в интерактивном режиме или для заданного объекта *object*);
- ввод одного символа знака вопроса после имени объекта выводит информацию об этом объекте – см. ниже.

Возможно, чаще всего используемой функцией вывода справочной информации, предоставляемой командной оболочкой IPython, является интроспекция с синтаксисом object?. Например:

```
In [4]: a = [5, 6]
In [5]: a?
Type:        list
String form: [5, 6]
Length:      2
Docstring:
Built-in mutable sequence.
(Встроенная изменяемая последовательность.)

If no argument is given, the constructor creates a new empty list.
(Если не задан аргумент, конструктор создает новый пустой список.)
The argument must be an iterable if specified.
(Если аргумент задан, то он обязательно должен быть итерируемым объектом.)
```

Здесь команда a? выводит подробную информацию об объекте a: его строковое представление (которое могло быть создано, например, командой print(a)), его длину (равнозначно вызову функции len(a)) и строку документации, связанную с классом, к которому относится рассматриваемый экземпляр: так как a – список list, здесь выводится краткое описание условий создания объекта типа list[56].

Синтаксис ? особенно полезен как оперативная справочная информация об аргументах, которые принимает функция или метод. Например:

```
In [6]: import numpy as np
In [7]: np.linspace?

Signature:
np.linspace(
    start,
    stop,
    num=50,
    endpoint=True,
    retstep=False,
    dtype=None,
    axis=0,
)
```

[56] Именно это подразумевается под интроспекцией: Python способен исследовать собственные объекты и предоставлять информацию о них.

```
Docstring:
Return evenly spaced numbers over a specified interval.

Returns `num` evenly spaced samples, calculated over the
interval [`start`, `stop`].

The endpoint of the interval can optionally be excluded.

.. versionchanged:: 1.16.0
    Non-scalar `start` and `stop` are now supported.

Parameters
----------
start : array_like
    The starting value of the sequence.
stop : array_like
    The end value of the sequence, unless `endpoint` is set to False.
    In that case, the sequence consists of all but the last of ``num + 1``
    evenly spaced samples, so that `stop` is excluded. Note that the step
    size changes when `endpoint` is False.
num : int, optional
    Number of samples to generate. Default is 50. Must be non-negative.
endpoint : bool , optional
    If True, `stop` is the last sample. Otherwise, it is not included.
    Default is True.
retstep : bool, optional
    If True, return (`samples`, `step`), where `step` is the spacing
    between samples.
dtype : dtype, optional
    The type of the output array. If `dtype` is not given, infer the data
    type from the other input arguments.

    .. versionadded:: 1.9.0

...
```

Для некоторых объектов синтаксис `object??` возвращает еще более подробную информацию, как, например, локацию в памяти и детали исходного кода.

Автоматическое дополнение ключевых слов с помощью клавиши Tab

Подобно многим средам командных строк в оболочках, IPython поддерживает автоматическое дополнение ключевых слов с помощью клавиши **Tab**: начните вводить имя объекта или ключевое слово, нажмите клавишу **Tab**, и имя/слово будет автоматически завершено, или будет выведен список вариантов, если существует более одной возможности. Например:

```
In [8]: w<TAB>
         while     %who_ls
         with      %whos
         %who      %%writefile
```

Если возобновить ввод до тех пор, пока слово не станет однозначно определяемым (например, добавить буквы `hi`), а затем снова нажать клавишу **Tab**, то будет выполнено автоматическое завершение до ключевого слова `while`. Варианты со знаками процента перед именами – это «магические функции», описанные в разделе 5.1.3.

Хронология команд

Возможно, вы уже использовали функцию хронологии команд в собственной командной оболочке языка Python (нажатие клавиш со стрелками вверх и вниз позволяет проходить по списку команд, ранее введенных в текущем сеансе). IPython сохраняет и введенные команды, и вывод результатов их работы в специальных переменных `In` и `Out` (это в действительности список и словарь соответственно, которые связаны с промптами в начале каждой строки ввода и вывода). Например:

```
In [9]: d = {'C': 'Cador', 'G': 'Galahad', 'T': 'Tristan', 'A': 'Arthur'}
In [10]: for a in 'ACGT':
....:        print(d[a])
....:
Arthur
Cador
Galahad
Tristan
In [11]: d = {'C': 'Cytosine', 'G': 'Guanine', 'T': 'Thymine', 'A': 'Adenine'}
In [12]: In[10]                                                              ❶
Out[12]: "for a in 'ACGT ':\n print(d[a])\n "
In [13]: exec(In[10])                                                        ❷
Adenine
Cytosine
Guanine
Thymine
```

❶ Обратите внимание: `In[10]` просто сохраняет строковую версию команды Python (в данном случае это цикл `for`), которая была введена с индексом 10.

❷ Для действительного выполнения команды (с текущим словарем `d`) необходимо передать ее во встроенную функцию Python `exec` (см. также магическую функцию `%rerun` в разделе 5.1.3).

Существует несколько дополнительных ускоренных приемов: псевдоним `_i`*N* означает то же самое, что `In[`*N*`]`, `_`*N* – то же, что `Out[`*N*`]`, а два самых последних вывода результатов возвращаются в специальных переменных `_` и `__` соответственно.

Для просмотра содержимого хронологии используется магическая функция `%history` или `%hist`. По умолчанию выводятся только ранее введенные команды, но часто более удобно выводить также номера строк, если воспользоваться ключом `-n`:

```
In [14]: %history -n
   1: 4 + 5
   2: print(1)
   3:
for i in range(4):
    print(i)
   4: a = [5, 6]
   5: a?
   6: import numpy as np
   7: np.linspace?
   8: d = {'C': 'Cador', 'G': 'Galahad', 'T': 'Tristan', 'A': 'Arthur'}
  10:
for a in 'ACGT':
    print(d[a])
  11: d = {'C': 'Cytosine', 'G': 'Guanine', 'T': 'Thymine', 'A': 'Adenine'}
  12: In[10]
  13: exec(In[10])
  14: %history -n
```

Для вывода конкретной строки или диапазона строк необходимо указать соответствующий номер или диапазон номеров при вызове функции `%history`:

```
In [15]: %history 4
a = [5, 6]

In [16]: %history -n 2-5
   2: print(1)
   3:
for i in range(4):
    print(i)
   4: a = [5, 6]
   5: a?

In [17]: %history -n 1-3 7 12-14
   1: 4 + 5
   2: print(1)
   3:
for i in range(4):
    print(i)
   7: np.linspace?
  12: In[10]
  13: exec(In[10])
  14: %history -n
```

Этот синтаксис используется и некоторыми другими магическими функциями IPython (см. следующий раздел). Кроме того, функция `%history` может принимать дополнительный ключ: `-o` показывает строки вывода вместе со строками ввода.

При нажатии клавиш **Ctrl**+**R** появляется промпт, выглядящий слегка загадочно:

```
I-search backward:
```

Здесь можно выполнить поиск в хронологии команд[57].

[57] Эта функция, возможно, знакома пользователям командной оболочки bash (в Unix-подобных системах) как `(reverse-i-search)'':`.

Взаимодействие с операционной системой

IPython упрощает выполнение команд операционной системы, в которой работает сеанс командной оболочки: любая команда с префиксом в виде символа восклицательного знака ! передается в командную строку операционной системы (в «системную командную оболочку»), а не выполняется как команда языка Python. Например, можно удалять файлы, выводить содержимое каталогов и даже выполнять другие программы и скрипты:

```
In [11]: !pwd                # Возвращает текущий рабочий каталог.
/Users/christian/research
In [12]: !ls                 # Выводит список файлов в этом каталоге.
Meetings          Papers            code              books
databases         temp-file
In [13]: !rm temp -file      # Удаляет файл temp-file.

In [14]: !ls
Meetings          Papers            code              books
databases
```

Обратите внимание: по техническим причинам[58] команды `cd` (в Unix-подобных системах) и `chdir` (в Windows) обязательно должны выполняться как магические функции IPython:

```
In [15]: %cd /             # Переход в корневой (root) каталог.
In [16]: !ls
Applications          Volumes           usr               Library
bin                   net               Network           cores
opt                   www               System            dev
private               sbin              Users             home
In [17]: %cd ~/temp        # Переход в каталог temp в домашнем каталоге пользователя.
In [18]: !ls
output.txt            test.py           readme.txt        utils
zigzag.py
```

Если вы работаете в ОС Windows и хотите включить буквенное обозначение диска (например, `C:`) в путевое имя каталога, то необходимо заключить путевое имя в кавычки:

```
%cd 'C:\My Documents'
```

Справка `!`*`command`*`?` и автозавершение с помощью клавиши **Tab**, описанные выше, работают в командах операционной системы.

Можно передавать значения переменных Python в команды операционной системы, указывая перед именем переменной символ доллара `$`:

[58] Системные команды, выполняемые методом `!`*`command`*, порождают собственный экземпляр командной оболочки, который удаляется сразу после выполнения, поэтому смена каталога действует только в этой порожденной командной оболочке, но не отражается в командной оболочке IPython.

```
In [19]: python_script = 'zigzag.py'
In [20]: !ls $python_script
zigzag.py
In [21]: text_files = '*.txt'
In [22]: text_file_list = !ls $text_files                    ❶
In [23]: text_file_list
output.txt   readme.txt
In [24]: readme_file = text_file_list[1]
In [25]: !cat $readme_file
This is the file readme.txt
Each line of the file appears as an item
in a list when returned from !cat readme.txt

In [26]: readme_lines = !cat $readme_file                    ❷

In [27]: readme_lines
Out[28]:
['This is the file readme.txt',
 'Each line of the file appears as an item',
 'in a list when returned from !cat readme.txt']
```

❶ Обратите внимание: вывод системной команды можно присвоить переменной Python. Здесь это список файлов с расширением *.txt* в текущем каталоге.

❷ Системная команда `cat` возвращает содержимое указанного текстового файла. IPython разделяет этот вывод по символу перехода на новую строку и присваивает полученный в результате список переменной `readme_lines`. См. также раздел 5.1.3.

5.1.3 Магические функции IPython

IPython предоставляет множество «магических» функций (часто их называют просто magics – это команды с префиксом `%`) для ускорения написания кода и экспериментов в командной оболочке IPython. Некоторые наиболее полезные магические функции описаны в этом разделе, но более подробную информацию вы найдете в официальной документации по IPython (https://ipython.org/documentation.html). IPython различает строковые магические функции (line magics), аргументы которых определяются в одной строке, и секционные магические функции (cell magics), префиксом которых являются два символа процента `%%`, – эти функции работают с последовательностями команд Python. В данном разделе рассматривается пример применения секционной магической функции `%%timeit`.

Список доступных в настоящее время магических функций можно получить с помощью команды `%lsmagic`.

Магическая функция `%automagic` переключает «автоматическую» настройку: по умолчанию она включена (`ON`), т. е. при вводе имени магической функции без `%` функция все равно будет выполнена, если только это имя не было связано с каким-либо идентификатором некоторого объекта Python (т. е. стало именем переменной). То же правило применяется и к системным командам:

```
In [x]: ls
output.txt          test.py          readme.txt          utils
zigzag.py
In [x]: ls = 0
In [x]: ls           # Теперь ls содержит целое число, но !ls по-прежнему будет работать.
Out[x]: 0
```

В табл. 5.1 кратко описаны некоторые полезные магические функции IPython. В следующих подразделах более подробно рассматриваются некоторые магические функции, назначение которых не совсем понятно на первый взгляд.

Таблица 5.1. Некоторые полезные строковые магические функции IPython

Магическая функция	Описание
`%alias`	Создание псевдонима для системной команды
`%alias_magic`	Создание псевдонима для существующей магической функции IPython
`%bookmark`	Взаимодействие с каталогом системы закладок IPython
`%cd`	Смена текущего рабочего каталога
`%dhist`	Вывод списка ранее посещенных каталогов
`%edit`	Создание или редактирование кода Python в текстовом редакторе и последующее выполнение этого кода
`%env`	Вывод списка переменных системной среды, таких как `$HOME`
`%history`	Вывод списка хронологии команд, введенных в текущем сеансе IPython
`%load`	Считывание кода из заданного файла и предоставление возможности его редактирования
`%macro`	Определение именованной макрокоманды, создаваемой из ранее введенных команд, для повторного выполнения в будущем
`%paste`	Вставка ввода, сохраненного в буфере вставки. Этот способ более предпочтителен, чем, например, **Ctrl+V**, поскольку правильно обрабатывает форматирование исходного кода
`%recall`	Вывод одной или нескольких строк ввода из хронологии команд в текущей строке ввода после промпта
`%rerun`	Повторное выполнение ранее введенной команды из пронумерованной хронологии команд
`%reset`	Восстановление исходного пространства имен в текущем сеансе IPython
`%run`	Выполнение файла, заданного по имени, как Python-скрипта в текущем сеансе
`%save`	Сохранение группы введенных строк или макрокоманды (определенной с помощью `%macro`) в файле с заданным именем
`%sx или !!`	Выполнение в командной оболочке: запуск заданной команды оболочки и сохранение ее вывода
`%timeit`	Определение времени выполнения заданной команды Python
`%who`	Вывод всех переменных, определенных в текущий момент
`%who_ls`	Аналогична `%who`, но возвращает имена переменных как список строк
`%whos`	Аналогична `%who`, но выводит более подробную информацию о каждой переменной

Псевдонимы и закладки

Команде системной командной оболочки можно присвоить псевдоним (alias): сокращенное имя системной команды, которое можно вызывать как собственную магическую функцию. Например, в Unix-подобных системах можно определить следующий псевдоним для вывода списка только подкаталогов, расположенных в текущем каталоге:

```
In [x]: %alias lstdir ls -d */
In [x]: %lstdir
Meetings/         Papers/         code/           books/
databases/
```

После этого ввод команды `%lstdir` дает тот же результат, что и выполнение системной команды `!ls -d */`. Если с помощью `%automagic` установлено значение `ON`, то этот псевдоним можно вызывать и без символа `%`, т. е. просто `lstdir`.

Магическая функция `%alias_magic` обеспечивает такую же функциональность для магических функций IPython. Например, если вы хотите использовать `%h` как псевдоним для `%history`, то введите:

```
In [x]: %alias_magic h history
```

При работе над крупными проектами часто необходимо перемещаться между различными каталогами. IPython предоставляет простую систему сопровождения списка закладок (bookmarks), которые работают как быстрые способы перехода в различные каталоги. Синтаксис этой магической функции:

```
%bookmark <name> [directory]
```

Если аргумент [*directory*] не указан, то по умолчанию закладка определяется для текущего рабочего каталога.

```
In [x]: %bookmark py ~/research/code/python
In [x]: %bookmark www /srv/websites
In [x]: %cd py
/Users/christian/research/code/python
```

Может случиться так, что каталог с тем же именем, что и закладка, находится в текущем рабочем каталоге. В этом случае настоящий каталог имеет преимущество, поэтому необходимо воспользоваться командой `%cd -b <name>` для перехода по закладке.

Ниже приведены примеры полезных команд с использованием функции закладок:

- `%bookmark -l` – вывод списка всех закладок;
- `%bookmark -d <name>` – удаление закладки *<name>*;
- `%bookmark -r` – удаление всех закладок.

Измерение времени выполнения кода

Магическая функция IPython `%timeit <statement>` измеряет время выполнения однострочной команды *<statement>*. Команда выполняется *N* раз в цикле, а каждый цикл повторяется *R* раз. *N* – это наиболее подходящее,

обычно большое число, выбираемое командной оболочкой IPython для получения осмысленных результатов, а *R* по умолчанию равно 3. Выводится среднее время по всем циклам для наилучшей из *R* итераций цикла. Например, для профилирования операции сортировки случайно размещенных чисел от 1 до 100:

```
In [x]: import random
In [x]: numbers = list(range(1, 101))
In [x]: random.shuffle(numbers)
In [x]: %timeit sorted(numbers)
100000 loops, best of 3: 13.2 µs per loop
```

Разумеется, время выполнения будет зависеть от системы (скорость процессора, объем памяти и т. д.). Цель многократного повторения выполнения операции состоит в том, чтобы учесть различия в скоростях из-за других процессов, работающих в системе. Можно явно определить *N* и *R*, передавая их значения с помощью ключей `-n` и `-r` соответственно:

```
In [x]: %timeit -n 10000 -r 5 sorted(numbers)
10000 loops, best of 5: 11.2 µs per loop
```

Секционная магическая функция `%%timeit` позволяет измерить время выполнения многострочного блока кода. Например, время выполнения простейшего алгоритма поиска множителей целого числа `n` можно измерить так:

```
In [x]: n = 150
In [x]: %%timeit
factors = set()
for i in range(1, n+1):
    if not n % i:
        factors.add(n // i)
....:

100000 loops, best of 3: 16.3 µs per loop
```

Повторный вывод и повторное выполнение кода

Для повторного выполнения одной или нескольких строк кода из хронологии командной оболочки IPython используется магическая функция `%rerun` с указанием номера строки или диапазона номеров строк:

```
In [1]: import math
In [2]: angles = [0, 30, 60, 90]
In [3]: for angle in angles:
    sine_angle = math.sin(math.radians(angle))
    print('sin({:3d}) = {:8.5f}'.format(angle, sine_angle))
.....:
sin(  0) =   0.00000
sin( 30) =   0.50000
sin( 45) =   0.70711
sin( 60) =   0.86603
sin( 90) =   1.00000
```

```
In [4]: angles = [15, 45, 75]
In [5]: %rerun 3
=== Executing: ===
for angle in angles:
    sine_angle = math.sin(math.radians(angle))
    print('sin({:3d}) = {:8.5f}'.format(angle, sine_angle))

=== Output: ===
sin( 15) =   0.25882
sin( 45) =   0.70711
sin( 75) =   0.96593

In [6]: %rerun 2-3
=== Executing: ===
angles = [0, 30, 45, 60, 90]
for angle in angles:
    sine_angle = math.sin(math.radians(angle))
    print('sin({:3d}) = {:8.5f}'.format(angle, sine_angle))

=== Output: ===
sin(  0) =   0.00000
sin( 30) =   0.50000
sin( 45) =   0.70711
sin( 60) =   0.86603
sin( 90) =   1.00000
```

Похожая магическая функция `%recall` помещает запрошенные строки кода после промпта командной строки, но не выполняет их до тех пор, пока пользователь не нажмет клавишу **Enter**, т. е. позволяет при необходимости изменить выведенный код.

Если возникает необходимость в частом повторном выполнении некоторой последовательности команд, то можно определить именованную макрокоманду (macro) для вызова этой последовательности. Здесь также нужно задать номера строк команд:

```
In [7]: %macro sines 3
Macro `sines` created. To execute, type its name (without quotes).
=== Macro contents: ===
for angle in angles:
sine_angle = math.sin(math.radians(angle))
print('sin({:3d}) = {:8.5f}'.format(angle, sine_angle))

In [8]: angles = [-45, -30, 0, 30, 45]
In [9]: sines
sin(-45) =  -0.70711
sin(-30) =  -0.50000
sin(  0) =   0.00000
sin( 30) =   0.50000
sin( 45) =   0.70711
```

Загрузка, выполнение и сохранение исходного кода

Для загрузки кода из внешнего файла в текущий сеанс командной оболочки IPython используется магическая функция

```
%load <filename>
```

Если требуются только конкретные строки из входного файла, то необходимо указать их после ключа `-r`. Эта магическая функция вводит заданные строки после промпта командной строки, после чего их можно редактировать перед выполнением.

Для загрузки и выполнения кода из файла применяется команда

```
%run <filename>
```

После имени файла *`filename`* можно передать любые ключи и параметры командной строки. По умолчанию IPython интерпретирует их так же, как системная командная оболочка. Ниже кратко описаны некоторые дополнительные ключи магической функции `%run`:

- `-i`: запуск скрипта в текущем пространстве имен IPython, а не в пустом пространстве имен (т. е. программа получает доступ к переменным, определенным в текущем сеансе IPython);
- `-e`: игнорировать вызовы `sys.exit()` и исключения `SystemExit`;
- `-t`: вывод информации о времени после завершения выполнения (передача целого числа в дополнительном ключе `-N` позволяет повторить выполнение заданное количество раз).

Например, для 10-кратного выполнения `my_script.py` из командной оболочки IPython с выводом информации о затраченном времени:

```
In [x]: %run -t -N10 my_script.py
```

Для сохранения группы введенных строк или макрокоманды в файле используется магическая функция `%save`. Номера строк определяются по тем же синтаксическим правилам, что и для функции `%history`. Расширение файла *.py* добавляется автоматически, если пользователь не указал его. Перед перезаписью существующего файла выводится запрос, и пользователь должен подтвердить перезапись. Например:

```
In [x]: %save sines1 1 8 3
The following commands were written to file `sines1.py`:
import math
angles = [-45, -30, 0, 30, 45]
for angle in angles:
    print('sin({:3d}) = {:8.5f}'.format(angle, math.sin(math.radians(angle))))

In [x]: %save sines2 1-3
The following commands were written to file `sines2.py`:
import math
angles = [0, 30, 60, 90]
for angle in angles:
    print('sin({:3d}) = {:8.5f}'.format(angle, math.sin(math.radians(angle))))
```

Наконец, для добавления строк кода в файл вместо его перезаписи используется ключ `-a`:

```
%save -a <filename> <line numbers>
```

Захват вывода команд оболочки

Магическая функция IPython `%sx` *command*, равнозначная выражению `!!`*command*, выполняет команду оболочки *command* и возвращает выводимый результат как список (разделенный на осмысленные части по символу перехода на новую строку, так что в итоге получается по одному элементу в строке). Этот список можно присвоить переменной для дальнейшей обработки. Например:

```
In [x]: current_working_directory = %sx pwd
In [x]: current_working_directory
['/Users/christian/temp']
In [x]: filenames = %sx ls
In [x]: filenames
Out[x]:
['output.txt',
 'test.py',
 'readme.txt',
 'utils',
 'zigzag.py']
```

Здесь `filenames` – это список отдельных имен файлов.

В действительности возвращаемый объект – это объект списка строк `IPython.utils.text.SList`. Среди множества дополнительных полезных возможностей, предоставляемых объектом `SList`, можно выделить собственный метод для разбивки каждой строки на поля, разделяемые пробельными символами: `fields`, метод сортировки по этим полям: `sort`, а также метод поиска в этом списке строк: `grep`. Например:

```
In [x]: files = %sx ls -l
In [x]: files
['total 8',
 '-rw-r--r--   1 christian staff      93   5 Nov 16:30 output.txt',
 '-rw-r--r--   1 christian staff   23258   5 Nov 16:31 readme.txt',
 '-rw-r--r--   1 christian staff     218   5 Nov 16:32 test.py',
 'drwxr-xr-x   2 christian staff      68   5 Nov 16:32 utils',
 '-rw-r--r--   1 christian staff     365   5 Nov 16:20 zigzag.py']
In [x]: del files[0]     # Удаление последней строки 'total 8', не являющейся именем файла.
In [x]: files.fields()
Out[x]:
[['-rw-r--r--', '1', 'christian', 'staff', '93', '5', 'Nov', '16:30', 'output.txt'],
['-rw-r--r--', '1', 'christian', 'staff', '23258', '5', 'Nov', '16:31', 'readme.txt'],
...
['-rw-r--r--', '1', 'christian', 'staff', '365', '5', 'Nov', '16:20', 'zigzag.py']]

In [x]: ['{} last modified at {} on {} {}'.format(f[8], f[7], f[5], f[6])
                              for f in files.fields()]
Out[x]:
['output.txt last modified at 16:30 on 5 Nov',
 'readme.txt last modified at 16:31 on 5 Nov',
 'test.py last modified at 16:32 on 5 Nov',
 'utils last modified at 16:32 on 5 Nov',
 'zigzag.py last modified at 16:20 on 5 Nov']
```

Метод `fields` также может принимать аргументы, определяющие индексы полей, которые необходимо вывести. Если задано более одного индекса, то поля соединяются пробелами:

```
In [x]: files.fields(0)     # Первое поле в каждой строке списка files.
Out[x]: ['-rw-r--r--', '-rw-r--r--', '-rw-r--r--', 'drwxr -xr-x', '-rw-r--r--']
In [x]: files.fields(-1)    # Последнее поле в каждой строке списка files.
Out[x]: ['output.txt', 'readme.txt', 'test.py', 'utils', 'zigzag.py']

In [x]: files.fields(8, 7, 5, 6)
Out[x]:
['output.txt 16:30 5 Nov',
'readme.txt 16:31 5 Nov',
'test.py 16:32 5 Nov',
'utils 16:32 5 Nov',
'zigzag.py 16:20 5 Nov']
```

Метод `sort`, предоставляемый объектами типа `SList`, может выполнять сортировку по заданному полю с дополнительной возможностью преобразования поля из строки в число, если требуется (например, для сравнения `10 > 9`). Следует отметить, что этот метод возвращает новый объект типа `SList`.

```
In [x]: files.sort(4) # Сортировка по размеру в алфавитно-цифровом порядке (не очень удобно).
Out[x]:
['-rw-r--r--  1 christian  staff     218  5 Nov 16:32 test.py',
 '-rw-r--r--  1 christian  staff   23258  5 Nov 16:31 readme.txt',
 '-rw-r--r--  1 christian  staff     365  5 Nov 16:20 zigzag.py',
 'drwxr-xr-x  2 christian  staff      68  5 Nov 16:32 utils',
 '-rw-r--r--  1 christian  staff      93  5 Nov 16:30 output.txt']

In [x]: files.sort(4, nums=True) # Сортировка по размеру в цифровом порядке (удобно).
Out[x]:
['drwxr-xr-x  2 christian  staff      68  5 Nov 16:32 utils',
 '-rw-r--r--  1 christian  staff      93  5 Nov 16:30 output.txt',
 '-rw-r--r--  1 christian  staff     218  5 Nov 16:32 test.py',
 '-rw-r--r--  1 christian  staff     365  5 Nov 16:20 zigzag.py',
 '-rw-r--r--  1 christian  staff   23258  5 Nov 16:31 readme.txt']
```

Метод `grep` возвращает элементы списка `SList`, которые содержат заданную строку[59]. Для поиска строки только в заданном поле используется аргумент `field`:

```
In [x]: files.grep('txt')             # Поиск строк, содержащих образец 'txt'.
Out[x]:
['-rw-r--r--   1 christian  staff      93  5 Nov 16:30 output.txt',
 '-rw-r--r--   1 christian  staff   23258  5 Nov 16:31 readme.txt']

In [x]: files.grep('16:32', field=7)      # Поиск файлов, созданных в 16:32.
Out[x]:
['-rw-r--r--   1 christian  staff     218  5 Nov 16:32 test.py',
 'drwxr-xr-x   2 christian  staff      68  5 Nov 16:32 utils']
```

[59] В действительности предполагается, что имена должны соответствовать еще и регулярным выражениям, но здесь мы не будем углубляться в эту тему.

Пример П5.1. Рибонуклеиновая кислота (РНК) кодирует аминокислоты пептидов как последовательность кодонов. Каждый кодон состоит из трех нуклеотидов, выбираемых из следующего «алфавита»: U (uracil – урацил), C (cytosine – цитозин), A (adenine – аденин) и G (guanine – гуанин).

Скрипт на Python *codon_lookup.py*, доступный по адресу https://scipython.com/eg/bab, создает словарь `codon_table`, отображающий кодоны в аминокислоты, при этом каждая аминокислота идентифицируется по ее однобуквенному обозначению (например, `R` = аргинин (arginine)). Конечные кодоны, обозначающие завершение кодирующего преобразования РНК, идентифицируются с помощью одиночного символа «звездочка» *. Кодон `AUG` обозначает начало преобразования в последовательность нуклеотидов, а также кодирование для аминокислоты метионин (methionine).

Этот скрипт можно выполнить в командной оболочке IPython с помощью команды `%run codon_lookup.py` (или с помощью загрузки и последующего выполнения командой `%load codon_lookup.py` после нажатия клавиши **Enter**):

```
In [x]: %run codon_lookup.py
In [x]: codon_table
Out[x]:
{'GCG': 'A',
 'UAA': '*',
 'GGU': 'G',
 'UCU': 'S',
 ...
 'ACA': 'T',
 'ACC': 'T'}
```

Определим функцию для преобразования последовательности РНК. Выполните команду `%edit` и введите приведенный ниже код в появившемся окне редактора.

```
def translate_rna(seq):
    start = seq.find('AUG')
    peptide = []
    i = start
    while i < len(seq)-2:
        codon = seq[i:i+3]
        a = codon_table[codon]
        if a == '*':
            break
        i += 3
        peptide.append(a)
    return ''.join(peptide)
```

После выхода из редактора этот код будет выполнен с определением функции `translate_rna`:

```
IPython will make a temporary file named: /var/folders/fj/yv29fhm91v7_6g
7sqsy1z2940000gp/T/ipython_edit_thunq9/ipython_edit_dltv_i.py
Editing... done. Executing edited code...
Out[x]: "def translate_rna(seq):\n    start = seq.find('AUG ')\n
peptide = []\
n    i = start\n    while i < len(seq)-2:\n        codon = seq[i:i+3]\n         a
 = codon_table[codon]\n        if a == '*':\n            break\n         i += 3\n
        peptide.append(a)\n    return ''.join(peptide)\n"
```

Теперь можно передать в эту функцию последовательность РНК для преобразования:

```
In[x]: seq = 'CAGCAGCUCAUACAGCAGGUAAUGUCUGGUCUCGUCCCCGGAUGUCGCUACCCACGAG
ACCCGUAUCCUACUUUCUGGGGAGCCUUUACACGGCGGUCCACGUUUUUCGCUACCGUCGUUUUCCCGGUGC
CAUAGAUGAAUGUU'
In [x]: translate_rna(seq)
Out[x]: 'MSGLVPGCRYPRDPYPTFWGAFTRRSTFFATVVFPVP'
```

Для считывания списка РНК-последовательностей (по одной строке) из текстового файла *seqs.txt* и их преобразования можно использовать магическую функцию `%sx` и системную команду `cat` (или в Windows системную команду `type`):

```
In [x]: seqs = %sx cat seqs.txt
In [x]: for seq in seqs:
   ...:     print(translate_rna(seq))
   ...:
MHMLDENLYDLGMKACHEGTNVLDKWRNMARVCSCDYQFK
MQGSDGQQESYCTLPFEVSGMP
MPVEWRTMQFQRLERASCVKDSTFKNTGSFIKDRKVSGISQDEWAYAMSHQMQPAAHYA
MIVVTMCQ
MGQCMRFAPGMHGMYSSFHPQHKEITPGIDYASMNEVETAETIRPI
```

5.1.4 Упражнения

Задачи

35.1.1. Усовершенствовать алгоритм поиска количества множителей целого числа, приведенный в разделе 5.1.3, следующим образом:

а) зациклить проверку пробного множителя `i` до тех пор, пока он не больше значения квадратного корня из `n` (почему нет необходимости проверять значения `i`, большие значения квадратного корня из `n`?);

б) использовать генератор (см. раздел 4.3.5).

Сравнить скорость выполнения двух вариантов решения этой задачи, используя магическую функцию IPython `%timeit`.

35.1.2. Используя самый быстрый алгоритм из предыдущей задачи (35.1.1), написать небольшой фрагмент кода для определения сверхсоставных чисел, меньших 100 000, и использовать секционную магическую функцию `%%timeit` для измерения времени выполнения этого кода. Сверхсоставное число – это положительное целое число, которое имеет больше делителей, чем любое меньшее положительное целое число, например 1, 2, 4, 6, 12, 24, 36, 48, ….

5.2 Блокнотная среда Jupyter Notebook

Jupyter Notebook предоставляет интерактивную среду для программирования на языке Python в веб-браузере[60]. Главное преимущество этой среды по сравнению с обычной методикой использования консольной командной обо-

[60] С версии 4 проект IPython Notebook был переименован в Jupyter Notebook с включением возможности связывания с другими языками программирования наряду с Python.

лочки IPython состоит в том, что код Python может объединяться с документацией (в том числе оформленной с помощью LaTeX), изображениями и даже с полноценными медиасредствами, такими как встроенные видеофрагменты. Виртуальные блокнотные среды Jupyter Notebook все больше используются научными работниками для обмена результатами их исследований, включающими процедуры вычисления, выполняемые с данными, так же просто, как результаты этих вычислений. Подобный формат упрощает совместную работу исследователей над проектом, а остальным предоставляет возможность тщательно проверить итоги исследований посредством воспроизведения вычислений с теми же данными.

5.2.1 Основы использования Jupyter Notebook

Запуск сервера Jupyter Notebook

Если блокнотная среда Jupyter Notebook уже установлена, то сервер, обеспечивающий браузерный интерфейс с командной оболочкой IPython, можно активизировать из командной строки:

```
jupyter notebook
```

После выполнения этой команды открывается окно веб-браузера с URL локального приложения Jupyter Notebook. По умолчанию это адрес http://localhost:8888, хотя по умолчанию можно определить другой номер порта, если 8888 уже используется.

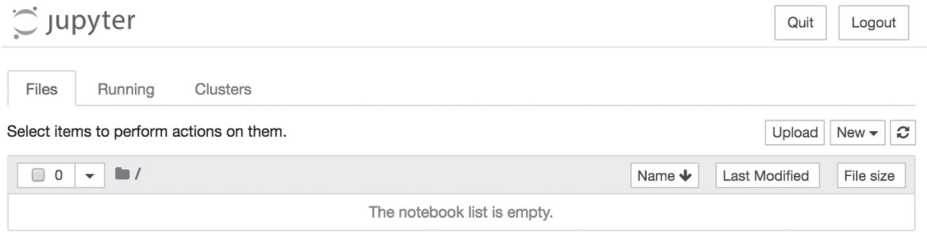


Рис. 5.1. Начальная страница (index) блокнотной среды Jupyter Notebook

Начальная страница (index) блокнотной среды Jupyter Notebook (см. рис. 5.1) содержит список блокнотов, доступных в текущий момент в каталоге, из которого был запущен сервер. Это также каталог, в котором по умолчанию будут сохраняться блокноты (с расширением *.ipynb*). Поэтому правильным решением будет выполнение приведенной выше команды в каталоге проекта, с которым вы работаете.

Начальная страница содержит три закладки: Files – список всех файлов, включая блокноты Jupyter Notebook и подкаталоги в текущем рабочем каталоге; Running – список блокнотов, в настоящий момент активных в текущем сеансе (даже если они не открыты в окне браузера); Clusters – предоставляет интерфейс к механизму параллельных вычислений IPython, но эта тема не рассматривается в данной книге.

На начальной странице можно создать новый блокнот (щелкнув по кнопке **New** > **Notebook: Python 3**) или открыть существующий (щелкнув по его имени). Для импорта существующего блокнота на начальную страницу можно либо щелкнуть по кнопке **Upload** (слева от кнопки **New**), либо перетащить в список блокнотов файл блокнота из любого приложения операционной системы, в котором указано имя этого файла.

Чтобы остановить работу сервера блокнотов, нажмите клавиши **Ctrl**+**C** в окне терминала, из которого был запущен сервер (и подтвердите останов после вывода промпта).

Редактирование блокнота Jupyter Notebook

Для начала работы с новым блокнотом щелкните по кнопке **New** и выберите ядро (должно предлагаться как минимум одно ядро с именем Python 3). Откроется новая вкладка браузера, содержащая интерфейс, где вы будете писать код и устанавливать соединение с ядром IPython, т. е. с процессом, отвечающим за выполнение кода и передачу полученных результатов обратно в браузер.

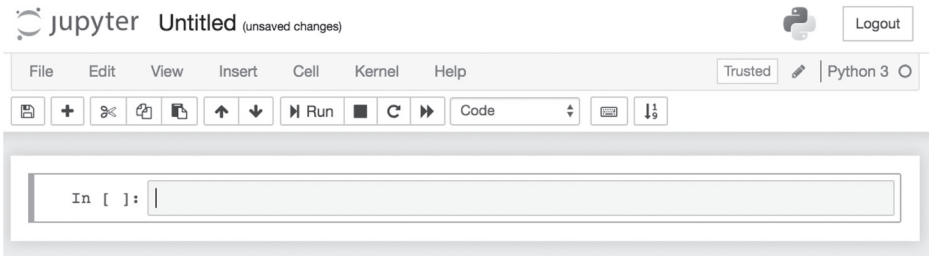


Рис. 5.2. Блокнотная среда Jupyter с новым документом-блокнотом

Новый документ-блокнот (см. рис. 5.2) состоит из заголовка, панели меню и панели инструментов, под которыми расположен промпт командной оболочки IPython, где вы будете вводить код и команды разметки текста (например, описания и документацию) как последовательность ячеек (cells).

В заголовке указано имя первого блокнота, открытого вами, – вероятнее всего, это будет имя Untitled (Без имени). Щелкните по имени, чтобы заменить его на более информативное название. Панель меню содержит команды сохранения, копирования, вывода (на печать и т. п.), реорганизации и прочих операций обработки документа Jupyter Notebook. На панели инструментов расположены значки на кнопках, предоставляющие более быстрые и удобные способы наиболее часто выполняемых операций, которые также доступны в панели меню.

Существует три типа ячеек ввода, в которых можно записывать содержимое блокнота:

- ячейки кода (code cells): тип ячейки по умолчанию. Этот тип ячеек состоит из выполняемого кода. При изучении данной главы вы будете писать в этих ячейках код Python, но Jupyter предоставляет механизм выполнения кода, написанного на других языках, таких как Julia и R;

- ячейки текста с разметкой (markdown cells): этот тип ячеек предоставляет богатые возможности оформления документации для исходного кода. При выполнении ввод в такую ячейку преобразуется в формат HTML, который может включать математические формулы, шрифтовое оформление, списки, таблицы, встроенные изображения и видеофрагменты;
- необработанные ячейки (raw cells): ввод в эти ячейки не изменяется блокнотной средой – полностью сохраняется содержимое и форматирование введенного текста.

Работа с ячейками

Каждая ячейка может состоять из одной или нескольких строк ввода, но содержимое ячейки не интерпретируется до тех пор, пока вы не «запустите» (т. е. выполните) ее. Можно выбрать требуемый вариант в панели меню (в спускающемся подменю **Cell**), или щелкнуть по кнопке «воспроизведения» **Run** в панели инструментов, или воспользоваться одним из сочетаний «быстрых клавиш»:

- **Shift**+**Enter** – выполняет ячейку, выводит любой результат, затем перемещает курсор в ячейку, расположенную ниже. Если ниже нет ячейки, то создает новую пустую ячейку;
- **Ctrl**+**Enter** – выполняет текущую ячейку, но оставляет курсор в ней. Удобно для быстрой проверки правильности работы «одноразовых» команд или для извлечения списка содержимого каталога;
- **Alt**+**Enter** – выполняет ячейку, выводит любой результат, затем вставляет результат и перемещает курсор в новую ячейку, созданную прямо под текущей.

Существуют еще две полезные «быстрые клавиши». При редактировании ячейки клавиши со стрелками позволяют перемещаться по содержимому ячейки (режим редактирования). Если в этом режиме нажать клавишу **Esc**, то выполняется переход в режим команд, в котором клавиши со стрелками позволяют перемещаться между ячейками. Чтобы снова войти в режим редактирования выбранной ячейки, необходимо нажать клавишу **Enter**.

В панели меню спускающееся подменю **Cell** предоставляет много способов выполнения ячеек блокнота: обычно требуется выполнение отдельной текущей ячейки или выполнение текущей ячейки и всех ячеек ниже нее.

Ячейки кода

В ячейку кода можно вводить все, что допустимо при написании Python-программы в редакторе или в обычной командной оболочке IPython. Код в текущей ячейке имеет доступ к объектам, определенным в других ячейках (обеспечивая их совместную работу). Например:

```
In [ ]:  n = 10
```

При нажатии клавиш **Shift**+**Enter** или при щелчке по кнопке **Run** выполняется эта инструкция (определение `n`, но без вывода) и открывается новая ячейка прямо под текущей:

```
In [1]:  n = 10

In [ ]:
```

Ввод следующих инструкций в этой новой ячейке

```
In [ ]:  sum_of_squares = n * (n+1) * (2*n+1) // 6
         print( '1**2 + 2**2 + ... + {}**2 = {}'.format(n,
             sum_of_squares))
```

и их выполнение вышеописанным способом генерирует вывод и открывает третью пустую ячейку ввода. Теперь весь документ блокнота выглядит следующим образом:

```
In [1]:  n = 10

In [2]:  sum_of_squares = n * (n+1) * (2*n+1) // 6
         print ('1**2 + 2**2 + ... + {}**2 = {}'.format(n,
             sum_of_squares))

Out[2]:  1**2 + 2**2 + ... + 10**2 = 385

In [ ]:
```

Можно отредактировать значение `n` в ячейке ввода 1 и повторно выполнить весь документ для обновления вывода. Следует отметить, что можно также установить новое значение для `n` после вычисления в ячейке 2:

```
In [3]:  n = 15
```

При выполнении ячейки 3, затем ячейки 2 вывод в ячейке 2 принимает следующий вид:

```
Out[2]:  1**2 + 2**2 + ... + 15**2 = 1240
```

даже если в ячейке выше сохраняется определение значения 10 для переменной `n`. Таким образом, если вы не выполняете весь документ с самого начала, то вывод не обязательно отображает результат выполнения скрипта, соответствующий выполнению ячеек в порядке их расположения.

В блокнотной среде Jupyter Notebook можно использовать все системные команды (с префиксами ! или !!) и магические функции IPython.

Ячейки текста с разметкой

Ячейки текста с разметкой выполняют преобразование введенного текста в формат HTML, применяя стили, соответствующие простому синтаксису, де-

монстрируемому ниже. Полную документацию можно найти здесь: https://daringfireball.net/projects/markdown/.

По этому адресу описаны наиболее полезные функциональные возможности. Полный блокнот Jupyter Notebook можно загрузить здесь: https://scipython.com/book/markdown.

Основные команды разметки текста

- Простые стили можно применять, просто помещая текст между звездочками или символами подчеркивания:

```
In [x]: Surrounding text by two asterisks denotes
            **bold style**; using one asterisk denotes
            *italic text*, as does _a single underscore_
```

Surrounding text by two asterisks denotes **bold style**; using one asterisk denotes *italic text*, as does *a single underscore*.

- Заголовки до шести уровней (от заголовка раздела самого высокого уровня до текста уровня абзаца) обозначаются символами # – от одного до шести соответственно: текст, следующий за этими символами, отображается шрифтами различного размера (в HTML используются элементы от <h1> до <h6>).
- Цитаты обозначаются одиночной угловой кавычкой >:

```
In [x]: > "Climb if you will, but remember that courage
          and strength are nought without prudence, and that
          a momentary negligence may destroy the happiness of
          a lifetime. Do nothing in haste; look well to each step;
          and from the beginning think what may be the end." -
          Edward Whymper
```

> "Climb if you will, but remember that courage and strength are nought without prudence, and that a momentary negligence may destroy the happiness of a lifetime. Do nothing in haste; look well to each step; and from the beginning think what may be the end." – Edward Whymper[61].

[61] «Покоряйте горы, если есть желание, но не забывайте, что отвага и сила ничего не стоят без благоразумия, что счастье длиной в целую жизнь может быть уничтожено секундной небрежностью. Ничего не делайте второпях, просчитывайте каждый шаг и с самого начала думайте о том, каким может быть конец». – Эдвард Уимпер. «Карабкаясь в Альпы».

- Примеры кода (не для выполнения, а для иллюстрации) располагаются между двумя пустыми строками и форматируются с помощью четырех пробелов (или табуляции). Результат выводится моноширинным шрифтом с отображением символов в том виде, в каком они были введены:

```
In [x]:       n = 57
              while n != 1:
                  if n % 2:
                      n = 3*n + 1
                  else:
                      n //= 2
```

```
n = 57
while n != 1:
    if n % 2:
        n = 3*n + 1
    else:
        n //= 2
```

- Примеры кода в строке текста создаются с помощью символов обратного апострофа (`):

```
In [x]: Here are some Python keywords: `for`, `while`
            and `lambda`.
```

Here are some Python keywords: `for`, `while` and `lambda`.

- Новые абзацы начинаются после пустой строки.

Код HTML в тексте с разметкой

Язык разметки, используемый в Jupyter Notebook, включает HTML, поэтому можно применять объекты и теги языка HTML прямо в тексте (например, тег `<em>` для выделения), а также стили CSS для создания прочих эффектов, таких как подчеркнутый текст. Даже такие сложные объекты HTML, как, например, таблицы, можно сформировать прямо в тексте.

```
In [x]: The following <em>Punnett table</em> is <span
            style="text-decoration: underline" >marked up</span>
            in HTML.

        <table style="text-align: center;">
        <tr>
        <th style="border-top:none; border-left:none;" rowspan="2"
            colspan="2"></th>
        <th colspan="2">Male</th>
        </tr>
        <tr>
        <th>A</th>
        <th>a</th>
        </tr>
        <tr>
        <th rowspan="2">Female</th>
        <th>a</th>
        <td style="background: #aaa;">Aa</td>
        <td>aa</td>
        </tr>
        <tr>
        <th>a</th>
        <td style="background: #aaa;">Aa</td>
        <td>aa</td>
        </tr>
        </table>
```

The following Punnett table is marked up in HTML.

		Male	
		A	**a**
Female	**a**	Aa	aa
	a	Aa	aa

Списки

Маркированные (ненумерованные) списки создаются с использованием любых маркеров *, + или – , а вложенные подсписки просто смещаются соответствующим образом.

```
In [x]: The inner planets and their satellites:

        * Mercury
        * Venus
        * Earth
            * The  Moon
        + Mars
            - Phoebus
            - Deimos
```

The inner planets and their satellites:

- Mercury
- Venus
- Earth
  - The Moon
- Mars
  - Phoebus
  - Deimos

Упорядоченные, или нумерованные, списки создаются с применением номеров (чисел) с точкой и пробелом перед каждым пунктом списка:

```
In [x]: 1. Symphony No. 1 in C major, Op. 21
        2. Symphony No. 2 in D major, Op. 36
        3. Symphony No. 3 in E-flat major ("Eroica"), Op. 55
```

1. Symphony No. 1 in C major, Op. 21
2. Symphony No. 2 in D major, Op. 36
3. Symphony No. 3 in E-flat major ("Eroica"), Op. 55

Ссылки

Существуют три способа создания ссылок в тексте с разметкой:

- ссылки в строке (inline links) – URL записывается в круглых скобках после текста в квадратных скобках, который должен быть преобразован в ссылку. Например:

```
In [x]: Here is a link to the
        [IPython website]( https://ipython.org/) .
```

Here is a link to the IPython website.

- гиперссылки (reference links) помечают текст, который должен быть преобразован в ссылку, помещая имя ссылки (содержащее буквы, числа и пробелы) в квадратных скобках. Предполагается, что имя определяется с использованием следующего синтаксиса [*имя*]: *url* где-либо в документе, как показано в приведенном ниже примере ячейки текста с разметкой:

```
In [x]: Some important mathematical sequences are the [prime
            numbers][primes],
        [Fibonacci sequence][fib] and the [Catalan
            numbers][catalan_numbers].

        ...

        [primes]: https://oeis.org/A000040
        [fib]: https://oeis.org/A000045
        [catalan_numbers]:  https://oeis.org/A000108 ]
```

Some important mathematical sequences are the primes , Fibonacci sequence and the Catalan numbers .

- автоматические ссылки (automatic links) – для них текст, по которому можно щелкнуть, представляет собой URL. Такие ссылки создаются простым размещением URL в угловых скобках:

```
In [x]: My website is < https://christianhill.co.uk >.
```

My website is https://christianhill.co.uk

Если ссылкой является файл в локальной системе, то в качестве URL задается путевое имя относительно каталога текущего блокнота с префиксом `files/`.

```
In [x]: Here is [a local data file](files/data/data0.txt).
```

Here is a a local data file .

Обратите внимание: при щелчке по любой ссылке открывается новая вкладка в браузере.

Математика

Математические формулы могут записываться на языке LaTeX и в дальнейшем отображаются с использованием библиотеки JavaScript MathJax. Формулы в строке текста выделяются одиночными символами доллара, а отдельно выводимые формулы выделяются двойными символами доллара:

```
In [x]: An inline equation appears within a sentence of text, as
            in the definition of the function $f(x) = \  sin(x^2)$;
            displayed equations get their own line(s) between
            lines of text:
        $$\int_0^\infty \mathrm{e}^{-x^2}dx =
            \frac{\sqrt{\pi}}{2}.$$
```

An inline equation appears within a sentence of text, as in the definition of the function $f(x) = \sin(x^2)$; displayed equations get their own line(s) between lines of text:

$$\int_0^\infty e^{-x^2}\,\mathrm{d}x = \frac{\sqrt{\pi}}{2}.$$

Изображения и видеофрагменты

Ссылки на файлы изображений работают точно так же, как обычные ссылки (и могут быть ссылками в строке текста или гиперссылками), но им предшествует восклицательный знак `!`. Текст в квадратных скобках между восклицательным знаком и ссылкой действует как текст, заменяющий изображение. Например:

```
In [x]:  ![An interesting plot of the Newton
             fractal](/files/images/newton_fractal.png)
         ![A remote link to a star
             image]( https://christianhill.co.uk/static/images/
             star.svg)
```

Для ссылок на видеофрагменты обязательным является использование тега HTML5 `<video>`, но следует отметить, что не все браузеры поддерживают видеоформаты в полной мере, например:

```
In [x]:  <video controls style="width: 500px; margin: 0 auto;
             display: block;" src="files/diffmap-animated.ogv" />
```

Данные, формирующие изображения, видео и другой локально позиционируемый контент, не включаются непосредственно в документ блокнота: эти файлы обязательно должны включаться в дистрибутивный комплект блокнота при распространении.

5.2.2 Преобразование документа блокнота в другие форматы

Инструментальное средство `nbconverter` устанавливается вместе с Jupyter Notebook для преобразования блокнотов из собственного формата .ipynb[62] в один из нескольких альтернативных форматов. Инструмент преобразования запускается из (системной) командной строки:

```
jupyter nbconvert --to <format> <notebook.ipynb>
```

где *`notebook.ipynb`* – имя файла блокнота Jupyter Notebook, который необходимо преобразовать, а *`format`* – требуемый формат вывода. По умолчанию (если

[62] В действительности это просто формат документа JSON (JavaScript Object Notation).

параметр *format* не задан) результатом становится статический HTML-файл, как описано в следующем разделе.

Преобразование в формат HTML

Команда

```
jupyter nbconvert <notebook.ipynb>
```

выполняет преобразование файла *notebook.ipynb* в формат HTML и создает файл *notebook.html* в текущем каталоге. Этот файл содержит все необходимые заголовки для независимой HTML-страницы, которая достаточно точно воспроизводит интерактивное представление, формируемое сервером Jupyter Notebook, но в виде статического документа.

Если необходим только код HTML, соответствующий документу блокнота без заголовка (т. е. без тегов <html>, <head>, <body> и т. п.), предназначенный для включения в существующую веб-страницу, то в приведенную выше команду добавляется ключ `--template basic`.

Все дополняющие файлы, такие как изображения, автоматически размещаются в каталоге с тем же основным (базовым) именем, что и сам блокнот, но с добавлением суффикса *_files*. Например, команда `jupyter nbconvert mynotebook.ipynb` генерирует файл *mynotebook.html* и подкаталог *mynotebook_files*.

Преобразование в формат LaTeX

Для экспорта блокнота в формате документа LaTeX применяется команда

```
jupyter nbconvert --to latex <notebook.ipynb>
```

Для автоматической генерации PDF-файла с помощью утилиты `pdflatex`, применяемой к преобразованному файлу *notebook.tex*, в приведенную выше команду добавляется ключ `--post pdf`.

Преобразование в формат Markdown

Команда

```
jupyter nbconvert --to markdown <notebook.ipynb>
```

выполняет преобразование всего блокнота в целом в формат markdown (https://daringfireball.net/projects/markdown/) (см. раздел 5.2.1): ячейки, уже содержащие текст в формате markdown, не изменяются, а ячейки кода размещаются в блоках с тройным обратным апострофом (```).

Преобразование в код Python

Команда

```
jupyter nbconvert --to python <notebook.ipynb>
```

выполняет преобразование файла *notebook.ipynb* в выполняемый скрипт Python. Если какая-либо из ячеек кода этого блокнота содержит магические

функции IPython, то сгенерированный скрипт может выполняться только в сеансе IPython. Ячейки в формате markdown и прочие текстовые ячейки будут преобразованы в комментарии в сгенерированном коде скрипта Python.

5.2.3 JupyterLab

Во время написания этой книги проект Project Jupyter тестировал интерактивную среду разработки (IDE) на основе браузера JupyterLab, которая расширяет функциональность виртуальной блокнотной среды Jupyter Notebook и обеспечивает совместную работу нескольких пользователей, поддержку метода «перетащи и брось» (drag-and-drop) для ячеек блокнота, доступ к терминалу (консоли) на основе браузера, автоматическое завершение ключевых слов и имен объектов, а также динамический предварительный просмотр текста с разметкой markdown. Поддерживается возможность установки специализированных сторонних виджетов, позволяющих загружать и исследовать данные в различных форматах прямо в браузере, а также интеграция с широко известными онлайновыми сервисами, такими как GitHub, Dropbox и Google Drive. Будет обеспечена полная обратная совместимость с существующими блокнотами Jupyter Notebook. Более подробную информацию о JupyterLab можно найти на веб-сайте проекта Project Jupyter https://jupyter.org/.

Глава 6

Библиотека NumPy

Библиотека NumPy стала фактическим стандартом в целом для научного программирования на Python. Ее ядром является объект `ndarray`, многомерный массив данных одного типа, который можно сортировать, изменять его форму, он может быть операндом в математических операциях и в статистическом анализе, его можно записывать в файлы, затем считывать из них и т. д. Реализации NumPy таких математических операций и алгоритмов обладают двумя главными преимуществами над основными объектами языка Python, которые мы использовали до сих пор. Во-первых, они реализованы как предварительно скомпилированный код на языке C, поэтому обеспечивают скорость выполнения, сравнимую со скоростью программы, написанной на C. Во-вторых, библиотека NumPy поддерживает векторизацию: единственная операция может быть выполнена над всем массивом в целом вместо применения явного цикла для последовательной обработки элементов массива. Например, сравним операцию умножения двух одномерных списков *a* и *b* из *n* чисел в обычном языке Python:

```
c = []
for i in range(n):
    c.append(a[i] * b[i])
```

и с использованием массивов NumPy[63]:

```
c = a * b
```

Поэлементное умножение выполняется оптимизированным, предварительно скомпилированным кодом на языке C, поэтому представляет собой весьма быструю операцию (намного быстрее при больших значениях `n`, чем альтернативный код на обычном Python). Отсутствие явной конструкции цикла и индексирования делает код более ясным, в меньшей степени подверженным ошибкам и приближает его к соответствующему стандартному математическому формату записи.

Вся функциональность библиотеки NumPy представлена в пакете `numpy`. Для использования настоятельно рекомендуется импортировать этот пакет командой

[63] Термины «массив NumPy» и `ndarray` в этой книге будут использоваться как взаимозаменяемые.

```
import numpy as np
```

и в дальнейшем обращаться к его атрибутам с помощью префикса `np` (например, `np.array`). Именно так используется библиотека NumPy в этой книге.

6.1 Основные методы массива

Классом массива NumPy является `ndarray`, который состоит из многомерной таблицы элементов, индексируемых кортежем целых чисел. В отличие от списков и кортежей Python, элементы не могут принадлежать к различным типам: каждый элемент в массиве NumPy имеет один и тот же тип, который определяется соответствующим объектом типа данных (`dtype`). Тип `dtype` массива определяет не только общий класс элемента (целое число, число с плавающей точкой и т. п.), но также его представление в памяти (например, сколько битов занимает элемент) – см. раздел 6.1.2.

Размеры массива NumPy называются осями (axes), а количество осей массива называется его рангом (rank)[64].

6.1.1 Создание массива

Создание простого массива

Самый простой способ создания небольшого массива NumPy – вызов конструктора `np.array` со списком или кортежем значений:

```
In [x]: import numpy as np
In [x]: a = np.array( (100, 101, 102, 103) )
In [x]: a
Out[x]: array([100, 101, 102, 103])
In [x]: b = np.array( [[1.,2.], [3.,4.]] )
Out[x]:
array([[ 1., 2.],
       [ 3., 4.]])
```

Обратите внимание: при передаче списка списков создается двумерный массив (это действует и для большего числа размерностей).

Индексация многомерного массива NumPy немного отличается от индексации обычного списка списков в Python: вместо указания индекса требуемого элемента в отдельных квадратных скобках `b[i][j]` используется кортеж целых чисел `b[i,j]`:

```
In [x]: b[0,1]          # То же самое, что и b[(0,1)].
Out[x]: 2.0
In [x]: b[1,1] = 0.     # Кортеж индексов используется и для присваивания.
Out[x]:
array([[ 1., 2.],
       [ 3., 0.]])
```

[64] Не следует путать ранг массива в библиотеке NumPy с концепцией ранга матрицы в линейной алгебре.

Тип данных логически выводится из типа элементов последовательности и «принудительно приводится» к наиболее общему типу, если заданы элементы различных, но совместимых типов:

```
In [x]: np.array( [-1, 0, 2.])     # Смешаны значения типов int и float: приведение к типу
float.
Out[x]: array([-1., 0., 2.])
```

Можно также явно установить тип данных с помощью необязательного аргумента `dtype` (см. раздел 6.1.2):

```
In [x]: np.array( [0, 4, -4], dtype=complex)
In [x]: array([ 0.+0.j,  4.+0.j, -4.+0.j])
```

Если массив большой или во время его создания значения элементов неизвестны, то предлагается несколько методов объявления массива заданной формы, заполненного значениями по умолчанию или произвольно выбранными значениями. Самый простой и быстрый метод `np.empty` принимает кортеж, описывающий форму, и создает массив без инициализации его элементов: начальные значения элементов не определены (обычно это случайный «мусор», содержащийся в памяти, выделенной Python для этого массива).

```
In [x]: np.empty((2,2))
Out[x]:
array([[ -2.31584178e+077,  -1.72723381e-077],
       [ 2.15686807e-314,    2.78134366e-309]])
```

Существуют также более удобные методы `np.zeros` и `np.ones`, создающие массив заданной формы с элементами, равными 0 и 1 соответственно. Методы `np.empty`, `np.zeros` и `np.ones` принимают необязательный аргумент `dtype`.

```
In [x]: np.zeros((3,2))      # По умолчанию для dtype определен тип  'float'.
Out[x]:
array([[ 0., 0.],
       [ 0., 0.],
       [ 0., 0.]])
In [x]: np.ones((3,3), dtype=int)
Out[x]:
array([[1, 1, 1],
       [1, 1, 1],
       [1, 1, 1]])
```

Если уже существует некоторый массив и необходимо создать другой массив той же формы, то следует воспользоваться методами `np.empty_like`, `np.zeros_like` и `np.ones_like`:

```
In [x]: a
Out[x]: array([100, 101, 102, 103])
In [x]: np.ones_like(a)
Out[x]: array([1, 1, 1, 1])
In [x]: np.zeros_like(a, dtype=float)
Out[x]: array([ 0., 0., 0., 0.])
```

Следует отметить, что создаваемый массив наследует тип данных dtype от исходного массива, поэтому, чтобы изменить тип данных, необходимо использовать аргумент dtype.

Инициализация массива из последовательности

Для создания массива, содержащего последовательность чисел, существуют два метода: np.arange и np.linspace. Метод np.arange равнозначен методу Python range, за исключением того, что он может генерировать последовательности чисел с плавающей точкой. Кроме того, он действительно выделяет память для элементов ndarray, а не возвращает объект, похожий на генератор, – сравните с разделом 2.4.3.

```
In [x]: np.arange(7)
Out[x]: array([0, 1, 2, 3, 4, 5, 6])
In [x]: np.arange(1.5, 3., 0.5)
Out[x]: array([ 1.5,  2. ,  2.5]))
```

Как и при использовании range, массивы, сгенерированные в этих примерах, не включают последние элементы 7 и 3. Но с методом arange связана одна проблема: из-за ограниченной точности арифметики чисел с плавающей точкой не всегда можно узнать, сколько элементов будет создано. По этой причине, а также потому, что часто требуется самый последний элемент определяемой последовательности, метод np.linspace может оказаться более удобным способом создания числовой последовательности[65]. Например, для генерации равномерно распределенных значений элементов массива из пяти чисел между 1 и 20, включая сами эти числа:

```
In [x]: np.linspace(1, 20, 5)
Out[x]: array([  1. ,    5.75,   10.5,   15.25,   20. ])
```

Метод np.linspace может принимать несколько необязательных аргументов с логическими значениями. Если для аргумента retstep задать значение True, то возвращается размер шага в генерируемой последовательности чисел:

```
In [x]: x, dx = np.linspace(0., 2*np.pi, 100, retstep=True)
In [x]: dx
Out[x]: 0.06346651825433926
```

Это помогает избежать отдельного вычисления dx = (end-start)/(num-1). В приведенном выше примере 100 точек между 0 и 2π, включая граничные значения, разделены шагом, равным 2π/99 = 0.0634665….

Установка для аргумента endpoint значения False позволит исключить конечную точку последовательности, как в методе np.arange:

```
In [x]: x = np.linspace(0, 5, 5, endpoint=False)
Out[x]: array([ 0.,  1.,  2.,  3.,  4.])
```

Обратите внимание: массиву, сгенерированному методом np.linspace, присваивается тип dtype чисел с плавающей точкой, даже если последовательность генерирует целые числа.

---

[65] Мы уже встречались с методом linspace при обсуждении примера П3.1.

Инициализация массива из функции

Для создания массива, инициализируемого значениями, которые вычисляются с помощью некоторой функции, используется метод NumPy `np.fromfunction`, принимающий в качестве аргументов эту функцию и кортеж, определяющий форму требуемого массива. Сама функция должна принимать количество аргументов, совпадающее с размерностями создаваемого массива: эти аргументы индексируют каждый элемент, в который функция возвращает значение. Приведенный ниже пример поможет лучше понять применение метода `np.fromfunction`:

```
In [x]: def f(i, j):
   ...:     return 2 * i * j
   ...:
In [x]: np.fromfunction(f,(4,3))
array([[  0.,   0.,   0.],
       [  0.,   2.,   4.],
       [  0.,   4.,   8.],
       [  0.,   6.,  12.]])
```

Функция `f` вызывается для каждого индекса в заданной форме, а возвращаемые значения используются для инициализации соответствующих элементов[66]. Простое выражение, подобное приведенному выше, можно заменить на анонимную лямбда-функцию (см. раздел 4.3.3), если это более предпочтительно:

```
In [x]: np.fromfunction(lambda i,j: 2*i*j, (4,3))
```

Пример П6.1. Для создания «гребенки» значений в массиве длиной *N* каждому *n*-му элементу присваивается значение единица, остальным – нули:

```
In [x]: N, n = 101, 5
In [x]: def f(i):
   ...:     return (i % n == 0) * 1
   ...:
In [x]: comb = np.fromfunction(f, (N,), dtype=int)
In [x]: print(comb)
[1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1
 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0
 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1]
```

Атрибуты *ndarray* для интроспекции

Массив NumPy знает свой ранг, форму, размер, тип `dtype` и одно или два других свойства: их можно определить прямо из атрибутов, описанных в табл. 6.1. Например:

```
In [x]: a = np.array(((1, 0, 1), (0, 1, 0)))
In [x]: a.shape
Out[x]: (2, 3)        # 2 строки, 3 столбца.
In [x]: a.ndim        # Ранг (число измерений).
```

[66] Обратите внимание: индексы передаются как массивы `ndarray`, и ожидается функция `f` для использования векторизованных операций.

```
Out[x]: 2
In [x]: a.size        # Общее количество элементов.
Out[x]: 6
In [x]: a.dtype
Out[x]: dtype('int64')
In [x]: a.data
Out[x]: <memory at 0x102387308 >
```

Таблица 6.1. Атрибуты массива ndarray

Атрибут	Описание
`shape`	Измерения массива: размер массива вдоль каждой из его осей, возвращается как кортеж целых чисел
`ndim`	Количество осей (измерений). Обратите внимание: `ndim == len(shape)`
`size`	Общее количество элементов в массиве, равное произведению элементов кортежа `shape`
`dtype`	Тип данных массива (см. раздел 6.1.2)
`data`	«Буфер» в памяти, содержащий действительные элементы массива
`itemsize`	Размер в байтах каждого элемента

Атрибут `shape` возвращает размеры осей в том порядке, в котором оси индексируются: двумерный массив с `n` строками и `m` столбцами имеет форму (`n`, `m`).

6.1.2 Основные типы данных (dtype) NumPy

До сих пор создаваемые в наших примерах массивы NumPy содержали целые числа или числа с плавающей точкой, и мы позволяли Python взять на себя ответственность о подробностях их представления. Но библиотека NumPy предоставляет мощный инструмент определения этих подробностей в явной форме с помощью объектов типа данных (data type). Это необходимо, потому что для организации взаимодействия (интерфейса) со скомпилированным кодом C более низкого уровня элементы массива NumPy обязательно должны сохраняться в совместимом формате: т. е. каждый элемент представлен в виде фиксированного количества байтов, которые интерпретируются особым способом.

Например, рассмотрим беззнаковое (unsigned) целое число, хранящееся в 2 байтах (16 бит) памяти (тип языка C `uint_16t`). Такое число может принимать значение между 0 и $2^{16} - 1 = 65\,535$. В Python нет собственного равнозначного типа, в точности соответствующего этому представлению: целые числа в Python являются знаковыми величинами, а память для них динамически выделяется по их конкретному размеру. Поэтому NumPy определяет объект типа данных `np.uint16` для описания данных, хранящихся в формате языка C.

Кроме того, в различных системах может отличаться порядок расположения байтов в таких числах – подобное отличие называется «порядок следования байтов» (endianness). При порядке «от старшего к младшему» (big endian) самый значимый (старший) байт размещается по наименьшему адресу памяти. При порядке «от младшего к старшему» (little endian) наименее значимый (младший) байт размещается по наименьшему адресу памяти. При создании пользовательских массивов NumPy будет использовать порядок, принятый по

умолчанию для аппаратуры, на которой выполняется программа, но при этом весьма важно правильно установить порядок следования байтов, если считываемый бинарный файл сгенерирован на другом компьютере.

Полный список числовых типов данных[67] приведен в официальной документации библиотеки NumPy (https://docs.scipy.org/doc/numpy/user/basics.types.html), но наиболее часто используемые перечислены в табл. 6.2. Все эти типы существуют в пакете `numpy`, поэтому на них можно ссылаться, например, так: `np.uint16`. К типам данных, которые создаются по умолчанию при использовании встроенных числовых типов языка Python, добавляется конечный символ подчеркивания: `np.float_`, `np.complex_` и `np.bool_`.

Таблица 6.2

Тип данных	Описание
`int_`	Тип целого числа по умолчанию, соответствующий типу `long` языка C: зависит от платформы
`int8`	Целое число в 1 байте: от –128 до 127
`int16`	Целое число в 2 байтах: от –32 768 до 32 767
`int32`	Целое число в 4 байтах: от –2 147 483 648 до 2 147 483 647
`int64`	Целое число в 8 байтах: от -2^{63} до $2^{63} - 1$
`uint8`	Беззнаковое целое число в 1 байте: от 0 до 255
`uint16`	Беззнаковое целое число в 2 байтах: от 0 до 65 535
`uint32`	Беззнаковое целое число в 4 байтах: от 0 до 4 294 967 295
`uint64`	Беззнаковое целое число в 8 байтах: от 0 до $2^{64} - 1$
`float_`	Тип числа с плавающей точкой по умолчанию, другое имя для типа `float64`
`float32`	Знаковое число с плавающей точкой обычной точности: ~ 10^{-38} до ~ 10^{38} с ~ 7 точными знаками после десятичной точки
`float64`	Знаковое число с плавающей точкой двойной точности: ~ 10^{-308} до ~ 10^{308} с ~ 15 точными знаками после десятичной точки
`complex_`	Тип комплексного числа по умолчанию, другое имя для типа `complex128`
`complex64`	Комплексное число обычной точности (представленное 32-битовыми компонентами с плавающей точкой для действительной и мнимой частей)
`complex128`	Комплексное число двойной точности (представленное 64-битовыми компонентами с плавающей точкой для действительной и мнимой частей)
`bool_`	Логический тип по умолчанию, представленный 1 байтом

Типы данных для чисел с плавающей точкой повышенной точности, такие как `float96`, `float128` и `longdouble`, также доступны, но их надежность вызывает сомнения: реализация этих типов зависит от платформы, поэтому во многих системах они в действительности не обеспечивают заявленную повышенную точность, а просто выравнивают элементы массива по соответствующим границам байтов в памяти.

[67] Строго говоря, эти типы являются типами скалярного массива (array scalar types), а не типами `dtype`, но для нас в контексте данной книги такое различие не имеет значения.

Для создания массива NumPy из элементов конкретного типа данных используется аргумент `dtype` для конструктора любого массива (`np.array`, `np.zeros` и т. д.). В этом аргументе передается либо объект типа данных (например, `np.uint16`), либо значение, которое можно преобразовать в такой объект. Достаточно часто для определения аргумента `dtype` используется строка, состоящая из буквы, определяющей расширенную категорию типа данных (целое, беззнаковое целое, комплексное число и т. д.), за которым следует число, соответствующее количеству байтов в этом типе. Например:

```
In [x]: b = np.zeros((3,3), dtype='u4')
```

Эта инструкция создает массив 3×3 беззнаковых 32-битовых (4-байтовых) целых чисел (равнозначных типу `np.uint32`). Список букв, обозначающих поддерживаемые типы данных, и их значения приведены в табл. 6.3.

Таблица 6.3[68]

Буква	Описание
`i`	Знаковое целое число
`u`	Беззнаковое целое число
`f`	Число с плавающей точкой[67]
`c`	Комплексное число с плавающей точкой
`b`	Логическое значение
`S, a`	Строка (последовательность символов фиксированной длины)
`U`	Unicode

Для определения порядка следования байтов используются префиксы: `>` (big endian – от старшего к младшему), `<` (little endian – от младшего к старшему), `|` (порядок следования байтов не имеет значения). Например:

```
In [x]: a = np.zeros((3,3), dtype='>f8')
In [x]: b = np.zeros((3,3), dtype='<f')
In [x]: c = np.empty((3,3), dtype='|S4')
```

Здесь создаются следующие массивы (соответственно): из чисел двойной точности с порядком следования байтов от старшего к младшему, из чисел обычной точности с порядком следования байтов от младшего к старшему и из строк, содержащих 4 символа (порядок следования байтов не важен).

В приведенных ниже примерах передается строка кода типа (typecode) в конструктор массива в аргументе `dtype`, но также возможно предварительное создание объекта `dtype` и его передача вместо строки со стандартным кодом типа:

```
In [x]: dt = np.dtype('f8')
In [x]: dt
dtype('float64')       # Число с плавающей точкой двойной точности.
In [x]: a = np.array([0., 1., -2.], dtype=dt)
```

[68] Обратите внимание: без определения размера в байтах аргумент `dtype='f'` создает тип данных для числа с плавающей точкой обычной точности, равнозначный типу `np.float32`.

Объекты dtype содержат большой набор полезных методов интроспекции:

```
In [x]: dt.str          # Строка, определяющая тип данных.
'<f8'
In [x]: dt.name         # Имя типа данных и размер в битах.
'float64'
In [x]: dt.itemsize     # Размер типа данных в байтах.
8
```

Для копирования массива в новый массив с другим типом данных передается требуемый аргумент dtype или код типа в метод astype:

```
In [x]: a = np.array([1.2345678, 2.5, 3.9])
In [x]: a.astype('float32')       # Приведение к числу с плавающей точкой обычной точности.
Out[x]: array([1.2345678, 2.5     , 3.9     ], dtype=float32)
In [x]: a.astype(np.uint8)        # Приведение к беззнаковому 1-байтовому целому числу.
Out[x]: array([1, 2, 3], dtype=uint8)
```

Строки в массивах NumPy – это строки байтов (bytestrings) фиксированного размера: каждый «символ» представлен одним байтом, в отличие от кодировки переменного размера UTF-8, обычно применяемой для представления Unicode-строк. Это необходимое условие, потому что массивы NumPy имеют предварительно определенный фиксированный размер, и все элементы занимают одинаковый объем памяти для эффективной индексации с постоянным шагом. Тем не менее строки Unicode кодируются с помощью UTF-8, представляя символы как коды с переменной длиной (см. раздел 2.3.3). Разумеется, в итоге любая строка сохраняется как последовательность байтов, а Python предоставляет методы для преобразования между различными кодировками. Например, в системе, по умолчанию использующей кодировку строк UTF-8:

```
In [x]: s = 'piñata'     # Строка Unicode в кодировке UTF-8.
In [x]: b = s.encode()
In [x]: b
b'pi\xc3\xb1ata'          # Строка байтов bytestring: символ ñ хранится в 2 байтах: hex C3B1.
In [x]: len(s), len(b)
(6,7)                     # Шесть символов в кодировке UTF-8, хранящихся в 7 байтах.
In [x]: arr = np.empty((2,2), 'S7')
In [x]: arr[:] = b        # Сохранение строки байтов (bytestring) b в массиве arr.
In [x]:
array([[b'pi\xc3\xb1ata', b'pi\xc3\xb1ata'],
       [b'pi\xc3\xb1ata', b'pi\xc3\xb1ata']],
      dtype='|S7')
In [x]: arr[0,0]          # Возвращает строку байтов bytestring.
b'pi\xc3\xb1ata'
In [x]: arr[0,0].decode()  # Декодирование строки байтов в строку, предполагая кодировку
                           # UTF-8.
'piñata'
```

6.1.3 Универсальные функции (unifunc)

В дополнение к основным арифметическим операциям сложения, деления и т. п. NumPy предоставляет многие из хорошо знакомых математических функций, которые содержатся в модуле `math` (см. раздел 2.2.2), реализованные в виде так называемых универсальных функций. Универсальные функции обрабатывают каждый элемент массива, возвращая в результате массив без необходимости явного применения цикла. Универсальные функции – это способ, которым библиотека NumPy обеспечивает векторизацию, позволяющую писать ясный, эффективный и удобный для сопровождения код. Например:

```
In [x]: x = np.linspace(1, 5, 5)
In [x]: x**2
Out[x]: array([ 1.,   4.,   9.,  16.,  25.])
In [x]: x - 1
Out[x]: array([ 0.,  1.,  2.,  3.,  4.])
In [x]: np.sqrt(x - 1)
Out[x]: array([ 0.,   1.,   1.41421356,   1.73205081,   2.])
In [x]: y = np.exp(-np.linspace(0., 2., 5))
In [x]: np.sin(x - y)
Out[x]: array([ 0.,   0.98431873,   0.48771645,   -0.59340065,   -0.98842844])
```

Умножение массивов выполняется поэлементно: умножение матриц реализовано с использованием оператора `@`[69] или функции NumPy `dot`:

```
In [x]: a = np.array( ((1, 2), (3, 4)) )
In [x]: b = a
In [x]: a * b         # Поэлементное умножение.
Out[x]:
array([[ 1,  4],
       [ 9, 16]])
In [x]: a @ b         # Умножение матриц; также a.dot(b) или np.dot(a, b).
Out[x]:
array([[ 7, 10],
       [15, 22]])
```

Операторы сравнения и логические операторы (`~`, `&`, `|` для not, and и or соответственно) также векторизованы, а их результаты выводятся в массивах логических значений:

```
In [x]: a = np.linspace(1, 6, 6)**3
In [x]: print(a)
[ 1. 8. 27. 64. 125. 216.]
In [x]: print(a > 100)
[False False False False True True]
In [x]: print((a < 10) | (a > 100))
[ True True False False True True]
```

[69] Оператор `@` появился в версии Python 3.5.

6.1.4 Специальные значения NumPy: `nan` и `inf`

В библиотеке NumPy определены два специальных значения для представления результатов вычислений, которые математически не определены или не являются конечными числами. Значение `np.nan` (NaN – Not a Number – не число) представляет результат вычисления, которое не является корректно определенной математической операцией (например, 0/0). Значение `np.inf` представляет бесконечность[70]. Например:

```
In [x]: a = np.arange(4, dtype='f8')
In [x]: a /= 0      # [0/0 1/0 2/0 3/0]

 ... RuntimeWarning: invalid value encountered in true_divide ...
 ... RuntimeWarning: divide by zero encountered in true_divide ...

In [x]: a
Out[x]: array([ nan,  inf,  inf,  inf])
```

Не следует проверять значения `nan` на равенство (`np.nan == np.nan` дает результат `False`). Вместо этого библиотека NumPy предоставляет методы `np.isnan`, `np.isinf` и `np.isfinite`:

```
In [x]: np.isnan(a)
Out[x]: array([ True, False, False, False], dtype=bool)
In [x]: np.isinf(a)
Out[x]: array([False,  True,  True,  True], dtype=bool)
In [x]: np.isfinite(a)
Out[x]: array([False, False, False, False], dtype=bool)
```

Следует отметить, что значение `nan` не является ни конечным, ни бесконечным и не равно самому себе. (Также см. раздел 10.1.4.)

Пример П6.2. Магический квадрат – это набор («сетка») чисел размером $N \times N$, в котором сумма элементов в каждой строке, столбце и главной диагонали равна одному и тому же числу (равна значению выражения $N(N^2 + 1)/2$). Метод создания магического квадрата для нечетного N описан ниже.

Шаг 1. Начать с середины верхней строки, присвоить $n = 1$.
Шаг 2. Вставить n в текущую позицию квадрата (сетки).
Шаг 3. Если $n = N^2$, то квадрат заполнен, поэтому остановить вычисления. Иначе увеличить n на единицу.
Шаг 4. Переместиться по диагонали вверх и вправо. Если при перемещении происходит выход за границы квадрата, то вернуться в первый столбец или в последнюю строку. Если эта ячейка уже заполнена, то переместиться по вертикали вниз в следующую ячейку.
Шаг 5. Вернуться к шагу 2.

Программа в листинге 6.1 создает и выводит магический квадрат.

---

[70] Эти значения определены в соответствии со стандартом IEEE-754 для чисел с плавающей точкой.

Листинг 6.1. Создание магического квадрата

```
# Создание магического квадрата N×N. N должно быть нечетным числом.
import numpy as np

N = 5
magic_square = np.zeros((N, N), dtype=int)

n = 1
i, j = 0, N//2

while n <= N**2:
    magic_square[i, j] = n
    n += 1
    newi , newj = (i - 1) % N, (j + 1)% N
    if magic_square[newi , newj]:
        i += 1
    else:
        i, j = newi , newj

print(magic_square)
```

С помощью этой программы можно, например, построить магический квадрат 5×5:

```
[[17 24  1  8 15]
 [23  5  7 14 16]
 [ 4  6 13 20 22]
 [10 12 19 21  3]
 [11 18 25  2  9]]
```

6.1.5 Изменение формы массива

Вне зависимости от ранга массива его элементы хранятся в последовательных локациях памяти, адресуемых по единому индексу (внутреннее представление одномерное, но, зная форму массива, Python способен преобразовать кортеж индексов в единый адрес памяти). Массивы NumPy хранятся в памяти в стиле языка C – по строкам (row-major order), т. е. при сохранении элементы упорядочиваются по последнему (самому правому) индексу. Например, в двумерном массиве за элементом `a[0, 0]` следует элемент `a[0, 1]`. Показанный ниже массив

```
In [x]: a = np.array( ((1, 2), (3, 4)) )
In [x]: print(a)
[[1 2]
 [3 4]]
```

сохраняется в памяти как группа последовательных элементов `[1, 2, 3, 4]`[71].

[71] Это полная противоположность порядку хранения элементов массива в Fortran – по столбцам, когда элементы должны сохраняться как `[1, 3, 2, 4]`.

Методы *flatten* и *ravel*

Предположим, что необходимо «выпрямить» многомерный массив вдоль одной оси. Библиотека NumPy предоставляет для этого два метода: `flatten` и `ravel`. Оба метода создают одномерный массив в соответствии с его внутренним порядком элементов (по строкам), как описано выше. Метод `flatten` возвращает независимую копию элементов и в общем случае медленнее, чем метод `ravel`, который пытается вернуть представление преобразованного в одно измерение массива. Представление массива – это новый массив NumPy, который в данном случае имеет форму, отличающуюся от формы исходного массива, но не содержит «собственных» элементов данных: он содержит ссылки на элементы другого массива. Таким образом, как и в изменяемых списках (раздел 2.4.1), изменение (переприсваивание) элементов одного массива влияет на другой массив. Следующий пример поможет понять работу этих двух методов:

```
In [x]: a = np.array( [[1, 2, 3], [4, 5, 6], [7, 8, 9]] )
In [x]: b = a.flatten()     # Создает независимую, одномерную копию массива a.
In [x]: b
Out[x]: array([1, 2, 3, 4, 5, 6, 7, 8, 9])
In [x]: b[3] = 0
In [x]: b
Out[x]: array([1, 2, 3, 0, 5, 6, 7, 8, 9])
In [x]: a         # Массив a не изменяется.
Out[x]:
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

Присваивание массиву `b` не изменяет массив `a`, потому что это абсолютно независимые объекты, которые не используют данные совместно. В отличие от этого подхода, одномерный массив, созданный как представление с помощью метода `ravel`, ссылается на те же самые внутренние данные:

```
In [x]: c = a.ravel()
In [x]: c
Out[x]:array([1, 2, 3, 4, 5, 6, 7, 8, 9])
In [x]: c[3] = 0
In [x]: c
Out[x]: array([1, 2, 3, 0, 5, 6, 7, 8, 9])
In [x]: a
Out[x]:
array([[1, 2, 3],
       [0, 5, 6],
       [7, 8, 9]])
```

Необходимо всегда помнить о том, что хотя метод `ravel` «работает наилучшим образом», возвращая представление внутренних данных, разнообразные операции с массивами (включая вырезание группы элементов (slicing), см. раздел 6.1.6) могут оставлять элементы хранящимися в несмежных локациях памяти. В этом случае у метода `ravel` нет другого выбора, кроме создания копии массива.

Методы *resize* и *reshape*

Размер массива может быть изменен (для самого исходного массива) с приведением к совместимой форме[72] с помощью метода `resize`, который в качестве аргументов принимает новые значения измерений.

```
In [x]: a = np.linspace(1, 4, 4)
In [x]: print(a)
[1. 2. 3. 4.]

In [x]: a.resize(2, 2) # Изменяет форму исходного массива, возвращаемого значения нет.
In [x]: print(a)
[[ 1.  2.]
 [ 3.  4.]]
```

Метод `reshape` возвращает представление исходного массива с изменением формы его элементов в соответствии с требованием. Исходный массив не изменяется, но оба объекта совместно используют внутренние данные.

```
In [x]: a = np.linspace(1, 4, 4)
In [x]: b = a.reshape(2, 2)
In [x]: print(a)
[ 1.  2.  3.  4.]

In [x]: print(b)
[[ 1.  2.]
 [ 3.  4.]]

In [x]: b[0, 0] = -99
In [x]: print(b)
[[-99.  2.
 [  3.  4.]]

In [x]: print(a)
[-99.   2.   3.   4.]
```

Транспонирование массива

Метод `transpose` возвращает представление массива с транспонированными осями. Для двумерного массива это обычная операция транспонирования матрицы:

```
In [x]: a = np.linspace(1, 6, 6).reshape(3, 2)
In [x]: a
Out[x]:
array([[ 1.,  2.],
       [ 3.,  4.],
       [ 5.,  6.]])
In [x]: a.transpose()          # Или просто a.T.
Out[x]:
array([[ 1.,  3.,  5.],
       [ 2.,  4.,  6.]]
```

[72] То есть к форме с тем же общим количеством элементов.

Обратите внимание: транспонирование одномерного массива возвращает этот же неизмененный массив:

```
In [x]: b = np.array([100, 101, 102, 103])
In [x]: b.transpose()
Out[x]: array([100, 101, 102, 103])
```

См. раздел 6.1.11 для получения более подробной информации о представлении векторов с помощью массивов NumPy.

Объединение и разделение массивов

Группа методов NumPy обеспечивает объединение и разделение массивов различными способами. Методы `np.vstack`, `np.hstack` и `np.dstack` объединяют массивы по вертикали (по последовательным строкам), по горизонтали (по последовательным столбцам) и по всем измерениям (вдоль третьей оси). Например:

```
In [x]: a = np.array([0, 0, 0, 0])
In [x]: b = np.array([1, 1, 1, 1])
In [x]: c = np.array([2, 2, 2, 2])
In [x]: np.vstack((a, b, c))
Out[x]:
array([[0, 0, 0, 0],
       [1, 1, 1, 1],
       [2, 2, 2, 2]])
In [x]: np.hstack((a, b, c))
Out[x]:
array([0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 2, 2])
In [x]: np.dstack((a, b, c))
Out[x]:
array([[[0, 1, 2],
        [0, 1, 2],
        [0, 1, 2],
        [0, 1, 2]]])
```

Обратите внимание: созданный массив содержит независимую копию данных из исходных массивов[73].

Противоположные операции `np.vsplit`, `np.hsplit` и `np.dsplit` разделяют один массив на несколько массивов по строкам, столбцам или «в глубину». В дополнение к ссылке на разделяемый массив эти методы требуют аргумента, определяющего, как должен разделяться массив. Если этот аргумент представлен одним целым числом, то массив разделяется на указанное количество массивов равного размера вдоль соответствующей оси. Например:

```
In [x]: a = np.arange(6)
In [x]: a
Out[x]: array([ 0, 1, 2, 3, 4, 5])
In [x]: np.hsplit(a, 3)
Out[x]: [array([ 0, 1]), array([ 2, 3]), array([ 4, 5])]
```

[73] Библиотека NumPy должна копировать данные, потому что итоговые данные должны храниться в одном непрерывном блоке памяти, а исходные массивы могут быть разреженными, т. е. данные могут размещаться в различных несмежных локациях.

Здесь можно видеть, что возвращается список объектов массивов. Если вторым аргументом является последовательность целочисленных индексов, то массив разделяется по этим индексам:

```
In [x]: a
Out[x]: array([ 0, 1, 2, 3, 4, 5])
In [x]: np.hsplit(a, (2, 3, 5))
[array([0, 1]), array([2]), array([3, 4]), array([5])]
```

Здесь последовательность целочисленных аргументов – то же самое, что список `[a[:2], a[2:3], a[3:5], a[5:]]`. В отличие от группы методов `np.hstack` и т. д., возвращаемые массивы являются представлениями исходных данных[74].

Пример П6.3. Предположим, что существует массив 3×3, в который необходимо добавить строку или столбец. Строка добавляется просто с помощью метода np.vstack:

```
In [x]: a = np.ones((3, 3))
In [x]: np.vstack( (a, np.array((2, 2, 2))) )
Out[x]:
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 2.,  2.,  2.]])
```

Но добавление столбца потребует несколько большего объема работы. Невозможно напрямую использовать метод `np.hstack`:

```
In [x]: a = np.ones((3, 3))
In [x]: np.hstack( (a, np.array((2, 2, 2))) )

... [Traceback information] ...
ValueError: all the input arrays must have same number of dimensions
            (все исходные массивы должны иметь одинаковое число измерений)
```

Причина ошибки в том, что метод `np.hstack` не может объединить массивы с различным количеством строк. В схематическом отображении это выглядит так:

```
[[ 1., 1., 1.],        [2., 2., 2.]
 [ 1., 1., 1.],    +                = ?
 [ 1., 1., 1.]]
```

Также невозможно транспонировать новую заданную строку, поскольку это одномерный массив, а его транспонирование дает ту же форму, что исходный массив. Поэтому сначала необходимо явно изменить форму одномерного массива:

---

[74] В NumPy это делается по соображениям эффективности – копирование больших объемов данных связано со значительными накладными расходами, а кроме того, нет необходимости в копировании данных для этих методов разделения массивов.

```
In [x]: a = np.ones((3, 3))
In [x]: b = np.array((2, 2, 2)).reshape(3, 1)
In [x]: b
array([[2],
       [2],
       [2]])
In [x]: np.hstack((a, b))
Out[x]:
array([[ 1.,  1.,  1.,  2.],
       [ 1.,  1.,  1.,  2.],
       [ 1.,  1.,  1.,  2.]])
```

6.1.6 Индексация и вырезание группы элементов из массива

Массив индексируется кортежем целых чисел, и, как для последовательностей Python, отрицательные индексы отсчитываются от конца оси. Вырезание и определение шага для него также поддерживаются аналогичным образом. Но следует особо отметить, что операция вырезания группы элементов из массива NumPy возвращает представление (view), а не копию данных, как для списков Python. Для одномерных массивов существует только один индекс:

```
In [x]: a = np.linspace(1, 6, 6)
In [x]: print(a)
[ 1.  2.  3.  4.  5.  6.]
In [x]: a[1:4:2]    # Элементы a[1] и a[3] (с шагом 2).
Out[x]: array([ 2.,  4.])
In [x]: a[3::-2]    # Элементы a[3] и a[1] (с шагом -2).
Out[x]: array([ 4.,  2.]
```

Для многомерных массивов индекс существует по каждой оси. Если необходимо выбрать каждый элемент по конкретной оси, то нужно заменить ее индекс на один столбец:

```
In [x]: a = np.linspace(1, 12, 12).reshape(4, 3)
In [x]: a
Out[x]:
array([[  1.,  2.,  3.],
       [  4.,  5.,  6.],
       [  7.,  8.,  9.],
       [ 10., 11., 12.]])
In [x]: a[3, 1]
Out[x]: 11.0
In [x]: a[2, :]       # Все элементы в третьей строке.
Out[x]:
array([ 7.,  8.,  9.])
In [x]: a[:, 1]       # Все элементы во втором столбце.
Out[x]: array([ 2.,  5.,  8.,  11.])
In [x]: a[1:-1, 1:]    # Средние строки, начиная со второго столбца.
Out[x]:
array([[ 5.,  6.],
       [ 8.,  9.]])
```

Этот и все последующие примеры вырезания элементов из массивов NumPy графически изображены на рис. 6.1.

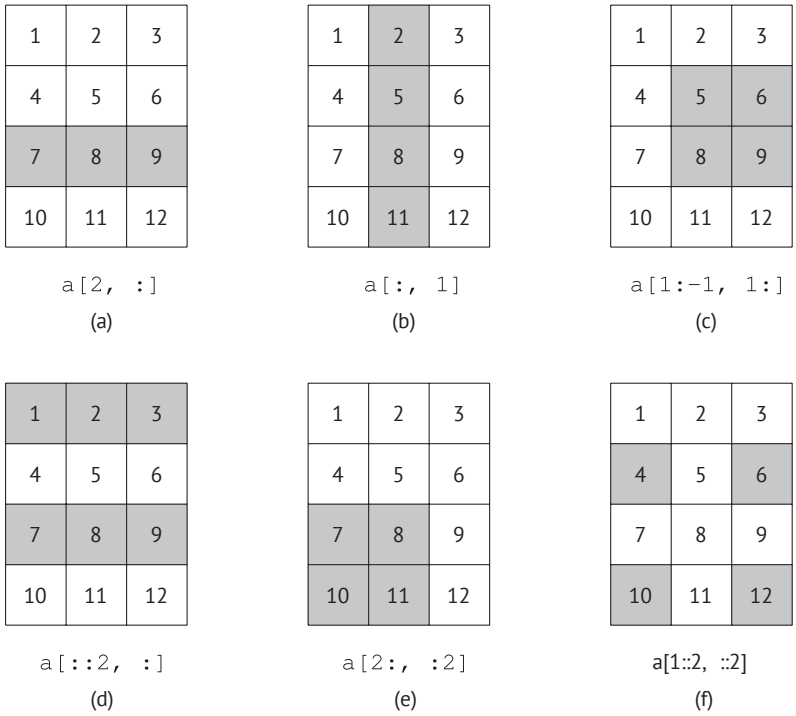


Рис. 6.1. Различные способы вырезания группы элементов из массива NumPy

Специализированная форма записи в виде многоточия (...) удобна для массивов высокого ранга: в индексе она представляет столько столбцов, сколько необходимо для указания всех оставшихся осей. Например, для четырехмерного массива `a[3, 1, ...]` равнозначно `a[3, 1, :, :]`, а `a[3, ..., 1]` равнозначно `a[3, :, :, 1]`.

Синтаксис с использованием символов двоеточия и многоточия работает и для присваивания:

```
In [x]: a[:, 1] = 0        # Для всех элементов во втором столбце установить значение ноль.
In [x]: print(a)
[[  1.  0.   3.]
 [  4.  0.   6.]
 [  7.  0.   9.]
 [ 10.  0.  12.]]
```

Более сложное индексирование

Массивы NumPy можно также индексировать последовательностями, которые не являются простыми кортежами целых чисел, а именно другими списками, массивами целых чисел и кортежами кортежей. Такое «продвинутое индексирование» создает новый массив с собственной копией исходных данных, а не представление:

```
In [x]: a = np.linspace(0., 0.5, 6)
In [x]: print(a)
[ 0.  0.1  0.2  0.3  0.4  0.5]
In [x]: ia = [1, 4, 5]       # Список индексов.
In [x]: print(a[ia])
[ 0.1  0.4  0.5]
In [x]: ia = np.array( ((1, 2), (3, 4)) )
In [x]: print(a[ia])         # Массив, формируемый по заданным индексам.
[[ 0.1  0.2]
 [ 0.3  0.4]]
```

Можно даже проиндексировать многомерный массив с помощью многомерных массивов индексов, произвольно выбирая отдельные элементы для создания массива заданной формы. Такой подход может привести к написанию весьма причудливого кода:

```
In [x]: a = np.linspace(1, 12, 12).reshape(4, 3)
In [x]: print(a)
[[  1.   2.   3.]
 [  4.   5.   6.]
 [  7.   8.   9.]
 [ 10.  11.  12.]]
In [x]: ia = np.array( ((1, 0), (2, 1)) )
In [x]: ja = np.array( ((0, 1), (1, 2)) )
In [x]: print(a[ia, ja])
[[ 4.   2.]
 [ 8.   6.]]
```

Здесь создается массив 2×2 (форма, определяемая массивами индексов) с элементами `a[1,0]`, `a[0,1]` в верхней строке и элементами `a[2,1]`, `a[1,2]` в нижней строке.

Вместо индексации массива с помощью последовательности целых чисел также возможно использование массива логических значений. Элементы `True` такого индексного массива обозначают элементы целевого массива, которые необходимо вернуть:

```
In [x]: a = np.array([-2, -1, 0, 1, 2])
In [x]: ia = np.array([False, True, False, True, True])
In [x]: print(a[ia])
[-1  1  2]
```

Поскольку операции сравнения векторизованы по элементам массивов точно так же, как математические операции, становится возможным применение некоторых полезных ускоренных приемов:

```
In [x]: print(a)
[-2  -1  0  1  2]
In [x]: ib = a < 0
In [x]: print(ib)
[ True  True False False False]
In [x]: a[ib] = 0      # Замена всех отрицательных элементов нулями.
In [x]: print(a)
[0 0 0 1 2]
```

В действительности нет необходимости сохранять вспомогательный массив логических значений `ib`, так как инструкция `a[a < 0] = 0` выполняет ту же самую работу:

```
In [x]: a = np.array([-2, -1, 0, 1, 2])
In [x]: a[a < 0] = 0
In [x]: print(a)
[0 0 0 1 2]
```

Логические операции `not`, `and` и `or` реализованы в логических массивах с помощью операторов `~`, `&` и `|` соответственно. Например:

```
In [x]: years = np.array([1900, 1904, 1990, 1993, 2000, 2014, 2016, 2100])
In [x]: leap_year = (years % 400 == 0) | (years % 4 == 0) & ~(years % 100 == 0)
In [x]: print(list(zip(years, leap_year)))
Out[x]: [(1900, False), (1904, True), (1990, False), (1993, False),
         (2000, True), (2014, False), (2016, True), (2100, False)]
```

Добавление оси

Для добавления оси (т. е. измерения) в массив выполняется вставка `np.newaxis` в требуемой позиции:

```
In [x]: a = np.linspace(1, 4, 4).reshape(2, 2)
In [x]: print(a)     # Массив a 2×2 (ранг = 2).
[[ 1.  2.]
 [ 3.  4.]]
In [x]: a.shape()
(2, 2)
In [x]: b = a[:, np.newaxis, :]
In [x]: print(b)     # Массив a 2×1×2 (ранг = 3).
[[[ 1.  2.]]

 [[ 3.  4.]]]
In [x]: b.shape
(2, 1, 2)
```

В действительности `np.newaxis` – это объект `None`, поэтому `None` можно использовать напрямую в этом месте, если необходимо.

Пример П6.4. Квадрат судоку (Sudoku) состоит из 9×9 ячеек с таким их содержимым, что в каждой строке, столбце и в каждой из 9 непересекающихся секций 3×3 числа от 1 до 9 содержатся только один раз. Программа в листинге 6.2 проверяет, является ли предлагаемая сетка правильным квадратом судоку.

Листинг 6.2. Проверка правильности квадрата судоку

```
import numpy as np

def check_sudoku(grid):
    """ Return True if grid is a valid Sudoku square , otherwise False. """
    # """ Возвращает True, если это правильный квадрат судоку, иначе возвращает False. """
    for i in range(9):
        # j, k индексируют верхний левый угол каждой секции 3×3.
```

```
        j, k = (i // 3) * 3, (i % 3) * 3
        if len(set(grid[i,:])) != 9 or len(set(grid[:,i])) != 9\                    ❶
                  or len(set(grid[j:j+3, k:k+3].ravel())) != 9:
            return False
    return True

sudoku = """145327698
            839654127
            672918543
            496185372
            218473956
            753296481
            367542819
            984761235
            521839764"""
# Преобразование заданной строки sudoku в массив целых чисел.
grid = np.array([[int(i) for i in line] for line in sudoku.split()])
print(grid)

if check_sudoku(grid):
    print('grid valid')
else:
    print('grid invalid')
```

❶ Здесь используется тот факт, что массив длиной 9 содержит девять неповторяющихся элементов, если множество, сформированное из этих элементов, имеет мощность 9. Не выполняется никаких проверок того, что сами элементы этого множества действительно являются числами от 1 до 9.

Сетки

Для вычислений многомерной функции на решетке, состоящей из множества точек, удобно применение сетки (mesh). В функцию `np.meshgrid` передается последовательность *N* одномерных массивов, представляющих координаты по каждому измерению, а функция возвращает набор *N*-мерных массивов, формирующих сетку координат, в которых должна быть вычислена заданная функция. Например, для случая с двумя измерениями:

```
In [x]: x = np.linspace(0, 5, 6)
In [x]: y = np.linspace(0, 3, 4)
In [x]: X, Y = np.meshgrid(x, y)
In [x]: X
Out[x]:
array([[ 0.,  1.,  2.,  3.,  4.,  5.],
       [ 0.,  1.,  2.,  3.,  4.,  5.],
       [ 0.,  1.,  2.,  3.,  4.,  5.],
       [ 0.,  1.,  2.,  3.,  4.,  5.]])
In [x]: Y
Out[x]:
array([[ 0.,  0.,  0.,  0.,  0.,  0.],
       [ 1.,  1.,  1.,  1.,  1.,  1.],
       [ 2.,  2.,  2.,  2.,  2.,  2.],
       [ 3.,  3.,  3.,  3.,  3.,  3.]])
```

Каждый массив X и Y можно проиндексировать с помощью индексов i, j: массив x повторяется как строки, последовательно («сверху вниз») формирующие массив X, а массив y заполняет столбцы массива Y. Таким образом, функцию от двух координат можно вычислить на этой сетке просто как f(X, Y).

Установка значения True для необязательного аргумента sparse приведет к возврату разреженной сетки для экономии памяти. В приведенном выше примере вместо двух массивов, имеющих общую форму (6, 4) для массивов с формами (1, 6) и (4, 1), можно выполнить операцию бродкастинга для этих массивов (см. раздел 6.1.7) с возвратом результата:

```
In [X]: X, Y = np.meshgrid(x, y, sparse=True)
In [X]: X
Out[X]: array([[ 0.,  1.,  2.,  3.,  4.,  5.]])
In [X]: Y
Out[X]:
array([[ 0.],
       [ 1.],
       [ 2.],
       [ 3.]])
```

6.1.7 ◊ Бродкастинг

Ранее мы уже видели, что простые операции, такие как сложение и умножение, могут выполняться поэлементно в двух массивах одинаковой формы (поддержка векторизации):

```
In [x]: a = np.array([1, 2, 3])
In [x]: b = np.array([0, 10, 100])
In [x]: a * b
Out[x]: array([ 0, 20, 300])
```

Бродкастинг (broadcasting) описывает правила, используемые библиотекой NumPy для выполнения операций, в которых массивы имеют различные формы. Это позволяет выполнить операцию с применением предварительно скомпилированных циклов языка C вместо более медленных циклов Python, но при этом существуют ограничения, согласно которым формы массивов могут быть приведены в соответствие друг другу. Эти правила применяются к каждому измерению массивов, начиная с последнего, и далее работают в обратном направлении (к начальному измерению). Два измерения, сравниваемых таким способом, называются совместимыми (compatible), если оба они равны или одно из них равно 1.

Простейшим примером бродкастинга является операция между массивом и скалярным значением (которое в рассматриваемом случае может считаться одномерным массивом длиной 1). Рассмотрим код такого примера:

```
In [x]: a = np.array([[1, 2, 3], [4, 5, 6]])
In [x]: b = 2
In [x]: c = a * b
In [x]: c
Out[x]:
array([[ 2,  4,  6],
       [ 8, 10, 12]])
```

Измерения массивов a и b являются совместимыми:

```
a:     2 x 3
b:         1
c:     2 x 3
```

Здесь для b может быть выполнен бродкастинг по двум измерениям массива a посредством повторения значения b для каждого элемента массива a. Подобным образом для массива формы (2,) может быть выполнен бродкастинг по обеим строкам массива a:

```
In [x]: b = np.array([1, 2, 3])
In [x]: c = a * b
In [x]: c
Out[x]:
array([[ 1,  4,  9],
       [ 4, 10, 18]])

a:     2 x 3
b:         3
c:     2 x 3
```

Таким образом, в каждой строке массива a ее элементы умножаются на соответствующие элементы одномерного массива b. Но попытка умножения массива a на массив, последнее измерение которого не равно 1 или 3, приведет к ошибке ValueError:

```
In [x]: b = np.array([1, 2])
In [x]: a * b

---------------------------------------------------------------------------
...
----> 1 a * b

ValueError: operands could not be broadcast together with shapes (2,3) (2,)
            (для операндов невозможно выполнить бродкастинг при формах (2,3) (2,))
```

В примере с разреженной сеткой, созданной в предыдущем разделе, массивы с формами (1, 6) и (4, 1) являются совместимыми. Например:

```
In [x]: f = X * Y
Out[x]: f
array([[  0.,   0.,   0.,   0.,   0.,   0.],
       [  0.,   1.,   2.,   3.,   4.,   5.],
       [  0.,   2.,   4.,   6.,   8.,  10.],
       [  0.,   3.,   6.,   9.,  12.,  15.]])
```

Процесс бродкастинга «растягивает» вторую ось массива Y от 1 до 6, чтобы обеспечить соответствие массиву X, а первую ось массива X – от 1 до 4 для соответствия массиву Y:

```
X:     1 x 6
Y:     4 x 1
f:     4 x 6
```

Для принудительного применения бродкастинга к массиву с недостаточным числом измерений, чтобы он соответствовал требованиям, всегда можно добавить ось с помощью метода `np.newaxis`. Например, одним из способов получения векторного произведения двух массивов является добавление измерения в один из них и бродкастинг при умножении:

```
In [x]: a = np.array([1, 2, 3])
In [x]: b = np.array([0, 10, 100])
In [x]: c = a[:, np.newaxis] * b
In [x]: c
Out[x]:
array([[  0,   20,  300],
       [  0,   40,  600],
       [  0,   60,  900]])
```

Здесь вместо обеспечения соответствия элементов в двух массивах форме `(3, )` добавление дополнительной оси в `a` создает массив формы `(3, 1)`, и это измерение растягивается вдоль массива `b`:

```
a[:,np.newaxis]:   3 x 1
              b:       3
              c:   3 x 3
```

6.1.8 Максимальные и минимальные значения

Для массивов NumPy существуют методы `min` и `max`, возвращающие минимальные и максимальные значения в массиве. По умолчанию возвращается одно значение из массива, преобразованного в одномерный. Для поиска максимальных и минимальных значений в заданной оси используется аргумент `axis`:

```
In [x]: a = np.array([[3, 0, -1, 1], [2, -1, -2, 4], [1, 7, 0, 4]])
In [x]: print(a)
[[ 3  0 -1  1]
 [ 2 -1 -2  4]
 [ 1  7  0  4]]
In [x]: a.min()     # "Глобальный" минимум.
Out[x]: -2
In [x]: a.max()     # "Глобальный" максимум.
Out[x]: 7
In [x]: print( a.min(axis=0) )
[ 1 -1 -2 1]        # Минимальные значения в каждом столбце.
In [x]: print( a.max(axis=1) )
[3 4 7]             # Максимальные значения в каждой строке.
```

Часто требуется не само максимальное (или минимальное) значение, а его индекс в массиве. Для этого существуют методы `argmin` и `argmax`. По умолчанию возвращаемый индекс вычисляется в массиве, преобразованном в одномерный, поэтому действительное значение индекса можно извлечь, используя представление массива, созданное методом `ravel`:

```
In [x]: a.argmin()
6
In [x]: a.ravel()[a.argmin()]
-2
In [x]: print(a.argmax(axis=0))
[0 2 2 1]       # Индексы по строкам для максимальных значений в каждом столбце.
In [x]: print(a.argmax(axis=1))
[0 3 1]         # Индексы по столбцам для максимальных значений в каждой строке.
```

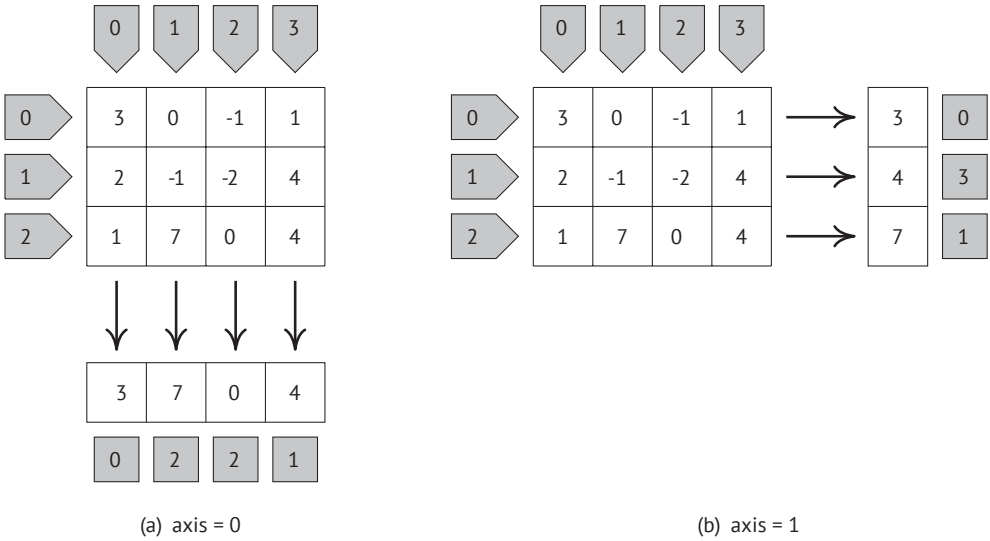


Рис. 6.2. а) `a.max(axis=0)` возвращает максимальные значения, а `a.argmax(axis=0)` возвращает индексы этих максимальных значений в массиве a (т. е. обрабатывает измерение «строка»); б) то же самое для `axis=1`: максимальные значения по каждой строке

На рис. 6.2 показан процесс для `axis=0` и для `axis=1`. Обратите внимание: если в столбце существует несколько равных максимальных значений, то возвращается индекс первого из них.

Пример П6.5. Рассмотрим следующие осциллирующие функции в интервале [0, L]:

$$f_n(x) = x(L - x)\sin(2\pi x/\lambda_n); \quad \lambda_n = 2L/n, \quad n = 1, 2, 3, \ldots.$$

Код в листинге 6.3 определяет двумерный массив, содержащий значения этих функций для $L = 1$ на сетке из $N = 100$ точек (строк) для $n = 1, 2, \ldots, 5$ (столбцов). Положение максимума и минимума в каждом столбце вычисляется с помощью методов `argmax(axis=0)` и `argmin(axis=0)`. (См. рис. 6.3.)

Листинг 6.3. Применение методов argmax и argmin

```
# eg6-array_maxmin.py
import numpy as np
import matplotlib.pyplot as plt

N = 100
L = 1
```

```
def f(i, n):
    x = i * L / N
    lam = 2 * L / (n+1)
    return x * (L-x) * np.sin(2*np.pi*x/lam)

a = np.fromfunction(f, (N+1, 5))
min_i = a.argmin(axis=0)
max_i = a.argmax(axis=0)
plt.plot(a, c='k')
plt.plot(min_i, a[min_i, np.arange(5)], 'v', c='k', markersize=10)
plt.plot(max_i, a[max_i, np.arange(5)], '^', c='k', markersize=10)
plt.xlabel(r'$x$')
plt.ylabel(r'$f_n(x)$')
plt.show()
```

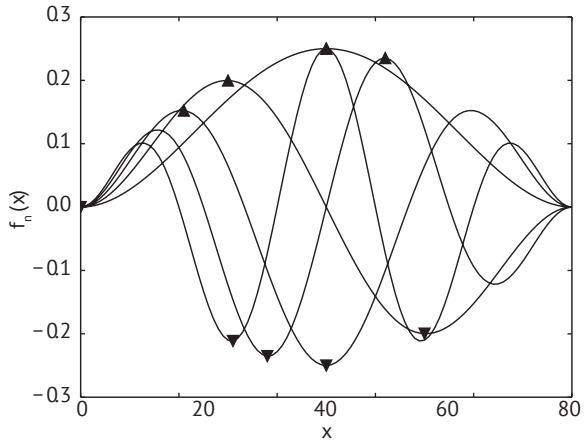


Рис. 6.3. Максимальные и минимальные значения функций $f_n(x)$, описанных в примере П6.5. Обратите внимание: для каждой функции возвращается только «глобальное» максимальное и минимальное значение, а если существует несколько точек с одинаковым максимальным или минимальным значением, то возвращается только первая точка

6.1.9 Сортировка массива

Массивы NumPy можно сортировать несколькими различными способами с помощью метода `sort`, упорядочивающего элементы в самом массиве. По умолчанию этот метод сортирует многомерные массивы по самой последней оси. Для сортировки по любой другой оси определяется аргумент `axis`. Например:

```
In [x]: a = np.array([5, -1, 2, 4, 0, 4])
In [x]: a.sort()
In [x]: print(a)
[-1  0  2  4  4  5]
In [x]: b = np.array([[0, 3, -2], [7, 1, 3], [4, 0, -1]])
In [x]: print(b)
[[ 0  3 -2]
 [ 7  1  3]
 [ 4  0 -1]]
In [x]: b.sort()         # Сортировка чисел по каждой строке.
```

```
In [x]: print(b)
[[-2  0  3]
 [ 1  3  7]
 [-1  0  4]]
```

Это то же самое, что и `b.sort(axis=1)`, – «для каждой строки упорядочить числа по столбцам». Для сортировки чисел в каждом столбце – «для каждого столбца упорядочить числа по строкам» – устанавливается значение аргумента `axis=0`:

```
In [x]: b = np.array([[0, 3, -2], [7, 1, 3], [4, 0, -1]])
In [x]: b.sort(axis=0)    # Сортировка чисел по каждому столбцу.
In [x]: print(b)
[[0  0 -2]
 [4  1 -1]
 [7  3  3]]
```

Для сортировки используется алгоритм quicksort (быстрая сортировка) – неплохой вариант в общем случае[75].

Заслуживают внимания и две другие функции сортировки. Метод `np.argsort` возвращает индексы, которые должны получиться при сортировке массива, вместо самих отсортированных элементов:

```
In [x]: a = np.array([3, 0, -1, 1])
In [x]: np.argsort(a)
Out[x]: array([2, 1, 3, 0])
```

Таким образом:

```
In [x]: a[np.argsort(a)]
Out[x]: array([-1, 0, 1, 3])
```

Метод `np.searchsorted` принимает отсортированный массив `a` и одно или несколько значений `v` и возвращает индексы в массиве `a`, по которым должны быть введены эти значения для сохранения упорядоченности массива:

```
In [x]: a = np.array([1, 2, 3, 4])
In [x]: np.searchsorted(a, 3.5)
Out[x]: 3
In [x]: np.searchsorted(a, (3.5, 0, 1.1))
Out[x]: array([3, 0, 1])
```

6.1.10 Структурированные массивы

Структурированные массивы, известные также как массивы записей, – это массивы, состоящие из строк значений, при этом каждое значение может иметь собственный тип данных и имя. Такие строки называют записями (records).

[75] Некоторые массивы можно отсортировать быстрее с помощью алгоритмов mergesort (сортировка слиянием) или heapsort (пирамидальная, или древовидная, сортировка). Эти алгоритмы можно выбрать, установив для необязательного аргумента `kind` значение строкового литерала `'mergesort'` или `'heapsort'`, например `b.sort(axis=1, kind='heapsort')`.

Этот тип массива очень похож на таблицу данных со строками (записями), состоящими из значений, разделенных на столбцы (поля), и предоставляет весьма удобный и естественный способ обработки научных данных, которые часто получены или представлены в табличной форме.

Структурированные массивы удобны при обработке небольших наборов неоднородных данных, но их функциональность доступна и на более высоком уровне в библиотеке pandas (см. главу 9), которая чаще всего более подходит для обработки больших наборов данных.

Создание структурированного массива

Структура массива записей определяется его типом данных `dtype` с использованием более сложного синтаксиса по сравнению с тем, которые мы видели ранее. Например:

```
In [x]: a = np.zeros(5, dtype='int8 , float32 , complex_')
In [x]: print(a)
[(0, 0.0, 0j) (0, 0.0, 0j) (0, 0.0, 0j) (0, 0.0, 0j) (0, 0.0, 0j)]
In [x]: a.dtype
dtype([('f0', '|i1'), ('f1', '<f4'), ('f2', '<c16')])
```

Здесь создан массив из пяти записей, каждая из которых содержит три поля, определяемых составным типом данных `dtype` с помощью строки `'int8, float32, complex_'`:

- первое поле – однобайтовое знаковое целое число (`int8`, описанное строкой `'|i1'`, – очевидно, что порядок следования байтов не имеет значения для однобайтового числа);
- второе поле – число с плавающей точкой обычной точности, сохраняемое в памяти (в моей системе) как 4-байтовая последовательность с порядком следования от младшего к старшему (little endian), определенное как `'<f4'`;
- третье поле – комплексное число с точностью по умолчанию, которое в моей системе хранится как 16-байтовая последовательность с порядком следования от младшего к старшему (`complex_` равнозначно `complex128` и соответствует типу данных `'<c16'`).

Поскольку имена полей не заданы явно, им присваиваются имена по умолчанию `'f0'`, `'f1'` и `'f2'`. Для явного именования полей структурного массива необходимо передать в конструктор dtype список кортежей (*`name`*`, dtype `*`descriptor`*), например:

```
In [x]: dt = np.dtype( [('time', 'f8'), ('signal', 'i4')] )
In [x]: a = np.zeros(10, dtype=dt)
In [x]: a
Out[x]:
array([(0.0, 0), (0.0, 0), ..., (0.0, 0)],
      dtype=[('time', '<f8'), ('signal', '<i4')])
```

Таким образом, можно выполнить визуализацию структурированного массива в виде таблицы значений данных с заголовками столбцов для каждого поля.

Присваивание записей в структурированном массиве выполняется вполне ожидаемо:

```
In [x]: a[0] = (0., 4)
In [x]: a[1:3] = [(0.5, -3), (1., -5)]
In [x]: a
Out[x]:
array([(0.0, 4), (0.5, -3), (1.0, -5), ..., (0.0, 0)],
      dtype=[('time', '<f8'), ('signal', '<i4')])
```

Но истинная мощь этой методики заключается в возможности ссылки на поле по его имени. Например, для установки значений столбца `'time'` в только что созданном массиве с помощью линейной последовательности:

```
In [x]: a['time'] = np.linspace(0., 4.5, 10)
In [x]: print(a)
[(0.0, 4) (0.5, -3) (1.0, -5) (1.5, 0) (2.0, 0) (2.5, 0) (3.0, 0) (3.5, 0) (4.0, 0) (4.5,
0)]
In [x]: print(a['time'][-1])
4.5
```

Аналогично для получения представления столбца можно обращаться к нему по имени:

```
In [x]: print(a['time'])
[ 0.   0.5  1.   1.5  2.   2.5  3.   3.5  4.   4.5]
In [x]: print( a['signal'].min() )
-5
```

Другие способы создания структурированного массива

Существует несколько (возможно, слишком много) способов определения типа `dtype`, описывающего структурированный массив. До сих пор мы использовали строку идентификаторов, разделенных запятыми, и список кортежей. Третий способ – использование словаря. Самый простой вариант применения – присваивание списка значений двум ключам `'names'` и `'formats'` – первый содержит имена полей, второй определяет их форматы:

```
In [x]: dt = np.dtype({ 'names': ['time', 'signal'],
                        'formats': ['f8', 'i4']
                      })
In [x]: a = np.zeros(10, dtype=dt)
```

Здесь определяется тот же структурированный массив записей (`time, signal`), что и в предыдущем разделе. Можно использовать третий ключ `'titles'` для более подробного описания каждого поля. В дальнейшем каждый заголовок (title) можно применять как псевдоним, соответствующий имени, для ссылки на это поле в массиве[76].

[76] В действительности title может быть любым объектом Python и использоваться для предоставления подробных «метаданных», относящихся к соответствующему полю.

```
In [x]: dt = np.dtype({'names': ['candidate', 'mark', 'grade'],
                       'formats': ['|S50', 'u1', '|S2'],
                       'titles': ['Candidate Name', 'Percentage Mark', 'Grade: A-F']})
In [x]: a = np.zeros(10, dtype=dt)
In [x]: a[0] = ('John Brown', 64, 'B-')
In [x]: a[1] = ('Jane Smith', 78, 'A')
In [x]: print(a['Candidate Name'])
[b'John Brown' b'Jane Smith' b'' b'' b'' b'' b'' b'' b'' b'']
In [x]: print(a['Percentage Mark'])
[64 78 0 0 0 0 0 0 0 0]
```

Сортировка структурированных массивов

Структурированные массивы можно сортировать, определяя требуемый порядок, по полям, используя для этого аргумент `order`. Например, отсортируем следующий структурированный массив:

```
In [x]: data = [ ('NiCd', 1.2, 0.14, 2000),
                 ('Lead acid', 2.1, 0.14, 700),
                 ('Lithium ion', 3.6, 0.46, 800) ]
In [x]: dtype = [ ('name', '|S20'),
                  ('voltage', 'f8'),
                  ('specific energy', 'f8'),
                  ('cycle durability', 'i4') ]
In [x]: a = np.array(data, dtype=dtype)
In [x]: a.sort(order='specific energy')
In [x]: print(a)
[(b'Lead acid', 2.1, 0.14, 700) (b'NiCd', 1.2, 0.14, 2000)
 (b'Lithium ion', 3.6, 0.46, 800)]

In [x]: a.sort(order=['specific energy', 'voltage'])
In [x]: print(a)
[(b'NiCd', 1.2, 0.14, 2000) (b'Lead acid', 2.1, 0.14, 700)
 (b'Lithium ion', 3.6, 0.46, 800)]
```

Здесь вторая операция сортирует записи по полю `'specific energy'`, а если значения одинаковы в двух и более записях, то выполняется дополнительная сортировка по полю `'voltage'`.

6.1.11 Массивы как векторы

Вектор с n компонентами может быть определен как обычный одномерный массив с n элементами.

В дополнение к поэлементным операциям, таким как сложение, вычитание векторов и т. д., для объектов массивов NumPy реализованы методы скалярного (dot) и векторного (cross) произведений:

```
In [x]: a = np.array([1, 0, -3])       # Вектор как одномерный массив.
In [x]: b = np.array([2, -2, 5])
In [x]: a.dot(b)                       # Или a @ b, или b.dot(a), или np.dot(a,b).
Out[x]: -13
In [x]: np.cross(a, b)
array([ -6, -11, -2])
```

Векторное произведение можно получить только для массива с двумя или тремя элементами; в рассматриваемом здесь примере предполагается, что третий компонент равен нулю. При использовании методов `dot` и `cross` для двух отдельных векторов необходимо убедиться в том, что они действительно являются векторами-строками, как показано выше, а не векторами-столбцами, представленными как массивы формы `(n, 1)`:

```
In [x]: a = np.array([[1], [0], [-3]])     # a 3×1 двумерный массив.
In [x]: b = np.array([[2], [-2], [5]])
In [x]: print(a)
[[ 1]
 [ 0]
 [-3]]
In [x]: np.dot(a,b)      # Попытка умножения матриц; не работает.
...

ValueError: shapes (3,1) and (3,1) not aligned: 1 (dim 1) != 3 (dim 0)
```

Если необходимо выполнить скалярное умножение двух векторов-столбцов с помощью метода `np.dot`, то они должны быть преобразованы в векторы-строки:

```
In [x]: np.dot(a.T[0], b.T[0])      # Транспонирование в векторы-строки.
Out[x]: -13
```

Это слегка запутанный прием: необходим индекс, потому что после транспонирования двумерного массива `(n, 1)` получается массив `(1, n)`, из которого требуется только первая строка для представления исходного вектора. При другом подходе можно работать, используя одномерное представление столбцов векторов, полученное с помощью метода `ravel`:

```
In [x]: a.ravel() @ b.ravel()      # То же самое, что a.ravel().dot(b.ravel())
Out[x]: -13
```

Для преобразования вектора-строки, представленного одномерным массивом формы `(n,)`, в вектор-столбец формы `(n,1)` необходимо добавить ось:

```
In [x]: r = np.array([3, 4, 5])
In [x]: c = r[:, np.newaxis]
In [x]: c
array([[3],
       [4],
       [5]])
```

6.1.12 Логические операции и операции сравнения

Библиотека NumPy предоставляет набор методов для сравнения и выполнения логических операций поэлементно в массивах. Наиболее полезные методы кратко описаны в табл. 6.4.

Таблица 6.4. Методы сравнения для массивов ndarray

Метод	Описание
`np.all(a)`	Определяет, все ли элементы массива a при вычислении дают результат `True`
`np.any(a)`	Определяет, дает ли каждый элемент массива a при вычислении результат `True`
`np.isreal(a)`	Проверка: является ли каждый элемент массива a действительным числом
`np.iscomplex(a)`	Проверка: является ли каждый элемент массива a комплексным числом
`np.isclose(a, b)`	Возвращает массив логических значений как результатов сравнений элементов массивов a и b на равенство в пределах некоторого допустимого отклонения
`np.allclose(a, b)`	Возвращает `True`, если все элементы массивов a и b равны в пределах некоторого допустимого отклонения

Методы `np.all` и `np.any` работают так же, как встроенные функции Python с теми же именами[77] (см. раздел 2.4.3):

```
In [x]: a = np.array([[1, 2, 0, 3], [4, 0, 1, 1]])
In [x]: np.any(a), np.all(a)
Out[x]: (True , False)    # Некоторые (но не все) элементы равнозначны True.
```

Методы `np.isreal` и `np.iscomplex` возвращают массивы логических значений:

```
In [x]: b = np.array([1, -1j, 0.5j, 0, 1-2.5j])
In [x]: np.isreal(b)
Out[x]: array([ True, False, False,  True, False], dtype=bool)
In [x]: np.iscomplex(b)
Out[x]: array([False,  True,  True, False,  True], dtype=bool)
```

Поскольку представление чисел с плавающей точкой не является точным, сравнение двух массивов типа `float` или `complex` с помощью оператора `==` не всегда надежно и потому не рекомендуется. Лучшее, что можно предпринять в этом случае, – сравнение двух значений на «близость» друг к другу в пределах некоторого (обычно малого) абсолютного или относительного допустимого отклонения – библиотека NumPy предоставляет метод `np.isclose(a,b)` для поэлементного сравнения двух массивов: метод возвращает `True` для всех элементов, удовлетворяющих условию

```
abs(a-b) <= (atol + rtol * abs(b))
```

при абсолютном допустимом отклонении `atol` и относительном допустимом отклонении `rtol`, которые по умолчанию равны 10^{-8} и 10^{-5} соответственно, но эти значения можно изменить, установив соответствующие аргументы[78]. Дополнительный аргумент `equal_nan` по умолчанию содержит значение `False`, означающее, что значения `nan` в совпадающих позициях двух массивов интерпретируются как различные. Чтобы интерпретировать эти элементы как равные, необходимо установить `equal_nan=True`.

[77] За исключением того, что они не могут работать с объектами генераторов и итераторов.

[78] Следует отметить, что это отношение не симметрично для массивов a и b, поэтому может оказаться, что `isclose(a,b)` не равно `isclose(b,a)`.

```
In [x]: a = np.array([1.66e-27, 1.38e-23, 6.63e-34, 6.02e23, np.nan])
In [x]: b = np.array([1.66e-27, 1.66e-27, 1.66e-27, 6.00e23, np.nan])
In [x]: np.isclose(a, b)
Out[x]: array([ True , True , True , False , False], dtype=bool)
In [x]: np.isclose(a, b, equal_nan=True)
Out[x]: array([ True , True , True , False , True], dtype=bool)
```

Обратите внимание: при сравнении очень малые числа определяются как равные, даже если они отличаются на несколько порядков величины. Чтобы устранить эту проблему, необходимо установить `atol=0`, дабы сравнение производилось только по относительному допустимому отклонению:

```
In [x]: np.isclose(a, b, atol=0)
Out[x]: array([ True , False , False , False , False], dtype=bool)
```

Метод `allclose(a,b)` возвращает единственное значение `True`, только если каждый элемент в массиве `a` равен соответствующему элементу в массиве `b` (в пределах допустимого отклонения, определяемого `atol` и `rtol`), иначе возвращается `False`.

```
In [x]: x = np.linspace(0, np.pi, 100)
In [x]: np.allclose(np.sin(x)**2, 1 - np.cos(x)**2)
Out[x]: True
```

6.1.13 Упражнения

Вопросы

B6.1.1. Чем отличаются объекты `np.ndarray` и `np.array`?

B6.1.2. Почему приведенная ниже инструкция не создает двумерный массив?

```
>>> np.array((1, 0, 0), (0, 1, 0), (0, 0, 1), dtype=float)
```

Как правильно создать этот массив?

B6.1.3. Чем отличаются (если отличия существуют) следующие инструкции?

```
>>> a = np.array([0, 0, 0])
>>> a = np.array([[0, 0, 0]])
```

B6.1.4. Объяснить поведение, показанное ниже:

```
In [x]: a, b = np.zeros((3,)), np.ones((3,))
In [x]: a.dtype = 'int'
In [x]: a
Out[x]: array([0, 0, 0])
In [x]: b.dtype = 'int'
In [x]: b
Out[x]: array([4607182418800017408, 4607182418800017408, 4607182418800017408])
```

Как правильно преобразовать массив одного типа данных в массив другого типа данных?

В6.1.5. Массив 3×4×4 создается следующей инструкцией:

```
In [x]: a = np.linspace(1, 48, 48).reshape(3, 4, 4)
```

Применить операции индексирования или вырезания к этому массиву для получения следующих результатов:

а) `20.0`

б) `[ 9. 10. 11. 12.]`

в) массив 4×4:

```
[[ 33.  34.  35.  36.]
 [ 37.  38.  39.  40.]
 [ 41.  42.  43.  44.]
 [ 45.  46.  47.  48.]]
```

г) массив 3×2:

```
[[  5.,   6.],
 [ 21.,  22.],
 [ 37.,  38.]]
```

д) массив 4×2:

```
[[ 36.  35.]
 [ 40.  39.]
 [ 44.  43.]
 [ 48.  47.]]
```

е) массив 3×4:

```
[[ 13.   9.   5.   1.]
 [ 29.  25.  21.  17.]
 [ 45.  41.  37.  33.]]
```

ж) (более трудное задание) используя массив индексов, получить массив 2×2:

```
[[  1.   4.]
 [ 45.  48.]]
```

В6.1.6. Написать выражение с использованием логической индексации, которое возвращает только те значения из массива, величина которых находится в интервале от 0 до 1.

В6.1.7. Почему при вычислении приведенного ниже выражения получается результат `True`, даже если два числа, переданные в метод `np.isclose()`, отличаются на большую величину, чем `atol`?

```
In [x]: np.isclose(-2.00231930436153, -2.0023193043615, atol=1.e-14)
Out[x]: True
```

В6.1.8. Объяснить, почему при вычислении приведенного ниже выражения получается результат `True`, даже если два приблизительных значения числа π отличаются более чем на 10^{-16}:

```
In [x]: np.isclose(3.1415926535897932, 3.141592653589793, atol=1.e-16, rtol=0)
Out[x]: True
```

в то время как следующее выражение выполняется правильно:

```
In [x]: np.isclose(3.14159265358979, 3.1415926535897, atol=1.e-14, rtol=0)
Out[x]: False
```

B6.1.9. Проверить магический квадрат, созданный в примере П6.2, на соответствие следующим условиям: в нем содержатся числа от 1 до N^2, сумма его строк, столбцов и главных диагоналей равна $N(N^2 + 1)/2$.

B6.1.10. Написать однострочную инструкцию, возвращающую `True`, если массив является монотонно возрастающей последовательностью, иначе возвращается `False`.

Совет: метод `np.diff` возвращает разность между смежными элементами последовательности. Например:

```
In [x]: np.diff([1, 2, 3, 3, 2])
Out[x]: array([ 1,  1,  0, -1])
```

◊ **B6.1.11.** (Более трудное задание) `dtype np.uint8` представляет беззнаковое целое число из 8 бит. Следовательно, его значение может находиться в интервале 0–255. Объяснить показанное ниже поведение:

```
In [x]: x = np.uint8(250)
In [x]: x * 2
Out[x]: 500

In [x]: x = np.array([250,], dtype=np.uint8)
In [x]: x * 2
Out[x]: array([244], dtype=uint8)
```

Задачи

36.1.1. Преобразовать данные из табл. 6.5 о различных видах китообразных в структурированный массив NumPy и упорядочить эти данные а) по массе тела и б) по популяции. В каждом случае определить индекс, по которому Bryde's whale (полосатик Брайда) (популяция: 100 000, масса: 25 т) должен быть вставлен, чтобы сохранить упорядоченность массива.

Таблица 6.5

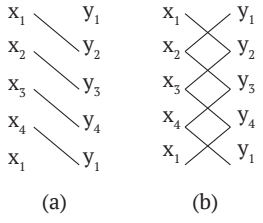
Наименование	Популяция	Масса тела, т
Bowhead whale (гренландский кит)	9000	60
Blue whale (синий кит)	20 000	120
Fin whale (финвал)	100 000	70
Humpback whale (горбатый кит)	80 000	30
Gray whale (серый кит)	26 000	35
Atlantic white-sided dolphin (атлантический белобокий дельфин)	250 000	0.235

Окончание табл. 6.5

Наименование	Популяция	Масса тела, т
Pacific white-sided dolphin (тихоокеанский белобокий дельфин)	1 000 000	0.15
Killer whale (косатка)	100 000	4.5
Narwhal (нарвал)	25 000	1.5
Beluga (белуха)	100 000	1.5
Sperm whale (кашалот)	2 000 000	50
Baiji (китайский речной дельфин)	13	0.13
North Atlantic right whale (северный гладкий кит)	300	75
North Pacific right whale (японский кит)	200	80
Southern right whale (южный гладкий кит)	7000	70

Текстовый файл, содержащий эти данные, можно загрузить отсюда: https://scipython.com/ex/bfk.

36.1.2. Алгоритм шнурования (shoelace algorithm) для вычисления площади простого многоугольника (т. е. многоугольника без отверстий и самопересечений) выполняется следующим образом: координаты N вершин (x,y) записываются в массив $N\times2$, затем координаты первой вершины повторяются в последней строке, чтобы получился массив $(N+1)\times2$. Далее (a) каждое значение координаты x в первых N строках умножается на значение координаты y в следующей нижней строке и вычисляется сумма $S_1 = x_1y_2 + x_2y_3 + \ldots + x_Ny_1$. Затем (b) каждое значение координаты y в первых N строках умножается на значение координаты x в следующей нижней строке и вычисляется сумма $S_2 = y_1x_2 + y_2x_3 + \ldots + y_Nx_1$. Площадь многоугольника равна $1/2|S_1 - S_2|$.



Реализовать этот алгоритм как функцию, которая принимает массив NumPy координат вершин и возвращает значение площади многоугольника. Не использовать циклы Python.

36.1.3. С помощью NumPy можно решить эту задачу без использования цикла (Python).

Формула нормализованной гауссовой функции со средним значением μ и стандартным отклонением σ:

$$g(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-\mu)}{2\sigma^2}\right).$$

Написать программу для вычисления и построения графиков гауссовых функций при μ = 0 и при трех значениях σ = 0.5, 1, 1.5. Использовать сетку из 1000 точек в интервале $-10 \leqslant x \leqslant 10$.

Проверить (прямым суммированием), что эти функции нормализованы с площадью 1.

Вычислить первую производную от этих функций на той же сетке, используя приближенную формулу центральных разностей

$$g'(x) \approx \frac{g(x+h) - g(x-h)}{2h}$$

при малом значении h, выбранном соответствующим образом.

6.2 Чтение и запись массива в файл

Научные данные часто считываются из текстового файла, который может содержать комментарии, пропущенные значения и пустые строки. Столбцы значений могут быть выровнены в соответствии с форматом полей фиксированной ширины либо отделены друг от друга одним или несколькими символами-разделителями (такими как пробелы, табуляции или запятые). Кроме того, в файле может содержаться описательный заголовок и даже сноски-примечания, которые затрудняют парсинг при использовании только строковых методов Python.

Библиотека NumPy предоставляет несколько функций для считывания данных из текстового файла. Самый простой метод `np.loadtxt` обрабатывает множество общих вариантов, а более интеллектуальный метод `np.genfromtxt` обеспечивает улучшенную обработку пропущенных значений и сносок. Эти и другие методы описаны в следующих разделах.

6.2.1 Методы `np.save` и `np.load`

Существует независимый от платформы бинарный формат для сохранения массива NumPy:

```
In [x]: np.save('my-array.npy', a)
```

Массив `a` сохраняется в бинарном файле *my-array.npy* (расширение *.npy* добавляется, если оно не задано явно). Затем массив можно снова загрузить с помощью NumPy в любой другой операционной системе командой

```
In [x]: a = np.load('my-array.npy')
```

В этом случае обязательно должно быть указано расширение *.npy*.

6.2.2 Метод np.loadtxt

Прототип этого метода:

```
np.loadtxt(fname, dtype=<class 'float'>, comments='#',
           delimiter=None, converters=None, skiprows=0,
           usecols=None, unpack=False, ndmin=0)
```

Ниже приводится описание аргументов:

- `fname` – единственный обязательный аргумент, который может быть именем файла, указателем на открытый файл или генератором, возвращающим строки данных для обработки (парсинга);
- `dtype` – тип данных массива, по умолчанию float, но его можно явно установить в этом аргументе `dtype`. По существу, именно здесь устанавливаются значения имен и типов данных для структурированного массива (см. раздел 6.1.10);
- `comments` – комментарии в файле обычно начинаются с некоторого символа, как, например, # (в исходном коде Python) или %. Чтобы библиотека NumPy игнорировала содержимое всех строк, начинающихся с такого символа, используется аргумент `comments`, для которого по умолчанию задано значение #;
- `delimiter` – строка, используемая для разделения столбцов данных в файле. По умолчанию задано значение `None`, означающее, что любое количество пробельных символов (пробелов, табуляций) разделяет данные. Для считывания файлов в формате csv (данные, разделенные запятыми) необходимо установить `delimiter=','`;
- `converters` – необязательный словарь, отображающий индекс столбца в функцию, преобразующую строковые значения из этого столбца в данные (например, типа `float`);
- `skiprows` – целое число, определяющее количество пропускаемых строк в начале файла, прежде чем начать считывание данных (например, для пропуска строк заголовка). По умолчанию равно 0 (заголовка нет);
- `usecols` – последовательность индексов столбцов, определяющая, какие столбцы из файла должны возвращаться как данные. По умолчанию задано значение `None`, означающее, что все столбцы должны быть обработаны и возвращены;
- `unpack` – по умолчанию таблица данных возвращается в одном массиве из строк и столбцов, отображающих структуру считываемого файла. При установке `unpack=True` этот массив будет преобразован (транспонирован) так, чтобы можно было выбирать отдельные столбцы и присваивать их различным переменным;
- `ndmin` – минимальное количество измерений, которое должен иметь возвращаемый массив. По умолчанию задано значение 0 (так что файл, содержащий единственное число, считывается как скалярное значение). Можно установить значение 1 или 2.

Например, для считывания первого, третьего и четвертого столбцов из файла *data.txt* в три отдельных одномерных массива выполняется следующая инструкция:

```
col1 , col3 , col4 = np.loadtxt('data.txt', usecols=(0, 2, 3), unpack=True)
```

Пример П6.6. Использование метода `np.lodtxt` лучше всего продемонстрировать на примере. Рассмотрим следующий текстовый файл с данными, относящимися к некоторой (вымышленной) группе студентов. Эти данные содержатся в файле *eg6-a-student-data.txt*, который можно скачать отсюда: https://scipython.com/eg/bac.

```
# Student data collected on 17 July 2014.
# Researcher: Dr Wicks, University College Newbury.

# The following data relate to N = 20 students. It
# has been totally made up and so therefore is 100%
# anonymous.

Subject Sex     DOB     Height   Weight      BP     VO2max
(ID)    M/F   dd/mm/yy    m        kg       mmHg   mL.kg-1.min-1
JW-1     M    19/12/95    1.82      92.4   119/76    39.3
JW-2     M    11/1/96     1.77      80.9   114/73    35.5
JW-3     F    2/10/95     1.68      69.7   124/79    29.1
JW-6     M    6/7/95      1.72      75.5   110/60    45.5
# JW-7   F    28/3/96     1.66      72.4   101/68    -
JW-9     F    11/12/95    1.78      82.1   115/75    32.3
JW-10    F    7/4/96      1.60      -      -/-       30.1
JW-11    M    22/8/95     1.72      77.2   97/63     48.8
JW-12    M    23/5/96     1.83      88.9   105/70    37.7
JW-14    F    12/1/96     1.56      56.3   108/72    26.0
JW-15    F    1/6/96      1.64      65.0   99/67     35.7
JW-16    M    10/9/95     1.63      73.0   131/84    29.9
JW-17    M    17/2/96     1.67      89.8   101/76    40.2
JW-18    M    31/7/96     1.66      75.1   -/-       -
JW-19    F    30/10/95    1.59      67.3   103/69    33.5
JW-22    F    9/3/96      1.70      -      119/80    30.9
JW-23    M    15/5/95     1.97      89.2   124/82    -
JW-24    F    1/12/95     1.66      63.8   100/78    -
JW-25    F    25/10/95    1.63      64.4   -/-      28.0
JW-26    M    17/4/96     1.69      -      121/82   39.
```

Вычислим по отдельности средний рост студентов мужского и женского пола. Для этого необходимы второй и четвертый столбцы, в них нет пропущенных данных, поэтому можно использовать метод `np.loadtxt`. Сначала сформируем запись `dtype` для этих двух полей, затем прочитаем соответствующие столбцы после пропуска первых девяти строк заголовка:

```
In [x]: fname = 'eg6-a-student-data.txt'
In [x]: dtype1 = np.dtype([('gender', '|S1'), ('height', 'f8')])
In [x]: a = np.loadtxt(fname, dtype=dtype1, skiprows=9, usecols=(1,3))
In [x]: a
Out[x]:
array([(b'M', 1.8200000524520874), (b'M', 1.7699999809265137 ,
       (b'F', 1.6799999475479126), (b'M', 1.7200000286102295),
       ...
       (b'M', 1.690000057220459)],
      dtype=[('gender', 'S1'), ('height', '<f8')])
```

Для вычисления среднего роста студентов мужского пола требуется индекс только тех записей, в которых поле `gender` содержит значение `M`. Для этой цели можно создать массив логических значений:

```
In [x]: m = a['gender'] == b'M'
In [x]: m
Out[x]: array([ True,  True, False,  True, ...,  True], dtype=bool)
```

В массиве `m` содержатся элементы `True` или `False` для каждой из 19 корректных записей (одна запись закомментирована), соответствующие мужскому или женскому полу студента. Поэтому значения роста студентов мужского пола могут выглядеть так:

```
In [x]: print(a['height'][m])
[ 1.82000005   1.76999998   1.72000003   1.72000003   1.83000004   1.63
  1.66999996   1.65999997   1.97000003   1.69000006]
```

Для вычисления средних значений роста необходимо выполнить следующие инструкции:

```
In [x]: m_av = a['height'][m].mean()
In [x]: f_av = a['height'][~m].mean()                                    ❶
In [x]: print('Male average: {:.2f} m, Female average: {:.2f} m'.format(m_av, f_av))
Male average: 1.75 m, Female average: 1.65 m
```

❶ Обратите внимание: выражение `~m` («не `m`») инвертирует (меняет на противоположные) логические значения в массиве `m`.

Для выполнения аналогичных аналитических операций с весами студентов необходимо проделать немного больше работы, потому что некоторые значения пропущены (обозначены символом дефиса «-»). Можно было бы воспользоваться методом `np.genfromtxt` (см. раздел 6.2.3), но вместо этого мы напишем метод преобразования. Будем заменять пропущенные значения абсолютно бессмысленным с физической точки зрения значением –99. Функция `parse_weight` ожидает строковый аргумент и возвращает значение типа `float`:

```
def parse_weight(s):
    try:
        return float(s)
    except ValueError:
        return -99.
```

Это функция, которую необходимо передать как метод преобразования для столбца 4:

```
In [x]: dtype2 = np.dtype([('gender', '|S1'), ('weight', 'f8')])
In [x]: b = np.loadtxt(fname, dtype=dtype2, skiprows=9, usecols=(1, 4),
                       converters={4: parse_weight})
```

Теперь необходимо маскировать некорректные данные и проиндексировать обрабатываемый массив с помощью массива логических значений, как это делалось ранее:

```
In [x]: mv = b['weight'] > 0     # Только элементы с индексом True - корректные данные.
In [x]: m_wav = b['weight'][mv & m].mean()       # Корректный элемент, пол мужской.
In [x]: f_wav = b['weight'][mv & ~m].mean()      # Корректный элемент, пол женский.
In [x]: print('Male average: {:.2f} kg,
               Female average: {:.2f} kg'.format(m_wav, f_wav))
Male average: 82.44 kg, Female average: 66.94 kg
```

Теперь считываем данные о кровяном давлении. Здесь возникает проблема, так как значения систолического и диастолического давления разделены не пробелом, а символом слеша (/). Один вариант решения этой проблемы – переформатирование каждой строки для замены слеша на пробел до передачи записей в метод `np.loadtxt`. Напомню, что `fname` может быть генератором, а не только именем файла или ссылкой на открытый файл: напишем соответствующую функцию генератора `reformat_lines`, которая принимает объект открытого файла `file` и после описанной выше замены передает строки (поочередно, по одной) этого файла в метод `np.loadtxt`. Это приведет к нарушению нумерации столбцов из-за побочного эффекта при разделении дат рождения на три столбца, поэтому в переформатированных строках значения кровяного давления теперь в столбцах с индексами 7 и 8.

Листинг 6.4. Считывание столбца значений кровяного давления

```
# eg6-a-read-bp.py
import numpy as np

fname = 'eg6-a-student-data.txt'
dtype3 = np.dtype([('gender', '|S1'), ('bps', 'f8'), ('bpd', 'f8')])

def parse_bp(s):
    try:
        return float(s)
    except ValueError:
        return -99.

def reformat_lines(fi):
    for line in fi:
        line = line.replace('/', ' ')
        yield line

with open(fname) as fi:
    gender, bps, bpd = np.loadtxt(reformat_lines(fi), dtype3 , skiprows=9,
                usecols=(1, 7, 8),converters={7: parse_bp , 8: parse_bp},
                unpack=True)

# Теперь необходимо как-то обработать эти данные.
```

6.2.3 Метод `np.genfromtxt`

Метод NumPy `genfromtxt` похож на метод `np.loadtxt`, но принимает немного больше аргументов и способен справляться с проблемой пропущенных элементов данных.

Перечисленные ниже аргументы совпадают с аргументами для метода `np.loadtxt`: `fname` (единственный обязательный аргумент), `dtype`, `comments`, `converters`, `usecols` и `unpack`.

Заголовки и примечания

Вместо аргумента `skiprows` метода `np.loadtxt` для метода `np.genfromtxt` существуют два необязательных аргумента `skip_header` и `skip_footer`, в которых определяется количество пропускаемых строк в начале и в конце файла соответственно.

Поля фиксированной ширины

Аргумент `delimiter` работает так же, как для метода `np.loadtxt`, но может быть задан еще в виде последовательности целых чисел, определяющих значения ширины для каждого считываемого поля, если между столбцами данных нет разделителей. Например, предположим, что приведенный ниже текстовый файл *data.txt* должен интерпретироваться как состоящий из четырех столбцов с размерами 2, 1, 9 и 3 символа (для наглядности пробелы обозначены специальным символом «␣»):

```
␣12␣␣100.231.03
␣11␣1201.842.04
␣11␣␣␣99.324.02
```

Следовательно, первая строка должна быть разделена так: `' 1', '2', ' 100.231', '.03'`. Здесь нет символов-разделителей, поэтому подобное разделение невозможно выполнить с помощью `np.loadtxt`, только с применением метода `np.genfromtxt`:

```
In [x]: np.genfromtxt(fname='data.txt', delimiter=[2, 1, 9, 3],
                      dtype='i4, i4, f8, f8')
array([(1, 2, 100.231, 0.03), (1, 1, 1201.842, 0.04), (1, 1, 99.324, 0.02)],
      dtype=[('f0', '<i4'), ('f1', '<i4'), ('f2', '<f8'), ('f3', '<f8')])
```

Такой способ позволяет получить требуемый результат.

Пропущенные данные

Если набор данных неполон, то метод `np.loadtxt` неспособен выполнить парсинг полей с пропущенными данными и преобразовать их в корректные значения для итогового массива – будет сгенерировано исключение. Но метод `np.genfromtxt` устанавливает пропущенные или некорректные элементы равными значениям по умолчанию, приведенным в табл. 6.6.

Таблица 6.6. Значения по умолчанию для заполнения пропущенных полей данных, используемые методом `np.genfromtxt`

Тип данных	Значение по умолчанию
int	-1
float	np.nan
bool	False
complex	np.nan + 0.j

Например, в файле с данными, разделенными запятыми, существует два способа обозначения пропущенных данных: пустые поля и элементы «???»:

```
10.1,4,-0.1,2
10.2,4,,0
10.3,???,,4
10.4,2,0.,
10.5,-1,???,3
```

В этом случае метод `np.genfromtxt` устанавливает для пропущенных полей значения по умолчанию:

```
In [x]: data = np.genfromtxt(fname='data.txt', dtype='f8, i4, f8, i4',
...:                         delimiter=',')
In [x]: print(data)
[(10.1, 4, -0.1, 2) (10.2, 4, nan, 0) (10.3, -1, nan, 4) (10.4, 2, 0.0, -1)
 (10.5, -1, nan, 3)]
```

Аргументы `missing_values` и `filling_values` позволяют более точно управлять значениями по умолчанию, которые должны использоваться в каждом столбце. Если аргумент `missing_values` определен как последовательность строк, то каждая строка связывается с соответствующим по порядку столбцом в файле данных. Если аргумент `missing_values` задан как словарь строковых значений, то ключи обозначают либо индексы столбцов (если это целые числа), либо имена столбцов (если это строки). Соответствующий аргумент `filling_values` отображает эти индексы или имена столбцов в значения по умолчанию. Если аргумент `filling_values` задан как единственное значение, то оно используется для пропущенных данных во всех столбцах.

Например, для замены некорректных значений (обозначенных как «???») в столбце 1 на 999, пропущенных или некорректных значений (также обозначенных как «???») в столбце 2 на –99 и пропущенных значений в столбце 3 на 0 выполняется следующая инструкция:

```
In [x]: data =np.genfromtxt(fname='data.txt', dtype='f8, i4, f8, i4',
...:                        delimiter=',', missing_values={1: '???', 2: '???'},
...:                        filling_values={1: 999, 2: -99., 3: 0})
...:
In [x]: print(data)
[(10.1, 4, -0.1, 2) (10.2, 4, -99.0, 0) (10.3, 999, -99.0, 4)
 (10.4, 2, 0.0, 0) (10.5, -1, -99.0, 3)]
```

Следует особо отметить, что пропущенные элементы во втором столбце были заменены на `999` вместо значения по умолчанию –1 – это весьма важно, если –1 является допустимым значением для данного столбца (но при этом необходим дополнительный код для распознавания и обработки специальных значений, таких как `999`)[79].

[79] Для более детальной обработки пропущенных значений см. документацию по методу `genfromtxt`, чтобы получить более подробную информацию об аргументе `usemask` и о маскируемых массивах (masked arrays) вообще.

Имена столбцов

Аргумент `names` предоставляет способ определения имен для столбцов считываемых данных. Если для этого аргумента задано значение `True`, то имена считываются из первой корректной строки после пропуска строк, количество которых определено аргументом `skip_header`. Если аргумент `names` представлен строкой имен, разделенных запятыми, или последовательностью строк, то эти строки используются как имена столбцов. По умолчанию для аргумента `names` задано значение `None`, а имена полей берутся из аргумента `dtype`, если они определены.

Пример П6.7. В эксперименте по исследованию эффекта Струпа (психология) в группе студентов замерялось время чтения 25 названий цветов, расположенных в случайном порядке, сначала написанных черным цветом, затем цветами, не соответствующими названию цвета (например, слово «красный» было написано синим цветом). Результаты представлены в текстовом файле *stroop.txt*, который можно скачать здесь: https://scipython.com/eg/baj. Пропущенные данные обозначены символом X.

```
Subject Number, Gender, Time (words in black), Time (words in color)
1,F,18.72,31.11
2,F,21.14,52.47
3,F,19.38,33.92
4,M,22.03,50.57
5,M,21.41,29.63
6,M,15.18,24.86
7,F,14.13,33.63
8,F,19.91,42.39
9,F,X,43.60
10,F,26.56,42.31
11,F,19.73,49.36
12,M,18.47,31.67
13,M,21.38,47.28
14,M,26.05,45.07
15,F,X,X
16,F,15.77,38.36
17,F,15.38,33.07
18,M,17.06,37.94
19,M,19.53,X
20,M,23.29,49.60
21,M,21.30,45.56
22,M,17.12,42.99
23,F,21.85,51.40
24,M,18.15,36.95
25,M,33.21,61.59
```

Можно прочитать эти данные с помощью метода `np.genfromtxt` и обработать результаты эксперимента, используя код в листинге 6.5.

Листинг 6.5. Анализ данных, полученных в ходе эксперимента по изучению эффекта Струпа

```
# eg6-stroop.py
import numpy as np

# Считывание данных из файла stroop.txt, определение пропущенных значений и
# замена их на значение NaN.
data = np.genfromtxt('stroop.txt', skip_header=1,                                    ❶
                     dtype=[('student', 'u8'), ('gender', 'S1'),
                            ('black', 'f8'), ('color', 'f8')],
                     delimiter=',',
                     missing_values='X')
nwords = 25

# Удаление некорректных строк из набора данных.
filtered_data = data[np.isfinite(data['black']) & np.isfinite(data['color'])]        ❷

# Извлечение строк по полям пола (M/F) и цвету слова (black/color) и нормализация
# по времени, затраченному на прочтение слова.
fb = filtered_data['black'][filtered_data['gender']==b'F'] / nwords
mb = filtered_data['black'][filtered_data['gender']==b'M'] / nwords
fc = filtered_data['color'][filtered_data['gender']==b'F'] / nwords
mc = filtered_data['color'][filtered_data['gender']==b'M'] / nwords

# Итоговая статистика: среднее значение и стандартное отклонение по полу и цвету слова.
mu_fb , sig_fb = np.mean(fb), np.std(fb)
mu_fc , sig_fc = np.mean(fc), np.std(fc)
mu_mb , sig_mb = np.mean(mb), np.std(mb)
mu_mc , sig_mc = np.mean(mc), np.std(mc)

print('Mean and (standard deviation) times per word (sec)')
print('gender |    black      |     color     | difference')
print('   F   | {:4.3f} ({:4.3f}) | {:4.3f} ({:4.3f}) |   {:4.3f}'
                    .format(mu_fb, sig_fb, mu_fc, sig_fc, mu_fc - mu_fb))
print('   M   | {:4.3f} ({:4.3f}) | {:4.3f} ({:4.3f}) |   {:4.3f}'
                    .format(mu_mb, sig_mb, mu_mc, sig_mc, mu_mc - mu_mb))
```

❶ При отсутствии каких-либо значений, определяемых аргументом `filling_values`, метод `np.genfromtxt` будет заменять некорректные поля значением `np.nan`.

❷ Необходимо рассмотреть только тех студентов, для которых замерено время в обеих частях эксперимента, поэтому здесь создается отфильтрованный набор данных.

Выводимый результат показывает существенное замедление скорости чтения слов, написанных другим цветом, по сравнению со скоростью чтения слов, написанных черным цветом:

```
Mean and (standard deviation) times per word (sec)
gender |    black      |    color     | difference
   F   | 0.770 (0.137) | 1.632 (0.306) |   0.862
   M   | 0.849 (0.186) | 1.679 (0.394) |   0.830
```

6.2.4 Метод np.savetxt

Метод `np.savetxt` сохраняет массив NumPy как текстовый файл. Его сигнатура приведена ниже:

```
np.savetxt(fname , X, fmt='%.18e', delimiter=' ',
           newline='\n', header='', footer='', comments='# ')
```

Описание аргументов:

- `fname` – имя файла или ссылка (дескриптор) на открытый файл, в который будут сохраняться данные массива;
- `X` – имя сохраняемого массива;
- `fmt` – строка, определяющая спецификаторы формата в стиле языка C, при выводе данных массива (подробности см. в разделе 2.3.7). По умолчанию это строка `'%.18e'`;
- `delimiter` – строка, разделяющая столбцы в файле вывода; по умолчанию – один пробел;
- `newline` – строка, разделяющая строки в файле вывода; по умолчанию используется Unix-стиль `'\n'`. Пользователи Windows могут предпочесть установку для newline последовательности символов, применяемую на их платформе: `'\r\n'`;
- `header` – строка (возможно, несколько строк), которая должна быть записана в начале файла вывода;
- `footer` – строка (возможно, несколько строк), которая должна быть записана в конце файла вывода;
- `comments` – строка, которая должна добавляться в строки `header` и `footer`, чтобы пометить их как комментарии. По умолчанию это строка `'# '`. Это удобно, если файл в дальнейшем будет считываться методами `np.loadtxt` или `np.genfromtxt`, а при использовании комментариев не требуется явно определять количество строк заголовка и примечания.

Пример П6.8. Период распада группы радиоактивных ядер можно имитировать следующим образом. Рассмотрим отрезок времени, разделенный на короткие дискретные интервалы продолжительностью $\Delta t \ll \tau$, где τ – полное время распада (которое относится ко времени полураспада $t_{1/2}$ как $\tau = t_{1/2}/\ln 2$). Вероятность того, что рассматриваемое ядро распадется за время Δt, равна $p = \Delta t/\tau$.

На каждом временно́м шаге этой имитации выполняется цикл по ядрам, не распавшимся на предыдущем шаге, и произвольным образом выбирается случайное число из равномерного распределения [0, 1): если это случайное число меньше p, то считается, что ядро распалось.

Код в листинге 6.6 определяет функцию для выполнения описанной выше имитации для набора из $N_0 = 500$ ядер радиоактивного углерода ^{14}C с периодом полураспада $t_{1/2} = 5730$ лет. Выполняется `nsims = 10` симуляций, и результаты сохраняются в файле с разделением данных запятыми *14C-sim.csv* с кратким описательным заголовком.

Листинг 6.6. Имитация радиоактивного распада ядер ^{14}C

```
import random
import numpy as np

def decay_sim(thalf , N0=500, tgrid=None , nhalflives=4):
    """Simulate the radioactive decay of N0 nuclei.

    thalf is the half -life in some units of time.
    If tgrid is provided , it should be a sequence of evenly -spaced time points
    to run the simulation on.
    If tgrid is None , it is calculated from nhalflives , the number of
    half -lives to run the simulation for.
    """
    # """Имитация радиоактивного распада N0 ядер.
    #
    # thalf - период полураспада в некоторых единицах времени.
    # Если задано значение tgrid, то оно должно быть последовательностью равномерно
    # распределенных точек времени для выполнения имитации.
    # Если tgrid содержит значение None, то оно вычисляется по nhalflives, числу,
    # соответствующему периоду полураспада, для выполнения имитации.
    #
    # """

    # Вычисление периода распада по периоду полураспада.
    tau = thalf / np.log(2)

    if tgrid is None:
        # Создание сетки из Nt точек времени до значения tmax.
        Nt, tmax = 100, thalf * nhalflives
        tgrid, dt = np.linspace(0, tmax, Nt, retstep=True)
    else:
        # tgrid задан: вывод Nt и шаг времени dt.
        Nt = len(tgrid)
        dt = tgrid[1] - tgrid[0]

    N = np.empty(Nt, dtype=int)
    N[0] = N0
    # Вероятность того, что заданное ядро распадется за время dt.
    p = dt / tau

    for i in range(1, Nt):
        # На каждом шаге времени начинаем обработку нераспавшихся ядер,
        # оставшихся с предыдущего шага.
        N[i] = N[i-1]
        # Поочередно рассматриваем каждое ядро и решаем, распадается оно или нет.
        for j in range(N[i-1]):
            r = random.random()
            if r < p:
                # Это ядро распадается.
                N[i] -= 1
    return tgrid, N

N0 = 500
# Период полураспада углерода 14C в годах.
thalf = 5730
```

```
# Использование Nt шагов времени до значений tmax в годах.
Nt, tmax = 100, 20000
tgrid = np.linspace(0, tmax , Nt)

# Повторение имитации "эксперимента" nsims раз.
nsims = 10
Nsim = np.empty((Nt, nsims))
for i in range(nsims):
    _, Nsim[:, i] = decay_sim(thalf , N0, tgrid)

# Сохранение временной сетки, за которой следуют результаты имитации в столбцах.
# Сохраняются целые значения для данных, и создается файл с данными, разделенными
# запятыми, с двухстрочным заголовком.
np.savetxt('14C-sim.csv', np.hstack((tgrid[:, None], Nsim)),
           fmt = '%d', delimiter=',',
           header=f'Simulations of the radioactive decay of {N0} 14C nuclei.\n'
                  f'Columns are time in years followed by {nsims} decay simulations.'
          )
```

Файл вывода *14C-sim.csv* будет содержать приблизительно такие данные:

```
# Simulations of the radioactive decay of 500 14C nuclei.
# (Имитации радиоактивного распада 500 ядер 14C.)
# Columns are time in years followed by 10 decays.
# (В столбцах указано время в годах, далее - данные о 10 имитациях распада.)
0,500,500,500,500,500,500,500,500,500,500
202,489,486,487,491,487,486,485,487,490,490
404,479,478,483,479,477,476,480,474,484,482
606,462,467,470,463,464,463,470,454,474,471
...
```

Этот файл можно считать в массив NumPy следующей инструкцией:

```
arr = np.loadtxt('14C-sim.csv', delimiter=',')
```

Также см. задачу 36.5.7.

6.2.5 Упражнения

Задачи

36.2.1. В следующем текстовом файле, доступном по адресу: https://scipython.com/ex/bfj, содержатся данные о вершинах-восьмитысячниках в алфавитном порядке.

```
ex6-2-b-mountain-data.txt This file contains a list of the 14
highest mountains in the world with their names, height, year
of first ascent, year of first winter ascent, and location as
longitude and latitude in degrees (d), minutes (m) and seconds
(s). Note: as of 2019, no winter ascent has been made of K2.
```

```
---------------------------------------------------------------------
Name          Height  First ascent First winter Location
                m         date     ascent date  (WGS84)
---------------------------------------------------------------------
Annapurna I   8091    3/6/1950    3/2/1987     28d35m46sN 83d49m13sE
Broad Peak    8051    9/6/1957    5/3/2013     35d48m39sN 76d34m06sE
Cho Oyu       8201    19/10/1954  12/2/1985    28d05m39sN 86d39m39sE
Dhaulagiri I  8167    13/5/1960   21/1/1985    27d59m17sN 86d55m31sE
Everest       8848    29/5/1953   17/2/1980    27d59m17sN 86d55m31sE
Gasherbrum I  8080    5/7/1958    9/3/2012     35d43m28sN 76d41m47sE
Gasherbrum II 8034    7/7/1956    2/2/2011     35d45m30sN 76d39m12sE
K2            8611    31/7/1954   -            35d52m57sN 76d30m48sE
Kangchenjunga 8568    25/5/1955   11/1/1986    27d42m09sN 88d08m54sE
Lhotse        8516    18/5/1956   31/12/1988   27d57m42sN 86d56m00sE
Makalu        8485    15/5/1955   9/2/2009     27d53m21sN 87d05m19sE
Manaslu       8163    9/5/1956    12/1/1984    28d33m0sN 84d33m35sE
Nanga Parbat  8126    3/7/1953    16/2/2016    35d14m15sN 74d35m21sE
Shishapangma  8027    2/5/1964    14/1/2005    28d21m8sN 85d46m47sE
---------------------------------------------------------------------
```

Использовать метод NumPy `genfromtxt` для считывания этих данных в соответствующий структурированный массив для определения следующих фактов:

а) самая низкая вершина-восьмитысячник;
б) самая северная, восточная, южная и западная вершина;
в) самое позднее первое восхождение на вершину;
г) первая вершина, на которую было совершено восхождение зимой.

Также создать другой структурированный массив, содержащий список вершин с указанием их высоты в футах и даты первого восхождения с упорядочением по возрастанию высоты[80].

36.2.2. Файл *busiest_airports.txt*, который можно загрузить отсюда: https://scipython.com/ex/bfa, предоставляет подробную информацию о 30 самых загруженных аэропортах в мире в 2014 г. Поля, разделенные табуляциями: трехбуквенный код IATA, название аэропорта, место расположения аэропорта, широта и долгота (оба значения в градусах).

Написать программу для определения расстояния между двумя аэропортами, определяемыми по их трехбуквенному коду IATA, с использованием формулы гаверсинуса (см., например, задачу 34.4.2) и с предположением о том, что радиус сферической поверхности Земли равен 6378.1 км.

36.2.3. Всемирный банк предоставляет обширную совокупность наборов данных по широкому диапазону «индикаторов», которые можно найти здесь: https://data.worldbank.org/. Наборы данных, относящиеся к показателям детской вакцинации против туберкулеза (BCG), полиомиелита (Pol3) и кори в трех странах Юго-Восточной Азии с 1960 по 2013 г., доступны здесь: https://scipython.com/ex/bfb. Поля разделены точками с запятой, а пропущенные данные обозначены как `'..'`.

[80] 1 м = 3.2808399 фута.

Использовать методы NumPy для считывания этих данных и создания трех графиков (по каждой вакцине отдельно) и сравнить показатели (коэффициенты) вакцинации в этих трех странах.

6.3 Статистические методы

Библиотека NumPy предоставляет несколько методов для выполнения статистического анализа по всему массиву в целом или по одной из его осей.

6.3.1 Порядковая статистика

Максимумы и минимумы

Мы уже использовали методы `np.min` и `np.max` для поиска минимальных и максимальных значений в массиве (эти методы также доступны по именам `np.amin` и `np.amax`). Если массив содержит одно или несколько значений NaN, то соответствующим минимальным или максимальным значением будет `np.nan`. Чтобы не учитывать значения NaN, следует использовать `np.nanmin` и `np.nanmax`:

```
In [x]: a = np.sqrt(np.linspace(-2, 2, 4))
In [x]: print(a)
[        nan         nan  0.          1.           1.41421356]
In [x]: np.min(a), np.max(a)
Out[x]: (nan, nan)
In [x]: np.nanmin(a), np.nanmax(a)
(0.0, 1.4142135623730951)
```

Нам также встречались функции `np.argmin` и `np.argmax`, которые возвращают индекс минимальных и максимальных значений в массиве, для них тоже существуют варианты `np.nanargmin` и `np.nanargmax`:

```
In [x]: np.argmin(a), np.argmax(a)
Out[x]: (0, 0)          # Первое значение nan в массиве.
In [x]: np.nanargmin(a), np.nanargmax(a)
Out[x]: (2, 4)          # Индексы значений 0, 1.41421356.
```

Родственные методы `np.fmin` / `np.fmax` и `np.minimum` / `np.maximum` сравнивают два массива элемент за элементом и возвращают другой массив той же формы. Первая пара методов не учитывает значения NaN, вторая пара методов передает значения NaN в итоговый массив. Например:

```
In [x]: np.fmin([1, -5, 6, 2], [0, np.nan, -1, -1])
array([ 0., -5., -1., -1.])          # Значения NaN не учитываются.
In [x]: np.maximum([1, -5, 6, 2], [0, np.nan, -1, -1])
array([ 1., nan, 6., 2.])            # Значения NaN передаются в итоговый массив.
```

Процентили

Метод `np.percentile` возвращает определенный процентиль `q` по данным некоторой оси (или по данным линеаризованной версии массива, если ось не задана). Минимум массива – это значение с $q = 0$ (с нулевым процентилем), максимум массива – это значение с $q = 100$ (с 100-м процентилем), а меди-

анное (срединное) значение соответствует q = 50 (50-му процентилю). Если в массиве несколько значений точно соответствуют требуемому значению q, то используется взвешенное среднее значение двух таких ближайших значений. Например:

```
In [x]: a = np.array([[0., 0.6, 1.2], [1.8, 2.4, 3.0]])
In [x]: np.percentile(a, 50)
1.5
In [x]: np.percentile(a, 75)
2.25
In [x]: np.percentile(a, 50, axis=1)
array([ 0.6, 2.4])
In [x]: np.percentile(a, 75, axis=1)
array([ 0.9, 2.7])
```

6.3.2 Средние значения, дисперсии и корреляции

Средние значения

В дополнение к методу `np.mean`, который вычисляет арифметическое среднее для значений в заданной оси массива, библиотека NumPy предоставляет методы для вычисления взвешенного среднего значения, медианы, стандартного отклонения и дисперсии. Взвешенное среднее значение вычисляется по формуле

$$\bar{x}_w = \frac{\sum_i^N w_i x_i}{\sum_i^N w_i},$$

где весовые коэффициенты w_i представлены как последовательность той же длины, что и массив. Например:

```
In [x]: x = np.array([1., 4., 9., 16.])
In [x]: np.mean(x)
7.5
In [x]: np.median(x)
6.5
In [x]: np.average(x, weights=[0., 3., 1., 0.])
5.25        # T. e. (3.*4. + 1.*9.) / (3. + 1.).
```

Если необходима сумма весовых коэффициентов, а также взвешенное среднее значение, то для аргумента `returned` устанавливается значение `True`. В следующем примере демонстрируется этот подход, а также определяются взвешенные средние значения в каждой строке (`axis=1` средние значения по столбцам двумерного массива):

```
In [x]: x = np.array( [[1., 8., 27], [-0.5, 1., 0.]] )
In [x]: av, sw = np.average(x, weights=[0., 1., 0.1], axis=1, returned=True)
In [x]: print(av)
[ 9.72727273  0.90909091]
In [x]: print(sw)
[ 1.1  1.1]
```

Таким образом, средние значения равны (1 × 8 + 0.1 × 27)/1.1 = 9.72727273 и (1 × 1.)/1.1 = 0.90909091, где 1.1 – сумма весовых коэффициентов.

Стандартные отклонения и дисперсии

Функция `np.std` вычисляет по умолчанию нескорректированное стандартное отклонение выборки по формуле:

$$\sigma_N = \sqrt{\frac{1}{N}\sum_i^N (x_i - \overline{x})^2},$$

где x_i – N наблюдаемых значений в массиве, а $\overline{x}$ – их среднее значение (меана). Для вычисления скорректированного стандартного отклонения выборки по формуле

$$\sigma = \sqrt{\frac{1}{N-\delta}\sum_i^N (x_i - \overline{x})^2}$$

в аргумент `ddof` передается значение δ, такое, что $N - \delta$ является числом степеней свободы в конкретной выборке. Например, если значения выборки извлекаются из совокупности независимо с заменой и используются для вычисления $\overline{x}$, то существует $N - 1$ степеней свободы в векторе разностей, используемом для вычисления σ: $(x_1 - \overline{x}, x_2 - \overline{x}, \ldots, x_N - \overline{x})$, следовательно, $\delta = 1$. Например:

```
In [x]: x = np.array([1., 2., 3., 4.])
In [x]: np.std(x)          # Или x.std(), нескорректированное стандартное отклонение.
1.1180339887498949
In [x]: np.std(x, ddof=1)   # Скорректированное стандартное отклонение.
1.2909944487358056
```

Функция `np.nanstd` вычисляет стандартное отклонение без учета значений `np.nan` (поэтому N – это количество не-NaN значений в массиве). В NumPy также имеются методы для вычисления дисперсии значений в массиве: `np.var` и `np.nanvar`.

Ковариацию возвращает метод `np.cov`. При самом простом вызове в него можно передать один двумерный массив `X`, в котором строки представляют переменные x_i, а столбцы – наблюдения значений каждой переменной. Тогда `np.cov(X)` возвращает матрицу ковариации C_{ij}, показывающую, как переменная x_i изменяется по сравнению с переменной x_j. Элемент C_{ij} называют оценкой ковариации переменных x_i и x_j:

$$C_{ij} \equiv \mathrm{cov}(x_i, x_j) = E[(x_i - \mu_i)(x_j - \mu_j)],$$

где μ_i – среднее значение переменной x_i, а $E[\ ]$ обозначает ожидаемое значение. Если существует N наблюдаемых значений для каждой переменной, то $\mu_i = 1/N\sum_k x_{ik}$. Тогда несмещенная оценка ковариации:

$$C_{ij} = 1 / (N - 1) \sum_k [(x_{ik} - \mu_i)(x_{jk} - \mu_j)].$$

Это поведение по умолчанию метода `np.cov`, но если для аргумента `bias` установлено значение 1, то N используется в знаменателе формулы для получения смещенной оценки ковариации. Кроме того, можно явно установить знаменатель равным $N - \delta$, передавая δ как значение аргумента `ddof` метода `cov`.

Пример П6.9. В качестве примера рассмотрим матрицу из пяти наблюдений для каждой из трех переменных x_0, x_1 и x_2, наблюдаемые значения которых содержатся в трех строках массива X:

```
X = np.array([ [0.1, 0.3, 0.4, 0.8, 0.9],
               [3.2, 2.4, 2.4, 0.1, 5.5],
               [10., 8.2, 4.3, 2.6, 0.9]
             ])
```

Ковариационная матрица представлена массивом 3×3:

```
In [x]: print( np.cov(X) )
[[  0.115 ,   0.0575,  -1.2325],
 [  0.0575,   3.757 ,  -0.8775],
 [ -1.2325,  -0.8775,  14.525 ]]
```

Элементы на главной диагонали C_{ii} – это дисперсии переменных x_i с предположением об $N - 1$ степенях свободы:

```
In [x]: print(np.var(X, axis=1, ddof=1))
[  0.115   3.757  14.525]
```

Несмотря на то что величину элементов матрицы ковариации не всегда легко интерпретировать (потому что она зависит от величины отдельных наблюдений, которые могут существенно отличаться для различных переменных), очевидно, что существует сильная антикорреляция между x_0 и x_2 ($C_{02} = -1.2325$: когда одно значение увеличивается, другое уменьшается) и слабая (несильная) корреляция между x_0 и x_1 ($C_{01} = 0.0575$: x_0 и x_1 не показывают сильный совместный тренд).

Матрица коэффициентов корреляции часто используется вместо матрицы ковариации, так как она нормализована посредством деления элементов C_{ij} на произведение стандартных отклонений переменных:

$$P_{ij} = \mathrm{corr}(x_i, x_j) = C_{ij} / \sigma_i \sigma_j = C_{ij} / \sqrt{(C_{ii} C_{jj})}.$$

Это означает, что элементы P_{ij} имеют значения от –1 до 1 включительно, а диагональные элементы $P_{ii} = 1$. В рассматриваемом здесь примере использование метода `np.corrcoef` дает следующий результат:

```
In [x]: print( np.corrcoef(X) )
[[  1.          0.0874779  -0.95363007]
 [  0.0874779   1.         -0.11878687]
 [ -0.95363007 -0.11878687  1.        ]]
```

В этой матрице коэффициентов корреляции легко заметить сильную антикорреляцию между x_0 и x_2 ($C_{0,2} = -0.954$) и отсутствие корреляции между x_1 и другими переменными (например, $C_{1,0} = 0.087$).

Оба метода `np.cov` и `np.corrcoeff` могут принимать второй объект типа массив, содержащий дополнительный набор переменных и наблюдений, поэтому их можно назвать парой одномерных массивов без объединения в единую матрицу:

```
In [x]: x = np.array([1., 2., 3., 4., 5.])
In [x]: y = np.array([0.08, 0.31, 0.41, 0.48, 0.62])
In [x]: print( np.corrcoef(x,y) )
[[ 1.          0.97787645]
 [ 0.97787645  1.        ]]
```

Таким образом:

```
np.corrcoef(x, y)
```

представляет собой удобный альтернативный вариант для

```
np.corrcoef(np.vstack((x,y)))
```

А если наблюдения располагаются в строках матрицы в переменных, соответствующих столбцам (вместо других позиций), то нет необходимости в транспонировании этой матрицы, нужно просто передать аргумент `rowvar=0` либо в `np.cov`, либо в `np.corrcoef`, а NumPy позаботится обо всем.

Пример П6.10. Отделение цифровых технологий Кембриджского университета (Cambridge University Digital Technology Group) ведет записи наблюдений за погодой с крыши своего корпуса с 1995 г. Эти данные доступны для скачивания в одном CSV-файле здесь: www.cl.cam.ac.uk/research/dtg/weather/.

Программа в листинге 6.7 определяет коэффициент корреляции между атмосферным давлением и температурой в этой местности.

Листинг 6.7. Вычисление коэффициента корреляции между температурой воздуха и атмосферным давлением

```
# eg6-pT.py
import numpy as np
import matplotlib.pyplot as plt

data = np.genfromtxt('weather -raw.csv', delimiter=',', usecols=(1, 4))
# Удаление всех строк, в которых пропущено значение T или значение p.
data = data[~np.any(np.isnan(data), axis=1)]
# Значения температуры фиксируются после умножения на 10, поэтому необходимо исключить
# этот множитель.
data[:,0] /= 10

# Вычисление коэффициента корреляции.
corr = np.corrcoef(data , rowvar=0)[0, 1]
print('p-T correlation coefficient: {:.4f}'.format(corr))
```

```
# Изображение данных на корреляционной диаграмме рассеяния: T по оси x, p по оси y.
plt.scatter(*data.T, marker='.')
plt.xlabel('$T$ /$\mathrm{^\circ C}$')
plt.ylabel('$p$ /mbar')
plt.show()
```

Вывод (см. рис. 6.4) дает коэффициент корреляции 0.0260: как и ожидалось, существует весьма слабая корреляция между температурой воздуха и атмосферным давлением (поскольку плотность воздуха также изменяется).

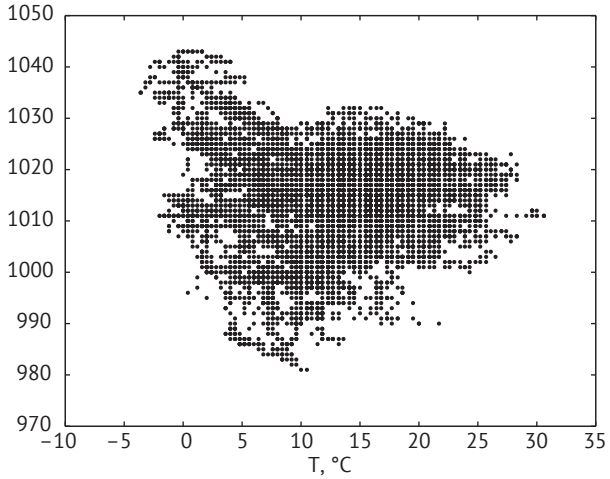


Рис. 6.4. В этом наборе данных корреляция между температурой воздуха и атмосферным давлением практически отсутствует

6.3.3 Гистограммы

Функция NumPy `np.histogram` создает гистограмму из значений массива. Набор интервалов (bins) определяется по нижнему и верхнему предельным значениям, и каждый интервал заполняется определенным количеством элементов массива, значения которых попадают в границы интервала. Например, рассмотрим следующий массив, содержащий оценки в процентах 10 студентов на экзамене:

```
In [x]: marks = np.array([45, 68, 56, 23, 60, 87, 75, 59, 63, 72])
```

Существует несколько способов определения интервалов гистограммы. Если аргумент `bins` является последовательностью, то ее значения определяют границы последовательных интервалов:

```
In [x]: bins = [20, 40, 60, 80, 100]
```

Эта последовательность определяет четыре интервала с диапазонами [20–40 %), [40–60 %), [60–80 %) и [80–100 %]. Все интервалы, кроме последнего, полуоткрытые, т. е. первый интервал включает метки от 20 %, в том числе саму

эту метку, до 40 %, не включая само это значение. Следует отметить, что для последовательности из $N + 1$ чисел требуется создание N интервалов. Метод `np.histogram` возвращает кортеж из значений гистограммы и ранее определенных границ интервалов (оба набора в виде массивов NumPy).

```
In [x]: hist, bins = np.histogram(marks, bins)
In [x]: hist
Out[x]: array([1, 3, 5, 1])

In [x]: bins
Out[x]: array([ 20, 40, 60, 80, 100])
```

Здесь можно видеть, что в интервале 20–4% имеется одна метка, в интервале 40–60 % находятся три метки и т. д.

Если необходимо создать определенное количество равномерно распределенных интервалов, то в аргументе `bins` вместо последовательности можно передать одно целое число:

```
In [x]: np.histogram(marks, bins=5)
Out[x]: (array([1, 1, 3, 3, 2]),
         array([ 23. ,  35.8,  48.6,  61.4,  74.2,  87. ]))
```

По умолчанию требуемое количество интервалов распределяется между минимальным и максимальным значениями массива (в показанном выше примере это значения 23 и 87). Чтобы определить другие значения минимума и максимума, необходимо передать соответствующий кортеж в аргументе `range`:

```
In [x]: np.histogram(marks, bins=5, range=(0, 100))
Out[x]: (array([0, 1, 3, 5, 1]),
         array([   0.,   20.,   40.,   60.,   80.,   100.]))
```

Метод `np.histogram` также принимает необязательный аргумент `density`: по умолчанию задано значение `False`, означающее, что возвращаемый массив гистограммы содержит количество значений из исходного массива в каждом интервале. Если для `density` задано значение `True`, то массив гистограммы будет содержать функцию плотности вероятности, нормализованную так, что интеграл по всей площади диапазона интервалов равен единице:

```
In [x]: hist, bins = np.histogram(marks, bins=5, range=(0,100),
                                  density=True)
In [x]: print(hist)
[ 0.     0.005  0.015  0.025  0.005]
In [x]: bin_width = 100/5
In [x]: print(np.sum(hist) * bin_width)
1.0
```

(Здесь под интегралом подразумевается площадь внутри гистограммы, представленная суммой произведений высоты каждого интервала на его ширину.)

Для вывода гистограммы с помощью `pyplot` используется метод `pyplot.hist` с передачей в него тех же аргументов, которые передавались в метод `np.histogram`:

```
In [x]: import matplotlib.pyplot as plt
In [x]: hist, bins, patches = plt.hist(marks, bins=5, range=(0, 100))   ❶
In [x]: hist, bins
Out[x]:
(array([ 0.,  1.,  3.,  5.,  1.]),
 array([   0.,   20.,   40.,   60.,   80.,   100.]))
In [x]: plt.show()
```

❶ В дополнение к счетчикам (`hist`) и границам (`bins`) метод `pyplot` возвращает список ссылок на «патчи» (patches), которые появляются на изображаемой фигуре (более подробно об этой дополнительной возможности см. раздел 7.7.4).

Полученная гистограмма изображена на рис. 6.5. См. также разделы 3.3.2 и 7.3.

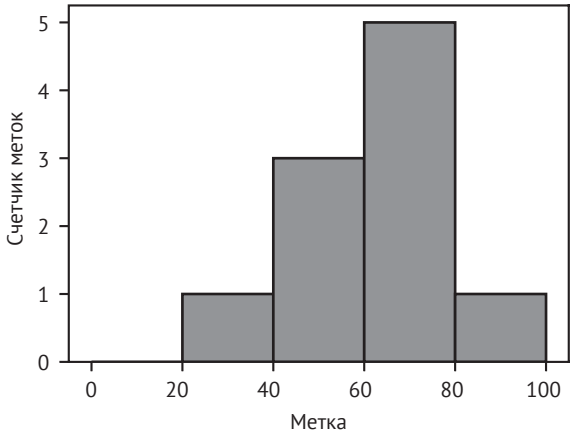


Рис. 6.5. Пример гистограммы

6.3.4 Упражнения

Задачи

36.3.1. В некоторой лотерее участвуют игроки, выбирающие шесть чисел без замены из интервала [1, 49]. Джекпот разделяется между игроками, угадавшими все шесть чисел («выпавших лотерейных шаров»), выбранных случайным образом в розыгрыше, который проводится каждые две недели (порядок угаданных чисел не имеет значения). Если ни один из игроков не угадал каждое выпавшее число, то джекпот считается неразыгранным и добавляется к джекпоту следующего розыгрыша.

Несмотря на то что эта лотерея честная в том смысле, что каждая комбинация случайно выбираемых чисел является равновероятной, было сделано следующее наблюдение: многие игроки отдают предпочтение выбору определенных чисел, например представляющих даты (т. е. большинство выбираемых чисел находится в интервале [1, 31], как будто эти числа ожидаются в большей степени, чем выбираемые случайно). Поэтому, чтобы избежать дележа джекпота, следовательно, получить максимальный выигрыш в одиночку, имеет смысл не использовать эти числа.

Проверить это предположение, установив, существует ли какая-либо связь (корреляция) между количеством шаров со значениями меньше 13 (представляющих номер месяца) и выигрышами джекпота одним человеком. Не учитывать розыгрыши, которые приводили к увеличению джекпота (т. е. без выигравших текущий джекпот). Необходимые данные можно скачать здесь: https://scipython.com/ex/bfe.

36.3.2. В этом разделе мы видели, как создать графическое изображение гистограммы из массива с помощью метода `pyplot.hist`, но предположим, что массивы `hist` и `bins` уже созданы с использованием метода `np.histogram` и необходимо построить итоговую гистограмму по этим массивам. Невозможно воспользоваться `pyplot.hist`, потому что эта функция ожидает для обработки массив исходных данных. Необходимо использовать `pyplot.bar`[81] для создания изображения по массиву `hist` в виде столбиковой диаграммы (bar chart, собственно, это и есть гистограмма).

36.3.3. Значения роста в сантиметрах в выборке из 1000 взрослых мужчин и 1000 взрослых женщин из определенной совокупности собраны в файлах данных *ex6-3-f-male-heights.txt* и *ex6-3-f-female-heights.txt*, доступных здесь: https://scipython.com/ex/bfd. Прочитать эти данные и вычислить среднее значение и стандартное отклонение по каждому полу. Создать гистограммы для этих двух наборов данных с использованием наиболее подходящего интервала и показать их графически на одном изображении.

Решить эту же задачу в единицах британской системы (в футах и дюймах).

6.4 Многочлены

Библиотека NumPy предоставляет обширный набор классов для представления многочленов (полиномов), включающих методы для вычисления многочленов, для алгебры многочленов, для вычисления корней и подгонки к нескольким типам основных полиномиальных функций. В этом разделе сначала рассматривается самый простой и наиболее известный основной объект – степенной ряд до перехода к обсуждению некоторых других канонических функций ортогонального полиномиального базиса.

6.4.1 Определение и вычисление многочлена

Полиномиальный (конечный) степенной ряд в качестве базиса использует степени x: 1 (= x^0), x, x^2, x^3, …, x^N с коэффициентами c_i:

$$P(x) = \sum_{i=0}^{N} c_0 + c_1 x + c_2 x^2 + c_3 x^3 + \ldots + c_N x^N.$$

[81] Документацию по применению этого метода см. здесь: https://matplotlib.org/api/_as_gen/matplotlib.pyplot.bar.html. Также см. раздел 7.3.

В этом разделе рассматривается использование обобщенного класса `Polynomial`, который предоставляет удобный естественный интерфейс во внутренней функциональности пакета `polynomial` библиотеки NumPy.

Обобщенный класс многочлена `numpy.polynomial.Polynomial`. Чтобы импортировать его напрямую, используйте инструкцию:

```
In [x]: from numpy.polynomial import Polynomial
```

Или если вся библиотека NumPy уже импортирована как np, то вместо того, чтобы часто обращаться к этому классу как `np.polynomial.Polynomial`, удобнее определить переменную:

```
In [x]: import numpy as np
In [x]: Polynomial = np.polynomial.Polynomial
```

Именно так мы будем ссылаться на класс `Polynomial` в этой книге.

Для определения объекта многочлена необходимо передать в конструктор `Polynomial` последовательность коэффициентов для постепенно возрастающих степеней x, начиная с c_0. Например, для представления многочлена

$$P(x) = 6 - 5x + x^2$$

определяется объект

```
In [x]: p = Polynomial([6, -5, 1])
```

Проверить коэффициенты объекта `Polynomial` можно командой `print` или обращением к его атрибуту `coef`.

```
In [x]: print(p)
poly([ 6. -5. 1.])
In [x]: p.coef
Out[x]: array([ 6., -5., 1.])
```

Обратите внимание: целочисленные коэффициенты, используемые для определения многочлена, были автоматически преобразованы в тип `float`. Также возможно использование комплексных коэффициентов.

Для вычисления многочлена при заданном значении `x` необходимо «вызвать» многочлен, как показано ниже:

```
In [x]: p(4)                  # Вычисление многочлена p при заданном одиночном значении x.
2.0
In [x]: x = np.linspace(-5, 5, 11)
In [x]: print(p(x))           # Вычисление p при заданной последовательности значений x.
Out[x]: [ 56.  42.  30.  20.  12.   6.   2.   0.   0.   2.   6.]
```

6.4.2 Алгебра многочленов

В обобщенном классе `Polynomial` реализованы знакомые операторы Python: `+`, `-`, `*`, `//`, `**`, `%` и `divmod` для объектов типа `Polynomial`. Использование этих операторов показано в следующих примерах, работающих с многочленами

$$P(x) = 6 - 5x + x^2,$$
$$Q(x) = 2 - 3x.$$

```
In [x]: p = Polynomial([6, -5, 1])
In [x]: q = Polynomial([2, -3])
In [x]: print(p + q)
poly([ 8. -8. 1.])

In [x]: print(p - q)
poly([ 4. -2. 1.])

In [x]: print(p * q)
poly([ 12. -28. 17. -3.])

In [x]: print(p // q)
poly([ 1.44444444 -0.33333333])

In [x]: print(p % q)
poly([ 3.11111111]) # T. e. 28/9
```

Деление многочлена на другой многочлен выполняется так же, как целочисленное деление (и используется тот же оператор `//`): т. е. результатом является третий многочлен (без обратных степеней `x`), возможно, с остатком от деления.

Таким образом, $p = q(-1/3x + 13/9) + 28/9$, и оператор `//` возвращает многочлен-частное $-1/3x + 13/9$. Остаток (который в общем случае может быть еще одним многочленом) возвращается, как можно было предположить, оператором взятия модуля `%`. Встроенный метод `divmod()` возвращает частное и остаток в кортеже:

```
In [x]: quotient, remainder = divmod(p, q)
In [x]: print(quotient)

poly([ 1.44444444 -0.33333333])       # То есть p(x) // q(x) равно 13/9 - x/3

In [x]: print(remainder)
poly([ 3.11111111])
```

Возведение в степень поддерживается с помощью оператора `**`, многочлены можно возводить только в неотрицательную целую степень:

```
In [x]: print(q ** 2)
poly([  4. -12.   9.])
```

Не всегда удобно создавать новый объект многочлена, чтобы воспользоваться этими операторами совместно с другим многочленом, поэтому многие из операторов, описанных выше, также работают со скалярными значениями:

```
In [x]: print(p * 2) # Умножение на скаляр.
poly([ 12. -10.   2.])

In [x]: print(p / 2) # Деление на скаляр.
poly([ 3.  -2.5  0.5])
```

и даже с кортежами, списками и массивами коэффициентов многочлена. Например, для умножения многочлена $P(x)$ на $x^2 - 2x^3$:

```
In [x]: print(p * [0, 0, 1, -2])
poly([  0.   0.   6. -17.  11.  -2.])
```

Один многочлен можно подставить в другой. Для вычисления $P(Q(x))$ просто используется выражение `p(q)`:

```
In [x]: print(p(q))
poly([ 0.  3.  9.])
```

Таким образом, $P(Q(x)) = 3x + 9x^2$.

6.4.3 Поиск корней многочлена

Корни многочлена возвращает метод `roots`. Одинаковые корни просто повторяются в возвращаемом массиве:

```
In [x]: p.roots()
array([ 2.,  3.])
In [x]: (q * q).roots()
array([ 0.66666667,  0.66666667])
In [x]: Polynomial([5, 4, 1]).roots()
array([-2.-1.j, -2.+1.j])
```

Многочлены можно также создавать по их корням с помощью метода `Polynomial.fromroots`:

```
In [x]: print( Polynomial.fromroots([-4, 2, 1]) )
poly([  8. -10.   1.   1.])
```

То есть $(x + 4)(x - 2)(x - 1) = 8 - 10x + x^2 + x^3$. Обратите внимание: при таком способе формирования многочлена коэффициент при наивысшей степени x всегда будет равен 1.

Пример П6.11. Емкости для хранения криогенных жидкостей и ракетного топлива часто имеют сферическую форму (почему?). Предположим, что конкретная сферическая емкость имеет радиус R и заполняется жидкостью до высоты h. Формула объема жидкости по высоте заполнения выводится (относительно) просто:

$$V = \pi Rh^2 - 1/3\pi h^3.$$

Предположим, что из этой емкости постоянно вытекает жидкость со скоростью потока $F = -dV/dt$. Как изменяется во времени высота заполнения жидкости h? Дифференцирование приведенного выше равенства по переменной времени t приводит к следующему результату:

$$(2\pi Rh - \pi h^2)dh/dt = -F.$$

Если начать с заполненной емкости ($h = 2R$) в момент времени $t = 0$, то это обыкновенное дифференциальное уравнение можно проинтегрировать и получить следующее выражение:

$$-1/3\pi h^3 + \pi Rh^2 + (Ft - 4/3\pi R^3) = 0,$$

т. е. кубический многочлен с переменной h. Поскольку это уравнение невозможно аналитически преобразовать относительно h, воспользуемся NumPy классом `Polynomial` для поиска $h(t)$, приняв радиус емкости $R = 1.5$ м и скорость вытекания жидкости 200 см^3/с.

Общий объем жидкости в полной емкости равен $V_0 = 4/3\pi R^3$. Очевидно, что емкость пуста, когда $h = 0$, что происходит в момент времени $T = V_0/F$, так как скорость потока постоянная. В любой произвольно взятый момент времени t можно вычислить h, определив корни приведенного выше уравнения.

Листинг 6.8. Определение высоты жидкости в сферической емкости

```
# eg6-c-spherical -tank -a.py
import numpy as np
import matplotlib.pyplot as plt
Polynomial = np.polynomial.Polynomial

# Радиус сферической емкости в м.
R = 1.5
# Скорость вытекания жидкости из емкости в м^3/с.
F = 2.e-4
# Общий объем емкости.
V0 = 4/3 * np.pi * R**3
# Общее время, за которое емкость становится пустой.
T = V0 / F

# Коэффициенты для членов в квадрате и в кубе, содержащихся
# в p(h) - многочлене, который необходимо решить относительно h.
c2, c3 = np.pi * R, -np.pi / 3

N = 100
# Массив из N меток времени от 0 до T включительно.
time = np.linspace(0, T, N)                                          ❶
# Создание соответствующего массива значений высоты h(t).
h = np.zeros(N)
for i, t in enumerate(time):
    c0 = F*t - V0
    p = Polynomial([c0, 0, c2, c3])
    # Вычисление трех корней этого многочлена.
    roots = p.roots()                                                ❷
    # Требуется один корень, для которого 0 <= h <= 2R.
    h[i] = roots[(0 <= roots) & (roots <= 2*R)][0]

plt.plot(time , h, 'o')
plt.xlabel('Time /s')
plt.ylabel('Height in tank /m')
plt.show()
```

❶ Создается массив меток времени от $t = 0$ до $t = T$.

❷ Для каждой метки времени вычисляются корни приведенного выше кубического многочлена. Только один из корней имеет физический смысл, а именно: $0 \leqslant h \leqslant 2R$ (высота уровня жидкости не может быть отрицательной или большей диаметра емкости), этот корень извлекается (с помощью логического индексирования) и сохраняется в массиве h.

На завершающем этапе изображается изменение *h* как функции от времени (см. рис. 6.6).

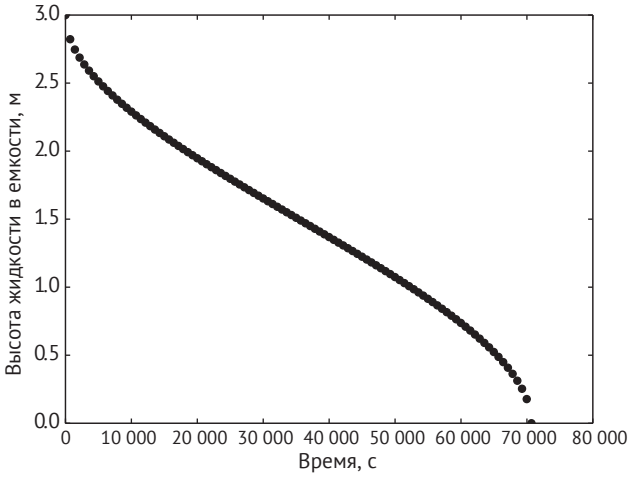


Рис. 6.6. Высота уровня жидкости как функция от времени *h*(*t*) для задачи о сферической емкости

6.4.4 Математический анализ

Многочлены можно дифференцировать с помощью метода `Polynomial.deriv`. По умолчанию этот метод возвращает первую производную, но в необязательном аргументе `m` можно определить возврат *m*-й производной:

```
In [x]: print(p)
poly([ 6. -5.  1.])      # Многочлен 6 - 5x + x^2.
In [x]: print(p.deriv())
poly([-5.  2.])
In [x]: print(p.deriv(2))
poly([ 2.])
```

Объект `Polynomial` можно также интегрировать с необязательной нижней границей *L* и постоянной интегрирования *k*, интерпретируемой, как показано в следующем примере:

$$\int_L^x 2-3xdx = \left[2x-\frac{3}{2}x^2\right]_L^x = 2x-\frac{3}{2}x^2-2L+\frac{3}{2}L^2,$$

$$\int 2-3xdx = 2x-\frac{3}{2}x^2+k.$$

По умолчанию *L* и *k* равны нулю, но их значения можно определить, передавая в аргументах `lbnd` и `k` в метод `Polynomial.integ`:

```
In [x]: print(q)
poly([ 2. -3.])
In [x]: print(q.integ())
poly([ 0.   2.  -1.5])
In [x]: print(q.integ(lbnd=1))
poly([-0.5  2.  -1.5])
In [x]: print(q.integ(k=2))
poly([ 2.   2.  -1.5])
```

Многочлены можно интегрировать многократно, передавая значение в аргументе `m` – количество требуемых операций интегрирования[82].

6.4.5 ◊ Классические ортогональные многочлены

В дополнение к классу `Polynomial`, представляющему простые степенные ряды, такие как $a_0 + a_1x + a_2x^2 + \ldots + a_nx^n$, библиотека NumPy предлагает классы для представления рядов, сформированных из любых классических ортогональных многочленов. Эти многочлены и их линейные комбинации широко используются в физике, статистике и математике. В текущей версии NumPy 1.17 представлены обобщенные классы многочленов: `Chebyshev`, `Legendre`, `Laguerre`, `Hermite` («версия для физиков») и `HertmiteE` («версия для вероятностных вычислений»). Есть много хороших книг, подробно описывающих свойства этих классов многочленов. Чтобы продемонстрировать их практическое использование, сосредоточимся на многочленах Лежандра[83], обозначаемых как $P_n(x)$. Они представляют собой решения дифференциального уравнения Лежандра:

$$\frac{d}{dx}\left[(1-x^2)\frac{d}{dx}P_n(x)\right]+n(n+1)P_n(x)=0.$$

Ниже приведено несколько первых многочленов Лежандра:

$$P_0(x) = 1,$$
$$P_1(x) = x,$$
$$P_2(x) = 1/2(3x^2 - 1),$$
$$P_3(x) = 1/2(5x^3 - 3x),$$
$$P_4(x) = 1/8(35x^4 - 30x^2 + 3).$$

Графики этих многочленов изображены на рис. 6.7.

---

[82] Могут быть заданы различные константы для каждой операции интегрирования с помощью массива значений, передаваемого в аргументе *k*.

[83] Многочлены Лежандра названы в честь французского математика Адриена-Мари Лежандра (Adrien-Marie Legendre) (1752–1833). В течение 200 лет до 2005 г. во многих публикациях ошибочно использовался портрет французского политика Луи Лежандра (Louis Legendre), который не имел никакого отношения к Лежандру-математику.

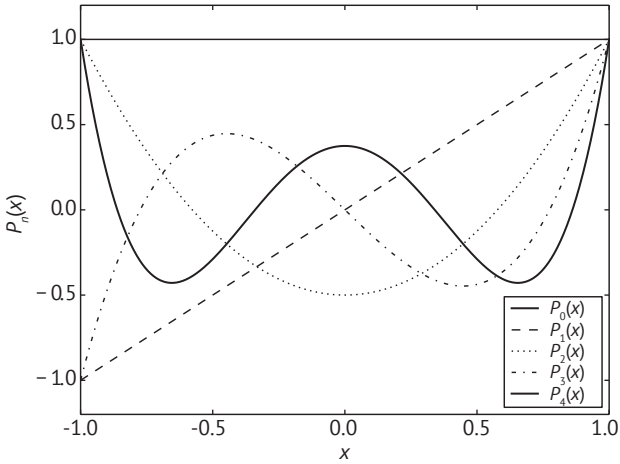


Рис. 6.7. Пять первых многочленов Лежандра *Pn*(*x*) при *n* = 0, 1, 2, 3, 4

Полезным свойством многочленов Лежандра является их ортогональность на интервале [−1, 1]:

$$\int_{-1}^{1} P_n(x)P_m(x)dx = \frac{2}{2n+1}\delta_{mn},$$

которая важна при их использовании в качестве основы для представления близко соответствующих функций[84].

Для создания линейной комбинации многочленов Лежандра коэффициенты передаются в конструктор класса `Legendre` точно так же, как и в конструктор `Polynomial`. Например, для создания разложения $5P_1(x) + 2P_2(x)$:

```
In [x]: Legendre = np.polynomial.Legendre
In [x]: A = Legendre([0, 5, 2])
```

Существующий объект многочлена можно преобразовать в ряд Лежандра с помощью метода `cast`:

```
In [x]: P = Polynomial([0, 1, 1])
In [x]: Q = Legendre.cast(P)
In [x]: print(Q)
leg([ 0.33333333  1.          0.66666667])
```

То есть $x + x^2 = 1/3P_0 + P_1 + 2/3P_2$.

Экземпляр одной базисной функции многочлена Лежандра можно создать, используя метод `basis`:

```
In [x]: L3 = Legendre.basis(3)
```

[84] В особенности в физике: для мультипольного разложения электростатических потенциалов.

Здесь создается объект, представляющий $P_3(x)$. Это равнозначно вызову `Legendre([0, 0, 0, 1])`. Для получения обычного степенного ряда можно выполнить обратное преобразование в объект `Polynomial`:

```
In [x]: print(Polynomial.cast(L3))
poly([ 0.  -1.5  0.   2.5])
```

В дополнение к описанным выше функциям объекта `Polynomial`, включая методы дифференцирования и интегрирования многочленов, обобщенные классы для классических ортогональных многочленов предлагают несколько полезных методов.

Метод `convert` выполняет преобразование между различными типами многочленов. Например, линейная комбинация $A(x) = 5P_1(x) + 2P_2(x) = 5x + 2½(3x^2 - 1) = -1 + 5x + 3x^2$ как степенной ряд одночленов (ряд Маклорена) представляется экземпляром класса `Polynomial` в следующем виде:

```
In [x]: A = Legendre([0, 5, 2])
In [x]: B = A.convert(kind=Polynomial)
In [x]: print(B)
In [x]: poly([-1.  5.  3.])
```

Так как объекты `A` и `B` представляют одну и ту же базисную функцию (просто она разложена на различные базисные множества), при вычислении они дают одинаковое значение при одинаковом заданном x и имеют одинаковые корни:

```
In [x]: A(-2) == B(-2)
Out[x]: True
In [x]: print(A.roots(), B.roots(), sep='\n')
[-1.84712709  0.18046042]
[-1.84712709  0.18046042]
```

6.4.6 Подгонка к многочленам

Широко распространенное практическое применение разложений на многочлены – подгонка и аппроксимация последовательностей данных. Модули многочленов библиотеки NumPy предоставляют методы для подгонки методом наименьших квадратов для функций. В этом разделе рассматривается функция `fit` обобщенных классов многочленов[85].

Атрибуты *domain* и *window*

В обычной одномерной задаче подгонки требуется многочлен, наилучшим образом соответствующий конечной непрерывной функции на некотором конечном интервале оси x (называемом доменом – domain). Но сами многочлены могут существенно отличаться друг от друга и расходиться при $x \to \pm\infty$. Поэтому

[85] Примечание: более старый класс `np.poly1d`, представляющий одномерные многочлены, остается доступным в версии 1.17 NumPy для обеспечения обратной совместимости. Документация здесь: https://docs.scipy.org/doc/numpy/reference/routines.polynomials.poly1d.html. Этот класс предлагает более простую, но менее надежную функцию подгонки методом наименьших квадратов `np.polyfit`. Но при написании нового кода рекомендуется пользоваться новым классом `Polynomial`.

существует риск при любой попытке вслепую применить подгонку методом наименьших квадратов непосредственно в области определения функции: подгоняемый многочлен часто становится причиной нестабильности вычислений, переполнения, потери значимости и прочих типов некорректности и плохой обусловленности (см. раздел 10.2). В качестве примера рассмотрим функцию

$$f(x) = e^{-\sin 40x}$$

на интервале (100, 100.1). В этой функции нет ничего особенного: ее поведение хорошо известно на всей области определения, и $f(x)$ принимает вполне предсказуемые значения в интервале от e^{-1} до e^{1}. Тем не менее прямая подгонка методом наименьших квадратов к многочлену четвертого порядка в указанном интервале (домене) дает следующий результат:

$$-11.881851 + 2379.22228x - 119.741202x^2 - 23828009.7x^3 + 1192894610x^4,$$

который очевидно является потенциальным источником вычислительной нестабильности и потери точности даже при обычных значениях x: эта аппроксимация $f(x)$ формируется из разностей весьма больших одночленов.

Каждый класс многочлена по умолчанию имеет окно (window), представляющее оптимальный интервал для формирования линейной комбинации при подгонке функции. Например, окном многочленов Лежандра является интервал [−1, 1], на котором строились графики в предыдущем разделе и в котором $P_n(x)$ ортогональны и $|P_n(x) < 1|$. Проблема заключается в том, что подгоняемая функция не соответствует окну выбранного многочлена. Следовательно, необходимо привести в соответствие область определения функции и окна многочлена. Это делается с помощью сдвига и масштабирования по оси x: т. е. посредством отображения точек из области определения функции в точки, находящиеся в окне подгоняемого многочлена. Функция многочлена `fit` делает это автоматически, так что для упомянутой выше функции подгонка методом наименьших квадратов к многочлену четвертой степени дает следующий результат:

```
In [x]: x = np.linspace(100, 100.1, 1001)
In [x]: f = lambda x: np.exp(-np.sin(40*x))
In [x]: p = Polynomial.fit(x, f(x), 4)
In [x]: print(p)
poly([ 1.49422551 -2.54641449  0.63284641  1.84246463 -1.02821956])
```

Домен и окно многочлена можно проверить, обратившись к атрибутам `domain` и `window` соответственно:

```
In [x]: p.domain
array([ 100. ,  100.1])
In [x]: p.window
array([-1.,  1.])
```

Важно отметить, что аргумент `x` отображается из домена в окно всегда при вычислении многочлена. Это означает, что два многочлена с различными доменами и/или окнами при вычислении могут давать различные значения,

даже если их коэффициенты одинаковы. Например, если заново создается объект `Polynomial` с теми же коэффициентами, что и подогнанный многочлен `p` в примере, приведенном выше:

```
In [x]: q = Polynomial ([1.49422551,  -2.54641449,   0.63284641,
                         1.84246463,  -1.02821956])
```

то новый многочлен имеет домен и окно, определенные по умолчанию, т. е. оба `(-1, 1)`:

```
In [x]: print(q.domain , q.window)
[-1.  1.] [-1.  1.]
```

поэтому вычисление многочлена `q`, например, в точке 100.05 отображает точку 100.05 в домене в точку 100.05 в окне и дает совершенно другой результат при вычислении `p` в той же точке его домена (которая отображается в точку `0.` окна):

```
In [x]: q(100.05), p(100.05)
(-101176442.96772559, 1.4942255113760108)
```

Легко показать, что функция отображения x в домене (a,b) в x' в окне (a',b') имеет следующий вид:

$$x' = m(x) = \chi + \mu x, \text{ где } \mu = (b' - a')/(b - a),\ \chi = b' - b(b' - a')/(b - a).$$

Это параметры, возвращаемые функцией многочлена `mapparams`:

```
In [x]: chi, mu = p.mapparms()
In [x]: print(chi, mu)
-2001.0, 20.0
```

Таким образом:

```
In [x]: print(q(chi + mu*100.05))
1.49422551
```

Значения атрибутов `domain` и `window` можно изменить прямым присваиванием:

```
In [x]: q.domain = np.array((100., 100.1))
In [x]: print(q(100.05))
1.49422551
```

Для вычисления многочлена с набором чисел, равномерно распределенных по точкам в его домене, например для построения графика многочлена, используется метод класса `Polynomial linspace`:

```
In [x]: p.linspace(5)
Out [x]:
(array([ 100.   ,  100.025,  100.05 ,  100.075,  100.1 ]),
 array([ 1.80280222,  2.63107256,  1.49422551,  0.54527422,  0.39490249]))
```

Метод `p.linspace` возвращает два массива с заданным количеством точек в домене многочлена, представляющих точки x, и со значениями многочлена, вычисленными в этих точках $p(x)$.

Метод *Polynomial.fit*

Метод класса `Polynomial` `fit` возвращает многочлен, подогнанный методом наименьших квадратов, `y` к данным из выборки значений `x`. В самом простом варианте использования для метода `fit` требуется передача только объектов типа массив `x` и `y` и значения `deg` – степени подгоняемого многочлена. Метод возвращает многочлен, который минимизирует сумму квадратических ошибок:

$$E = \sum_i |y_i - p(x_i)|^2.$$

Например:

```
In [x]: x = np.linspace(400, 700, 1000)
In [x]: y = 1 / x**4
In [x]: p = Polynomial.fit(x, y, 3)
```

создает кубический многочлен, подогнанный наилучшим образом к функции x^{-4} на интервале [400, 700].

Взвешенная подгонка методом наименьших квадратов определяется передачей в аргументе `w` последовательности взвешенных значений, имеющей длину, равную длине массивов `x` и `y`. Возвращается многочлен, минимизирующий сумму взвешенных квадратических ошибок:

$$E = \sum_i w_i^2 |y_i - p(x_i)|^2.$$

Домен и окно подгоняемого многочлена можно определить в аргументах `domain` и `window`, по умолчанию минимальный домен включает все используемые точки `x`.

Разумным действием является проверка качества подгонки перед использованием возвращенного многочлена. Установка аргумента `full=True` заставляет `fit` возвращать два объекта: подогнанный многочлен и список различных статистических данных о самой операции подгонки:

```
In [x]: deg = 3
In [x]: p, [resid , rank , sing_val , rcond] = Polynomial.fit(x, y, deg, full=True)
In [x]: p
Out[x]:
Polynomial([  1.07041864e-11,    -1.16488662e-11,    1.02545751e-11,
        -5.64068914e-12], [ 400.,    700.], [-1.,   1.])

In [x]: resid
Out[x]: array([ 4.57180972e-23])

In [x]: rank
Out[x]: 4
```

```
In [x]: sing_val
Out[x]: array([ 1.3843828 ,  1.32111941,  0.50462215,  0.28893641])

In [x]: rcond
Out[x]: 2.2204460492503131e-13
```

Этот список можно проанализировать, чтобы узнать, насколько точно функция-многочлен соответствует исходным данным. Значение `resid` – это сумма квадратов разностей:

$$\texttt{resid} = \sum_i |y_i - p(x_i)|^2.$$

Здесь меньшее значение `resid` означает более точную подгонку. Значения `rank` и `sing_val` – это ранг и сингулярные значения матрицы, инвертированной в алгоритме наименьших квадратов для поиска коэффициентов многочлена: плохая обусловленность этой матрицы может привести к весьма неточной подгонке (особенно если степень подгоняемого многочлена слишком высока). Значение `rcond` – это коэффициент выравнивания (отсечения) для малых отдельных значений в этой матрице: значения, меньшие этого коэффициента, устанавливаются в ноль в процессе подгонки (для защиты от ложных артефактов, возникающих при ошибке округления), и генерируется исключение `RankWarning`. Если это происходит, то, возможно, данные слишком загрязнены (в них весьма высок уровень шума) или не точно описываются многочленом заданной степени. Следует отметить, что подгонка методом наименьших квадратов должна всегда выполняться с двойной точностью, а кроме того, следует помнить о переподгонке данных (попытке установить соответствие с функцией со слишком большим количеством коэффициентов, т. е. с многочленом слишком высокого порядка).

Пример П6.12. Прямолинейная наилучшая подгонка – это просто особый случай полиномиальной подгонки методом наименьших квадратов (при `deg=1`). Рассмотрим следующие данные, полученные при поглощении (абсорбции) A на пути длиной 5 мм, который проходит ультрафиолетовый свет с длиной волны 280 нм, протеином, как функцию от концентрации [*P*]:

Таблица 6.7

[*P*], мкг/мл	*A*
0	2.287
20	3.528
40	4.336
80	6.909
120	8.274
180	12.855
260	16.085
400	24.797
800	49.058
150	89.400

Предполагается, что поглощение линейно связано с концентрацией протеина: $A = m[P] + A_0$, где A_0 – поглощение при отсутствии протеина (например, из-за раствора и экспериментальных компонентов).

Листинг 6.9. Прямолинейная подгонка данных о поглощении

```
# eg6-polyfit.py
import numpy as np
import matplotlib.pyplot as plt
Polynomial = np.polynomial.Polynomial

# Данные: концентрация conc = [P] и поглощение A.
conc = np.array([0, 20, 40, 80, 120, 180, 260, 400, 800, 1500])
A = np.array([2.287, 3.528, 4.336, 6.909, 8.274, 12.855, 16.085, 24.797,
              49.058, 89.400])

cmin, cmax = min(conc), max(conc)
pfit, stats = Polynomial.fit(conc, A, 1, full=True, window=(cmin , cmax),
                                                    domain=(cmin , cmax))

print('Raw fit results:', pfit , stats , sep='\n')

A0, m = pfit
resid, rank, sing_val, rcond = stats
rms = np.sqrt(resid[0]/len(A))

print('Fit: A = {:.3f}[P] + {:.3f}'.format(m, A0),
      '(rms residual = {:.4f})'.format(rms))

plt.plot(conc, A, 'o', color='k')
plt.plot(conc, pfit(conc), color='k')
plt.xlabel('[P] /$\mathrm{\mu g\cdot mL^{-1}}$')
plt.ylabel('Absorbance')
plt.show()
```

Вывод этой программы показывает качественную прямолинейную подгонку для исходных данных (также см. рис. 6.8):

```
Raw fit results:
poly([ 1.92896129  0.0583057 ])
[array([ 2.47932733]), 2, array([ 1.26633786,  0.62959385]), 2.2204460492503131e-15]
Fit: A = 0.058[P] + 1.929 (rms residual = 0.4979)
```

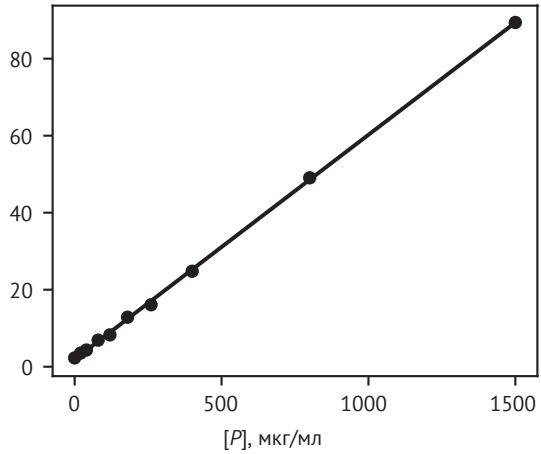


Рис. 6.8. Прямая как результат наилучшей подгонки методом наименьших квадратов для данных о поглощении как функции от концентрации

6.4.7 Упражнения

Вопросы

B6.4.1. Третья производная от функции многочлена $P(x) = 3x^3 + 2x - 7$ равна 18, но почему приведенное ниже вычисление дает результат False?

```
In [x]: Polynomial((-7, 2, 0, 3)).deriv(3) == 18
Out[x]: False
```

B6.4.2. Найти и классифицировать критические точки многочлена

$$f(x) = (x^2 + x - 11)^2 + (x^2 + x - 7)^2.$$

Задачи

36.4.1. Распространение светящейся области (пламени), образовавшейся при (ядерном) взрыве, можно проанализировать на основе начальной энергии E, освобожденной при взрыве ядерного оружия. Британский физик Джеффри Тейлор (Geoffrey Taylor) использовал пространственный анализ для демонстрации того, что радиус этой сферы $R(t)$ должен быть связан с начальной энергией E, плотностью воздуха ρ_{air} и временем t следующим образом:

$$R(t) = CE^{1/3}\rho_{air}^{-1/3}t^{2/3},$$

где с помощью задач о моделировании ударной волны Тейлор вывел приблизительную оценку безразмерной константы $C \approx 1$. Используя данные, полученные в результате неклассифицированных по времени изображений первого ядерного взрыва в Нью-Мексико, Тейлор подтвердил этот закон и вычислил приближенное значение (в то время неизвестное) начальной энергии E. Использовать

логарифмический (по обеим осям) график для подгонки данных из табл. 6.8[86] для моделирования и подтверждения зависимости от времени радиуса R. Принять плотность воздуха $\rho_{air} = 1.25$ кг/м^3, определить начальную энергию E и выразить ее значение в джоулях (Дж) и в «килотоннах, т. е. в тротиловом (TNT) эквиваленте», где значение энергии, освобождаемой при взрыве 1 т тротила (тринитротолуола – TNT), приблизительно определено равным 4.184×10^9 Дж.

Таблица 6.7. Радиус светящейся области (пламени), образовавшейся при ядерном испытании «Trinity», как функция от времени

t, мс	*R*, м	*t*, мс	*R*, м	*t*, мс	*R*, м
0.1	11.1	1.36	42.8	4.34	65.6
0.24	19.9	1.50	44.4	4.61	67.3
0.38	25.4	1.65	46.0	15.0	106.5
0.52	28.8	1.79	46.9	25.0	130.0
0.66	31.9	1.93	48.7	34.0	145.0
0.80	34.2	3.26	59.0	53.0	175.0
0.94	36.3	3.53	61.1	62.0	185.0
1.08	38.9	3.80	62.9		
1.22	41.0	4.07	64.3		

Примечание: эти данные можно загрузить отсюда: https://scipython.com/ex/bfg.

36.4.2. Найти среднее значение и дисперсию для обеих переменных *x* и *y*, коэффициент корреляции и уравнение прямой линейной регрессии для каждого из четырех наборов данных, приведенных в табл. 6.9. Прокомментировать эти значения в информационной области графика, отображающего данные.

Таблица 6.9. Четыре выборки данных для анализа среднего значения, дисперсии и корреляции

x_1	y_1	x_2	y_2	x_3	y_3	x_4	y_4
10.0	8.04	10.0	9.14	10.0	7.46	8.0	6.58
8.0	6.95	8.0	8.14	8.0	6.77	8.0	5.76
13.0	7.58	13.0	8.74	13.0	12.74	8.0	7.71
9.0	8.81	9.0	8.77	9.0	7.11	8.0	8.84
11.0	8.33	11.0	9.26	11.0	7.81	8.0	8.47
14.0	9.96	14.0	8.10	14.0	8.84	8.0	7.04
6.0	7.24	6.0	6.13	6.0	6.08	8.0	5.25
4.0	4.26	4.0	3.10	4.0	5.39	19.0	12.50
12.0	10.84	12.0	9.13	12.0	8.15	8.0	5.56
7.0	4.82	7.0	7.26	7.0	6.42	8.0	7.91
5.0	5.68	5.0	4.74	5.0	5.73	8.0	6.89

Примечание: эти данные можно загрузить отсюда: https://scipython.com/ex/bft.

[86] *G. I. Taylor* (1950). Proc. Roy. Soc. London A201, 159.

36.3.4. Уравнение Ван дер Ваальса, описывающее состояние (модель) газа, можно записать в виде следующей формулы как зависимость давления p газа от его молярного объема V и температуры T:

$$p = RT / (V - b) - a / V^2,$$

где a и b – специальные молекулярные константы, а $R = 8.314$ Дж / К·моль – универсальная газовая константа. Формулу легко преобразовать для вычисления температуры по заданному давлению и объему, но ее форма, представляющая молярный объем в отношении к давлению и температуре, является кубическим уравнением:

$$pV^3 - (pb + RT)V^2 + aV - ab = 0.$$

Все три корня этого уравнения ниже критической точки (T_c, p_c) являются действительными: наибольший и наименьший соответствуют молярному объему газообразной фазы и жидкой фазы соответственно. Выше критической точки, где не существует жидкая фаза, только один корень является действительным и соответствует молярному объему газа (в этой области его также называют сверхкритической жидкостью, или сверхкритической средой). Критическая точка определяется по условию $(\partial p/\partial V)_T = (\partial^2 p/\partial V^2)_T = 0$, и для идеального газа Ван дер Ваальса выводятся формулы:

$$T_c = 8a/27Rb, \;\; p_c = a/27b^2.$$

Для аммиака константы Ван дер Ваальса $a = 4.225$ л^2·бар/моль^2 и $b = 0.03707$ л/моль.

а) Найти критическую точку для аммиака, затем определить молярный объем при комнатной температуре и давлении (298 К, 1 атм) и при следующих условиях (500 К, 12 МПа).

б) Изотерма – это множество точек (p, V) при постоянной температуре, соответствующее уравнению состояния газа. Построить изотерму (p в зависимости от V) для аммиака при температуре 350 К, используя уравнение Ван дер Ваальса, и сравнить ее с изотермой при температуре 350 К для идеального газа, уравнение состояния которого имеет вид $p = RT/V$.

36.4.4. Первые ступени ракеты Saturn V, которая выводила на орбиту Apollo 11, создавали ускорение, увеличивающееся во время выполнения этой операции (в основном за счет уменьшения массы при выгорании топлива). Это ускорение можно смоделировать (в м/с^2) как функцию от времени после запуска t в c – квадратным многочленом:

$$a(t) = 2.198 + (2.842\times10^{-2})t + (1.061\times10^{-3})t^2.$$

Определить расстояние, пройденное ракетой в конце фазы работы среднего двигателя первой ступени, т. е. через 2 мин 15.2 с после запуска.

(Более трудная задача.) С учетом константы «вертикальный температурный градиент» (lapse rate) $\Gamma = -dT/dz = 8$ К/км и температуры у земной поверхности 302 К определить, за какое время и на какой высоте z ракета достигла скорости, при которой число Маха равно 1. Во время соответствующей фазы пуска принять средний угол тангажа равным 12° и предположить, что скорость звука можно вычислять как функцию от абсолютной температуры по формуле:

$$c = \sqrt{(\gamma RT/M)},$$

где константы $\gamma = 1.4$ и $R = 8.314$ Дж/К·моль, а средняя молярная масса атмосферы $M = 0.0288$ кг/моль.

6.5 Линейная алгебра

6.5.1 Основные операции с матрицами

Операции с матрицами могут выполняться с использованием обычных двумерных массивов NumPy, в том числе скалярное умножение, матричное (векторное) умножение, поэлементное умножение и транспонирование:

```
In [x]: A = np.array([[0, 0.5], [-1, 2]])
In [x]: A
Out[x]:
array([[ 0. ,  0.5],
       [-1. ,  2. ]])
In [x]: A * 5                 # Умножение на скаляр.
Out[x]:
array([[  0. ,   2.5],
       [ -5. ,  10. ]])
In [x]: B = np.array([[2, -0.5], [3, 1.5]])

In [x]: B
Out[x]:
array([[ 2. , -0.5],
       [ 3. ,  1.5]])

In [x]: A.dot(B)              # Или np.dot(A, B): произведение матриц.
Out[x]:
array([[ 1.5 ,  0.75],
       [ 4. ,   3.5 ]])

In [x]: A * B                 # Поэлементное умножение.
Out[x]:
array([[ 0.  , -0.25],
       [-3.  ,  3. ]])
In [x]: A.transpose()         # Или просто A.T.
Out[x]:
array([[ 0. , -1. ],
       [ 0.5,  2. ]])
```

Обратите внимание: операция транспонирования возвращает представление исходной матрицы.

Матрица тождественного преобразования (единичная матрица) возвращается при передаче двух измерений матрицы в метод `np.eye`:

```
In [x]: np.eye(3, 3)
Out[x]:
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

Произведение матриц

В библиотеке NumPy содержатся методы для выполнения векторного и матричного произведений. Например:

```
In [x]: a = np.array([1, 2, 3])
In [x]: b = np.array([0, 1, 2])
In [x]: np.inner(a, b)    # Внутреннее скалярное произведение; здесь то же самое, что a.dot(b).
Out[x]: 8

In [x]: np.outer(a, b)   # Внешнее векторное произведение.
Out[x]:
array([[0, 1, 2],
       [0, 2, 4],
       [0, 3, 6]])
```

Для возведения матрицы в (целочисленную) степень требуется метод из модуля `np.linalg`:

```
In [x]: A = np.array([[0, 0.5], [-1, 2]])
In [x]: np.linalg.matrix_power(A, 3)              # То же самое, что A @ A @ A.
Out[x]:
array([[-1.  ,  1.75],
       [-3.5 ,  6.  ]])
```

Обратите внимание: оператор ** выполняет поэлементное возведение в степень:

```
In [x]: A ** 3                          # То же самое, что A * A * A.
Out[x]:
array([[ 0.   ,  0.125],
       [-1.   ,  8.   ]])
```

Пример П6.13. Один из способов создания матрицы вращения на двумерной плоскости:

$$R = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix},$$

при котором точки поворачиваются в плоскости *xy* против часовой стрелки на угол $\theta = 30°$ относительно начала координат:

```
In [x]: theta = np.radians(30)
In [x]: c, s = np.cos(theta), np.sin(theta)
In [x]: R = np.array([[c, -s], [s, c]])
In [x]: print(R)
[[ 0.8660254 -0.5      ]
 [ 0.5        0.8660254]]
```

Например, компоненты единичного вектора (орта) оси *x* после выполнения этого поворота вычисляются с помощью скалярного произведения:

```
In [x]: v = np.array([1, 0])
In [x]: R @ v
Out[x]: array([0.8660254, 0.5      ])
```

Другие свойства матриц

Норму матрицы или вектора возвращает функция `np.linalg.norm`. Можно вычислить несколько различных норм (см. документацию), но используемой по умолчанию является норма Фробениуса для двумерных массивов:

$$\|A\| = \left(\sum_{i,j} |a_{i,j}|^2 \right)^{1/2}$$

и норма Евклида для одномерных массивов:

$$\|a\| = \left(\sum_{i} |z_i|^2 \right)^{1/2} = \sqrt{|z_0|^2 + |z_1|^2 + \ldots + |z_{n-1}|^2}.$$

Таким образом:

```
In [x]: np.linalg.norm(A)
Out[x]: 2.2912878474779199

In [x]: c = np.array([1, 2j, 1 - 1j])
In [x]: np.linalg.norm(c)
Out[x]: 2.6457513110645907          # sqrt(1 + 4 + 2).
```

Функция `np.linalg.det` возвращает определитель матрицы, а обычная функция NumPy `np.trace` возвращает след матрицы (сумму ее диагональных элементов):

```
In [x]: np.linalg.det(A)
Out[x]: 0.5

In [x]: np.trace(A)
Out[x]: 2.0
```

Ранг матрицы определяется с помощью метода `np.linalg.matrix_rank`:

```
In [x]: np.linalg.matrix_rank(A)       # Матрица A имеет полный ранг.
Out[x]: 2
```

```
In [x]: D = np.array([[1,1],[2,2]])    # Матрица с неполным рангом (с дефицитом ранга).

In [x]: np.linalg.matrix_rank(D)
Out[x]: 1
```

Для поиска обращенной матрицы используется метод `np.linalg.inv`. Если попытка определения обращенной матрицы заканчивается неудачно, то генерируется исключение `LinAlgError`:

```
In [x]: np.linalg.inv(A)
Out[x]:
array([[ 4., -1.],
       [ 2.,  0.]])

In [x]: np.linalg.inv(D)
...
LinAlgError: Singular matrix
```

Пример П6.14. Значения силы переменного тока в замкнутых контурах, обозначенные как I_1, I_2, I_3 в электрической схеме на рис. 6.9, можно проанализировать с использованием анализа цепей методом контурных токов.

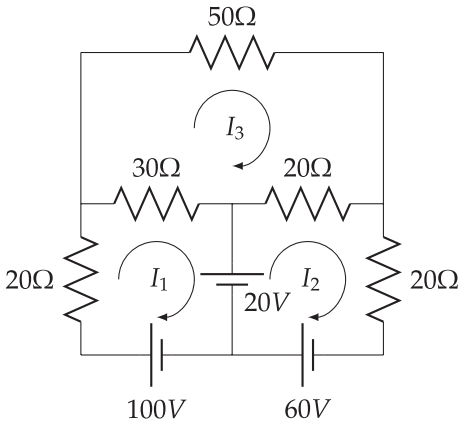


Рис. 6.9

Для каждого замкнутого контура можно применить закон Кирхгофа ($\sum_k V_k = 0$) в совокупности с законом Ома ($V = IR$) для получения трех совместных уравнений:

$$\begin{aligned} 50I_1 - 30I_3 &= 80, \\ 40I_2 - 20I_3 &= 80, \\ -30I_1 - 20I_2 + 100I_3 &= 0. \end{aligned}$$

Эту систему уравнений можно записать в матричной форме как $\boldsymbol{RI} = \boldsymbol{V}$:

$$\begin{pmatrix} 50 & 0 & -30 \\ 0 & 40 & -20 \\ -30 & -20 & 100 \end{pmatrix}\begin{pmatrix} I_1 \\ I_2 \\ I_3 \end{pmatrix} = \begin{pmatrix} 80 \\ 80 \\ 0 \end{pmatrix}.$$

Здесь можно было бы воспользоваться методом, обеспечивающим стабильность вычислений, `np.linalg.solve` (см. раздел 6.5.3), для определения значений силы тока в контурах ***I***, но в данной системе с четко определенным поведением[87] мы найдем эти значения с помощью левостороннего умножения на обращенную матрицу $\mathbf{R}^{-1}$:

$$\mathbf{R}^{-1}\mathbf{RI} = \mathbf{I} = \mathbf{R}^{-1}\boldsymbol{V}.$$

Используем методы массивов NumPy:

```
In [x]: R = np.array([[50, 0, -30], [0, 40, -20], [-30, -20, 100]])
In [x]: V = np.array([80, 80, 0])
In [x]: I = np.linalg.inv(R) @ V
In [x]: print(I)
[[ 2.33333333]
 [ 2.61111111]
 [ 1.22222222]]
```

Таким образом, $I_1 = 2.33$ А, $I_2 = 2.61$ А, $I_3 = 1.22$ А.

Пример П6.15. Здесь определена матрица **B**, с которой можно работать, как показано ниже:

$$B = \begin{pmatrix} 1 & 3-j \\ 3j & -1+j \end{pmatrix},\ B^T = \begin{pmatrix} 1 & 3j \\ 3-j & -1+j \end{pmatrix}$$

$$B^\dagger = \begin{pmatrix} 1 & -3j \\ 3+j & -1-j \end{pmatrix},\ B^{-1} = \begin{pmatrix} -\frac{1}{20}-\frac{3}{20}j & \frac{1}{20}-\frac{7}{20}j \\ \frac{3}{20}+\frac{3}{20}j & -\frac{1}{20}+\frac{1}{20}j \end{pmatrix}.$$

```
In [x]: B = np.array([[1, 3 - 1j], [3j, -1 + 1j]])
In [x]: print(B)
[[ 1.+0.j  3.-1.j]
 [ 0.+3.j -1.+1.j]]

In [x]: print(B.T)                  # Транспонированная матрица.
[[ 1.+0.j  0.+3.j]
 [ 3.-1.j -1.+1.j]]
```

---

[87] В общем случае обращение матрицы может быть задачей с плохо определенными условиями, но в рассматриваемом здесь случае матрица точно обращается без затруднений. См. в разделе 10.2.2 более подробную информацию об определении условий задачи.

```
In [x]: print(B.conj().T)          # Эрмитово-сопряженная матрица.
[[ 1.-0.j  0.-3.j]
 [ 3.+1.j -1.-1.j]]

In [x]: print(np.linalg.inv(B))    # Обращенная матрица.
[[-0.05-0.15j  0.05-0.35j]
 [ 0.30+0.15j -0.05+0.1j ]]
```

В пакете NumPy также доступны некоторые другие часто применяемые операции с матрицами, в том числе определение следа, определителя, собственных значений и (правых) собственных векторов матриц:

```
In [x]: print(np.trace(B))
1j
In [x]: print(np.linalg.det(B))
(-4-8j)
In [x]: eigenvalues, eigenvectors = np.linalg.eig(B)
In [x]: print(eigenvalues, eigenvectors , sep='\n\n')
[ 2.50851535+2.09456868j -2.50851535-1.09456868j]

[[ 0.77468569+0.j         -0.52924821+0.38116633j]
 [ 0.18832434+0.60365224j  0.75802940+0.j        ]]
```

6.5.2 Собственные значения и собственные векторы

Для вычисления собственных значений и (правых) собственных векторов обобщенного квадратного массива формы (n, n) используется метод `np.linalg.eig`, который возвращает собственные значения `w` как массив формы `(n,)` и нормализованные собственные векторы `v` как массив комплексных чисел формы `(n, n)`. Собственные значения не возвращаются в каком-либо определенном порядке, но собственное значение `w[i]` соответствует собственному вектору `v[:, i]`. Следует отметить, что собственные векторы организованы по столбцам. Если при вычислении собственного значения по каким-либо причинам не обеспечивается сходимость, то генерируется исключение `LinAlgError`.

```
In [x]: vals, vecs = np.linalg.eig(A)
In [x]: vals
Out[x]: array([ 0.29289322,  1.70710678])

In [x]: np.isclose(np.sum(vals), A.trace())                    ❶
Out[x]: True

In [x]: vecs
Out[x]:
array([[-0.86285621, -0.28108464],
       [-0.50544947, -0.95968298]])
```

❶ Проверка: сумма собственных значений должна быть равна следу матрицы.

Если матрица эрмитова или действительно симметричная, то можно воспользоваться методом `np.linalg.eigh`. Этот метод принимает дополнительный аргумент `UPL0`, значением которого может быть `'L'` или `'U'` в зависимо-

сти от того, используется нижняя или верхняя треугольная часть матрицы. По умолчанию задано значение `'L'`.

Еще два метода `np.linalg.eigvals` и `np.linalg.eigvalsh` возвращают только собственные значения (но не собственные векторы) обобщенной и эрмитовой матриц соответственно.

С версии NumPy 1.8 эти и большинство других методов модуля `linalg` следуют обычным правилам бродкастинга, поэтому можно одновременно работать с несколькими матрицами: предполагается, что каждая матрица должна сохраняться в последних двух измерениях. Например, можно работать с массивом формы `(3, 2, 2)`, представляющим три матрицы Паули 2×2:

$$\sigma_x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \sigma_y = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \sigma_z = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}.$$

```
In [x]: pauli_matrices = np.array((
                ((0, 1), (1, 0)),        # Матрица Паули sigma_x.
                ((0, -1j), (1j, 0)),     # Матрица Паули sigma_y.
                ((1, 0), (0, -1))        # Матрица Паули sigma_z.
               ))
In [x]: np.linalg.eigh(pauli_matrices)
Out[x]:
(array([[-1.,  1.],
        [-1.,  1.],
        [-1.,  1.]]),
 array([[[ -0.70710678+0.j       ,  0.70710678+0.j         ],
         [ 0.70710678+0.j        ,  0.70710678+0.j         ]],

        [[-0.70710678-0.j        , -0.70710678+0.j         ],
         [ 0.00000000+0.70710678j,  0.00000000 -0.70710678j]],

        [[ 0.00000000+0.j        ,  1.00000000+0.j         ],
         [ 1.00000000+0.j        ,  0.00000000+0.j         ]]]))
```

Пример П6.16. Линейное преобразование в двух измерениях можно визуализировать, используя его воздействие на единичный квадрат, определяемый двумя ортогональными базисными векторами (ортами) $\hat{\imath}$ и $\hat{\jmath}$. В общем случае это можно представить с помощью матрицы 2×2 **T**, которая работает с вектором **v** для его отображения из векторного пространства, охватываемого одним базисом, в другое векторное пространство, охватываемое другим базисом: **v'** = **Tv**. Собственные векторы при таком преобразовании могут масштабироваться, но не изменяют ориентацию, как показано в коде в листинге 6.10 для матрицы преобразования:

$$T = \begin{pmatrix} \frac{3}{2} & \frac{1}{2} \\ \frac{1}{2} & \frac{3}{2} \end{pmatrix}.$$

Результат преобразования для множества точек в декартовой плоскости также визуализируется на итоговом графике (см. рис. 6.10).

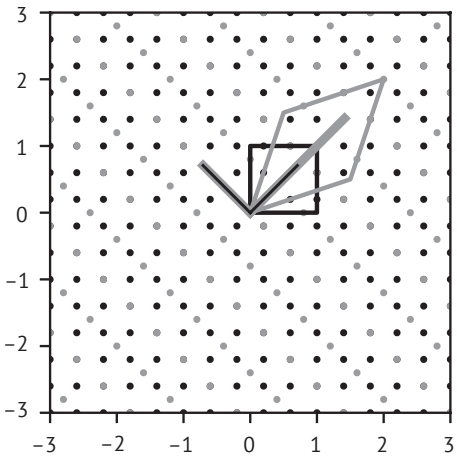


Рис. 6.10. Результат линейного преобразования, выполненного с использованием матрицы T в обычной декартовой системе координат: единичный квадрат (показанный черным цветом) растягивается вдоль одной диагонали и сжимается вдоль другой диагонали. Здесь также показаны промасштабированные собственные векторы

Листинг 6.10. Линейные преобразования по двум измерениям

```
import numpy as np
import matplotlib.pyplot as plt

# Установка области декартовой системы координат по заданным точкам.
XMIN, XMAX, YMIN, YMAX = -3, 3, -3, 3
N = 16
xgrid = np.linspace(XMIN, XMAX, N)
ygrid = np.linspace(YMIN, YMAX, N)
grid = np.array(np.meshgrid(xgrid, ygrid)).reshape(2, N**2)                    ❶

# Пока не преобразованные единичные базисные векторы i и j.
basis = np.array([[1,0], [0,1]])

def plot_quadrilateral(basis, color='k'):
    """Plot the quadrilateral defined by the two basis vectors."""
    # """Построение четырехугольника, определяемого двумя базисными векторами."""
    ix, iy = basis[0]
    jx, jy = basis[1]
    plt.plot([0, ix, ix+jx, jx, 0], [0, iy, iy+jy, jy, 0], color)

def plot_vector(v, color='k', lw=1):
    """Plot vector v as a line with a specified color and linewidth."""
    # """Построение вектора v как прямой с заданным цветом и толщиной линии."""
    plt.plot([0, v[0]], [0, v[1]], c=color , lw=lw)

def plot_points(grid, color='k'):
    """Plot the grid points in a specified color."""
    # """Построение точек сетки с заданным цветом."""
    plt.scatter(*grid , c=color, s=2, alpha=0.5)
```

```
def apply_transformation(basis, T):
    """Return the transformed basis after applying transformation T."""
    # """Возвращает преобразованный базис после применения преобразования T."""
    return (T @ basis.T).T

# Непреобразованная точечная сетка и единичный квадрат.
plot_points(grid)
plot_quadrilateral(basis)

# Применение матрицы преобразования S к сформированному изображению.
S = np.array(((1.5, 0.5),(0.5, 1.5)))
tbasis = apply_transformation(basis, S)
plot_quadrilateral(tbasis, 'r')
tgrid = S @ grid                                                    ❷
plot_points(tgrid, 'r')

# Определение собственных значений и собственных векторов S, и...
vals, vecs = np.linalg.eig(S)
print(vals, vecs)
if all(np.isreal(vals)):
    # ...если все они действительные, то изобразить их на схеме.
    v1, v2 = vals
    e1, e2 = vecs.T
    plot_vector(v1*e1, 'r', 3)
    plot_vector(v2*e2, 'r', 3)
    plot_vector(e1, 'k')
    plot_vector(e2, 'k')

# Убедиться, что график соблюдает соотношение 1:1 (т. е. квадраты выглядят как квадраты),
# и установить границы изображения.
plt.axis('square')
plt.xlim(XMIN, XMAX)
plt.ylim(YMIN, YMAX)

plt.show()
```

❶ Необходимо изменить форму сетки из $N \times N$ точек, преобразовав ее в массив из $2 \times N^2$ координат...

❷ ...которые могут быть преобразованы одной строкой кода с использованием векторизованной операции `S @ grid`.

6.5.3 Решение уравнений

Линейные скалярные уравнения

Библиотека NumPy предоставляет эффективный и обеспечивающий стабильность вычислений метод решения систем линейных скалярных уравнений. Систему уравнений

$$\begin{aligned} m_{11}x_1 + m_{12}x_2 + \ldots + m_{1n}x_1 &= b_1 \\ m_{21}x_1 + m_{22}x_2 + \ldots + m_{2n}x_2 &= b_2 \\ &\cdots \\ m_{n1}x_1 + m_{n2}x_2 + \ldots + m_{nn}x_n &= b_n \end{aligned}$$

можно записать как матричное уравнение **Mx** = **b**:

$$\begin{pmatrix} m_{11} & m_{12} & \cdots & m_{n1} \\ m_{21} & m_{22} & \cdots & m_{21} \\ \vdots & & \ddots & \vdots \\ m_{n1} & m_{n1} & \cdots & m_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}.$$

Решение этой системы уравнений (вектор **x**) возвращает метод `np.linalg.solve`. Например, три совместимых уравнения

$$\begin{gathered} 3x - 2y = 8, \\ -2x + y - 3z = -20, \\ 4x + 6y + z = 7 \end{gathered}$$

можно представить как матричное уравнение **Mx** = **b**:

$$\begin{pmatrix} 3 & -2 & 0 \\ -2 & 1 & -3 \\ 4 & 6 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 8 \\ -20 \\ 7 \end{pmatrix}$$

и решить, передавая аргументы, соответствующие матрице **M** и вектору **b**, в метод `np.linalg.solve`:

```
In [x]: M = np.array([[3, -2, 0], [-2, 1, -3], [4, 6, 1]])
In [x]: b = np.array([8, -20, 7])
In [x]: np.linalg.solve(M, b)
Out[x]: array([ 2., -1.,  5.])
```

Таким образом, решение этой системы уравнений: $x = 2$, $y = -1$, $z = 5$.

Если не существует единственного решения (для неквадратной или сингулярной матрицы **M**), то генерируется исключение `LinAlgError`.

Решение систем линейных уравнений методом наименьших квадратов (методом «наилучшего приближения»)

Когда система уравнений **Mx** = **b** не имеет единственного решения, можно рассмотреть способ решения методом наименьших квадратов, который минимизирует норму L^2 $||b - Mx||^2$ (сумму разностей, возведенную в квадрат), используя для этого метод `np.linalg.lstsq`. Это тип задачи, описываемой как избыточно определенная (или переопределенная) (больше точек данных, чем две неизвестные величины *m* и *c*). После передачи `M` и `b` метод `np.linalg.lstsq` возвращает решение в виде массива `x`, сумму квадратов разностей, ранг матрицы `M` и сингулярные значения `M`.

Ранг матрицы `M` определяется по количеству ее сингулярных значений: по умолчанию сингулярное значение считается равным нулю, если оно меньше дополнительного необязательного параметра `rcond`, кратного наибольшему сингулярному значению `M`. Если для `rcond` явно установлено значение `None`, оно будет принято равным машинной точности, кратной наибольшему изме-

рению `M`. До версии 1.14 NumPy значение по умолчанию `rcond` принималось равным самой машинной точности (для использования этого значения по умолчанию необходимо задать `rcond = -1`). На момент написания данной книги если для `rcond` не определено никакое значение, то выводится предупреждающее сообщение `FutureWarning`.

Обычный вариант использования этого метода – поиск «прямой наилучшего соответствия» $y = mx + c$ для некоторых данных, которые предполагаются линейно связанными, как в примере П6.17.

Пример П6.17. Закон Бугера–Ламберта–Бера связывает концентрацию c вещества в образце раствора с интенсивностью света, проходящего через этот образец I_t с заданной длиной пути l при известной длине волны λ:

$$I_t = I_0 e^{-\alpha c l},$$

где I_0 – интенсивность света на входе в вещество, α – коэффициент поглощения при длине волны λ.

После проведения ряда измерений, позволяющих определить часть света, которая прошла сквозь раствор, I_t/I_0, коэффициент поглощения α можно вычислить методом наименьших квадратов посредством подгонки к прямой линии:

$$y = \ln(I_t/I_0) = -\alpha c l.$$

Несмотря на то что эта прямая проходит через начало координат ($y = 0$ при $c = 0$), мы будем выполнять подгонку для более общего линейного отношения:

$$y = mc + k,$$

где $m = -\alpha l$ с проверкой k на приближение к нулю.

При рассмотрении образца раствора с длиной пройденного пути 0.8 см при измерениях были получены данные, приведенные в табл. 6.10: отношение I_t/I_0 при пяти различных концентрациях:

Таблица 6.10

c, моль/л	I_t/I_0
0.4	0.886
0.6	0.833
0.8	0.784
1.0	0.738
1.2	0.694

Форма матриц, представляющих систему уравнений для решения методом наименьших квадратов:

$$\begin{pmatrix} c_1 & 1 \\ c_2 & 1 \\ c_3 & 1 \\ c_4 & 1 \\ c_5 & 1 \end{pmatrix} \begin{pmatrix} m \\ k \end{pmatrix} = \begin{pmatrix} T_1 \\ T_2 \\ T_3 \\ T_4 \\ T_5 \end{pmatrix},$$

где $T = \ln(I_t/I_0)$. Код в листинге 6.11 определяет m, следовательно, и коэффициент поглощения α с использованием метода `np.linalg.lstsq`.

Листинг 6.11. Линейная подгонка методом наименьших квадратов для данных, соответствующих закону Бугера–Ламберта–Бера

```
# eg6-beer -lambert -lstsq.py
import numpy as np
import matplotlib.pyplot as plt
# Длина пути, см.
path = 0.8
# Данные: концентрации (M) и It/I0.
c = np.array([0.4, 0.6, 0.8, 1.0, 1.2])
It_over_I0 = np.array([0.891, 0.841, 0.783, 0.744, 0.692])

n = len(c)
A = np.vstack((c, np.ones(n))).T
T = np.log(It_over_I0)

x, resid, _, _ = np.linalg.lstsq(A, T, rcond=None)                    ❶
m, k = x
alpha = - m / path
print('alpha = {:.3f} M-1.cm-1'.format(alpha))
print('k =', k)
print('rms residual = ', np.sqrt(resid[0]))

plt.plot(c, T, 'o')
plt.plot(c, m*c + k)
plt.xlabel('$c\;/\mathrm{M}$')
plt.ylabel('$\ln(I_\mathrm{t}/I_0)$')
plt.show()
```

❶ Здесь _ – имя фиктивной (внутренней) переменной, применяемое по соглашению для объекта, который не нужно сохранять и использовать в дальнейшем.

При выполнении выводится значение наилучшей подгонки коэффициента $\alpha = 0.393$ M^{-1} $см^{-1}$ и значение k, сравнимое с ошибкой эксперимента:

```
alpha = 0.393 M-1.cm-1
k = 0.0118109033334
rms residual = 0.0096843591966
```

На рис. 6.11 показаны данные и прямая подгонки.

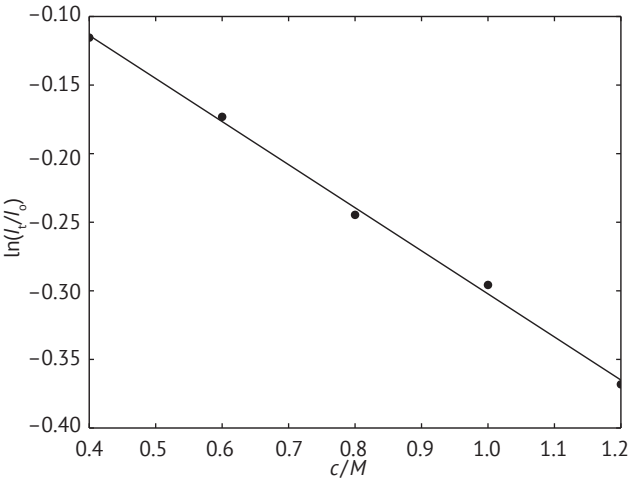


Рис. 6.11. Прямая наилучшей подгонки по методу наименьших квадратов для объединения данных как функции от концентрации

6.5.4 Упражнения

Вопросы

B6.5.1. Показать, что три матрицы Паули, приведенные в разделе 6.5.2, являются унитарными (единичными). То есть что $\sigma_p^{\dagger}\sigma_p = I_2$ при $p = x, y, z$, где I_2 – матрица тождественности, а символ † обозначает эрмитову сопряженность (сопряженное транспонирование).

B6.5.2. Тикер-таймер, часто используемый в школьных физических экспериментах, представляет собой устройство, которое наносит точечные метки на полоску бумажной ленты через равномерные интервалы времени, тогда как лента перемещается через тикер-таймер с некоторой (возможно, переменной) скоростью. Приведенные ниже данные соответствуют позициям (в см) меток на ленте, протягиваемой сквозь тикер-таймер падающим грузом. Метки наносятся через каждые 1/10 с.

```
x = [1.3, 6.0, 20.2, 43.9, 77.0, 119.6, 171.7, 233.2, 304.2, 384.7,
     474.7, 574.1, 683.0, 801.3, 929.2, 1066.4, 1213.2, 1369.4, 1535.1,
     1710.3, 1894.9]
```

Выполнить подгонку этих данных к функции $x = x_0 + v_0 t + 1/2gt^2$ и определить приблизительное значение ускорения свободного падения g.

Задачи

36.5.1. В физике единицы измерения Планка определены так, что пять универсальных физических постоянных: c (скорость света), G (гравитационная постоянная), $\hbar$ (редуцированная постоянная Планка, или постоянная Планка–Дирака), $(4\pi\varepsilon_0)^{-1}$ (постоянная Кулона) и k_B (постоянная Больцмана) – установле-

ны равными единице. Размерности этих величин устанавливаются в единицах длины (L), массы (M), времени (T), электрического заряда (Q) и термодинамической температуры (Θ). В табл. 6.11 для этих постоянных указаны их значения в системе единиц СИ.

Таблица 6.11. Некоторые физические постоянные, их значения и единицы измерения

Обозначение	Описание	Значение	Размерности
c	Скорость света	2.99792458×10^{8} м/с	L/T
G	Гравитационная постоянная	6.67384×10^{-11} м^3/кг·с	L^3/M·T
$\hbar$	Редуцированная постоянная Планка	$1.054571726\times10^{-34}$ Дж·с	L^2·M/T
$(4\pi\varepsilon_0)^{-1}$	Постоянная Кулона	$8.9875517873681764\times10^{9}$ Н·м^2/Кл^2	L^3·M/T^2·Q^2
k_B	Постоянная Больцмана	1.3806488×10^{-23} Дж/К	L^2·M/T^2·Θ

Предлагается следующая матрица отношений между этими постоянными и их размерностями:

$$\begin{array}{c} \\ c \\ G \\ h \\ (4\pi\varepsilon_0)^{-1} \\ k_B \end{array} \begin{array}{c} \begin{array}{ccccc} L & M & T & Q & \Theta \end{array} \\ \begin{pmatrix} 1 & 0 & -1 & 0 & 0 \\ 3 & -1 & -2 & 0 & 0 \\ 2 & 1 & -1 & 0 & 0 \\ 3 & 1 & -2 & -2 & 0 \\ 2 & 1 & -2 & 0 & -1 \end{pmatrix} \end{array}$$

Используя обращение этой матрицы, определить значения в системе СИ длины, массы, времени, электрического заряда и температуры в основных единицах измерения Планка, т. е. комбинации этих физических постоянных, которые в результате дают размерности L, M, T, Q и Θ. Например, длина Планка вычисляется по формуле $l_p = \sqrt{(\hbar G/c^3)} = 1.616199\times10^{-35}$ м.

36.5.2 Матрица (симметричная), представляющая тензор моментов инерции совокупности масс m_i с координатами (x_i, y_i, z_i) относительно центра масс всей системы:

$$I = \begin{pmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{xy} & I_{yy} & I_{yz} \\ I_{xz} & I_{yz} & I_{zz} \end{pmatrix},$$

где

$$I_{xx} = \sum_i m_i(y_i^2 + z_i^2), \quad I_{yy} = \sum_i m_i(x_i^2 + z_i^2), \quad I_{zz} = \sum_i m_i(x_i^2 + y_i^2),$$
$$I_{xy} = -\sum_i m_i x_i y_i, \qquad I_{yz} = -\sum_i m_i y_i z_i, \qquad I_{xz} = -\sum_i m_i x_i z_i.$$

Существует преобразование системы координат, такое, что эта матрица является диагональной: оси подобной преобразованной системы координат называются главными осями, а элементы, расположенные на диагонали матрицы моментов инерции $I_a \leqslant I_b \leqslant I_c$, являются главными моментами инерции.

Написать программу для вычисления главных моментов инерции молекулы с учетом ее положения и масс ее атомов относительно некоторого произвольно выбранного начала координат. Программа должна сначала переопределить координаты атомов относительно центра масс, затем определить главные моменты инерции как собственные значения матрицы **I**.

Молекулы можно классифицировать по отношениям между значениями I_a, I_b, I_c, как показано ниже:

- $I_a = I_b = I_c$ – сферический волчок;
- $I_a = I_b < I_c$ – сплющенный симметричный волчок;
- $I_a < I_b = I_c$ – вытянутый симметричный волчок;
- $I_a < I_b < I_c$ – асимметричный волчок.

Определить главные моменты инерции и классифицировать молекулы NH_3, CH_4, CH_3Cl и O_3 с помощью данных, доступных здесь: https://scipython.com/ex/bfh. Также определить постоянные вращения A, B и C, связанные с моментами инерции формулой $Q = h/(8\pi^2 c I_q)$ ($Q = A, B, C$; $q = a, b, c$) и обычно измеряемые в 1/см.

36.5.3. Метод `numpy.linalg.svd` возвращает сингулярное разложение (singular value decomposition – SVD) матрицы **M** в виде массивов **U**, **Σ** и **V**, соответствующих факторизации $\mathbf{M} = \mathbf{U\Sigma V}^{\dagger}$, где символ † обозначает эрмитову сопряженность (сопряженное транспонирование).

Сингулярное разложение (SVD) и собственная декомпозиция связаны так, что левые сингулярные векторы-строки **U** являются собственными векторами $\mathbf{MM}^*$, а правые сингулярные векторы-столбцы **V** являются собственными векторами $\mathbf{M}^*\mathbf{M}$. Кроме того, диагональные элементы массива **Σ** являются квадратными корнями из ненулевых собственных значений $\mathbf{MM}^*$ и $\mathbf{M}^*\mathbf{M}$.

Показать, что это вариант (частный случай) особого случая **M**, матрицы 3×3 со случайно выбранными действительными элементами, сравнивая вывод метода `numpy.linalg.svd` с выводом метода `numpy.linalg.eig`.

Совет: сингулярные значения **M** сортируются в убывающем порядке, но собственные значения, возвращаемые методом `numpy.linalg.eig`, не располагаются в каком-либо определенном порядке. Оба метода создают нормализованные собственные векторы, но могут отличаться по знаку (не принимать во внимание возможность того, что любое из собственных значений может иметь собственное пространство с размерностью больше 1).

36.5.4. Рассмотреть матрицу-столбец

$$\mathbf{F}_n = \begin{pmatrix} p_n \\ q_n \end{pmatrix},$$

описывающую набор неотрицательных целых чисел меньше 10^n ($n \geqslant 0$), которые содержат (p_n) или не содержат (q_n) цифру 5. То есть при $n = 1$ $p_1 = 1$ и $q_1 = 9$. Найти рекурсивное отношение на основе матрицы для определения $\mathbf{F}_{n+1}$ по $\mathbf{F}_n$.

Сколько чисел меньше 10^{10} содержат цифру 5?
Для каждого $n \leqslant 10$ найти p_n и проверить, что $p_n = 10^n - 9^n$.

36.5.5. Матрицу

$$\mathbf{F}_n = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$$

можно использовать для создания ряда Фибоначчи посредством многократно повторяемого умножения: элемент F_{11}^n матрицы $\mathbf{F}^n$ – это $(n + 1)$-е число Фибоначчи (при $n = 0, 1, 2, \ldots$). Использовать представление NumPy матрицы **F** для вычисления первых 10 чисел Фибоначчи.

Можно показать, что

$$\mathbf{F}^n = \mathbf{C}\mathbf{D}^n\mathbf{C}^{-1}, \text{ где } \mathbf{D} = \mathbf{C}^{-1}\mathbf{F}\mathbf{C}$$

– диагональная матрица, связанная с **F** через преобразование подобия при взаимодействии с матрицей **C**. Использовать эту связь для вычисления 1100-го числа Фибоначчи.

36.5.6. Уравнение конического сечения в неявной форме может быть записано как многочлен второй степени

$$Q = Ax^2 + Bxy + Cy^2 + Dx + Ey + F = 0,$$

или в матричной форме с использованием вектора однородных координат

$$\mathrm{x} = \begin{pmatrix} x \\ y \\ z \end{pmatrix},$$

как $\mathbf{x}^T\mathbf{Q}\mathbf{x} = 0$, где

$$Q = \begin{pmatrix} A & B/2 & D/2 \\ B/2 & C & E/2 \\ D/2 & E/2 & F \end{pmatrix}.$$

Конические сечения можно классифицировать в соответствии с описанными ниже свойствами **Q**, при этом частичная матрица $\mathbf{Q}_{33}$ имеет вид

$$Q_{33} = \begin{pmatrix} A & B/2 \\ B/2 & C \end{pmatrix}.$$

- Если $\det\mathbf{Q} = 0$, то коническое сечение является вырожденным и представлено в одной из следующих форм:
  - если $\det\mathbf{Q}_{33} < 0$, то уравнение представляет две пересекающиеся прямые;
  - если $\det\mathbf{Q}_{33} = 0$, то уравнение представляет две параллельные прямые;
  - если $\det\mathbf{Q}_{33} > 0$, то уравнение представляет единственную точку.

- Если $\det\mathbf{Q} \neq 0$, то:
  - если $\det\mathbf{Q}_{33} < 0$, то коническим сечением является гипербола;
  - если $\det\mathbf{Q}_{33} = 0$, то коническим сечением является парабола;
  - если $\det\mathbf{Q}_{33} < 0$, то коническим сечением является эллипс, а в частном случае:
    - если $A = C$ и $B = 0$, то эллипс становится кругом.

Написать программу для классификации конических сечений, представленных шестью коэффициентами A, B, C, D, E и F.

Некоторые варианты для тестирования (коэффициенты не принимаются равными нулю):

- гипербола: $B = 1, F = -9$;
- парабола: $A = 1/2, D = 2, E = -1/2$;
- круг: $A = 1/2, C = 1/2, D = -2, E = -3, F = 2$;
- эллипс: $A = 9, C = 4, F = -36$;
- две параллельные прямые: $A = 1, F = -1$;
- единственная точка: $A = 1, C = 1$.

36.5.7. В примере П6.8 создавался текстовый файл с данными, разделенными запятыми, содержащий результаты 10 имитаций радиоактивного распада множества из 500 ядер ^{14}C. В каждом столбце имитации приведено число нераспавшихся ядер как функция от времени $N(t)$, а в первом столбце указана сетка точек времени (в годах).

Вывести средние значения по данным имитаций, доступным здесь: https://scipython.com/ex/bac, и использовать метод NumPy `np.linalg.lstsq` для выполнения линейной подгонки методом наименьших квадратов. Принять период полураспада ^{14}C $t_{1/2} = \tau \ln 2$, где

$$N(t) = N(0)e^{-t/\tau} \Rightarrow \ln[N(t)] = \ln[N(0)] - t/\tau.$$

6.6 Случайная выборка

Модуль `random` библиотеки NumPy предоставляет методы для получения случайных чисел по любому из нескольких распределений, а также удобные способы выбора случайных элементов из массива и перемешивания в случайном порядке содержимого массива.

Как и модуль стандартной библиотеки `random` (см. раздел 4.5.1), модуль `np.random` использует генератор псевдослучайных чисел (ГПСЧ) вихрь Мерсенна (Mersenne Twister). Способ начального посева ГПСЧ зависит от операционной системы, но можно изменить посев с помощью любого хешируемого объекта (например, любого неизменяемого объекта, такого как целое число), передаваемого при вызове `np.random.seed`. Например, можно воспользоваться методом `randint`, как показано ниже:

```
In [x]: np.random.seed(42)
In [x]: np.random.randint(1, 10, 10)    # 10 случайных целых чисел в интервале [1, 10).
array([7, 4, 8, 5, 7, 3, 7, 8, 5, 4])
In [x]: np.random.randint(1, 10, 10)
array([8, 8, 3, 6, 5, 2, 8, 6, 2, 5])
```

```
In [x]: np.random.randint(1, 10, 10)
array([1, 6, 9, 1, 3, 7, 4, 9, 3, 5])
In [x]: np.random.seed(42)              # Изменение посева для ГПСЧ.
In [x]: np.random.randint(1,10, 10)
array([7, 4, 8, 5, 7, 3, 7, 8, 5, 4])   # Тот же результат, что и в первом примере.
```

6.6.1 Равномерно распределенные случайные числа

Случайные числа с плавающей точкой

Основной метод получения случайных чисел `random_sample`[88] в качестве аргумента принимает форму массива и создает массив заданной формы, заполненный числами, случайно выбранными из равномерного распределения в интервале [0,1), т. е. в интервале от 0 до 1, включая 0, но исключая 1:

```
In [x]: np.random.random_sample((3,2))
array([[ 0.92338355,  0.2978852 ],
       [ 0.75175429,  0.88110707],
       [ 0.16759816,  0.32203783]])
```

(если метод вызывается без аргументов, то возвращается одно случайное число). Если необходимы числа, выбираемые из равномерного распределения в интервале [*a*,*b*), то потребуется дополнительная работа:

```
In [x]: a, b = 10, 20
In [x]: a + (b - a) * np.random.random_sample((3, 2))
array([[ 18.07084068,  12.11591797],
       [ 14.08171741,  19.34857282],
       [ 13.06759203,  11.07003867]])
```

В равномерном распределении для каждого числа существует одинаковая вероятность его выбора, как можно наблюдать на гистограмме с большим количеством выборок (см. рис. 6.12):

```
In [x]: plt.hist(np.random.random_sample(10000), bins=100)
In [x]: plt.show()
```

Метод `np.random.rand` работает аналогично, но размерности требуемых массивов передаются в него как отдельные аргументы. Например:

```
In [x]: np.random.rand(2, 3)
Out[x]:
array([[ 0.61075227,  0.37459455,  0.95670676],
       [ 0.25276732,  0.1601836 ,  0.3746576 ]])
```

[88] Метод `np.random.random_sample` также доступен через псевдонимы `np.random.random`, `np.random.ranf` и `np.random.sample`.

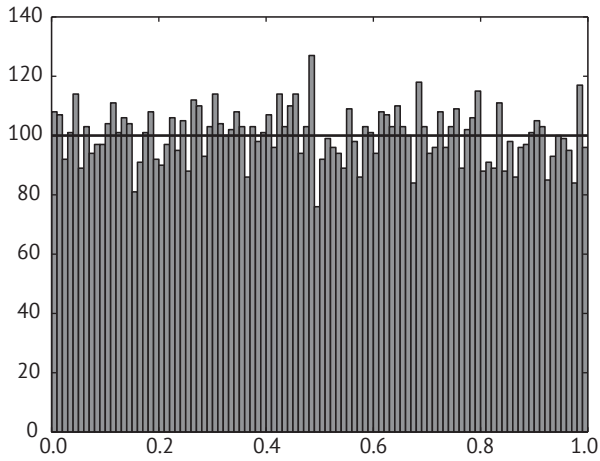


Рис. 6.12. Гистограмма по 10 000 случайных выборок из равномерного распределения на интервале [0,1), выполненных с помощью метода `np.random.random_sample`

Случайные целые числа

Выборка случайных целых чисел поддерживается несколькими методами. Метод `np.random.randint` принимает три аргумента: `low`, `high` и `size`.

- Если переданы оба аргумента `low` и `high`, то случайные числа выбираются из дискретного полуоткрытого интервала `[low, high)`[89].
- Если передан только аргумент `low` без аргумента `high`, то выборка выполняется из полуоткрытого интервала `[0, low)`.
- Аргумент `size` – это форма требуемого массива случайных целых чисел. Если этот аргумент не передан, то, как при вызове метода np.random.rand, возвращается единственное случайное целое число.

```
In [x]: np.random.randint(4)             # Случайное целое число из интервала [0, 4).
2
In [x]: np.random.randint(4, size=10)    # 10 случайных целых чисел из интервала [0, 4).
array([3, 2, 2, 2, 0, 2, 2, 1, 3, 1])
In [x]: np.random.randint(4, size=(3, 5)) # Массив случайных целых чисел из интервала [0, 4).
array([[0, 1, 1, 2, 2],
       [2, 0, 3, 3, 0],
       [0, 1, 0, 1, 1]])
In [x]: np.random.randint(1, 4, (3, 5))  # Массив случайных целых чисел из интервала [1, 4).
array([[1, 1, 1, 3, 2],
       [1, 1, 2, 1, 3],
       [1, 3, 1, 3, 1]])
```

Метод `np.random.randint` может оказаться полезным для выбора случайных элементов (с заменой) из массива с применением выбора по случайным индексам:

---

[89] Обратите внимание: это отличается от поведения метода стандартной библиотеки `random.randint(a,b)` (см. раздел 4.5.1), который выбирает с равной вероятностью числа из закрытого интервала `[a,b]`.

```
In [x]: a = np.array([6, 6, 6, 7, 7, 7, 7, 7, 7])
In [x]: a[np.random.randint(len(a), size=5)]
array([7, 7, 7, 6, 7])
```

Другой метод выборки случайных целых чисел `np.random.random_integers` имеет тот же синтаксис, но возвращает целые числа, выбранные из равномерного распределения в закрытом интервале `[low, high]` (если передан аргумент `high`) или `[0, low]` (если аргумент `high` отсутствует).

Пример П6.18. Описанные выше методы получения случайных целых чисел можно использовать для создания выборки из множества равномерно распределенных действительных чисел, хотя для этого потребуется выполнить некоторую дополнительную работу: для выбора числа из `n` равномерно распределенных действительных чисел между `a` и `b` (включительно) используется следующая инструкция:

```
In [x]: a + (b - a) * (np.random.random_integers(n) - 1) / (n - 1.)
```

Например, для выборки из множества [1/2, 3/2, 5/2, 7/2]:

```
In [x]: a, b, n = 0.5, 3.5, 4
In [x]: a + (b - a) * (np.random.random_integers(n, size=10) - 1) / (n - 1.)
array([ 1.5,  0.5,  1.5,  1.5,  3.5,  2.5,  3.5,  3.5,  3.5,  3.5])
```

Пример П6.19. В широко известном эксперименте группе добровольцев предлагается подбросить правильную («честную») монету 100 раз и записывать результат каждого броска: орел – H (head) или решка – T (tail). Вообще говоря, достаточно легко выявить участников такого эксперимента, которые подделывают результаты, записывая то, что им кажется случайной последовательностью орлов H и решек T вместо действительного подбрасывания монеты, поскольку обманщики непроизвольно стараются не включать слишком длинные серии повторяющихся результатов, которые вполне можно ожидать при случайных бросках.

Но если бы у обманщиков был доступ к интерпретатору Python, то они могли бы создать более правдоподобный набор случайных результатов:

```
In [x]: res = ['H', 'T']
In [37]: tosses = ''.join([res[i] for i in np.random.randint(2, size=100)])
In [38]: tosses
Out[38]: 'TTHHTHHTTHHHTHTTHHHTHHTHTTHHTHHTTTTHHHHHHHHTTTHTTHHHHHHHTHHHTHHHH
THTTTHTTHHHHTHTTTTHTTTHTHHTTHHHHHHH'
```

В этом виртуальном эксперименте встречается последовательность из восьми орлов подряд и две последовательности из семи орлов подряд:

```
TAILS      | i |   HEADS
--------------------------
           | 8 | *
           | 7 | **
           | 6 |
           | 5 |
        ** | 4 | **
       *** | 3 | ***
   ******* | 2 | ******
********** | 1 | ********
```

6.6.2 Случайные числа из неравномерных распределений

Полный набор случайных распределений, поддерживаемых библиотекой NumPy, описан в официальной документации (https://docs.scipy.org/doc/numpy-1.14.0/reference/routines.random.html). В этом разделе подробно рассматриваются только нормальное и биномиальное распределения, а также распределение Пуассона.

Нормальное распределение

Нормальное распределение вероятностей описывается функцией Гаусса:

$$P(x) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2} \right),$$

где μ – среднее значение, σ – стандартное отклонение. Метод библиотеки NumPy `np.random.normal` выбирает случайные элементы из нормального распределения. Среднее значение и стандартное отклонение определяются аргументами `loc` и `scale` соответственно, по умолчанию передаются значения 0 и 1. Форма возвращаемого массива определяется атрибутом `size`.

```
In [x]: np.random.normal()
-0.34599057326978105
In [x]: np.random.normal(scale=5., size=3)
array([  4.38196707,  -5.17358738,  11.93523167])
In [x]: np.random.normal(100., 8., size=(4, 2))
array([[ 107.730434  ,  101.06221195],
       [ 100.75627505,   88.79995561],
       [  88.82658615,   94.89630767],
       [ 105.91254312,   98.21190741]])
```

Также имеется возможность выбирать числа из стандартного нормального распределения (для которого μ = 0 и σ = 1) с помощью метода `np.random.randn`. Как и метод `random.rand`, он принимает в качестве аргументов измерения требуемого массива:

```
In [x]: np.random.randn(2, 2)
array([[-1.25092263,  2.6291925 ],
       [ 0.34158642,  0.40339403]])
```

Несмотря на то что метод `np.rndom.randn` не предоставляет возможности явного определения среднего значения и стандартного отклонения, можно достаточно легко изменять масштаб стандартного распределения:

```
In [x]: mu, sigma = 100., 8.
In [x]: mu + sigma * np.random.randn(4, 2)
array([[ 104.92454826,  98.84646729],
       [ 109.43568726,  92.9568489 ],
       [  90.21632016,  96.25271625],
       [ 102.65745451,  89.94890264]])
```

Пример П6.20. Нормальное распределение можно изобразить графически по выбранным данным в виде гистограммы (см. рис. 6.13):

```
In [x]: mu, sigma = 100., 8.
In [x]: samples = np.random.normal(loc=mu, scale=sigma, size=10000)
In [x]: counts, bins, patches = plt.hist(samples, bins=100, density=True)
In [x]: plt.plot(bins, 1/(sigma * np.sqrt(2 * np.pi)) *
...                np.exp( -(bins - mu)**2 / (2 * sigma**2) ), lw=2)
In [x]: plt.show()
```

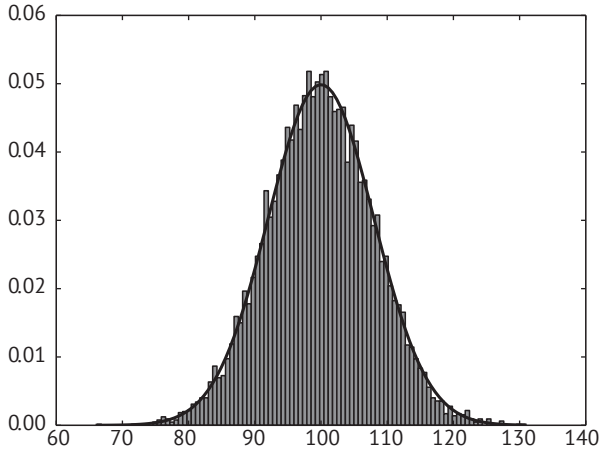


Рис. 6.13. Гистограмма по 10 000 случайных выборок из нормального распределения, выполненным с помощью метода `np.random.normal`

Биномиальное распределение

Биномиальное распределение вероятностей описывает число конкретных исходов в последовательности n испытаний Бернулли, т. е. в n независимых экспериментах, каждый из которых может дать только один из двух возможных исходов (результатов), например да/нет, успех/неудача, орел/решка и т. д. Если вероятность одного конкретного исхода (скажем, успеха) равна p, то вероятность того, что последовательность испытаний дает в точности k таких исходов, равна

$$\binom{n}{k} p^k (1-p)^{n-k}, \text{ где } \binom{n}{k} = \frac{n!}{k!(n-k)!}.$$

Например, при подбрасывании правильной («честной») монеты вероятность выпадения орла при каждом броске равна 1/2. Следовательно, вероятность выпадения в точности трех орлов при четырех бросках равна $4(1/2)(1/2)^3 = 1/4$, где множитель $\binom{4}{3} = 4$ записям для четырех возможных равнозначных исходов:

ТНННН, НТНН, ННТН, НННТ.

Для выборки из биномиального распределения, описанного параметрами n и p, используется метод np.random.binomial(n,p). И в этом случае форму массива выборки можно задать в третьем аргументе size:

```
In [x]: np.random.binomial(4, 0.5)
2
In [x]: np.random.binomial(4, 0.5, (4, 4))
array([[1, 2, 2, 4],
       [2, 1, 3, 2],
       [2, 3, 1, 1],
       [2, 4, 2, 3]])
```

Пример П6.21. Существуют два стабильных изотопа углерода ^{12}C и ^{13}C (радиоактивные ядра ^{14}C в природе существуют только в чрезвычайно малых количествах порядка триллионных частей). С учетом распространенности изотопа ^{13}C, равной $x = 0.01017$ (т. е. около 1 %), вычислим относительные количества бакминстерфуллерена C_{60} с точным количеством атомов ^{13}C: ноль, один, два, три и четыре. (Это важно при исследованиях фуллеренов методом ядерного магнитного резонанса (ЯМР), например потому, что только ядра ^{13}C обладают магнитными свойствами, поэтому их можно обнаружить методом ЯМР.)

Количество атомов ^{13}C в совокупности атомов углерода, случайно выбранных из совокупности с преобладанием природных изотопов, соответствует биномиальному распределению: вероятность того, что из n атомов m окажутся атомами ^{13}C (следовательно, $n - m$ будут атомами ^{12}C), равна

$$p_m(n) = \binom{n}{m} x^m (1-x)^{n-m}.$$

Разумеется, можно точно вычислить $p_m(60)$ по этой формуле при $0 \leqslant m \leqslant 4$, но мы также можем создать имитацию выборки с помощью метода np.random.binomial.

Листинг 6.12. Моделирование распределения атомов ^{13}C в бакминстерфуллерене C_{60}

```
# eg6-e-c13-a.py
import numpy as np

n, x = 60, 0.0107
mmax = 4
m = np.arange(mmax + 1)

# Оценка распространенности по случайной выборке из биномиального распределения.
ntrials = 10000
pbin = np.empty(mmax + 1)
for r in m:
    pbin[r] = np.sum(np.random.binomial(n, x, ntrials) == r)/ntrials                    (1)

# Вычисление и сохранение биномиальных коэффициентов nCm.
nCm = np.empty(mmax + 1)
nCm[0] = 1
for r in m[1:]:
    nCm[r] = nCm[r - 1] * (n - r + 1) / r
# "Точный" ответ (результат) из биномиального распределения.
p = nCm * x**m * (1-x)**(n-m)
```

```
print('Abundances of C60 as (13C)[m](12C)[60-m]')
print('m "Exact" Estimated')
print('-'*24)
for r in m:
    print('{:1d} {:6.4f} {:6.4f}'.format(r, p[r], pbin[r]))
```

❶ Для каждого значения r в массиве m выбирается большое число повторений испытаний (ntrial) из биномиального распределения, описанного количеством атомов $n = 60$ и вероятностью $x = 0.0107$. Сравнение этих случайно выбранных значений с заданным значением r дает массив логических значений, которые можно просуммировать (напомню, что значение True вычисляется как 1, а значение False вычисляется как 0). Далее при делении на ntrial получается оценка вероятности точного количества r атомов типа ^{13}C, а оставшиеся атомы принадлежат к типу ^{12}C.

Явно выполняемый цикл по m можно было бы исключить, создав массив формы (ntrials, mmax+1), содержащий все элементы выборки и производя суммирование по первой оси этого массива с применением операции сравнения с массивом m:

```
samples = np.random.binomial(n, x, (ntrials , mmax + 1))
pbin = np.sum(samples == m, axis=0) / ntrials
```

Значения распространенности атомов $^{13}C_m$ $^{12}C_{60-m}$, вычисляемые программой в листинге 6.12, выводятся в следующем виде.

```
Abundances of C60 as (13C)[m](12C)[60-m]
m   "Exact"    Estimated
------------------------
0   0.5244      0.5199
1   0.3403      0.3348
2   0.1086      0.1093
3   0.0227      0.0231
4   0.0035      0.0031
```

Таким образом, почти 48 % молекул C_{60} содержат как минимум одно ядро с магнитными свойствами.

Распределение Пуассона

Распределение Пуассона описывает вероятность наступления конкретного числа независимых событий в заданном интервале времени, если эти события возникают с известной средней частотой. Это распределение также используется для вычисления вероятностей наступления событий в заданных интервалах по другим типам измерений, таким как расстояние или объем. Распределение вероятностей Пуассона для количества событий k определяется следующей формулой:

$$P(k) = \lambda^k e^{-\lambda} / k!,$$

где параметр λ – это ожидаемое (среднее) количество событий, возникающих в рассматриваемом интервале[90]. В библиотеке NumPy реализация `np.random.poisson` принимает λ как первый аргумент (по умолчанию равный 1), и, как и ранее, форму требуемого массива элементов выборки можно задать во втором аргументе `size`. Например, если я получаю в среднем 2.5 сообщения электронной почты в час, то выборку количеств электронных писем, которые я буду получать каждый час в течение следующих 8 ч, можно сформировать следующим образом:

```
In [x]: np.random.poisson(2.5, 8)
array([4, 1, 3, 0, 4, 1, 3, 2])
```

Пример П6.22. Фермент эндонуклеазы EcoRI используется как ограничивающий фермент, который отсекает ДНК по последовательности нуклеиновой кислоты GAATTC. Предположим, что рассматриваемая молекула ДНК содержит 12 000 основных пар и 50 % содержимого G + C. Распределение Пуассона можно использовать для прогнозирования вероятности того, что EcoRI не сможет рассечь эту молекулу при следующих условиях.

Распознаваемая последовательность GAATTC состоит из шести основных пар нуклеотидов. Вероятность того, что любая рассматриваемая последовательность из шести элементов соответствует GAATTC, равна $1/4^6 = 1/4096$, поэтому ожидаемое количество рассекающих групп для EcoRI в этой молекуле ДНК равно $\lambda = 12000/4096 = 2.93$. Следовательно, по распределению Пуассона ожидаемая вероятность того, что данная эндонуклеаза не сможет рассечь молекулу ДНК, равна:

$$P(0) = \lambda^0 e^{-\lambda} / 0! = 0.053,$$

т. е. около 5.3 %. Стохастическая имитация вероятностей:

```
In [x]: lam = 12000 / 4**6
In [x]: N = 100000
In [x]: np.sum(np.random.poisson(lam, N) == 0) / N
Out[x]: 0.053699999999999998
```

6.6.3 Случайные выборки, перемешивания и перестановки

При наличии некоторого массива значений часто возникает необходимость случайного выбора одного или нескольких элементов (с заменой или без замены). Для этого используется метод `np.random.choice`. Он принимает один обязательный аргумент, одномерную последовательность, и возвращает случайно выбранный из этой последовательности элемент:

```
In [x]: np.random.choice([ 1,  5,  2, -5,  5,  2,  0])
2
In [x]: np.random.choice(np.arange(10))
7
```

[90] Распределение Пуассона – это предельный случай биномиального распределения при $n \to \infty$ и $p \to 0$, так что $\lambda = np$ стремится к некоторому конечному постоянному значению.

Второй аргумент `size` управляет формой возвращаемого массива случайно выбранных элементов, как и в ранее рассмотренных методах. По умолчанию элементы последовательности выбираются случайно с использованием равномерного распределения с заменой. Чтобы выбирать элементы без замены, необходимо установить аргумент `replace=False`.

```
In [x]: a = np.array([1, 2, 0, -1, 1])
In [x]: np.random.choice(a, 6)       # Шесть случайно выбранных элементов из массива a.
array([ 1, -1, 2, 1, -1, 1])
In [x]: np.random.choice(a, (2, 2), replace=False)
array([[ 2, -1],
       [ 1,  0]])
In [x]: np.random.choice(a, (3, 2), replace=False)
... <some traceback information > ...
ValueError: Cannot take a larger sample than population when 'replace=False'
            (Невозможно получить выборку большего размера при 'replace=False')
```

В последнем примере показано, что, как и ожидалось, невозможно выбрать больше элементов, чем количество элементов в исходной последовательности, если выборка выполняется без замены.

Для определения вероятности выбора каждого элемента необходимо передать как аргумент `p` последовательность той же длины, что и совокупность, из которой производится выбор. Сумма вероятностей должна быть равна 1.

```
In [x]: a = np.array([1, 2, 0, -1, 1])
In [x]: np.random.choice(a, 5, p=[0.1, 0.1, 0., 0.7, 0.1])
Out[x]: array([-1, -1, -1, -1, 1])
In [x]: np.random.choice(a, 2, False , p=[0.1, 0.1, 0., 0.8, 0.])
Out[x]: array([-1, 2])           # Выборка без замены.
```

Существуют два метода для перестановки содержимого массива: `np.random.shuffle` случайным образом изменяет порядок элементов в самом исходном массиве, тогда как `np.random.permutation` сначала создает копию массива, сохраняя исходный массив без изменений:

```
In [x]: a = np.arange(6)
In [x]: np.random.permutation(a)
array([4, 2, 5, 1, 3, 0])
In [x]: a
array([0, 1, 2, 3, 4, 5])
In [x]: np.random.shuffle(a)
In [x]: a
array([5, 4, 1, 3, 0, 2])
```

Оба метода работают только с первым измерением массива:

```
In [x]: a = np.arange(6).reshape(3, 2)
In [x]: a
array([[0, 1],
       [2, 3],
       [4, 5]])
In [x]: a.random.permutation(a)       # Перестановка строк, но не столбцов.
array([[2, 3],
       [4, 5],
       [0, 1]])
```

6.6.4 Упражнения

Вопросы

В6.6.1. Объяснить различие между кодом

```
In [x]: a = np.array([6, 6, 6, 7, 7, 7, 7, 7, 7])
In [x]: a[np.random.randint(len(a), size=5)]
array([7, 7, 7, 6, 7])       # (Например.)
```

и кодом

```
In [x]: np.random.randint(6, 8, 5)
array([6, 6, 7, 7, 7])       # (Например.)
```

В6.6.2. В примере П6.18 использовался метод `np.random.random_integers` для выборки из равномерного распределения по числам с плавающей точкой [1/2, 3/2, 5/2, 7/2]. Как можно сделать то же самое, воспользовавшись методом `np.random.randint`?

В6.6.3. Американская лотерея Mega Millions во время написания этой книги использовала выбор 5 чисел из 70 и одного из 25. Джекпот распределяется между игроками, выбравшими номера, совпадающие со случайно выбранными. Какова вероятность выиграть джекпот? Написать однострочный код Python, используя библиотеку NumPy, для выбора набора случайных чисел для игрока.

В6.6.4. Предположим, что книга из n страниц содержит m опечаток. Если опечатки независимы друг от друга, то вероятность нахождения опечатки на конкретной странице равна $p = 1/n$, а распределение вероятностей можно считать биномиальным. Написать короткую программу для управления проверкой виртуального «печатного тиража» книги с $n = 500$, $m = 400$ и определить вероятность Pr того, что отдельно взятая страница содержит две или более опечаток.

Сравнить с результатом, прогнозируемым по распределению Пуассона, с параметром средней ожидаемой частоты событий $\lambda = m/n$, $Pr = 1 - e^{-\lambda}(\lambda^0/0! + \lambda/1!)$.

Задачи

З6.6.1. Выполнить имитацию эксперимента, произведенного `ntrials` раз, в котором для каждого испытания подбрасывается `n` монет и каждый раз записывается общее количество выпавших орлов (heads).

Построить график полученных результатов в виде соответствующей гистограммы и сравнить с ожидаемым биномиальным распределением вероятностей выпадения орлов.

З6.6.2. Классическую задачу, впервые предложенную Жоржем-Луи Леклерком, графом де Бюффон (Georges-Louis Leclerc, Comte de Buffon), можно сформулировать следующим образом.

Пусть существует плоскость, расчерченная параллельными прямыми, расположенными на расстоянии d друг от друга. Какова вероятность того, что игла длиной $l \leqslant d$, брошенная случайным образом на эту плоскость, пересечет одну из прямых?

Эту задачу можно решить аналитически, получив ответ $2l/\pi d$. Необходимо показать, что это решение приближенно получается для случая $l = d$, используя метод имитации случайных событий (метод Монте-Карло), т. е. имитируя эксперимент с большим количеством случайных ориентаций бросаемой иглы.

С этой задачей связана другая задача о бросании круглой монеты радиусом a на пол, покрытый квадратными плитками со стороной d. Показать, что вероятность пересечения брошенной монеты края плитки равна $1 - (d - 2a)^2/d^2$, и подтвердить это с помощью имитации методом Монте-Карло.

36.6.3. Некоторые бактерии, например кишечная палочка (Escherichia coli), имеют спиральные жгутики, позволяющие им перемещаться к аттрактантам, таким как питательные вещества, – этот процесс известен как хемотаксис. Когда жгутики вращаются против часовой стрелки, бактерия двигается вперед. При вращении жгутиков по часовой стрелке бактерия двигается беспорядочно, изменяя ориентацию. Сочетание таких перемещений позволяет бактерии выполнять несимметричное случайное блуждание: если бактерия чувствует, что перемещается по градиенту концентрации к аттрактанту, то жгутики будут чаще вращаться против часовой стрелки, чем по часовой, чтобы продолжать движение в этом направлении. Но если движение происходит от аттрактанта, то более вероятно вращение жгутиков по часовой стрелке, т. е. беспорядочные перемещения со случайными изменениями ориентации в поисках «румба», указывающего направление к аттрактанту.

Хемотаксис кишечной палочки можно смоделировать (весьма) упрощенно, рассматривая перемещения бактерии в двумерном «мире» с существующим аттрактантом с постоянным градиентом концентрации, направленным из некоторой локации. В каждой последовательности интервалов времени модель бактерии определяет, перемещается ли она в направлении увеличения или уменьшения этого градиента и следует ли продолжать движение или случайным образом искать нужное направление в соответствии с некоторой парой вероятностей.

Написать программу реализации этой упрощенной модели хемотаксиса для мира, состоящего из единичных квадратов, с аттрактантом, расположенным в центре этого мира. Изобразить локации 10 моделей бактерий, начальные позиции которых равномерно распределены по единичной окружности с центром в локации аттрактанта.

36.6.4. Одним из способов имитации излучин реки (меандров) является усреднение большого количества случайных блужданий[91]. Используется система координат (x, y), начальная точка случайных блужданий $A = (0,0)$, целевая конечная точка $B = (b,0)$. Выбрав начальный угол ориентации движения ϕ_0 относительно направления AB, необходимо на каждом шаге изменять этот угол на случайную величину, выбираемую из нормального распределения со средним значением $\mu = 0$ и стандартным отклонением σ, и выполнять перемещение на единичное расстояние в этом направлении. Отбрасывать все варианты блуж-

[91] *B. Hayes*. American Scientist 94, 490 (2006); *H. von Schelling*. General Electric Report No. 64GL92.

дания, которые после n шагов не завершаются в пределах одной единицы расстояния от точки B (такие варианты будут составлять большинство).

Написать программу для поиска усредненного пути, соответствующего описанным выше ограничениям для $b = 10$, используя значения $\phi_0 = 110°$, $\sigma = 17°$, $n = 40$ и 10^6 испытаний методом случайного блуждания. Изобразить графически приемлемые блуждания и их усредненную траекторию, которая должна напоминать излучину реки.

6.7 Дискретные преобразования Фурье

6.7.1 Одномерные быстрые преобразования Фурье

В состав NumPy входит `numpy.fft` – библиотека реализации быстрых преобразований Фурье (БПФ, FFT) для вычисления дискретного преобразования Фурье (ДПФ) с применением широко известного алгоритма Кули–Тьюки (Cooley-Tukey algorithm)[92]. Формула дискретного преобразования Фурье для функции, определенной в n точках f_m при $m = 1, 2, \ldots, n - 1$, используемая в NumPy:

$$F_k = \sum_{m=0}^{n-1} f_m \exp\left(-\frac{2\pi i m k}{n}\right),\ k = 0,\ 1,\ 2,\ \ldots,\ n-1. \tag{6.1}$$

В библиотеке NumPy основным методом дискретного преобразования Фурье для действительных и комплексных функций является `np.fft.fft`. Если входная функция сигнала f определяется во временно́м измерении, то выходное преобразование Фурье F определяется в частотном измерении и возвращается при вызове функции `fft(f)` в стандартном порядке: `F[:n/2]` – положительные значения частоты в возрастающем порядке, `F[n/2+1:]` содержит отрицательные значения частоты в убывающем порядке, а `F[n/2]` – (положительные и отрицательные) значения частоты Найквиста[93]. Величины `np.abs(F)`, `np.abs(F)**2` и `np.angle(F)` – это амплитудный спектр, спектр мощности и фазочастотный спектр соответственно.

Интервалы частоты, соответствующие значениям `F`, вычисляются методом `np.fft.fftfreq(n,d)`, где `d` – шаг дискретизации выборки. Для четного n это равнозначно:

$$0, \frac{1}{dn}, \frac{2}{dn}, \ldots, \frac{n/2-1}{dn}, -\frac{n/2}{dn}, -\frac{n/2-1}{dn}, \ldots, -2, -1.$$

Для смещения спектра так, чтобы компонента нулевой частоты находилась в центре, необходимо вызвать метод `np.fft.fftshift`. Для отмены этого смещения вызывается метод `np.fft.ifftshift`.

Например, рассмотрим следующую форму колебаний сигнала во временно́м измерении с добавлением некоторого синтезированного гауссова шума:

$$f(t) = 2\sin(20\pi t) + \sin(100\pi t).$$

[92] *J. W. Cooley and J. W. Tukey.* Math. Comput. 19, 297–301 (1965).

[93] Здесь предполагается, что n – четное число.

```
In [x]: A1, A2 = 2, 1
In [x]: freq1,freq2 = 10, 50
In [x]: fsamp = 500
In [x]: t = np.arange(0, 1, 1/fsamp)
In [x]: n = len(t)
In [x]: f = A1*np.sin(2*np.pi*freq1*t) + A2*np.sin(2*np.pi*freq2*t)
In [x]: f += 0.2 * np.random.randn(n)
In [x]: plt.plot(t, f)
In [x]: plt.xlabel('Time /s')
In [x]: plt.show()
```

График этой формы колебаний сигнала изображен на рис. 6.14.

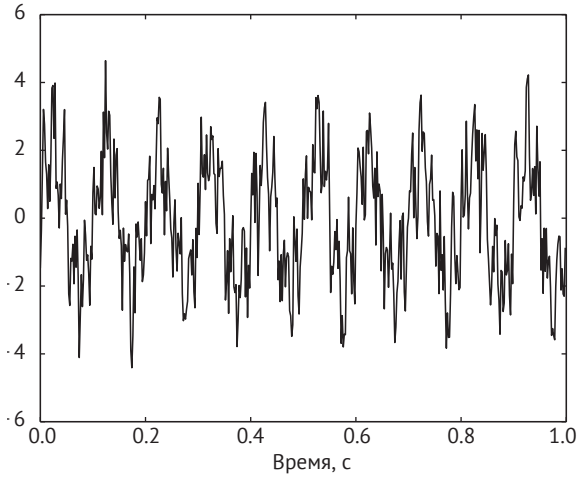


Рис. 6.13. Форма колебаний сигнала с шумом, описанная в тексте

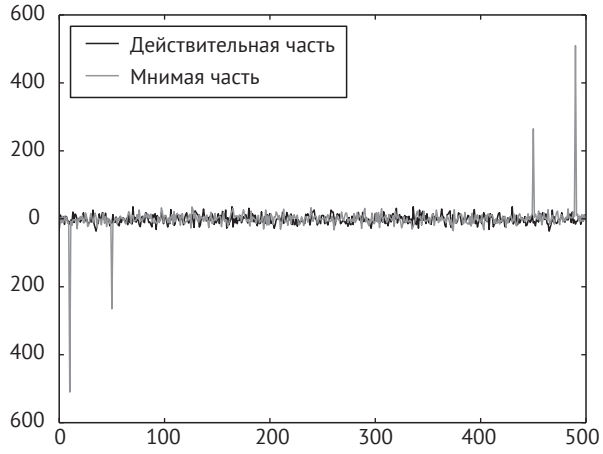


Рис. 6.15. Преобразование Фурье формы колебаний сигнала с шумом с двумя частотными компонентами, возвращаемая методом `np.fft.fft`

Преобразование Фурье для этой функции является комплексным, его действительная и мнимая компоненты изображены на рис. 6.15.

```
In [x]: F = np.fft.fft(f)
In [x]: plt.plot(F.real, 'k', label='real')
In [x]: plt.plot(F.imag, 'gray', label='imag')
In [x]: plt.legend(loc=2)
In [x]: plt.show()
```

Теперь рассмотрим смещенный амплитудный спектр с компонентой нулевой частоты в центре[94]:

```
In [x]: freq = np.fft.fftfreq(n, 1/fsamp)
In [x]: F_shifted = np.fft.fftshift(F)
In [x]: freq_shifted = np.fft.fftshift(freq)
In [x]: plt.plot(freq_shifted, np.abs(F_shifted))
In [x]: plt.xlabel('Frequency /Hz')
In [x]: plt.show()
```

Этот график изображен на рис. 6.16.

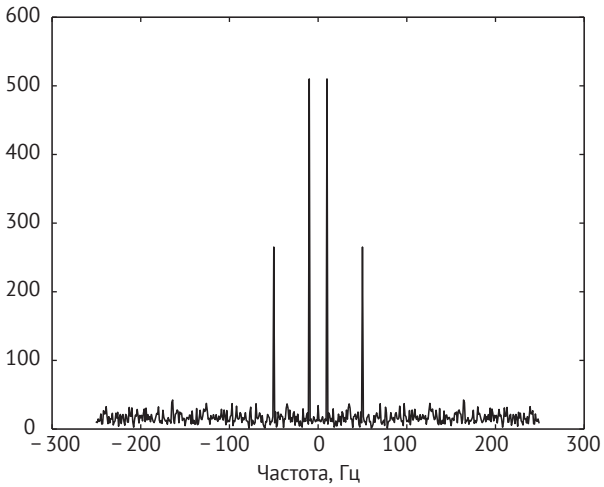


Рис. 6.16. Преобразование Фурье формы колебаний сигнала с шумом с двумя частотными компонентами, изображенное совместно с частотой

Теперь, поскольку рассматриваемая здесь входная функция действительная, ее преобразование Фурье является эрмитовым: комплексные отрицательные компоненты частоты являются сопряженными с положительными компонентами частоты, поэтому не содержат какой-либо дополнительной информации. Таким образом, необходимо работать только с первой половиной массива F. Изображение вместе с соответствующими (положительными) значениями частоты как амплитудного спектра показано на рис. 6.17:

[94] Здесь смещение выполняется только в демонстрационных целях: отметим, что в действительности нет необходимости в сдвиге, а нужно просто отобразить оба массива `freq` и `F` вместе.

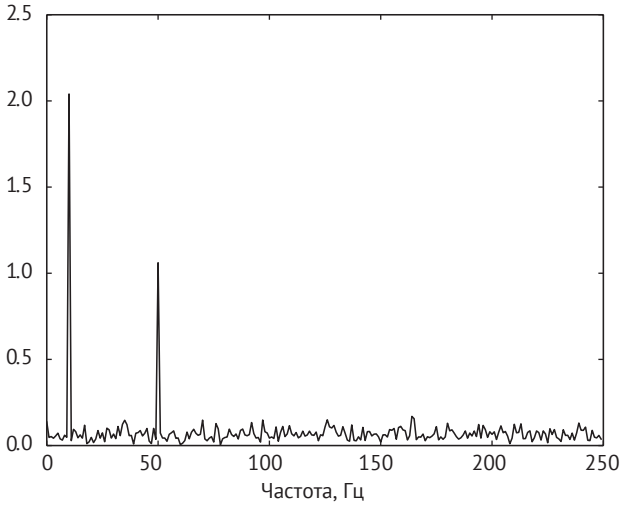


Рис. 6.17. Положительные компоненты частоты преобразования Фурье для формы колебаний сигнала с шумом, нормализованные для выделения их интенсивностей

```
In [x]: spec = 2/n * np.abs(F[:n//2])                    ❶
In [x]: plt.plot(freq[:n//2], spec, 'k')
In [x]: plt.xlabel('Frequency /Hz')
In [x]: plt.show()
```

❶ Обратите внимание: поскольку был определен способ выполнения этого дискретного преобразования Фурье, требуется коэффициент нормализации $2/n$ для точного восстановления исходных значений амплитуды каждой компоненты.

Амплитуды сигналов с частотой 10 Гц и 50 Гц легко решаются в этом спектре. Обратное преобразование Фурье определяется формулой

$$f_m = \frac{1}{n}\sum_{k=0}^{n-1} F_k \exp\left(\frac{2\pi imk}{n}\right) \quad m = 0,1,2,\ldots,n-1$$

Его возвращает метод np.fft.ifft.

Если, как уже было отмечено ранее, массив входной функции содержит действительные значения и необходимы только неотрицательные компоненты частоты, то можно воспользоваться методами `np.fft.rfft`, `np.fft.irfft`, `np.fft.rfftfreq`.

6.7.2 Двумерные быстрые преобразования Фурье

Выполнение дискретных преобразований Фурье и обратных им преобразований в двух и более измерениях обеспечивается методами `np.fft.fft2`, `np.fft.ifft2`, `np.fft.fftn` и `np.fft.ifftn`. Двумерное дискретное преобразование Фурье определяется формулой

$$F_{jk} = \sum_{p=0}^{m-1}\sum_{q=0}^{n-1} f_{pq} \exp\left[-2\pi\left(\frac{pj}{m} + \frac{qk}{n}\right)\right],$$
$$j = 0,\ 1,\ 2,\ \ldots,\ m-1;\ k = 0,\ 1,\ 2,\ \ldots,\ n-1,$$

где $j = 0, 1, 2, \ldots, m - 1$; $k = 0, 1, 2, \ldots, n - 1$.

Измерения более высокого порядка определяются аналогично.

Пример П6.23. Двумерное дискретное преобразование Фурье интенсивно используется при обработке изображений[95]. Например, умножение дискретного преобразования Фурье изображения на двумерную гауссову функцию – это общепринятый способ размытия изображения посредством уменьшения величины его высокочастотных компонентов.

Код в листинге 6.13 создает изображение случайно размещенных небольших квадратов, затем размывает изображение с помощью гауссова фильтра.

Листинг 6.13. Размытие изображения с использованием гауссова фильтра

```
# eg6-fft2 -blur.py
import numpy as np
import matplotlib.pyplot as plt

# Размер изображения, длина стороны квадрата, количество квадратов.
ncols, nrows = 120, 120
sq_size, nsq = 10, 20
# Массив изображения (0 = фон, 1 = квадрат) и массив логических значений для позиций,
# в которых разрешено добавлять квадраты так, чтобы они не касались друг друга
# и границ изображения.
image = np.zeros((nrows, ncols))
sq_locs = np.zeros((nrows, ncols), dtype=bool)
sq_locs[1:-sq_size -1:,1:-sq_size -1] = True

def place_square():
    """ Place a square at random on the image and update sq_locs. """
    # """ Размещение квадрата в случайной локации изображения и обновление sq_locs. """
    # valid_locs - массив индексов элементов True в sq_locs.
    valid_locs = np.transpose(np.nonzero(sq_locs))
    # Случайный выбор одного такого элемента и добавление квадрата так, чтобы его верхний
    # левый угол располагался в этой локации; затем обновление sq_locs.
    i, j = valid_locs[np.random.randint(len(valid_locs))]
    image[i:i+sq_size, j:j+sq_size] = 1
    imin, jmin = max(0, i-sq_size -1), max(0, j-sq_size -1)
    sq_locs[imin:i+sq_size+1, jmin:j+sq_size+1] = False

# Добавление требуемого количества квадратов в изображение.
for i in range(nsq):
    place_square()
plt.imshow(image)
plt.show()
```

---

[95] Следует отметить, что в составе библиотеки SciPy имеется полнофункциональный пакет обработки изображений `scipy.ndimage`, не рассматриваемый в этой книге. Пример в листинге 6.13 предназначен только для демонстрации синтаксиса и формата реализации двумерного быстрого преобразования Фурье в библиотеке NumPy.

```
# Применение двумерного ДПФ и центрирование значений частот.
ftimage = np.fft.fft2(image)
ftimage = np.fft.fftshift(ftimage)
plt.imshow(np.abs(ftimage))
plt.show()

# Создание и применение гауссова фильтра.
sigmax, sigmay = 10, 10
cy, cx = nrows/2, ncols/2
x = np.linspace(0, nrows, nrows)
y = np.linspace(0, ncols, ncols)
X, Y = np.meshgrid(x, y)
gmask = np.exp(-(((X-cx)/sigmax)**2 + ((Y-cy)/sigmay)**2))

ftimagep = ftimage * gmask
plt.imshow(np.abs(ftimagep))
plt.show()

# Завершающий этап: применение обратного преобразования и вывод размытого изображения.
imagep = np.fft.ifft2(ftimagep)
plt.imshow(np.abs(imagep))
plt.show()
```

Результат работы этой программы показан на рис. 6.18.

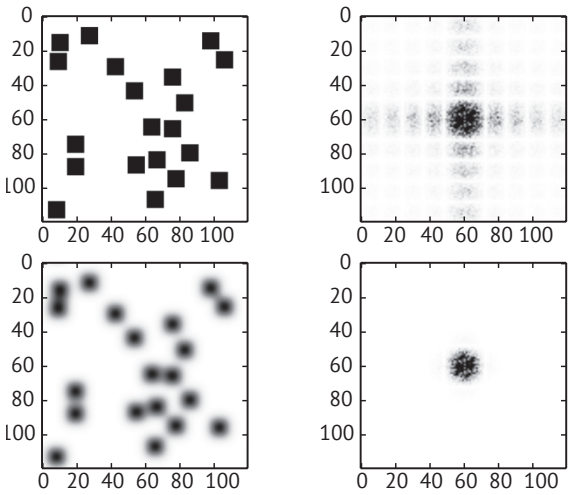


Рис. 6.18. Размытие изображения с помощью гауссова фильтра, примененного к двумерному преобразованию Фурье сгенерированной картинки

6.7.3 Упражнения

Вопросы

В6.7.1. Сравнить скорость выполнения алгоритма `np.fft.fft` из библиотеки NumPy и непосредственной (собственной) реализации формулы 6.1.

Совет: необходимо интерпретировать формулу 6.1 как операцию умножения матриц, т. е. скалярное произведение массива с n значениями функции (используются случайные значения) и массива $n \times n$ с элементами $\exp(-2\pi imk/n)$ при $m, k = 0, 1, \ldots, n-1$. Использовать магическую функцию IPython `%timeit`.

Задачи

36.7.1. Рассмотреть сигнал в терминах времени, определяемый функцией

$$f(t) = \cos(2\pi\nu t)e^{-t/\tau},$$

с частотой $\nu = 250$ Гц, затухающий экспоненциально со временем жизни $\tau = 0.2$ с. Изобразить эту функцию, сэмплируемую с частотой 1000 Гц, и ее дискретное преобразование Фурье в сравнении с частотой. Средствами подходящего графического отображения исследовать эффект аподизации в дискретном преобразовании Фурье, отсекая временну́ю последовательность после а) 0.5 с; б) 0.2 с.

36.7.2. Прямоугольную волновую форму с периодом T можно определить с помощью следующей функции:

$$f_{sq}(t) = \begin{cases} 1 & t < T/2 \\ -1 & t < T/2 \end{cases}$$

для $f(t) = f(t + nT)$ при $n = \pm 1, \pm 2, \ldots$.

Изобразить прямоугольную волновую форму с периодом $T = 1$ (и соответственно с циклической частотой $\nu = 1$) при $0 \leqslant t < 2$, используя сетку из 2048 точек времени в этом интервале. Вычислить и отобразить дискретное преобразование Фурье для этой волновой формы.

Разложение (в ряд) Фурье для этой функции представляет собой бесконечную последовательность:

$$f_{sq}(t) = \frac{4}{\pi}\sum_{k=1}^{\infty}\frac{1}{2k-1}\sin\left[2\pi(2k-1)\nu t\right].$$

Сравнить функцию прямоугольной волновой формы с этим разложением в ряд Фурье, усеченным до 3, 9 и 18 членов. Также сравнить их преобразования Фурье (нормализованные соответствующим образом): отсутствующие частоты в каждой усеченной последовательности должны выглядеть как нули в соответствующем преобразовании Фурье, в то время как присутствующие члены ряда будут иметь значения интенсивности $4/[\pi(2k-1)]$.

36.7.3. Библиотека `scipy` предоставляет подпрограмму для считывания файлов `.wav` как массивов NumPy:

```
In [x]: from scipy.io import wavfile
In [x]: sample_rate, wav = wavfile.read(<filename >)
```

Для файлов со стереозвуком массив `wav` имеет форму `(n, 2)`, где `n` – количество звуковых сэмплов.

Использовать подпрограммы `np.fft` для идентификации аккордов в звуковом файле *chords.wav*, который можно скачать здесь: https://scipython.com/ex/bfi. Какой основной аккорд содержится в этом файле?

Частоты музыкальных нот в равномерно темперированном строе, для которого A_4 = 440 Гц, представлены в форме словаря в файле *notes.py*.

Глава 7

Библиотека Matplotlib

Вероятно, Matplotlib является самым широко распространенным пакетом языка Python для графического отображения данных. Этот пакет можно использовать через процедурный интерфейс `pyplot` в очень быстро выполняемых скриптах для создания простых визуализаций данных (см. главу 3). Но, как описано в данной главе, если уделить немного больше внимания, то можно также получить изображения высокого качества для журнальных статей, книг и прочих публикаций. Функциональность для создания трехмерных графиков несколько ограничена (см. раздел 7.6), потому что эта библиотека предназначена главным образом для формирования двумерных изображений.

7.1 Линейные графики и точечные диаграммы

Matplotlib – крупный пакет, организованный в форме иерархической структуры: на самом высоком уровне находится модуль `matplotlib.pyplot`. Этот модуль предоставляет «среду конечного автомата» с интерфейсом, похожим на интерфейс MATLAB, и позволяет пользователю добавлять элементы графика (точки данных, линии, аннотации и т. п.) с помощью простых вызовов функций. Данный интерфейс был представлен в главе 3.

На более низком уровне, который обеспечивает более продвинутое и гибко настраиваемое использование, Matplotlib предлагает объектно-ориентированный интерфейс, позволяющий создавать объект рисунка (figure) с прикрепленным к нему одним или несколькими объектами осей (axes). В дальнейшем большинство изображений, аннотаций и прочих специализированных объектов существуют на этих осях. Именно такой подход применяется и описывается в этой главе.

Для использования библиотеки Matplotlib подобным способом рекомендуется выполнить следующие команды импорта:

```
import matplotlib.pyplot as plt
import numpy as np
```

7.1.1 Построение графика на одном объекте оси

Объект самого верхнего уровня, содержащий все элементы графика, называется `Figure`. Для создания объекта рисунка `Figure` вызывается метод `plt.figure`. Какие-либо аргументы не требуются, но дополнительные параметры специальной настройки можно задать, устанавливая значения, описанные в табл. 7.1. Например:

```
In [x]: # По умолчанию рисунок с заголовком "Figure 1".
In [x]: fig = plt.figure(
)
In [x]: # Небольшой рисунок (4.5" x 2") с красным фоном.
In [x]: fig = plt.figure('Population density', figsize=(4.5, 2.),
  ....:                  facecolor='red')
```

Таблица 7.1. Аргументы для метода `plt.figure`

Аргумент	Описание
num	Идентификатор рисунка – если значение не задано, то используется целое число, начиная с 1, и увеличивается на 1 для каждого следующего создаваемого рисунка. Кроме того, использование строки установит заголовок окна этого рисунка при выводе его с помощью метода plt.show()
figsize	Кортеж размеров рисунка (width, height) – к сожалению, только в дюймах
dpi	Разрешение рисунка в точках на дюйм
facecolor	Цвет фона рисунка
edgecolor	Цвет границ рисунка

Чтобы действительно изобразить данные графически, необходимо создать объект оси `Axes` – область рисунка, содержащую оси, графические метки, надписи, линии и маркеры графика и т. п. Самый простой рисунок, состоящий из одного объекта `Axis`, создается и возвращается следующим методом:

```
In [x]: ax = fig.add_subplot()
```

В объекте `Axis ax` можно действительно графически изображать данные с помощью метода `ax.plot`. Самые важные свойства метода `plot` были описаны в главе 3. Но здесь следует отметить, что метод `plot` в действительности возвращает список (`list`) объектов, представляющих графически изображаемые линии. При простейшем варианте использования изображается одна линия, поэтому список состоит из одного объекта типа `Line2D`, который можно присвоить переменной, если это необходимо. Рассмотрим более полный пример – приведенное ниже сравнение цепной линии $y = \cosh(x)$ и ее параболическую аппроксимацию $y = 1 + x^2/2$.

```
import matplotlib.pyplot as plt
import numpy as np

fig = plt.figure()
ax = fig.add_subplot()

x = np.linspace(-2, 2, 1000)
line_cosh, = ax.plot(x, np.cosh(x))                    ❶
line_quad, = ax.plot(x, 1 + x**2 / 2)

plt.show()
```

❶ Обратите внимание на синтаксис `line_cosh, = ...` для присваивания переменной `line_cosh` возвращаемого объекта линии, а не списка, содержащего этот объект.

Две изображенные этим кодом линии показаны на рис. 7.1.

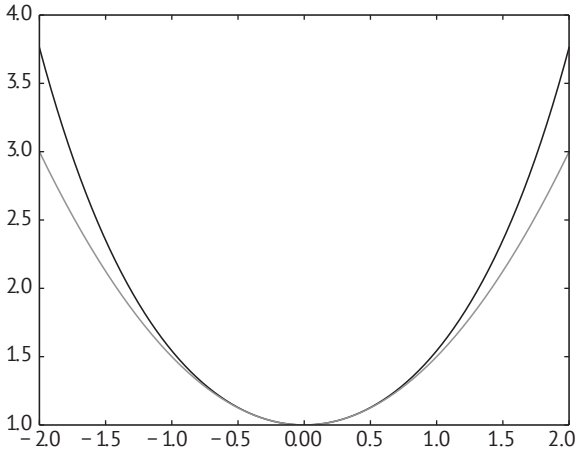


Рис. 7.1. Простой график из двух линий в одном объекте Axes

Если в методе `fig.add_subplot()` не требуется передача аргументов, то объекты fig и ax можно создать в одной строке кода с помощью удобной функции `plt.subplots()`[96]:

```
fig, ax = plt.subplots()
```

7.1.2 Границы графика

По умолчанию Matplotlib отображает все данные, переданные в метод `plot`, и устанавливает границы оси соответствующим образом. Чтобы установить для границы оси другое значение, используются методы `ax.set_xlim` и `ax.set_ylim`. Кроме того, можно установить обе границы или значения границ по отдельности с помощью аргументов `left`, `right` (или `xmin`, `xmax`) и `bottom`, `top` (или `ymin`, `ymax`). Незаданные значения границ остаются неизменными. Например:

```
x = np.linspace(-3, 3, 1000)
y = x**3 + 2 * x**2 - x - 1
fig = plt.figure()
ax = fig.add_subplot()
ax.plot(x, y)

ax.set_xlim(-1, 2)           # Установлены границы оси x: от -1 до 2.
ax.set_ylim(bottom=0)        # ymin=0: график будет "отсечен" по нижней границе bottom.
```

Если значение `bottom` больше значения `top` или значение `right` меньше значения `left`, то направление соответствующей оси будет изменено на противоположное, т. е. значения по этой оси будут уменьшаться слева направо (или снизу вверх) (см. задачу 37.4.5).

[96] В эту функцию также можно передать аргументы `nrows` и `ncols` для сетки нескольких внутренних графиков и `subplot_kw` – словарь именованных аргументов, передаваемых при вызове `add_subplot` для каждого внутреннего графика. Дополнительные именованные аргументы передаются при вызове `plt.figure()`.

Если необходимо изменить направление оси на противоположное без изменения значений границ, то вызываются методы `ax.invert_xaxis()` и `ax.invert_yaxis()`, позволяющие сделать это.

7.1.3 Стили линий, маркеры и цвета

Как мы уже видели ранее, стиль графика можно определить, передавая дополнительные аргументы в метод `plot()`. По умолчанию для линии назначается сплошной стиль, толщина 1.5 пт и цвет, определяемый порядком, в котором эта линия добавляется в график.

Другой стиль линии можно выбрать из предварительно определенных вариантов с помощью аргумента `linestyle` (или просто `ls`). Допустимые строковые значения для передачи в этом аргументе (включая пустую строку для отсутствия отображения линии) показаны в табл. 7.2.

Таблица 7.2. Стили линий Matplotlib

	(Линия не отображается)
-	Сплошная
--	Штриховая
:	Пунктирная
-.	Штрихпунктирная

Можно еще более подробно определить стиль линии, передавая в аргументе `dashes` последовательность значений, описывающих повторяющийся шаблон штрихов в пунктах (пт). Например, `dashes=[2, 4, 8, 4, 2, 4]]` представляет шаблон из точки (2 пт), пробела (4 пт), штриха (8 пт), пробела (4 пт), точки (2 пт), пробела (4 пт), который должен повторяться, формируя стиль линии. Для получения того же результата можно вызвать метод изображаемой линии `set_dashes`, как показано в следующем фрагменте кода:

```
x = np.linspace(-np.pi, np.pi, 1000)
line, = plt.plot(x, np.sin(x))
line.set_dashes([2, 4, 8, 4, 2, 4])      # Шаблон точка-штрих-точка.
```

Толщина линии определяется значением аргумента `lineweight` (или просто `lw`) в пунктах (пт).

Цвет линии передается в аргументе `color` (или просто `c`), используемом одним из следующих способов:

- строка: буква или имя – одно из значений, приведенных в табл. 7.3. Обозначенные одной буквой цвета более светлые, а заданные по умолчанию «живые» цвета более приятны для глаз;
- строка: HTML-строка из шести шестнадцатеричных цифр с префиксом `'#'`, например `'#ffff00'` обозначает желтый цвет;
- строка: строковое представление числа типа `float` со значением от `0.` до `1.` (например, `'0.4'`) определяет шкалу оттенков серого цвета от черного (0.) до белого (1.);
- кортеж чисел типа `float` в диапазоне от `0.` до `1.`: компоненты RGB, например `(0.5, 0., 0.)` – это темно-красный цвет.

Таблица 7.3. Буквенные коды цветов Matplotlib

Коды основных цветов	Живые цвета
b = синий	tab:blue
g = зеленый	tab:orange
r = красный	tab:green
c = бирюзовый	tab:red
m = фиолетовый	tab:purple
y = желтый	tab:brown
k = черный	tab:pink
w = белый	tab:gray
	tab:olive
	tab:cyan

По умолчанию объект `Line2D`, создаваемый при вызове метода `plot` в объекте `Axes`, не включает маркеры: символы, выводимые в каждой точке данных графика. Для их добавления необходимо определить один из односимвольных кодов маркера, приведенных в табл. 7.4, и передать его в аргументе `marker`:

```
ax.plot(x, y, marker='v')        # Маркер: треугольник вершиной вниз.
```

Таблица 7.4. Некоторые стили маркеров Matplotlib (односимвольные строковые коды)

Код	Маркер	Описание
.	·	Точка
o	○	Кружок
+	+	Знак плюс
x	×	Крестик
D	◇	Ромб
v	▽	Треугольник вершиной вниз
^	△	Треугольник вершиной вверх
s	□	Квадрат
*	★	Звезда (пятиконечная)

Другие свойства маркеров можно определить с помощью аргументов, перечисленных в табл. 7.5.

Таблица 7.5. Свойства маркеров Matplotlib

Аргумент	Сокращенный вариант	Описание
`markersize`	ms	Размер маркера в пунктах (пт)
`markevery`		Если явно установлено положительное целочисленное значение N, то маркер выводится в каждой N-й точке. По умолчанию установлено значение None – маркер выводится в каждой точке

Окончание табл. 7.5

Аргумент	Сокращенный вариант	Описание
`markerfacecolor`	mfc	Цвет заливки маркера
`markeredgecolor`	mec	Цвет границы маркера
`markeredgewidth`	mew	Толщина границы маркера в пунктах (пт)

Свойства маркеров Matplotlib можно настраивать более детально, подробности см. в официальной документации (https://matplotlib.org/api/markers_api.html).

7.1.4 Точечные диаграммы

Обычная двумерная точечная диаграмма изображает данные как точки в декартовой системе координат (осей). Иногда не имеет смысла или не приносит никакой пользы упорядочивание данных, следовательно, нет необходимости соединять точки данных линиями. Метод `pyplot.scatter` создает точечную диаграмму (график рассеяния). В дополнение к одномерным последовательностям *x*- и *y*-данных, как для `pyplot.plot`, цвета и размеры маркеров точек данных можно устанавливать отдельно, передавая последовательность соответствующих значений той же длины, что и сама последовательность данных, в аргументах `c` и `s` соответственно. Размер маркера указывается в пт^2 (в квадратных пунктах), так что площадь маркеров пропорциональна значениям, передаваемым в аргументе `s`. Работа с размером маркеров является общепринятым способом обозначения третьего измерения данных, как показано в следующем примере.

Пример П7.1. Для исследования зависимости между уровнем рождаемости, средней продолжительностью жизни и доходом на душу населения можно воспользоваться точечной диаграммой. Размеры маркеров устанавливаются пропорционально объему ВВП на душу населения, но их необходимо немного масштабировать, чтобы они не становились слишком большими (см. рис. 7.2).

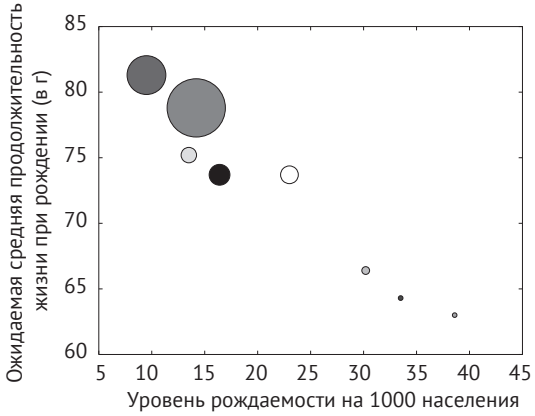


Рис. 7.2. Точечная диаграмма с различными размерами маркеров, обозначающими объем ВВП в каждой стране

Листинг 7.1. Точечная диаграмма демографических данных по восьми странам

```
# eg7-scatter.py

import numpy as np
import matplotlib.pyplot as plt

countries = ['Brazil', 'Madagascar', 'S. Korea', 'United States',
             'Ethiopia', 'Pakistan', 'China', 'Belize']
# Уровень рождаемости на 1000 населения.
birth_rate = [16.4, 33.5, 9.5, 14.2, 38.6, 30.2, 13.5, 23.0]
# Ожидаемая средняя продолжительность жизни при рождении, в г.
life_expectancy = [73.7, 64.3, 81.3, 78.8, 63.0, 66.4, 75.2, 73.7]
# Доход на душу населения, определенный в 2000 г. в долл. США.
GDP = np.array([4800, 240, 16700, 37700, 230, 670, 2640, 3490])

fig, ax = plt.subplots()

# Некоторые произвольно выбираемые цвета:
colors = range(len(countries))
ax.scatter(birth_rate, life_expectancy, c=colors , s=GDP/20)
ax.set_xlim(5, 45)
ax.set_ylim(60, 85)
ax.set_xlabel('Birth rate per 1000 population')
ax.set_ylabel('Life expectancy at birth (years)')

plt.show()
```

7.2 Специализированная настройка и улучшение качества графика

7.2.1 Линии сетки

Линии сетки – это вертикальные (идущие от оси *x*) и горизонтальные (идущие от оси *y*) прямые на графике, помогающие точнее определить положение числовых значений точек данных. По умолчанию линии сетки не отображаются, но их можно вывести, вызывая метод `grid` из объекта `Axis` (для добавления и горизонтальных, и вертикальных линий сетки) или используя объект `xaxis` или `yaxis` существующего экземпляра `Axis` (для выбора требуемых линий сетки). Например:

```
ax.yaxis.grid(True)       # Включение отображения горизонтальных линий сетки.
```

или

```
ax.grid(True)             # Включение отображения всех линий сетки.
```

Свойства линий сетки устанавливаются с помощью аргументов `linestyle`, `linewidth`, `color` и т. д. точно так же, как и свойства линий графика.

Два типа линий сетки соответствуют главным и второстепенным меткам на осях (см. ниже): их можно выбрать с помощью аргумента `which`, который может

принимать значения 'major', 'minor' или 'both'. По умолчанию (если значение не задано явно) принимается which='major'.

```
ax.xaxis.grid(True , which='minor', c='b')   # Второстепенные линии сетки для оси x, цвет
                                             # синий.
```

7.2.2 Логарифмические шкалы

По умолчанию Matplotlib отображает данные в линейном масштабе. Для установки логарифмической шкалы необходимо вызвать соответствующий метод или оба метода объекта Axis:

```
ax.set_xscale('log')
ax.set_yscale('log')
```

По умолчанию используются логарифмы по основанию 10 (десятичные логарифмы), но основание (целочисленное) можно установить с помощью дополнительных аргументов basex и basey. Неположительные значения данных будут маскироваться как некорректные по умолчанию. Если требуется обработка отрицательных значений «симметрично» относительно положительных значений, как, например, $\log(-|x|) = -\log(|x|)$, то используется значение аргумента 'symlog' вместо 'log'. См. также вопрос В7.4.1.

7.2.3 Добавление заголовков, надписей и описаний условных обозначений

Обозначения (метки) осей можно добавить во внутренний объект Axes с помощью методов ax.set_xlabel и ax.set_ylabel.

Надписи с описанием условных обозначений линий и прочих элементов графика определяются при добавлении атрибута label в вызов метода plt.plot. Но само описание условных обозначений (легенда) не выводится, если не вызван явно метод legend из отображаемого объекта Axis (например, ax.legend()). Внешний вид легенды можно настраивать весьма подробно, но чаще всего требуется передаваемый в метод legend() дополнительный аргумент loc, определяющий положение легенды на графике (см. табл. 3.1).

Существуют два типа заголовка, которые можно определить для рисунка: fig.suptitle добавляет отцентрированный заголовок для всего рисунка в целом. Рисунок может содержать более одного внутреннего графика, поэтому ax.title добавляет заголовок к одному конкретному внутреннему графику[97].

Пример П7.2. Данные из файла *eg7-marriage-ages.txt*, который можно скачать по адресу https://scipython.com/eg/bag, представляют средний возраст при первом вступлении в брак в США по 13 десятилетиям, начиная с 1890 г., и изображаются на графике программой из листинга 7.2. Линии сетки отображаются по обеим осям с помощью вызова метода ax.grid(), также используются специально настроенные маркеры для самих точек данных (см. рис. 7.3).

---

[97] Более подробно об этом см. документацию https://matplotlib.org/api/legend_api.html.

Рис. 7.3. Средний возраст при первом вступлении в брак в США, 1890–2010 гг.

Листинг 7.2. Средний возраст при первом вступлении в брак в США в зависимости от времени

```
# eg7-marriage-ages.py
import numpy as np
import matplotlib.pyplot as plt

year, age_m, age_f = np.loadtxt('eg7-marriage-ages.txt', unpack=True, skiprows=3)
fig, ax = plt.subplots()

# Изображение значений возраста мужчин и женщин соответствующими символами как маркерами.
ax.plot(year, age_m, marker='$\u2642$', markersize=14, c='blue', lw=2,
mfc='blue', mec='blue')
ax.plot(year, age_f, marker='$\u2640$', markersize=14, c='magenta', lw=2,
mfc='magenta', mec='magenta')
ax.grid()

ax.set_xlabel('Year')
ax.set_ylabel('Age')
ax.set_title('Median age at first marriage in the USA, 1890-2010')

plt.show()
```

Пример П7.3. Хронологические данные о населении пяти городов США содержатся в файлах *boston.tsv*, *houston.tsv*, *detroit.tsv*, *san_jose.tsv*, *phoenix.tsv* как столбцы, разделенные символами табуляции (год, численность населения). Эти файлы можно скачать здесь: https://scipython.com/eg/bat.

Программа в листинге 7.3 графически отображает эти данные в одной системе координатных осей с различными стилями линий для каждого города.

Листинг 7.3. Численность населения пяти городов США в зависимости от времени

```
# eg7-populations.py
import matplotlib.pyplot as plt
import numpy as np

fig, ax = plt.subplots()

cities = ['Boston', 'Houston', 'Detroit', 'San Jose', 'Phoenix']
# Стили линий: сплошная, штриховая, пунктирная, штрихпунктирная и точка-точка-штрих.
linestyles = [{'ls': '-'}, {'ls': '--'}, {'ls': ':'}, {'ls': '-.'},
              {'dashes': [2, 4, 2, 4, 8, 4]}]
for i, city in enumerate(cities):
    filename = '{}.tsv'.format(city.lower()).replace(' ', '_')                    ❶
    yr, pop = np.loadtxt(filename, unpack=True)
    line, = ax.plot(yr, pop/1.e6, label=city, c='k', **linestyles[i])
ax.legend(loc='upper left')
ax.set_xlim(1800, 2020)
ax.set_xlabel('Year')
ax.set_ylabel('Population (millions)')
plt.show()
```

❶ Обратите внимание: название города используется для определения имени соответствующего файла.

Полученный график показан на рис. 7.4.

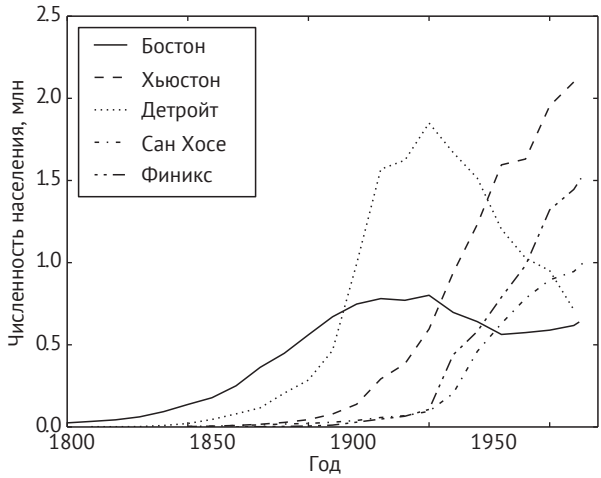


Рис. 7.4. Графики изменения численности населения в пяти городах США

7.2.4 Свойства шрифта

Текстовые элементы графика (заголовки, легенда, метки осей и т. п.) можно настраивать с помощью аргументов, перечисленных в табл. 7.6. Например:

```
ax.title('Plot Title', fontsize=18, fontname='Times New Roman', color='blue')
```

Таблица 7.6. Аргументы, определяющие свойства шрифта для текстовых элементов графика

Аргумент	Описание
fontsize	Размер шрифта в пт (например, 12, 16)
fontname	Название шрифта (например, 'Courier', 'Arial')
family	Семейство (тип) шрифта (например, 'sans-serif', 'cursive', 'monospace')
fontweight	Начертание шрифта (например, 'normal', 'bold')
fontstyle	Стиль шрифта (например, 'normal', 'italic')
color	Любой спецификатор цвета Matplotlib (например, 'r', '#ff00ff')

Для использования одинаковых свойств шрифта для всех текстовых элементов проще всего установить значения параметров конфигурации Matplotlib `rc`, используя для этого словарь значений. Для этого потребуется отдельная предварительная команда импорта[98]:

```
from matplotlib import rc
font_properties = {'family' : 'monospace',
                   'weight' : 'bold',
                   'size'   : 22}
rc('font', **font_properties)                                    ❶
# Теперь все текстовые элементы будут отображаться на графиках моноширинным полужирным
# шрифтом размером 22 пт.
```

❶ Напомню, что синтаксис `**kwargs` позволяет передавать пары (ключ, значение) из словаря `kwargs` в любой метод (функцию) как именованные аргументы (см. раздел 4.2.2).

7.2.5 Штриховые метки на осях

Matplotlib наилучшим образом размещает метки, представляющие значения (штриховые метки), на каждой оси, но иногда возникает необходимость в специализированной настройке таких меток, например более или менее частое расположение штриховых меток или изменение соответствующих им надписей.

Чаще всего требуется просто назначить для штриховых меток заданную последовательность соответствующих значений: это выполняется при помощи методов `ax.set_xticks` и `ax.set_yticks` из объекта `Axes` графика. Например:

```
ax.set_xticks([0, 1, 3.5, 6.5, 15])
```

[98] Также можно отредактировать файл конфигурации Matplotlib *matplotlibrc*, чтобы установить разнообразные типы параметров и стилей графика, см. документацию здесь: https://matplotlib.org/users/customizing.html.

Следует отметить, что штриховые метки не обязательно размещаются с равномерным шагом.

Для замены существующих числовых подписей к штриховым меткам передается последовательность строк соответствующей длины в методы `ax.set_xticklabels` и `ax.set_yticklabels`, как показано в примере П7.4 (листинг 7.4)[99].

Пример П7.4. Программа в листинге 7.4 создает график экспоненциального периода радиоактивного распада, описываемый формулой $y = Ne^{-t/\tau}$, размеченный по значениям времени существования ($n\tau$ при $n = 0, 1, \ldots$), так что после каждого интервала времени существования значение y уменьшается с коэффициентом, кратным e. График показан на рис. 7.5.

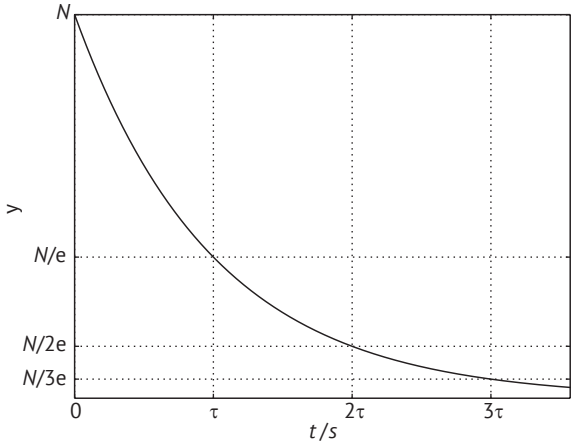


Рис. 7.5. График экспоненциального периода радиоактивного распада со специально настроенными подписями к штриховым меткам на осях

Листинг 7.4. Экспоненциальный период радиоактивного распада в интервалах времени существования

```
# eg7-ticks-exp-decay.py
import numpy as np
import matplotlib.pyplot as plt

# Начальное значение y при t=0, время существования (c).
N, tau = 10000, 28
# Максимальное рассматриваемое время (c).
tmax = 100
# Подходящая сетка точек времени и сам экспоненциальный график распада.
t = np.linspace(0, tmax, 1000)
```

[99] Следует отметить, что настройка подписей к штриховым меткам напрямую описанным здесь способом до некоторой степени отделяет график от исходных данных. Отдельный модуль `matplotlib.ticker` предназначен для конфигурирования размещения и форматирования штриховых меток на осях: API этого модуля не рассматривается в данной книге, но он подробно описан в документации: https://matplotlib.org/api/ticker_api.html.

```
y = N * np.exp(-t/tau)

fig = plt.figure()
ax = fig.add_subplot()
ax.plot(t, y)

# Число интервалов времени существования, входящих в изображаемый на графике отрезок времени.
ntau = tmax // tau + 1
# Штриховые метки xticks при 0, tau, 2*tau, ..., ntau*tau; yticks для соответствующих
# значений y.
xticks = [i*tau for i in range(ntau)]
yticks = [N * np.exp(-i) for i in range(ntau)]
ax.set_xticks(xticks)
ax.set_yticks(yticks)

# Подписи к штриховым меткам xtick: 0, tau, 2tau, ...
xtick_labels = [r'$0$', r'$\tau$'] + [r'${}\tau$'.format(k) for k in range(2, ntau)]    ❶
ax.set_xticklabels(xtick_labels)

# Соответствующие подписи к штриховым меткам ytick: N, N/e, N/2e, ...
ytick_labels = [r'$N$', r'$N/e$'] + [r'$N/{}e$'.format(k) for k in range(2, ntau)]      ❷
ax.set_yticklabels(ytick_labels)

ax.set_xlabel(r'$t\;/\mathrm{s}$')
ax.set_ylabel(r'$y$')
ax.grid()
plt.show()
```

❷ Подписи к штриховым меткам по оси *x*: 0, τ, 2τ, ...
❷ Подписи к штриховым меткам по оси *y*: *N*, *N*/*e*, *N*/2*e*, ...

Обратите внимание: длина последовательности подписей к штриховым меткам обязательно должна соответствовать длине списка требуемых значений штриховых меток.

Для одновременного удаления всех подписей к штриховым меткам необходимо задать для них пустой список, например:

```
ax.set_yticklabels([])
```

При этом сами штриховые метки сохраняются. Если нужно удалить и подписи, и штриховые метки на оси, то используется такой способ:

```
ax.set_yticks([])
```

Существуют два типа штриховых меток на осях: основные и вспомогательные. По умолчанию отображаются только основные штриховые метки. Более мелкие и чаще расположенные вспомогательные штриховые метки проще всего сделать видимыми, вызвав метод

```
ax.minorticks_on()
```

Более детальную специализированную настройку штриховых меток на осях и подписей к ним, включая отображение вспомогательных штриховых меток только на одной оси, можно выполнить с помощью удобного метода `ax.tick_params`, который принимает аргументы, описанные в табл. 7.7.

Таблица 7.7. Наиболее часто используемые аргументы для метода ax.tick_params

Аргумент	Описание
axis	Настраиваемая ось: `'x'`, `'y'` или `'both'`. По умолчанию `'both'`
which	Тип настраиваемых штриховых меток: `'major'`, `'minor'` или `'both'`. По умолчанию `'major'`
direction	Ориентация (направление) штриховых меток: `'in'`, `'out'` или `'inout'`. По умолчанию `'in'`
length	Длина штриховых меток в пт
width	Толщина штриховых меток в пт
pad	Расстояние между штриховой меткой и подписью к ней в пт
labelsize	Размер подписи к штриховой метке в пт
color	Цвет штриховой метки (любой спецификатор Matplotlib)
labelcolor	Цвет подписи к штриховой метке (любой спецификатор Matplotlib)

Объекты `ax.xaxis` и `ax.yaxis` содержат метод `set_ticks_position`, который принимает единственный аргумент, определяющий, где отображаются штриховые метки, – для `ax.xaxis`: `'top'`, `'bottom'`, `'both'` (по умолчанию) или `'none'`; для `ax.yaxis`: `'left'`, `'right'`, `'both'` (по умолчанию) или `'none'`.

Пример П7.5. Программа в листинге 7.5 создает график с основными и вспомогательными штриховыми метками на осях, настраивает их так, чтобы они были более плотными и широкими, чем задано по умолчанию, и чтобы основные штриховые метки были направлены внутрь и наружу относительно области графика.

Листинг 7.5. Специально настроенные штриховые метки

```
# eg7-tick -customization.py

import numpy as np
import matplotlib.pyplot as plt

# Выбор функций в rn с точками по оси абсцисс при 0 <= x < 1.
rn = 100
rx = np.linspace(0, 1, rn, endpoint=False)

def tophat(rx):
    """ Top hat function: y = 1 for x < 0.5, y = 0 for x >= 0.5 """
    # """ Прямоугольная функция: y = 1 при x < 0.5, y = 0 при x >= 0.5 """
    ry = np.ones(rn)
    ry[rx >=0.5] = 0
    return ry
```

```
# Словарь функций для выбора.
ry = {'half-sawtooth': lambda rx: rx.copy(),
      'top-hat': tophat,
      'sawtooth': lambda rx: 2 * np.abs(rx -0.5)}

# Повторение выполнения выбранной функции nrep раз.
nrep = 4
x = np.linspace(0, nrep, nrep*rn, endpoint=False)
y = np.tile(ry['top-hat'](rx), nrep)                                    ❶

fig, ax = plt.subplots()
ax.plot(x, y, 'k', lw=2)

# Добавление небольшого сдвига по изображаемой линии для улучшения визуализации.
ax.set_ylim(-0.1, 1.1)
ax.set_xlim(x[0]-0.5, x[-1]+0.5)
# Настройка штриховых меток и включение отображения сетки на графике.
ax.minorticks_on()
ax.tick_params(which='major', length=10, width=2, direction='inout')
ax.tick_params(which='minor', length=5, width=2, direction='in')
ax.grid(which='both')
plt.show()
```

❶ Этот метод `np.tile` создает массив, повторяя содержимое заданного массива `nrep` раз. Для построения графика другой периодической функции выберите здесь `'half-sawtooth'` или `'sawtooth'`.

Построенный график показан на рис. 7.6.

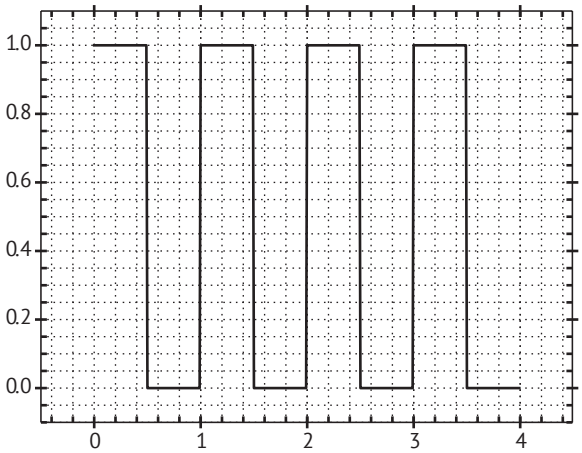


Рис. 7.6. График периодической функции, изображенный на схеме с линиями сетки и специально настроенными штриховыми метками осей

7.2.6 Диаграммы погрешностей

Для создания отображаемой линии графика с диаграммами погрешностей используется метод `ax.errorbar` вместо метода `ax.plot`. В дополнение к обычным аргументам `plot` метод `errorbar` позволяет задать характеристики погрешностей в *x*- и *y*-координатах посредством передачи следующих типов значений в аргументах `xerr` и `yerr`:

- `None`: для этой координаты не создавать диаграмму погрешностей;
- скалярное значение (например, `xerr=0.2`): все значения связываются с симметричными диаграммами погрешностей в плюс/минус-диапазоне заданного значения (т. е. ±2);
- объект типа массив длиной `n` или формы `(n, 1)` (например, `yerr=[0.1, 0.15, 0.1]`): симметричные диаграммы погрешностей отображаются в плюс/минус-диапазоне значений в этой последовательности для каждой из *n* точек данных (т. е. ±0.1, ±0.15, ±0.1);
- объект типа массив формы `(2, n)` (т. е. две строки для каждой из `n` точек данных): диаграммы погрешностей, которые могут быть асимметричными, отображаются с использованием отрицательных значений из первой строки и положительных значений из второй строки.

Таблица 7.8. Наиболее часто используемые аргументы для метода ax.errorbar

Аргумент	Описание
`x, y`	Данные для построения графика
`xerr, yerr`	Погрешности по координатам *x* и *y*; описание см. в тексте
`fmt`	Символ формата графика (маркер для точки данных). Чтобы вывести только диаграммы погрешностей, необходимо установить значение None или задать пустую строку ''
`ecolor`	Любой спецификатор цвета Matplotlib для диаграмм погрешностей. По умолчанию None – используется цвет соединительной линии между маркерами данных
`elinewidth`	Толщина линий диаграммы погрешностей в пунктах (пт). Для определения той же толщины линии, что и для отображаемых данных, используется значение None
`capsize`	Длина штрихов-ограничителей диаграммы погрешностей в пунктах (пт). По умолчанию None – диаграммы погрешностей без штрихов-ограничителей
`errorevery`	Положительное целое число, определяющее периодичность выборочного отображения диаграмм погрешностей, например при errorevery=10 диаграммы погрешностей отображаются только для каждой 10-й точки данных

Внешний вид диаграмм погрешностей можно настроить с помощью аргументов, описанных в табл. 7.8. Например:

```
# Некоторые данные:
x = np.array([ 0.3, 0.5, 0.7, 0.9])
y = np.array([ 1. , 2. , 2.5, 3.9])
# Постоянные симметричные диаграммы погрешностей в диапазоне +/-0.05 для данных по оси x.
xerr = 0.05
# Асимметричные переменные диаграммы погрешностей для данных по оси y.
yerr = np.array([[ 0.1 ,  0.25,  0.5 ,  0.4 ],
                 [ 0.1 ,  0.15,  0.2 ,  0.  ]])
ax.errorbar(x, y, yerr, xerr, fmt='o', ls='')
```

Пример П7.6. Перед оперением некоторые виды птиц теряют вес (массу) пропорционально площади поверхности крыльев для максимального увеличения аэродинамической эффективности. В файле *fledging-data.csv*, доступном по адресу https://scipython.com/eg/bad, приведены значения нагрузки на крыло (отношение массы тела к площади крыльев) как средние значения для двух родов стрижей за две недели до оперения с соответствующими погрешностями наблюдений[100].

Программа в листинге 7.6 выполняет взвешенную подгонку этих данных и создает график с диаграммами погрешностей.

Листинг 7.6. Изменения значений нагрузки на крыло для стрижей перед оперением

```
# eg7-fledging.py
import numpy as np
import matplotlib.pyplot as plt

# Считывание данных: день перед оперением, нагрузка на крыло и погрешность для двух родов.
dt = np.dtype([('day', 'i2'), ('wl1', 'f8'), ('wl1-err', 'f8'),
               ('wl2', 'f8'), ('wl2-err', 'f8')])
data = np.loadtxt('fledging -data.csv', dtype=dt, delimiter=',')

# Взвешенная подгонка экспоненциального уменьшения к данным. Это задача линейной оценки по
# методу наименьших квадратов, так как y = Aexp(-Bx) => ln y = ln A - Bx = mx + c.
p1_fit = np.poly1d(np.polyfit(data['day'], np.log(data['wl1']), 1,                    ❶
                   w=np.log(data['wl1'])**-2))
p2_fit = np.poly1d(np.polyfit(data['day'], np.log(data['wl2']), 1,
                   w=np.log(data['wl2'])**-2))
wl1fit = np.exp(p1_fit(data['day']))
wl2fit = np.exp(p2_fit(data['day']))

# Отображение точек данных с соответствующими диаграммами погрешностей и результатами
подгонки.
fig, ax = plt.subplots()

# Данные wl1: белые кружки, черные границы, с диаграммами погрешностей.
ax.errorbar(data['day'], data['wl1'], yerr=data['wl1-err'], ls='', marker='o',
            color='k', mfc='w', mec='k', capsize=3)
ax.plot(data['day'], wl1fit, 'k', lw=1.5)

# Данные wl2: черные закрашенные кружки, с диаграммами погрешностей.
ax.errorbar(data['day'], data['wl2'], yerr=data['wl2-err'], ls='', marker='o',
            color='k', mfc='k', mec='k'), capsize=3
ax.plot(data['day'], wl2fit , 'k', lw=1.5)

ax.set_xlim(15, 0)
ax.set_ylim(0.003, 0.012)
ax.set_xlabel('days pre-fledging')
ax.set_ylabel('wing loading ($\mathrm{g\,mm^{-2}}$)')
plt.show()
```

❶ Точки данных при подгонке взвешиваются с коэффициентом $1/\sigma^2$, где σ – оцениваемый уровень одного среднеквадратического (стандартного) отклонения при измерениях.

[100] *J. Wright et al.* Proc. R. Soc. B 273, 1895 (2006).

На рис. 7.7 показан результат этой подгонки. Для исследуемых родов стрижей сначала заметно различие в средних значениях нагрузки на крыло, затем наблюдается сходимость по мере приближения к сроку оперения.

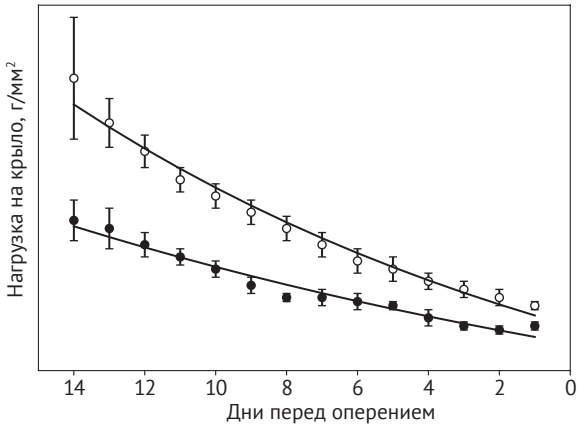


Рис. 7.7. Подгонка временны́х рядов значений нагрузки на крыло для двух родов неоперившихся стрижей

7.2.7 Несколько графиков в одном рисунке

Для создания рисунка с несколькими внутренними графиками (т. е. с несколькими объектами `Axes`) из объекта `Figure` вызывается метод `add_subplot`, аргумент которого определяет, где должен быть размещен данный внутренний график. Каждый вызов этого метода возвращает объект `Axes`. Рисунки с более чем 10 внутренними графиками встречаются весьма редко, поэтому обычный аргумент представляет собой трехзначное число, где каждая цифра обозначает количество строк, количество столбцов и количество внутренних графиков. Количество внутренних графиков увеличивается по столбцам в каждой строке, затем выполняется переход в следующую строку. Например, рисунок, состоящий из трех строк с двумя столбцами внутренних графиков, можно сформировать, добавляя требуемые объекты `Axes`:

```
In [x]: fig = plt.figure()
In [x]: ax1 = fig.add_subplot (321) # Верхний левый внутренний график.
In [x]: ax2 = fig.add_subplot (322) # Верхний правый внутренний график.
In [x]: ax3 = fig.add_subplot (323) # Средний левый внутренний график.
...
In [x]: ax6 = fig.add_subplot (326) # Нижний правый внутренний график.
```

Другой вариант: для создания рисунка и добавления в него всех внутренних графиков одновременно вызывается метод `plt.subplots`, принимающий аргументы `nrows` и `ncols` (в дополнение к аргументам, перечисленным в табл. 7.1) и возвращающий объект `Figure` и массив объектов `Axes`, который можно проиндексировать по каждой отдельной оси:

```
In [x]: fig, axes = plt.subplots(nrows=3, ncols=2)
In [x]: axes.shape
Out[x]: (3, 2)

In [x]: ax1 = axes[0, 0]      # Верхний левый внутренний график.
In [x]: ax2 = axes[2, 1]      # Нижний правый внутренний график.
```

В действительности полезным практическим приемом создания графика с одним объектом `Axes` является вызов метода `subplots()` без аргументов:

```
In [x]: fig, ax = plt.subplots()
In [x]: ax.plot(x, y)       # Не требуется индексирование одного созданного объекта Axes.
```

В рисунках с несколькими внутренними графиками существует риск наложения друг на друга надписей, заголовков и меток. Если это происходит, то необходимо вызвать метод `tight_layout`, и Matplotlib оптимизирует размещение внутренних графиков и обеспечит достаточное пространство между ними.

Пример П7.7. Рассмотрим металлический брусок с площадью поперечного сечения A, сначала при равномерно распределенной температуре θ_0, до которой происходит мгновенное нагревание точно в центре (поперечного сечения) при передаче определенного количества энергии *H*. Последующие значения температуры бруска (относительно θ_0), рассматриваемые как функция от времени *t* и координаты положения *x*, определяются уравнением одномерной диффузии:

$$\theta(x,t) = \frac{H}{c_p A}\frac{1}{\sqrt{Dt}}\frac{1}{\sqrt{4\pi}}\exp\left(-\frac{x^2}{4Dt}\right),$$

где c_p и D – удельная теплоемкость конкретного металла на единицу объема и коэффициент теплопроводности (которые предполагаются постоянными относительно температуры). Код в листинге 7.7 создает графики $\theta(x, t)$ для трех заданных интервалов времени и сравнивает графики по двум металлам с различными коэффициентами теплопроводности, но с одинаковой удельной теплоемкостью – это медь и железо.

Листинг 7.7. Уравнение одномерной диффузии, применяемое к температуре двух различных металлических брусков

```
# eg7-diffusion1d.py
import numpy as np
import matplotlib.pyplot as plt

# Площадь поперечного сечения бруска в м3, энергия, добавляемая в позиции x = 0 в Дж.
A, H = 1.e-4, 1.e3
# Температура в K при t = 0.
theta0 = 300

# Обозначение металла как химического элемента, удельная теплоемкость на единицу объема
# (Дж/м3.K).
# Коэффициенты теплопроводности (м2/с) для Cu и Fe.
metals = np.array([('Cu', 3.45e7, 1.11e-4), ('Fe', 3.50e7, 2.3e-5)],
                  dtype=[('symbol', '|S2'), ('cp', 'f8'), ('D', 'f8')])
```

```
# Металлический брусок расширяется от -xlim до xlim (м).
xlim, nx = 0.05, 1000
x = np.linspace(-xlim, xlim, nx)

# Вычисление распределения температур в этих трех интервалах времени.
times = (1e-2, 0.1, 1)
# Создание внутренних графиков: три строки для интервалов времени, по одному столбцу для
# каждого металла.
fig, axes = plt.subplots(nrows=3, ncols=2, figsize=(7, 8))
for j, t in enumerate(times):
    for i, metal in enumerate(metals):
        symbol, cp, D = metal
        ax = axes[j, i]
        # Решение уравнения диффузии.
        theta = theta0 + H/cp/A/np.sqrt(D*t * 4*np.pi) * np.exp(-x**2/4/D/t)
        # Создание графика с преобразованием расстояний в см и с добавлением надписей.
        ax.plot(x*100, theta , 'k')
        ax.set_title('{}, $t={}$ s'.format(symbol.decode('utf8'), t))
        ax.set_xlim(-4, 4)
        ax.set_xlabel('$x\;/\mathrm{cm}$')
        ax.set_ylabel('$\Theta\;/\mathrm{K}$')

# Настройка оси y, чтобы для каждого металла применялась одинаковая шкала при одинаковой t.
for j in (0, 1, 2):
    ymax = max(axes[j, 0].get_ylim()[1], axes[j, 1].get_ylim ()[1])
    print(axes[j, 0].get_ylim(), axes[j, 1].get_ylim())
    for i in (0, 1):
        ax = axes[j, i]
        ax.set_ylim(theta0, ymax)
        # Обеспечение вывода только трех штриховых меток по оси y.
        ax.set_yticks([theta0, (ymax + theta0)/2, ymax])

# Внутренние графики не должны перекрывать друг друга: это исправляет метод tight_layout().
fig.tight_layout()
plt.show()
```

Поскольку медь является лучшим проводником тепла, распространение повышения температуры наблюдается быстрее в этом металле (см. рис. 7.8).

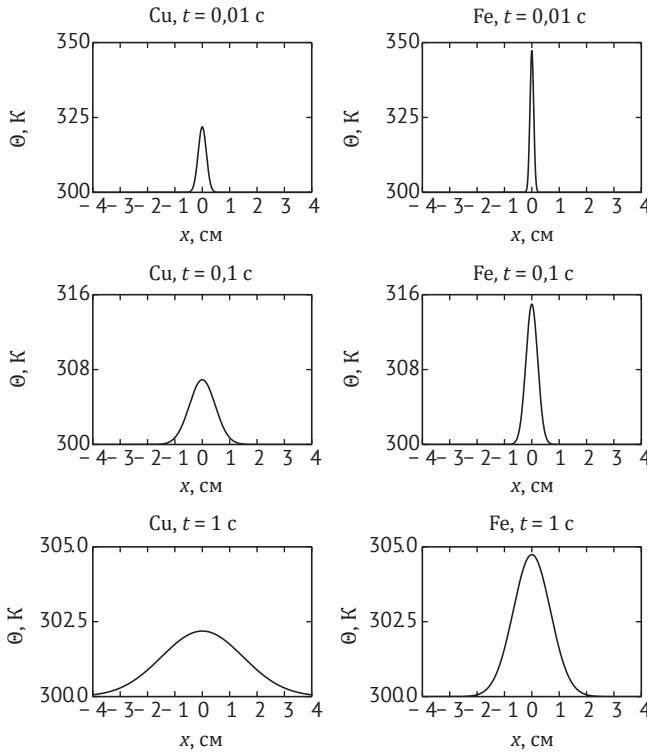


Рис. 7.8. Несколько решений уравнения одномерной диффузии для значений температуры в двух металлических брусках

Для более детальной настройки размещения внутренних графиков вызывается метод `fig.subplots_adjust()`. Этот метод принимает одно из ключевых слов `left`, `bottom`, `right`, `top`, `wspace` или `hspace`, для которых можно задать дробные значения высоты и ширины рисунка соответственно для определения положения границ внутренних графиков: левой стороны (по умолчанию 0.125), правой стороны (0.9), нижней границы (0.1), верхней границы (0.9), свободного пространства между графиками по вертикали (0.2) и по горизонтали (0.2). Практический пример использования этого метода – создание «склеенных» внутренних графиков, которые используют общую ось, как показано в примере П7.8.

Пример П7.8. Код в листинге 7.8 создает рисунок с 10 внутренними графиками, изображающими варианты функции $\sin(n\pi x)$ при $n = 0, 1, \ldots, 9$. Пространство между графиками сконфигурировано так, что они «переходят друг в друга» по вертикали (см. рис. 7.9).

Листинг 7.8. Десять внутренних графиков с нулевым промежуточным пространством по вертикали

```
import numpy as np
import matplotlib.pyplot as plt

nrows = 10
fig, axes = plt.subplots(nrows , 1)
# Нулевое промежуточное пространство по вертикали между внутренними графиками.
fig.subplots_adjust(hspace=0)

x = np.linspace(0, 1, 1000)

for i in range(nrows):
    # n = nrows для верхнего внутреннего графика, n = 0 для нижнего внутреннего графика.
    n = nrows - i
    axes[i].plot(x, np.sin(n * np.pi * x), 'k', lw=2)
    # Необходимы штриховые метки только на нижней границе каждого внутреннего графика.
    axes[i].xaxis.set_ticks_position('bottom')
    if i < nrows -1:
        # Настройка штриховых меток в узлах (нулевых) каждого варианта функции синуса.
        axes[i].set_xticks(np.arange(0, 1, 1/n))
        # Подписи к штриховым меткам нужны только на нижнем внутреннем графике по оси x.
        axes[i].set_xticklabels('')
    axes[i].set_yticklabels('')
plt.show()
```

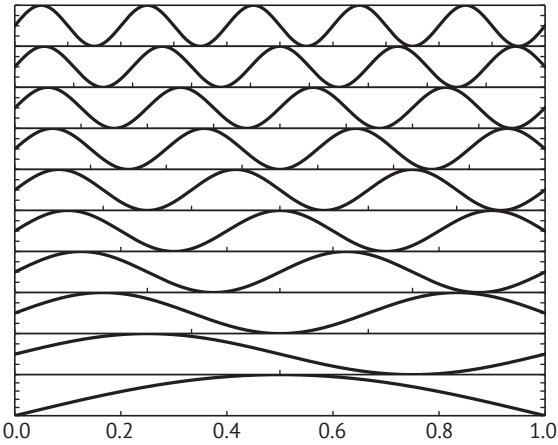


Рис. 7.9. Десять внутренних графиков функции $\sin(n\pi x)$ при $n = 0, 1, \ldots, 9$ с удаленным промежуточным свободным пространством между ними

7.2.8 Сохранение рисунков

Сохранение рисунков для вывода на печать

Как было отмечено в разделе 7.1.1, размер рисунка Matplotlib можно устанавливать (в дюймах) с помощью аргумента `figsize` для метода `plt.figure()`. Для линейных графиков высокого качества (в отличие от нерегулярных изображений, тепловых карт и т. п.) сохранение в файле векторного формата позволяет получить наилучшее представление при печати, которое можно масштабировать без потерь качества. Удачный вариант выбора в общем случае – формат встроенного PostScript (EPS) или PDF:

```
# Рисунок шириной 6" и высотой 4".
fig = plt.figure(figsize=(6, 4))
ax = fig.add_subplot()

# Вывод рисунка в файл.

plt.savefig('my-figure.eps')
```

Для рисунков, которые невозможно эффективно векторизовать (например, нерегулярные изображения), необходим растровый формат. В этом случае для управления разрешением печатной версии существует дополнительный аргумент `dpi` (dots per inch – количество точек на дюйм), который обязательно должен быть определен как для метода `plt.figure()`, так и для метода `plt.savefig()`. Из подходящих форматов можно назвать JPG и PNG. Например, рисунок с качеством, достаточным для публикации в журнальной статье, можно создать следующим образом:

```
DPI = 300          # Минимального разрешения 300 dpi обычно достаточно.
fig = plt.figure(figsize=(3.5, 3), dpi=DPI)
ax = fig.add_subplot()

# Вывод рисунка в файл.

plt.savefig('my-figure.png', dpi=DPI)
```

Сохранение рисунков для использования в режиме онлайн

В первую очередь следует отметить, что при различных разрешениях (количестве пикселов на единицу измерения) разнообразных моделей мониторов не существует способа однозначного определения физических размеров изображения при выводе его на экран пользователя. Но есть возможность создания графического файла рисунка с заданными размерами в пикселах (часто это все, что требуется для веб-страниц). Это достаточно сложно сделать: поскольку в `figsize` ожидаются размеры в дюймах, приходится подбирать подходящее значение DPI и вычислять эти размеры по требуемой ширине и высоте в пикселах:

```
DPI = 100              # Это разумный вариант выбора.
WIDTH, HEIGHT = 800, 800
fig = plt.figure(figsize=(WIDTH / DPI, HEIGHT / DPI), dpi=DPI)

# Вывод рисунка в файл.

plt.savefig('my-figure.png', dpi=DPI)    # Важно: здесь еще раз определяется DPI.
```

7.3 Столбиковые диаграммы, круговые диаграммы и диаграммы в полярных координатах

7.3.1 Столбиковые диаграммы и гистограммы

Основной функцией модуля `pyplot` для создания столбиковой диаграммы является `ax.bar`, которая формирует график из прямоугольных полос, определяемых их левой границей и высотой. Например:

```
ax.bar([0, 1, 2], [40, 80, 20])
```

По умолчанию задано значение ширины прямоугольников `0.8`, но это значение можно изменить с помощью (третьего) аргумента `width`. Если необходимо отцентрировать прямоугольники по вертикали, то для аргумента `align` устанавливается значение `'center'` или выполняется вычисление требуемого положения левых границ:

```
w = 0.5
x, y = np.array([0, 1, 2]), np.array([40, 80, 20])
ax.bar(x, y, w, align='center')   # Самый простой способ центрирования столбцов по вертикали.
ax.bar(x - w/2, y, w)             # Или вычисление положения их левых границ.
```

Дополнительные аргументы, в том числе аргументы, определяющие диаграммы погрешностей, описаны в табл. 7.9.

Таблица 7.9. Аргументы методов ax.bar и ax.barh

Аргумент	Описание
`left`	Последовательность координат *x* левых границ столбцов (но см. также `align`)
`width`	Ширина столбцов. Если задано скалярное значение, то ширина всех столбцов одинакова. Можно передать массив с разными значениями ширины столбцов
`bottom`	Координаты y нижних границ столбцов
`height`	Последовательность значений высоты столбцов
`color`	Цвет заливки столбцов (скалярное значение или массив)
`edgecolor`	Цвет границ столбцов (скалярное значение или массив)
`linewidth`	Значения толщины линий границ столбцов в пт (скалярное значение или массив)
`xerr, yerr`	Предельные значения для диаграмм погрешностей, как для метода `errorbar` (скалярное значение или массив)
`error_kw`	Словарь именованных аргументов, соответствующих специализированным параметрам настройки внешнего вида диаграмм погрешностей (см. табл. 7.8)
`align`	По умолчанию значение `'edge'` определяет выравнивание столбцов по их левым границам (для вертикальных столбцов) или по нижним границам (для горизонтальных столбцов). Значение `'center'` центрирует столбцы по их осям
`log`	Если установлено значение `True`, то используется ось с логарифмической шкалой
`orientation`	`'vertical'` (по умолчанию) или `'horizontal'`
`hatch`	Определяет тип штриховки столбцов: один из символов `'/'`, `'\'`, `'|'`, `'-'`, `'+'`, `'x'`, `'o'`, `'O'`, `'.'`, `'*'`. Повторение символа дает более плотную штриховку

По умолчанию метод `ax.bar` создает диаграмму с вертикальными столбцами. Диаграмма с горизонтальными столбцами формируется при установке значения `orientation='horizontal'` или при использовании аналогичного метода `ax.barh`.

Пример П7.9. Программа в листинге 7.9 создает столбиковую диаграмму частоты встречаемости букв английского алфавита по результатам вычисления оценки при анализе текста романа Германа Мелвилла «Моби Дик» (Moby-Dick)[101]. Вертикальные столбцы отцентрированы и помечены соответствующей буквой (см. рис. 7.10).

Листинг 7.9. Частота встречаемости букв в тексте романа «Моби Дик»

```
# eg7-charfreq.py
import numpy as np
import matplotlib.pyplot as plt

text_file = 'moby-dick.txt'

letters = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
# Инициализация словаря счетчиков букв: { ' A ': 0, ' B ': 0, ...}.
lcount = dict([(letter, 0) for letter in letters])

# Считывание текста и подсчет встречающихся букв.
for letter in open(text_file).read():
    try:
        lcount[letter.upper()] += 1
    except KeyError:
        # Игнорировать символы, не являющиеся буквами.
        pass

# Общее количество букв.
norm = sum(lcount.values())

fig, ax = plt.subplots()
# Столбиковая диаграмма с расположением букв по горизонтальной оси и вычисленной частотой
# встречаемости букв в процентах, определяющей высоту каждого столбца.
x = range(26)
ax.bar(x, [lcount[letter]/norm * 100 for letter in letters], width=0.8,
       color='g', alpha=0.5, align='center')
ax.set_xticks(x)
ax.set_xticklabels(letters)
ax.tick_params(axis='x', direction='out')
ax.set_xlim(-0.5, 25.5)
ax.yaxis.grid(True)
ax.set_ylabel('Letter frequency , %')
plt.show()
```

[101] Свободно распространяемый файл с текстом этого романа можно скачать, например, с сайта проекта Гутенберга www.gutenberg.org/ebooks/2701.

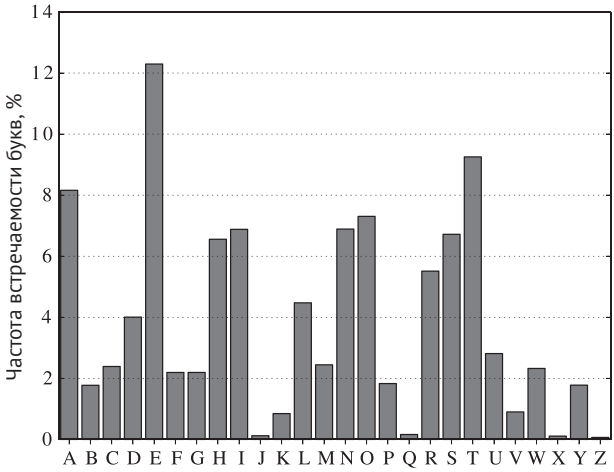


Рис. 7.10. Частота встречаемости букв в тексте романа «Моби Дик»

Для одноцветных графиков иногда полезно выделять столбцы различными типами штриховки. Для этого можно воспользоваться аргументом `hatch`, в котором определяется любой предварительно определенный шаблон штриховки (см. табл. 7.9), как показано в примере П7.10.

Пример П7.10. Файл *germany-energy-sources.txt*, который можно скачать по адресу https://scipython.com/eg/bae, содержит данные о возобновляемых источниках электрической энергии, используемых в Германии с 1990 по 2018 г.:

```
Renewable electricity generation in Germany in GWh (million kWh)
Year      Hydro      Wind    Biomass  Photovoltaics
2018      17974    109951      50851          45784
2017      20150    105693      50917          39401
2016      20546     79924      50928          38098
…
```

Программа в листинге 7.10 отображает эти данные в виде составной столбиковой диаграммы, используя шаблоны штриховки Matplotlib для обозначения различных источников энергии (см. рис. 7.11).

Листинг 7.10. Визуализация производства электроэнергии из возобновляемых источников в Германии

```
# eg7-germany -alt-energy.py
import numpy as np
import matplotlib.pyplot as plt

data = np.loadtxt('germany-energy-sources.txt', skiprows=2, dtype='f8')
years = data[:, 0]
n = len(years)

# Преобразование ГВт в ТВт.
data[:, 1:] /= 1000
```

```
fig, ax = plt.subplots()
sources = ('Hydroelectric', 'Wind', 'Biomass', 'Photovoltaics')
hatch = ['oo', '', 'xxxx', '//']
bottom = np.zeros(n)
bars = [None]*n
for i, source in enumerate(sources):
    bars[i] = ax.bar(years, bottom=bottom, height=data[:, i+1], color='w',          ❶
                     hatch=hatch[i], align='center', edgecolor='k')
    bottom += data[:, i+1]

ax.set_xticks(years[::2])    # Для удобства чтения помечен каждый четный год.
plt.xticks(rotation=90)
ax.set_xlim(1989, 2019)
ax.set_ylabel('Renewable Electricity (TWh)')
ax.set_title('Renewable Electricity Generation in Germany , 1990-2018')
plt.legend(bars, sources, loc='best')                                           ❷
plt.show()
```

❶ Для включения в график легенды каждый объект столбиковой диаграммы[102] обязательно должен быть сохранен в списке `bars`, который

❷ передается в метод `ax.legend` с соответствующей последовательностью надписей `sources`.

Рис. 7.11. Составная столбиковая диаграмма производства электроэнергии из возобновляемых источников в Германии, 1990–2018 гг.

[102] В действительности это Container «художников».

7.3.2 Круговые диаграммы

В Matplotlib круговую (секторную) диаграмму можно построить очень просто – нужно передать массив значений в метод `ax.pie`. Значения нормализуются по их сумме, если эта сумма больше 1, иначе значения напрямую интерпретируются как доли. Надписи, проценты, вынесенные сегменты и прочие эффекты определяются аргументами, описанными в табл. 7.10, и демонстрируются в примере П7.11.

Таблица 7.10. Аргументы для метода ax.pie

Аргумент	Описание
colors	Последовательность спецификаторов цвета Matplotlib для заливки сегментов
labels	Последовательность строк надписей для сегментов
explode	Последовательность значений, определяющих дробную часть радиуса круговой диаграммы, на которую смещается каждый клинообразный сегмент (0 для отсутствия эффекта выноса сегмента)
shadow	True или False: определяет изображение или отсутствие декоративной тени под круговой диаграммой
startangle	Определяет поворот «начальной позиции» круговой диаграммы на заданное число градусов против часовой стрелки относительно горизонтальной оси
autopct	Строка формата для подписей к сегментам: соответствующие значения в процентах или функция, генерирующая требуемую строку из данных
pctdistance	Радиальное положение текста подписи autopct относительно радиуса круговой диаграммы. По умолчанию 0.6 (т. е. внутри диаграммы, но это может оказаться неприемлемым для слишком узких сегментов)
labeldistance	Радиальное положение текста надписи label относительно радиуса круговой диаграммы. По умолчанию 1.1 (снаружи, рядом с границей круговой диаграммы)
radius	Радиус круговой диаграммы (по умолчанию 1). Этот параметр полезен при создании перекрывающихся круговых диаграмм с различными радиусами

Пример П7.11. Программа в листинге 7.11 графически изображает данные о выбросе в атмосферу парниковых газов с учетом массы в «углеродном эквиваленте» (данные взяты из отчета IPCC 2007 г.)[103].

Листинг 7.11. Круговая диаграмма данных о выбросе в атмосферу парниковых газов

```
# eg7-pie.py
import numpy as np
import matplotlib.pyplot as plt

# Выброс в атмосферу парниковых газов за год в млрд т углеродного эквивалента (GtCe).
gas_emissions = np.array([(r'$\mathrm{CO_2}$-d', 2.2),
                          (r'$\mathrm{CO_2}$-f', 8.0),
                          ('Nitrous\nOxide', 1.0),
                          ('Methane', 2.3),
                          ('Halocarbons', 0.1)],
                         dtype=[('source', 'U17'), ('emission', 'f4')])
```

[103] IPCC, Climate Change 2007: Synthesis Report. Contribution of Working Groups I, II and III to the Fourth Assessment Report of the Intergovernmental Panel on Climate Change [CoreWriting Team, Pachauri, R. K and Reisinger, A. (eds.)]. Geneva, Switzerland (2007).

```
# Пять оттенков бежевого цвета.
colors = ['#C7B299', '#A67C52', '#C69C6E', '#754C24', '#534741']

explode = [0, 0, 0.1, 0, 0]                                                    ❶

fig, ax = plt.subplots()
ax.axis('equal')        # Чтобы диаграмма выглядела круглой.
ax.pie(gas_emissions['emission'], colors=colors , shadow=True , startangle=90,
       explode=explode , labels=gas_emissions['source'], autopct='%.1f%%',     ❷
       pctdistance=1.15, labeldistance =1.3)

plt.show()
```

❶ Сегмент, соответствующий оксиду азота, вынесен на расстояние 10 % (от радиуса).

❷ Значения в процентах форматируются с одним десятичным знакоместом после десятичной точки (`autopct='%.1f%%'`).

Полученная в результате круговая диаграмма показана на рис. 7.12.

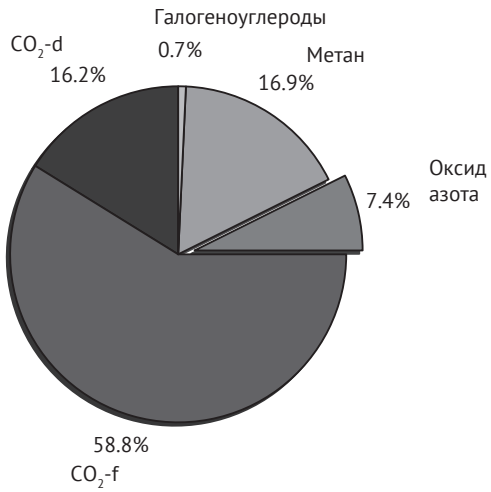


Рис. 7.12. Выброс в атмосферу парниковых газов в процентах по пяти различным источникам: CO_2-d обозначает выброс CO_2 из-за уничтожения лесов; CO_2-f обозначает выброс CO2 при сжигании природного углеводородного топлива

В последние годы круговые диаграммы стали менее распространенными (их презрительно называют «Comic Sans визуализации данных»), несомненно, разумнее избегать их использования при сравнении большого количества категорий или весьма похожих значений. К счастью, Matplotlib не поддерживает трехмерные круговые диаграммы.

7.3.3 Диаграммы в полярных координатах

Диаграмма в полярных координатах с радиусом *r*, определяемая как функция от угла θ, создается с помощью метода `pyplot.polar`, как показано в разделе 3.3.1, или посредством определения представления по умолчанию при добавлении внутреннего графика в рисунок:

```
fig = plt.figure()
ax = fig.add_subplot(projection='polar')
```

В примере П7.12 демонстрируются оба способа.

Пример П7.12. Система направленных антенн (антенная решетка) может использоваться для ориентации радиоволн в определенном направлении с помощью регулирования их количества, геометрической конфигурации, относительных амплитуд и относительных фаз[104]. Рассмотрим систему из *n* изотропных антенн (с равномерным излучением во всех направлениях) в позициях $\boldsymbol{d}_j$, равномерно распределенных с интервалом *d* по оси *x* от начала координат:

$$d_0 = 0, d_1 = d\hat{x}, \ldots, d_{n-1} = (n-1)d\hat{x}.$$

Если одна антенна генерирует вектор излучения $\boldsymbol{F}(\boldsymbol{k})$, где $\boldsymbol{k} = k\hat{r} = (2\pi/\lambda)\hat{r}$, то суммарный вектор излучения всех *n* антенн равен

$$F_{tot}(k) = \sum_{j=0}^{n-1} w_j e^{j i \boldsymbol{k}\cdot\boldsymbol{d}_j} F(k) = A(k)F(k),$$

где w_j – относительная комплексная амплитуда возбуждения *j*-й антенны, представляющая ее амплитуду и фазу, а множитель $A(\boldsymbol{k})$ известен как множитель (коэффициент) решетки антенн. Можно выбрать $w_0 = 1$, чтобы определить относительные комплексные амплитуды возбуждения для каждой антенны в начальный момент времени. Далее выбирается вариант рассмотрения только по азимутальному (φ) распространению сигнала для излучения в плоскости *xy* с установкой угла в полярных координатах θ = π/2. В этом случае получаем:

$$A(\phi) = \sum_{j=0}^{n-1} w_j e^{jikd\cos\phi}.$$

Типовая относительная мощность излучения (коэффициент усиления) равна квадрату этой величины. Для двух одинаковых антенн:

$$g(\varphi) = |A(\varphi)|^2 = |w_0 + w_1 e^{ikd\cos\varphi}|^2.$$

В коде из листинга 7.12 относительная величина – коэффициент направленного действия (КНД) $10\log_{10}(g/g_{\max})$ – изображена графически на рис. 7.13 как функция от φ для случая двух антенн на графике в полярных координатах при $d = \lambda$ и $w_0 = 1, w_1 = -i$.

---

[104] *J. Orfanidis*. Electromagnetic Waves and Antennas, Rutgers University, http://eceweb1.rutgers.edu/~orfanidi/ewa/.

Листинг 7.12. График коэффициента направленного действия (КНД) для системы из двух антенн

```
import numpy as np
import matplotlib.pyplot as plt

def gain(d, w):
    """Return the power as a function of azimuthal angle, phi."""
    # """Возвращает мощность как функцию от азимутального угла phi."""
    phi = np.linspace(0, 2*np.pi, 1000)
    psi = 2*np.pi * d / lam * np.cos(phi)
    A = w[0] + w[1]*np.exp(1j*psi)
    g = np.abs(A)**2
    return phi, g

def get_directive_gain(g, minDdBi=-20):
    """Return the "directive gain" of the antenna array producing gain g."""
    # """Возвращает КНД системы направленных антенн, генерирующей сигнал g."""
    DdBi = 10 * np.log10(g / np.max(g))
    return np.clip(DdBi, minDdBi, None)                                    ❶

# Длина волны, интервал между антеннами, относительные комплексные амплитуды возбуждения.
lam = 1
d = lam
w = np.array([1, -1j])
# Вычисление сигнала и направленного сигнала; график в полярных координатах.
phi, g = gain(d, w)
DdBi = get_directive_gain(g)

plt.polar(phi, DdBi)
ax = plt.gca()                                                             ❷
ax.set_rticks([-20, -15, -10, -5])                                         ❸
ax.set_rlabel_position (45)                                                ❹
plt.show()
```

❶ Чтобы наилучшим образом отобразить самую интересную область графика с наивысшей мощностью, «отсекаются» значения, меньшие значения `minDdBi`.

❷ Для детальной настройки графика необходим объект `Axes` в контексте этого конкретного графика. Объект `Axes` возвращается методом `plt.gca()` (gca – get current axes – получить текущие оси).

❸ Метод `set_rticks` определяет позиции штриховых меток на радиальных осях полярных координат.

❹ Метод `set_rlabel_position` определяет угловые позиции меток на круговых осях полярных координат.

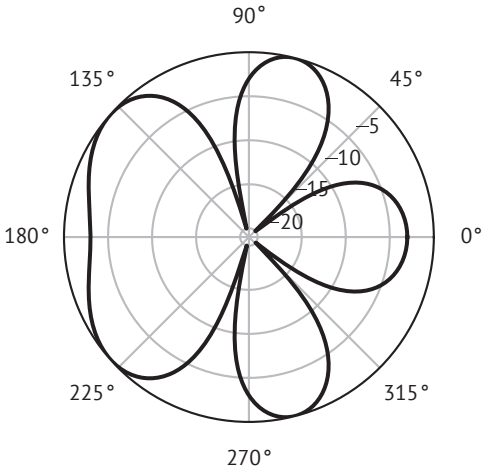


Рис. 7.13. Коэффициент направленного действия (КНД) системы из двух антенн при $d = \lambda$, $w_0 = 1, w_1 = -i$

Методы бродкастинга библиотеки NumPy (см. раздел 6.1.7) предоставляют более естественный способ расширения функциональности кода из листинга 7.12 для произвольного количества антенн. В следующем примере (листинг 7.13) метод объекта `Figure` `add_subplot` вызывается с аргументом `projection='polar'` и возвращает соответствующий объект `Axes` `ax`.

Листинг 7.13. График коэффициента направленного действия (КНД) для системы из трех антенн

```
import numpy as np
import matplotlib.pyplot as plt

def gain(d, w):
    """Return the power as a function of azimuthal angle, phi."""
    # """Возвращает мощность как функцию от азимутального угла phi."""
    phi = np.linspace(0, 2*np.pi, 1000)
    psi = 2*np.pi * d / lam * np.cos(phi)
    j = np.arange(len(w))
    A = np.sum(w[j] * np.exp(j * 1j * psi[:, None]), axis=1)                    ❶
    g = np.abs(A)**2
    return phi, g

def get_directive_gain(g, minDdBi=-20):
    """Return the "directive gain" of the antenna array producing gain g."""
    # """Возвращает КНД системы направленных антенн, генерирующей сигнал g."""
    DdBi = 10 * np.log10(g / np.max(g))
    return np.clip(DdBi, minDdBi, None)

# Длина волны, интервал между антеннами, относительные комплексные амплитуды возбуждения.
lam = 1
d = lam / 2
w = np.array([1, -1, 1])
```

```
# Вычисление сигнала и направленного сигнала; график в полярных координатах.
phi, g = gain(d, w)
DdBi = get_directive_gain(g)

fig = plt.figure()
ax = fig.add_subplot(projection='polar')
ax.plot(phi, DdBi)                                                    ❷
ax.set_rticks([-20, -15, -10, -5])
ax.set_rlabel_position(45)
plt.show()
```

❶ Здесь вычисляется сумма всех членов формулы коэффициентов (множителей) решетки антенн: при добавлении оси в массив `psi` эта сумма вычисляется для каждой угловой позиции φ.

❷ Обратите внимание: поскольку тип представления графика уже определен ранее, здесь необходим метод `ax.plot`, а не метод `ax.polar`.

Полученный в результате график показан на рис. 7.14.

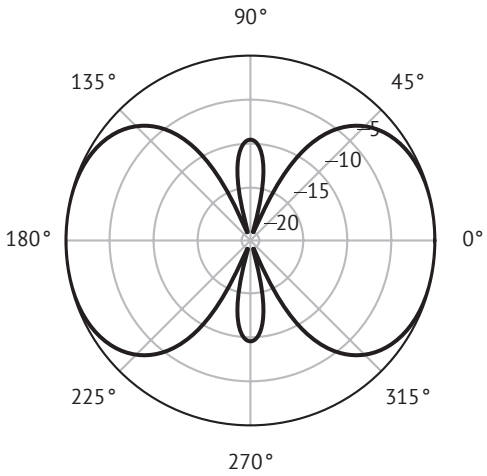


Рис. 7.14. Коэффициент направленного действия (КНД) системы из нескольких (трех) антенн при $d = \lambda/2$, $w_0 = 1$, $w_1 = -1$, $w_2 = 1$

7.4 Аннотации для графиков

Matplotlib предоставляет несколько способов для добавления разнообразных типов аннотаций в графики. В следующих разделах описаны наиболее важные методы добавления текста, стрелок, линий и фигур.

7.4.1 Добавление текста

Метод `ax.text(x, y, s)` является основным инструментом, используемым для добавления строки текста `s` в позиции `(x, y)` (в координатах данных) в конкретном объекте оси. Свойства шрифта можно определить, дополнительно передавая словарь пар (*`keyword, value`*) в аргументе `fontdict` (см. табл. 7.6).

Отдельные именованные аргументы (например, `fontsize=20`) также можно использовать для детальной настройки шрифта таким способом.

Если текстовая аннотация ссылается на какое-либо свойство данных, то обычно требуется поведение по умолчанию, при котором при размещении используются координаты данных, так что обеспечивается одинаковая позиция относительно данных, даже если границы графика изменяются. Но если требуется разместить текст в координатах осей, т. е. (0, 0) – нижний левый угол системы координатных осей, а (1, 1) – верхний правый угол, то передается именованный аргумент `transform=ax.transAxes`, где `ax` – объект `Axes`, в котором определяются координаты.

7.4.2 Стрелки и текст

Метод `ax.annotate` похож на метод `ax.text` (но с совершенно отличающимся синтаксисом, что причиняет большие неудобства), но изображает стрелку Matplotlib от текста к заданной точке на графике. Важные аргументы для метода `ax.annotate`:

- `s` – строка, выводимая как текстовая надпись;
- `xy` – кортеж `(x, y)`, определяющий координаты положения аннотации (т. е. куда будет направлена стрелка);
- `xytext` – кортеж `(x, y)`, определяющий координаты текстовой надписи (т. е. где начинается стрелка);
- `xycoords` – необязательная строка, определяющая тип координат, передаваемых в аргументе `xy`: доступно несколько возможных вариантов[105], но чаще всего используются следующие:
  - `'data'` – координаты данных, по умолчанию;
  - `'figure fraction'` – дробные координаты по отношению к размеру рисунка: `(0, 0)` – нижний левый угол, `(1, 1)` – верхний правый угол;
  - `'axes fraction'` – дробные координаты по отношению к осям: `(0, 0)` – нижний левый угол, `(1, 1)` – верхний правый угол;
- `textcoords` – то же, что для `xycoords`, необязательная строка, определяющая тип координат, передаваемых в аргументе `xytext`. Для этой строки допускается дополнительное значение: `'offset points'` определяет, что кортеж `xytext` представляет смещение в пт от позиции `xy`;
- `arrowprops` – если передается, то определяет свойства и стиль стрелки, изображаемой между позициями `xytext` и `xy` (см. ниже).

Дополнительные именованные аргументы интерпретируются как свойства объекта `Text`, выводимого как надпись (например, `fontsize` и `color`). Кроме того, важна пара аргументов `verticalalignment` (или `va`) и `horizontalalignment` (или `ha`), которая определяет, как надпись выравнивается относительно позиции `xytext`. Допустимые значения: `'center'`, `'right'`, `'left'`, `'top'`, `'bottom'` и `'baseline'` соответственно.

В простейшем варианте применения `ax.annotate` просто добавляет текстовую надпись в график (без стрелки). Например:

```
ax.annotate('My Label', xy=(0.5, 0.8), fontsize=16, xycoords='axes fraction', ha='center')
```

[105] См. документацию: https://matplotlib.org/api/text_api.html.

Эта команда добавляет надпись `'My Label'` в центре около верхней границы осей с размером текста 16 пт. Обратите внимание: поскольку здесь нет стрелки или какой-либо другой линии, аргумент `xytext` не нужен, а надпись размещается прямо в позиции `xy`.

Аргумент `arrowprops` представляет собой словарь, определяющий стиль линии или стрелки, соединяющей надпись в позиции `xytext` с заданной точкой `xy`. Существует слегка запутанный набор возможных элементов, которые можно поместить в этот словарь, но использование наиболее важных элементов продемонстрировано ниже в примере П7.13.

Пример П7.13. Программа в листинге 7.14 создает график с восемью стрелками с различными стилями (см. рис. 7.15).

Листинг 7.14. Аннотации со стрелками в Matplotlib

```
# eg7-arrows.py
import numpy as np
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
x = np.linspace(0, 1)
ax.plot(x, x, 'o')

ax.annotate('default line', xy=(0.15, 0.1), xytext=(0.6, 0.1),
            arrowprops={'arrowstyle': '-'}, va='center')
ax.annotate('dashed line', xy=(0.25, 0.2), xytext=(0.6, 0.2),
            arrowprops={'arrowstyle': '-', 'ls': 'dashed'}, va='center')
ax.annotate('default arrow', xy=(0.35, 0.3), xytext=(0.6, 0.3),
            arrowprops={'arrowstyle': '->'}, va='center')
ax.annotate('thick blue arrow', xy=(0.45, 0.4), xytext=(0.6, 0.4),
            arrowprops={'arrowstyle': '->', 'lw': 4, 'color': 'blue'},
            va='center')
ax.annotate('double -headed arrow', xy=(0.45, 0.5), xytext=(0.01, 0.5),
            arrowprops={'arrowstyle': '<->'}, va='center')
ax.annotate('arrow with closed head', xy=(0.55, 0.6), xytext=(0.1, 0.6),
            arrowprops={'arrowstyle': '-|>'}, va='center')
ax.annotate('a really thick red arrow\nwith not much space', xy=(0.65, 0.7),
            xytext=(0.1, 0.7), va='center', multialignment='right',
            arrowprops={'arrowstyle': '-|>', 'lw': 8, 'ec': 'r'})
ax.annotate('a really thick red arrow\nwith space between\nthe tail and the'
            'label', xy=(0.85, 0.9), xytext=(0.1, 0.9), va='center',
            multialignment='right',
            arrowprops={'arrowstyle': '-|>', 'lw': 8, 'ec': 'r', 'shrinkA': 10})

plt.show()
```

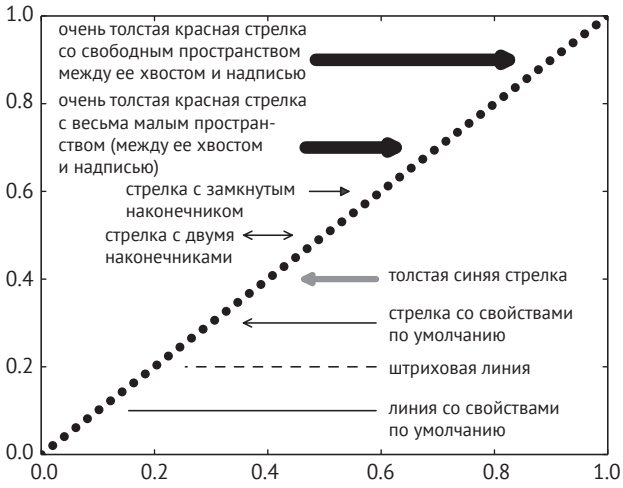


Рис. 7.15. Пример использования различных стилей стрелок

Пример П7.14. Еще один пример графика с аннотацией – курс акций BP plc (компании British Petroleum) (LSE: BP) с добавлением пары значительных событий. Необходимые данные для этого примера можно скачать с сайта Yahoo! Finance: https://uk.finance.yahoo.com/q/hp?s=BP.L.

Листинг 7.15. Изображение временно́й последовательности курса акций на графике с аннотацией

```
import datetime
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.dates import strpdate2num
from datetime import datetime

def date_to_int(s):                                                    ❶
    epoch = datetime(year=1970, month=1, day=1)
    date = datetime.strptime(s, '%Y-%m-%d')
    return (date - epoch).days

def bindate_to_int(bs):
    return date_to_int(bs.decode('ascii'))

dt = np.dtype([('daynum','i8'), ('close', 'f8')])
share_price = np.loadtxt('bp-share -prices.csv', skiprows=1, delimiter=',',
                         usecols=(0, 4), converters={0: bindate_to_int}, dtype=dt)
fig, ax = plt.subplots()
ax.plot(share_price['daynum'], share_price['close'], c='g')
ax.fill_between(share_price['daynum'], 0, share_price['close'], facecolor='g',   ❷
                alpha=0.5)

daymin, daymax = share_price['daynum'].min(), share_price['daynum'].max()
ax.set_xlim(daymin, daymax)

price_max = share_price['close'].max()
```

```
def get_xy(date):
    """ Return the (x, y) coordinates of the share price on a given date. """
    # """Возвращает координаты (x, y) курса акций в конкретную дату. """
    x = date_to_int(date)
    return share_price[np.where(share_price['daynum']==x)][0]

# Горизонтальная стрелка и надпись.
x, y = get_xy('1999-10-01')
ax.annotate('Share split', (x, y), xytext = (x+1000, y), va='center',
            arrowprops=dict(facecolor='black', shrink=0.05))
# Вертикальная стрелка и надпись.
x, y = get_xy('2010-04-20')
ax.annotate('Deepwater Horizon\noil spill', (x, y), xytext = (x, price_max*0.9),
            arrowprops=dict(facecolor='black', shrink=0.05), ha='center')

years = range(1989, 2015, 2)

ax.set_xticks([date_to_int('{:4d}-01-01'.format(year)) for year in years])
ax.set_xticklabels(years , rotation=90)                                    ❸

plt.show()
```

❶ Необходимы дополнительные действия для считывания столбца данных: сначала декодируется считываемая из файла строка байтов в кодировку ASCII (`bindate_to_int`), затем используется метод `datetime` (см. раздел 4.5.3) для преобразования в целое число дней, прошедших с некоторой контрольной даты (эпохи): здесь выбрано использование эпохи Unix 1 января 1970 г. (`date_to_int`).

❷ Метод `ax.fill_between` выполняет заливку одним цветом области под линией графика.

❸ Подписи для меток года поворачиваются (ориентируются по вертикали), чтобы обеспечить достаточное место для их размещения (читаются снизу вверх).

На рис. 7.16 показан итоговый график.

Рис. 7.16. Курс акций компании BP plc на графике с аннотациями

7.4.3 Линии и перекрывающие прямоугольники

Добавить произвольную прямую линию в график Matplotlib можно простой передачей данных, соответствующих начальной и конечной точкам этой линии в метод `ax.plot`, например:

```
ax.plot([x1, x2], [y1, y2], color='k', lw=2)
```

Эта команда изображает отрезок прямой между точками `(x1, y1)` и `(x2, y2)`. Разумеется, такой способ становится утомительным, если требуется изобразить много не связанных друг с другом линий, но для горизонтальных и вертикальных линий существует пара удобных методов `ax.hlines` и `ax.vlines`. Метод `ax.hlines` принимает обязательные аргументы `y`, `xmin`, `xmax` и изображает горизонтальные линии с координатами `y` для каждого значения из последовательности `y` (если значение `y` передано как скаляр, то изображается одна линия). Аргументы `xmin` и `xmax` определяют начало и конец каждой линии, они могут быть скалярными значениями (в этом случае все линии имеют одинаковые начальные и конечные координаты `x`) или последовательностью (отдельных значений для каждой координаты, заданной в аргументе `y`). Метод `ax.vlines` изображает вертикальные линии, а его обязательные аргументы `x`, `ymin`, `ymax` аналогичны по смыслу аргументам метода `ax.xlines`.

Пример П7.17. Код в листинге 7.16 демонстрирует некоторые варианты применения методов `ax.vlines` и `ax.hlines` (см. рис. 7.17).

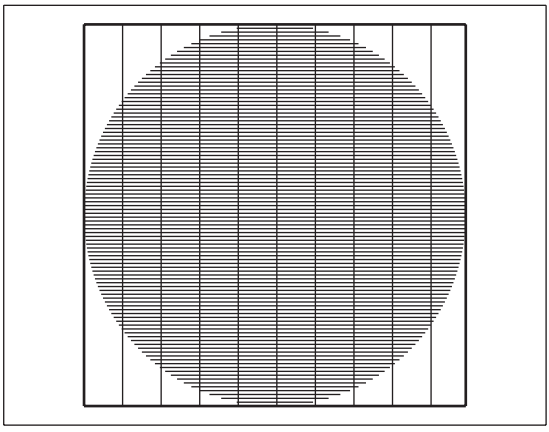


Рис. 7.17. Фигура, созданная из вертикальных и горизонтальных линий

Листинг 7.16. Некоторые варианты применения методов ax.vlines и ax.hlines

```
# eg7-circle -lines.py
import numpy as np
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
```

```
ax.axis('equal')

# Круг, сформированный из горизонтальных линий.
y = np.linspace(-1, 1, 100)
xmax = np.sqrt(1 - y**2)
ax.hlines(y, -xmax, xmax, color='g')

# Изображение квадрата из более толстых линий, окаймляющего круг.
ax.vlines(-1, -1, 1, lw=2, color='r')
ax.vlines(1, -1, 1, lw=2, color='r')
ax.hlines(-1, -1, 1, lw=2, color='r')
ax.hlines(1, -1, 1, lw=2, color='r')
# Несколько равномерно распределенных вертикальных линий.
ax.vlines(y[::10], -1, 1, color='b')

# Удаление штриховых меток на осях и надписей.
ax.xaxis.set_visible(False)
ax.yaxis.set_visible(False)
# Небольшое выравнивание снаружи квадратной области.
ax.set_xlim(-1.1, 1.1)
ax.set_ylim(-1.1, 1.1)

plt.show()
```

На статических графиках, таких как иллюстрации, предназначенные для вывода на печать, методы `ax.hlines` и `ax.vlines` работают успешно, но следует отметить, что ограничения линий не изменяются при изменении ограничений осей в интерактивном графике. Существуют еще два метода `ax.axhline` и `ax.axvline`, которые просто изображают горизонтальную или вертикальную линию по соответствующей оси при любых текущих ограничениях. Метод `ax.axhline` принимает аргументы `y`, `xmin`, `xmax`, но они обязательно должны быть скалярными значениями (поэтому для создания нескольких линий требуются повторные вызовы), а `xmin`, `xmax` определяются в дробных координатах, так что 0 представляет левую границу графика, а 1 – его правую границу. Аналогично определяются аргументы `x`, `ymin`, `ymax` метода `ax.axvline`. Ниже приведено несколько примеров:

```
ax.axhline(100, 0, 1)   # Горизонтальная линия параллельно всей оси x с координатой y =
100.
ax.axhline(100)         # То же самое: xmin и xmax по умолчанию равны 0 и 1.
# Толстая, синяя, штриховая вертикальная линия с координатой x = 5 относительно центра оси y.
ax.axvline(5, 0.4, 0.6, c='b', lw=4, ls='--')
```

Методы `ax.axhspan` и `ax.axvspan` аналогичны описанным выше, но создают горизонтальный или вертикальный перекрывающий прямоугольник, расположенный вдоль соответствующей оси. В метод `ax.axhspan` передаются аргументы `ymin`, `ymax` (в координатах данных) и `xmin`, `xmax` (в дробных единицах измерения осей). Аналогичным образом метод `ax.axvspan` принимает аргументы `xmin`, `xmax`, `ymin`, `ymax`. Дополнительные именованные аргументы можно использовать для определения стиля перекрывающего прямоугольника (который принадлежит к типу объекта `Patch`, см. табл. 7.11).

Таблица 7.11. Именованные аргументы для определения стиля перекрывающих прямоугольников

Аргумент	Описание
`alpha`	Настройка прозрачности в альфа-канале (0–1)
`color`	Настройка обоих цветов facecolor и edgecolor прямоугольника
`edgecolor, ec`	Настройка цвета границы прямоугольника
`facecolor, fc`	Настройка цвета заливки прямоугольника
`fill`	Определяет, должен ли закрашиваться прямоугольник (True или False)
`hatch`	Настройка типа штриховки прямоугольника – один из символов: '/', '\', '|', '-', '+', 'x', 'o', 'O', '.', '*'. Повторение символов дает более плотную штриховку
`linestyle, ls`	Настройка стиля линии прямоугольника: 'solid', 'dashed', 'dashdot', 'dotted'
`linewidth, lw`	Настройка толщины линии прямоугольника в пт

Пример П7.16. Программа в листинге 7.17 представляет простой график длин волн с аннотациями, показывающий различные области электромагнитного спектра с помощью методов `text`, `axvline`, `axhline` и `axvspan` (см. рис. 7.18).

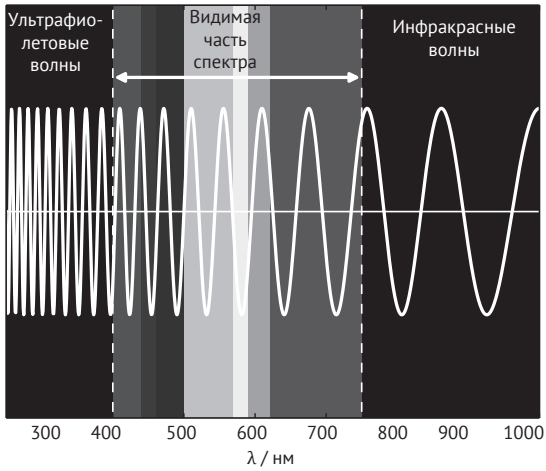


Рис. 7.18. Графическое представление электромагнитного спектра

Листинг 7.17. Графическое представление электромагнитного спектра в диапазоне 250–1000 нм

```
# eg7-annotate.py
import numpy as np
import matplotlib.pyplot as plt

# Диапазон длин волн, нм.
lmin, lmax = 250, 1000
x = np.linspace(lmin, lmax, 1000)
# Волна с плавно увеличивающейся длиной.
wv = (np.sin(10 * np.pi * x / (lmax+lmin -x)))[::-1]
```

```
fig = plt.figure()
ax = fig.add_subplot(facecolor='k')
ax.plot(x, wv, c='w', lw=2)
ax.set_xlim(250, 1000)
ax.set_ylim(-2, 2)

# Надпись и разделительная линия различных областей электромагнитного спектра.
ax.text(310, 1.5, 'UV', color='w', fontdict={'fontsize': 20})
ax.text(530, 1.5, 'Visible', color='k', fontdict={'fontsize': 20})
ax.annotate('', (400, 1.3), (750, 1.3), arrowprops={'arrowstyle': '<|-|>',
                                                    'color': 'w', 'lw': 2})
ax.text(860, 1.5, 'IR', color='w', fontdict={'fontsize': 20})
ax.axvline(400, -2, 2, c='w', ls='--')
ax.axvline(750, -2, 2, c='w', ls='--')
# Горизонтальная "ось", проходящая через центр волны.
ax.axhline(c='w')
# Удаление штриховых меток и подписей к ним на оси y; подписи к штриховым меткам на оси x.
ax.yaxis.set_visible(False)
ax.set_xlabel(r'$\lambda\;/\mathrm{nm}$')

# Завершающий этап: добавление нескольких цветных прямоугольников, представляющих цвета радуги
# в видимой области спектра.
# Словарь, отображающий области длин волн (нм) в приблизительно соответствующие значения RGB.
rainbow_rgb = { (400, 440): '#8b00ff', (440, 460): '#4b0082',
                (460, 500): '#0000ff', (500, 570): '#00ff00',
                (570, 590): '#ffff00', (590, 620): '#ff7f00',
                (620, 750): '#ff0000'}
for wv_range, rgb in rainbow_rgb.items():
    ax.axvspan(*wv_range, color=rgb, ec='none', alpha=1)
plt.show()
```

7.4.4 ◊ Круги, многоугольники и прочие фигуры

Почти все элементы, которые изображаются на рисунке Matplotlib, являются подклассами абстрактного базового класса `Artist`. Этот класс включает линии (как объекты типа `Line2D`) и текст (как объекты типа `Text`)[106]. Далее важный набор отображаемых объектов наследуется от подкласса `Patch` (являющегося производным от класса `Artist`): двумерная фигура. Секторы (клинья) круговой диаграммы (см. раздел 7.3) и стрелки в аннотации (см. раздел 7.4) представляют собой примеры таких объектов, которые уже встречались ранее.

Для добавления фигуры в объект `Axes` создается патч («заплатка»), использующий один из классов, подробно описанных в документации Matplotlib (https://matplotlib.org/api/artist_api.html), и вызывается метод `ax.add_patch(`*`patch`*`)`. Для определения цвета, толщины линий, прозрачности и т. д. при создании патча передается один или несколько именованных аргументов, описанных в табл. 7.11.

[106] В действительности существует два типа объектов Artist: примитивы и контейнеры. Примитивы – это графические объекты (такие как простые линии типа `Line2D`), а контейнеры – это элементы рисунка, в которые включены отображаемые графические объекты (например, `Axes`).

Практическое использование нескольких типов объектов `Patch` описано в следующих подразделах.

Окружности и эллипсы

Для окружности `Circle` определяется центр в точке `xy = (x, y)` (в координатах данных) и радиус `r`. Окружность создается следующими командами:

```
from matplotlib.patches import Circle
circle = Circle(xy, r, **kwargs)
```

В объект `Axes` созданная окружность добавляется с помощью метода `ax.add_patch`:

```
ax.add_patch(circle)
```

Поддерживаемые именованные аргументы, обозначенные `**kwargs`, – это обычные параметры стиля патча, описанные в табл. 7.11.

Патчи типа `Ellipse` похожи на окружности, но принимают аргументы `width` и `height` (полная длина горизонтальной и вертикальной осей эллипса перед его поворотом) и `angle` (угол поворота эллипса против часовой стрелки в градусах).

```
from matplotlib.patches import Ellipse
ellipse = Ellipse(xy, width, height, angle, **kwargs)
```

Пример П7.17. Код в листинге 7.18 считывает рост и массу тела 260 женщин и 247 мужчин из набора данных, опубликованного Хайнцем (Heinz) и др.[107] Этот набор данных можно скачать здесь: https://scipython.com/eg/bai. Код создает график пар (рост, масса) для каждого человека на точечной диаграмме, и для каждого пола изображает 3σ ковариационный эллипс вокруг средней точки. Размеры этого эллипса принимаются по (промасштабированным) собственным значениям ковариационной матрицы, и эллипс поворачивается так, чтобы его главная полуось располагалась вдоль наибольшего собственного вектора.

Листинг 7.18. Анализ отношения роста и массы тела 507 здоровых людей

```
# eg7-body-mass-height.py
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.patches import Ellipse

FEMALE, MALE = 0, 1
dt = np.dtype([('mass', 'f8'), ('height', 'f8'), ('gender', 'i2')])
data = np.loadtxt('body.dat.txt', usecols=(22, 23, 24), dtype=dt)

fig, ax = plt.subplots()
```

[107] *G. Heinz et al.* Journal of Statistical Education 11(2), (2003). Статья доступна здесь: https://doi.org/10.1080/10691898.2003.11910711.

```
def get_cov_ellipse(cov, center , nstd , **kwargs):
    """
    Return a matplotlib Ellipse patch representing the covariance matrix
    cov centered at center and scaled by the factor nstd.
    """
    # """
    # Возвращает патч Ellipse Matplotlib, представляющий ковариационную матрицу cov,
    # отцентрированную по средней точке и промасштабированную с коэффициентом nstd.
    # """

    # Поиск и сортировка собственных значений и собственных векторов в убывающем порядке.
    eigvals, eigvecs = np.linalg.eigh(cov)
    order = eigvals.argsort()[::-1]
    eigvals, eigvecs = eigvals[order], eigvecs[:, order]

    # Угол поворота против часовой стрелки созданного эллипса.
    vx, vy = eigvecs[:, 0][0], eigvecs[:, 0][1]
    theta = np.arctan2(vy, vx)                                          ❶

    # Ширина и высота отображаемого эллипса.
    width, height = 2 * nstd * np.sqrt(eigvals)
    return Ellipse(xy=center, width=width, height=height,
                   angle=np.degrees(theta), **kwargs)

labels, colors =['Female', 'Male'], ['magenta', 'blue']
for gender in (FEMALE, MALE):
    sdata = data[data['gender']==gender]
    height_mean = np.mean(sdata['height'])
    mass_mean = np.mean(sdata['mass'])
    cov = np.cov(sdata['mass'], sdata['height'])
    ax.scatter(sdata['height'], sdata['mass'], color=colors[gender],
               label=labels[gender])
    e = get_cov_ellipse(cov, (height_mean, mass_mean), 3,
                        fc=colors[gender], alpha=0.4)
    ax.add_patch(e)

ax.set_xlim(140, 210)
ax.set_ylim(30, 120)
ax.set_xlabel('Height /cm')
ax.set_ylabel('Mass /kg')
ax.legend(loc='upper left', scatterpoints=1)
plt.show()
```

❶ Функция `np.arctan2` возвращает «арктангенс по двум аргументам»: `np.arctan2(y, x)` – угол в радианах между положительным направлением оси *x* и точкой (*x*, *y*) (точнее, прямой из начала координат до точки (*x*, *y*)).

На рис. 7.19 показан итоговый график.

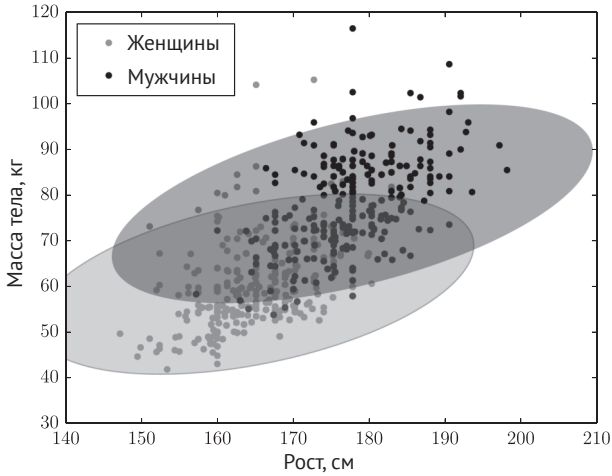


Рис. 7.19. Точечные диаграммы по каждому полу, отображающие отношение массы тела и роста 507 студентов с соответствующими ковариационными эллипсами с аннотацией

Прямоугольники

Патчи типа `Rectangle` создаются аналогично эллипсам `Ellipse`:

```
from matplotlib.patches import Rectangle
rectangle = Rectangle(xy, width, height, angle, **kwargs)
```

Но здесь кортеж `xy=(x,y)` определяет координаты нижнего левого угла прямоугольника. Разумеется, квадрат – это просто прямоугольник с равными значениями `width` и `height`.

Многоугольники

Патч типа `Polygon` создается с помощью передачи массива формы `(N, 2)`, в котором каждая строка представляет координаты вершины (*x*, *y*). Если дополнительный аргумент `closed` содержит значение `True` (по умолчанию), то многоугольник будет замкнутым, т. е. начальная и конечная точки вершин совпадают. Это показано в примере П7.18.

Пример П7.18. Код в листинге 7.19 создает изображение нескольких цветных фигур (см. рис. 7.20).

Листинг 7.19. Создание цветных фигур

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.patches import Polygon , Circle , Rectangle

red, blue, yellow, green = '#ff0000', '#0000ff', '#ffff00', '#00ff00'
square = Rectangle((0.7, 0.1), 0.25, 0.25, facecolor=red)
circle = Circle((0.8, 0.8), 0.15, facecolor=blue)
```

```
triangle = Polygon(((0.05, 0.1), (0.396, 0.1), (0.223, 0.38)), fc=yellow)
rhombus = Polygon(((0.5, 0.2), (0.7, 0.525), (0.5, 0.85), (0.3, 0.525)), fc=green)
fig = plt.figure(facecolor='k')
ax = fig.add_subplot(aspect='equal')
for shape in (square, circle, triangle, rhombus):
    ax.add_patch(shape)
ax.axis('off')

plt.show()
```

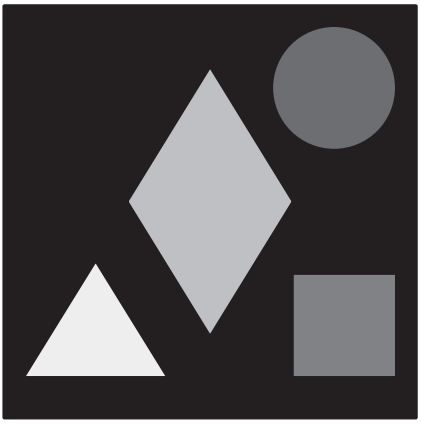


Рис. 7.20. Некоторые цветные фигуры, созданные с использованием патчей Matplotlib

Вопросы

B7.4.1. Сравнить графики функции $y = x^3$ при $-10 \leqslant x \leqslant 10$, используя логарифмическую шкалу для оси x, для оси y и для обеих осей. В чем различие между использованием `ax.set_xscale('log')` и `ax.set_xscale('symlog')`?

B7.4.2. Изменить пример П7.9 так, чтобы создавалась горизонтальная столбиковая диаграмма со столбцами в порядке уменьшения частоты встречаемости букв (т. е. чтобы самая часто встречающаяся буква E была представлена нижним столбцом).

Задачи

37.4.1. Индекс бигмака, определяемый журналом The Economist, – это неофициальный способ определения (измерения) паритета покупательной способности (ППС) между двумя валютами. Предпосылка его определения состоит в том, что различие между ценой бигмака в закусочной Макдональд в какой-либо валюте, преобразованная в долл. США по превалирующему обменному курсу, и ценой бигмака в США является мерой переоценки или недооценки сравниваемой валюты (по отношению к доллару).

В файлах https://scipython.com/eg/bga представлена хронология цен на бигмак и обменные курсы для четырех валют. Для каждой валюты необходимо вычислить процент переоценки или недооценки по формуле

$$\frac{(\textit{местная цена бигмака, преобразованная в долл США} - \textit{цена бигмака в США})}{(\textit{цена бигмака в США})} \times 100$$

и построить график изменения этой величины как функции от времени.

37.4.2. Построить график в виде гистограммы по данным табл. 7.12, содержащей количество случаев заболевания вирусом лихорадки Западного Нила в США в период с 1999 по 2008 г. Два типа заболевания – нейроинвазивный и ненейроинвазивный – должны быть отображены как отдельные столбцы на одном графике для каждого года.

Таблица 7.12

Год	Нейроинвазивный тип	Ненейроинвазивный тип
1999	59	3
2000	19	2
2001	64	2
2002	2946	1210
2003	2866	6996
2004	1148	1391
2005	1309	1691
2006	1495	2774
2007	1227	117
2008	689	667

37.4.3. Кружковая («пузырьковая») диаграмма представляет собой тип точечной диаграммы, который может отображать три измерения данных с использованием позиции (точки данных как координаты *x* и *y*) и размера маркера. Метод `plt.scatter` может создавать кружковые диаграммы, принимая размер маркера в атрибуте `s` (в кв. пт. – так что площадь маркера пропорциональна величине в третьем измерении – см. пример П7.1).

Файлы *gdp.tsv*, *bmi_men.tsv* и *population_total.tsv*, доступные для скачивания по адресу https://scipython.com/eg/bgc, содержат следующие данные, начиная с 2007 г. для каждой страны: ВВП на душу населения в международных долл., зафиксированных в ценах 2005 г., индекс массы тела (ИМТ) мужчин (в кг/м^2) и общая численность населения. Создать кружковую диаграмму отношения ИМТ и ВВП, на которой численность населения соответствует размеру круговых маркеров. Предупреждение: для некоторых стран отдельные точки данных отсутствуют.

Дополнительное задание: обозначить цветом кружки по континентам, используя список из файла *continents.tsv*.

37.4.4. Национальное управление океанических и атмосферных исследований США (NOAA) создает набор данных о концентрации в атмосфере двуокиси углерода (CO_2) с 1958 г. Данные свободно доступны для всех здесь: ftp://aftp.cmdl.noaa.gov/products/trends/co2/co2_mm_mlo.txt. Используя эти данные,

создать график «интерполированных» и «трендовых» значений концентрации CO_2 во времени на одном рисунке.

37.4.5. Написать программу для построения графика функции Планка $D(\lambda)$ для спектральной плотности энергетической светимости абсолютно черного тела при температуре T как функции от длины волны λ для Солнца (T = 5778 K):

$$B(\lambda) = \frac{2hc^2}{\lambda^5} \frac{1}{\exp(hc / \lambda k_B T) - 1}.$$

Использовать массив NumPy для хранения значений функции $B(\lambda)$ в диапазоне от 100 до 5000 нм, но установить сокращенный диапазон длин волн от 4000 до 0 нм. Необходимые физические константы можно принять в следующем виде: $h = 6.626 \times 10^{-34}$ Дж·с, $c = 2.998 \times 10^{8}$ м/с и $k_B = 1.381 \times 10^{-23}$ Дж/К.

37.4.6. Воспроизвести рис. 7.21, используя патчи типа `Circle`.

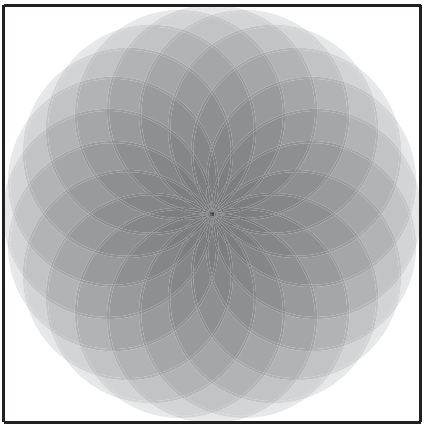


Рис. 7.21. Изображение, полученное с помощью патчей типа Circle Matplotlib

7.5 Контурные диаграммы и тепловые карты

До сих пор мы рассматривали построение графиков только для одномерных данных (т. е. функции только от одной координаты). Библиотека Matplotlib также поддерживает несколько способов построения графиков данных, являющихся функциями двух измерений.

7.5.1 Контурные диаграммы

В модуле `pyplot` метод `contour` создает контурную диаграмму из предоставленного двумерного массива. При самом простом вызове `contour(Z)` не требуется никаких других аргументов: значения (x, y) индексируются в двумерном массиве `Z`, а интервалы контуров выбираются автоматически. Для явного определения значений координат (x, y) необходимо передать их в виде `contour(X, Y, Z)`. Массивы `X` и `Y` обязательно должны иметь ту же форму, что и массив `Z` (например, как это сделано при использовании метода `np.meshgrid`, см. раздел 6.1.6),

или должны быть одномерными массивами, такими, что длина массива `X` равна числу столбцов массива `Z`, а длина массива `Y` равна числу строк массива `Z`.

Уровнями контуров можно управлять с помощью дополнительного аргумента: либо скалярного значения `N`, определяющего общее количество уровней контуров, либо последовательности `V`, явно перечисляющей значения `Z`, для которых необходимо отображать контуры.

Цвета контуров определяются в соответствии с цветовой картой (colormap) Matplotlib, принятой по умолчанию. В этом процессе данные нормализуются линейно в интервал `[0, 1]`, который далее отображается в список цветов, используемых для определения стилей контуров по соответствующим значениям. Модуль `matplotlib.cm` предоставляет несколько схем цветовых карт[108]: некоторыми наиболее часто применяемыми на практике являются схемы `cm.viridis` (с версии Matplotlib 2.0 схема по умолчанию), `cm.hot`, `cm.bone`, `cm.winter`, `cm.jet`, `cm.Greys` и `cm.hsv`. Если необходимо использовать цветовые схемы с измененными на противоположные цветами, то необходимо добавлять суффикс `_r` в конец имени схемы (например, `cm.hot_r`).

Метод `contour` поддерживает и другой способ определения цветов – в аргументе `colors`, в котором передается либо один спецификатор цвета Matplotlib, либо последовательность таких спецификаторов. На одноцветных контурных диаграммах контуры, соответствующие отрицательным значениям, изображаются штриховыми линиями. Для толщины линий контуров можно определять стили по отдельности или для всех вместе с помощью аргумента `linewidths`.

Пример П7.19. Код в листинге 7.20 создает график электростатического потенциала электрического диполя **p** = (qd, 0, 0) на плоскости (x, y) при $q = 1.602 \times 10^{-19}$ Кл, d = 1 пм с использованием точечной аппроксимации диполя (см. рис. 7.22).

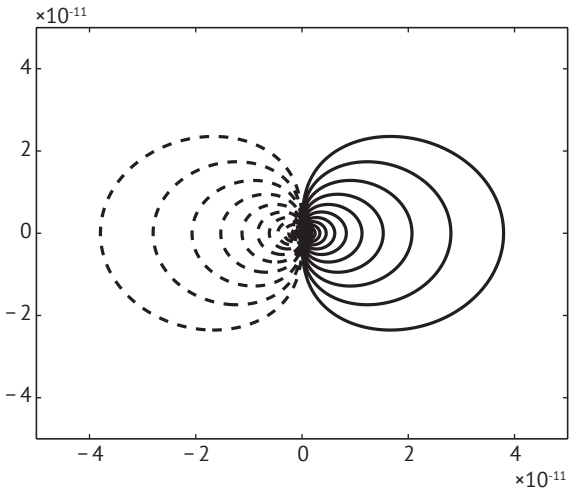


Рис. 7.22. Контурная диаграмма электростатического потенциала точечного диполя

[108] Полный список см. на веб-странице https://matplotlib.org/tutorials/colors/colormaps.html.

Листинг 7.20. Электростатический потенциал точечного диполя

```
# eg7-elec-dipole-pot.py
import numpy as np
import matplotlib.pyplot as plt

# Заряд диполя (Кл), диэлектрическая постоянная свободного пространства (Ф/м).
q, eps0 = 1.602e-19, 8.854e-12
# Дипольное расстояние +q, -q (м) и подходящее сочетание параметров.
d = 1.e-12
k = 1/4/np.pi/eps0 * q * d

# Декартова система координат с началом в диполе (м).
X = np.linspace(-5e-11, 5e-11, 1000)
Y = X.copy()
X, Y = np.meshgrid(X, Y)

# Электростатический потенциал диполя (V) с использованием точечной аппроксимации диполя.
Phi = k * X / np.hypot(X, Y)**3

fig, ax = plt.subplots()
# Отображение контуров по значениям Phi, определяемым по уровням.
levels = np.array([10**pw for pw in np.linspace(0, 5, 20)])
levels = sorted(list(-levels) + list(levels))
# Одноцветная контурная диаграмма потенциала.
ax.contour(X, Y, Phi, levels=levels, colors='k', linewidths=2)
plt.show()
```

Для добавления надписей к контурам необходимо сохранить объект `ContourSet`, возвращаемый после вызова метода `ax.contour`, и передать этот объект в метод `ax.clabel` (возможно, с некоторыми дополнительными параметрами, определяющими свойства шрифта). Еще один метод `ax.contourf` принимает те же аргументы, что и метод `contour`, но изображает закрашенные контуры (т. е. с заливкой областей контуров). Методы `ax.contour` и `ax.contourf` можно использовать вместе, как показано в примере П7.20.

Пример П7.20. Программа в листинге 7.21 создает график функции с закрашенными контурами, добавляет надписи для контуров и определяет некоторые специализированные стили для цветов контуров (см. рис. 7.23).

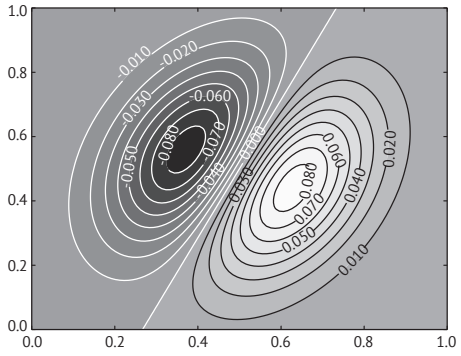


Рис. 7.23. Двумерный график, изображающий контуры с надписями

Листинг 7.21. Пример изображения контуров с заливкой цветом и определением стилей

```
# eg7-2dgau.py
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm

X = np.linspace(0, 1, 100)
Y = X.copy()
X, Y = np.meshgrid(X, Y)
alpha = np.radians(25)
cX, cY = 0.5, 0.5
sigX, sigY = 0.2, 0.3
rX = np.cos(alpha) * (X-cX) - np.sin(alpha) * (Y-cY) + cX
rY = np.sin(alpha) * (X-cX) + np.cos(alpha) * (Y-cY) + cY

Z = (rX-cX)*np.exp(-((rX-cX)/sigX)**2) * np.exp(-((rY-cY)/sigY)**2)
fig = plt.figure()
ax = fig.add_subplot()

# Реверсированная цветовая карта Greys для контуров с заливкой.
cpf = ax.contourf(X, Y, Z, 20, cmap=cm.Greys_r)
# Установить цвета контуров и надписей так, чтобы они изображались белым цветом, когда
# заливка контура темная (Z < 0), и черным цветом, когда заливка контура светлая (Z >= 0).
colors = ['w' if level <0 else 'k' for level in cpf.levels]
cp = ax.contour(X, Y, Z, 20, colors=colors)
ax.clabel(cp, fontsize=12, colors=colors)
plt.show()
```

7.5.2 Тепловые карты

Еще одним способом отображения двумерных данных является тепловая карта (heatmap): изображение, в котором цвет каждого пиксела определяется соответствующим значением в массиве данных. Практическое использование методов Matplotlib `ax.imshow`, `ax.pcolor` и `ax.pcolormesh` рассматривается в этом разделе.

Метод `ax.imshow`

Метод объекта `Axes` `ax.imshow` выводит изображение в заданных осях. При самом простом варианте использования метод принимает двумерный массив и отображает его значения в пикселы изображения в соответствии с некоторой схемой интерполяции и нормализации. Если данные массива взяты из изображения, считанного с помощью метода Matplotlib `image.imread`, то обычно это все, что требуется:

```
In [x]: import matplotlib.pyplot as plt
In [x]: import matplotlib.image as mpimg
In [x]: im = mpimg.imread('image.jpg')
In [x]: plt.imshow(im)
In [x]: plt.show()
```

(В рассматриваемом здесь примере `im` – это трехмерный массив формы `(n, m, 3)`, в котором координата «глубины» соответствует красному, зеленому и синему компонентам цвета каждого пиксела в изображении типа `n-by-m`.)

Метод `imshow` часто используется для визуализации матриц или других двумерных массивов данных. Если изображение, полученное для итогового рисунка, имеет размер, отличающийся от размеров массива, то применяется некоторый тип схемы интерполяции: например, для визуализации матрицы 10×10 в виде изображения 100×100 пикселов требуется аппроксимация огромного количества промежуточных точек. По умолчанию принята схема интерполяции `'nearest'`, наиболее точная для исходных данных, но иногда она может выглядеть слишком «плотной» (перенасыщенной). Существует много других схем интерполяции (подробные описания схем интерполяции см. в официальной документации: https://matplotlib.org/api/_as_gen/matplotlib.axes.Axes.imshow.html): практическое применение схемы `interpolation='bilinear'` показано ниже в примере П7.21 – схема позволяет получить своеобразно выглядящее размытое изображение даже для массивов небольшого размера. Следует отметить, что метод `imshow` принимает аргумент `cmap`, которому присваивается цветовая карта точно так же, как это делалось для метода `ax.contourf`.

Пример П7.21. В коде листинга 7.22 сравниваются две схемы интерполяции `'bilinear'` и `'nearest'` (принятая по умолчанию), которая должна выглядеть чрезвычайно «плотной» (т. е. более точно соответствующей исходным данным): см. рис. 7.24.

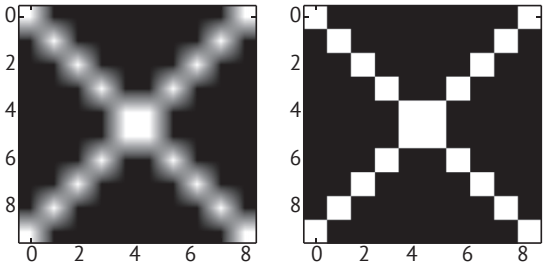


Рис. 7.24. Визуализация небольшой матрицы с использованием метода ax.imshow с двумя различными схемами интерполяции

Листинг 7.22. Сравнение применения различных схем интерполяции для визуализации небольшого массива с помощью метода imshow()

```
# eg7-matrix-show.py
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm

# Создание массива с единицами, образующими фигуру в форме 'X'.
a = np.eye(10, 10)
a += a[::-1,:]
fig = plt.figure()
```

```
ax1 = fig.add_subplot (121)
# Билинейная интерполяция - фигура будет выглядеть размытой.
ax1.imshow(a, interpolation='bilinear', cmap=cm.Greys_r)

ax2 = fig.add_subplot (122)
# Интерполяция по умолчанию 'nearest' interpolation - наиболее точная, но выглядит
"угловато".
ax2.imshow(a, cmap=cm.Greys_r)
plt.show()
```

В последние версии Matplotlib включен метод объекта `Axes` `matshow`, который можно использовать вместо метода `imshow` и устанавливать требуемые параметры по умолчанию для визуализации матриц.

Пример П7.22. Папоротник Барнсли (Barnsley fern) – это фрактал, напоминающий один из видов папоротника, костенец черный. Фрактал формируется посредством изображения последовательности точек на плоскости (x, y), начиная с начала координат (0, 0), сгенерированной следующими аффинными преобразованиями f_1, f_2, f_3, f_4, при этом каждое преобразование применяется к предыдущей точке и выбирается случайным образом с вероятностями $p_1 = 0.01$, $p_2 = 0.85$, $p_3 = 0.07$ и $p_4 = 0.07$:

$$f_1(x,y) = \begin{pmatrix} 0 & 0 \\ 0 & 0.16 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix},$$

$$f_2(x,y) = \begin{pmatrix} 0.85 & 0.04 \\ -0.04 & 0.85 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} 0 \\ 1.6 \end{pmatrix},$$

$$f_3(x,y) = \begin{pmatrix} 0.2 & -0.26 \\ 0.23 & 0.22 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} 0 \\ 1.6 \end{pmatrix},$$

$$f_4(x,y) = \begin{pmatrix} -0.15 & 0.28 \\ 0.26 & 0.24 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} 0 \\ 0.44 \end{pmatrix}.$$

Этот алгоритм реализован в программе из листинга 7.23, а результат показан на рис. 7.25.

Листинг 7.23. Папоротник Барнсли

```
# eg7-fern.py
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm

f1 = lambda x, y: (0., 0.16*y)
f2 = lambda x, y: (0.85*x + 0.04*y, -0.04*x + 0.85*y + 1.6)
f3 = lambda x, y: (0.2*x - 0.26*y, 0.23*x + 0.22*y + 1.6)
f4 = lambda x, y: (-0.15*x + 0.28*y, 0.26*x + 0.24*y + 0.44)
fs = [f1, f2, f3, f4]

npts = 50000
# Размер поля изображения (в пикселах).
```

```
width, height = 300, 300
aimg = np.zeros((width, height))

x, y = 0, 0
for i in range(npts):
    # Случайный выбор преобразования и его применение.
    f = np.random.choice(fs, p=[0.01, 0.85, 0.07, 0.07])
    x, y = f(x, y)
    # Отображение (x, y) в координаты пиксела.
    # Обратите внимание: мы "знаем", что -2.2 < x < 2.7 и 0 <= y < 10.

    ix, iy = int(width / 2 + x * width / 10), int(y * height / 12)
    # Установка для этой точки массива значения 1 для пометки точки в изображении папоротника.
    aimg[iy, ix] = 1

plt.imshow(aimg[::-1,:], cmap=cm.Greens)
plt.show()
```

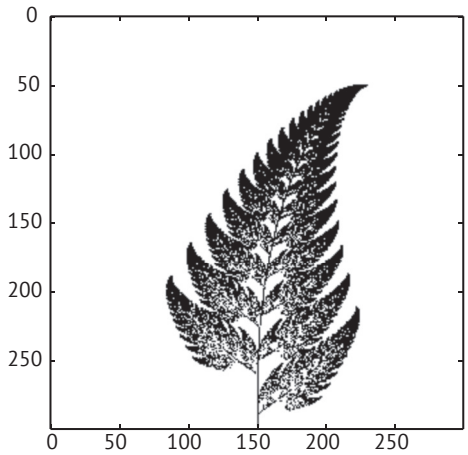


Рис. 7.25. Фрактал папоротник Барнсли

Методы ax.pcolor и ax.pcolormesh

Существует еще пара методов Matplotlib, с которыми вам придется иметь дело: `ax.pcolor` и `ax.pcolormesh`. Они очень похожи друг на друга. Детальные различия не рассматриваются в этой книге, но `pcolormesh` гораздо быстрее, чем `pcolor`, поэтому рекомендуется именно этот вариант для `imshow`. Наиболее существенное различие состоит в том, что `imshow` соблюдает соглашение, используемое в сообществе обработки изображений, по которому начало координат помещается в левый верхний угол, а метод `pcolor` связывает начало координат с левым нижним углом.

Цветовые шкалы

Зачастую удобно иметь легенду, поясняющую связь цветов на графике со значениями массива, используемыми для создания этого графика. Такая

легенда добавляется с помощью метода `fig.colorbar`. В самом простом варианте вызывается `fig.colorbar(mappable)`, где `mappable` – это объект `Image`, `ContourSet` или другой подходящий объект, к которому применяется цветовая шкала, а новый объект `Axes` содержит эту создаваемую цветовую шкалу (и выделяет для нее соответствующую область на рисунке). Далее этот объект можно настраивать и добавлять к нему надписи, как показано в следующих примерах.

Пример П7.23. Код в листинге 7.24 считывает данные из файла с результатами измерений максимальной дневной температуры в Бостоне в 2019 г. и размещает эти данные на тепловой карте с легендой в виде цветовой шкалы с надписями (см. рис. 7.26). Файл данных можно скачать здесь: https://scipython.com/eg/bah.

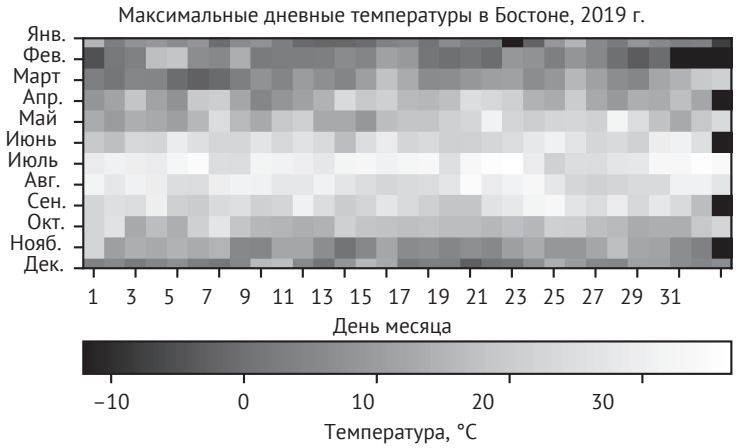


Рис. 7.26. Тепловая карта максимальных дневных температур в Бостоне в течение 2019 г.

Листинг 7.24. Тепловая карта дневных температур в Бостоне в 2019 г.

```
# eg7-heatmap.py

import numpy as np
import matplotlib.pyplot as plt

# Считывание исходных данных из входного файла.
dt = np.dtype([('month', np.int), ('day', np.int), ('T', np.float)])
data = np.genfromtxt('boston2019.dat', dtype=dt, usecols=(1, 2, 3),
                     delimiter=(4, 2, 2, 6))

# В создаваемой тепловой карте значение nan означает "нет такой даты", например 31 июня.
heatmap = np.empty((12, 31))
heatmap[:] = np.nan

for month, day, T in data:
    # Массивы NumPy индексируются с нуля, но для номеров дней и месяцев это неприемлемо.
    heatmap[month-1, day-1] = T
```

```
# Создание тепловой карты, настройка и добавление подписей к штриховым меткам на осях.
fig = plt.figure()
ax = fig.add_subplot()
im = ax.imshow(heatmap, interpolation='nearest')
ax.set_yticks(range(12))
ax.set_yticklabels(['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun',
                    'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'])
days = np.array(range(0, 31, 2))
ax.set_xticks(days)
ax.set_xticklabels(['{:d}'.format(day+1) for day in days])
ax.set_xlabel('Day of month')
ax.set_title('Maximum daily temperatures in Boston , 2019')

# Добавление цветовой шкалы вдоль нижней границы тепловой карты и необходимых подписей.
cbar = fig.colorbar(ax=ax, mappable=im, orientation='horizontal')                    ❶
cbar.set_label('Temperature , $^\circ\mathrm{C}$')

plt.show()
```

❶ «mappable»-объект, передаваемый в метод `fig.colorbar`, – это объект `AxesImage`, возвращаемый методом `ax.imshow`.

Пример П7.24. Уравнение диффузии в двумерном пространстве

$$\frac{\partial U}{\partial t} = D\left(\frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2}\right),$$

где D – коэффициент диффузии. Простое численное решение этого уравнения в области единичного квадрата $0 \leqslant x \leqslant 1$, $0 \leqslant y \leqslant 1$ подразумевает аппроксимацию $U(x, y; t)$ дискретной функцией $u_{i,j}^{(n)}$, где $x = i\Delta x$, $y = j\Delta y$ и $t = n\Delta t$. Применение аппроксимации методом конечных разностей дает следующую формулу:

$$\frac{u_{i,j}^{(n+1)} - u_{i,j}^{(n)}}{\Delta t} = D\left[\frac{u_{i+1,j}^{(n+1)} - 2u_{i,j}^{(n)} + u_{i-1,j}^{(n)}}{(\Delta x)^2} + \frac{u_{i,j+1}^{(n)} - 2u_{i,j}^{(n)} + u_{i,j-1}^{(n)}}{(\Delta y)^2}\right],$$

следовательно, состояние системы в момент времени $n + 1$, $u_{i,j}^{(n+1)}$ можно вычислить по ее состоянию в момент времени n, $u_{i,j}^{(n)}$, используя уравнение

$$u_{i,j}^{(n+1)} = u_{i,j}^{(n)} = D\Delta t\left[\frac{u_{i+1,j}^{(n+1)} - 2u_{i,j}^{(n)} + u_{i-1,j}^{(n)}}{(\Delta x)^2} + \frac{u_{i,j+1}^{(n)} - 2u_{i,j}^{(n)} + u_{i,j-1}^{(n)}}{(\Delta y)^2}\right].$$

Рассмотрим уравнение диффузии, примененное к металлической пластине, имеющей начальную температуру T_{cold}, при контакте с диском заданного размера, имеющим температуру T_{hot}. Предполагается, что на краях пластины постоянно сохраняется температура T_{cool}. В приведенном ниже фрагменте кода применяется последняя формула для наблюдения за распространением температуры в пластине. Можно показать, что максимальный интервал времени Δt, который можно принять без потери стабильности процесса, равен

$$\Delta t = \frac{1}{2D}\frac{(\Delta x \Delta y)^2}{(\Delta x)^2 + (\Delta y)^2}.$$

В коде из листинга 7.25 каждый вызов `do_timestep` обновляет массив NumPy `u` по результатам, полученным в предыдущем интервале времени `u0`. Простейшей методикой применения уравнения в частных разностях является использование цикла Python:

```
for i in range(1, nx -1):
    for j in range(1, ny -1):
        uxx = (u0[i+1,j] - 2*u0[i,j] + u0[i-1,j]) / dx2
        uyy = (u0[i,j+1] - 2*u0[i,j] + u0[i,j-1]) / dy2
        u[i,j] = u0[i,j] + dt * D * (uxx + uyy)
```

Но этот способ работает чрезвычайно медленно, а использование векторизации позволит заменить эти явные циклы на гораздо более быстрый предварительно скомпилированный код C, примененный в реализации массива NumPy.

Состояние системы выводится как изображение в четырех различных изменяемых состояниях (см. рис. 7.27).

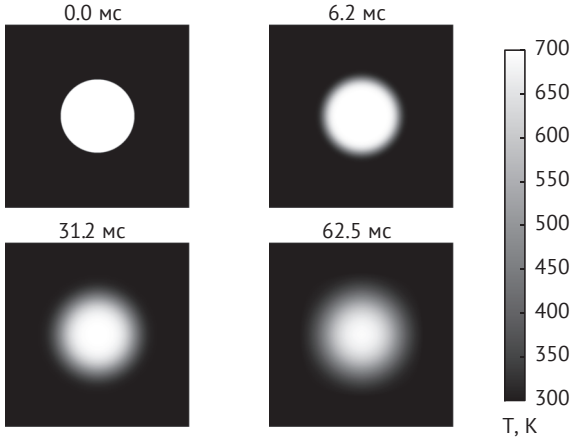


Рис. 7.27. Представление изменения температуры (кругового) диска в четыре момента времени после его мгновенного нагревания

Листинг 7.25. Уравнение диффузии в двумерном пространстве, примененное к распространению температуры в стальной пластине

```
# eg7-diffusion2d.py
import numpy as np
import matplotlib.pyplot as plt

# Размер пластины, мм.
w = h = 10.
# Интервалы в направлениях x, y, мм.
dx = dy = 0.1


# Коэффициент термической диффузии стали, мм2/с.
D = 4.
```

```
Tcool, Thot = 300, 700

nx, ny = int(w/dx), int(h/dy)

dx2, dy2 = dx*dx, dy*dy
dt = dx2 * dy2 / (2 * D * (dx2 + dy2))

u0 = Tcool * np.ones((nx, ny))
u = u0.copy()

# Начальные условия - кольцо с внутренним радиусом r, толщиной dr с центром в (cx, cy) (мм).
r, cx, cy = 2, 5, 5
r2 = r**2
for i in range(nx):
    for j in range(ny):
        p2 = (i*dx-cx)**2 + (j*dy-cy)**2
        if p2 < r2:
            u0[i, j] = Thot

def do_timestep(u0, u):
    # Распространение с правосторонней разностью (разностью вперед) по времени,
    # с центральной разностью в пространстве.
    u[1:-1, 1:-1] = u0[1:-1, 1:-1] + D * dt * (
          (u0[2:, 1:-1] - 2*u0[1:-1, 1:-1] + u0[:-2, 1:-1])/dx2
          + (u0[1:-1, 2:] - 2*u0[1:-1, 1:-1] + u0[1:-1, :-2])/dy2 )

    u0 = u.copy()
    return u0, u

# Количество интервалов времени.
nsteps = 101
# Вывод 4 изображений в определенные моменты времени.
mfig = [0, 10, 50, 100]
fignum = 0
fig = plt.figure()
for m in range(nsteps):
    u0, u = do_timestep(u0, u)
    if m in mfig:
        fignum += 1
        print(m, fignum)
        ax = fig.add_subplot(220 + fignum)
        im = ax.imshow(u.copy(), cmap=plt.get_cmap('hot'), vmin=Tcool , vmax=Thot)
        ax.set_axis_off()
        ax.set_title('{:.1f} ms'.format(m*dt*1000))
fig.subplots_adjust(right=0.85)
cbar_ax = fig.add_axes([0.9, 0.15, 0.03, 0.7])                                    (1)
cbar_ax.set_xlabel('$T$ / K', labelpad=20)
fig.colorbar(im, cax=cbar_ax)
plt.show()
```

(1) Для настройки общей цветовой шкалы `colorbar` для четырех графиков определяются собственные объекты шкалы `Axes`, `cbar_ax` и выделяется место для нее с помощью метода `fig.subplots_adjust`. Во всех графиках используется одинаковый цветовой диапазон, определяемый значениями `vmin` и `vmax`, поэтому не имеет значения, который из них передается в первом аргументе в метод `fig.colorbar`.

7.5.3 Упражнения

Вопросы

B7.5.1. Создать график функции sinc в декартовой системе координат на плоскости. Функция $\text{sinc}(r) = \sin r / r$, где $r = \sqrt{(x^2 + y^2)}$.

B7.5.2. Данные в файле с разделенными запятыми значениями *birthday-data.csv*, доступном для скачивания по адресу https://scipython.com/eg/bgd, сообщают о количестве новорожденных, зарегистрированных Национальным центром медицинской статистики из системы Центров по контролю и профилактике заболеваний США за каждый день года как общий итог за период 1969–1988 гг. Столбцы обозначают номер месяца (1 = январь, 12 = декабрь), номер дня и количество живорожденных.

Использовать NumPy для вычисления оценки по каждому дню года вероятности дня рождения конкретного человека в этот день. Создать график вероятностей в виде тепловой карты, как в примере П7.23, и исследовать все его свойства, заслуживающие внимания.

Совет: данные требуют небольшой «очистки» – сначала необходимо просмотреть файл данных, чтобы обнаружить все некорректные записи.

Задачи

37.5.1. Так называемая «игра в хаос» (chaos game) представляет собой алгоритм генерации фракталов. Сначала определяется n вершин правильного многоугольника, а начальная точка (x_0, y_0) выбирается случайным образом внутри этого многоугольника. Затем генерируется последовательность точек, начиная с (x_0, y_0), в которой каждая точка является некоторой частью r расстояния между предыдущей точкой и случайно выбранной вершиной многоугольника. Например, алгоритм, примененный с параметрами $n = 3$, $r = 0.5$, создает треугольник Серпинского (Sierpinski triangle).

Написать программу построения фракталов с использованием алгоритма игры в хаос.

37.5.2. Расширить функциональность кода из примера П7.17, включив в него контуры индекса массы тела, определяемого формулой ИМТ = (масса тела, кг) / (рост, м)^2. Вывести на график эти контуры для разделения по предлагаемым категориям: «дефицит массы тела» (< 18.5), «избыточная масса тела» (> 25) и «ожирение» (> 30). Разместить вручную подписи к контурам так, чтобы они не перекрывали точки данных на точечной диаграмме, и отформатировать значения в подписях до одного знакоместа после десятичной точки.

37.5.3. Уравнение адвекции в двумерном пространстве можно записать в следующем виде:

$$\frac{\partial U}{\partial t} = -v_x \frac{\partial U}{\partial x} - v_y \frac{\partial U}{\partial y},$$

где $\mathbf{v} = (v_x, v_y)$ – вектор поля скоростей, содержащий компоненты скорости v_x и v_y, которые могут изменяться как функция от позиции (x, y). Применяя методику, аналогичную используемой в примере П7.24, можно дискретизировать

это уравнение и решить его численными методами. При распространении с правосторонней разностью (разностью вперед) по времени и с центральной разностью в пространстве получаем:

$$u_{i,j}^{(n+1)} = u_{i,j}^{(n)} - \Delta t\left[v_{x;i,j}\frac{u_{i+1,j}^{(n)} - u_{i-1,j}^{(n)}}{2\Delta x} + v_{y;i,j}\frac{u_{i,j+1}^{(n)} - u_{i,j-1}^{(n)}}{2\Delta x}\right].$$

Реализовать это приближенное численное решение в области $0 \leqslant x < 10$, $0 \leqslant y < 10$ при дискретизации с шагами $\Delta x = \Delta y = 0.1$ и при начальном условии

$$u_0(x,y) = \exp\left(-\frac{(x-c_x)^2 + (y-c_y)^2}{\alpha^2}\right),$$

где $(c_x, c_y) = (5, 5)$ и $\alpha = 2$. Принять поле скоростей циркулирующим с постоянной скоростью 0.1 относительно начала координат в точке (7, 5).

37.5.4. Множество Жюлиа (Julia set), связанное с функцией комплексного переменного $f(z) = z^2 + c$, можно изобразить, применяя следующий алгоритм.

Для каждой точки z_0 в комплексной плоскости, такой, что $-1.5 \leqslant \mathrm{Re}[z_0] \leqslant 1.5$ и $-1.5 \leqslant \mathrm{Im}[z_0] \leqslant 1.5$, выполняется итерация, соответствующая выражению $z_{n+1} = z_n^{\,2} + c$. Цвет пиксела в изображении в этой области комплексной плоскости определяется в соответствии с количеством итераций, требуемых для превышения $|z|$ некоторого критического значения $|z|_{\max}$ (или назначается черный цвет, если превышение не произошло даже после определенного максимального количества итераций $n_{\max}$).

Написать программу графического представления множества Жюлиа при $c = -0.1 + 0.65i$, используя значения $|z|_{\max} = 10$ и $n_{\max} = 500$.

37.5.5. Средние высоты гектадов – квадратов размером 10×10 км, используемых национальным картографическим агентством Ordnance Survey Великобритании при создании карт страны, собраны в форме массива NumPy в файле *gb-alt.npy*, доступном для скачивания здесь: https://scipython.com/eg/bgb.

Создать карту Британских островов, используя эти данные, с помощью метода `ax.imshow`, а также сформировать дополнительные карты, принимая среднее повышение уровня моря: а) 25 м, б) 50 м, в) 200 м. В каждом случае вычислить процент оставшейся площади суши относительно существующей в настоящее время.

7.6 Трехмерные графики

Библиотека Matplotlib главным образом предназначена для создания двумерных графиков, но она поддерживает и функциональные возможности для построения трехмерных графиков, вполне достаточные для множества целей. Простейшим способом создания трехмерного графика является импорт объекта `Axes3D` из модуля `mpl_toolkits.mplot3d` и определение для аргумента `projection` внутреннего графика значения `'3d'`:

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```

```
fig = plt.figure()
ax = fig.add_subplot(projection='3d')
```

После этого соответствующий объект `Axes` может отображать данные в трех измерениях как линейный график, точечную диаграмму, каркасную диаграмму или объемную поверхностную диаграмму[109].

7.6.1 Каркасные и объемные поверхностные диаграммы

Простейшим типом объемной поверхностной диаграммы является каркасная диаграмма («проволочная модель»), изображающая линии в трехмерной перспективе, объединяя предоставленный двумерный массив точек `Z` с сеткой значений данных, переданных в двумерных массивах `X` и `Y` (как для методов `imshow` и `contour`). По умолчанию линии в трехмерном представлении изображаются для каждой точки в массиве, но если точек слишком много, то можно определить значения аргументов `rstride` и `cstride`, чтобы задать шаг изображаемых точек в строках и столбцах массива соответственно.

Метод `ax.plot_surface` работает так же, но создает объемную поверхностную диаграмму из закрашенных элементов (патчей). Для патчей можно установить единый цвет в аргументе `color` или определить стиль с помощью специализированной цветовой схемы в аргументе `cmap`. Для метода `ax.plot_surface` аргументам `rstride` и `cstride` по умолчанию присваивается значение 10. Практическое применение обоих методов демонстрируется в примере П7.25.

Пример П7.25. Код в листинге 7.26 демонстрирует использование разнообразных параметров при создании объемных поверхностных диаграмм. Результаты показаны на рис. 7.28.

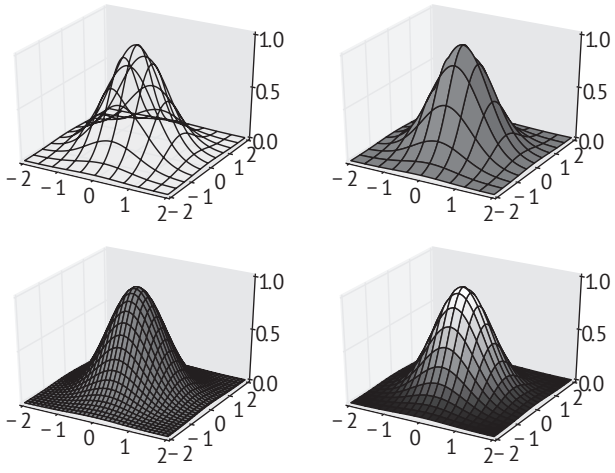


Рис. 7.28. Четыре различные трехмерные поверхностные диаграммы одной функции

[109] Существует даже возможность создания трехмерных контурных графиков и столбиковых диаграмм, хотя полезность их использования на практике сомнительна.

Листинг 7.26. Четыре трехмерные поверхностные диаграммы простой двумерной гауссовой функции

```
# eg7-3d-surface-plots.py
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.cm as cm

L, n = 2, 400
x = np.linspace(-L, L, n)
y = x.copy()
X, Y = np.meshgrid(x, y)
Z = np.exp(-(X**2 + Y**2))

fig, ax = plt.subplots(nrows=2, ncols=2, subplot_kw={'projection': '3d'})
ax[0, 0].plot_wireframe(X, Y, Z, rstride=40, cstride=40)
ax[0, 1].plot_surface(X, Y, Z, rstride=40, cstride=40, color='m')
ax[1, 0].plot_surface(X, Y, Z, rstride=12, cstride=12, color='m')
ax[1, 1].plot_surface(X, Y, Z, rstride=20, cstride=20, cmap=cm.hot)
for axes in ax.flatten():
    axes.set_xticks([-2, -1, 0, 1, 2])
    axes.set_yticks([-2, -1, 0, 1, 2])
    axes.set_zticks([0, 0.5, 1])
fig.tight_layout()
plt.show()
```

В интерактивном графике направление взгляда можно изменить, если щелкнуть и, не отпуская кнопку мыши, перетаскивать любую точку в области графика. Для фиксации конкретного направления взгляда в статическом изображении графика необходимо явно передать требуемый угол возвышения и азимутальный угол (в градусах, в указанном порядке) в метод `ax.view_init`, как показано в примере П7.26.

Пример П7.26. Параметрическое определение тора с главным радиусом *c* и радиусом образующей окружности *a* записывается в следующем виде:

$$x = (c + a\cos\theta)\cos\varphi$$
$$y = (c + a\cos\theta)\sin\varphi$$
$$z = a\sin\theta$$

Значения θ и φ должны находиться в интервале от 0 до 2π. Код в листинге 7.27 выводит два визуальных представления тора, изображаемого как объемная поверхностная диаграмма (см. рис. 7.29).

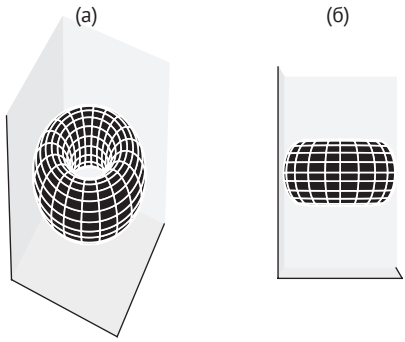


Рис. 7.29. Два визуальных представления одного и того же тора: а) θ = 36°, φ = 26°, б) θ = 0°, φ = 0°

Листинг 7.27. Трехмерная поверхностная диаграмма тора

```
# eg7-torus-surface.py
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

n = 100

theta = np.linspace(0, 2.*np.pi, n)
phi = np.linspace(0, 2.*np.pi, n)
theta, phi = np.meshgrid(theta, phi)                                    ❶
c, a = 2, 1
x = (c + a*np.cos(theta)) * np.cos(phi)
y = (c + a*np.cos(theta)) * np.sin(phi)
z = a * np.sin(theta)

fig = plt.figure()
ax1 = fig.add_subplot(121, projection='3d')
ax1.set_zlim(-3, 3)
ax1.plot_surface(x, y, z, rstride=5, cstride=5, color='k', edgecolors='w')  ❷
ax1.view_init(36, 26)                                                   ❸
ax2 = fig.add_subplot(122, projection='3d')
ax2.set_zlim(-3, 3)
ax2.plot_surface(x, y, z, rstride=5, cstride=5, color='k', edgecolors='w')
ax2.view_init(0, 0)
ax2.set_xticks([])
plt.show()
```

❶ Необходимы независимые значения θ и φ в интервале (0, 2π), чтобы использовать метод `meshgrid`.

❷ Обратите внимание: можно использовать именованные аргументы, такие как `edgecolors`, для определения стиля многоугольных патчей, создаваемых методом `ax.plot_surface`.

❸ Угол возвышения (взгляда) над плоскостью *xy* равен 36°, азимутальный угол в плоскости *xy* равен 26°.

7.6.2 Линейные графики и точечные диаграммы

Линейные графики и точечные диаграммы в трех измерениях создаются тем же способом, что и для двух измерений: основные вызываемые методы `ax.plot(x, y, z)` и `ax.scatter(x, y, z)` соответственно, где x, y, z – одномерные массивы равной длины. Но для таких графиков возможно добавление только ограниченной аннотации, если не пользоваться методами с расширенным набором функциональных возможностей.

Пример П7.27. В листинге 7.28 показан простой пример трехмерного графика спирали (винтовой линии), которая может представлять, например, свет с круговой поляризацией (см. рис. 7.30).

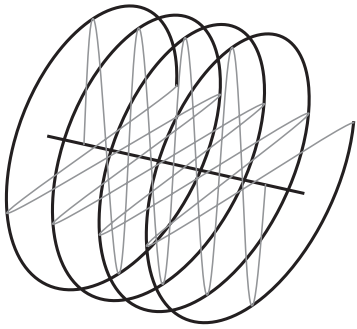


Рис. 7.30. Изображение света с круговой поляризацией в виде спирали на трехмерном графике

Листинг 7.28. Изображение спирали на трехмерном графике

```
# eg7-circular-polarization.py
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

n = 1000
fig = plt.figure()
ax = fig.add_subplot(projection='3d')

# График спирали по оси x.
theta_max = 8 * np.pi
theta = np.linspace(0, theta_max , n)
x = theta
z = np.sin(theta)
y = np.cos(theta)
ax.plot(x, y, z, 'b', lw=2)

# Прямая, проходящая через центр спирали.
ax.plot((-theta_max*0.2, theta_max * 1.2), (0, 0), (0, 0), color='k', lw=2)
# Компоненты sin/cos спирали (например, компоненты электрического и магнитного полей
# электромагнитной волны с круговой поляризацией).
ax.plot(x, y, 0, color='r', lw=1, alpha=0.5)
ax.plot(x, [0]*n, z, color='m', lw=1, alpha=0.5)
```

```
# Удаление плоскостей осей, штриховых меток и подписей.
ax.set_axis_off()
plt.show()
```

7.7 Анимация

В этом разделе предлагается краткое введение в практическое использование класса `FuncAnimation` для создания анимированных графиков и диаграмм в программе (скрипте) на языке Python или в виртуальной блокнотной среде Jupyter Notebook. Библиотека Matplotlib предоставляет функциональные возможности анимации в модуле `animation`, который необходимо явно импортировать перед использованием:

```
import matplotlib.animation as animation
```

7.7.1 Анимация данных на графике

Простая анимированная линия

Класс `FuncAnimation` создает эффект анимации, многократно вызывая предоставленную функцию `func`, которая обновляет объекты, отображаемые в объекте рисунка Matplotlib `Figure` `fig`. Дополнительные аргументы описаны в табл. 7.13.

Таблица 7.13. Аргументы для FuncAnimation

Аргумент	Описание
`fig`	Объект Matplotlib `Figure`, который необходимо анимировать
`func`	Функция, вызываемая для создания каждого кадра анимации посредством манипуляции объектами в рисунке `Figure`
`frames`	Источник объекта, передаваемого в функцию `func` для каждого кадра. Если задано значение `None` (по умолчанию), то передается постоянно увеличивающийся целочисленный индекс. Также можно передавать итерируемые объекты или функцию генератора
`init_func`	Функция, вызываемая для создания пустого кадра анимации. Если определен аргумент `blit=True`, то этот аргумент является обязательным
`fargs`	Любые дополнительные аргументы, передаваемые в функцию `func`
`interval`	Интервалы времени для паузы между кадрами в мс (по умолчанию 200)
`repeat`	Флаг, логическое значение, определяющее, должна ли анимация зацикливаться (повторяться) или нет (по умолчанию `True`)
`blit`	Флаг, логическое значение, определяющее, должна ли использоваться операция комбинирования битовых карт изображений для оптимизации анимации (по умолчанию `False`). Подробнее см. в тексте

Объект рисунка `Figure` и содержащиеся в нем объекты `Axes` должны быть созданы до вызова `FuncAnimation`, и любые ссылки на изображаемые объекты необходимо сохранять, чтобы с этими данными могла работать функция анимации. Например, данные (x, y), изображаемые в объекте `Line2D`, могут быть (пере)настроены с помощью собственного метода `set_data`. Это показано в коде листинга 7.29, где анимируется затухающая синусоидальная кривая.

Пример П7.28. В коде листинга 7.29 выполняется анимация затухающей синусоидальной кривой, которая, например, может представлять затухающий звук при ударе по камертону с постоянной (фиксированной) частотой:

$$M(t) = \sin(2\pi ft)e^{-\alpha t}.$$

Листинг 7.29. Анимация затухающей синусоидальной кривой

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

# Интервал времени для анимации (с), максимальное время анимации (с).
dt, tmax = 0.01, 5
# Частота сигнала (1/с), коэффициент затухания (1/с).
f, alpha = 2.5, 1
# Эти списки будут содержать данные для построения графика.
t, M = [], []

# Изображение пустого графика, но с предварительной установкой ограничений по осям x и y.
fig, ax = plt.subplots()
line, = ax.plot([], [])                                                    ❶
ax.set_xlim(0, tmax)
ax.set_ylim(-1, 1)
ax.set_xlabel('t /s')
ax.set_ylabel('M (arb. units)')

def animate(i):
    """Draw the frame i of the animation."""
    # """Отображение кадра i анимации."""
    global t, M                                                            ❷
    # Добавление этого момента времени, соответствующих ему данных и определение данных
    # для линии, изображаемой на графике.
    _t = i*dt
    t.append(_t)
    M.append(np.sin(2*np.pi*f*_t) * np.exp(-alpha*_t))
    line.set_data(t, M)

# Интервал между кадрами в мс, общее количество используемых кадров.
interval, nframes = 1000 * dt, int(tmax / dt)                              ❸
# Анимация повторяется один раз (установка repeat=False, чтобы анимация не зацикливалась).
ani = animation.FuncAnimation(fig, animate, frames=nframes, repeat=False,
interval=interval)
plt.show()
```

❶ Напомню, что метод `ax.plot` возвращает кортеж объектов `Line2D`, даже если изображается только одна линия. Необходимо сохранить ссылку на этот кортеж, чтобы получить возможность работать с его данными в функции анимации `animate`.

❷ Объявляя списки `t` и `M` глобальными (`global`) объектами, мы получаем возможность изменять их содержимое из тела функции `animate`.

❸ При установке интервала времени между кадрами (паузы) равным (в мс) используемому в этой программе интервалу времени анимация выглядит «происходящей в реальном времени».

Завершающий кадр анимации показан на рис. 7.31.

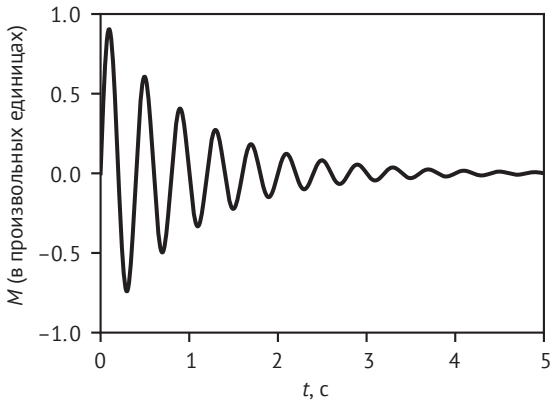


Рис. 7.31. Завершающий кадр анимации затухающей синусоидальной кривой

Комбинирование битовых карт изображений

В примере из предыдущего раздела весь объект линии должен был перерисовываться для каждого кадра. При весьма большом объеме данных или при сложном изображении это может замедлять анимацию. В этом случае может помочь операция комбинирования битовых карт изображения – блиттинг (blitting – Bit Blt, сокращение от bit block transfer), методика из области компьютерной графики, позволяющая в цикле анимации перерисовывать только те части изображения, которые изменяются между кадрами. Это исключает необходимость перерисовки всех отображаемых точек данных.

Для использования операции блиттинга требуется некоторый дополнительный код: необходимо определить метод, передаваемый в аргументе `init_func` объекту `FuncAnimation`. Этот передаваемый метод создает пустой кадр, но возвращает последовательность объектов анимации, которые должны быть перерисованы в каждом кадре. Функция `func`, вызываемая для каждого отображаемого кадра, также должна возвращать последовательность измененных объектов анимации.

Пример П7.29. Код в листинге 7.30 повторяет анимацию из примера П7.28, но использует методику блиттинга и явно передает аргументы в функцию анимации вместо объявления их глобальными объектами в теле этой функции.

Листинг 7.30. Анимация затухающей синусоидальной кривой с использованием blit=True

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

# Интервал времени для анимации (с), максимальное время анимации (с).
dt, tmax = 0.01, 5
# Частота сигнала (1/с), коэффициент затухания (1/с).
f, alpha = 2.5, 1
# Эти списки будут содержать данные для построения графика.
t, M = [], []
```

```
# Изображение пустого графика, но с предварительной установкой ограничений по осям x и y.
fig, ax = plt.subplots()
line, = ax.plot([], [])
ax.set_xlim(0, tmax)
ax.set_ylim(-1, 1)
ax.set_xlabel('t /s')
ax.set_ylabel('M (arb. units)')

def init():
    return line,

def animate(i, t, M):
    """Draw the frame i of the animation."""
    # """Отображение кадра i анимации."""

    # Добавление этого момента времени, соответствующих ему данных и определение данных
    # для линии, изображаемой на графике.
    _t = i*dt
    t.append(_t)
    M.append(np.sin(2*np.pi*f*_t) * np.exp(-alpha*_t))
    line.set_data(t, M)
    return line,

# Интервал между кадрами в мс, общее количество используемых кадров.
interval, nframes = 1000 * dt, int(tmax / dt)
# Анимация повторяется один раз (установка repeat=False, чтобы анимация не зацикливалась).
ani = animation.FuncAnimation(fig, animate, frames=nframes, init_func=init,          (1)
                              fargs=(t, M), repeat=False, interval=interval, blit=True)
plt.show()
```

(1) Любые объекты, присвоенные аргументу `fargs` метода `FuncAnimation`, будут обработаны в функции анимации.

7.7.2 Анимация других объектов Matplotlib

Для анимации других объектов Matplotlib, таких как патчи и надписи аннотации, ссылка на них должна быть сохранена и обработана в каждом кадре. Подобно объектам типа `Line2D`, содержащим метод `set_data`, эти прочие классы также имеют методы-«установщики» (setter-методы) (например, методы `set_center`, `set_radius` для патча `Circle`), использование которых демонстрируется в примере П7.30.

Пример П7.30. Программа в листинге 7.31 выполняет анимацию прыгающего мяча из начального положения $(0, y_0)$ с начальной скоростью $(v_{x0}, 0)$. Положение мяча, хронология траектории и надпись-метка высоты изменяются в каждом кадре.

Здесь в аргументе `frames` для `FuncAnimation` передается функция генератора `get_pos`, которая возвращает следующее положение мяча на каждой итерации. Это положение обрабатывается в функции анимации `animate` вместо целочисленного индекса текущего кадра.

Листинг 7.31. Анимация прыгающего мяча

```
import matplotlib.pyplot as plt
import matplotlib.animation as animation

# Ускорение свободного падения, м/с2.
g = 9.81
# Максимальное значение x траектории мяча, отображаемой на графике.
XMAX = 5
# Коэффициент упругого восстановления при ударе (-v_up/v_down).
cor = 0.65
# Интервал времени для анимации.
dt = 0.005

# Начальное положение и векторы скоростей.
x0, y0 = 0, 4
vx0, vy0 = 1, 0

def get_pos(t=0):
    """A generator yielding the ball ' s position at time t."""
    # """Генератор, определяющий положение мяча в момент времени t."""
    x, y, vx, vy = x0, y0, vx0, vy0
    while x < XMAX:                                                    ❶
        t += dt
        x += vx0 * dt
        y += vy * dt
        vy -= g * dt
        if y < 0:
            # Прыжок.
            y = 0
            vy = -vy * cor
        yield x, y

def init():
    """Initialize the animation figure."""
    # """Инициализация анимируемого изображения."""
    ax.set_xlim(0, XMAX)
    ax.set_ylim(0, y0)
    ax.set_xlabel('$x$ /м')
    ax.set_ylabel('$y$ /м')
    line.set_data(xdata, ydata)
    ball.set_center((x0, y0))
    height_text.set_text(f'Height: {y0:.1f} m')
    return line, ball, height_text

def animate(pos):
    """For each frame , advance the animation to the new position , pos."""
    # """Для каждого кадра анимация перемещается в новое положение pos."""
    x, y = pos
    xdata.append(x)
    ydata.append(y)
    line.set_data(xdata , ydata)
    ball.set_center((x, y))
    height_text.set_text(f'Height: {y:.1f} m')
    return line, ball, height_text
```

```
# Настройка нового рисунка Figure с равным отношением сторон, чтобы мяч выглядел круглым.
fig, ax = plt.subplots()
ax.set_aspect('equal')

# Это объекты, которые необходимо постоянно отслеживать.
line, = ax.plot([], [], lw=2)
ball = plt.Circle((x0, y0), 0.08)
height_text = ax.text(XMAX*0.5, y0*0.8, f'Height: {y0:.1f} m')
ax.add_patch(ball)
xdata, ydata = [], []

interval = 1000*dt
ani = animation.FuncAnimation(fig, animate, get_pos, blit=True,
                              interval=interval, repeat=False, init_func=init)
plt.show()
```

❶ Функция генератора будет сохранять создаваемый вектор положения мяча (`x, y`) до тех пор, пока координата *x* мяча не достигнет значения `XMAX`. Затем, когда генератор исчерпается и выдаст значение `None`, анимация завершится.

Завершающий кадр анимации показан на рис. 7.32.

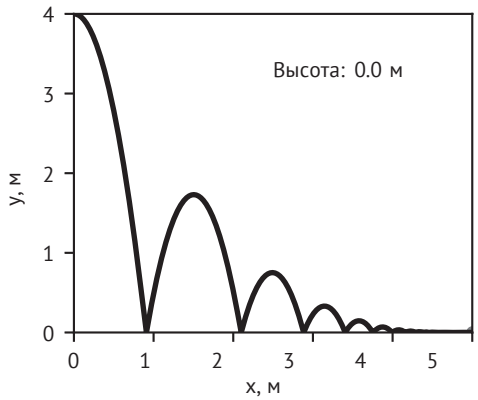


Рис. 7.32. Завершающий кадр анимации прыгающего мяча

7.7.3 Упражнения

Задачи

37.7.1. Использовать класс `FuncAnimation` из библиотеки Matplotlib для создания анимации качающегося маятника с начальным максимальным углом отклонения от вертикали и нулевой начальной скоростью. Выполнить численное интегрирование уравнения движения и повторить анимацию после завершения одного полного периода колебаний.

37.7.2. Изменить код примера П7.24 для создания анимации распространения температуры в металлической пластине по времени.

37.7.3. Лаборатория реактивного движения (JPL) НАСА сопровождает веб-сервис и базу данных HORIZONS, которую можно использовать для вычисления астрономических таблиц (эфемеридов – траекторий объектов Солнечной системы по времени). Предварительно подобранные данные с этого ресурса можно скачать здесь: https://scipython.com/eg/bas. Использовать эти данные для создания анимации траектории исследовательского космического аппарата Вояджер 2 с момента его запуска в августе 1977 г. до конца 1999 г. В этот период включено несколько гравитационных промежуточных маневров, когда аппарат проходил рядом с крупными планетами. Необходимо рассматривать только координаты (X, Y) значимых космических тел.

Глава 8

Библиотека SciPy

SciPy – это библиотека модулей языка Python для научных расчетов, которая предоставляет более специализированные функциональные возможности, чем общие структуры данных и математические алгоритмы библиотеки NumPy. Например, библиотека SciPy содержит модули для вычисления специальных функций, часто применяемых в научной и инженерной деятельности для оптимизации, интегрирования, интерполяции и обработки изображений. Как и в библиотеке NumPy, многие внутренние алгоритмы библиотеки SciPy выполняются как предварительно скомпилированный код на языке C, обеспечивая высокую скорость. Кроме того, как NumPy и сам Python, библиотека SciPy является свободно распространяемым программным обеспечением.

Для практического использования подпрограмм библиотеки SciPy необходимо ознакомиться с немного новым синтаксисом, и в этой главе основное внимание сосредоточено на примерах практического применения этой библиотеки в коротких программах, связанных с научными и инженерными расчетами.

8.1 Физические константы и специальные функции

Полезный пакет `scipy.constants` содержит принятые по международным стандартам значения и допустимые погрешности физических констант. Кроме того, пакет `scipy.special` предоставляет множество алгоритмов для вычисления функций, часто используемых в научных исследованиях, математическом анализе и инженерных расчетах, в том числе следующие:

- функции Эйри;
- эллиптические функции и интегралы;
- функции Бесселя, их нули, производные и интегралы;
- сферические функции Бесселя;
- разнообразные статистические функции и распределения;
- гамма- и бета-функции;
- функция распределения (вероятности) ошибок;
- интегралы Френеля;
- функции Лежандра и присоединенные функции Лежандра;
- разнообразные ортогональные многочлены;
- гипергеометрические функции;

- функции параболического цилиндра;
- функции Матьё;
- сфероидальные функции.

Все функции подробно описаны в документации (https://docs.scipy.org/doc/scipy/reference/special.html), а в этом разделе основное внимание сосредоточено на нескольких характерных примерах их применения.

Большинство перечисленных выше специальных функций в библиотеке SciPy реализованы как универсальные функции, т. е. они поддерживают бродкастинг и векторизацию (автоматический проход по массиву в цикле), поэтому вполне ожидаемо корректно работают с массивами NumPy.

8.1.1 Физические константы

Библиотека SciPy содержит рекомендованные международным Комитетом по данным для науки и техники CODATA в выпуске 2018 г. значения многих физических констант (https://physics.nist.gov/cuu/Constants/). Константы вместе с единицами их измерения и допустимыми погрешностями хранятся в словаре `scipy.constants.physical_constants`, в котором ключом является строка идентификации. Например:

```
In [x]: import scipy.constants as pc
In [x]: pc.physical_constants['Avogadro constant']
Out[x]: (6.022140857e+23, 'mol^-1', 7400000000000000.0)
```

Удобные методы `value`, `unit` и `precision` извлекают соответствующие свойства констант:

```
In [x]: pc.value('electron mass')
Out[x]: 9.1093837015e-31
In [x]: pc.unit('electron mass')
Out[x]: 'kg'
In [x]: pc.precision('electron mass')
3.0737534961217373e-10
```

Для удобства требуемое значение физической константы обычно присваивается переменной в начале программы, например:

```
In [x]: muB = pc.value('Bohr magneton')
```

Полный список констант и их названий приведен в официальной документации SciPy (https://docs.scipy.org/doc/scipy/reference/constants.html), но в табл. 8.1 перечислены наиболее важные значения. Значения некоторых особенно важных констант присвоены переменным в модуле `scipy.constants` (в единицах СИ), поэтому их можно импортировать напрямую:

```
In [x]: from scipy.constants import c, R, k
In [x]: c, R, k   # Скорость света, универсальная газовая постоянная, постоянная Больцмана.
Out[x]: (299792458.0, 8.314462618, 1.380649e-23)
```

Таблица 8.1. Некоторые физические константы из модуля scipy.constants

Строка идентификации константы	Переменная	Значение	Единицы измерения
'atomic mass constant'	m_u	1.6605390666e-27	кг
'Avogadro constant'	N_A	6.02214076e+23	1/моль
'Bohr magneton'		9.2740100783e-24	Дж/Тл
'Bohr radius'		5.29177210903e-11	м
'Boltzmann constant'	k	1.380649e-23	Дж/К
'electron mass'	m_e	9.1093837015e-31	кг
'elementary charge'	e	1.602176634e-19	Кл
'Faraday constant'		96485.33212	Кл/моль
'fine-structure constant'	alpha	0.0072973525693	
'molar gas constant'	R	8.314462618	Дж / (К·моль)
'neutron mass'	m_n	1.67492749804e-27	кг
'Newtonian constant of gravitation'	G	6.6743e-11	$м^3$ / ($с^2$ · кг)
'Planck constant'	h	6.62607015e-34	Дж · с
'proton mass'	m_p	1.67262192369e-27	кг
'Rydberg constant'	Rydberg	10973731.56816	1/м
'speed of light in vacuum'	c	299792458.0	м/с

Для приведенного выше примера имена переменных приведены в табл. 8.1. Возможно, более удобным вам покажется использование значений из `scipy.constants`, но при этом следует помнить о том, что после публикации новых выпусков CODATA значений констант этот пакет может обновляться – это значит, что ваш код может выдавать немного отличающиеся результаты с различными версиями SciPy. Значения в табл. 8.1 взяты из версии 1.4 SciPy, в которую включены новые определения основных единиц системы СИ, принятые в 2019 г.

Кроме того, в пакете `scipy.constants` определены полезные коэффициенты и методы преобразования, которые также включают представления префиксов системы СИ. Например:

```
In [x]: import scipy.constants as pc
In [x]: pc.atm
Out[x]: 101325.0              # 1 атм в Па.
In [x]: pc.bar
Out[x]: 100000.0              # 1 бар в Па.
In [x]: pc.torr
Out[x]: 133.32236842105263    # 1 торр (мм рт.ст.) в Па.
In [x]: pc.zero_Celsius
Out[x]: 273.15                # 0 градС в К.
In [x]: pc.micro              # А также нано, пико, мега, гига и т. д.
Out[x]: 1e-06
```

Пример П8.1. Здесь используется словарь `scipy.constants.physical_constants` для определения констант, которые считаются наименее точными. Для этого требуются относительные погрешности значений констант. Код в листинге 8.1 использует структурированный массив для вычисления относительных погрешностей и выводит константы, которые наименее точно определены.

Листинг 8.1. Наименее точно определенные физические константы

```
import numpy as np

from scipy.constants import physical_constants

def make_record(k, v):
    """
    Return the record for this constant from the key and value of its entry
    in the physical_constants dictionary.
    """
    # """Возвращает запись для этой константы, сформированную из ключа и значения
    #    соответствующего элемента словаря физических констант."""
    name = k
    val, units, abs_unc = v
    # Вычисление относительной погрешности в ppm (в частях на миллион).
    rel_unc = abs_unc / abs(val) * 1.e6
    return name, val, units, abs_unc, rel_unc

dtype = [('name', 'S50'), ('val', 'f8'), ('units', 'S20'),
         ('abs_unc', 'f8'), ('rel_unc', 'f8')]
constants = np.array([make_record(k, v) for k, v in physical_constants.items()],
                     dtype=dtype)
constants.sort(order='rel_unc')

# Список из 10 констант с наибольшими относительными погрешностями.
for rec in constants[-10:]:
    print('{:.0f} ppm: {:s} = {:g} {:s}'.format(rec['rel_unc'],
              rec['name'].decode(), rec['val'], rec['units'].decode()))
```

Вывод результата выполнения этой программы:

```
90 ppm: tau Compton wavelength over 2 pi = 1.11056e-16 m
90 ppm: tau mass energy equivalent in MeV = 1776.82 MeV
193 ppm: W to Z mass ratio = 0.88153
348 ppm: deuteron rms charge radius = 2.12799e-15 m
428 ppm: proton mag. shielding correction = 2.5689e-05
428 ppm: proton magn. shielding correction = 2.5689e-05
829 ppm: shielding difference of t and p in HT = 2.414e-08
990 ppm: shielding difference of d and p in HD = 2.02e-08
1346 ppm: weak mixing angle = 0.2229
2258 ppm: proton rms charge radius = 8.414e-16 m
```

8.1.2 Функции Эйри и Бесселя

Функции Эйри Ai(x) и Bi(x) – это линейно независимые решения дифференциального уравнения Эйри $y'' - xy = 0$, которые используются в квантовой механике, оптике, электродинамике и других областях физики. Функции `(Ai, Bi)` и их производные `(Aip, Bip)` возвращает функция `scipy.special.airy`. Единственный обязательный аргумент x может быть комплексным числом или массивом NumPy:

```
In [x]: Ai, Aip, Bi, Bip = airy(0)
In [x]: Ai, Aip, Bi, Bip
(0.35502805388781722, -0.25881940379280682, 0.61492662744600068, 0.44828835735382638)
```

Первые `nt` нулей функций Эйри и их производные возвращает функция `scipy.special.ai_zeros(nt)`:

```
In [x]: a, ap, ai, aip = ai_zeros(2)      # Массивы для первых двух нулей функции Ai.
In [x]: a[1], ap[1], ai[1], aip[1]        # Содержимое второго нуля:
Out[x]: ( -4.0879494441309721, -3.248197582179837, -0.41901547803256406,
          -0.80311136965486463)
In [x]: airy(a[1])[0]                     # Функция Ai(a) должна = 0.
Out[x]: 1.2774882441379295e-15            # Достаточно близко к нулю.
In [x]: airy(ap[1])[1]                    # Производная Aip(ap) должна = 0.
Out[x]: -3.2322209157744908e-16           # Достаточно близко к нулю.
In [x]: airy(ap[1])[0]                    # Ai(ap) возвращается как ai выше.
Out[x]: -0.41901547803256395
In [x]: airy(a[1])[1]                     # Aip(a) возвращается как aip выше.
Out[x]: -0.80311136965486396
```

Пример П8.2. Рассмотрим частицу массой m, перемещающуюся в постоянном гравитационном поле, таком, что потенциальная энергия частицы на высоте z над поверхностью равна mgz. Если частица совершает упругие прыжки по поверхности, то классическая плотность вероятности, соответствующая положению частицы, записывается формулой

$$P_{cl}(z) = \frac{1}{\sqrt{z_{\max}(z_{\max} - z)}},$$

где $z_{\max}$ – максимальная высота, достигаемая частицей.

Поведение по законам квантовой механики этой системы можно описать в виде решения независимого от времени (стационарного) уравнения Шрёдингера:

$$-\frac{\hbar}{2m}\frac{\mathrm{d}^2\psi}{\mathrm{d}z^2} + mgz\psi = E\psi,$$

которое упрощается при изменении масштаба координат $q = z/\alpha$, где $\alpha = (\hbar / 2m^2 g)^{1/3}$:

$$\frac{\mathrm{d}^2\psi}{\mathrm{d}q^2} - (q - q_E)\psi = 0, \text{ где } q_E = \frac{E}{mg\alpha}.$$

Решениями этого дифференциального уравнения являются функции Эйри. Граничное условие $\psi(z) \to 0$ как $z \to \infty$ в частном случае приводит к следующей форме записи:

$$\psi(q) = N_E \mathrm{Ai}(q - q_E),$$

где N_E – константа нормализации.

Второе граничное условие $\psi(q = 0) = 0$ приводит к квантованию в понятиях квантового числа $n = 1, 2, 3, \ldots$, с масштабируемыми значениями энергии q_E, определяемыми по нулям функции Эйри: $\mathrm{Ai}(-q_E) = 0$.

Программа в листинге 8.2 вычисляет и строит графики классических и квантовых распределений вероятностей $P_{cl}(z)$ и $|\psi(z)|^2$ при $n = 1$ и $n = 16$ (см. рис. 8.1).

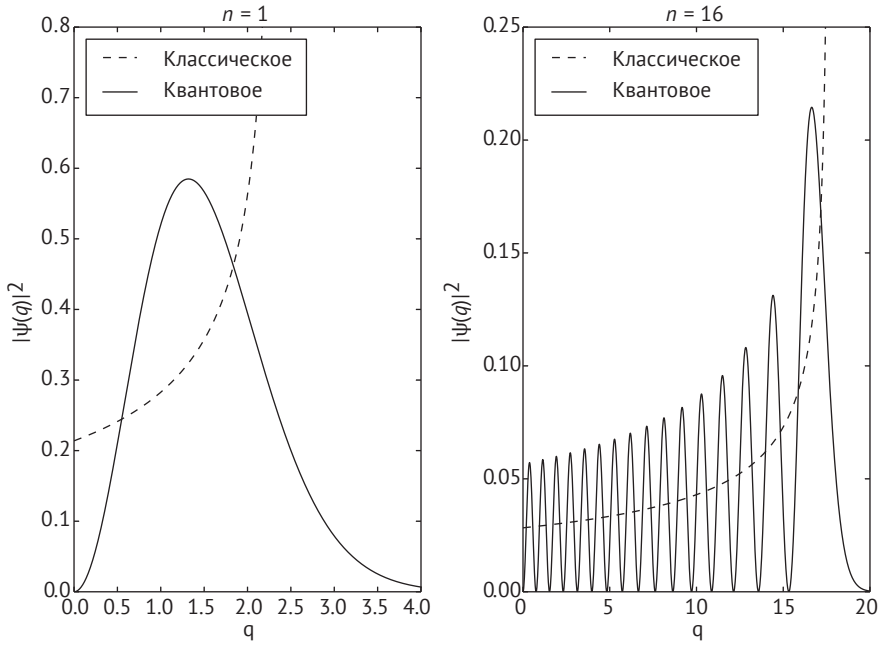


Рис. 8.1. Сравнение классических и квантовых распределений вероятностей для частицы, перемещающейся в постоянном гравитационном поле на двух различных энергетических уровнях

Листинг 8.2. Плотности распределения вероятностей для частицы в однородном гравитационном поле

```
# eg8-qm-gravfield.py
import numpy as np
from scipy.special import airy , ai_zeros
import matplotlib.pyplot as plt
```

```
nmax = 16

# Поиск первых nmax нулей функции Ai(x).
a, _, _, _ = ai_zeros(nmax)                                                    ❶
# Действительное граничное условие Ai(-qE) = 0 при q = 0, так что:
qE = -a

def prob_qm(n):
    """
    Return the quantum mechanical probability density for a particle moving
    in a uniform gravitational field.
    Возвращает плотность распределения вероятностей по правилам квантовой механики для
    частицы, перемещающейся в однородном гравитационном поле.
    """
    # Волновая функция квантовой механики пропорциональна функции Эйри Ai(q - qE),
    # где значение qE, соответствующее квантовому числу n, индексируется по n - 1.
    psi, _, _, _ = airy(q-qE[n-1])                                             ❷
    # Возвращает плотность распределения вероятностей после приближенной, но эффективной
    # нормализации.
    P = psi**2
    return P / (sum(P) * dq)                                                   ❸

def prob_cl(n):
    """
    Return the classical probability density for a particle bouncing
    elastically in a uniform gravitational field.
    Возвращает классическую плотность распределения вероятностей для частицы, перемещающейся
    упругими скачками в однородном гравитационном поле.
    """
    # Классическая плотность распределения вероятностей уже нормализована.
    return 0.5/np.sqrt(qE[n-1]*(qE[n-1]-q))

# Основное (стандартное) состояние n = 1.
q, dq = np.linspace(0, 4, 1000, retstep=True)
plt.plot(q, prob_cl(1), label='Classical')
plt.plot(q, prob_qm(1), label='Quantum')
plt.ylim(0, 0.8)
plt.legend()
plt.show()

# Возбужденное состояние n = 16.
q, dq = np.linspace(0, 20, 1000, retstep=True)
plt.plot(q, prob_cl(16), label='Classical')
plt.plot(q, prob_qm(16), label='Quantum')
plt.ylim(0, 0.25)
plt.legend(loc='upper left')
plt.show()
```

❶ Здесь используется функция `scipy.special.ai_zeros` для получения собственных значений $n = 1$ и $n = 16$.

❷ Функция `scipy.special.airy` находит соответствующие волновые функции, следовательно, и плотности распределения вероятностей.

❸ Для наглядности нормализация выполняется приблизительно с помощью весьма упрощенного численного интегрирования.

Функции Бесселя – еще одна важная группа функций со множеством приложений в физике и инженерной деятельности. Библиотека SciPy предоставляет несколько методов для вычисления функций Бесселя, их производных и нулей:

- `jn(v, x)` и `jv(v, x)` возвращают функцию Бесселя первого рода от `x` порядка `v` ($J_v(x)$). Значение `v` может быть действительным или целым числом;
- `yn(n, x)` и `yv(v, x)` возвращают функцию Бесселя второго рода от `x` целочисленного порядка `n` ($Y_n(x)$) и действительного порядка `v` ($Y_v(x)$) соответственно;
- `in(n, x)` и `iv(v, x)` возвращают модифицированную функцию Бесселя первого рода от `x` целочисленного порядка `n` ($I_n(x)$) и действительного порядка `v` ($I_v(x)$) соответственно;
- `kn(n, x)` и `kv(v, x)` возвращают модифицированную функцию Бесселя второго рода от `x` целочисленного порядка `n` ($K_n(x)$) и действительного порядка `v` ($K_v(x)$) соответственно;
- функции `jvp(v, x)`, `yvp(v, x)`, `ivp(v, x)` и `kvp(v, x)` возвращают производные перечисленных выше функций. По умолчанию возвращается первая производная. Чтобы получить n-ю производную, необходимо определить дополнительный аргумент `n`;
- некоторые функции можно использовать для получения нулей функций Бесселя. Вероятно, самыми полезными являются функции `jn_zeros(n, nt)`, `jnp_zeros(n, nt)`, `yn_zeros(n, nt)` и `ynp_zeros(n, nt)`, которые возвращают первые `nt` нулей $J_n(x)$, $J_n'(x)$, $Y_n(x)$ и $Y_n'(x)$.

Пример П8.3. Вибрации тонкой круговой мембраны, натянутой на жесткий круговой каркас (как на барабане (ударном инструменте)), можно описать как собственную (нормальную) форму колебаний, записанную с использованием функций Бесселя:

$$z(r, \theta; t) = AJ_n(kr) \sin n\theta \cos kvt,$$

где (r, θ) описывает положение в полярных координатах с началом в центре мембраны, t – время, v – константа, зависящая от силы натяжения и плотности поверхности барабана. Формы колебаний помечаются целочисленными значениями $n = 0, 1, \ldots$ и $m = 1, 2, 3, \ldots$, где k – m-й ноль функции J_n.

Программа в листинге 8.3 создает график колебаний мембраны в $n = 3$, $m = 2$ собственной (нормальной) форме в момент времени $t = 0$ (см. рис. 8.2).

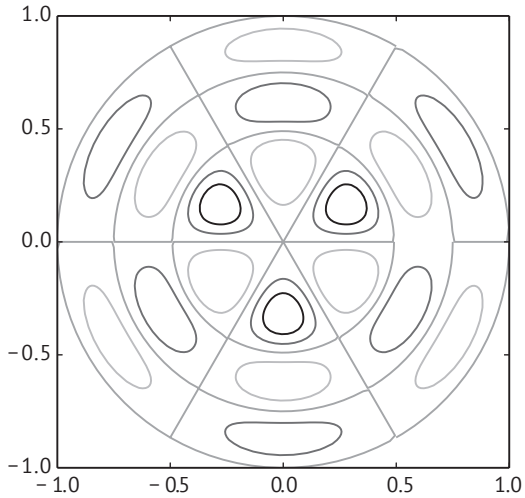


Рис. 8.2. Собственная (нормальная) форма $n = 3$, $m = 2$ вибрации круговой мембраны барабана

Листинг 8.3. Собственные (нормальные) формы вибрации круговой мембраны барабана

```
# eg8-drum-normal-modes.py
import numpy as np
from scipy.special import jn, jn_zeros
import matplotlib.pyplot as plt

# Разрешить вычисления до предельного значения m = mmax.
mmax = 5

def displacement(n, m, r, theta):
    """
    Calculate the displacement of the drum membrane at (r, theta; t = 0)
    in the normal mode described by integers n >= 0, 0 < m <= mmax.
    Вычисление колебаний мембраны барабана с параметрами (r, theta; t = 0) в собственной
    (нормальной) форме, описываемой целочисленными значениями n >= 0, 0 < m <= mmax.
    """
    # Выбор m-го нуля функции Бесселя Jn.
    k = jn_zeros(n, mmax+1)[m]
    return np.sin(n*theta) * jn(n, r*k)

# Положения поверхности мембраны барабана определяются в полярных координатах.
r = np.linspace(0, 1, 100)
theta = np.linspace(0, 2 * np.pi, 100)

# Создание массивов декартовых координат (x, y)…
x = np.array([rr*np.cos(theta) for rr in r])
y = np.array([rr*np.sin(theta) for rr in r])
# … и вертикальных перемещений (z) для требуемой собственной формы колебаний
# в момент времени t = 0.
n, m = 3, 2
z = np.array([displacement(n, m, rr, theta) for rr in r])
```

```
plt.contour(x, y, z)
plt.show()
```

Пример П8.4. В 1953 г. Розалинд Франклин (Rosalind Franklin) опубликовала свою главную работу[110], в которой содержалась дифракционная картина (диаграмма) рентгеновских лучей при анализе тимусной ДНК (ДНК вилочковой железы), на которой было показана X-форма дифракционных пятен, свидетельствующая о спиральной структуре ДНК.

Дифракционная картина однородной непрерывной спирали состоит из последовательности «слоевых линий» с зазором $1/p$ в обратном пространстве, где p – шаг спирали (высота одного полного витка спирали, измеряемая параллельно ее оси). Распределение интенсивности вдоль n-й линии уровня пропорционально квадрату n-й функции Бесселя $J_n(2\pi rR)$, где r – радиус спирали, R – радиальная координата во взаимно обратном пространстве.

Рассмотрим дифракционную картину для спирали с p = 34 Å и r = 10 Å. Код в листинге 8.4 создает изображение дифракционной диаграммы рентгеновских лучей для этой спирали в формате SVG (см. рис. 8.3).

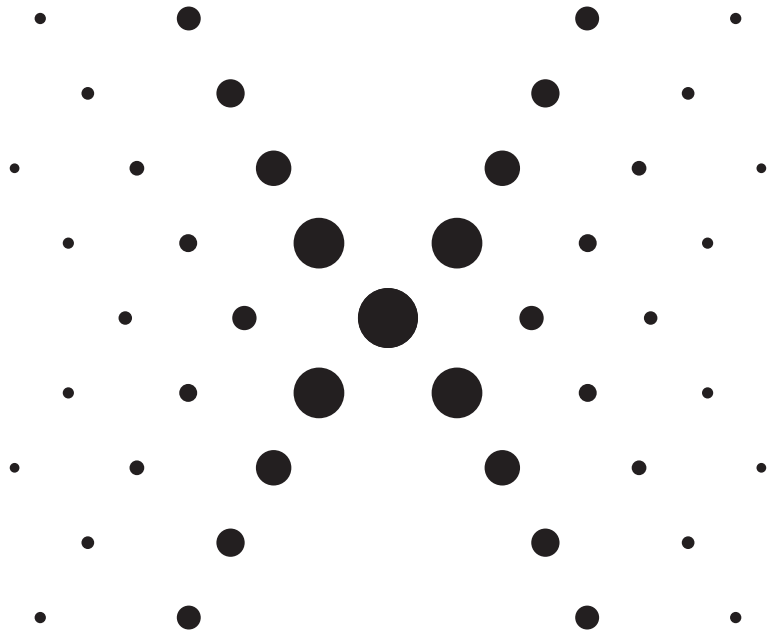


Рис. 8.3. Дифракционная картина (диаграмма) для однородной непрерывной спирали

[110] *R. E. Franklin and R. G. Gosling*. Nature 171, 740 (1953).

Листинг 8.4. Генерация изображения дифракционной диаграммы для однородной непрерывной спирали

```
# eg8-dna-diffraction.py
import numpy as np
from scipy.special import jn
import matplotlib.pyplot as plt

# Вертикальный диапазон дифракционной диаграммы: изображение nlayer линий уровней выше
# и ниже центральной горизонтали.
nlayers = 5
ymin, ymax = -nlayers, nlayers

# Горизонтальный диапазон дифракционной диаграммы x = 2pi.r.R.
xmin, xmax = -10, 10
npts = 4000
x = np.linspace(xmin, xmax, npts)

# Дифракционная картина вдоль каждой линии уровня: |Jn(x)|^2
# для n = 0, 1, ..., nlayers - 1.
layers = np.array([jn(i, x)**2 for i in range(nlayers)])                    ❶

# Получение индексов максимумов на каждом уровне.
maxi = [(np.diff(np.sign(np.diff(layers[i,:]))) < 0).nonzero()[0] + 1       ❷
                                                    for i in range(nlayers)]

# Создание изображения в формате SVG с использованием кругов различных радиусов
# для дифракционных пятен.
svg_name='eg8-dna-diffraction.svg'
canvas_width = canvas_height = 500
fo = open(svg_name , 'w')
print(""" <?xml version="1.0" encoding="utf -8"?>
          <svg xmlns="www.w3.org/2000/svg"
               xmlns:xlink="www.w3.org/1999/xlink"
               width="{}" height="{}" style="background: {}">""".format(
                    canvas_width , canvas_height , '#ffffff'), file=fo)

def svg_circle(r, cx, cy):
    """ Return the SVG mark up for a circle of radius r centered at (cx, cy). """
    """ Возвращает параметры изображения SVG для круга радиусом r с центром в (cx, cy). """
    return r'<circle r="{}" cx="{}" cy="{}"/>'.format(r, cx, cy)

# Для каждого дифракционного пятна на каждом уровне на холсте изображается круг.
# Радиус круга - промасштабированное максимальное значение интенсивности дифракции
# с верхним пороговым значением spot_max_radius, так как более близкие к центру
# пятна имеют весьма высокую интенсивность.
spot_scaling, spot_max_radius = 50, 20
for i in range(nlayers):
    for j in maxi[i]:
        sx = (x[j] - xmin)/(xmax - xmin) * canvas_width                     ❸
        sy = (i - ymin)/(ymax - ymin) * canvas_height
        spot_radius = min(layers[i, j]*spot_scaling, spot_max_radius)
        print(svg_circle(spot_radius, sx, sy), file=fo)
        if i:
            # Диаграмма симметрична относительно центральной горизонтали:
            # поэтому дублируются уровни с i > 0.
            sy = canvas_height - sy
            print(svg_circle(spot_radius, sx, sy), file=fo)

print(r'</svg>', file=fo)
```

❶ Двумерный массив `layers` содержит значения интенсивности дифракции на каждой линии уровня, вычисляемой как квадрат функции Бесселя.
❷ Для построения дифракционной диаграммы необходимо найти индексы максимальных значений в массиве `layers`: эта строка кода находит такие максимумы, определяя позиции разностей между соседними элементами, где происходит переход от положительных разностей к отрицательным.
❸ Отображение координат `(x, y)` во взаимно обратном пространстве дифракционной диаграммы в координаты холста `(sx, sy)`.

8.1.3 Гамма- и бета-функции; эллиптические интегралы

Гамма-функция определяется несобственным интегралом

$$\Gamma(x) = \int_0^\infty t^{x-1} e^{-t} \mathrm{d}t,$$

для действительных значений $x > 0$ и распространяется на отрицательные значения x и комплексные числа с помощью аналитического продолжения. Гамма-функция часто встречается в задачах интегрирования, комбинаторике, а также в выражениях для других специальных функций.

Гамма-функцию и ее натуральный логарифм возвращают методы `gamma(x)` и `gammaln(x)`. Также существуют методы для вычисления неполных (незавершенных) гамма-функций (получаемых заменой нижнего или верхнего предела приведенного выше интеграла на параметр `a`) и обратных им функций. Здесь эти функции не рассматриваются подробно.

Пример П8.5. Гамма-функция связана с факториалом формулой $\Gamma(x) = (x - 1)!$, и обе формулы графически изображаются с помощью кода в листинге 8.5 (см. рис. 8.4). Обратите внимание: функция $\Gamma(x)$ не определена для отрицательных целых x, при которых на графике возникают разрывы.

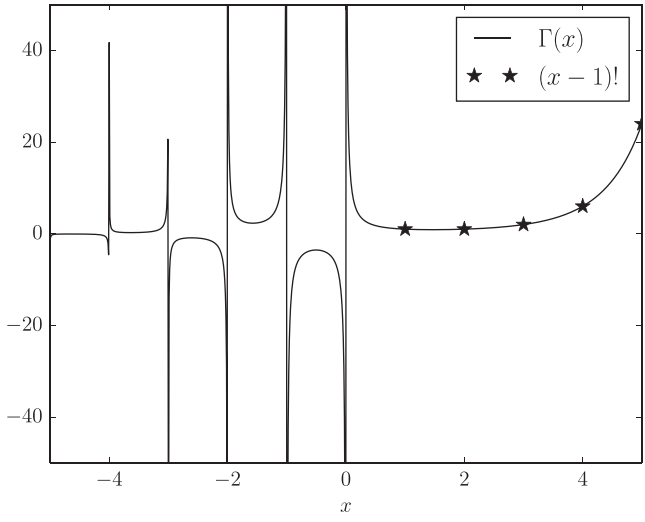


Рис. 8.4. Гамма-функция на действительной числовой оси $\Gamma(x)$ и $(x - 1)!$ при целых $x > 0$

Листинг 8.5. Гамма-функция на действительной числовой оси

```
# eg3-gamma.py
import numpy as np
from scipy.special import gamma
import matplotlib.pyplot as plt

# Гамма-функция.
ax = plt.linspace(-5, 5, 1000)
plt.plot(ax, gamma(ax), ls='-', c='k', label='$\Gamma(x)$')

# Факториал (x - 1)! для x = 1, 2, ..., 6.
ax2 = plt.linspace(1, 6, 6)
xm1fac = np.array([1, 1, 2, 6, 24, 120])
plt.plot(ax2, xm1fac, marker='*', markersize=12, markeredgecolor='r',
         ls='', c='r', label='$(x-1)!$')

plt.ylim(-50, 50)
plt.xlim(-5, 5)
plt.xlabel('$x$')
plt.legend()
plt.show()
```

Бета-функция представлена определенным интегралом

$$\mathrm{B}(a, b) = \int_0^1 t^{a-1}(1-t)^{b-1}dt,\ a > 0, b > 0.$$

Бета-функция тесно связана с гамма-функцией: $\mathrm{B}(a,b) = \Gamma(a)\Gamma(b)/\ \Gamma(a+b)$. В модуле `scipy.special` методы `beta(a, b)` и `betaln(a, b)` возвращают бета-функцию и ее натуральный логарифм соответственно. Как и для гамма-функции, существует неполная (незавершенная) бета-функция $\mathrm{B}(a, b; x)$, получаемая заменой верхнего предела интеграла на x. Методы `betainc(a, b, x)` и `betaincinv(a, b, y)` возвращают эту функцию и обратную ей функцию.

Пример П8.6. Точное определение классической механики для маятника слишком сложно, а уравнения движения в учебных материалах (для начинающих) обычно решаются только для малых перемещений относительно положения равновесия. В этом случае период колебаний $T \approx 2\pi\sqrt{(L/g)}$, а колебания гармонические.

Обобщенное решение требует применения эллиптических интегралов, но при рассмотрении особого случая колебаний маятника с амплитудой 180° (т. е. ±90° относительно его положения равновесия) получаем следующее выражение для периода колебаний:

$$T = 2\sqrt{\frac{2l}{g}}\int_0^{\pi/2}\frac{\mathrm{d}\theta}{\sqrt{\cos\theta}}.$$

При подстановке $x = \sin^2\theta$ этот интеграл преобразовывается в бета-функцию:

$$\int_0^{\pi/2}\frac{\mathrm{d}\theta}{\sqrt{\cos\theta}}=\frac{1}{2}\int_0^1 x^{-1/2}(1-x)^{-3/4}\mathrm{d}x=\frac{1}{2}B\left(\frac{1}{2},\frac{1}{4}\right).$$

Следовательно:

$$T=\sqrt{2}B\left(\frac{1}{2},\frac{1}{4}\right)\sqrt{\frac{l}{g}}.$$

Для определения периода колебаний маятника в единицах $\sqrt{l/g}$:

```
In [x]: import numpy as np
In [x]: from scipy.special import beta
In [x]: np.sqrt(2) * beta(0.5, 0.25)
7.4162987092054875
```

(Сравните с гармонической аппроксимацией 2π = 6.283185.)

Группа эллиптических интегралов и связанных с ними функций образует важный класс математических объектов, которые уже исследованы во всех подробностях. Они применяются в геометрии, криптографии, математическом анализе и во многих областях физики. Полные эллиптические интегралы первого и второго рода $K(m)$ и $E(m)$ определены при $0 \leqslant m \leqslant 1$ в следующем виде:

$$K(m)=\int_0^{\pi/2}\frac{\mathrm{d}\theta}{\sqrt{1-m\sin^2\theta}},$$
$$E(m)=\int_0^{\pi/2}\sqrt{1-m\sin^2\theta}\,\mathrm{d}\theta.$$

Их значения для параметра m возвращают методы `ellipk(m)` и `ellipe(m)`. Неполные эллиптические интегралы (определяемые заменой верхнего предела π/2 на переменную φ) возвращают методы `ellipkinc(phi, m)` и `ellipeinc(phi, m)` соответственно[111]:

$$K(\phi,m)=\int_0^{\phi}\frac{\mathrm{d}\theta}{\sqrt{1-m\sin^2\theta}},$$
$$E(\phi,m)=\int_0^{\phi}\sqrt{1-m\sin^2\theta}\,\mathrm{d}\theta.$$

[111] Необходимо весьма внимательно следить за формой записи эллиптических интегралов. Во многих источниках используется форма записи $F(\varphi, m)$ вместо $K(\varphi, m)$ для интеграла первого рода, в определении переставляются местами аргументы (т. е. $F(m, \varphi)$ или используется параметр k^2 вместо m:

$$F(\phi,k)=F(\phi\,|\,k^2)=\int_0^{\phi}\frac{\mathrm{d}\theta}{\sqrt{1-k^2\sin^2\theta}},$$
$$E(\phi,k)=E(\phi\,|\,k^2)=\int_0^{\phi}\sqrt{1-k^2\sin^2\theta}\,\mathrm{d}\theta.$$

Пример П8.7. Задача поиска длины дуги эллипса является причиной, по которой эллиптические интегралы получили свое название. Параметрическое уравнение эллипса с использованием длины главной полуоси *a* и малой полуоси *b* записывается в виде

$$x = a \sin \varphi$$
$$y = b \cos \varphi.$$

Элементарная (бесконечно малая) длина вдоль периметра эллипса вычисляется по формуле:

$$\mathrm{d}s = \sqrt{\mathrm{d}x^2 + \mathrm{d}y^2} = \sqrt{a^2 \cos^2 \phi + b^2 \sin^2 \phi}\,\mathrm{d}\,\phi = a\sqrt{1 - e^2 \sin^2 \phi}\,\mathrm{d}\,\phi,$$

где $e = \sqrt{(1 - b^2 / a^2}$ – *эксцентриситет*. Следовательно, длину дуги можно записать с использованием неполных эллиптических интегралов второго рода:

$$\int \mathrm{d}s = a\int_{\phi_1}^{\phi_2} \sqrt{1 - e^2 \sin^2 \phi}\,\mathrm{d}\phi = a\left[E(e;\phi_2) - E(e;\phi_1)\right].$$

Орбита Земли представляет собой эллипс с длиной главной полуоси 149 598 261 км и эксцентриситетом 0.01671123. Вычислим расстояние, пройденное Землей за один оборот по орбите, и сравним его с вычисленным при допущении круговой орбиты с радиусом 1 а. е. (астрономическая единица) ≡ 149 597 870.7 км.

Уравнение периметра эллипса можно записать с использованием приведенного выше выражения при $\varphi_1 = 0$, $\varphi_2 = 2\pi$:

$$P = a[E(e, 2\pi) - E(e, 0)] = 4aE(e),$$

поскольку полный периметр равен четырем четвертям (периметра) эллипса, а четверть периметра эллипса можно выразить через полный эллиптический интеграл второго рода. Получим:

```
In [x]: import numpy as np
In [x]: from scipy.special import ellipe
In [x]: a, e = 149_598_261, 0.01671123 # Главная полуось (км), эксцентриситет.
In [x]: pe = 4 * a * ellipe(e*e)
In [x]: print(pe)
939887967.974                          # "Точный" результат.
In [x]: AU = 149_597_870.7             # Средний радиус орбиты, км.
In [x]: pc = 2 * np.pi * AU
In [x]: print(pc)
939951143.1675915                      # Предполагается круговая орбита.
In [x]: (pc - pe) / pe * 100
0.0067215663638305143
```

Таким образом, процент погрешности при вычислении периметра при предположении, что орбита является окружностью, равен приблизительно 0.0067 %.

8.1.4 Функция распределения вероятности ошибок и связанные с ней функции

Функция распределения вероятности ошибок, определяемая формулой

$$\operatorname{erf}(z) = \frac{2}{\sqrt{\pi}} \int_0^z e^{-t^2} \mathrm{d}t,$$

для действительного или комплексного z не имеет простого выражения в конечном виде (в замкнутой форме), поэтому должна определяться исключительно численными методами. Модуль `scipy.special` содержит несколько методов, связанных с вычислением функции распределения вероятности ошибок:

- `erf(z)` – функция распределения вероятности ошибок (или просто функция ошибок);
- `erfc(z)` – дополнительная функция ошибок, $\operatorname{erfc}(z) = 1 - \operatorname{erf}(z)$. Для больших значений z использование этой функции дает более точный результат, чем простое вычитание $\operatorname{erf}(z)$ из 1;
- `erfcx(z)` – масштабированная дополнительная функция ошибок $e^{z^2} \operatorname{erfc}(z)$;
- `erfinv(y)` – функция, обратная функции ошибок;
- `erfcinv(y)` – функция, обратная дополнительной функции ошибок;
- `wofz(z)` – функция Фаддеевой, масштабированная дополнительная (комплексная) функция ошибок с комплексным аргументом:

  $$w(z) = e^{-z^2} \operatorname{erfc}(-iz) = \operatorname{erfcx}(-iz),$$

  которая применяется в задачах, связанных с физикой плазмы и распространения излучения;
- `dawsn(z)` – связанный с функцией ошибок интеграл, известный как интеграл Доусона:

  $$D(z) = e^{-z^2} \int_0^z e^{t^2} \mathrm{d}t.$$

Пример П8.8. Волновая функция, соответствующая основному состоянию одномерного квантового генератора гармонических (синусоидальных) колебаний, может быть выражена через параметр $\alpha = \sqrt{mk} / \hbar$, где m – масса, k – постоянная взаимодействия:

$$\psi_0(x) = (\alpha/\pi)^{1/4} \exp(-\alpha x^2/2).$$

Плотность распределения вероятностей в определенной позиции генератора определяется формулой $P_0(x) = |\psi_0(x)|^2$, а ненулевое значение вне классических точек экстремума $\pm\alpha^{-1/2}$ – явление, известное как туннельный эффект. Вычислим вероятность туннельного эффекта для генератора в состоянии ψ_0.

Волновая функция симметрична по оси x, поэтому вероятность туннельного эффекта равна:

$$P(x<-\alpha)+P(x>\alpha)=2P(x>\alpha)=2\sqrt{\frac{\alpha}{\pi}}\int_{\alpha^{-1/2}}^{\infty}\exp\left(-\alpha x^2\right)\mathrm{d}x=\frac{2}{\sqrt{\pi}}\int_1^{\infty}e^{-y^2}\mathrm{d}y=\ \mathrm{erfc}(1).$$

Дополнительную функцию ошибок можно вычислить напрямую:

```
In [x]: from scipy.special import erfc
In [x]: erfc(1)
0.15729920705028516
```

То есть вероятность туннельного эффекта приблизительно равна 16 %.

Пример П8.9. Линейный контур Фогта используется в моделировании и анализе распространения излучения в атмосфере. Это свертка (сверточная аппроксимация) контура Гаусса $G(x; \sigma)$ и лоренцева контура $L(x; \gamma)$:

$$V(x;\sigma,\gamma)=\int_{-\infty}^{\infty}G(x';\sigma)L(x-x';\gamma)\,\mathrm{d}x',\ \text{где}$$
$$G(x;\sigma)=\frac{1}{\sigma\sqrt{2\pi}}\exp\left(-\frac{x^2}{2\sigma^2}\right)\ \text{и}\ L(x;\gamma)=\frac{\pi/\gamma}{x^2+\gamma^2}.$$

Здесь γ – полуразмах (полуширина) на уровне половинной амплитуды (HWHM) лоренцева контура, а σ – стандартное отклонение контура Гаусса, связанное с его HWHM как $\alpha = \sigma\sqrt{(2\ln 2)}$. Если учитывать частоту ν, то $x = \nu - \nu_0$, где ν_0 – центр линии.

Для контура Фогта не существует конечной формы, но он связан с действительной частью функции Фаддеевой $w(z)$ следующим образом:

$$V(x; \sigma, \gamma) = \mathrm{Re}[w(z)] / (\sigma\sqrt{(2\pi)}),\ \text{где}\ z = (x + i\gamma) / \sigma\sqrt{2}.$$

Программа в листинге 8.6 создает график контура Фогта для $\gamma = 0.1$, $\alpha = 0.1$ и сравнивает его с соответствующими контурами Гаусса и Лоренца (см. рис. 8.5). Приведенные выше уравнения реализованы в трех функциях `G`, `L` и `V`, определенных в коде листинга 8.6.

Листинг 8.6. Сравнение форм контуров Лоренца, Гаусса и Фогта

```
# eg8-voigt.py
import numpy as np
from scipy.special import wofz
import matplotlib.pyplot as plt

def G(x, alpha):
    """ Return Gaussian line shape at x with HWHM alpha """
    """ Возвращает форму контура Гаусса в точке x при HWHM alpha. """
    return np.sqrt(np.log(2) / np.pi) / alpha * np.exp(-(x / alpha)**2 * np.log(2))

def L(x, gamma):
    """ Return Lorentzian line shape at x with HWHM gamma """
    """ Возвращает форму контура Лоренца в точке x при HWHM gamma. """
    return gamma / np.pi / (x**2 + gamma**2)
```

```
def V(x, alpha , gamma):
    """
    Return the Voigt line shape at x with Lorentzian component HWHM gamma
    and Gaussian component HWHM alpha.
    """
    """
    Возвращает форму контура Фогта в точке x с лоренцевым компонентом HWHM gamma
    и гауссовым компонентом HWHM alpha.
    """
    sigma = alpha / np.sqrt(2 * np.log(2))
    return np.real(wofz((x + 1j*gamma)/sigma/np.sqrt(2))) / sigma / np.sqrt(2*np.pi)

alpha, gamma = 0.1, 0.1
x = np.linspace(-0.8, 0.8, 1000)
plt.plot(x, G(x, alpha), ls=':', c='k', label='Gaussian')
plt.plot(x, L(x, gamma), ls='--', c='k', label='Lorentzian')
plt.plot(x, V(x, alpha , gamma), c='k', label='Voigt')
plt.legend()
plt.show()
```

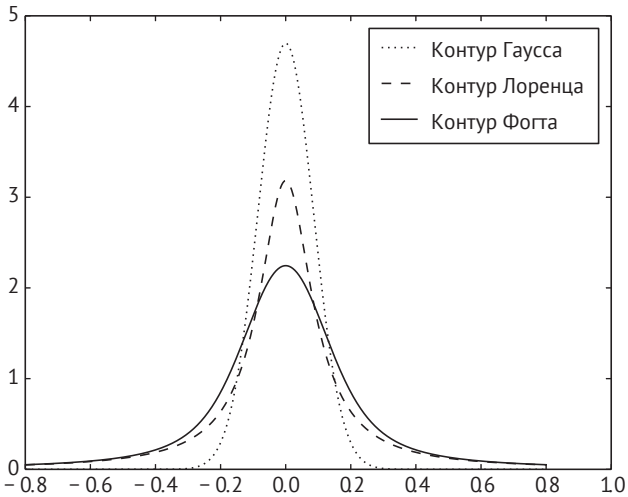


Рис. 8.5. Сравнение форм контуров Лоренца, Гаусса и Фогта при $\gamma = \alpha = 0.1$

8.1.5 Интегралы Френеля

Интегралы Френеля используются в оптике и определяются следующими формулами:

$$S(z) = \int_0^z \sin\left(\frac{t^2}{2\pi}\right) \mathrm{d}t,\ C(z) = \int_0^z \cos\left(\frac{t^2}{2\pi}\right) \mathrm{d}t.$$

Оба интеграла возвращаются в кортеже для действительного или комплексного аргумента `z` метода `fresnel(z)` из модуля `scipy.special`. Связанная с ними функция `fresnel_zeros(nt)` возвращает `nt` первых комплексных нулей интегралов $S(z)$ и $C(z)$.

Пример П8.10. Интегралы Френеля играют важную роль в описании эффектов дифракции в оптике, но их роль не менее важна в практическом проектировании путепроводов (транспортных развязок). Кривая, описываемая параметрическими уравнениями $(x,y) = (S(t), C(t))$, называется клотоидой (или спиралью Эйлера (часто ее также называют спиралью Корню)) и обладает тем свойством, что ее кривизна пропорциональна расстоянию, отмеряемому вдоль ее траектории. Таким образом, транспортное средство, двигающееся с постоянной скоростью, будет иметь постоянный коэффициент углового ускорения при движении по такой кривой – это означает, что водитель может поворачивать руль с постоянной скоростью, что повышает безопасность транспортной развязки.

Код, приведенный ниже, создает графическое изображение спирали Эйлера при $-10 \leqslant t \leqslant 10$ (см. рис. 8.6).

```
In [x]: import numpy as np
In [x]: from scipy.special import fresnel
In [x]: import matplotlib.pyplot as plt
In [x]: t = np.linspace(-10, 10, 1000)
In [x]: plt.plot(*fresnel(t), c='k')
In [x]: plt.show()
```

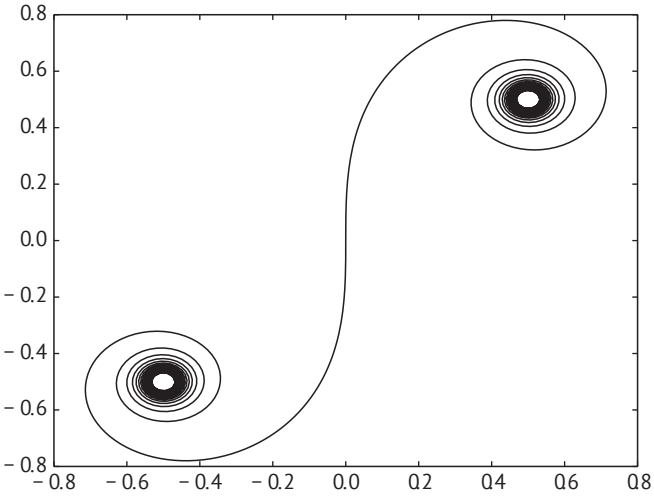


Рис. 8.6. Спираль Эйлера

8.1.6 Биномиальные коэффициенты и интегралы от показательной функции

Биномиальный коэффициент $\binom{n}{k} \equiv {}^{n}C_{k}$ возвращает метод `binom(n, k)` из модуля `scipy.special`.

Для вычисления разнообразных форм интеграла от показательной функции предлагаются различные методы. Стандартную форму возвращает метод `expi(z)`:

$$\mathrm{Ei}(z) = \int_{-\infty}^{z} \frac{e^t}{t} \mathrm{d}t, \ \left|\arg(-z) < \pi\right|.$$

Метод `expn(n, x)` возвращает значение интеграла

$$\int_1^{\infty} \frac{e^{-xt}}{t^n} \mathrm{d}t.$$

При $n = 1$ более быстрые вычисления и более точный результат обеспечивает метод `expl(z)`:

$$\int_1^{\infty} \frac{e^{-zt}}{t^n} \mathrm{d}t.$$

Пример П8.11. Любой интеграл вида

$$\int f(z) e^z \mathrm{d}z,$$

где $f(z) = P(z)/Q(z)$ является дробно-рациональной функцией (многочленом), можно привести к виду

$$\int R(z) e^z \mathrm{d}z + \sum_i \int \frac{e^z}{(z - a_i)^{n_i}} \mathrm{d}z,$$

где $R(z)$ – многочлен (который может быть нулем), получаемый разложением на частично вычислимые функции. Здесь первый интеграл можно вычислить стандартными методами (многократно повторяемым интегрированием по частям). При условии, что путь интегрирования не проходит через какие-либо сингулярные (критические) точки подынтегральной функции, второй член можно записать с использованием интегралов от показательной функции.

Например, рассмотрим интеграл

$$I = \int_{-\infty}^{-2} \frac{e^z}{z^2(z-1)} \mathrm{d}z.$$

Легко показать, что:

$$\frac{1}{z^2(z-1)} = \frac{1}{z-1} - \frac{1}{z} - \frac{1}{z^2}.$$

поэтому приведенный выше интеграл можно записать в виде трех членов:

$$I = \int_{-\infty}^{-2} \frac{e^z}{z-1} \mathrm{d}z - \int_{-\infty}^{-2} \frac{e^z}{z} \mathrm{d}z - \int_{-\infty}^{-2} \frac{e^z}{z^2} \mathrm{d}z.$$

Второй интеграл здесь – это просто $-Ei(-2)$, а подстановка $u = z - 1$ приводит первый интеграл к $eEi(-3)$. Последний интеграл можно записать в терминах $En(z)$ или при дальнейшем сокращении интегрированием по частям:

$$\int_{-\infty}^{-2} \frac{e^z}{z^2} dz = -\frac{e^{-2}}{2} + Ei(-2).$$

Следовательно:

$$I = eEi(-3) - 2Ei(-2) - e^{-2}/2.$$

С помощью библиотеки SciPy интеграл вычисляется следующим образом:

```
In [x]: import numpy as np
In [x]: from scipy.special import expi
In [x]: np.e * expi(-3) - 2*expi(-2) - np.exp(-2)/2
-0.0053357974213484663
```

8.1.7 Ортогональные многочлены и сферические гармонические функции

Модуль `scipy.special` содержит множество функций для вычисления разнообразных видов ортогональных многочленов, в том числе многочленов Лежандра, Якоби, Лагерра, Эрмита и многих разновидностей многочленов Чебышева. Обобщенное имя для этих функций `eval_poly(n, x)`, где `n` – порядок многочлена, `x` – подобная массиву последовательность значений, для которых вычисляется многочлен. В табл. 8.2 приведены имена некоторых функций вычисления многочленов.

Таблица 8.2. Некоторые методы вычисления ортогональных многочленов в библиотеке SciPy

Метод	Описание
`eval_legendre(n, x)`	Многочлен Лежандра $P_n(x)$
`eval_chebyt(n, x)`	Многочлен Чебышева первого рода $T_n(x)$
`eval_chebyu(n, x)`	Многочлен Чебышева второго рода $U_n(x)$
`eval_hermite(n, x)`	Многочлен Эрмита («для физиков») $H_n(x)$
`eval_jacobi(n, alpha, beta, x)`	Многочлен Якоби $P_n^{(\alpha,\beta)}(x)$
`eval_laguerre(n, x)`	Многочлен Лагерра первого рода $L_n(x)$
`eval_genlaguerre(n, alpha, x)`	Обобщенный многочлен Лагерра первого рода $L_n^{\alpha}(x)$

Сферические гармонические функции, используемые в библиотеке SciPy, определяются формулой:

$$Y_n^m(\phi,\theta) = \sqrt{\frac{(2n+1)}{4\pi}\frac{(n-m)!}{(n+m)!}} P_n^m(\cos\phi)e^{im\theta},$$

где $n = 0, 1, 2, \ldots$ называется степенью, а $m = -n, -n+1, \ldots, n$ – порядком сферической гармонической функции. Функции $P_n^m(x)$ – соответствующие многочлены Лежандра. Как и для многих других специальных функций, в разнообразных областях применения используются различные частные соглашения и методы нормализации, поэтому весьма важно тщательно проверять их и вносить соответствующие изменения при практическом использовании этих функций.

В частности, во многих других областях используется l для обозначения степени гармонической функции, а в определении меняются местами θ и φ. Поэтому необходимо уточнить, что в библиотеке SciPy θ – азимутальный (меридиональный, «по долготе») угол (принимающий значения от 0 до 2π), а φ – полярный (широтный, поперечный) угол (от 0 до π).

Метод `scipy.special.sph_harm` вызывается с аргументами:

```
scipy.special.sph_harm(m, n, theta , phi)
```

где `theta` и `phi` могут быть объектами типа массив.

Пример 8.12. Визуализация сферических гармонических функций связана с некоторыми трудностями, поскольку эти функции комплексные и определяются в угловых координатах (θ, φ). Один из способов – изображение действительной части только на единичной сфере. Библиотека Matplotlib предоставляет набор инструментов для таких трехмерных графиков mplot3d, используемый в листинге 8.7, который создает график, показанный на рис. 8.7[112].

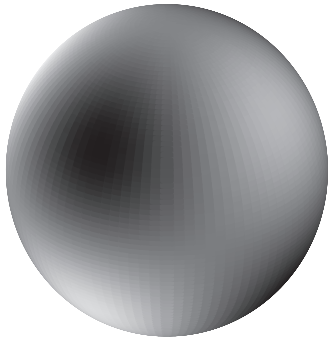


Рис. 8.7. Изображение сферической гармонической функции при $l = 3$, $m = 2$

Листинг 8.7. Сферическая гармоническая функция, определенная при $l = 3$, $m = 2$

```
# eg8-spherical-harmonics.py
import matplotlib.pyplot as plt
from matplotlib import cm, colors
from mpl_toolkits.mplot3d import Axes3D
import numpy as np
from scipy.special import sph_harm

phi = np.linspace(0, np.pi, 100)
theta = np.linspace(0, 2*np.pi, 100)
phi, theta = np.meshgrid(phi, theta)

# Декартовы координаты единичной сферы.
x = np.sin(phi) * np.cos(theta)
y = np.sin(phi) * np.sin(theta)
z = np.cos(phi)
```

[112] См. раздел 7.6 и документацию здесь: https://matplotlib.org/mpl_toolkits/mplot3d/.

```
m, l = 2, 3

# Вычисление сферической гармонической функции Y(l, m) и ее нормализация в интервал [0, 1].
fcolors = sph_harm(m, l, theta, phi).real
fmax, fmin = fcolors.max(), fcolors.min()
fcolors = (fcolors - fmin)/(fmax - fmin)

# Установка отношения сторон равным 1, чтобы изображение функции выглядело сферическим.
fig = plt.figure(figsize=plt.figaspect(1.))
ax = fig.add_subplot(projection='3d')
ax.plot_surface(x, y, z, rstride=1, cstride=1, facecolors=cm.jet(fcolors))
# Отключение осевых плоскостей.
ax.set_axis_off()
plt.show()
```

8.1.8 Упражнения

Вопросы

В8.1.1. Изменить одну строку в программе примера П8.1, чтобы вывести 10 наиболее точно определенных констант (исключая те, значение которых задано по определению).

В8.1.2. Использовать константы и коэффициенты преобразования библиотеки SciPy для вычисления числового выражения плотности N/V молекул идеального газа при стандартной температуре и давлении (T = 0° C, p = 1 атм). Закон для идеального газа $pV = Nk_BT$.

Задачи

З8.1.1. Использовать метод `scipy.special.binom` для создания изображения треугольника Паскаля с биномиальными коэффициентами $\binom{n}{k}$ до n = 8.

З8.1.2. Диск (узор) Эйри – это круговая дифракционная картина, полученная при прохождении света через равномерно освещённое круглое отверстие. Она состоит из яркого центрального диска, окруженного концентрическими менее яркими кольцами (чередующимися с темными концентрическими кольцами). Математически узор Эйри можно описать с использованием функции Бесселя первого рода

$$I(\theta) = I_0(2J_1(x) / x)^2,$$

где θ – угол наблюдения, а $x = ka \sin\theta$, a – радиус отверстия, $k = 2\pi/\lambda$ – угловое волновое число для света с длиной волны λ.

Изобразить узор Эйри как $I(x)/I_0$ при $-10 \leqslant x \leqslant 10$ и вывести из позиции первого минимума этой функции максимальную разрешающую способность (в дуговых (угловых) секундах) человеческого глаза (диаметр зрачка 3 мм) при длине волны 500 нм.

38.1.3. Написать функцию `get_wv`, принимающую молярную энергию разрыва связи D0 в единицах измерения кДж/моль и возвращающую длину волны фотона, соответствующую этой энергии для одной молекулы, в нм. Энергия фотона с длиной волны λ равна $E = hc/\lambda$.

Например:

```
In [x]: get_wv(497)
Out[x]: 240.69731528286377
```

38.1.4. Эллипсоид – это трехмерная фигура, ограниченная поверхностью, которую описывает каноническое уравнение:

$$x^2/a^2 + y^2/b^2 + z^2/c^2 = 1,$$

где a, b, c – главные полуоси. Если $a = b = c$, то эллипсоид является сферой. Объем эллипсоида вычисляется по простой формуле:

$$V = 4/3\ \pi abc.$$

В общем случае для площади поверхности эллипсоида не существует конечной (вычислимой) формулы, но ее можно выразить через неполные эллиптические интегралы первого и второго рода $K(\varphi,k)$ и $E(\varphi,k)$:

$$S = 2\pi c^2 + \frac{2\pi ab}{\sin\phi}\left(K(\phi,k^2)\cos^2\phi + E(\phi,k^2)\sin^2\phi\right),$$

где

$\cos\varphi = c/a$, $k = (a\sqrt{(b^2 - c^2)}) / (b\sqrt{(a^2 - c^2)})$, а система координат выбрана так, что $a \geqslant b \geqslant c$.

Определить функцию `ellipsoid_surface` для вычисления площади поверхности обобщенного эллипсоида и сравнить полученные результаты для эллипсоидов различной формы с вычислением по следующей приближенной формуле:

$$S \approx 2\pi c^2 + 2\pi ab\left(1 - \frac{b^2 - c^2}{6b^2}\left(1 - \frac{3b^2 + 10c^2}{56b^2}r^2\right)\right),$$

где $r = \varphi / \sin\varphi$.

38.1.5. Снижение уровня воды или изменение гидравлического горизонта s (мера давления воды выше некоторого геодезического репера), расстояние r от источника (водоема) в момент времени t. Вода из источника выкачивается с постоянной скоростью Q. Этот процесс можно смоделировать, используя метод (уравнение) Тейса:

$$s(r,t) = H_0 - H(r,t) = Q / (4\pi T) \times W(u), \quad \text{где } u = (r^2S) / (4Tt).$$

Здесь H_0 – гидравлический горизонт при отсутствии источника, S – коэффициент запасов подземных вод (объем воды, высвобождаемый в единицу времени,

уменьшается на H на единицу площади), а T – коэффициент фильтрации (проницаемости) (мера количества воды, перемещаемого горизонтально в единицу времени). Функция источника $W(u)$ – это просто интеграл от показательной функции $E_1(u)$.

Для источника, из которого выкачивается вода со скоростью Q = 1000 м^3/день с водоносного горизонта, описываемого параметрами H_0 = 20 м, S = 0.0003, T = 1000 м^2/день, определить высоту гидравлического горизонта как функцию от r после завершения интервала времени t = 1 день выкачивания.

Сравнить полученный результат с приближенной версией уравнения Тейса, известной как уравнение Джейкоба, в котором функция источника принимается приближенно $W(u) \approx -\gamma - \ln u$, где γ = 0.577215664... – константа Эйлера–Маскерони.

38.1.6. Некоторые электронные компоненты охлаждаются кольцевыми радиаторами (теплоотводами), отводящими тепло от компонента и обеспечивающими бóльшую площадь поверхности, с которой тепло рассеивается в окружающее пространство.

Эффективность охлаждения кольцевого радиатора толщиной 2w при внутреннем радиусе r_0 и внешнем радиусе r_1 можно записать с использованием измененных функций Бесселя первого и второго рода:

$$\eta = \frac{2r_0}{\beta (r_1^2 - r_0^2)} \frac{K_1(u_0)I_1(u_1) - I_1(u_0)K_1(u_1)}{K_0(u_0)I_1(u_1) + I_0(u_0)K_1(u_1)},$$

где $u_0 = \beta r_0$, $u_1 = \beta r_1$ и

$$\beta = \sqrt{\frac{h_c}{\kappa w}}.$$

Здесь h_c – коэффициент теплопередачи (принимается постоянным на всей поверхности радиатора), κ – коэффициент теплопроводности материала, из которого сделан радиатор.

Вычислить эффективность охлаждения алюминиевого кольцевого радиатора с размерами r_0 = 5 мм, r_1 = 10 мм, w = 0.1 мм. Принять h_c = 10 Вт/м^2·К и κ = 200 Вт/м·К.

Вычислить рассеивание (отвод) тепла Q (произведение эффективности, площади поверхности радиатора и разности температур) при температуре компонента T_0 = 400 К и температуре окружающей среды T_e = 300 К.

8.2 Интегрирование и обыкновенные дифференциальные уравнения

Пакет `scipy.integrate` содержит методы для вычисления определенных интегралов. С их помощью можно вычислять собственные (с конечными пределами) и несобственные (с бесконечными пределами) интегралы. Эти методы также позволяют выполнять численное интегрирование систем обыкновенных дифференциальных уравнений.

8.2.1 Определенные интегралы от одной переменной

Основной программой численного интегрирования является `scipy.integrate.quad`, основанная на старой «заслуженной» библиотеке FORTRAN 77 QUADPACK. Здесь используется адаптивная квадратура для приближенного вычисления значения интеграла посредством деления его области интегрирования на меньшие интервалы, выбираемые итеративно для соответствия конкретному пределу допускаемой погрешности (т. е. оценочной абсолютной или относительной погрешности). В самой простой форме метод принимает три аргумента: объект функции Python, соответствующий интегрируемой функции, `func` и пределы интегрирования `a` и `b`. В аргументе `func` обязательно должен передаваться хотя бы один объект, если этот аргумент содержит более одного объекта, то интегрирование выполняется по координате, соответствующей первому значению аргумента. При простом способе применения удобным способом определения `func` являются `lambda`-выражения. Например, для получения значения интеграла $\int_1^4 x^{-2}\,dx = 3/4$ в численном виде:

```
In [x]: from scipy.integrate import quad
In [x]: f = lambda x: 1/x**2
Out[x]: quad(f, 1, 4)
(0.7500000000000002, 1.913234548258995e-09)
```

метод `quad` возвращает два значения в кортеже – значение интеграла и оценку абсолютной погрешности полученного результата.

Для вычисления несобственных интегралов используется специальное значение `np.inf`:

```
In [x]: quad(lambda x: np.exp(-x**2), 0, np.inf)
Out[x]: (0.8862269254527579, 7.101318390472462e-09)
In [x]: np.sqrt(np.pi)/2      # Результат, полученный аналитическим способом.
Out[x]: 0.88622692545275794
```

Обратите внимание: при этом вызове `quad` даже не указывается имя интегрируемой функции, а просто передается сама функция как анонимный `lambda`-объект.

Для более сложных функций требуется явное определение объекта функции Python с помощью ключевого слова `def`:

```
In [x]: def g(x):
...:        if abs(x) < 0.5:
...:            return -x
...:        return x - np.sign(x)
...:
In [x]: quad(g, -0.6, 0.8)
Out[x]: ( -0.06000000000000002, 6.661338147750941e-17)
```

Функции с сингулярными (особыми, критическими) точками или с точками разрыва могут создавать проблемы для программы численного интегрирования, даже если требуемый интеграл однозначно определен. Например, функ-

ция sinc (кардинальный синус) $f(x) = \sin(x)/x$ имеет устранимую особую точку при $x = 0$, в которой при обычном применении метода `quad` возникает критическая ошибка:

```
In [x]: sinc = lambda x: np.sin(x)/x
In [x]: quad(sinc, -2, 2)
...: RuntimeWarning: invalid value encountered in double_scalars
                     (некорректное значение обнаружено в double_scalars)
Out[37]: (nan, nan)
```

Устранить эту проблему можно, передавая в метод `quad` список таких точек разрыва в аргументе `points` (список не обязательно должен быть упорядоченным):

```
In [x] quad(sinc, -2, 2, points=[0,])
(3.210825953605389, 3.5647329017567276e-14)
```

Следует отметить, что точки разрыва не могут быть заданы при бесконечных пределах интегрирования.

Аргументы `epsrel` и `epsabs` позволяют определить требуемую точность интегрирования как относительную и абсолютную погрешность соответственно. По умолчанию для обоих аргументов принято значение `1.49e-8`, но интегрирование можно выполнить быстрее, если нужно получить менее точный результат. В качестве примера рассмотрим быстро изменяющуюся функцию $f(x) = e^{-|x|}\sin^2 x^2$:

```
In [x]: f = lambda x: np.sin(x**2)**2 * np.exp(-np.abs(x))
In [x]: quad(f, -1, 2, epsabs=0.1)
Out[x]: (0.29551455828969975, 0.001529571827911671)
In [x]: quad(f, -1, 2, epsabs=1.49e-8)   # (Абсолютная погрешность, принятая по умолчанию.)
Out[x]: (0.29551455505239044, 4.449763315720537e-10)
```

Обратите внимание: значение `epsabs` – это только требуемая верхняя граница: действительная оцениваемая точность результата может быть гораздо лучшей, т. е. в действительности полученный результат может оказаться более точным, чем эта оценка.

Если функция принимает один или несколько параметров в дополнение к своему основному аргументу, то эти дополнительные параметры необходимо передать в метод `quad` как кортеж в аргументе `args`. Например, интеграл

$$I_{n,m} = \int_{-\pi/2}^{\pi/2} \sin^n x \cos^m x \mathrm{d}x$$

можно определить в численном выражении следующим образом:

```
In [x]: def f(x, n, m):
....:       return np.sin(x)**n * np.cos(x)**m
....:
In [x]: n, m = 2, 1
In [x]: quad(f, -np.pi/2, np.pi/2, args=(n, m))
(0.6666666666666666, 1.625746841018571e-13)
```

Обратите внимание: здесь дополнительные параметры `n` и `m` передаются как аргументы интегрируемой функции после координаты, определяющей направление интегрирования по (x).

Пример П8.13. Рассмотрим тор со средним радиусом *R* и радиусом поперечного сечения *r*. Объем этой фигуры можно вычислить аналитически в декартовых координатах как объем тела вращения:

$$V = 2\int_{R-r}^{R+r} 2\pi xz\mathrm{d}x, \text{ где z}=\sqrt{r^2-(x-R)^2}.$$

Центр тора является началом координат, а ось *z* принимается за ось симметрии.

Вычисление интеграла утомительно, но приводит к стандартным методам: $V = 2\pi^2 Rr^2$. Здесь мы применим численный метод с использованием значений $R = 4$, $r = 1$:

```
In [x]: R, r = 4, 1
In [x]: f = lambda x, R, r: x * np.sqrt(r**2 - (x-R)**2)
In [x]: V, _ = quad(f, R-r, R+r, args=(R, r))
In [x]: V *= 4 * np.pi
In [x]: Vexact = 2 * np.pi**2 * R * r**2
In [x]: print('V = {} (exact: {})'.format(V, Vexact))
Out[x]: V = 78.95683520871499 (exact: 78.95683520871486)
```

8.2.2 Интегралы от двух и нескольких переменных

В модуле `scipy.integrate` методы `dblquad`, `tplquad` и `nquad` вычисляют двойные, тройные и кратные интегралы соответственно. Поскольку в общем случае пределы по одной координате могут зависеть от другой координаты, синтаксис вызова перечисленных выше методов немного сложнее.

Метод `dblquad` вычисляет двойной интеграл:

$$\int_a^b \int_{g(x)}^{h(x)} f(x,y)\mathrm{d}y\mathrm{d}x.$$

Здесь *f*(*x*,*y*) передается как функция по меньшей мере двух переменных `func(y, x, …)`. Эта фуункция обязательно должна принимать `y` как свой первый аргумент, а `x` как второй аргумент. Пределы интегрирования передаются в `dblquad` в следующих четырех аргументах. Сначала два аргумента `a` и `b` определяют нижний и верхний пределы интегрирования по `x`, как и для метода `quad`. Следующие два аргумента `gfun` и `hfun` определяют нижний и верхний пределы интегрирования по *y*, и они обязательно должны быть вызываемыми объектами, принимающими один аргумент – число с плавающей точкой, значение `x`, при котором этот предел применяется (т. е. эти аргументы сами обязательно должны быть функциями от `x`). Если любой из пределов интегрирования по *y* не зависит от *x*, то `gfun` или `hfun` могут возвращать постоянное значение.

Как простой пример рассмотрим интеграл

$$\int_1^4 \int_0^2 x^2 y \, dy dx,$$

который можно вычислить следующим образом:

```
In [x]: f = lambda y, x: x**2 * y
In [x]: a, b = 1, 4
In [x]: gfun = lambda x: 0
In [x]: hfun = lambda x: 2
In [x]: dblquad(f, a, b, gfun, hfun)
Out[x]: (42.00000000000001, 4.662936703425658e-13)
```

Здесь оба аргумента `gfun` и `hfun` вызываются с передачей в них значения `x`, но они возвращают постоянные значения (`0` и `2` соответственно) вне зависимости от переданного значения.

Разумеется, все показанные выше операции в исходном коде можно свернуть в одну строку:

```
In [x]: dblquad(lambda y, x: x**2 * y, 1, 4, lambda x: 0, lambda x: 2)
Out[x]: (42.00000000000001, 4.662936703425658e-13)
```

Двойной интеграл можно использовать для вычисления площади некоторой двумерной фигуры, ограниченной одной или несколькими функциями. Для примера в полярных координатах рассмотрим область внутри кривой $r = 2 + 2\sin\theta$, но вне окружности с радиусом $r = 2$ при θ в интервале $[0, 2\pi]$ (см. рис. 8.8).

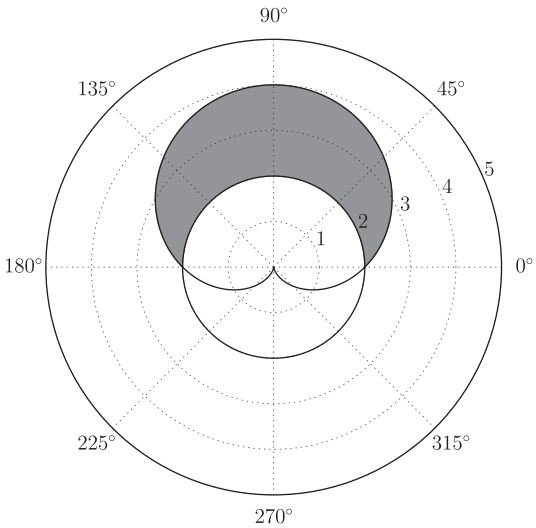


Рис. 8.8. Фигура, определенная как область внутри кривой $r = 2 + 2\sin\theta$, но вне окружности с радиусом $r = 2$

Эти фигуры пересекаются в точках $\theta = 0, \pi$, поэтому требуемый интеграл имеет следующий вид:

$$A = \int_0^{\pi} \int_2^{2+2\sin\theta} r\mathrm{d}r\mathrm{d}\theta,$$

где *r dr* dθ – бесконечно малый элемент области в полярных координатах. Этот частный определенный интеграл без затруднений вычисляется аналитически (A = 8 + π), поэтому результат применения численного метода легко проверить:

```
In [x]: r1, r2 = lambda theta: 2, lambda theta: 2 + 2*np.sin(theta)
In [x]: A, _ = dblquad(lambda r, theta: r, 0, np.pi, r1, r2)
Out[x]: 11.141592653589791
In [x]: 8 + np.pi         # Точный результат.
Out[x]: 11.141592653589793
```

Вычисляемая функция – это просто `r`, определяемая выражением `lambda r, theta: r`. Во внутреннем интеграле пределы по *r* равны 2 и 2 + 2sinθ. Для внешнего интеграла значение угла θ определяется от 0 до π.

Метод `tplquad` вычисляет тройные интегралы и принимает функцию от трех переменных `func(z, y, x)` и шесть дополнительных аргументов: постоянные пределы интегрирования по *x* `a` и `b`, пределы интегрирования по *y* `gfun(x)` и `hfun(x)` (которые являются функциями, как и для метода `dblquad`) и пределы интегрирования по *z* `qfun(x, y)` и `rfun(x, y)` (функции от `x` и `y` именно в таком порядке).

Интегрирование с более высокой кратностью выполняется методом `scipy.integrate.nquad`, который здесь не рассматривается (документацию и примеры см. здесь: https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.nquad.html).

Пример П8.14. Объем единичной сферы 4π/3 можно записать в виде тройного интеграла в сферических полярных координатах с постоянными пределами интегрирования:

$$\int_0^{2\pi} \int_0^{\pi} \int_0^1 r^2 \sin\theta \, \mathrm{d}r\mathrm{d}\theta\mathrm{d}\phi.$$

```
In [x]: from scipy.integrate import tplquad
In [x]: tplquad(lambda phi, theta, r: r**2 * np.sin(theta),
                0, 1,
                lambda theta: 0, lambda theta: np.pi,
                lambda theta, phi: 0, lambda theta, phi: 2*np.pi)
Out[x]: (4.18879020478639, 4.650491330678174e-14)
```

Кроме того, можно записать это выражение в декартовых координатах с пределами интегрирования в форме функций:

$$8\int_0^1 \int_0^{\sqrt{1-x^2-y^2}} \mathrm{d}z\mathrm{d}y\mathrm{d}x,$$

где интеграл находится в положительном октанте трехмерной декартовой системы координат.

```
In [x]: A, _ = tplquad(lambda z, y, x: 1,
                       0, 1,
                       lambda x: 0, lambda x: np.sqrt(1 - x**2),
                       lambda x, vy: 0, lambda x, y: np.sqrt(1 - x**2 - y**2))
In [x]: 8*A
Out[x]: 4.188790204786391
```

Пример П8.15. В этом примере определяется масса и центр масс тетраэдра, ограниченного координатными осевыми плоскостями и плоскостью $x + y + z = 1$ с плотностью $\rho = \rho(x,y,z)$, где плотность $\rho(x,y,z)$ представлена как лямбда-функция. Проверим это на функциях $\rho = 1$, $\rho = x$ и $\rho = x^2 + y^2 + z^2$.

Масса может быть записана в виде тройного интеграла плотности в зависимости от объема тетраэдра:

$$m = \int_V \rho(x,y,z)\,\mathrm{d}V = \int_0^1 \int_0^{1-x} \int_0^{1-x-y} \rho(x,y,z)\,\mathrm{d}z\mathrm{d}y\mathrm{d}x,$$

а координаты центра масс определяются следующими интегралами:

$$m\bar{x} = \int_V x\rho(x,y,z)\,\mathrm{d}V,\ m\bar{y} = \int_V y\rho(x,y,z)\,\mathrm{d}V,\ m\bar{z} = \int_V z\rho(x,y,z)\,\mathrm{d}V.$$

Программа в листинге 8.8 использует метод `scipy.integrate.tplquad` для выполнения требуемых операций интегрирования (которые могут быть решены и аналитически).

Листинг 8.8. Вычисление массы и центра масс тетраэдра с учетом его трех различных плотностей

```
# eg8-tetrahedron-cofm.py

import numpy as np
from scipy.integrate import tplquad

# Пределы интегрирования по x, y, z.
a, b = 0, 1
gfun, hfun = lambda x: 0, lambda x: 1 - x
qfun, rfun = lambda x, y: 0, lambda x, y: 1 - x - y
lims = (a, b, gfun, hfun, qfun, rfun)                                    ❶

# Три различные функции плотности.
rhos = [lambda x, y, z: 1,
        lambda x, y, z: x,
        lambda x, y, z: x**2 + y**2 + z**2]

for rho in rhos:
    # Масса как тройной интеграл от rho по объему.
    m, _ = tplquad(rho, *lims)
    # Центр масс (xbar, ybar, zbar).
    mxbar, _ = tplquad(lambda x, y, z: x * rho(x, y, z), *lims)
    mybar, _ = tplquad(lambda x, y, z: y * rho(x, y, z), *lims)
    mzbar, _ = tplquad(lambda x, y, z: z * rho(x, y, z), *lims)
    xbar, ybar, zbar = mxbar / m, mybar / m, mzbar / m

    print('mass = {:g}, CofM = ({:g}, {:g}, {:g})'.format(m, xbar, ybar, zbar))
```

❶ Обратите внимание: шесть аргументов, представляющих пределы тройного интеграла (две константы и две пары лямбда-функций), упакованы в кортеж `lims` (здесь скобки не обязательны).

Вывод результатов:

```
mass = 0.166667, CofM = (0.25, 0.25, 0.25)
mass = 0.0416667, CofM = (0.4, 0.2, 0.2)
mass = 0.05, CofM = (0.277778, 0.277778, 0.277778)
```

8.2.3 Обыкновенные дифференциальные уравнения

Обыкновенные дифференциальные уравнения (ОДУ) можно решать в численном виде с помощью метода `scipy.integrate.odeint` или метода `scipy.integrate.solve_ivp` («solve an initial value problem» – «решение задачи с начальными условиями (или задачи Коши)»). Второй метод появился в версии 1.0 библиотеки SciPy, и именно он рекомендуется для практического использования. Тем не менее в ранее написанном коде продолжает применяться `odeint`, который описан в приложении С. Метод `solve_ivp` решает дифференциальные уравнения первого порядка, но для решения ДУ более высоких порядков необходимо сначала разложить их в систему ДУ первого порядка, как описано ниже.

Решение одного ОДУ первого порядка

В самом простом варианте использования для решения одного ОДУ первого порядка

$$dy/dt = f(t, y)$$

метод `solve_ivp` принимает три аргумента: объект функции, возвращаемой dy/dt, начальный и конечный моменты времени для интегрирования и набор начальных условий y_0. Механизм решения ОДУ выбирает и возвращает последовательность подходящих моментов (точек) времени (если они не заданы), в которых выполняется интегрирование.

Например, рассмотрим дифференциальное уравнение первого порядка, описывающее скорость реакции A → P в выражении концентрации реагента A:

$$d[\mathrm{A}]/dt = -k[\mathrm{A}].$$

Для этого примера существует легко выводимое аналитическое решение:

$$[\mathrm{A}] = [\mathrm{A}]_0 e^{-kt},$$

где $[\mathrm{A}]_0$ – начальная концентрация реагента [A].

Для решения этого уравнения в численном виде с помощью `solve_ivp` необходимо записать его в форме, показанной выше, с единственной зависимой переменной $y(t) \equiv [\mathrm{A}]$, которая является функцией от независимой переменной t (время). Получаем:

$$dy/dt = -ky.$$

Необходимо определить функцию, возвращающую dy/dt, как $f(t,y)$ (в общем случае это функция и от t, и от y), которая в нашем случае весьма проста:

```
def dydt(t, y):
    return -k * y
```

Здесь порядок аргументов важен. Начальный и конечный моменты (точки) времени `t_span` должны передаваться как кортеж `(t0, tf)`, а начальные условия должны быть представлены объектом типа массив, даже если, как в рассматриваемом здесь случае, передается только одно значение. Получаем:

```
soln = solve_ivp(dydt, (t0, tf), [y0])
```

Возвращаемый объект `soln` – это экземпляр класса `OdeResult`, который определяет ряд важных свойств, включая массивы `soln.t` для точек времени, используемых при интегрировании, `soln.y` со значениями решения в этих точках времени и `soln.success` – логический флаг, сообщающий, успешно или нет механизм решения ДУ достиг требуемой конечной точки времени.

Программа, сравнивающая численный и аналитический результаты для реакции при $k = 0.2$ 1/с и $y(0) \equiv [A]_0 = 100$, приведена в листинге 8.9. Полученный график изображен на рис. 8.9.

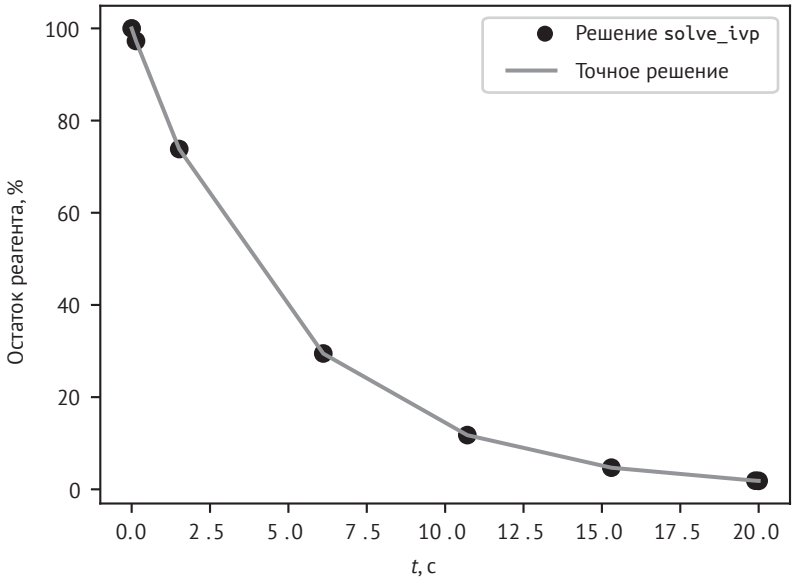


Рис. 8.9. Экспоненциальное снижение концентрации реагента в реакции первого порядка: точное решение и решение численным методом с использованием точек времени, выбранных механизмом решения ОДУ

Листинг 8.9. Кинетика реакции первого порядка

```
import numpy as np
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt

# Постоянная скорость реакции первого порядка, 1/с.
k = 0.2
# Начальное условие для y: 100 % реагента в момент времени t = 0.
y0 = 100

# Начальная и конечная точки (моменты) времени для интегрирования.
t0, tf = 0, 20

def dydt(t, y):
    """ Return dy/dt = f(t, y) at time t. """
    """ Возвращает dy/dt = f(t, y) в момент времени t. """
    return -k * y

# Интегрирование дифференциального уравнения.
soln = solve_ivp(dydt, (t0, tf), [y0])
t, y = soln.t, soln.y[0]

# Построение графика и сравнение численного и точного (аналитического) решений.
plt.plot(t, y, 'o', color='k', label=r'\texttt{solve\_ivp}')
plt.plot(t, y0 * np.exp(-k*t), color='gray', label='Exact')
plt.xlabel(r'$t\;/\mathrm{s}$')
plt.ylabel('Remaining reactant (\%)')
plt.legend()
plt.show()
```

Такая методика хорошо подходит, если все, что требуется, – это конечная концентрация реагента, но для отслеживания изменений концентрации с более высокой разрешающей способностью по времени можно передать специальную последовательность точек времени в аргументе `t_eval`:

```
# Удачно подобранная решетка из 21 точки времени в интервале 0-20 с для отслеживания реакции.
t0, tf = 0, 20
t_eval = np.linspace(t0, tf, 21)
# Интегрирование дифференциального уравнения.
soln = solve_ivp(dydt, (t0, tf), [y0], t_eval=t_eval)
t, y = soln.t, soln.y[0]
```

И все же лучше присвоить аргументу `dense_output` значение `True`, чтобы определить объект `OdeSolution` с именем `sol` как один из возвращаемых объектов. Это можно использовать для генерации интерполяционных значений решения для промежуточных значений точек времени:

```
# Начальная и конечная точки времени для интегрирования.
t0, tf = 0, 20

# Интегрирование дифференциального уравнения.
soln = solve_ivp(dydt, (t0, tf), [y0], dense_output=True)
t = np.linspace(t0, tf, 20)
y = soln.sol(t)[0]
```

Обратите внимание: объект `soln.sol` является вызываемым: значение независимой переменной – времени – передается в него, и возвращается массив решения в этот момент времени. Здесь имеется только одна зависимая переменная y, поэтому массив индексируется по `[0]`.

График с точками решения y, аналогичный полученному в предыдущем примере, показан на рис. 8.10.

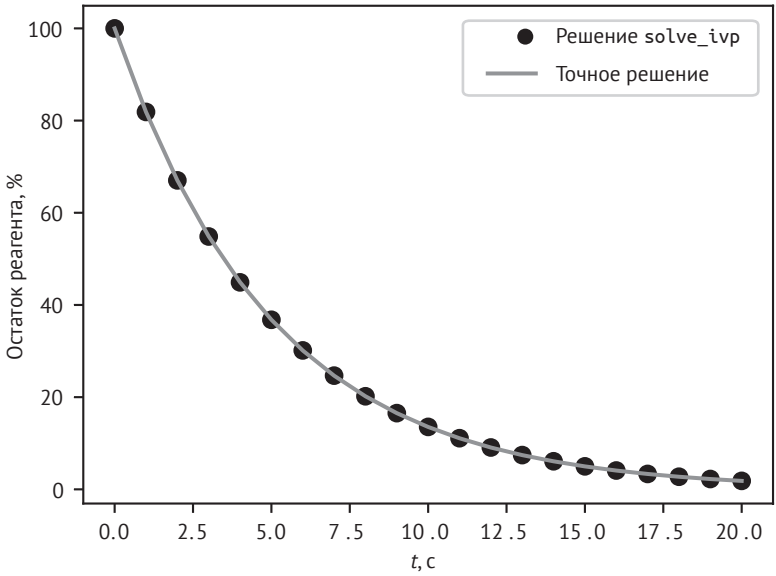


Рис. 8.10. Экспоненциальное снижение концентрации реагента в реакции первого порядка: точное решение и решение численным методом с использованием предварительно определенных точек времени

Как и при использовании группы методов `quad`, если функция, возвращающая производную, требует дополнительных аргументов, то их можно передать в метод `solve_ivp` в параметре `args`. В приведенном выше примере k относится к глобальной области видимости, но эту переменную можно было бы передать явно:

```
def dydt(t, y, k):
    return -k * y
```

Обратите внимание: дополнительные параметры обязательно должны указываться после независимой и зависимой переменных. Тогда вызов метода

`solve_ivp` мог бы выглядеть следующим образом:

```
soln = solve_ivp(dydt, (t0, tf), [y0], args=(k,))
```

Как ни странно, но возможность передачи дополнительных аргументов в `args` была добавлена только в версии SciPy 1.4. Другим способом передачи аргументов из области видимости вызывающей стороны в функцию вычисления производной `dydt` является использование в качестве обертки для этой функции лямбда-выражения:

```
soln = solve_ivp(lambda t, y: dydt(t, y, k), (t0, tf), y0, t_eval=t)
```

Система взаимосвязанных ОДУ первого порядка

Метод `solve_ivp` также может решать систему взаимосвязанных ОДУ первого порядка с несколькими зависимыми переменными: $y_1(t), y_2(t), \ldots, y_n(t)$:

$$\frac{dy_1}{dt} = f_1(y_1, y_2, \ldots, y_n; t),$$
$$\frac{dy_2}{dt} = f_2(y_1, y_2, \ldots, y_n; t),$$
$$\ldots$$
$$\frac{dy_n}{dt} = f_n(y_1, y_2, \ldots, y_n; t).$$

В этом случае функция, передаваемая в метод `solve_ivp()`, обязательно должна возвращать последовательность производных dy_1/dt, dy_2/dt, ..., dy_n/dt для каждой зависимой переменной, т. е. вычисляются все указанные выше функции $f_i(y_1, y_2, \ldots, y_n; t)$ для каждого значения y_i, переданного в функцию в последовательности `y`. Определение этой функции:

```
def deriv(t, y):
    # y = [y1, y2, y3, ...] - последовательность зависимых переменных.
    dy1dt = f1(y, t)        # Вычисляется dy1/dt как f1(y1, y2, ..., yn; t).
    dy2dt = f2(y, t)        # Вычисляется dy2/dt как f2(y1, y2, ..., yn; t).
    # ... и т. д.
    # Возвращаются вычисленные производные в последовательности, например в кортеже:
    return dy1dt, dy2dt, ..., dyndt
```

В качестве конкретного примера рассмотрим реакцию, происходящую в форме двух этапов реакций первого порядка: A → B → P с константами скорости k_1 и k_2. Уравнения, определяющие скорость изменения концентрации реагентов A и B:

$$d[\mathrm{A}]/dt = -k_1[\mathrm{A}],$$

$$d[\mathrm{B}]/dt = k_1[\mathrm{A}] - k_2[\mathrm{B}].$$

И в данном случае можно решить эту пару взаимосвязанных уравнений аналитически, но для численного решения пусть $y_1 \equiv [\mathrm{A}]$ и $y_2 \equiv [\mathrm{B}]$:

$$dy_1/dt = -k_1 y_1,$$

$$dy_2/dt = k_1 y_1 - k_2 y_2.$$

Код в листинге 8.10 выполняет интегрирование этих уравнений при $k_1 = 0.2$ 1/с, $k_2 = 0.8$ 1/с и при начальных условиях $y_1(0) = 100$, $y_2(0) = 0$, затем сравнивает полученный результат с аналитическим решением (см. рис. 8.11).

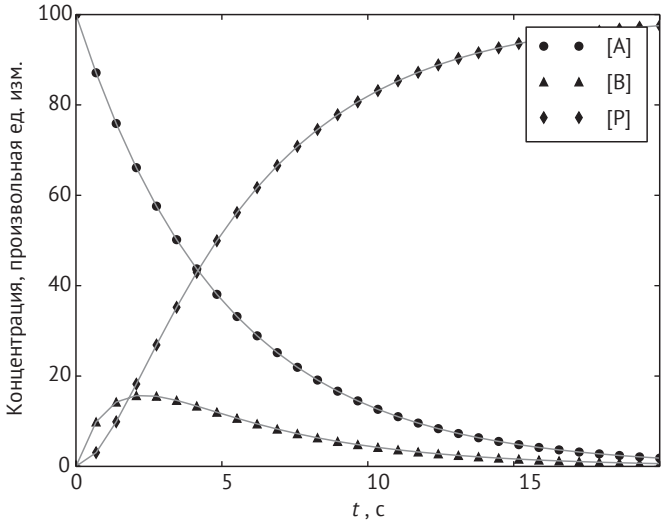


Рис. 8.11. Две взаимосвязанные реакции первого порядка: числовое и точное (аналитическое) решения

Листинг 8.10. Две взаимосвязанные реакции первого порядка

```
import numpy as np
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt

# Константы скорости реакций первого порядка, 1/с.
k1, k2 = 0.2, 0.8
# Начальные условия для y1, y2: [A](t=0) = 100, [B](t=0) = 0.
A0, B0 = 100, 0

# Правильно выбранная сетка точек времени для этой реакции.
t0, tf = 0, 20

def dydt(t, y, k1, k2):
    """ Return dy_i/dt = f(y_i, t) at time t. """
    """ Возвращает dy_i/dt = f(y_i, t) в момент времени t. """
    y1, y2 = y
    dy1dt = -k1 * y1
    dy2dt = k1 * y1 - k2 * y2
    return dy1dt, dy2dt
```

```
# Интегрирование дифференциального уравнения.
y0 = A0, B0
soln = solve_ivp(dydt, (t0, tf), y0, dense_output=True, args=(k1, k2))          ❶

t = np.linspace(t0, tf, 100)
A, B = soln.sol(t)

# [P] определяется по закону сохранения вещества (массы).
P = A0 - A - B

# Аналитический результат.
Aexact = A0 * np.exp(-k1*t)
Bexact = A0 * k1/(k2-k1) * (np.exp(-k1*t) - np.exp(-k2*t))
Pexact = A0 - Aexact - Bexact

plt.plot(t, A, 'o', label='[A]')
plt.plot(t, B, '^', label='[B]')
plt.plot(t, P, 'd', label='[P]')
plt.plot(t, Aexact)
plt.plot(t, Bexact)
plt.plot(t, Pexact)
plt.xlabel(r'$t\;/\mathrm{s}$')
plt.ylabel('Concentration (arb. units)')
plt.legend()
plt.show()
```

❶ И в этом случае, если используется метод `solve_ivp` из версии SciPy до 1.4, то не существует способа прямой передачи дополнительных аргументов `k1` и `k2` в функцию вычисления производной. Использование лямбда-выражения как обертки для этой функции устраняет проблему:

```
soln = solve_ivp(lambda t, y: dydt(t, y, k1, k2), (t0, tf), y0, dense_output=True)
```

Обыкновенное дифференциальное уравнение второго порядка

Для решения обыкновенного дифференциального уравнения (ОДУ) более высокого порядка, чем первый, необходимо сначала привести его к системе ОДУ первого порядка. В общем случае любое дифференциальное уравнение с одной зависимой переменной порядка n можно записать как систему n дифференциальных уравнений первого порядка с n зависимыми переменными.

Например, уравнение, описывающее действие генератора гармонических колебаний, представляет собой дифференциальное уравнение второго порядка:

$$d^2x/dt^2 = -\omega^2 x,$$

где x – смещение относительно положения равновесия, ω – угловая частота. Это уравнение можно разложить на два дифференциальных уравнения первого порядка, как показано ниже:

$$dx_1/dt = x_2,$$

$$dx_2/dt = -\omega^2 x_1,$$

где x_1 идентифицируется с помощью x и x_2 через dx/dt.

Эту пару взаимосвязанных ДУ первого порядка можно решить, как показано в предыдущем разделе (см. листинг 8.11).

Листинг 8.11. Решение, описывающее действие генератора гармонических колебаний

```
import numpy as np
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt

# Частота генератора гармонических колебаний (1/с).
omega = 0.9
# Начальные условия для x1 = x и x2 = dx/dt в момент времени t = 0.
A, v0 = 3, 0        # см, см/с.
x0 = A, v0

# Правильно выбранная сетка точек времени.
t0, tf = 0, 20

def dxdt(t, x):
    """ Return dx/dt = f(t, x) at time t. """
    """ Возвращает dx/dt = f(t, x) в момент времени t. """
    x1, x2 = x
    dx1dt = x2
    dx2dt = -omega**2 * x1
    return dx1dt, dx2dt

# Интегрирование дифференциального уравнения.
soln = solve_ivp(dxdt, (t0, tf), x0, dense_output=True)
t = np.linspace(t0, tf, 100)
x1, x2 = soln.sol(t)

# Построение графика и сравнение численного и точного (аналитического) решений.
plt.plot(t, x1, 'o', color='k', label=r'\texttt{solve_ivp()}')
plt.plot(t, A * np.cos(omega * t), color='gray', label='Exact')
plt.xlabel(r'$t\;/\mathrm{s}$')
plt.ylabel(r'$x\;/\mathrm{cm}$')
plt.legend()
plt.show()
```

Полученный в результате выполнения кода из листинга 8.11 график показан на рис. 8.12.

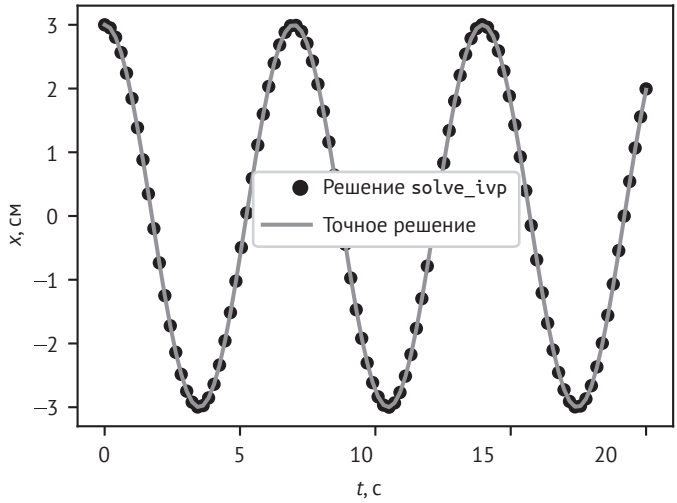


Рис. 8.12. Генератор гармонических колебаний: численное и точное (аналитическое) решения

Пример П8.16. На объект, медленно опускающийся в вязкой жидкости под воздействием силы тяжести, действует сила сопротивления (сила сопротивления Стокса), которая изменяется линейно относительно скорости движения объекта. Уравнение движения можно записать как дифференциальное уравнение второго порядка:

$$m(d^2z/dt^2) = -c(dz/dt) + mg',$$

где z – положение объекта как функция от времени t, c – константа лобового сопротивления, которая зависит от формы объекта и вязкости жидкости, и

$$g' = g(1 - \rho_{fluid}/\rho_{obj})$$

– действующее (в данных условиях) ускорение свободного падения, которое рассчитывается для выталкивающей силы, действующей в рассматриваемой жидкости (плотность ρ_{fluid}), смещаемой объектом (плотность ρ_{obj}). Для небольшой сферы радиусом r в жидкости с вязкостью η закон Стокса прогнозирует $c = 6\pi\eta r$.

Рассмотрим сферу из платины ($\rho = 21.45$ г/см^3) радиусом 1 мм, изначально находящуюся в покое, опускающуюся в ртути ($\rho = 13.53$ г/см^3, $\eta = 1.53 \times 10^{-3}$ Па·с). Приведенное выше дифференциальное уравнение второго порядка может быть решено аналитически, но для интегрирования в численном виде с использованием метода `solve_ivp` это уравнение обязательно необходимо представить как два ОДУ первого порядка:

$$\frac{\mathrm{d}z}{\mathrm{d}t} = \dot{z},$$
$$\frac{\mathrm{d}^2\dot{z}}{\mathrm{d}t^2} = \frac{\mathrm{d}\dot{z}}{\mathrm{d}t} = g' - \frac{c}{m}\dot{z}.$$

В коде из листинга 8.12 функция `deriv` вычисляет эти производные и передается в метод `solve_ivp` с начальными условиями ($z = 0$, $\dot{z} = 0$) и сеткой точек времени.

Листинг 8.12. Вычисление движения сферы, опускающейся под воздействием силы тяжести и выталкивающей силы Стокса

```
# eg8-stokes-drag.py
import numpy as np
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt

# Платиновая сфера, опускающаяся из состояния покоя в ртути.

# Ускорение свободного падения (м/с2).
g = 9.81
# Плотности (кг/м3).
rho_Pt, rho_Hg = 21450, 13530
# Вязкость ртути (Па.с).
eta = 1.53e-3

# Радиус и масса сферы.
r = 1.e-3 # radius (m)
m = 4*np.pi/3 * r**3 * rho_Pt
# Константа лобового сопротивления по закону Стокса.
c = 6 * np.pi * eta * r
# Действующее в данных условиях ускорение свободного падения.
gp = g * (1 - rho_Hg/rho_Pt)

def deriv(t, z):
    """ Return the dz/dt and d2z/dt2. """
    """ Возвращает dz/dt и d2z/dt2. """
    dz0 = z[1]
    dz1 = gp - c/m * z[1]
    return dz0, dz1

t0, tf = 0, 20
t_eval = np.linspace(t0, tf, 50)
# Начальные условия: z = 0, dz/dt = 0 в момент времени t = 0.
z0 = (0, 0)

# Интегрирование двух дифференциальных уравнений.
sol = solve_ivp(deriv, (t0, tf), z0, t_eval=t_eval)
t = sol.t
z, zdot = sol.y
plt.plot(t, zdot)

print('Estimate of terminal velocity = {:.3f} m.s-1'.format(zdot[-1]))

# Точное решение: конечная скорость в вязкой среде vt (м/с) характеристическое время tau (с).
v0, vt, tau = 0, m*gp/c, m/c
print('Exact terminal velocity = {:.3f} m.s-1'.format(vt))
z = vt*t + v0*tau*(1-np.exp(-t/tau)) + vt*tau*(np.exp(-t/tau)-1)
zdot_exact = vt + (v0-vt)*np.exp(-t/tau)
plt.plot(t, zdot_exact)
plt.xlabel('$t$ /s')
plt.ylabel('$\dot{z}\;/\mathrm{m\, s^{-1}}$')
plt.show()
```

График, полученный при выполнении этой программы, показан на рис. 8.13: результаты численного и аналитического решений неразличимы в заданном здесь масштабе, но приведены до трех знаков после десятичной точки при выводе:

```
Estimate of terminal velocity = 11.266 m.s-1
Exact terminal velocity = 11.285 m.s-1
```

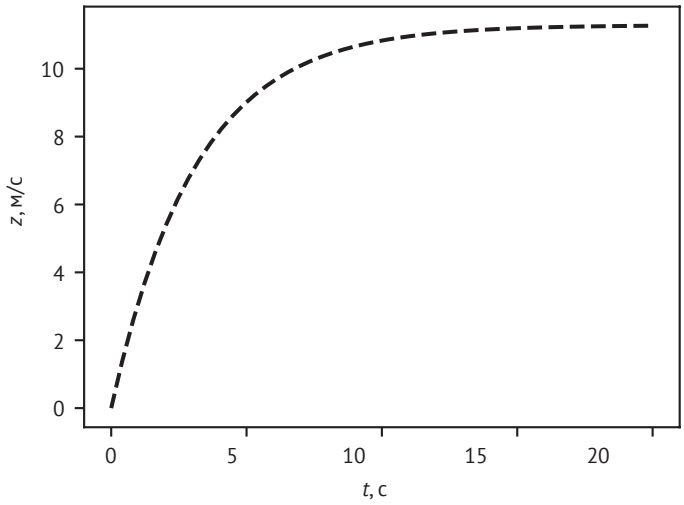


Рис. 8.13. График скорости платиновой сферы, опускающейся в ртути, как функция от времени, смоделированная по закону Стокса

Метод `solve_ivp` можно сконфигурировать для использования различных алгоритмов для решения системы ОДУ с помощью установки значения для атрибута `method`. Список возможных вариантов значений приведен в табл. 8.3. По умолчанию установлено значение `'RK45'` – явный метод Рунге-Кутты порядка 5(4) – это надежная обобщенная методика для нежестких задач.

Таблица 8.3. Методы интегрирования ОДУ, определяемые для scipy.integrate.solve_ivp

Метод	Описание
`'RK45'`	Явный метод Рунге-Кутты порядка 5(4)
`'RK23'`	Явный метод Рунге-Кутты порядка 3(2)
`'Radau'`	Неявный метод Рунге-Кутты семейства Радау Radau IIA порядка 5: подходят для решения жестких задач
`'BDF'`	Приближенная формула дифференцирования назад (BDF) (обратного дифференцирования), явный метод, подходящий для решения жестких задач
`'LSODA'`	Гибкий метод, который может автоматически определять жесткость задачи и переключаться между методом (алгоритмом) Адамса (для нежестких задач) и BDF (для жестких задач)

Задача называется жесткой, если требуется численный метод для принятия бесконечно малых шагов в ее интервалах интегрирования по отношению к сглаженности точного основного (аналитического) решения. Жесткие задачи часто возникают, когда члены в ОДУ представляют переменную, изменяющуюся по величине

в сильно различающихся шкалах времени. Методы 'Radau', 'BDF' и 'LSODA' заслуживают рассмотрения для пробного применения, если вы предполагаете, что ОДУ является жесткой задачей (системой), как показано в примере П8.17.

Пример П8.17. Классический пример жесткой системы ОДУ – кинетический анализ автокаталитической химической реакции Робертсона[113], включающей три разновидности $x = [X]$, $y = [Y]$, $z = [Z]$ при начальных условиях $x = 1$, $y = z = 0$:

$$\dot{x} \equiv \frac{dx}{dt} = -0.04x + 10^4 yz,$$
$$\dot{y} \equiv \frac{dy}{dt} = 0.04x - 10^4 yz - 3 \times 10^7 y^2,$$
$$\dot{z} \equiv \frac{dz}{dt} = 3 \times 10^7 y^2.$$

Следует отметить, что шкалы времени этих реакций значительно отличаются друг от друга, особенно для [Y].

По алгоритму Рунге-Кутты на интервале времени [0, 500]:

```
def deriv(t, y):
    x, y, z = y
    xdot = -0.04 * x + 1.e4 * y * z
    ydot = 0.04 * x - 1.e4 * y * z - 3.e7 * y**2
    zdot = 3.e7 * y**2
    return xdot, ydot, zdot

t0, tf = 0, 500
y0 = 1, 0, 0
soln = solve_ivp(deriv, (t0, tf), y0)
print(soln)
```

В итоге получаем:

```
 message: 'The solver successfully reached the end of the integration interval.'
        ('Механизм решения успешно достиг конца интервала интегрирования.')
    nfev: 6123410
    njev: 0
     nlu: 0
     sol: None
  status: 0
 success: True
       t: array ([0.00000000e+00, 6.36669332e-04, 1.06518798e-03, ...,
                  4.99999288e+02, 4.99999819e+02, 5.00000000e+02])
t_events: None
       y: array ([[1.00000000e+00, 9.99974534e-01, 9.99957394e-01, ...,
                   4.19780946e-01, 4.19780771e-01, 4.19780487e-01],
                  [0.00000000e+00, 2.20107324e-05, 3.00616449e-05, ...,
                   2.41400796e-06, 2.47838908e-06, 2.72514279e-06],
                  [0.00000000e+00, 3.45561028e-06, 1.25439771e-05, ...,
                   5.80216640e-01, 5.80216750e-01, 5.80216788e-01]])
```

[113] *H. H. Robertson*. The solution of a set of reaction rate equations, in J. Walsh (Ed.), Numerical Analysis: An Introduction, pp. 178–182, Academic Press, London (1966).

но это стоило 6 млн вызовов функции `soln.nfev`. Метод `'Radau'` обходится дешевле и работает гораздо быстрее:

Листинг 8.13. Решение системы химических реакций Робертсона

```
import numpy as np
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt

def deriv(t, y):
    """ODEs for Robertson ' s chemical reaction system."""
    """Система ОДУ для химических реакций Робертсона."""
    x, y, z = y
    xdot = -0.04 * x + 1.e4 * y * z
    ydot = 0.04 * x - 1.e4 * y * z - 3.e7 * y**2
    zdot = 3.e7 * y**2
    return xdot , ydot , zdot

# Начальный и конечный моменты времени.
t0, tf = 0, 500
# Начальные условия: [X] = 1; [Y] = [Z] = 0.
y0 = 1, 0, 0
# Решение с использованием метода, позволяющего привести задачу к жесткой системе ОДУ.
soln = solve_ivp(deriv, (t0, tf), y0, method='Radau')
print(soln.nfev, 'evaluations required.')

# Построение графика концентраций как функции от времени. Масштабирование [Y] по 10**YFAC,
# чтобы вариации были видимы на оси, используемой для [X] и [Z].
YFAC = 4
plt.plot(soln.t, soln.y[0], label='[X]')
plt.plot(soln.t, 10**YFAC*soln.y[1], label=r'$10^{}\times$[Y]'.format(YFAC))
plt.plot(soln.t, soln.y[2], label='[Z]')
plt.xlabel('time /s')
plt.ylabel('concentration /arb. units')
plt.legend()
plt.show()
```

Для вывода результата потребовалось всего лишь 248 вызовов функции. Значения концентрации [*X*], [*Y*] и [*Z*], изменяющиеся по времени, показаны на рис. 8.14.

Метод `solve_ivp` также может обнаруживать события, возникающие при интегрировании системы дифференциальных уравнений, и реагировать на них определенным образом. В аргументе `events` можно передавать одну или несколько функций, которые должны возвращать ноль, если состояние системы соответствует возникшему событию. В качестве весьма простого примера рассмотрим движение автомобиля со скоростью $v = 20$ м/с, на который начинает воздействовать сила торможения, замедляющая его движение с постоянным отрицательным ускорением $a = dv/dt = -3$ м/с^2. За какое время автомобиль остановится полностью? Аналитически определяемый ответ очевиден: 20/3 = 6.67 с.

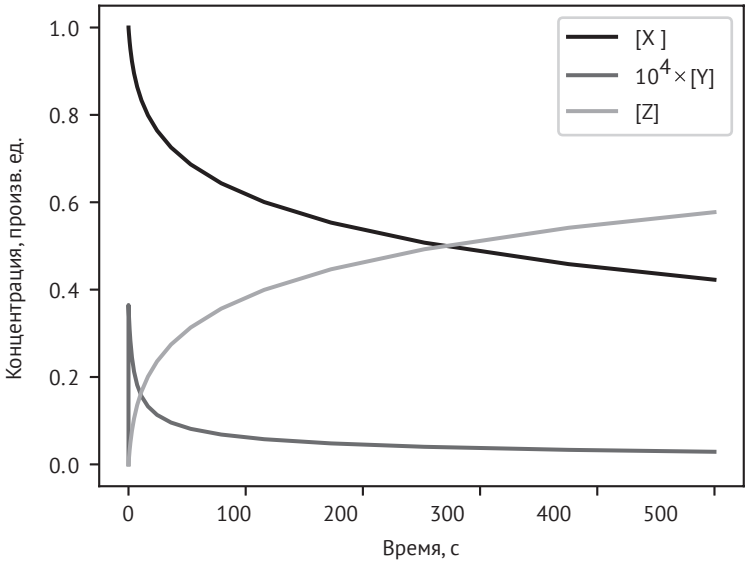


Рис. 8.14. Система химических уравнений (реакций) Робертсона, проинтегрированная численно методом Радау Radau IIA

Для решения задачи в численном виде можно определить функцию события, которая возвращает скорость автомобиля: время, при котором эта функция возвращает 0, присваивается переменной объекта решения `t_events`:

```
In [x]: def car_stopped(t, y):
...:        return y[0]
...:
In [x]: t0, tf = 0, 100     # Общий рассматриваемый интервал времени, с.
In [x]: v0 = 20             # Начальная скорость, м/с.
In [x]: solve_ivp(lambda t, y: -3, (t0, tf), [v0], events=car_stopped)
Out[x]:
  message: 'The solver successfully reached the end of the integration interval.'
          ('Механизм решения успешно достиг конца интервала интегрирования.')
     nfev: 26
     njev: 0
      nlu: 0
      sol: None
   status: 0
  success: True
        t: array([ 0.       ,   0.14614572,    1.60760288,   16.22217454,
                 100.      ])
 t_events: [array ([6.66666667])]
        y: array([[ 20.     ,    19.56156285,    15.17719135,   -28.66652362,
                 -280.       ]])
 y_events: [array([[ -1.77635684e-15]])]
```

Следует отметить, что время полной остановки, содержащееся в массиве `t_events`, вычислено точно: оно получено с использованием алгоритма поиска корней после того, как процесс интегрирования ОДУ обнаружил изменение знака значения, возвращаемого из функции события `car_stopped`. Тем не менее процесс интегрирования был продолжен и после этого момента времени для получения решений, не имеющих смысла с точки зрения физики (после остановки автомобиль не двигается назад). Можно принудительно заставить метод `solve_ivp` прекратить вычисления после заданного события, добавив логический объект `terminal=True`:

```
In [x]: car_stopped.terminal = True
In [x]: solve_ivp(lambda t, y: -3, (t0, tf), [v0], events=car_stopped)
Out[x]:
 message: 'A termination event occurred.'
         ('Обнаружено событие завершения вычислений.')
    nfev: 20
    njev: 0
     nlu: 0
     sol: None
  status: 1
 success: True
       t: array([0.       ,  0.14614572,  1.60760288,  6.66666667])
t_events: [array ([6.66666667])]
       y: array([[ 2.00000000e+01,   1.95615629e+01,   1.51771914e+01,
                  -1.77635684e-15]])
```

Атрибут `terminal=True` обязательно должен быть присоединен к функции `car_stopped` после ее определения, иначе он не будет доступен методу `solve_ivp`[114].

Пример П8.18. Сферический снаряд массой m с некоторой начальной скоростью перемещается под воздействием двух сил: гравитации $\boldsymbol{F}_g = -mg\hat{z}$ и силы сопротивления воздуха (замедляющей движение) $\boldsymbol{F}_D = -1/2c\rho Av^2\boldsymbol{v}/|\boldsymbol{v}| = -1/2c\rho Av\boldsymbol{v}$, действующей в направлении, противоположном вектору скорости снаряда, и пропорциональной квадрату этой скорости (при наиболее реалистичных условиях). Здесь c – коэффициент сопротивления воздуха, ρ – плотность воздуха, A – площадь поперечного сечения снаряда.

Таким образом, система уравнений движения снаряда имеет следующий вид:

$$m\ddot{x} = -k\sqrt{\dot{x}^2 + \dot{z}^2}\dot{x},$$
$$m\ddot{z} = -k\sqrt{\dot{x}^2 + \dot{z}^2}\dot{z} - mg,$$

где $v = |\boldsymbol{v}| = \sqrt{\dot{x}^2 + \dot{z}^2}$ и $k = 1/2c\rho A$. Эти уравнения можно разложить в следующую систему из четырех ОДУ первого порядка при $u_1 \equiv x$, $u_2 \equiv \dot{x}$, $u_3 \equiv z$, $u_4 \equiv \dot{z}$:

[114] Этот тип изменения функции после ее определения стал доступен в Python с версии 2.1 (документ PEP 232) как пример monkey-patching («обезьяний», или «партизанский», патч, изменяющий поведение программы во время выполнения). Некоторые программисты считают это плохим стилем, так как данный прием отделяет определение метода от его реальной функциональности.

$$\dot{u}_1 = u_2,$$
$$\dot{u}_2 = -\frac{k}{m}\sqrt{u_2^2 + u_4^2}\,u_2,$$
$$\dot{u}_3 = u_4,$$
$$\dot{u}_4 = -\frac{k}{m}\sqrt{u_2^2 + u_4^2}\,u_4 - mg.$$

Код в листинге 8.14 интегрирует эту систему уравнений и определяет два события: попадание в цель (снаряд возвращается на поверхность земли при $z = 0$) и достижение максимальной высоты (в которой компонента z скорости снаряда равна нулю). Устанавливается дополнительный атрибут `hit_target.direction = -1`, гарантирующий, что `hit_target` сгенерирует событие, только если возвращаемое значение (высота полета снаряда) изменяет свой знак с положительного на отрицательный, иначе событие сгенерируется при запуске снаряда, так как $z_0 = 0$. Другие варианты: `direction = 1` – генерируется событие, когда возвращаемое значение меняет свой знак с отрицательного на положительный или `direction = 0` (по умолчанию) – событие генерируется, когда возвращаемое значение равно нулю по любому направлению.

Листинг 8.14. Вычисление и построение графика траектории сферического снаряда с учетом сопротивления воздуха

```
import numpy as np
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt

# Коэффициент сопротивления воздуха, радиус снаряда (м), площадь (м2) и масса (кг).
c = 0.47
r = 0.05
A = np.pi * r**2
m = 0.2
# Плотность воздуха (кг/м3), ускорение свободного падения (м/с2).
rho_air = 1.28
g = 9.81
# Эта константа определяется для удобства вычислений.
k = 0.5 * c * rho_air * A

# Начальная скорость и угол вылета снаряда (от горизонтальной плоскости).
v0 = 50
phi0 = np.radians(65)

def deriv(t, u):
    x, xdot, z, zdot = u
    speed = np.hypot(xdot, zdot)
    xdotdot = -k/m * speed * xdot
    zdotdot = -k/m * speed * zdot - g
    return xdot, xdotdot, zdot, zdotdot
```

```
# Начальные условия: x0, v0_x, z0, v0_z.
u0 = 0, v0 * np.cos(phi0), 0., v0 * np.sin(phi0)
# Интегрирование до момента времени tf, если раньше не случится попадание в цель.
t0, tf = 0, 50

def hit_target(t, u):
    # Попадание в цель фиксируется, если координата z равна 0.
    return u[2]

# Прекращение процесса интегрирования, если зафиксировано попадание в цель.
hit_target.terminal = True
# Обязательно необходимо движение со снижением (не останавливаться до начала
# движения с повышением)
hit_target.direction = -1

def max_height(t, u):
    # Максимальная высота достигается, когда компонента скорости z равна нулю.
    return u[3]

soln = solve_ivp(deriv, (t0, tf), u0, dense_output=True, events=(hit_target, max_height))
print(soln)
print('Time to target = {:.2f} s'.format(soln.t_events [0][0]))
print('Time to highest point = {:.2f} s'.format(soln.t_events [1][0]))

# Правильно выбранная сетка точек времени от 0 до момента попадания в цель.
t = np.linspace(0, soln.t_events[0][0], 100)

# Получение решения по сетке точек времени и построение графика траектории.
sol = soln.sol(t)
x, z = sol[0], sol[2]
print('Range to target, xmax = {:.2f} m'.format(x[-1]))
print('Maximum height, zmax = {:.2f} m'.format(max(z)))
plt.plot(x, z)
plt.xlabel('x /m')
plt.ylabel('z /m')
plt.show()
```

Вывод результата:

```
Time to target = 6.34 s
Time to highest point = 2.79 s
Range to target, xmax = 64.12 m
Maximum height, zmax = 49.42 m
```

Созданный график траектории полета снаряда показан на рис. 8.15.

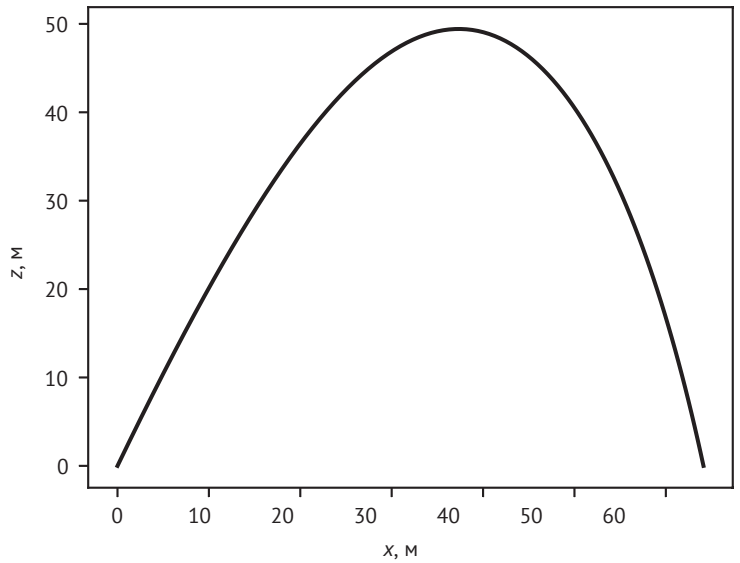


Рис. 8.15. Траектория полета сферического снаряда, выпущенного с начальной скоростью v_0 = 50 м/с под углом φ_0 = 50° с учетом сопротивления воздуха

8.2.4 Упражнения

Вопросы

B8.2.1. Использовать метод `scipy.integrate.quad` для вычисления следующего интеграла:

$$\int_0^6 \lfloor x \rfloor - 2\left\lfloor \frac{x}{2} \right\rfloor \mathrm{d}x.$$

B8.2.2. Использовать метод `scipy.integrate.quad` для вычисления следующих определенных интегралов (большинство из них можно также представить в конечной форме в заданном интервале, но это затруднительно).

а)

$$\int_0^1 \frac{x^4(1-x)^4}{1+x^2}\,\mathrm{d}x.$$

(Сравнить с 22/7 − π.)

б) Этот интеграл применяется в теории (модели) Дебая, описывающей теплоемкость кристаллических веществ при низких температурах:

$$\int_0^\infty \frac{x^3}{e^x - 1}\,\mathrm{d}x.$$

(Сравнить с $\pi^4/15$.)

в) Интеграл, который иногда называют мечтой второкурсника:

$$\int_0^1 x^{-x}\ \mathrm{d}x.$$

(Сравнить полученное значение с суммой $\sum_{n=1}^{\infty} n^{-n}$.)

г)
$$\int_0^1 \left[\ln(1/x)\right]^P\ \mathrm{d}x.$$

(Сравнить с $p!$ для целочисленного $0 \leqslant p \leqslant 10$.)

д)
$$\int_0^{2\pi} e^{z\cos\theta}\ \mathrm{d}\theta.$$

(Сравнить с $2\pi I_0(z)$, где $I_0(z)$ – модифицированная функция Бесселя первого рода при $0 \leqslant z \leqslant 2$.)

B8.2.3. Использовать метод `scipy.integrate.dblquad` для вычисления π методом интегрирования постоянной функции $f(x,y) = 4$ в четверти круга с единичным радиусом в квадранте $x > 0$, $y > 0$.

B8.2.4. Что сделано неправильно в приведенной ниже попытке вычисления площади единичного круга (π) как двойного интеграла в полярных координатах?

```
In [x]: dblquad(lambda r, theta: r, 0, 1, lambda r: 0, lambda r: 2*np.pi)
Out[x]: (19.739208802178712, 2.1914924100062363e-13)
```

Задачи

38.2.1. Площадь поверхности вращения относительно оси x в интервале от a до b, определяемой функцией $y = f(x)$, выражается с помощью интеграла

$$S = 2\pi\int_a^b y\ \mathrm{d}s,\ \text{где}\ \mathrm{d}s = \sqrt{1+\left(\frac{dy}{dx}\right)^2}\ \mathrm{d}x.$$

Использовать это выражение при написании метода вычисления площади поверхности вращения для функции $y = f(x)$ относительно оси x, применяя объекты функций Python, которые возвращают y и dy/dx. Проверить метод для параболоида, полученного вращением параболы $f(x) = \sqrt{x}$ относительно оси x в интервале $a = 0$, $b = 1$. Сравнить с точным результатом $\pi(5^{3/2} - 1)/6$.

38.2.2. Интеграл от функции секанса

$$\int_0^{\theta} \sec\phi\ \mathrm{d}\phi$$

при $-\pi/2 < \theta < \pi/2$ весьма важен в навигации и в теории картографических проекций. Его можно записать в конечной форме как обратную функцию Гудермана:

$$gd^{-1}(\theta) = \ln|\sec\theta + \tan\theta|.$$

Использовать метод `scipy.integrate.quad` для вычисления значений этого интеграла в важном интервале θ, приведенном выше, и сравнить графически с точным решением.

38.2.3. Рассмотреть тор с равномерной плотностью, единичной массой, радиусом средней линии R и радиусом поперечного сечения r. Объем и моменты инерции такого тора можно вычислить аналитически и получить следующие результаты:

$$V = 2\pi^2 R r^2,$$
$$I_z = R^2 + 3/4r^2,$$
$$I_x = I_y = 1/2R^2 + 5/8r^2,$$

где центр масс тора совпадает с началом координат, а ось z принята за ось симметрии тора.

Для решения этой задачи применяется численный метод. В цилиндрических координатах (ρ, θ, z) объем и моменты инерции можно записать так:

$$V = 2\int_0^{2\pi}\int_{R-r}^{R+r}\int_0^{\sqrt{r^2-(\rho-R)^2}} \rho\,dz\,d\rho\,d\theta,$$
$$I_z = \frac{2}{V}\int_0^{2\pi}\int_{R-r}^{R+r}\int_0^{\sqrt{r^2-(\rho-R)^2}} \rho^3\,dz\,d\rho\,d\theta,$$
$$I_x = I_y = \frac{2}{V}\int_0^{2\pi}\int_{R-r}^{R+r}\int_0^{\sqrt{r^2-(\rho-R)^2}} (\rho^2\sin^2\theta+z^2)\rho\,dz\,d\rho\,d\theta.$$

Вычислить эти интегралы для тора с размерами R = 4, r = 1 и сравнить с точными значениями.

38.2.4. Брюсселятор (Brusselator) – это теоретическая модель для автокаталитической реакции. Модель предполагает следующий ход реакции, в которой компоненты A и B считаются имеющими постоянную концентрацию, а компоненты D и E удаляются по мере их создания. Значения концентрации компонентов X и Y могут демонстрировать колебательное поведение (возникновение небольших локальных пиков) при определенных условиях.

$$\begin{aligned} A &\to X && k_1,\\ 2X + Y &\to 3X && k_2,\\ B + X &\to Y + D && k_3,\\ X &\to E && k_4. \end{aligned}$$

Для удобства вводятся масштабированные количества:

$$x = [\mathrm{X}]\sqrt{\frac{k_2}{k_4}},\ y = [\mathrm{Y}]\sqrt{\frac{k_2}{k_4}},$$
$$a = [\mathrm{A}]\frac{k_1}{k_4}\sqrt{\frac{k_2}{k_4}},\ b = [\mathrm{B}]\frac{k_3}{k_4},$$

и масштабирование времени по коэффициенту k_4, что приводит к безразмерным уравнениям:

$$dx/dt = a - (1 - b)x + x^2y,$$
$$dy/dt = bx - x^2y.$$

Показать, как приведенные выше уравнения прогнозируют изменение x и y: а) при $a = 1$, $b = 1.8$ и б) при $a = 1$, $b = 2.02$, построив график для каждого случая: i) x, y как функции от (безразмерного) времени и ii) y как функции от x.

38.2.5. Уравнение, описывающее колебания маятника, состоящего из некоторой массы на конце легкого жесткого стержня длиной l, можно записать в следующем виде:

$$d^2\theta/dt^2 = -(g/l)\sin\theta,$$

где θ – угол отклонения маятника от вертикали.

Приняв $l = 1$ м и $g = 9.81$ м/с^2, определить дальнейшее движение маятника, если оно начинается из состояния покоя с начальным углом $\theta_0 = 30°$. Сравнить это (колебательное) движение с гармоническим приближением, вычисляемым с предположением о том, что угол θ имеет малую величину. Для такого гармонического приближения существует аналитическое решение $\theta = \theta_0 \cos(\omega t)$ при $\omega = \sqrt{(g/l)}$.

38.2.6. Простой механизм образования озона в стратосфере состоит из следующих четырех реакций (известных как цикл Чепмена):

$O_2 + h\nu \rightarrow 2O$	$k_1 = 3 \times 10^{-12}$ 1/с,
$O_2 + O + M \rightarrow O_3 + M$	$k_2 = 1.2 \times 10^{-33}$ см^6/молекула^2·с
$O_3 + h\nu' \rightarrow O + O_2$	$k_3 = 5.5 \times 10^{-4}$ 1/с,
$O + O_3 \rightarrow 2O_2$	$k_4 = 6.9 \times 10^{-16}$ см^3/молекула·с,

где M – не участвующее в реакции третье тело, присутствие которого предполагается в общей молекулярной концентрации воздуха на рассматриваемой высоте. Из этих реакций выводятся следующие кинетические уравнения (скоростей) реакций для [O], [O_3] и [O_2]:

$$d[O_2]/dt = -k_1[O_2] - k_2[O_2][O][M] + k_3[O_3] + 2k_4[O][O_3],$$
$$d[O]/dt = 2k_1[O_2] - k_2[O_2][O][M] + k_3[O_3] - k_4[O][O_3],$$
$$d[O_3]/dt = k_2[O_2][O][M] - k_3[O_3] - k_4[O][O_3].$$

Эти константы скоростей реакции (коэффициенты кинетики) применяются на высоте 25 км, где [M] = 9×10^{17} молекул/см^3. Написать программу для определения концентраций O_3 и O как функции от времени на этой высоте (необходимо считать, что значение $[O_2]$ остается практически постоянным). Начальные условия $[O_2]_0 = 0.21M$, $[O]_0 = [O_3]_0 = 0$, интегрирование выполнить для интервала 10^8 с (начать с полного отсутствия и учесть, что для создания озонового слоя по описанному выше механизму требуется около трех лет). Сравнить равновесные концентрации с результатом, полученным аналитическим приближенным методом с использованием приближенной методики расчета стационарного процесса:

$$[O_3] = \sqrt{\frac{k_1 k_2}{k_3 k_4}}[O_2][M]^{1/2}, \quad \frac{[O]}{[O_3]} = \frac{k_3}{k_2[O_2][M]}.$$

38.2.7. Гиперион – спутник Сатурна неправильной (несферической) формы, известный своим хаотическим вращением. Его движение можно смоделировать в соответствии с приведенным ниже описанием.

Орбита Гипериона (H) вокруг Сатурна (S) представляет собой эллипс с главной полуосью *a* и эксцентриситетом *e*. Пусть P – это точка максимального приближения Гипериона к Сатурну (периапсида). Ее расстояние от планеты SH является функцией от истинной аномалии (орбитального угла φ, измеряемого относительно прямой SP), следовательно:

$$r = a(1 - e^2) / (1 + e \cos \varphi).$$

Определить угол θ между осью наименьшего главного момента инерции (в общем смысле это самая длинная ось спутника) и прямой SP, а также количественное значение Ω, являющееся масштабированным коэффициентом изменения θ относительно φ (т. е. скорость, с которой Гиперион вращается по орбите Сатурна), по следующей формуле:

$$\Omega = a^2/r^2 \, d\theta/d\phi.$$

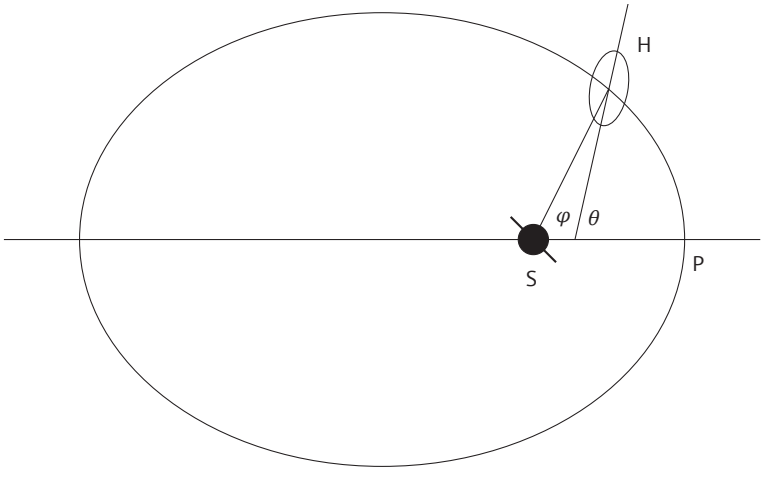


Рис. 8.16

Далее это можно записать в виде:

$$d\Omega/d\varphi = -(B - A)/C \;\; 3/2(1 - r^2) \;\; a/r \;\; \sin[2(\theta - \varphi)],$$

где A, B, C – главные моменты инерции.

Использовать метод `scipy.integrate.solv_ivp` для вычисления и построения графика скорости вращения Ω как функции от φ при начальных условиях: а) $\varphi = \Omega = 0$, б) $\theta = 0$, $\Omega = 2$ при $\varphi = 0$. Принять $e = 0.1$ и $(B - A)/C = 0.265$.

38.2.8. Цепь радиоактивного распада ^{212}Pb до стабильного изотопа ^{208}Pb можно рассматривать как показанную ниже последовательность шагов с известными постоянными скорости распада k_i:

$$\begin{aligned}
&^{212}\text{Pb} \rightarrow {}^{212}\text{Bi} + \beta^- && k_1 = 1.816 \times 10^{-5}\ 1/\text{c},\\
&^{212}\text{Bi} \rightarrow {}^{208}\text{Tl} + \alpha && k_2 = 6.931 \times 10^{-5}\ 1/\text{c},\\
&^{212}\text{Bi} \rightarrow {}^{212}\text{Po} + \beta^- && k_3 = 1.232 \times 10^{-4}\ 1/\text{c},\\
&^{208}\text{Tl} \rightarrow {}^{208}\text{Pb} + \beta^- && k_4 = 3.851 \times 10^{-3}\ 1/\text{c},\\
&^{212}\text{Po} \rightarrow {}^{208}\text{Pb} + \alpha && k_5 = 2.310\ 1/\text{c}.
\end{aligned}$$

С учетом показанных ниже дифференциальных уравнений первого порядка, описывающих скорости изменения для каждого элемента, построить графики их концентраций как функции от времени:

$$\begin{gathered}
d[^{212}\text{Pb}]/dt = -k_1[^{212}\text{Pb}],\\
d[^{212}\text{Bi}]/dt = k_1[^{212}\text{Pb}] - k_2[^{212}\text{Bi}] - k_3[^{212}\text{Bi}],\\
d[^{208}\text{Tl}]/dt = k_2[^{212}\text{Bi}] - k_4[^{208}\text{Tl}],\\
d[^{212}\text{Po}]/dt = k_3[^{212}\text{Bi}] - k_5[^{212}\text{Po}],\\
d[^{208}\text{Pb}]/dt = k_4[^{208}\text{Tl}] + k_5[^{212}\text{Po}].
\end{gathered}$$

Если все промежуточные компоненты реакций J интерпретировать как «находящиеся в устойчивом состоянии» (т. е. $d[\text{J}]/dt = 0$), то приближенную формулу для вычисления концентрации ^{208}Pb как функции от времени можно записать так:

$$[^{208}\text{Pb}] = [^{212}\text{Pb}]_0(1 - e^{-k_1 t}).$$

Сравнить «точный» результат, полученный численным интегрированием приведенных выше дифференциальных уравнений, с результатом этого приближенного решения.

38.2.9. Простая модель распространения пламени зажженной спички учитывает радиус пламени y с изменениями во времени следующим образом:

$$dy/dt = \alpha y^2 - \beta y^3,$$

где α и β – некоторые константы, связанные с перемещением кислорода через поверхность пламени и скоростью его потребления внутри пламени. Изначально пламя имеет малый размер $y(0) \lll \alpha/\beta$, но начиная с некоторого мо-

мента быстро растет, пока не достигнет устойчивого состояния с постоянным радиусом (если предположить неограниченную подачу горючего материала).

Принять $\alpha = \beta = 1$ и решить это ОДУ в численном виде, используя метод `scipy.integrate.solve_ivp` с применением наиболее подходящей методики интегрирования на интервале времени $(0, 5/y(0))$ при а) $y(0) = 0.01$, б) $y(0) = 0.0001$. Сколько шагов интегрирования по времени необходимо принять в каждом случае?

Точное решение можно записать в следующем виде:

$$y(t) = \alpha / \beta[1 + W(ae^{a-\alpha^2 t/\beta})],$$

где $a = \alpha/(\beta y(0))$ и $W(x)$ – W-функция Ламберта, реализованная в библиотеке SciPy как метод `scipy.special.lambertw`. Сравнить точность различных вариантов численных решений с результатом вычисления по этой формуле.

8.3 Интерполяция

Пакет `scipy.interpolation` содержит множество разнообразных функций и классов для интерполяции и сплайнов в одном и в нескольких измерениях. Некоторые наиболее важные функции и классы рассматриваются в этом разделе.

8.3.1 Одномерная интерполяция

Функциональность самой простой и очевидной одномерной интерполяции предоставляет метод `scipy.interpolate.interp1d`. Метод принимает массивы точек `x` и `y`, а возвращает функцию, которую можно вызывать для генерации интерполируемых значений в промежуточных точках `x`. По умолчанию принята линейная схема интерполяции, но возможны и другие варианты схемы (см. табл. 8.4), как показано в примере П8.19.

Таблица 8.4. Методы (схемы) интерполяции, передаваемые в аргументе kind в метод scipy.interpolate.inerp1d

kind	Описание
`'linear'`	Принятая по умолчанию линейная интерполяция, использующая только значения из исходных массивов данных, охватывающих требуемую точку
`'nearest'`	«Притягивание» (привязка) к ближайшей точке данных
`'zero'`	Сплайн нулевого порядка: интерполирует по последнему наблюдаемому значению при проходе по массивам данных
`'slinear'`	Интерполяция сплайном первого порядка (на практике то же самое, что `'linear'`)
`'quadratic'`	Интерполяция сплайном второго порядка
`'cubic'`	Интерполяция кубическим сплайном
`'previous'`	Используется предыдущая точка данных
`'next'`	Используется следующая точка данных

Пример П8.19. Этот пример демонстрирует некоторые из методик интерполяции, доступных при использовании метода `scipy.interpolation.interp1d` (см. рис. 8.17).

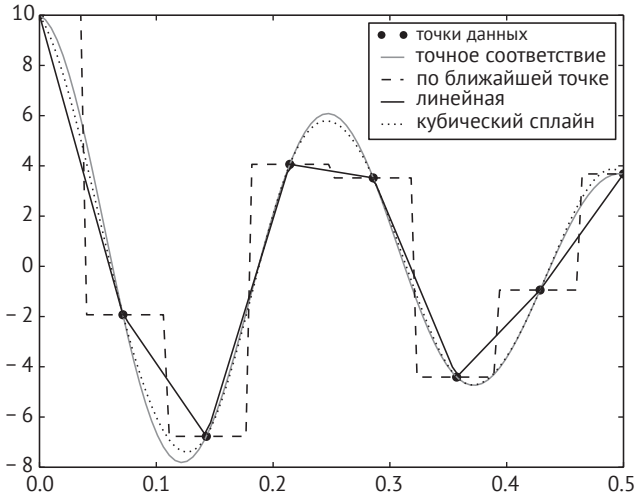


Рис. 8.17. Графическое отображение различных методик одномерной интерполяции при использовании метода scipy.interpolation.interp1d

Листинг 8.15. Сравнение типов одномерной интерполяции при использовании метода scipy.interpolation.interp1d

```
# eg8-interp1d.py
import numpy as np
from scipy.interpolate import interp1d
import matplotlib.pyplot as plt

A, nu, k = 10, 4, 2

def f(x, A, nu, k):
    return A * np.exp(-k*x) * np.cos(2*np.pi * nu * x)

xmax, nx = 0.5, 8
x = np.linspace(0, xmax, nx)
y = f(x, A, nu, k)

f_nearest = interp1d(x, y, kind='nearest')
f_linear = interp1d(x, y)
f_cubic = interp1d(x, y, kind='cubic')

x2 = np.linspace(0, xmax, 100)
plt.plot(x, y, 'o', label='data points')
plt.plot(x2, f(x2, A, nu, k), label='exact')
plt.plot(x2, f_nearest(x2), label='nearest')
plt.plot(x2, f_linear(x2), label='linear')
plt.plot(x2, f_cubic(x2), label='cubic')
plt.legend()
plt.show()
```

8.3.2 Многомерная интерполяция

Рассмотрим два типа многомерной интерполяции, соответствующих структурированному (т. е. с данными, размещенными по сетке некоторого типа) и неструктурированному источникам данных.

Интерполяция данных, структурированных по прямоугольной сетке

Самым простым методом двумерной интерполяции является `scipy.interpolation.interp2d`. Для него требуется двумерный массив значений `z` и два одномерных массива координат `x` и `y`, соответствующих значениям данных. В этих массивах не обязательно соблюдать постоянный интервал между значениями. В аргументе `kind` можно передать один из трех поддерживаемых типов сплайна интерполяции: `'linear'` (по умолчанию), `'cubic'` или `'quintic'`.

Пример П8.20. Здесь вычисляется функция

$$z(x,y) = \sin(\pi x/2)e^{y/2}$$

по сетке из точек (x,y), которые неравномерно распределены по направлению y. Затем используется метод `scipy.interpolate.interp2d` для интерполяции этих значений в более гладкую, равномерно распределенную сетку (x,y) (см. рис. 8.18).

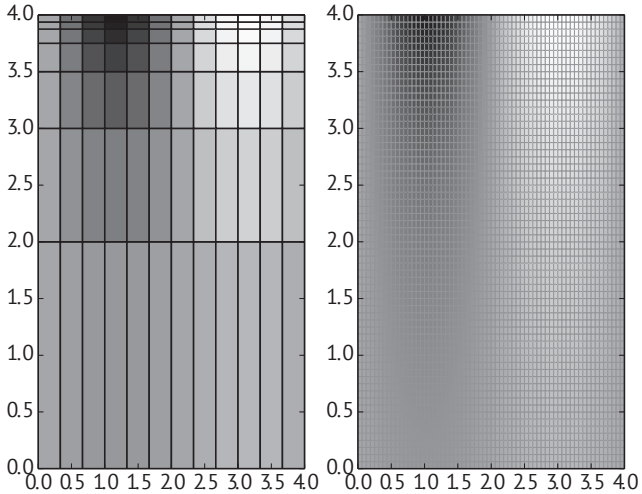


Рис. 8.18. Двумерная интерполяция с использованием метода scipy.interpolation.interp2d

Листинг 8.16. Двумерная интерполяция с использованием метода scipy.interpolation.interp2d

---

```
# eg8-interp2d.py
import numpy as np
from scipy.interpolate import interp2d
import matplotlib.pyplot as plt
```

```
x = np.linspace(0, 4, 13)
y = np.array([0, 2, 3, 3.5, 3.75, 3.875, 3.9375, 4])
X, Y = np.meshgrid(x, y)
Z = np.sin(np.pi*X/2) * np.exp(Y/2)

x2 = np.linspace(0, 4, 65)
y2 = np.linspace(0, 4, 65)
f = interp2d(x, y, Z, kind='cubic')                    ❶
Z2 = f(x2, y2)

fig, ax = plt.subplots(nrows=1, ncols=2)
ax[0].pcolormesh(X, Y, Z)

X2, Y2 = np.meshgrid(x2, y2)
ax[1].pcolormesh(X2, Y2, Z2)

plt.show()
```

❶ Обратите внимание: для метода `interp2d` требуются одномерные массивы `x` и `y`.

Если набор координат (x,y) образует равномерно распределенную сетку, то самым быстрым способом интерполяции значений из массива `z` является использование объекта `scipy.interpolate.RectBivariateSpline`, как показано в примере П8.21.

Пример П8.21. В коде из листинга 8.17 вычисляется функция

$$z(x,y) = e^{-4x^2}e^{-y^2/4}$$

по равномерной крупной сетке, затем выполняется интерполяция по сетке с более мелким шагом (см. рис. 8.19).

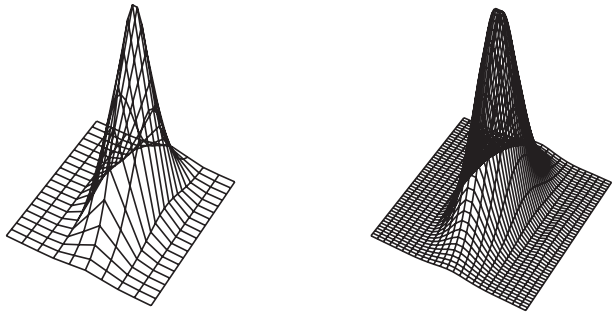


Рис. 8.19. Двумерная интерполяция с переходом от крупной прямоугольной сетки (изображение слева) к сетке с более мелким шагом (изображение справа) с использованием объекта scipy.interpolate.RectBivariateSpline

Листинг 8.17. Интерполяция с переходом к уплотненной равномерной двумерной сетке с использованием объекта scipy.interpolate.RectBivariateSpline

```
# eg8-RectBivariateSpline.py
import numpy as np
from scipy.interpolate import RectBivariateSpline
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Равномерная сетка с укрупненным шагом.
dx, dy = 0.4, 0.4
xmax, ymax = 2, 4
x = np.arange(-xmax, xmax, dx)
y = np.arange(-ymax, ymax, dy)
X, Y = np.meshgrid(x, y)
Z = np.exp(-(2*X)**2 - (Y/2)**2)

interp_spline = RectBivariateSpline(y, x, Z)                    ❶

# Равномерная сетка с более мелким шагом.
dx2, dy2 = 0.16, 0.16
x2 = np.arange(-xmax, xmax, dx2)
y2 = np.arange(-ymax, ymax, dy2)
X2, Y2 = np.meshgrid(x2, y2)
Z2 = interp_spline(y2, x2)

fig, ax = plt.subplots(nrows=1, ncols=2, subplot_kw={'projection': '3d'})
ax[0].plot_wireframe(X, Y, Z, color='k')

ax[1].plot_wireframe(X2, Y2, Z2, color='k')
for axes in ax:
    axes.set_zlim(-0.2, 1)
    axes.set_axis_off()

fig.tight_layout()
plt.show()
```

❶ Обратите внимание: для функции Z, определяемой здесь с использованием настройки meshgrid, метод RectBivariateSpline ожидает передачи соответствующих массивов y и x именно в таком порядке (в отличие от метода interp2d)[115].

Интерполяция неструктурированных данных

Для интерполяции неструктурированных данных, т. е. точек данных, размещенных в произвольных координатах (x,y) сетки, можно использовать метод scipy.interpolate.griddata. Основной способ применения для двух измерений:

```
scipy.interpolate.griddata(points, values, xi, method='linear')
```

[115] Проблема связана со способом индексирования сетки meshgrid, основанным на соглашениях, принятых для совместимости с MATLAB.

где предоставляемые данные передаются как одномерный массив `values` с координатами `points`, который передается как кортеж массивов `x` и `y` или как один массив формы `(n, 2)`, где `n` – длина массива `values`. Объект `xi` – это массив координатной сетки, по которой выполняется интерполяция (форма этого массива `(m, 2)`). Доступные методики (типы) интерполяции: `'linear'` (по умолчанию), `'nearest'` и `'cubic'`.

Пример П8.22. Код в листинге 8.18 демонстрирует применение различных типов (методик) интерполяции, доступных для `scipy.interpolate.griddata`, с использованием 400 точек, случайно выбранных из исследуемой функции. Результаты можно сравнить по рис. 8.20.

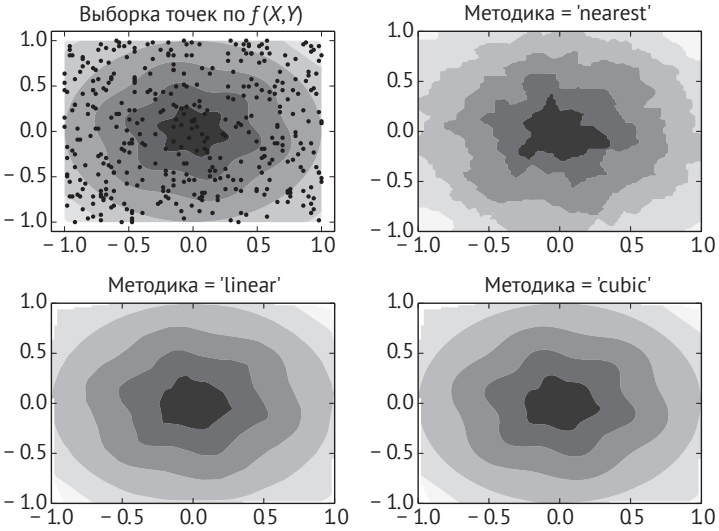


Рис. 8.20. Некоторые различные схемы интерполяции для метода scipy.interpolate.griddata

Листинг 8.18. Интерполяция по неструктурированному массиву двумерных точек с использованием scipy.interpolate.griddata

```
# eg8-gridinterp.py
import numpy as np
from scipy.interpolate import griddata
import matplotlib.pyplot as plt

x = np.linspace(-1, 1, 100)
y = np.linspace(-1, 1, 100)
X, Y = np.meshgrid(x, y)

def f(x, y):
    s = np.hypot(x, y)
    phi = np.arctan2(y, x)
    tau = s + s * (1 - s) / 5 * np.sin(6 * phi)
    return 5 * (1 - tau) + tau
```

```
T = f(X, Y)
# Выбор npts случайных точек из дискретной области значений моделируемой функции.
npts = 400
px, py = np.random.choice(x, npts), np.random.choice(y, npts)

fig, ax = plt.subplots(nrows=2, ncols=2)
# Построение графика моделируемой функции и случайно выбранной группы точек.
ax[0, 0].contourf(X, Y, T)
ax[0, 0].scatter(px, py, c='k', alpha=0.2, marker='.')
ax[0, 0].set_title('Sample points on f(X, Y)')

# Интерполяция с использованием трех различных методик и построение графика.
for i, method in enumerate(('nearest', 'linear', 'cubic')):
    Ti = griddata((px, py), f(px, py), (X, Y), method=method)
    r, c = (i + 1) // 2, (i + 1) % 2
    ax[r, c].contourf(X, Y, Ti)
    ax[r, c].set_title("method = '{}'".format(method))

fig.tight_layout()
plt.show()
```

8.4 Оптимизация, подгонка данных и численные методы решения уравнений

Пакет `scipy.optimize` предоставляет реализации множества общеизвестных алгоритмов для минимизации многомерных функций (с дополнительными ограничениями или без них), для подгонки данных методом наименьших квадратов и для решения (определения корней) уравнений со многими неизвестными. В этом разделе приведен общий обзор наиболее важных доступных вариантов реализации, но следует всегда помнить о том, что наилучший выбор алгоритма будет зависеть от конкретной анализируемой функции. Для произвольной функции нет никакой уверенности в том, что конкретный метод будет сходящимся к требуемому минимуму (или корню и т. п.) или что сходимость будет обеспечена достаточно быстро. Некоторые алгоритмы подходят в большей степени для конкретных функций, чем для других, поэтому чем больше известно об анализируемой функции, тем лучше. Библиотеку SciPy можно сконфигурировать так, чтобы выводилось предупреждающее сообщение при критических сбоях конкретного алгоритма, и это сообщение, как правило, может помочь при анализе возникшей проблемы.

Более того, возвращаемый результат часто зависит от исходной предпосылки, передаваемой алгоритму, – рассмотрим двумерную функцию как пейзаж с несколькими долинами, разделенными крутыми горными хребтами: начальная предпосылка, размещенная в одной из долин, вероятно, приведет большинство алгоритмов к блужданию по склону и обнаружению минимума в этой долине (даже если это не глобальный минимум) без необходимости взбираться на горные хребты. Точно так же вы можете ожидать (но без каких-либо гарантий), что большинство численных методов решения уравнений возвращают корень, «ближайший» к изначальной предпосылке.

8.4.1 Минимизация

Программы оптимизации библиотеки SciPy минимизируют функцию одной или нескольких переменных $f(x_1, x_2, \ldots, x_n)$. Для поиска максимума определяется минимум функции $-f(x_1, x_2, \ldots, x_n)$.

Некоторые алгоритмы минимизации требуют только саму анализируемую функцию, для других необходима первая производная этой функции по каждой переменной (частные производные) в массиве, известном как матрица Якоби (якобиан):

$$J(f) = (\partial f/\partial x_1, \partial f/\partial x_2, \ldots, \partial f/\partial x_n).$$

Некоторые алгоритмы пытаются оценить матрицу Якоби численными методами, если ее невозможно представить как отдельную функцию.

Более того, для некоторых интеллектуальных алгоритмов оптимизации требуется информация о вторых (частных) производных анализируемой функции в виде симметричной матрицы значений, называемой матрицей Гессе (гессианом):

$$H(f) = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_2 \partial x_1} & \cdots & \frac{\partial^2 f}{\partial x_n \partial x_1} \\ \frac{\partial^2 f}{\partial x_1 \partial x_2} & \frac{\partial^2 f}{\partial x^2} & \cdots & \frac{\partial^2 f}{\partial x_n \partial x_2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_1 \partial x_n} & \frac{\partial^2 f}{\partial x_2 \partial x_n} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{pmatrix}.$$

Матрица Якоби представляет локальный градиент функции нескольких переменных, а матрица Гессе представляет локальный радиус кривизны.

Минимизация без ограничений

Обобщенный алгоритм минимизации для скалярной функции нескольких переменных `scipy.optimize.minimize` принимает два обязательных аргумента:

```
minimize(fun, x0, ...)
```

Первый аргумент – объект функции `fun` для вычисления минимизируемой функции: эта функция должна принимать массив значений `x`, определяющий точки, в которых должны выполняться вычисления $(x_1, x_2, \ldots, x_n)$, с последующими любыми требуемыми аргументами. Второй обязательный аргумент `x0` – это массив значений, представляющих начальные предпосылки, с которых алгоритм минимизации должен начать работу.

В этом разделе демонстрируется использование метода minimize с функцией Химмельблау, простой функцией от двух переменных с некоторыми сложными свойствами, которые делают ее вполне подходящей тестовой функцией для алгоритмов оптимизации. Функция Химмельблау имеет следующий вид:

$$f(x,y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2.$$

Область $-5 \leqslant x \leqslant 5$, $-5 \leqslant y \leqslant 5$ содержит один локальный максимум

$$f(-0.270845, -0.923039) = 181.617$$

(хотя функция «взбирается выше по склону» за пределами этой области). В этой же области существуют четыре минимума:

$$f(3,2) = 0,$$
$$f(-2.805118, 3.131312) = 0,$$
$$f(-3.779310, -3.283186) = 0,$$
$$f(3.584428, -1.848126) = 0$$

и четыре седловые точки (точки перегиба). На рис. 8.21 показан контурный график данной функции.

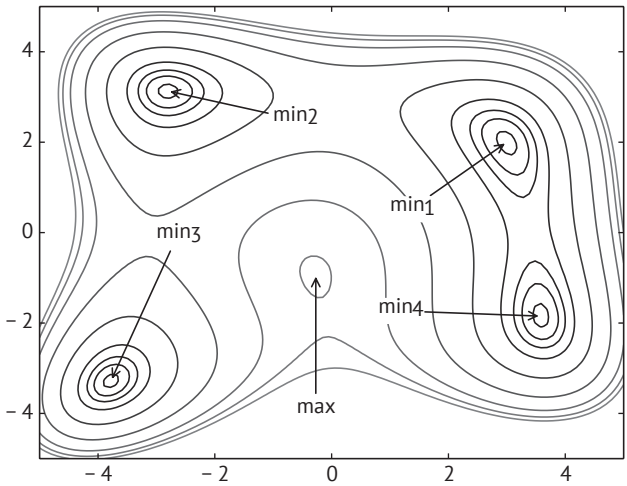


Рис. 8.21. Контурный график (диаграмма линий уровня) функции Химмельблау

Функцию Химмельблау можно определить на Python обычным способом:

```
In [x]: def f(X):
...:        x, y = X
...:        return (x**2 + y - 11)**2 + (x + y**2 - 7)**2
```

где для лучшего понимания массив `X`, содержащий (x_1, x_2), распакован в именованные переменные $x_1 \equiv x$ и $x_2 \equiv y$.

Для поиска минимума вызывается метод `minimize` с некоторыми начальными предпосылками, например $(x,y) = (0,0)$:

```
In [x]: from scipy.optimize import minimize
In [x]: minimize(f, (0, 0))
      jac: array([ -8.77780211e-06,  -3.52519449e-06])
  message: 'Optimization terminated successfully.'
      fun: 6.15694370233122e-13
     njev: 16
 hess_inv: array([[ 0.01575433, -0.00956965],
                  [-0.00956965,  0.03491686]])
   status: 0
     nfev: 64
  success: True
        x: array([ 2.99999989,  1.99999996])
```

Метод `minimize` возвращает объект типа словарь с информацией о минимизации. Важные поля словаря описаны в табл. 8.5: если минимизация завершилась успешно, то минимум передается как x в этом объекте – здесь получена сходимость, близкая к действительному минимуму $f(3,2) = 0$.

Таблица 8.5. Словарь, содержащий информацию о минимизации, возвращаемый методом scipy.optimize.minimize

Ключ	Описание
`success`	Логическое значение, сообщающее, была минимизация успешной или нет
`x`	Если минимизация успешна, то содержит решение: значения ($x_1, x_2, \ldots, x_n$), в которых найден минимум функции. Если алгоритм минимизации завершился неудачно, то x содержит точку аварийного останова
`fun`	Если минимизация успешна, то содержит значение функции в точке минимума, определенной как x
`message`	Строка, описывающая результат минимизации
`jac`	Значение матрицы Якоби: если минимизация успешна, то значения в этом массиве должны быть близкими к нулю
`hess, hess_inv`	Матрица Гессе и обратная ей матрица (если использовалась)
`nfev, njev, nhev`	Количество операций вычисления анализируемой функции, ее якобиана и гессиана

Таблица 8.6. Некоторые из методов оптимизации, используемые методом scipy.optimize.minimize

Значение аргумента method	Описание
`BFGS`	Алгоритм BFGS (Broyden-Fletcher-Goldfarb-Shanno), принятый по умолчанию для минимизации без ограничений или граничных условий
`Nelder-Mead`	Алгоритм Нелдера–Мида, также известный как симплекс-метод (спуска) или метод деформируемого многогранника (амебы). Не требует производных
`CG`	Метод сопряженных градиентов
`Powell`	Метод Пауэлла (для этого алгоритма не требуются производные)
`dogleg`	Ломано-линейный алгоритм в доверительной области (неограниченная минимизация). Требуются матрицы Якоби и Гессе (которые обязательно должны быть положительно определенными)
`TNC`	Усеченный алгоритм Ньютона для минимизации с граничными условиями

Окончание табл. 8.6

Значение аргумента method	Описание
`l-bfgs-b`	Минимизация с ограничениями и граничными условиями с использованием алгоритма L-BFGS-B
`slsqp`	Метод минимизации «последовательное программирование методом наименьших квадратов» (sequential least-squares programming) с граничными условиями и ограничениями по равенству и неравенству
`cobyla`	Метод «оптимизации с ограничениями с использованием линейной аппроксимации» (constrained optimization by linear approximation) для минимизации с ограничениями

Алгоритм, используемый методом `minimize`, определяется с помощью передаваемой в аргументе `method` строки, допустимые варианты которой приведены в табл. 8.6. Алгоритм, применяемый по умолчанию, BFGS – эффективный квазиньютоновский метод, который может аппроксимировать матрицу Якоби, если она не предоставлена и не используется матрица Гессе. Но этот алгоритм испытывает затруднения при поиске минимума функции Химмельблау:

```
In [x]: mf = lambda X: -f(X) # Для поиска максимума минимизируется функция -f(x, y).
In [x]: minimize(mf, (0.1, -0.2))
Out[x]:
      fun: -1.2100579056485772e+35
 hess_inv: array([[  0.254751 , -0.43222419],
                  [-0.43222419,  0.83976276]])
      jac: array([0., 0.])
  message: 'Optimization terminated successfully.'
     nfev: 68
      nit: 2
     njev: 17
   status: 0
  success: True
        x: array([ 3.45579856e+08, -5.71590777e+08])
```

Начиная с (0.1, –0.2) алгоритм BFGS «сбивается с курса» на одном из крутых склонов функции Химмельблау, и сходимость не обеспечивается. К сожалению, в этом случае алгоритм не может определить критическую ошибку и возвращает значение True в флаге success (иногда, но не во всех случаях может выводиться сообщение об ошибке: `'Desired error not necessarily achieved due to precision loss.'` [`'Требуемое значение погрешности не гарантируется из-за потери точности.'`] – это зависит от настройки уровня погрешности в конкретной системе). В действительности, чтобы обеспечить успешное завершение процесса минимизации, необходимо начать достаточно близко к точке максимума:

```
In [x]: minimize(mf, (-0.2,-1))
Out[x]:
      jac: array([ 3.81469727e-06, 1.90734863e-06])
  message: 'Optimization terminated successfully.'
      fun: -181.61652152258262
     njev: 8
```

```
 hess_inv: array([[ 0.0232834 , -0.00626945],
                  [-0.00626945,  0.06137267]])
   status: 0
     nfev: 32
  success: True
        x: array([-0.27084453, -0.92303852])
```

Разумеется, это не очень помогает, если заранее неизвестно, где находится максимум. Попробуем применить другой алгоритм минимизации и начнем с произвольно выбранной предпосылки (0,0):

```
In [x]: minimize(mf, (0, 0), method='Nelder-Mead')
Out[x]:
  status: 0
    nfev: 115
 success: True
 message: 'Optimization terminated successfully.'
     fun: -181.61652150549165
     nit: 59
       x: array([-0.27086815, -0.92300745])
```

Алгоритм Нелдера–Мида – это симплекс-метод, который не требует вычисления или оценки производных функции, так что он не пытается взбираться на крутые склоны функции. Тем не менее этот алгоритм выполняет вычисления по 115 функциям для обеспечения сходимости к локальному максимуму.

В качестве последнего примера рассмотрим метод `dogleg`, который требует передачи в метод `minimize` функций, вычисляющих матрицы Якоби и Гессе. Для функции Химмельблау необходимые производные имеют простые аналитические формы:

$$\partial f/\partial x = 4x(x^2 + y - 11) + 2(x + y^2 - 7),$$
$$\partial f/\partial y = 2(x^2 + y - 11) + 4y(x + y^2 - 7),$$
$$\partial^2 f/\partial x^2 = 12x^2 + 4y - 42,$$
$$\partial^2 f/\partial y^2 = 12y^2 + 4x - 26,$$
$$\partial^2 f/\partial y \partial x = \partial^2 f/\partial x \partial y = 4x + 4y.$$

Для вычисления матриц Якоби и Гессе можно написать следующий код:

```
In [x]: def df(X):
...:        x, y = X
...:        f1, f2 = x**2 + y - 11, x + y**2 - 7
...:        dfdx = 4*x*f1 + 2*f2
...:        dfdy = 2*f1 + 4*y*f2
...:        return np.array([dfdx, dfdy])
...:
In [x]: def ddf(X):
...:        x, y = X
...:        d2fdx2 = 12*x**2 + 4*y - 42
...:        d2fdy2 = 12*y**2 + 4*x - 26
...:        d2fdxdy = 4*(x + y)
...:        return np.array([[d2fdx2, d2fdxdy], [d2fdxdy, d2fdy2]])
...:
In [x]: mdf = lambda X: -df(X)                                    ❶
In [x]: mddf = lambda X: -ddf(X)
```

❶ Обратите внимание: как и для самой функции, при поиске максимума необходимо использовать отрицательные матрицы Якоби и Гессе: они определены как лямбда-функции `mdf` и `mddf`.

```
In [x]: minimize(mf, (0, 0), jac=mdf, hess=mddf, method='dogleg')
Out[x]:
     jac: array([ -1.26922473e-10,   1.23685240e-09])
 message: 'Optimization terminated successfully.'
     fun: -181.6165215225827
    hess: array([[ 44.81187272,   4.77553259],
                 [  4.77553259,  16.85937624]])
     nit: 4
    njev: 5
       x: array([-0.27084459, -0.92303856])
  status: 0
    nfev: 5
 success: True
    nhev: 4
```

Алгоритм успешно сходится в точке локального максимума после пяти операций вычисления функции, пяти операций вычисления матрицы Якоби и четырех операций вычисления матрицы Гессе.

◊ Оптимизация с ограничениями

Иногда необходимо найти максимум или минимум объекта функции с одним или несколькими ограничениями. Чтобы воспользоваться описанным выше методом в качестве примера, можно попытаться найти один минимум функции $f(x,y)$, который удовлетворяет условию $x > 0$, $y > 0$, или значение минимума функции на прямой $x = y$.

Алгоритмы `l-bfgs-b`, `tnc` и `slsqp` поддерживают аргумент `bounds` для передачи в метод `minimize`. Значение аргумента `bounds` – это последовательность кортежей, каждый из которых содержит пары `(min, max)` для каждой переменной функции, определяющие границы минимизации для соответствующей переменной. Если в каком-либо направлении не существует ограничений, то используется значение `None`.

Например, если выполняется попытка найти минимум функции $f(x,y)$, начиная с $(-1/2, -1/2)$ без определения каких-либо границ, то метод `slsqp` обеспечивает сходимость (почти точную) к минимуму в точке (–2.805118, 3.131312):

```
In [x]: minimize(f, (-0.5,-0.5), method='slsqp')
Out[x]:
     jac: array([-0.00721077,   0.00037714,   0.        ])
 message: 'Optimization terminated successfully.'
     fun: 4.0198760213901536e-07
     nit: 10
    njev: 10
       x: array([-2.80522924,   3.131319  ])
  status: 0
    nfev: 46
 success: True
```

Чтобы оставаться в квадранте $x < 0$, $y < 0$, необходимо установить значения `bounds` без минимальных границ по x и y и максимальные границы $x = 0$ и $y = 0$:

```
In [x]: xbounds = (None, 0)
In [x]: ybounds = (None, 0)
In [x]: bounds = (xbounds, ybounds)
In [x]: minimize(f, (-0.5, -0.5), bounds=bounds, method='slsqp')
Out[x]:
     jac: array([-0.00283595, -0.00034243,  0.       ])
 message: 'Optimization terminated successfully.'
     fun: 4.115667606325133e-08
     nit: 11
    njev: 11
       x: array([-3.77933774, -3.28319868])
  status: 0
    nfev: 50
 success: True
```

Предположим, что необходимо найти точки экстремума функции Химмельблау, которые также соответствуют условию $x = y$ (т. е. лежат на диагонали графика, изображенного на рис. 8.21). Из методов минимизации, перечисленных в табл. 8.6, два – `cobyla` и `slsqp` – позволяют применять ограничения, поэтому необходимо воспользоваться одним из них.

Ограничения определяются как аргумент `constrains` для метода `minimize` в виде последовательности словарей, определяющих строковые ключи: `'type'` – тип ограничения и `'fun'` – вызываемый объект, реализующий это ограничение. Значением `'type'` может быть `'eq'` или `'ineq'` для ограничения, основанного на равенстве (например, $x = y$) или на неравенстве (например, $x > 2y - 1$). Обратите особое внимание: метод `cobyla` не поддерживает ограничения, основанные на равенстве.

Функция ограничения, основанного на равенстве, должна возвращать ноль, если ограничение соблюдено, а функция ограничения, основанного на неравенстве, должна возвращать неотрицательное значение, если ограничение соблюдено.

Для поиска минимума функции $f(x,y)$ с ограничением $x = y$ можно воспользоваться методом `slsqp` с функцией ограничения, основанного на равенстве, возвращающей $x - y$:

```
In [x]: con = {'type': 'eq', 'fun': lambda X: X[0] - X[1]}
In [x]: minimize(f, (0, 0), constraints=con, method='slsqp')
     jac: array([-16.33084416,  16.33130538,  0.       ])
 message: 'Optimization terminated successfully.'
     fun: 8.0000000007160867
     nit: 7
    njev: 7
       x: array([ 2.54138438,  2.54138438])
  status: 0
    nfev: 32
 success: True
```

Метод сходится к одному из минимумов (существует и другой минимум, который можно найти, начав, например, с (−2, −2). А как насчет максимума?

```
In [x]: minimize(mf, (0, 0), constraints=con, method='slsqp')
Out[x]:
     jac: array([ 0.,  0.,  0.])
 message: 'Singular matrix C in LSQ subproblem'
     fun: -3.1826053300603689e+68
     nit: 4
    njev: 4
       x: array([ -1.12315113e+17,  -1.12315113e+17])
  status: 6
    nfev: 16
 success: False
```

В этом случае поиск заканчивается неудачно – алгоритм пытается «взобраться на крутой склон долины». Более подходящим является выбор алгоритма `cobyla`, но этот алгоритм не поддерживает ограничения, основанные на равенстве, поэтому придется сформировать требуемое ограничение с помощью пары неравенств: $x = y$, если не выполняются оба неравенства $x > y$ и $x < y$:

```
In [x]: con1 = {'type': 'ineq', 'fun': lambda X: X[0] - X[1]}
In [x]: con2 = {'type': 'ineq', 'fun': lambda X: X[1] - X[0]}
In [x]: minimize(mf, (0, 0), constraints=(con1, con2), method='cobyla')
Out[x]:
  status: 1
    nfev: 34
 success: True
 message: 'Optimization terminated successfully.'
     fun: -179.12499987327624
   maxcv: 0.0
       x: array([-0.49994148, -0.49994148])
```

Здесь функция ограничения, определенная в `con1`, возвращает неотрицательное значение, если $x > y$, а функция, определенная в `con2`, возвращает неотрицательное значение, если $x < y$. Единственный вариант выполнения этих обоих ограничений – равенство $x = y$.

Минимизация функции одной переменной

Если минимизируемая функция является одномерной (т. е. зависит только от одной скалярной переменной), то для нее предлагается более быстрый алгоритм `scipy.optimize.minimize_scalar`. Чтобы просто получить минимум, эту функцию можно вызвать с аргументом `method='brent'`, который определяет использование реализации метода Брента для нахождения минимума.

В идеальном случае сначала нужно установить интервал поиска минимума, предоставив для x значения (a, b, c) такие, что $f(a) > f(b)$ и $f(c) > f(b)$. Это можно сделать с помощью аргумента `bracket`, в котором передается кортеж `(a, b, c)`. Если это невозможно или слишком сложно, то передается интервал из двух значений x, в котором начинается поиск ограниченной области минимума (в направлении «спуска», т. е. убывания функции). Если значение аргумента `bracket` не задано, то поиск ограниченной области минимума начинается с интервала (0,1).

На рис. 8.22 приведен пример графика многочлена с двумя минимумами и одним максимумом.

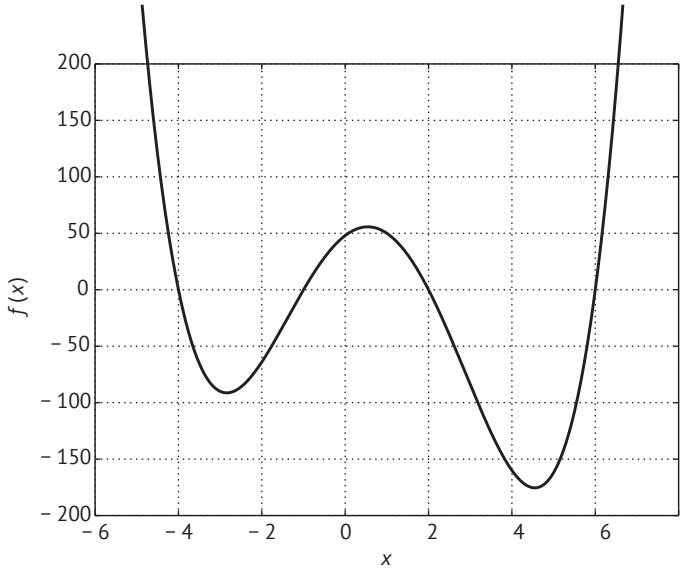


Рис. 8.22. График многочлена $f(x) = x^4 - 3x^3 - 24x^2 + 28x + 48$

Без заданного значения аргумента `bracket` метод `minimize_scalar` обеспечивает сходимость к минимуму в точке –2.841 для функции, показанной на рис. 8.22:

```
In [x]: Polynomial = np.polynomial.Polynomial
In [x]: from scipy.optimize import minimize_scalar
In [x]: f = Polynomial( (48., 28., -24., -3., 1.))
In [x]: minimize_scalar(f)
Out[x]:
  fun: -91.32163915433344
 nfev: 11
    x: -2.8410443265958261
  nit: 10
```

Если определить ограниченную область поиска другого минимума, передавая значения `(a, b, c)` = `(3, 4, 6)`, которые, как можно видеть на рис. 8.22, соответствуют условию $f(a) > f(b) < f(c)$, то алгоритм обеспечивает сходимость в точке 4.549:

```
In [x]: minimize_scalar(f, bracket=(3, 4, 6))
Out[x]:
  fun: -175.45563549487974
 nfev: 11
    x: 4.5494683642571934
  nit: 10
```

Наконец, для поиска максимума метод `minimize_scalar` вызывается с передачей в него функции $-f(x)$. На этот раз инициализируется поиск в ограниченной области минимума функции $-f(x)$ с помощью пары значений (–1, 0):

```
In [x]: minimize_scalar(-f, bracket=(-1, 0))
Out[x]:
  fun: -55.734305899213226
 nfev: 9
    x: 0.54157595897344157
  nit: 8
```

Пример П8.23. Простая модель оболочки корпуса дирижабля описывает ее как тело вращения, полученное из пары четвертей эллипсов, соединенных по их (равным) малым полуосям. Малая полуось кормового эллипса принимается более длинной, чем та, которая представляет нос дирижабля с коэффициентом $\alpha = 6$. Уравнения, описывающие поперечное сечение (в вертикальной плоскости) оболочки корпуса дирижабля, можно записать в следующем виде:

$$y = \begin{cases} \dfrac{b}{a}\sqrt{x(2a-x)} & (x \le a), \\ \dfrac{b}{a}\sqrt{a^2 - \dfrac{(x-a)^2}{a^2}} & (a < x \le \alpha(a+1)). \end{cases}$$

Лобовое сопротивление для такого корпуса определяется по формуле:

$$D = 1/2\rho_{air} v^2 V^{2/3} C_{DV},$$

где ρ_{air} – плотность воздуха, v – скорость дирижабля, V – объем оболочки корпуса, C_{DV} – коэффициент лобового сопротивления, приблизительно определяемый по следующей эмпирической формуле[116]:

$$C_{DV} = \mathrm{Re}^{-1/6}[0.172(l/d)^{1/3} + 0.252(d/l)^{1.2} + 1.032(d/l)^{2.7}].$$

Здесь $\mathrm{Re} = \rho_{air} vl/\mu$ – число Рейнольдса, μ – коэффициент динамической вязкости воздуха, l – длина корпуса дирижабля, d – его максимальный диаметр ($= 2b$).

Предположим, что необходимо минимизировать лобовое сопротивление с учетом изменения параметров a и b, но при постоянном общем объеме корпуса дирижабля $V = 2/3\pi ab^2(1 + \alpha)$. Программа в листинге 8.19 решает эту задачу, используя алгоритм `slsqp` для дирижабля класса «Гинденбург» объемом 200 000 м^3.

Листинг 8.19. Минимизация лобового сопротивления для корпуса дирижабля

```
# eg8-airship.py
import numpy as np
from scipy.optimize import minimize

# Плотность воздуха (кг/м3) и коэффициент динамической вязкости воздуха (Па·с)
# на высоте полета.
rho, mu = 1.1, 1.5e-5
# Воздушная скорость (м/с) на высоте полета.
v = 30
```

[116] *F. Hoerner*. Fluid Dynamic Drag, Hoerner Fluid Dynamics (1965).

```
def CDV(L, d):
    """ Calculate the drag coefficient. """
    """ Вычисление коэффициента лобового сопротивления воздуха. """
    Re = rho * v * L / mu        # Число Рейнольдса.
    r = L / d                    # Аэродинамическое качество.
    return (0.172 * r**(1/3) + 0.252 / r**1.2 + 1.032 / r**2.7) / Re**(1/6)

def D(X):
    """ Return the total drag on the airship envelope. """
    """ Возвращает общий коэффициент лобового сопротивления для корпуса дирижабля. """
    a, b = X
    L = a * (1+alpha)
    return 0.5 * rho * v**2 * V(X)**(2/3) * CDV(L, 2*b)

# Постоянный общий объем корпуса дирижабля (м3).
V0 = 2.e5
# Параметр, описывающий конусообразность хвостовой части корпуса дирижабля.
alpha = 6

def V(X):
    """ Возвращает объем корпуса дирижабля. """
    a, b = X
    return 2 * np.pi * a * b**2 * (1+alpha) / 3

# Минимизация лобового сопротивления с заданным ограничением: объем должен быть равен V0.
a0, b0 = 70, 45      # Начальные предпосылки для a, b.
con = {'type': 'eq', 'fun': lambda X: V(X)-V0}
res = minimize(D, (a0, b0), method='slsqp', constraints=con)
if res['success']:
    a, b = res['x']
    L, d = a * (1+alpha), 2*b     # Длина, наибольший диаметр.
    print('Optimum parameters: a = {:g} m, b = {:g} m'.format(a, b))
    print('V = {:g} m3'.format(V(res['x'])))
    print('Drag, D = {:g} N'.format(res['fun']))
    print('Total length, L = {:g} m'.format(L))
    print('Greatest diameter, d = {:g} m'.format(d))
    print('Fineness ratio, L/d = {:g}'.format(L/d))
else:
    # Сходимость не обеспечена: вывод итогового словаря.
    print('Failed to minimize D!', res, sep='\n')
```

Это немного надуманный пример, поскольку для постоянного параметра α условие неизменяемости объема V означает, что значения a и b не являются независимыми, но решение найдено достаточно быстро:

```
Optimum parameters: a = 32.9301 m, b = 20.3536 m
V = 200000 m3
Drag, D = 20837.6 N
Total length, L = 230.51 m
Greatest diameter, d = 40.7071 m
Fineness ratio, L/d = 5.66266
```

Настоящие размеры реально существовавшего дирижабля «Гинденбург»: l = 245 м, d = 41 м с отношением l/d = 5.98, так что мы получили неплохой приближенный результат.

8.4.2 Нелинейная подгонка методом наименьших квадратов

В библиотеке SciPy обобщенной программой нелинейной подгонки методом наименьших квадратов является `scipy.optimize.leastsq`, основная сигнатура вызова которой:

```
scipy.optimize.leastsq(func, x0, args=())
```

Эта программа пытается подогнать последовательность точек данных `y` к моделируемой функции `f`, которая зависит от одного или нескольких параметров подгонки. В метод `leastsq` передается соответствующий объект функции `func`, которая возвращает разность между `y` и `f` (невязки). Для метода `leastsq` также требуется начальная предпосылка для подгоняемых параметров `x0`. Если для функции `func` необходимы какие-либо другие аргументы (обычно массив данных `y` и одна или несколько независимых переменных), то они передаются в последовательности `args`. Например, рассмотрим подгонку искусственно зашумленной затухающей функции косинуса $f(t) = Ae^{-t/T} \cos 2\pi\nu t$ (см. рис. 8.23).

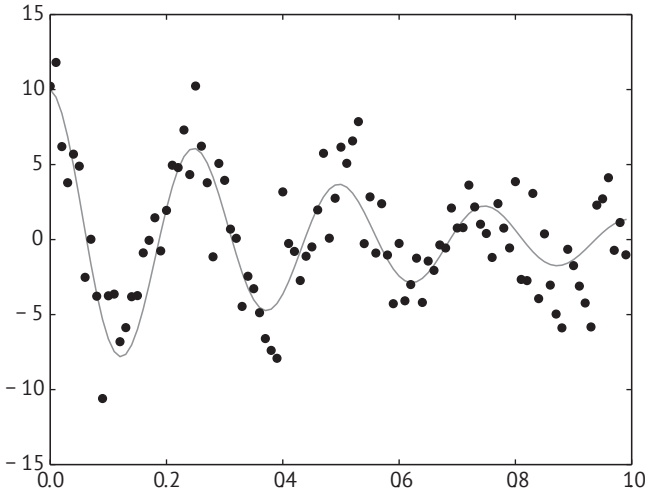


Рис. 8.23. Искусственно зашумленная затухающая функция косинуса

```
In [x]: import numpy as np
In [x]: import matplotlib.pyplot as plt

In [x]: A, freq, tau = 10, 4, 0.5
In [x]: def f(t, A, freq, tau):
...:        return A * np.exp(-t/tau) * np.cos(2*np.pi * freq * t)
...:
In [x]: tmax, dt = 1, 0.01
In [x]: t = np.arange(0, tmax, dt)
In [x]: yexact = f(t, A, freq, tau)
In [x]: y = yexact + np.random.randn(len(yexact))*2
In [x]: plt.plot(t, yexact)
In [x]: plt.plot(t, y)
In [x]: plt.show()
```

Для подгонки этого зашумленного набора данных `y` к параметрам `A`, `freq` и `tau` (будем считать, что нам они неизвестны) сначала определим необходимую функцию `residuals`:

```
In [x]: def residuals(p, y, t):
...:        A, freq, tau = p
...:        return y - f(t, A, freq, tau)
```

Первый аргумент – последовательность параметров `p`, которые для лучшего понимания распаковываются в именованные переменные. Необходимые дополнительные аргументы: набор точек данных `y` и независимая переменная `t`. Теперь определим для параметров некоторые начальные предпосылки, которые не должны быть слишком необдуманными, и вызовем метод `leastsq`:

```
In [x]: from scipy.optimize import leastsq
In [x]: p0 = 5, 5, 1
In [x]: plsq = leastsq(residuals, p0, args=(y, t))
In [x]: plsq[0]
Out[x]: [ 9.33962672   4.04958427   0.48637434]
```

Как и другие программы оптимизации из библиотеки SciPy, метод `leastsq` можно настроить так, чтобы он возвращал больше информации о выполненной работе, но здесь выводятся только решения (параметры наилучшей подгонки), которые всегда являются первым элементом в кортеже `plsq`.

Действительные значения `A, freq, tau = 10, 4, 0.5`, поэтому с учетом шума в данных результат вполне приемлемый. Соответствующее графическое отображение:

```
In [x]: plt.plot(t, y, 'o', c='k', label='Data')
In [x]: plt.plot(t, yexact, c='gray', label='Exact')
In [x]: pfit = plsq[0]
In [x]: plt.plot(t, f(t, *pfit), c='k', label='Fit')
In [x]: plt.legend()
In [x]: plt.show()
```

Результат подгонки показан на рис. 8.24.

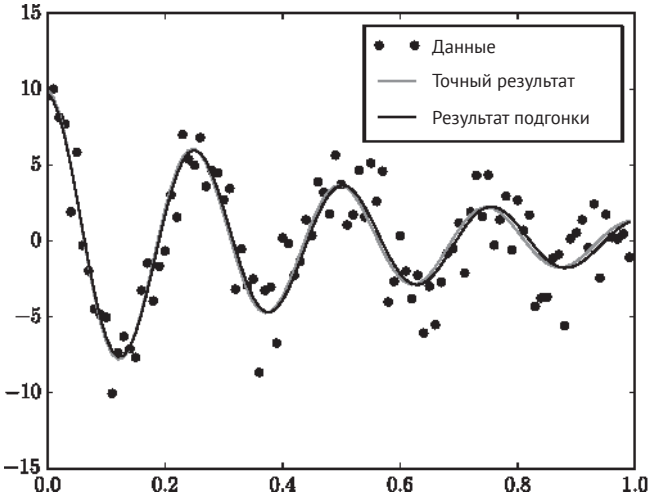


Рис. 8.24. Нелинейная подгонка методом наименьших квадратов для зашумленной затухающей функции косинуса

В метод `leastsq` также можно передать матрицу Якоби, если она известна, как показано в примере П8.24.

Пример П8.24. Здесь рассматривается зашумленная последовательность точек данных, которые необходимо подогнать к эллипсу. Уравнение эллипса можно записать как нелинейную функцию от угла θ ($0 \leqslant \theta \leqslant 2\pi$), который зависит от параметров a (главная полуось эллипса) и e (эксцентриситет эллипса):

$$r(\theta; a, e) = a(1 - e^2) / (1 - e \cos \theta).$$

Для подгонки последовательности точек данных (θ, r) к этой функции сначала напишем код функции Python, принимающей два аргумента: независимая переменная `theta` и кортеж параметров `p = (a, e)`. Необходимая для минимизации функция определяет разность между этой моделируемой функцией и данными `r`, определенную как метод `residuals`:

```
plsq = leastsq(residuals, p0, args=(r, theta))
```

Но, если это вообще возможно, лучше также передать матрицу Якоби (первая производная функции подгонки с учетом подгоняемых параметров). Эти формулы без затруднений вычисляются и реализуются:

$$\partial f/\partial a = (1 - e^2) / (1 - e \cos \theta),$$
$$\partial f/\partial e = a[\cos \theta(1 + e^2) - 2e] / (1 - e \cos \theta)^2.$$

Но функция, необходимая для минимизации, представляет собой функцию невязок $r - f$, поэтому требуются отрицательные значения этих производных. В листинге 8.20 приведен работающий код, а на рис. 8.25 – результат подгонки.

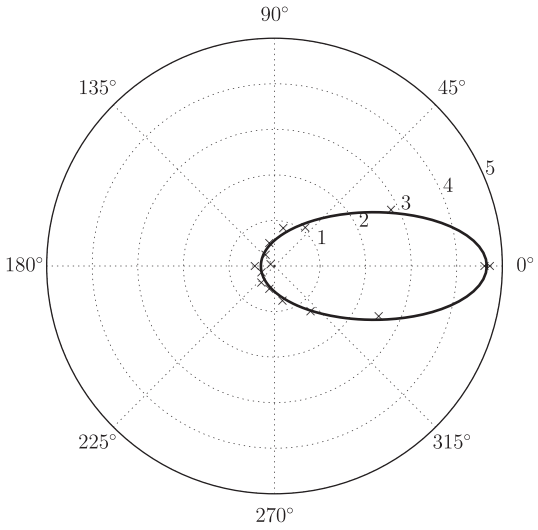


Рис. 8.25. Нелинейная подгонка методом наименьших квадратов набора данных к уравнению эллипса в полярных координатах

Листинг 8.20. Нелинейная подгонка методом наименьших квадратов к эллипсу

```
# eg8-leastsq.py
import numpy as np
from scipy import optimize
import matplotlib.pyplot as plt

def f(theta, p):
    a, e = p
    return a * (1 - e**2)/(1 - e*np.cos(theta))

# Данные для подгонки.
theta = np.array([0.0000, 0.4488, 0.8976, 1.3464, 1.7952, 2.2440, 2.6928,
                  3.1416, 3.5904, 4.0392, 4.4880, 4.9368, 5.3856, 5.8344, 6.2832])
r = np.array([4.6073, 2.8383, 1.0795, 0.8545, 0.5177, 0.3130, 0.0945, 0.4303,
              0.3165, 0.4654, 0.5159, 0.7807, 1.2683, 2.5384, 4.7271])

def residuals(p, r, theta):
    """ Return the observed - calculated residuals using f(theta , p). """
    """ Возвращает невязки значений наблюдаемое-вычисляемое с использованием f(theta, p).
"""
    return r - f(theta, p)

def jac(p, r, theta):
    """ Calculate and return the Jacobian of residuals. """
    """ Вычисление и возвращение матрицы Якоби невязок. """
    a, e = p
    da = (1 - e**2)/(1 - e*np.cos(theta))
    de = a * (np.cos(theta) * (1 + e**2) - 2*e) / (1 - e*np.cos(theta))**2
    return -da, -de

# Начальные предпосылки для a, e.
p0 = (1, 0.5)
plsq = optimize.leastsq(residuals, p0, Dfun=jac, args=(r, theta), col_deriv=True)
print(plsq)
```

```
plt.polar(theta, r, 'x')
theta_grid = np.linspace(0, 2*np.pi, 200)
plt.polar(theta_grid, f(theta_grid, plsq[0]), lw=2)
plt.show()
```

В библиотеку SciPy также включен метод подбора (аппроксимации) кривой `scipy.optimize.curve_fit`, которая может напрямую подгонять данные к функции (без дополнительной функции, вычисляющей невязки) и поддерживает подгонку взвешенным (с весовыми коэффициентами) методом наименьших квадратов. Сигнатура вызова этого метода

```
curve_fit(f, xdata, ydata, p0, sigma, absolute_sigma)
```

где `f` – функция, к которой подгоняются данные (`xdata, ydata`). Аргумент `p0` – это начальная предпосылка для параметров. Если определен аргумент `sigma`, то в нем передаются весовые коэффициенты для значений `ydata`. Если для аргумента `absolute_sigma` задано значение `True`, то значения `sigma` интерпретируются как стандартное (среднеквадратическое) отклонение ошибки (т. е. абсолютные весовые коэффициенты). По умолчанию `absolute_sigma=False`, и значения `sigma` интерпретируются как относительные весовые коэффициенты.

Метод `curve_fit` возвращает `popt` – значения наилучшей подгонки параметров и `pcov` – матрицу ковариации параметров.

Пример П8.25. Для демонстрации практического использования метода `curve_fit` для подгонки методом наименьших квадратов с весовыми коэффициентами и без них программа в листинге 8.21 подгоняет функцию кривой распределения Лоренца, центрированную в точке x_0 с половинной шириной на уровне половинной амплитуды (HWHM) γ и амплитудой A:

$$f(x) = A\gamma^2 / (\gamma^2 + (x - x_0)^2)$$

по некоторым искусственно искаженным (зашумленным) данным. Параметры подгонки A, γ и x_0. Данные, близкие к центральной линии, имитируются как более искаженные, чем остальные.

Листинг 8.21. Подгонка методом наименьших квадратов с весовыми коэффициентами и без них с использованием метода curve_fit

```
# eg8-curve -fit.py
import numpy as np
from scipy.optimize import curve_fit
import matplotlib.pyplot as plt

x0, A, gamma = 12, 3, 5

n = 200
x = np.linspace(1, 20, n)
yexact = A * gamma**2 / (gamma**2 + (x-x0)**2)
```

```
# Добавление некоторого шума с использованием sigma с коэффициентом 0.5 вне особенно
# зашумленной области в окрестности x0, где значение sigma равно 3.
sigma = np.ones(n)*0.5
sigma[np.abs(x-x0+1)<1] = 3
noise = np.random.randn(n) * sigma
y = yexact + noise

def f(x, x0, A, gamma):
    """ The Lorentzian entered at x0 with amplitude A and HWHM gamma. """
    """ Функция кривой распределения Лоренца в точке x0 с амплитудой A и HWHM gamma. """
    return A *gamma**2 / (gamma**2 + (x-x0)**2)

def rms(y, yfit):
    return np.sqrt(np.sum((y-yfit)**2))

# Подгонка без весовых коэффициентов.
p0 = 10, 4, 2
popt, pcov = curve_fit(f, x, y, p0)
yfit = f(x, *popt)
print('Unweighted fit parameters:', popt)
print('Covariance matrix:'); print(pcov)
print('rms error in fit:', rms(yexact , yfit))
print()

# Подгонка с весовыми коэффициентами.
popt2, pcov2 = curve_fit(f, x, y, p0, sigma=sigma, absolute_sigma=True)
yfit2 = f(x, *popt2)
print('Weighted fit parameters:', popt2)
print('Covariance matrix:'); print(pcov2)
print('rms error in fit:', rms(yexact , yfit2))

plt.plot(x, yexact, label='Exact')
plt.errorbar(x, y, yerr=noise, elinewidth=0.5, c='0.5', marker='+', lw=0, label='Noisy
data')
plt.plot(x, yfit, label='Unweighted fit')
plt.plot(x, yfit2, label='Weighted fit')
plt.ylim(-1, 4)
plt.legend(loc='lower center')
plt.show()
```

Как можно видеть на рис. 8.26, подгонка без весовых коэффициентов искажается в зашумленной области. При подгонке с весовыми коэффициентами данным в этой области присваивается меньший вес, поэтому параметры получаются более близкими к истинным значениям, следовательно, улучшается качество подгонки в целом. Ниже приведен вывод результата:

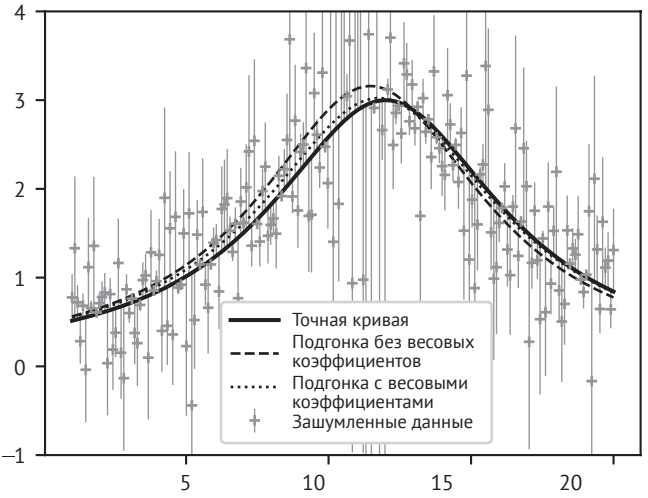


Рис. 8.26. Пример подгонки методом наименьших квадратов с использованием метода scipy.optimize.curve_fit

```
Unweighted fit parameters: [ 11.61282984   3.64158981   3.93175714]
Covariance matrix:
[[ 0.0686249  -0.00063262  0.00231442]
 [-0.00063262  0.06031262 -0.07116127]
 [ 0.00231442 -0.07116127  0.16527925]]
rms error in fit: 4.10434012348

Weighted fit parameters: [ 11.90782988   3.0154818   4.7861561 ]
Covariance matrix:
[[ 0.01893474 -0.00333361  0.00639714]
 [-0.00333361  0.01233797 -0.02183039]
 [ 0.00639714 -0.02183039  0.06062533]]
rms error in fit: 0.694013741786
```

8.4.3 Численные методы решения уравнений

Модуль `scipy.optimize` предоставляет несколько методов для вычисления корней одномерных и многомерных функций. Здесь рассматриваются только алгоритмы, относящиеся к функциям одной переменной: `brentq`, `brenth`, `ridder` и `bisect`. Для каждого из этих методов требуется непрерывная функция $f(x)$ и пара чисел, определяющая ограничивающий интервал для поиска корней, т. е. значения a и b, такие, что корень находится в интервале $[a, b]$ и $\mathrm{sgn}[f(a)] = -\mathrm{sgn}[f(b)]$. Подробные описания алгоритмов, используемых в этих методах поиска корней уравнений, можно найти в обычных книгах по численным методам анализа[117].

В общем случае основным методом поиска корней удобной для анализа (аналитической) функции является `scipy.optimize.brentq`, реализующий версию метода Брента с обратной квадратичной экстраполяцией (`scipy.optimize.`

[117] Например: *Press et al.* Numerical Recipes. The Art of Scientific Computing, 3rd edn., Cambridge University Press, Cambridge (2007).

`brenth` – похожий алгоритм, но с гиперболической экстраполяцией). В качестве примера рассмотрим следующую функцию в интервале $-1 \leqslant x \leqslant 1$:

$$f(x) = 1/5 + x\cos(3/x).$$

График этой функции (см. рис. 8.27) позволяет предположить, что корень находится между –0.7 и –0.5.

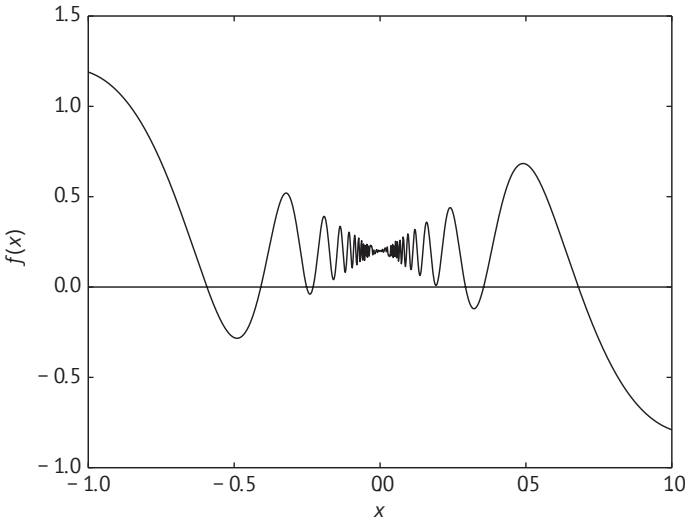


Рис. 8.27. Функция $f(x) = 1/5 + x\cos(3/x)$ и ее корни

```
In [x]: f = lambda x: 0.2 + x*np.cos(3/x)
In [x]: x = np.linspace(-1, 1, 1000)
In [x]: plt.plot(x, f(x))
In [x]: plt.axhline(0, color='k')
In [x]: plt.show()

In [x]: from scipy.optimize import brentq
In [x]: brentq(f, -0.7, -0.5)
Out[x]: -0.5933306271014237
```

Алгоритм поиска корней, известный как метод Риддерса, реализован в функции `scipy.optimize.ridder`. Более медленный, но весьма надежный (для непрерывных функций) метод бисекции представлен как `scipy.optimize.bisect`.

Поиск корней (решение уравнений) с использованием алгоритма Ньютона–Рафсона может быть очень быстрым (квадратическим) для многих непрерывных функций, для которых можно вычислить первую производную $f'(x)$. Если можно написать исходный код для аналитического выражения первой производной $f'(x)$, то это закодированное выражение передается в метод `scipy.optimize.newton` как аргумент `fprime` вместе с начальной точкой `x0`, которая (в общем случае) должна располагаться как можно ближе к вычисляемому корню. Нет необходимости определять ограничивающий интервал для корня. Если первую производную $f'(x)$ передать невозможно, то метод `newton` исполь-

зует метод секущих. Если выбрано удачное положение, в котором можно определить вторую производную $f''(x)$ и передать ее в аргументе `fprime2` так же, как и первую производную, то вместо алгоритма Ньютона–Рафсона используется метод Галлея (который сходится даже быстрее, чем основной алгоритм Ньютона–Рафсона).

Следует отметить, что условием остановки при выполнении итерационного алгоритма, используемого методом `newton`, является размер шага, поэтому нет полной уверенности в том, что будет обеспечена сходимость к требуемому корню: результат должен быть проверен вычислением функции при возвращенном значении для проверки его близости к нулю.

Пример П8.26. В экологии уравнение Эйлера–Лотки описывает рост населения как функцию $P(x)$ – долю людей, доживших до возраста x, и $m(x)$ – средний коэффициент рождаемости для женщин, живущих в рассматриваемый интервал времени:

$$\sum_{x=\alpha}^{\beta} P(x)m(x)e^{-rx} = 1,$$

где α и β – границы репродуктивного возраста, определяющие дискретный коэффициент роста $\lambda = e^r$. Значение $r = \ln \lambda$ известно как истинный коэффициент естественного прироста населения Лотки.

В статье Лесли и Рэнсона[118] описаны измерения $P(x)$ и $m(x)$ для мышей-полевок (Microtus agrestis) с интервалами времени, равными восьми неделям. Данные измерений приведены в табл. 8.7.

Таблица 8.7. Данные о популяции мышей-полевок, собранные Лесли и Рэнсоном

x, недели	*m*(*x*)	*P*(*x*)
8	0.6504	0.83349
16	2.3939	0.73132
24	2.9727	0.58809
32	2.4662	0.43343
40	1.7043	0.29277
48	1.0815	0.18126
56	0.6683	0.10285
64	0.4286	0.05348
72	0.3000	0.02549

Сумма $R_0 = \sum_{x=\alpha}^{\beta} P(x)m(x)$ позволяет получить отношение между общим количеством женщин, рождающихся в последующих поколениях: население

[118] *P. H. Leslie and R. M. Ranson*. The mortality, fertility and rate of natural increase of the vole (Microtus agrestis) as observed in the laboratory, J. Anim. Ecol. 9, 27 (1940).

растет, если $R_0 > 1$, а r определяет скорость этого роста. Чтобы найти r, Лесли и Рэнсон использовали приближенный численный метод. Код в листинге 8.22 определяет r с помощью прямого поиска действительного корня уравнения Лотки–Эйлера (можно показать, что здесь существует только один корень).

Листинг 8.22. Решение уравнения Эйлера–Лотки

```
# eg8-euler-lotka.py
import numpy as np
from scipy.optimize import brentq

# Данные из таблицы 6, приведенной в статье:
# P. H. Leslie and R. M. Ranson, J. Anim. Ecol. 9, 27 (1940).
x = np.linspace(8, 72, 9)
m = np.array( [0.6504, 2.3939, 2.9727, 2.4662, 1.7043,
               1.0815, 0.6683, 0.4286, 0.3000] )
P = np.array( [0.83349, 0.73132, 0.58809, 0.43343, 0.29277,
               0.18126, 0.10285, 0.05348, 0.02549] )

# Вычисление произведения последовательности f и R0, отношение между количеством
# рождений женщин в последующих поколениях.
f = P * m
R0 = np.sum(f)
if R0 > 1:
    msg = 'R0 > 1: population grows'
else:
    msg = 'Population does not grow'

# Уравнение Эйлера-Лотки: поиск единственного действительного корня в r.
def func(r):
    return np.sum(f * np.exp(-r * x)) - 1

# Локализация корня и решение с использованием метода scipy.optimize.brentq.
a, b = 0, 10
r = brentq(func, a, b)
print('R0 = {:.3f} ({})'.format(R0, msg))
print('r = {:.5f} (lambda = {:.5f})'.format(r, np.exp(r)))
```

Вывод результата выполнения программы показан ниже:

```
R0 = 5.904 (R0 > 1: population grows)
r = 0.08742 (lambda = 1.09135)
```

Это значение r можно сравнить с приближенным значением, полученным Лесли и Рэнсоном и сопровожденным их комментарием:

«Искомый корень равен 0.087703, он немного превышает оценку значения r, к которому стремится эта последовательность. Это значение находится между 0.0861 (третья степень приближения) и 0.0877, но ближе к последнему, чем к первому. Вероятно, погрешность возникла в последнем десятичном знаке».

Пример П8.27. Метод Ньютона–Рафсона для поиска корней функции принимает начальную предпосылку для корня x_0 и выполняет процедуру поиска посредством постепенно улучшаемых приближений по формуле:

$$x_{n+1} = x_n - f(x_n)/f'(x_n).$$

Таким образом, на каждой итерации корень приближенно вычисляется как x_{n+1}, координата по оси x точки пересечения касательной к графику функции в точке $f(x_n)$. Если алгоритм применяется к функциям комплексной переменной z, то метод можно использовать для создания фракталов любопытной формы, рассматривая сходимость корня к множеству чисел на комплексной плоскости. Код в листинге 8.23 генерирует изображение фрактала (рис. 8.28), выделяя цветом соответствующие точки на комплексной плоскости, используемые как начальные предпосылки при поиске корня.

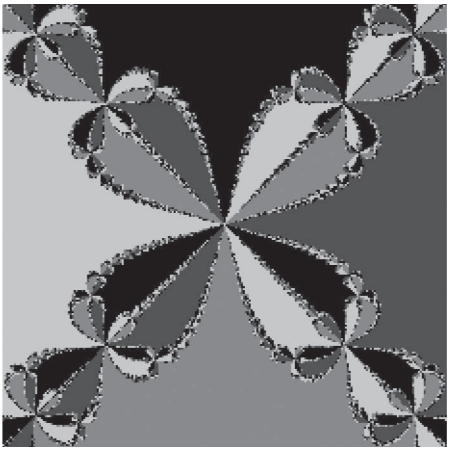


Рис. 8.28. Фрактал Ньютона для функции $f(z) = z^4 - 1$. Замысловатые самоподобные структуры наблюдаются для начальных предпосылок z_0 в интервале между корнями $(-1, 1, -i, i)$

Листинг 8.23. Создание изображения фрактала Ньютона

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap

# Список цветов для обозначения различных корней.
colors = ['b', 'r', 'g', 'y']

TOL = 1.e-8

def newton(z0, f, fprime, MAX_IT=1000):
    """The Newton-Raphson method applied to f(z).

    Returns the root found, starting with an initial guess, z0, or False
    if no convergence to tolerance TOL was reached within MAX_IT iterations.
```

```
    Метод Ньютона-Рафсона, применяемый к функции f(z).
    Возвращает найденный корень, начиная с исходной предпосылки z0, или False,
    если не достигнута сходимость в пределах допустимой погрешности TOL за MAX_IT итераций.
    """

    z = z0
    for i in range(MAX_IT):
        dz = f(z)/fprime(z)
        if abs(dz) < TOL:
            return z
        z -= dz
    return False

def plot_newton_fractal(f, fprime, n=200, domain=(-1, 1, -1, 1)):
    """Plot a Newton Fractal by finding the roots of f(z).

    The domain used for the fractal image is the region of the complex plane
    (xmin, xmax, ymin, ymax) where z = x + iy, discretized into n values along
    each axis.

    Создание изображения фрактала Ньютона по найденным корням функции f(z).
    Для изображения фрактала используется область комплексной плоскости
    (xmin, xmax, ymin, ymax), где z = x + iy с дискретизацией по n значениям
    на каждой оси.
    """
    roots = []
    m = np.zeros((n, n))

def get_root_index(roots, r):
    """Get the index of r in the list roots.

    If r is not in roots, append it to the list.

    Определение индекса r в списке корней.
    Если r нет в списке корней, то добавить его в этот список.
    """
    try:
        return np.where(np.isclose(roots, r, atol=TOL))[0][0]
    except IndexError:
        roots.append(r)
        return len(roots) - 1

xmin, xmax, ymin, ymax = domain
for ix, x in enumerate(np.linspace(xmin, xmax, n)):
    for iy, y in enumerate(np.linspace(ymin, ymax, n)):
        z0 = x + y*1j
        r = newton(z0, f, fprime)
        if r is not False:
            ir = get_root_index(roots, r)
            m[iy, ix] = ir
nroots = len(roots)
if nroots > len(colors):
    # Использование "непрерывной" цветовой схемы, если найдено слишком много корней.
    cmap = 'hsv'
else:
    # Использование списка цветов для цветовой схемы: по одному цвету для каждого корня.
    cmap = ListedColormap(colors[:nroots])
plt.imshow(m, cmap=cmap, origin='lower')
```

```
plt.axis('off')
plt.show()

f = lambda z: z**4 - 1
fprime = lambda z: 4*z**3
plot_newton_fractal(f, fprime, n=500)
```

8.4.4 Упражнения

Вопросы

В8.4.1. Использовать метод `scipy.optimize.brentq` для поиска решений уравнения

$$x + 1 = 1 / (x - 3)^3.$$

В8.4.2. Метод `scipy.optimize.newton` ошибается при поиске корней перечисленных ниже функций с заданной начальной точкой x_0. Объяснить, почему это происходит, и найти корни, изменив вызов метода `newton` или используя другой метод.

а) $f(x) = x^3 - 5x, \quad x_0 = 1.$

б) $f(x) = x^3 - 3x + 1, \quad x_0 = 1.$

в) $f(x) = 2 - x^5, \quad x_0 = 0.01.$

г) $f(x) = x^4 - (4.29)x^2 - 5.29, \quad x_0 = 0.8.$

В8.4.3. Траектория снаряда в плоскости xz, запущенного из исходной точки под углом θ_0 с начальной скоростью $v_0 = 25$ м/с, описывается формулой

$$z = x \operatorname{tg} \theta_0 - (g / 2v_0^2 \cos^2 \theta_0)\, x^2.$$

Если известно, что снаряд пролетел через точку с координатами (5, 15), то с помощью метода Брента определить возможные значения угла θ_0.

Задачи

38.4.1. Для прямоугольной области площадью A = 10 000 м^2, примыкающей к прямому участку реки, необходимо построить ограду (границу у реки ограживать не нужно). При каких размерах области a, b длина ограды будет минимальной? Проверить и убедиться, что алгоритм минимизации с ограничениями дает тот же результат, что и алгебраическое решение.

3.8.4.2. Найти все корни функции

$$f(x) = 1/5 + x \cos(3/x),$$

используя: а) метод `scipy.optimize.brentq`, б) метод `scipy.optimize.newton`.

38.4.3. Закон смещения Вина утверждает, что длина волны максимального излучения абсолютно черного тела, описываемого законом излучения Планка, пропорциональна $1/T$:

$$\lambda_{max} T = b,$$

где b – постоянная смещения Вина. Рассматривая распределение Планка для плотности излучаемой энергии как функцию от длины волны

$$u(\lambda, T) = (8\pi^2 hc / \lambda^5)\,(1 / (e^{hc/\lambda k_B T} - 1)),$$

определить значение постоянной смещения Вина b, используя метод `scipy.optimize.minimize_scalar` для поиска максимума функции $u(\lambda,T)$ при температурах в диапазоне 500 K ⩽ T ⩽ 6000 K и подогнать λ_{max} к прямой относительно $1/T$. Сравнить полученный результат с «точным» значением постоянной смещения Вина, доступным в пакете `scipy.constants` (см. раздел 8.1.1).

◊ **38.4.4.** Рассмотреть одномерный случай «ямы с бесконечными стенками» – модели квантовой механики для частицы, заключенной в «ящике» определенной формы (−1 ⩽ x ⩽ 1), описываемой уравнением Шрёдингера

$$-d^2\psi/dx^2 = E\psi$$

в единицах измерения энергии, для которых $\hbar^2/(2m) = 1$, где m – масса частицы. Точное решение для основного состояния этой системы записывается в виде

$$\psi = \cos(\pi x/2), \quad E = \pi^2/4.$$

Приближенное решение можно получить, используя вариационный принцип с минимизацией ожидаемого значения энергии контрольной (проверочной) волновой функции:

$$\psi_{\text{trial}} = \sum_{n=0}^{N} a_n \varphi_n(x)$$

с учетом коэффициентов a_n. Считая, что базисные функции имеют форму симметричного многочлена:

$$\varphi_n = (1 - x)^{N-n+1}(x + 1)^{n+1},$$

использовать методы `scipy.optimize.minimize` и `scipy.integrate.quad` для поиска оптимального значения для ожидаемой величины энергии (отношение Рэлея–Рица):

$$\mathcal{E} = \frac{\left\langle \psi_{trial} \middle| \hat{\mathrm{H}} \middle| \psi_{trial} \right\rangle}{\left\langle \psi_{trial} \middle| \psi_{trial} \right\rangle} = \frac{\int_{-1}^{1} \psi_{trial} \dfrac{\mathrm{d}^2}{\mathrm{d}x^2} \psi_{trial}\, \mathrm{d}x}{\int_{-1}^{1} \psi_{trial} \psi_{trial}\, \mathrm{d}x}.$$

Сравнить оцениваемую величину энергии $\mathcal{E}$ с точным значением при N = 1, 2, 3, 4. (Совет: рекомендуется использовать объекты `np.polynomial.Polynomial` для представления базисной и контрольной волновой функции.)

Глава 9

Анализ данных с помощью pandas

9.1 Введение в pandas

9.1.1 Что такое pandas

pandas – это широко распространенная библиотека на языке Python с открытым исходным кодом для обработки и анализа данных. В отличие от основной структуры данных библиотеки NumPy, главный объект pandas `DataFrame` может содержать неоднородные типы данных (числа с плавающей точкой, целые числа, строки, даты и т. д.), которые можно структурировать в виде иерархии и индексировать. Библиотека предоставляет большое количество векторизованных функций для очистки, преобразования и агрегации данных эффективными способами с использованием характерных приемов, аналогичных применяемым в библиотеке NumPy. Имя библиотеки образовано от английского словосочетания «panel data» (известного в другом варианте как «longitudinal data» – «данные многомерного временно́го ряда при долговременном наблюдении»), которое обозначает наборы данных нескольких переменных, наблюдаемых в течение нескольких (многих) интервалов времени для одного объекта.

На домашней странице pandas https://pandas.pydata.org/ содержится подробная информация о самой последней доступной версии библиотеки и инструкции по ее скачиванию и установке. В этой главе мы будем следовать общепринятому соглашению по импорту pandas как псевдонима `pd`:

```
import pandas as pd
```

Основными структурами данных pandas являются `Series` и `DataFrame`, представляющие одномерную последовательность значений и таблицу данных соответственно. В этом вводном разделе будут подробно описаны их главные свойства и практическое применение. В следующих разделах рассматриваются более продвинутые функциональные возможности и примеры приложений. pandas – большая и сложная библиотека с огромным спектром функциональных возможностей, но в этой главе описаны только базовые функции и варианты использования. Более подробные примеры представлены на веб-сайте книги.

9.1.2 Объект Series

В самой простой форме объект Series можно создать точно так же, как одномерный массив NumPy:

```
In [x]: river_lengths = pd.Series([6300, 6650, 6275, 6400])
In [x]: river_lengths
Out[x]:
0    6300
1    6650
2    6275
3    6400
dtype: int64
```

Объекту Series можно присвоить строку имени и тип данных dtype:

```
In [x]: river_lengths = pd.Series([6300, 6650, 6275, 6400], name='Length /km', dtype=float)
Out[x]:
0    6300.0
1    6650.0
2    6275.0
3    6400.0
Name: Length /km, dtype: float64
```

Но, в отличие от массива NumPy, каждый элемент в последовательности Series библиотеки pandas связан с индексом. Так как в приведенном выше примере индекс не был установлен явно, для индексирования по умолчанию используется последовательность целых чисел (начиная с 0):

```
In [x]: river_lengths.index
Out[x]: RangeIndex(start=0, stop=4, step=1)
```

RangeIndex – это объект pandas, работающий с эффективным использованием памяти, подобно встроенной функции Python range, предоставляющей монотонную последовательность целых чисел. Часто с помощью этого объекта удобно ссылаться на строки объекта Series с некоторыми другими метками, отличающимися от целочисленного индекса. Явную индексацию элементов можно обеспечить, передавая последовательность как аргумент index, или при создании объекта Series из словаря:

```
In [x]: river_lengths = pd.Series(data=[6300, 6650, 6275, 6400],
...:                       index=['Yangtze', 'Nile', 'Mississippi', 'Amazon'],
...:                       name='Length /km')
```

или:

```
In [x]: river_lengths = pd.Series(data={'Yangtze': 6300, 'Nile': 6650,
...:                                    'Mississippi': 6275, 'Amazon': 6400},
...:                              name='Length /km')
In [x]: river_lengths
Out[x]:
Yangtze        6300
```

```
Nile           6650
Mississippi    6275
Amazon         6400
Name: Length /km, dtype: int64
```

Это обеспечивает весьма выразительный и удобный способ ссылки на элементы последовательности `Series` с использованием меток индекса вместо целых чисел, в том числе и на отдельные элементы последовательности:

```
In [x]: river_lengths['Nile']
Out[x]: 6650
```

вместо `river_lengths[1]` или для представления элементов в другом порядке:

```
In [x]: river_lengths[['Amazon', 'Nile', 'Yangtze']]
Out[x]:
Amazon     6400
Nile       6650
Yangtze    6300
Name: Length /km, dtype: int64
```

вместо `river_lengths[ [3, 1, 0] ]`. Вырезание элементов в стиле Python также работает, как и ожидается:

```
In [x]: river_lengths[2::-1]
Out[x]:
Mississippi    6275
Nile           6650
Yangtze        6300
Name: Length /km, dtype: int64
```

Возможно даже использование нотации, похожей на стиль вырезания, но следует обратить особое внимание на то, что в данном случае указанный конечный элемент включается в вырезаемую группу:

```
In [x]: river_lengths['Nile':'Amazon']
Out[x]:
Nile           6650
Mississippi    6275
Amazon         6400
Name: Length /km, dtype: int64
```

Определенная метка индекса является корректным идентификатором языка Python, поэтому можно обращаться к строке данных как к атрибуту последовательности `Series`:

```
In [x]: river_lengths.Mississippi
Out[x]: 6275
```

Разумеется, возможно выполнение числовых операций с данными `Series` с использованием векторизации, как и для массивов NumPy:

```
In [x]: KM_TO_MILES = 0.621371
In [x]: river_lengths *= KM_TO_MILES
In [x]: river_lengths.name = 'Length /miles'
In [x]: river_lengths
Out[x]:
Yangtze        3914.637300
Nile           4132.117150
Mississippi    3899.103025
Amazon         3976.774400
Name: Length /miles, dtype: float64
```

В приведенном выше примере для обновления был также выбран атрибут `name` объекта `Series`. Обратите внимание: атрибут `dtype` был тоже изменен соответствующим образом с `int64` на `float64`, чтобы соответствовать значениям нового типа.

Операции сравнения и фильтрации объекта `Series` с помощью логических операций создают новый объект `Series`:

```
In [x]: river_lengths > 4000
Out[x]:
Nile            True
Amazon         False
Yangtze        False
Mississippi    False
Name: Length /miles, dtype: bool

In [x]: river_lengths[river_lengths <= 4000]
Out[x]:
Amazon         3976.774400
Yangtze        3914.637300
Mississippi    3899.103025
Name: Length /miles, dtype: float64
```

Операции проверки на существование элемента в `Series` используют индекс, а не значение:

```
In [x]: 'Yangtze' in river_lengths
Out[x]: True
```

Содержимое объекта `Series` можно сортировать по индексу или по значениям, используя методы `Series.sort_index` и `Series.sort_values` соответственно. По умолчанию эти методы возвращают новый объект `Series`, но их также можно применять для обновления исходного объекта `Series` с помощью аргумента `inplace=True`. Следующий аргумент `ascending` может содержать значение `True` (по умолчанию) или `False` для определения порядка сортировки:

```
In [x]: river_lengths.sort_index()
Out[x]:
Amazon         3976.774400
Mississippi    3899.103025
Nile           4132.117150
Yangtze        3914.637300
Name: Length /miles, dtype: float64
```

```
In [x]: river_lengths.sort_values(ascending=False, inplace=True)
In [x]: river_lengths
Out[x]:
Nile           4132.117150
Amazon         3976.774400
Yangtze        3914.637300
Mississippi    3899.103025
Name: Length /miles, dtype: float64
```

При объединении двух последовательностей они упорядочиваются по меткам индекса.

```
In [x]: masses = pd.Series({'Ganymede': 1.482e23,
                            'Callisto': 1.076e23,
                            'Io': 8.932e22,
                            'Europa': 4.800e22,
                            'Moon': 7.342e22,
                            'Earth': 5.972e24}, name='mass /kg')
In [x]: radii = pd.Series({'Ganymede': 2.634e6,
                           'Io': 1.822e6,
                           'Moon': 1.737e6,
                           'Earth': 6.371e6}, name='radius /m')
In [x]: from scipy.constants import G
In [x]: surface_g = G * masses / radii**2
In [x]: surface_g.name = 'surface gravity /m.s-2'
In [x]: surface_g.index.name = 'Body'
In [x]: surface_g
Body
Callisto         NaN
Earth       9.819650
Europa           NaN
Ganymede    1.425634
Io          1.795740
Moon        1.624075
Name: surface gravity /m.s-2, dtype: float64
```

Следует отметить, что если невозможно установить связь по индексам (метка индекса в одной последовательности `Series` отсутствует в другой последовательности), то получается нечисловой результат NaN («Not a Number»). Это можно проверить методами `isnull` и `notnull`:

```
In [x]: surface_g.isnull()
Out[x]:
Body
Callisto     True
Earth       False
Europa       True
Ganymede    False
Io          False
Moon        False
Name: surface gravity /m.s-2, dtype: bool
```

Для возврата списка без каких-либо отсутствующих значений необходимо отфильтровать его с помощью конструкции `surface_g[surface_g.notnull()]` или воспользоваться методом `dropna`:

```
In [x]: surface_g.dropna()
Out[x]:
Body
Earth       9.819650
Ganymede    1.425634
Io          1.795740
Moon        1.624075
Name: surface gravity /m.s-2, dtype: float64
```

Для преобразования объекта `Series` в массив `ndarray` библиотеки NumPy (с удалением индекса и прочих метаданных) используется свойство `values`:

```
In [x]: surface_g.values
Out[x]: array([       nan, 9.81964974,       nan, 1.42563409, 1.79573967,
               1.62407526])
```

Пример П9.1. Элементы NaN в объекте `Series` библиотеки pandas можно заменить на заданное значение, используя метод `fillna`:

```
In [x]: ser1 = pd.Series({'b': 2, 'c': -5, 'd': 6.5}, index=list('abcd'))
In [x]: ser1
Out[x]:
a    NaN
b    2.0
c   -5.0
d    6.5
dtype: float64

In [x]: ser1.fillna(1, inplace=True)
In [x]: ser1
Out[x]:
a    1.0
b    2.0
c   -5.0
d    6.5
dtype: float64
```

Бесконечности (представленные числовым значением с плавающей точкой `inf`) можно заменить с помощью метода `replace`, который принимает скалярное значение или последовательность значений и заменяет их на другое единственное значение:

```
In [x]: ser2 = pd.Series([-3.4, 0, 0, 1], index=ser1.index)
In [x]: ser2
Out[x]:
a   -3.4
b    0.0
c    0.0
d    1.0
dtype: float64

In [x]: ser3 = ser1 / ser2
In [x]: ser3
```

```
Out[x]:
a   -0.294118
b         inf
c        -inf
d    6.500000
dtype: float64

In [x]: ser3.replace([np.inf, -np.inf], 0)
Out[x]:
a   -0.294118
b    0.000000
c    0.000000
d    6.500000
dtype: float64
```

(Предполагается, что библиотека NumPy импортирована инструкцией `import numpy as np`.)

9.1.3 Объект DataFrame

Создание объекта DataFrame

Объект DataFrame – эта двумерная таблица данных, которую можно интерпретировать как упорядоченный набор столбцов объекта Series с одинаковым индексом. Для создания простого объекта DataFrame из словаря необходимо присвоить последовательности значений[119] именам столбцов, как ключам:

```
In [x]: data = {'mass': [1.482e23, 1.076e23, 8.932e22, 4.800e22, 7.342e22],
                'radius': [2.634e6, None , 1.822e6, None , 1.737e6],
                'parent': ['Jupiter', 'Jupiter', 'Jupiter', 'Jupiter', 'Earth']
               }
In [x]: index = ['Ganymede', 'Callisto', 'Io', 'Europa', 'Moon']
In [x]: df = pd.DataFrame(data, index=index)
In [x]: df
Out[x]:
                  mass     radius   parent
Ganymede  1.482000e+23  2634000.0  Jupiter
Callisto  1.076000e+23        NaN  Jupiter
Io        8.932000e+22  1822000.0  Jupiter
Europa    4.800000e+22        NaN  Jupiter
Moon      7.342000e+22  1737000.0    Earth
```

Элементам, значения которых были представлены как None в исходных данных, присвоены значения NaN в объекте DataFrame. Возможно, потребуется переименование столбца или строки индекса: для этого вызывается метод rename, объявляющий, какая именно ось ('index' (то же самое, что 'rows', но по умолчанию) или 'columns'[120]) содержит метку/метки, которые должны быть переименованы. В метод также передается словарь, отображающий каждую исходную метку в заменяющее ее значение. Напомню о необходимости уста-

[119] Здесь (неопределенные явно) единицы измерения принимаются в системе СИ: кг и м.

[120] Можно также ссылаться на строки и столбцы объекта DataFrame как axis=0 и axis=1 соответственно.

новки значения `inplace=True`, если требуется изменение исходного `DataFrame`, а не возврат новой копии этого объекта. Например:

```
In [x]: df.rename({'parent': 'planet'}, axis='columns', inplace=True)
In [x]: df.rename({'Moon': 'The Moon'}) # Изменение метки строки индекса.
Out[x]:
                 mass     radius   planet
Ganymede  1.482000e+23  2634000.0  Jupiter
Callisto  1.076000e+23        NaN  Jupiter
Io        8.932000e+22  1822000.0  Jupiter
Europa    4.800000e+22        NaN  Jupiter
The Moon  7.342000e+22  1737000.0    Earth
```

Эта последняя инструкция вернула новый объект `DataFrame`, но не изменила исходный объект `df`.

Доступ к строкам, столбцам и ячейкам

Доступ к отдельному столбцу можно получить по индексу или по атрибуту (если его имя является корректным идентификатором Python):

```
In [x]: df['mass'] # Или df.mass.
Out[x]:
Ganymede    1.482000e+23
Callisto    1.076000e+23
Io          8.932000e+22
Europa      4.800000e+22
Moon        7.342000e+22
Name: mass, dtype: float64
```

Поскольку этот столбец является объектом `Series` библиотеки pandas, отдельные значения можно извлечь по позиции или по ссылке на метку индекса:

```
In [x]: df['mass'][2]
Out[x]: 8.932e+22
In [x]: df['mass']['Io']         # Или df['mass'].Io, или df.mass.Io
Out[x]: 8.932e+22
```

Для извлечения столбцов и отдельных значений этот способ успешно работает, но при попытке присваивания выводится предупреждающее сообщение:

```
In [x]: df['radius']['Callisto'] = 2.410e6
/Users/christian/envs/py37/bin/ipython:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame
(Попытка установки значения в копии фрагмента, вырезанного из DataFrame)
…
```

В рассматриваемом здесь случае попытка удалась:

```
In [x]: df['radius']['Callisto']
Out[x]: 2410000.0
```

но выведенное сообщение – это общее предупреждение о том, что «цепочное индексирование» ([..][..]) может привести к непредсказуемым результатам, когда применяется для присваивания: в зависимости от того, как данные сохраняются в памяти, вероятно, что индексирующее выражение выдаст копию данных, а не их представление. Именно о возможности присваивания копии, а не измененных данных непосредственно в оригинале предупреждает сообщение SettingWithCopyWarning. Поэтому рекомендуется избегать цепочного индексирования в операторах присваивания.

Методы объекта DataFrame loc и iloc можно использовать для надежного и безопасного доступа и присваивания столбцам, строкам (записям) и ячейкам, поэтому их использование настоятельно рекомендуется. Метод loc выбирает элементы из строк и метки столбцов:

```
In [x]: df.loc['Europa']
Out[x]:
mass      4.8e+22
radius        NaN
planet    Jupiter
Name: Europa, dtype: object
```

Единственная строка данных, проиндексированная по метке Europa, возвращается как объект Series. Если требуется только подмножество столбцов, то передаются их имена в последовательности по второй оси[121]:

```
In [x]: df.loc['Europa', ['mass', 'planet']]
Out[x]:
mass      4.8e+22
planet    Jupiter
Name: Europa, dtype: object
```

Вырезание группы элементов, расширенное индексирование и логическое индексирование также поддерживаются методом loc:

```
In [x]: df.loc[:, 'mass']   # То же самое, что df ['mass'], - возвращает объект Series.
Out[x]:
Ganymede    1.482000e+23
Callisto    1.076000e+23
Io          8.932000e+22
Europa      4.800000e+22
Moon        7.342000e+22
Name: mass, dtype: float64

In [x]: df.loc['Ganymede ':'Io', ['mass ', 'radius ']]
Out[x]:
mass radius
Ganymede 1.482000e+23 2634000.0
Callisto 1.076000e+23 2410000.0
Io 8.932000e+22 1822000.0
```

[121] Обратите внимание: несмотря на то что цепочное индексирование ссылается на ячейку в формате «столбец, строка»: df[col][row], метод loc определяет ячейки в противоположном порядке: df.loc[row, col] или df.loc[row][col].

```
In [x]: df.loc[['Moon', 'Europa'], 'planet ']
Out[x]:
Moon        Earth
Europa    Jupiter
Name: planet, dtype: object

In [x]: df.loc[df.planet=='Jupiter', 'radius']
Out[x]:
Ganymede    2634000.0
Callisto    2410000.0
Io          1822000.0
Europa      NaN
Name: radius, dtype: float64
```

Таким образом, значение одной ячейки можно извлечь по меткам строки и столбца:

```
In [x]: df.loc['Europa', 'mass']
Out[x]: 4.8e+22
```

Это безопасный способ изменения данных в объекте `DataFrame`:

```
In [x]: df.loc['Europa', 'radius'] = 1.561e6 # Нет предупреждения, данные изменены в исходном
                                             # объекте.
In [x]: df.loc['Europa']
Out[x]:
mass        4.8e+22
radius    1.561e+06
parent      Jupiter
Name: Europa, dtype: object
```

Метод `loc` чрезвычайно часто используется в сочетании с логической индексацией для фильтрации строк по значениям столбцов. Например, по массам спутников Юпитера:

```
In [x]: df.loc[df.planet=='Jupiter', 'mass']
Out[x]:
Ganymede    1.482000e+23
Callisto    1.076000e+23
Io          8.932000e+22
Europa      4.800000e+22
Name: mass, dtype: float64
```

Строки, соответствующие спутникам с радиусом менее 2000 км:

```
In [x]: df.loc[df.radius < 2.e6]
Out[x]:
                mass     radius   planet
Io      8.932000e+22  1822000.0  Jupiter
Europa  4.800000e+22  1561000.0  Jupiter
Moon    7.342000e+22  1737000.0  Earth
```

Второй метод `iloc` извлекает данные по позиции числового индекса:

```
In [x]: df.iloc[1]          # Вторая строка.
Out[x]:
mass      1.076e+23
radius     2.41e+06
parent      Jupiter
Name: Callisto, dtype: object

In [x]: df.iloc[:, [1, 2]]     # Все строки, второй и третий столбцы.
Out[x]:
             radius   planet
Ganymede  2634000.0  Jupiter
Callisto  2410000.0  Jupiter
Io        1822000.0  Jupiter
Europa    1561000.0  Jupiter
Moon      1737000.0    Earth

In [x]: df.iloc[-1, 1]        # Последняя строка, второй столбец.
Out[x]: 1737000.0
```

Кроме того, для одного скалярного значения существуют методы `at` и `iat`:

```
In [x]: df.at['Moon', 'mass']        # То же самое, что df.loc['Moon', 'mass']
Out[x]: 7.342e+22
In [x]: df.iat[-1, 0]                # То же самое, что df.iloc[-1, 0]
Out[x]: 7.342e+22
```

Пример П9.2. Существует потенциальный источник путаницы при использовании метода `loc` для `Series` или `DataFrame` с целочисленным индексом: важно помнить, что метод `loc` всегда ссылается на индекс меток, тогда как метод `iloc` принимает целочисленный индекс позиции (начинающийся с нуля):

```
In [x]: df = pd.DataFrame(np.arange(12).reshape(4, 3) + 10,
                          index=[1, 2, 3, 4], columns=list('abc'))
In [x]: df
Out[x]:
    a   b   c
1  10  11  12
2  13  14  15
3  16  17  18
4  19  20  21

In [x]: df.loc[1]     # Строка с индексом *метки* 1 (первая строка).
Out[x]:
a    10
b    11
c    12
Name: 1, dtype: int64

In [x]: df.iloc[1]    # Строка с индексом *позиции* 1 (строка, помеченная индексом 2).
a    13
b    14
c    15
Name: 2, dtype: int64
```

Следует также отметить, что индекс меток не обязательно должен быть неповторяющимся:

```
In [x]: df.index = [1, 2, 2, 3]      # Изменение индекса меток.
In [x]: df
Out[x]:
    a   b   c
1  10  11  12
2  13  14  15
2  16  17  18
3  19  20  21

In [x]: df.loc[2]     # Объект DataFrame: все строки с меткой 2.
Out[x]:
    a   b   c
2  13  14  15
2  16  17  18

In [x]: df.iloc[2]    # Объект Series: существует только одна строка с позицией по индексу 2.
Out[x]:
a    16
b    17
c    18
Name: 2, dtype: int64
```

Совместное использование объектов Series и DataFrame

Другой способ создания объекта DataFrame – из словаря с вложениями или из словаря объектов Series. В любом случае ключи внешнего словаря содержат имена столбцов, а объекты Series и внутренние (вложенные) словари интерпретируются как строки:

```
boeing_wingspan = pd.Series({'B747 -8': 68.4, 'B777 -9': 64.8, 'B787 -10': 60.12},
                            name='wingspan')
boeing_length = pd.Series({'B747 -8': 76.3, 'B777 -9': 76.7, 'B787 -10': 68.28},
                            name='length')
boeing_range = pd.Series({'B777 -9': 13940, 'B787 -10': 11910}, name='range', dtype=float)

# Создание DataFrame из словаря объектов Series.
df_boeing = pd.DataFrame({'wingspan': boeing_wingspan, 'length': boeing_length,
                          'range': boeing_range})

# Создание DataFrame из словаря с вложенными словарями.
df_airbus = pd.DataFrame({'range': {'A350 -1000': 16100, 'A380 -800': 14800},
                          'wingspan': {'A350 -1000': 64.75, 'A380 -800': 79.75},
                          'length': {'A350 -1000': 73.8, 'A380 -800': 72.72} })

In [x]: df_boeing
Out[x]:
         wingspan   length     range
B747-8      68.40    76.30       NaN
B777-9      64.80    76.70   13940.0
B787-10     60.12    68.28   11910.0
```

```
In [x]: df_airbus
Out[x]:
           range  wingspan   length
A350-1000  16100     64.75    73.80
A380-800   14800     79.75    72.72
```

Обратите внимание: отсутствующие значения в столбцах становятся значением NaN в объекте DataFrame. Для объединения двух объектов DataFrame используется метод pd.concat[122]:

```
In [x]: pd.concat((df_airbus, df_boeing))
Out[x]:
           length    range  wingspan
A350-1000   73.80  16100.0     64.75
A380-800    72.72  14800.0     79.75
B747-8      76.30      NaN     68.40
B777-9      76.70  13940.0     64.80
B787-10     68.28  11910.0     60.12
```

Применение метода df_airbus.append(df_boeing) должно давать точно такой же результат.

Для добавления одного столбца в объект DataFrame необходима операция присваивания последовательности значений или объект Series:

```
In [x]: df_airbus['speed'] = [950, 903]
In [x]: df_airbus

Out[x]:
           range  wingspan  length  speed
A350-1000  16100     64.75   73.80    950
A380-800   14800     79.75   72.72    903
```

При объединении объектов DataFrame с различными столбцами неизвестные значения заменяются на значения NaN:

```
In [x]: df_aircraft = pd.concat((df_airbus, df_boeing))
In [x]: df_aircraft

Out[x]:
           length    range  speed  wingspan
A350-1000   73.80  16100.0  950.0     64.75
A380-800    72.72  14800.0  903.0     79.75
B747-8      76.30      NaN    NaN     68.40
B777-9      76.70  13940.0    NaN     64.80
B787-10     68.28  11910.0    NaN     60.12
```

[122] Следует отметить, что для методов concat и append требуется копирование данных в новый объект DataFrame, и для больших наборов данных такой подход может оказаться медленным и неэффективным с точки зрения использования памяти. В этом случае, если возможно, лучше предварительно выделить объем памяти требуемого размера для пустого объекта DataFrame и вставлять данные непосредственно в этот объект.

Следует отметить, что операция извлечения объекта `Series` как строки или столбца возвращает представление (view) объекта `DataFrame`, поэтому изменения в извлеченном объекте `Series` будут отображаться в исходном объекте `DataFrame`:

```
In [x]: speeds = df_aircraft['speed ']
In [x]: speeds['B747-8','B787-10'] = 903, 956      # Изменения данных df_aircraft.
In [x]: jumbo = df_aircraft.loc['B747-8']
In [x]: jumbo.range = 15000                        # Изменение данных df_aircraft.
In [x]: df_aircraft
Out[x]:
           length    range  speed  wingspan
A350-1000   73.80  16100.0  950.0     64.75
A380-800    72.72  14800.0  903.0     79.75
B747-8      76.30  15000.0  903.0     68.40
B777-9      76.70  13940.0    NaN     64.80
B787-10     68.28  11910.0  956.0     60.12
```

Для удаления столбца из объекта `DataFrame` используется ключевое слово языка Python `del`:

```
In [x]: del df_aircraft['speed']   # Внимание: НЕ del df_aircraft.speed.
In [x]: df_aircraft

Out[x]:
           length    range  wingspan
A350-1000   73.80  16100.0     64.75
A380-800    72.72  14800.0     79.75
B747-8      76.30  15000.0     68.40
B777-9      76.70  13940.0     64.80
B787-10     68.28  11910.0     60.12
```

Функцию `drop` можно использовать для выборочного удаления строк и столбцов из объекта `DataFrame`. Если не определен аргумент `inplace=True`, то возвращается новый объект:

```
In [x]: df_aircraft.drop(['A350-1000', 'A380-800'])     # Удаление строк по умолчанию.
Out[x]:
         length    range  wingspan
B747-8    76.30  15000.0     68.40
B777-9    76.70  13940.0     64.80
B787-10   68.28  11910.0     60.12

In [x]: df_aircraft.drop(['length', 'wingspan'], axis='columns', inplace=True)
In [x]: df_aircraft
Out[x]:
             range
A350-1000  16100.0
A380-800   14800.0
B747-8     15000.0
B777-9     13940.0
B787-10    11910.0
```

9.1.4 Сортировка, арифметические и статистические функции

Как можно было ожидать, многие наиболее полезные функции для анализа данных доступны в библиотеке pandas.

Пример П9.3. Файл *india-data.csv*, доступный для скачивания по адресу https://scipython.com/eg/bak, содержит столбцы демографических данных по 36 штатам и союзным территориям Индии. При чтении содержимого этого файла с использованием инструкции

```
In [x]: df = pd.read_csv('india -data.csv', index_col=0)
```

(более подробное описание примененного метода см. в следующем разделе) полученный в результате объект DataFrame содержит индекс по названию штата / союзной территории и показанные ниже столбцы:

```
In [x]: df.index
Out[x]:
Index(['Uttar Pradesh', 'Maharashtra', 'Bihar', 'West Bengal',
       ...
       'Dadra and Nagar Haveli', 'Daman and Diu', 'Lakshadweep'],
      dtype='object', name='State/UT')

In [x]: df.columns
Out[x]:
Index(['Male Population', 'Female Population', 'Area (km2)',
       'Male Literacy (%)', 'Female Literacy (%)', 'Fertility Rate'],
      dtype='object')
```

Можно быстро просмотреть объект `DataFrame` с помощью метода `df.head(`*n*`)`, который выводит первые `n` строк (или первые пять строк, если значение `n` не задано):

```
In [x]: df.head()
Out[x]:
               Male Population  ...  Female Literacy (%)
State/UT                        ...
Uttar Pradesh         104480510  ...                59.26
Maharashtra            58243056  ...                75.48
Bihar                  54278157  ...                53.33
West Bengal            46809027  ...                71.16
Madhya Pradesh         37612306  ...                60.02

[5 rows x 5 columns]
```

Функции библиотеки pandas существенно упрощают вычисления значений новых столбцов в полученном объекте `DataFrame`:

```
In [x]: df['Population'] = df['Male Population'] + df['Female Population']
In [x]: total_pop = df['Population'].sum()
In [x]: print(f'Total population: {total_pop:,d}')
Total population: 1,210,754,977
```

```
In [x]: df['Population Density (km-2)'] = df['Population'] / df['Area (km2)']
In [x]: df.loc['West Bengal', 'Population Density (km-2)']
Out[x]: 1028.440091490896        # Плотность населения в Западной Бенгалии.

In [x]: total_pop / df['Area (km2)'].sum()
Out[x]: 368.3195047153525        # Средняя плотность населения.
```

Максимальные и минимальные значения вычисляются тем же способом, что и в библиотеке NumPy, например:

```
In [x]: df['Male Literacy (%)'].min()
Out[x]: 73.39
```

Вероятно, более полезны методы `idxmin` и `idxmax`, которые возвращают индексы меток минимальных и максимальных значений соответственно:

```
In [x]: df['Area (km2)'].idxmax()      # Наибольший по площади штат / союзная территория.
Out[x]: 'Rajasthan'
```

Разумеется, возвращаемое значение можно передать в метод `df.loc`, чтобы получить полную строку. Например, строку, соответствующую штату / союзной территории с наибольшей плотностью населения:

```
In [x]: df.loc[df['Population Density (km-2)'].idxmax()]
Out[x]:
Male Population              8887326
Female Population            7800615
Area (km2)                   1484
Male Literacy (%)            91.03
Female Literacy (%)          80.93
Population                   16687940
Population Density (km-2)    1.124524e+04
Name: Delhi, dtype: float64
```

Корреляцию статистических данных между объектами `DataFrames` или `Series` можно вычислить с помощью метода `corr`:

```
In [x]: df['Female Literacy (%)'].corr(df['Fertility Rate'])
Out[x]: -0.7361949271996956
```

В этом случае (сравниваются два столбца данных) вычисляется единственный коэффициент корреляции. В более общем случае возвращается корреляционная матрица как новый `DataFrame`. Средства библиотеки pandas можно использовать для быстрого создания разнообразных простых графиков и диаграмм с надписями из объекта `DataFrame` с помощью группы методов `df.plot`. По умолчанию эти методы в качестве внутренней основы используют средства библиотеки Matplotlib, поэтому их синтаксис совпадает с описанным в главе 7. Например:

```
In [x]: df.plot.scatter('Female Literacy (%)', 'Fertility Rate')
```

На рис. 9.1 показан созданный график.

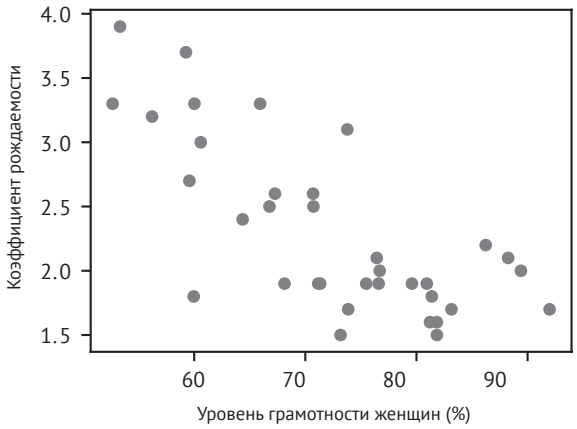


Рис. 9.1. Точечная диаграмма коэффициента рождаемости в зависимости от уровня грамотности женщин в 36 штатах и союзных территориях Индии

9.2 Чтение и запись объектов Series и DataFrame

9.2.1 Чтение текстовых файлов

Текстовые файлы с разделителями

Основным методом для чтения текстовых файлов с данными в объекте `DataFrame` является `pd.read_csv`. Метод работает в основном так же, как метод из библиотеки NumPy `genfromtxt`, но обладает дополнительной функциональностью для операций именования столбцов и создания индексов в объектах `DataFrame`. Метод принимает не менее 49 возможных аргументов, но ниже описаны только наиболее важные из них:

- `filepath_or_buffer` (обязательный) – путь к считываемому файлу: это может быть локальный файл или URL для скачивания данных из интернета;
- `sep` – разделитель столбцов. По умолчанию запятая `','`, но можно использовать значение `'/s+'` для столбцов, разделенных пробельными символами, `'\t'` для разделителей-табуляций или `None` для того, чтобы библиотека pandas сама попыталась логически определить символ-разделитель. См. также `delim_whitespace`;
- `delimiter` – псевдоним («алиас») для аргумента `sep`;
- `header` – номера строк (индексы), используемые для имен столбцов. По умолчанию `header=0`: использование первой строки для имен столбцов. Примечание: если в файле нет заголовка, то можно присвоить значение `header=None` и определить имена столбцов в аргументе `names`;
- `names` – последовательность неповторяющихся имен столбцов, которые будут использоваться. Если в файле нет заголовка, то можно установить значение `header=None` в дополнение к определению списка имен `names`;

- `index_col` – столбцы (столбец), используемые как метки строк в объекте `DataFrame`;
- `usecols` – последовательность индексов столбцов (как для метода `loadtxt` из библиотеки NumPy) или имен столбцов, идентифицирующих соответствующие столбцы, которые должны считываться в объект `DataFrame`;
- `squeeze` – если требуемые данные состоят из одного столбца, то при `squeeze=True` будет возвращать объект `Series` вместо объекта `DataFrame` по умолчанию;
- `converters` – словарь функций для преобразования значений в заданных столбцах входного файла в значения данных для объекта `DataFrame`. Ключами словаря могут быть индексы или имена столбцов;
- `skiprows` – целое число, определяющее количество строк от начала файла, которые будут пропущены перед чтением данных, или последовательность, определяющая индексы пропускаемых строк;
- `skipfooter` – количество строк, пропускаемых в конце файла (по умолчанию 0);
- `nrows` – количество строк, считываемых из файла: это удобно при чтении подмножества строк из очень большого файла для тестирования или обследования содержащихся в нем данных;
- `na_values` – строка или последовательность строк, интерпретируемых как значения NaN, в дополнение к значениям по умолчанию, включающим `'NaN'`, `'NA'`, `'NULL'` и `'#N/A'` (полный список см. в официальной документации);
- `parse_dates` – установка значения `True` позволяет интерпретировать (синтаксически) индекс(ы) столбца (столбцов) как последовательность объектов `datetime` (см. раздел 9.3.2). Для этого аргумента доступны и другие варианты значений (см. официальную онлайновую документацию);
- `comment` – определяет один символ, например `'#'`, который при обнаружении в начале строки сообщает о том, что всю строку необходимо пропустить;
- `skip_blank_lines` – по умолчанию задано значение `True` – пропуск пустых строк во входном файле. При установке значения `False` пустые строки интерпретируются как строки из значений `NaN`;
- `delim_whitespace` – можно установить для этого аргумента значение `True` вместо определения аргумента `sep='\s+'`, чтобы установить, что столбцы данных разделяются пробельными символами.

Пример П9.4. Файл *ionization-energies.csv*, доступный для скачивания по адресу https://scipython.com/eg/baq, содержит значения энергии ионизации (в эВ) некоторых элементов периодической таблицы:

```
Ionization Energies (eV) of the first few elements of the periodic table
Element, IE1, IE2, IE3, IE4, IE5
H, 13.59844
He, 24.58741, 54.41778
Li, 5.39172, 75.64018, 122.45429
Be, 9.3227, 18.21116, 153.89661, 217.71865
B, 8.29803, 25.15484, 37.93064, 259.37521, 340.22580
```

```
C, 11.26030, 24.38332, 47.8878, 64.4939, 392.087
N, 14.53414, 29.6013, 47.44924, 77.4735, 97.8902
O, 13.61806, 35.11730, 54.9355, 77.41353, 113.8990
F, 17.42282, 34.97082, 62.7084, 87.1398, 114.2428
Ne, 21.5646, 40.96328, 63.45, 97.12, 126.21
Na, 5.13908, 47.2864, 71.6200, 98.91, 138.40
```

Эти данные можно прочитать в объект `DataFrame`, как показано ниже. Здесь предполагается, что нас интересуют только первые два столбца периодической таблицы и первые четыре значения энергии ионизации:

```
In [x]: df = pd.read_csv('ionization -energies.csv', skiprows=1, index_col=0,          ❶
...:                     usecols=range(5), nrows=11)
In [x]: df.columns = df.columns.str.strip()                                            ❷
In [x]: print('Second ionization energy of Li: {} eV'.format(df.loc['Li'].IE2))

Second ionization energy of Li: 75.64018 eV
```

❶ Обратите внимание: аргумент `usecols` включает столбец, который необходимо определить как индекс в `DataFrame`, а аргумент `nrows` включает заголовки столбцов (но не пропущенные строки).

❷ Пробельные символы, окружающие столбцы, не удаляются автоматически. Библиотека pandas предоставляет разнообразные методы для обработки строк в пространстве имен «методов доступа» `str`, которые можно применять ко всем именам столбцов в одной инструкции. Это быстрее, чем использование метода `rename`:

```
df.rename(columns=lambda s: s.strip(), inplace=True)
```

Пример П9.5. Приведенный ниже текстовый файл, доступный для скачивания по адресу https://scipython.com/eg/bao, содержит данные о 13 витаминах, важных для здоровья человека.

```
List of vitamins, their solubility (in fat or water) and recommended dietary
allowances for men / women.
Data from the US Food and Nutrition Board, Institute of Medicine, National
Academies

Vitamin A    Fat      900ug/700ug

Vitamin B1   Water    1.2mg/1.1mg
Vitamin B2   Water    1.3mg/1.1mg
Vitamin B3   Water    16mg/14mg
Vitamin B5   Water    5mg
Vitamin B6   Water    1.5mg/1.4mg
Vitamin B7   Water    30ug
Vitamin B9   Water    400ug
Vitamin B12  Water    2.4ug

Vitamin C    Water    90mg/75mg
Vitamin D    Fat      15ug
Vitamin E    Fat      15mg
```

```
Vitamin K    Fat      110ug/120ug
--- Data for guidance only, consult your physician ---
```

Рекомендуемые (ежедневные) диетические дозы в некоторых строках указаны в двух единицах измерения в последнем столбце, в некоторых случаях они различны для мужчин и женщин. Если требуется синтаксический разбор этого столбца, чтобы получить среднее значение в мкг, то можно воспользоваться функцией преобразования, как показано в коде листинга 9.1.

Листинг 9.1. Считывание текстовой таблицы с данными о витаминах

```
import pandas as pd

def average_rda_in_micrograms(col):
    def ensure_micrograms(s):
        if s.endswith('ug'):
            return float(s[:-2])
        elif s.endswith('mg'):
            return float(s[:-2]) * 1000
        raise ValueError(f'Unrecognised units in {s}')
    fields = col.split('/')
    return sum([ensure_micrograms(s) for s in fields]) / len(fields)

df = pd.read_csv('vitamins.txt', delim_whitespace=True, skiprows=4,
                 skipfooter=1, header=None, usecols=(1, 2, 3),
                 converters={'RDA': average_rda_in_micrograms},
                 names=['Vitamin', 'Solubility', 'RDA'],
                 index_col=0
                )
```

В этом коде пропускаются четыре строки заголовка и одна строка нижнего колонтитула (пустые строки пропускаются автоматически). Значение индекса устанавливается по первому реально используемому столбцу (`index_col=0`, идентифицирующему витамин). Функция преобразования вычисляет средние величины для встречающихся числовых значений (после преобразования их в мкг), при этом предполагается, что несколько значений разделены символом косой черты (/).

Текстовые файлы с содержимым постоянной ширины

Метод `read_fwf` считывает отформатированные файлы с содержимым постоянной ширины. Значения ширины полей передаются как список кортежей в аргументе `colspecs`, как полуоткрытые интервалы полей, предназначенных для чтения в каждой строке, т. е. `(i, j)` ссылается на поля, начиная с индекса `i` до индекса `j-1`. Другой вариант: если интервалы непрерывные, то список значений ширины полей можно передать в аргументе `widths`.

Вернемся к примеру использования метода `np.genfromtxt` из раздела 6.2.3. Показанный ниже небольшой файл *data.txt* состоит из четырех столбцов со значениями ширины 2, 1, 9 и 3 символа (пробелы обозначены символом ‘␣’):

```
␣12␣␣100.231.03
␣11␣1201.842.04
␣11␣␣␣99.324.02
```

Для считывания содержимого этого файла с помощью средств библиотеки pandas используется следующий формат вызова метода:

```
df = pd.read_fwf('data.txt',
                 colspecs=[(0, 2), (2, 3), (3, 12), (12, 15)], header=None)
```

или с учетом того, что интервалы непрерывные:

```
df = pd.read_fwf('data.txt', widths=(2, 1, 9, 3), header=None)
```

В результате получаем объект `DataFrame` со следующим содержимым:

```
   0  1          2     3
0  1  2    100.231  0.03
1  1  1   1201.842  0.04
2  1  1     99.324  0.02
```

9.2.2 Запись в текстовые файлы

Метод объекта `DataFrame` `to_csv` выводит данные из этого объекта в текстовый файл, отформатированный в соответствии со значениями аргументов, описанных ниже[123]:

- `path_or_buf` – путевое имя файла или файловый объект для вывода. Если задано значение `None`, то содержимое `DataFrame` возвращается как строка;
- `sep` – один символ – разделитель полей (по умолчанию запятая `','`);
- `na_rep` – строка, используемая для представления отсутствующих данных (по умолчанию пустая строка `''`);
- `float_format` – спецификатор формата в стиле языка C (см. раздел 2.3.7) для чисел с плавающей точкой;
- `columns` – последовательность, идентифицирующая выводимые столбцы;
- `header` – по умолчанию значение `True`, обозначающее, что имена столбцов должны выводиться. Можно задать значение `False` или список имен столбцов;
- `index` – по умолчанию значение `True`, обозначающее, что имена строк должны выводиться;
- `compression` – одно из допустимых значений `'infer'`, `'gzip'`, `'bz2'`, `'zip'`, `'xz'`, `None` для обозначения необходимости и способа сжатия файла вывода. По умолчанию задано значение `'infer'`: библиотека pandas сама определяет требуемый метод сжатия по расширению файла.

[123] Полное описание см. в документации здесь: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.to_csv.html.

Пример П9.6. Для записи файла данных о витаминах, разделенных запятыми, из объекта `DataFrame`, созданного в примере П9.5, можно воспользоваться методом `to_csv`, как показано ниже:

```
df.to_csv('vitamins.csv', float_format='%.1f', columns=['Solubility', 'RDA'])
```

Содержимое записанного файла:

```
Vitamin,Solubility,RDA
A,Fat,800.0
B1,Water,1150.0
B2,Water,1200.0
B3,Water,15000.0
B5,Water,5000.0
B6,Water,1450.0
B7,Water,30.0
B9,Water,400.0
B12,Water,2.4
C,Water,82500.0
D,Fat,15.0
E,Fat,15000.0
K,Fat,115.0
```

9.2.3 Файлы Microsoft Excel

Библиотека pandas предоставляет возможность считывать содержимое объекта `DataFrame` из файлов Excel с расширениями *.xls* и *.xlsx* с помощью функции `pd.read_excel`. Возможно, потребуется отдельная установка пакета xlrd[124] с помощью менеджера пакетов Python или утилиты `pip` из командной строки, например:

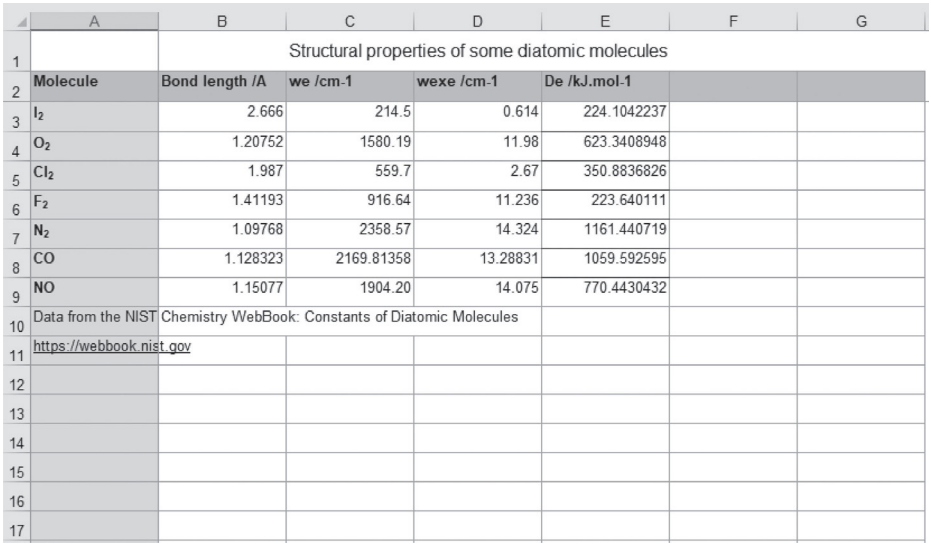
```
pip install xlrd
```

Путевое имя файла для документа Excel передается как первый аргумент в метод `read_excel`. Большинство дополнительных аргументов уже было описано выше для метода `read_csv` – они аналогичны, за исключением того, что в аргументе `usecols` можно передавать список индексов столбцов или строку, содержащую диапазон меток столбцов Excel, например `'B:K'`, `'A,D,G:K'`.

По умолчанию для чтения используется только первый лист файла (документа). Чтобы читать другой лист или несколько листов, необходимо передать один или несколько индексов либо имен листов в аргументе `sheet_name`.

Пример П9.7. Файл Excel *bond-lengths.xlsx*, доступный для скачивания по адресу https://scipython.com/eg/bbk, содержит данные о длинах связей, постоянных колебаний и энергиях разрыва связи некоторых двухатомных молекул. Единственный лист имеет имя `'Diatomics'`. Столбец A содержит формулу молекулы, в первой строке – заголовок, во второй строке указаны имена столбцов. Кроме того, имеется нижний колонтитул из двух строк, как показано на рис. 9.2.

[124] https://pypi.org/project/xlrd/.



	A	B	C	D	E	F	G
1		Structural properties of some diatomic molecules					
2	**Molecule**	**Bond length /A**	**we /cm-1**	**wexe /cm-1**	**De /kJ.mol-1**		
3	I_2	2.666	214.5	0.614	224.1042237		
4	O_2	1.20752	1580.19	11.98	623.3408948		
5	Cl_2	1.987	559.7	2.67	350.8836826		
6	F_2	1.41193	916.64	11.236	223.640111		
7	N_2	1.09768	2358.57	14.324	1161.440719		
8	CO	1.128323	2169.81358	13.28831	1059.592595		
9	NO	1.15077	1904.20	14.075	770.4430432		
10	Data from the NIST Chemistry WebBook: Constants of Diatomic Molecules						
11	https://webbook.nist.gov						
12							
13							
14							
15							
16							
17							

Рис. 9.2. Лист Excel, содержащий данные о структурных свойствах некоторых двухатомных молекул

Приведенную ниже инструкцию можно использовать для считывания этих данных в объект `DataFrame`:

```
df = pd.read_excel('bond-lengths.xlsx',
        index_col=0,              # Первый столбец содержит метки индекса.
        skipfooter=2,             # Две последние строки листа отбрасываются.
        header=1,                 # Имена столбцов берутся из второй строки.
        usecols='A:E',            # Использовать столбцы Excel с метками A-E
        sheet_name='Diatomics'    # Чтение данных из указанного здесь листа.
        )
print(df)
```

```
          Bond length /A     we /cm-1  wexe /cm-1  De /kJ.mol -1
Molecule
I2              2.666000    214.50000     0.61400     224.104224
O2              1.207520   1580.19000    11.98000     623.340895
Cl2             1.987000    559.70000     2.67000     350.883683
F2              1.411930    916.64000    11.23600     223.640111
N2              1.097680   2358.57000    14.32400    1161.440719
CO              1.128323   2169.81358    13.28831    1059.592595
NO              1.150770   1904.20000    14.07500     770.443043
```

Если вам не повезло и приходится записывать данные в файл электронной таблицы Excel, то воспользуйтесь методом `to_excel`, как показано в примере П9.8. Здесь также может потребоваться разрешение зависимостей: если модуль `openpyxl` (https://pypi.org/project/openpyxl/) отсутствует, то его можно установить с помощью менеджера пакетов или командой `pip`:

```
pip install openpyxl
```

Пример П9.8. Для создания некоторых данных, которые необходимо записать в файл, программа в листинге 9.2 генерирует объект DataFrame со значениями высоты полета снаряда, выпущенного под тремя различными углами (значения углов – столбцы), как функции от времени (в строках):

Листинг 9.2. Высота полета снаряда как функция от времени

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# Ускорение свободного падения, м/с2.
g = 9.81

# Сетка времени, с.
t = np.linspace(0, 5, 500)
# Углы запуска снаряда, град.
theta0 = np.array([30, 45, 80])
# Начальная скорость запуска снаряда, м/с.
v0 = 20

def z(t, v0, theta0):
    """Return the height of the projectile at time t > 0."""
    """Возвращает высоту снаряда в момент времени t > 0."""
    return -g/2 * t**2 + v0*t*np.sin(theta0)

def x(t, v0, theta0):
    """Return the range of the projectile at time t > 0."""
    """Возвращает расстояние, пройденное снарядом, в момент времени t > 0."""
    return v0 * t * np.cos(theta0)

# Пустой DataFrame со столбцами для различных значений углов запуска.
df = pd.DataFrame(columns=theta0 , index=t)
# Заполнение объекта df значениями высоты полета снаряда как функции от времени.
for theta in theta0:
    df[theta] = z(t, v0, np.radians(theta))
# Когда снаряд падает на землю (z <= 0), установить данные о высоте как некорректные.
df[df <=0] = np.nan

# Создание рисунка Matplotlib с графиками траекторий полета снаряда.
fig, ax = plt.subplots()
for theta in theta0:
    ax.plot(x(t, v0, np.radians(theta)), df[theta], label=f'${theta}^\circ$')

# Максимальная высота, достигнутая снарядом, при каждом значении начального угла theta0.
heights = df.max()
print(heights)
# Определение ограничений по оси y с небольшим выравниванием по верхней границе;
# подписи к осям.
ax.set_ylim(0, heights.max()*1.05)
ax.set_xlabel('Range /m')
ax.set_ylabel('Height /m')
ax.legend()
plt.show()
```

На рис. 9.3 показаны графики траекторий полета снаряда, сгенерированные кодом из листинга 9.2.

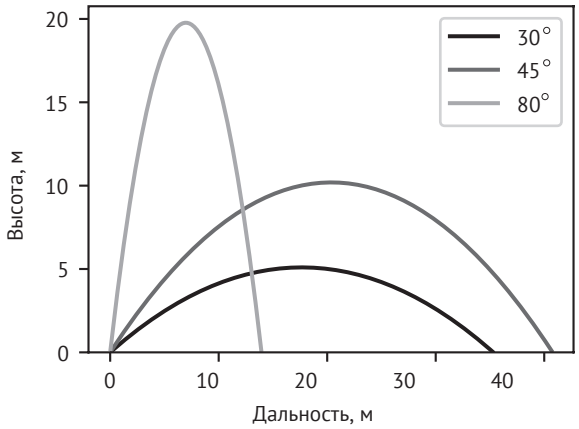


Рис. 9.3. Траектории полета снаряда, выпущенного с начальной скоростью v_0 = 20 м/с под тремя различными углами

Для сохранения содержимого объекта `DataFrame` `df` в файле Excel на одном листе необходимо воспользоваться методом `to_excel`:

```
df.to_excel('projectile.xlsx', sheet_name='Dependence on angle')
```

Чтобы записать данные в файл Excel с несколькими листами, необходимо создать объект `pd.ExcelWriter` и вызывать метод `to_excel` для каждого объекта pandas, который должен быть записан в файл:

```
with pd.ExcelWriter('projectile2.xlsx') as writer:
    for theta in theta0:
        # Сохраняются только корректные данные для каждой траектории.
        ser = df[theta].dropna()
        # Изменение индекса объекта Series на шкалу дальности вместо шкалы времени.
        ser.index = x(ser.index, v0, np.radians(theta))
        ser.to_excel(writer, sheet_name=f'{theta} deg')
```

9.2.4 Веб-скрейпинг

В библиотеке pandas метод `read_html` можно использовать для синтаксического разбора (парсинга) веб-страниц с целью извлечения данных, содержащихся в HTML-таблицах. Метод возвращает список объектов `DataFrame`, а значения аргументов, принятые по умолчанию для этого метода, позволяют достаточно эффективно выполнять требуемую работу на большинстве правильно оформленных веб-страниц. Наиболее часто используемые и полезные аргументы кратко описаны ниже.

- `io` – URL, путевое имя файла или файловый объект, из которого считывается код HTML для парсинга.

- `match` – необязательная строка для поиска в таблице: обрабатываются и возвращаются только таблицы, содержащие эту строку[125].
- `header` – индексная строка, которая будет использоваться для заголовков столбцов. По умолчанию `None` – используются заголовки ячеек в формате HTML `<th>`, если они присутствуют.
- `index_col` – столбцы (столбец), используемые как метки строк в объекте `DataFrame`.
- `attrs` – словарь атрибутов HTML для идентификации требуемой таблицы, например `attrs={'id': 'data-table'}`.
- `thousands` – символ-разделитель, используемый для разделения групп разрядов в больших числах. По умолчанию запятая `','`.
- `decimal` – символ, используемый для обозначения десятичной точки. По умолчанию точка `'.'`, как принято в США, Великобритании, Австралии, Японии, Китае и Южной Корее. В континентальных европейских странах чаще используется запятая `','`.
- `na_values` – строка (строки), используемая для обозначения нечисловых данных NaN, как для метода `read_csv`.

Пример П9.9. Во время написания этой книги первая таблица на странице «Википедии» https: //en.wikipedia.org/wiki/List_of_wine-producing_regions содержала столбцы рейтинга, названия страны и объема производства вина для основных стран-производителей вина в мире. Пример синтаксического разбора (парсинга) содержимого этой таблицы средствами библиотеки pandas:

```
In [x]: dfs = pd.read_html(
             'https://en.wikipedia.org/wiki/List_of_wine-producing_regions',
             index_col=1, match="Wine production by country")                    ❶
In [x]: dfs[0].head()                                                           ❷
Out[x]:
                                Rank     Production(tonnes)
Country(with link to wine article)
Italy                              1               4796900
France                             2               4607850
Spain                              3               4293466
United States                      4               3300000
China                              5               1700000
```

❶ В этом случае таблица идентифицируется по совпадению заданной строки с текстом в элементе `<caption>` первой таблицы `<table>` на этой странице.

❷ `dfs` – это список, содержащий один элемент – объект `DataFrame`, полученный при парсинге найденной таблицы.

---

[125] Значением match может быть регулярное выражение.

9.2.5 Упражнения

Задачи

39.2.1. Веб-страница https://scipython.com/eg/bab предоставляет таблицы с данными по общим объемам озона в октябре в единицах Добсона[126] и по двум компонентам хлорфторуглерода F11 и F12 в числе частей на триллион по объему (pptv) за период 1957–1984 гг. См. Farman et al., Nature 315, 207 (1985).

Прочитать и выполнить парсинг этих данных и изобразить их на наиболее подходящей диаграмме.

39.2.2. На странице https://en.wikipedia.org/wiki/Abundances_of_the_elements_(data_page) Википедия предоставляет список значений распространенности элементов на Солнце и в Солнечной системе в виде таблицы HTML (наряду с другими подобными данными). Использовать метод read_html из библиотеки pandas для чтения и парсинга данных Кайе и Лаби (Kaye and Laby) (столбец с заголовком «Y1») и создать столбиковую диаграмму (гистограмму), демонстрирующую правило Оддо–Гаркинса: элементы с четными атомными номерами более распространены, чем соседние с ними элементы с нечетными атомными номерами.

39.2.3. Диаграмма Герцшпрунга–Рассела (или Рессела) классифицирует звезды на точечной диаграмме: каждая звезда представлена точкой, координатой *x* которой является активная температура, а координатой *y* – светимость (звездная величина), мера электромагнитного излучения звезды. Страницу https://scipython.com/ex/bak можно использовать для получения версии базы данных HYG-database[127], в которой представлены данные о 119 614 звездах. Прочитать эти данные с помощью методов библиотеки pandas и создать диаграмму Герцшпрунга–Рассела. Столбец светимости идентифицируется как `'lum'` в заголовке, а температуру звезды можно вычислить по ее показателю цвета (также обозначаемому как (B–V) и определяемому по столбцу с меткой `'ci'`), воспользовавшись формулой Баллестероса:

$$T / K = 4600\left(\frac{1}{0.92(B-V)+1.7}+\frac{1}{0.92(B-V)+0.62}\right).$$

Следует отметить, что светимость лучше всего отображается на логарифмической шкале, а ось температуры обычно изображается в обратном направлении (с уменьшением температуры слева направо на диаграмме).

39.2.4. Transport for London (TfL) – это публично-правовая корпорация, являющаяся функциональным подразделением администрации Большого Лондона и отвечающая за управление транспортной системой Лондона, включая

[126] Единица Добсона (Dobson unit) равна слою чистого озона 10 мкм (0.01 мм) при стандартных давлении и температуре, образующегося из общего объема озона в атмосфере над какой-либо областью поверхности Земли.

[127] База данных https://github.com/astronexus/HYG-Database, опубликованная под защитой лицензии Creative Commons Attribution-ShareAlike.

координацию работы общественного транспорта, содержание главных дорог и светофоров. TfL открыто публикует документ в формате Excel, доступный для скачивания по адресу https://scipython.com/ex/bam, в котором содержатся статистические данные об использовании сети подземного транспорта (The Tube – лондонское метро) в форме количества входящих и выходящих пассажиров за «обычный» день на каждой станции за период 2007–2017 гг.

Прочитать этот документ средствами библиотеки pandas и проанализировать его, чтобы определить: а) самую загруженную станцию в обычный день в 2017 г., б) станцию с наибольшим приростом пассажиропотока за период 2007–2017 гг., в) станцию с наибольшей относительной разностью между количеством пассажиров в рабочие дни и в обычное воскресенье в 2017 г.

39.2.5. База данных HITRAN (https://hitran.org) предоставляет список значений интенсивности молекулярных линий для моделирования лучистого теплообмена в атмосферах планет (астрофизика молекулярных линий). Собственный формат этой базы данных представлен в виде записей длиной 160 символов с фиксированной шириной полей.

Использовать pandas для чтения файла *CO2-transitions.par*, доступного для скачивания здесь: https://scipython.com/ex/ban (в этом файле также можно найти описания всех его полей). Построить график зависимости интенсивности молекулярных линий от длины волны при лучистом теплообмене в инфракрасной области спектра (λ = от 10 мм до 700 нм, соответствующая волновым числам $\tilde{\nu}$ = от 1 1/см до приблизительно 14 000 1/см), где диоксид углерода (CO_2) отвечает в значительной степени за возникновение парникового эффекта в атмосфере Земли.

9.3 Более сложное индексирование

9.3.1 Иерархические индексы с использованием MultiIndex

Внутренняя структура объекта DataFrame в действительности представляет собой двумерный массив данных: для представления данных в более высоких измерениях чаще всего используется иерархическое индексирование для организации нескольких уровней в одном индексе. Если данные неполные («разреженные») или разнородные, то такой способ более эффективен, чем создание многомерного массива NumPy. Например, рассмотрим набор данных о среднемесячной температуре и осадках в пяти городах Европы. Этот набор можно рассматривать как трехмерный с измерениями («осями») «город» (city), «месяц» (month) и «тип данных» (data type) (последнее измерение означает температуру или осадки). Для пяти городов (Париж, Берлин, Вена, Лондон, Мадрид) и четырех месяцев (январь, апрель, июль, октябрь) необходимо рассматривать в общей сложности 40 точек данных.

Можно было бы создать уже знакомый привычный одноуровневый индекс, состоящий из кортежей `(city, month)`, но такой подход не очень удобен и не обеспечивает гибкость. Вместо этого можно создать иерархический индекс с двумя уровнями из последовательности кортежей с двумя элементами, используя для этого метод `pd.MultiIndex.from_tuples`:

```
In [x]: cities = ('Paris', 'Berlin', 'Vienna', 'London', 'Madrid')
In [x]: months = ('Jan', 'Apr', 'Jul', 'Oct')
In [x]: index = pd.MultiIndex.from_tuples(
...:           (city, month) for city in cities for month in months)
In [x]: index
Out[x]:
MultiIndex([( 'Paris', 'Jan'),
            ( 'Paris', 'Apr'),
            ( 'Paris', 'Jul'),
            ( 'Paris', 'Oct'),
            ('Berlin', 'Jan'),
            ...
            ('Madrid', 'Jul'),
            ('Madrid', 'Oct')],
          )
```

Многомерные индексы `MultiIndex` в этой форме (декартово, или прямое, произведение двух и более последовательностей (множеств)) применяются настолько часто, что существует более удобная функция `from_product` для такой операции создания иерархического индекса:

```
index = pd.MultiIndex.from_product((cities, months))
```

Можно создать объект `DataFrame` с этим индексом, присвоив ему массив данных формы `(20, 2)`:

```
In [x]: index.names = ['City', 'Month']

In [x]: # Среднемесячная температура (град С) в каждом городе по месяцам: январь, апрель, июль,
       # октябрь.
In [x]: temps = [[4.9, 11.5, 20.5, 13.0], [0.1, 9.0, 19.1, 9.4],
...:             [0.3, 10.7, 20.8, 10.2], [5.2, 9.9, 18.7, 12.0],
...:             [6.3, 12.9, 25.6, 15.1]
...:            ]
In [x]: # Среднемесячное количество осадков (мм) в каждом городе по месяцам: январь,
        # апрель, июль, октябрь.
In [x]: rainfall = [[51.0, 51.8, 62.3, 61.5], [37.2, 33.7, 52.5, 32.2],
...:                [38., 45., 70., 38.], [55.2, 43.7, 44.5, 68.5],
...:                [33., 45., 12., 60.]
...:               ]

In [x]: arr = np.array((temps, rainfall)).reshape((2, 20)).T
In [x]: df = pd.DataFrame(arr, index=index, columns=['Mean temperature /degC',
...:                                                 'Mean rainfall /mm'])

In [x]: df
Out[x]:
               Mean temperature /degC    Mean rainfall /mm
City   Month
Paris  Jan                      4.9                 51.0
       Apr                     11.5                 51.8
       Jul                     20.5                 62.3
       Oct                     13.0                 61.5
```

```
Berlin Jan                    0.1                37.2
       Apr                    9.0                33.7
       Jul                   19.1                52.5
       Oct                    9.4                32.2
Vienna Jan                    0.3                38.0
       Apr                   10.7                45.0
       Jul                   20.8                70.0
       Oct                   10.2                38.0
London Jan                    5.2                55.2
       Apr                    9.9                43.7
       Jul                   18.7                44.5
       Oct                   12.0                68.5
Madrid Jan                    6.3                33.0
       Apr                   12.9                45.0
       Jul                   25.6                12.0
       Oct                   15.1                60.0
```

Метод `loc` можно использовать для создания иерархического индекса `MultiIndex` в объекте `DataFrame`:

```
In [x]: df.loc['Vienna']
Out[x]:
       Mean temperature /degC   Mean rainfall /mm
Month
Jan                       0.3                38.0
Apr                      10.7                45.0
Jul                      20.8                70.0
Oct                      10.2                38.0

In [x]: df.loc[('Paris', 'Jul')]
Out[x]:
Mean temperature /degC    20.5
Mean rainfall /mm         62.3
Name: (Paris , Jul), dtype: float64

In [x]: df.loc[('Paris', 'Jul'), 'Mean rainfall /mm']
Out[x]: 62.3
```

Но для выполнения операции вырезания из `MultiIndex` сначала необходимо его отсортировать:

```
In [x]: df['Berlin':'London']
Out[x]: ...
UnsortedIndexError: 'Key length (1) was greater than MultiIndex lexsort depth (0)'
```

Это в некоторой степени непонятное сообщение является результатом работы механизма, с помощью которого pandas оптимизирует операции вырезания только для индексов, упорядоченных лексикографически. Существует несколько методов сортировки `MultiIndex`, но самым простым способом является использование метода `sort_index`, который мы уже применяли ранее:

```
In [x]: df.sort_index(inplace=True)
In [x]: df['Berlin':'London']
Out[x]:
              Mean temperature /degC   Mean rainfall /mm
City   Month
Berlin Apr                      9.0                33.7
       Jan                      0.1                37.2
       Jul                     19.1                52.5
       Oct                      9.4                32.2
London Apr                      9.9                43.7
       Jan                      5.2                55.2
       Jul                     18.7                44.5
       Oct                     12.0                68.5
```

Следует отметить, что здесь сортировка выполнена не только по городам, но и по месяцам в алфавитном порядке. Один из способов сохранения вывода в хронологическом порядке – присваивание месяцам номеров вместо изменения меток индекса:

```
In [x]: df2 = df.rename({'Jan': 1, 'Apr': 4, 'Jul': 7, 'Oct': 10})
In [x]: df2.sort_index(inplace=True)
In [x]: df2.loc['Vienna', 'Mean temperature /degC']
Out[x]:
Month
1      0.3
4     10.7
7     20.8
10    10.2
Name: Mean temperature /degC, dtype: float64
```

Удобная функция `xs` позволяет более простым способом выбирать данные, проиндексированные на различных уровнях `MultiIndex`, и не требует предварительной сортировки индекса. Например, для извлечения данных о погоде в январе во всех городах:

```
In [x]: df.xs('Jan', level=1)      # Поиск на втором уровне MultiIndex по образцу 'Jan'.
Out[x]:
        Mean temperature /degC   Mean rainfall /mm
City
Berlin                     0.1                37.2
London                     5.2                55.2
Madrid                     6.3                33.0
Paris                      4.9                51.0
Vienna                     0.3                38.0
```

Для объектов `Series` и `DataFrame` с иерархическим индексом строк можно изменить форму так, чтобы создавался `MultiIndex` по столбцам, используя для этого функцию `unstack()`:

```
In [x]: df.unstack()
Out[x]:
       Mean temperature /degC              ... Mean rainfall /mm
Month                      Apr   Jan   Jul  ...               Jan   Jul   Oct
City                                        ...
Berlin                     9.0   0.1  19.1  ...              37.2  52.5  32.2
London                     9.9   5.2  18.7  ...              55.2  44.5  68.5
Madrid                    12.9   6.3  25.6  ...              33.0  12.0  60.0
Paris                     11.5   4.9  20.5  ...              51.0  62.3  61.5
Vienna                    10.7   0.3  20.8  ...              38.0  70.0  38.0

[5 rows x 8 columns]
```

9.3.2 Метки времени и временны́е ряды

В библиотеке pandas имеется объект `Timestamp`, представляющий некоторый момент времени с некоторой точностью. Метод `to_datetime` предоставляет мощный и гибкий способ парсинга обычной строки, предназначенной для чтения человеком, с преобразованием ее в объект `Timestamp`. В приведенных ниже примерах все вычисления меток времени выполняются для полуночи 12 марта 2020 г.: `Timestamp('2020-03-12 00:00:00')`. Следует отметить, что поскольку эта дата неоднозначна, по умолчанию она интерпретируется в формате, принятом в США: ММ/ДД/ГГ. Чтобы строка интерпретировалась в формате ДД/ММ/ГГ, необходимо установить параметр `dayfirst=True`.

```
pd.to_datetime('2020-03-12')
pd.to_datetime('12/3/20', dayfirst=True)
pd.to_datetime('3/12/20')
pd.to_datetime('12 March , 2020')
pd.to_datetime('12th of March 2020')
pd.to_datetime('Mar 12, 2020')
```

Значения времени также обрабатываются с соблюдением корректности:

```
In [x]: pd.to_datetime('9:05 21 August 2017')
Out[x]: Timestamp('2017-08-21 09:05:00')
In [x]: pd.to_datetime('21 August 2017 09:05:23')
Out[x]: Timestamp('2017-08-21 09:05:23')
```

Индексы можно формировать как диапазон меток времени `Timestamp` с равномерно распределенными интервалами при помощи функции `date_range`. Диапазоны можно определять, передавая начальную и конечную даты или начальную дату и количество периодов (интервалов). По умолчанию интервал диапазона равен одному дню, но это значение управляется аргументом `freq` (его допустимые значения см. в табл. 9.1).

Таблица 9.1. Некоторые строковые коды для обозначения интервалов (частоты и смещения) времени в библиотеке pandas

Код	Описание
A	Конец года
M	Конец месяца
W	Неделя
D	Календарный день
H	Час
T	Минута
S	Секунда
L	Миллисекунда (мс)
U	Микросекунда (мкс)

```
In [x]: pd.date_range('1997-03-12', '1997-03-15')
Out[x]: DatetimeIndex(['1997-03-12', '1997-03-13', '1997-03-14', '1997-03-15'],
              dtype='datetime64[ns]', freq='D')

In [x]: pd.date_range('1997-03-12', periods=4)
Out[x]: DatetimeIndex(['1997-03-12', '1997-03-13', '1997-03-14', '1997-03-15'],
              dtype='datetime64[ns]', freq='D')

In [x]: pd.date_range('1997-03', periods=4, freq='M')                    ❶
Out[x]: DatetimeIndex(['1997-03-31', '1997-04-30', '1997-05-31', '1997-06-30'],
              dtype='datetime64[ns]', freq='M')

In [x]: pd.date_range('1997-03', periods=4, freq='MS')                   ❷
Out[x]: DatetimeIndex(['1997-03-01', '1997-04-01', '1997-05-01', '1997-06-01'],
              dtype='datetime64[ns]', freq='MS')
```

❶ По умолчанию диапазоны по месяцам, определенные значением `freq='M'`, маркируются по концу месяца. То же самое относится и к диапазонам по годам (`freq='A'`).

❷ Для установки меток времени по началу каждого месяца используется значение `freq='MS'` (для диапазонов по годам `freq='AS'` – по началу года).

В библиотеке pandas определены различия между меткой времени (представленной объектом `Timestamp`) и интервалом времени, т. е. временем, прошедшим между некоторыми двумя моментами. Интервал времени представлен объектом `Period`, а его начальная и конечная точки (моменты времени) доступны через атрибуты `start_time` и `end_time`. Синтаксис создания интервалов времени аналогичен синтаксису создания диапазонов дат:

```
In [x]: p = pd.Period('2020-04', freq='M')
In [x]: t = pd.Timestamp('2020-04-03 14:30')
In [x]: p.start_time < t < p.end_time
Out[x]: True
```

Часто возникает необходимость в реорганизации временны́х рядов с измененной (более высокой или низкой) частотой. Метод `resample` позволяет это сделать: он возвращает объект `Resampler`, который можно использовать для агрегации данных некоторым наиболее подходящим способом. Например, при субдискретизации (реорганизации данных с более широким интервалом времени), возможно, лучше всего подходит вычисление среднего значения, минимума, максимума или суммы значений в преобразуемом диапазоне. Пример П9.10 поможет лучше понять эту операцию.

Пример П9.10. Файл *river-level.csv*, доступный для скачивания по адресу https://scipython.com/eg/bal, содержит список значений высоты в м над уровнем моря Читтерн-Брук, небольшой реки в графстве Уилтшир, Англия. Приведены минимальное, среднее и максимальное значения высоты в каждый день от 1 января 2014 г. до 31 декабря 2016 г.

Приведенный ниже код считывает эти данные и создает график ежедневных значений высоты уровня реки с учетом ежемесячных средних, минимальных и максимальных значений.

```
import pandas as pd
import matplotlib.pyplot as plt

df = pd.read_csv('river -level.csv', index_col=0, comment='#', parse_dates=True)        ❶

rs_monthly = df.resample('M')

df['avg_level'].plot(label='Daily average')
rs_monthly['avg_level'].mean().plot(label='Monthly average')
rs_monthly['min_level'].min().plot(label='Monthly minimum')
rs_monthly['max_level'].max().plot(label='Monthly maximum')

plt.xlabel('Date')
plt.ylabel('River level /m')
plt.gca().legend()
plt.show()
```

❶ Обратите внимание: необходимо установить значение `parse_dates=True`, чтобы библиотека pandas правильно интерпретировала первый столбец как `DatetimeIndex`.

На рис. 9.4 показан полученный график.

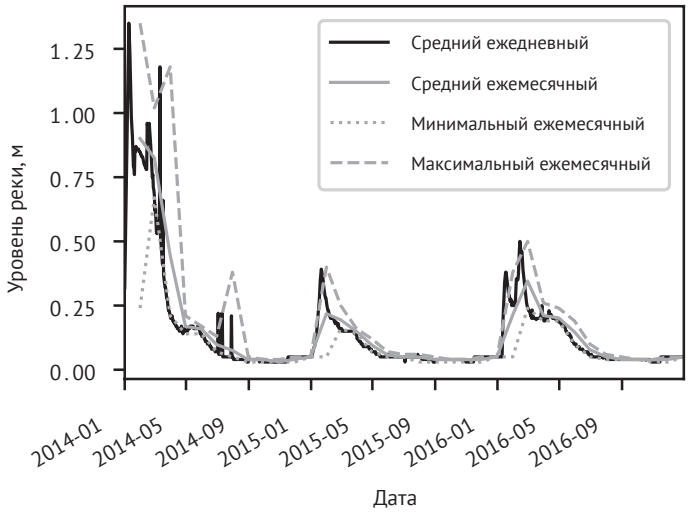


Рис. 9.4. Уровень реки Читтерн-Брук в м за период 2014–2016 гг.

9.3.3 Упражнения

Задачи

39.3.1. Использовать pandas для считывания файла *tb-cases.txt*, доступного для скачивания по адресу https://scipython.com/ex/bao, в котором представлено число выявленных случаев туберкулеза в США по отдельным штатам за 1993–2018 гг. Создать объект `DataFrame` с иерархическим индексом (`MultiIndex`), состоящим из названия штата и года. Отобразить эти данные на наиболее подходящем графике и определить штат с самым высоким относительным ростом случаев заболевания туберкулезом за рассматриваемый период.

39.3.2. Численность населения каждого штата США за период 1993–2018 гг. содержится в файле *US-populations.txt*, доступном для скачивания здесь: https://scipython.com/ex/bap. Прочитать эти данные в объект `DataFrame` с созданием соответствующего индекса и проанализировать их с целью обнаружения любых интересных трендов. Затем объединить эти данные с данными из задачи 39.3.1, чтобы определить штаты с наибольшим и наименьшим преобладанием случаев заболевания туберкулезом на душу населения в 2018 г.

9.4 Очистка и обследование данных

В ходе любых научных исследований, особенно экспериментальных исследований, генерируются наборы данных, содержащие некорректные или отсутствующие по каким-либо причинам значения. Точки данных могут быть потеряны или могут выйти за пределы допустимого диапазона измерительного инструмента. Кроме того, точки данных могут быть зафиксированы

неправильно или могут быть получены не полностью из различных источников. Библиотека pandas предоставляет разнообразные методы для обработки подобных отсутствующих значений, включая функции для удаления или замены некорректных точек данных на средние значения или значения по умолчанию.

Эта книга не пытается предоставить полное руководство по методам научных исследований, но читатель должен знать о методиках обработки отсутствующих или некорректных данных с целью проведения последующего анализа так, чтобы прийти к верному ряду заключений.

9.4.1 Отсутствующие значения

По умолчанию контрольное значение (сигнальная метка) обозначает отсутствующие данные как NaN. В разделе 9.1.2 уже использовались методы `isnull()` и `notnull()` для проверки наличия или отсутствия подобных значений, а также метод `dropna()`, который возвращает новый объект `DataFrame` со строками, содержащими только ненулевые данные:

```
In [x]: df = pd.DataFrame([[   1.1, np.nan, np.nan,   10.3],
...:                       [   0.8, np.nan,    3.6,    2.9],
...:                       [   1.2,    2.5,    1.6,    2.7],
...:                       [np.nan, np.nan, np.nan, np.nan],
...:                       [np.nan, np.nan,    3.6,   5.3]],
...:                      columns=list('ABCD'))
In [x]: df
     A    B    C     D
0  1.1  NaN  NaN  10.3
1  0.8  NaN  3.6   2.9
2  1.2  2.5  1.6   2.7
3  NaN  NaN  NaN   NaN
4  NaN  NaN  3.6   5.3

In [x]: df.dropna()
Out[x]:
     A    B    C    D
2  1.2  2.5  1.6  2.7
```

Иногда требуется отбросить только строки (или столбцы), состоящие полностью из значений NaN. В этом случае передается аргумент `how='all'` вместо используемого по умолчанию `how='any'`:

```
In [x]: df.dropna(how='all')
Out[x]:
     A    B    C     D
0  1.1  NaN  NaN  10.3
1  0.8  NaN  3.6   2.9
2  1.2  2.5  1.6   2.7
4  NaN  NaN  3.6   5.3
```

Также можно определить пороговое значение количества NaN для выполнения операции отбрасывания столбца или строки:

```
In [x]: df.dropna(thresh=3, axis=1)    # Отбрасываются только столбцы с тремя и более NaN.
Out[x]:
     A    C     D
0  1.1  NaN  10.3
1  0.8  3.6   2.9
2  1.2  1.6   2.7
3  NaN  NaN   NaN
4  NaN  3.6   5.3
```

Другим способом исключения значений NaN является их замена на корректные данные, соответствующие некоторому процессу. Эту задачу выполняет метод `fillna()`. Наиболее часто используемые варианты применения этого метода рассматриваются в следующих примерах.

Замена всех значений NaN на одно значение:

```
In [x]: df.fillna(3.6)
Out[x]:
     A    B    C     D
0  1.1  3.6  3.6  10.3
1  0.8  3.6  3.6   2.9
2  1.2  2.5  1.6   2.7
3  3.6  3.6  3.6   3.6
4  3.6  3.6  3.6   5.3
```

Замена значений NaN на самое последнее обнаруженное корректное значение вниз по столбцам («заполнение в прямом направлении»):

```
In [x]: df.fillna(method='ffill')
Out[x]:
     A    B    C     D
0  1.1  NaN  NaN  10.3
1  0.8  NaN  3.6   2.9
2  1.2  2.5  1.6   2.7
3  1.2  2.5  1.6   2.7
4  1.2  2.5  3.6   5.3
```

Замена значений NaN на самое последнее обнаруженное корректное значение по строкам:

```
In [x]: df.fillna(method='ffill', axis=1)
Out[x]:
     A    B    C     D
0  1.1  1.1  1.1  10.3
1  0.8  0.8  3.6   2.9
2  1.2  2.5  1.6   2.7
3  NaN  NaN  NaN   NaN
4  NaN  NaN  3.6   5.3
```

Передача словаря столбцов или индекса имен позволяет получить большую степень управления заполняемыми значениями. После этого цепочные вызовы могут предоставить мощный и гибкий способ очистки данных. Например, для замены отсутствующих данных в столбцах `A` и `C` средними значениями по этим столбцам:

```
In [x]: df.fillna({'A': df['A'].mean(), 'C': df['C'].mean()})
Out[x]:
          A    B         C     D
0  1.100000  NaN  2.933333  10.3
1  0.800000  NaN  3.600000   2.9
2  1.200000  2.5  1.600000   2.7
3  1.033333  NaN  2.933333   NaN
4  1.033333  NaN  3.600000   5.3
```

Еще один пример:

```
In [x]: df.fillna({'A': df['A'].mean(), 'B': df['B'].mean()}).fillna(0)
Out[x]:
          A    B    C     D
0  1.100000  2.5  0.0  10.3
1  0.800000  2.5  3.6   2.9
2  1.200000  2.5  1.6   2.7
3  1.033333  2.5  0.0   0.0
4  1.033333  2.5  3.6   5.3
```

Может оказаться, что в наборе обрабатываемых данных используется другое контрольное значение (сигнальная метка) для обозначения некорректных данных, например `-1` или `-99`. Метод `replace` способен правильно обработать и такие данные:

```
In [x]: ser = pd.Series([1.2, 3.5, -99, -99, 4.0, -99, -0.5])
In [x]: ser.replace(-99, np.nan)
Out[x]:
0    1.2
1    3.5
2    NaN
3    NaN
4    4.0
5    NaN
6   -0.5
dtype: float64
```

Метод `replace` также принимает словарь, отображающий некорректные значения в заменяющие их значения:

```
In [x]: ser.replace({-99: 0, -0.5: np.nan})
Out[x]:
0    1.2
1    3.5
2    0.0
3    0.0
4    4.0
5    0.0
6    NaN
dtype: float64
```

9.4.2 Повторяющиеся значения

Метод объекта `DataFrame` `duplicated()` возвращает объект `Series`, содержащий логические значения, определяющие, повторяет ли каждая строка предыдущую строку, а метод `drop_duplicates()` исключает такие строки. По умолчанию оба метода обрабатывают все столбцы, а для удаления строк с повторяющимися элементами в одном столбце или в нескольких столбцах передается имя столбца или явно заданная последовательность имен столбцов. Следующий аргумент `keep` определяет, сохраняется первая обнаруженная строка (`'first'`, значение по умолчанию) или последняя обнаруженная строка (`'last'`).

```
In [x]: df = pd.DataFrame([['Lithium', 'Li', 3, 6, 0.0759],
...:                       ['Lithium', 'Li', 3, 7, 0.9241],
...:                       ['Sodium', 'Na', 11, 23, 1],
...:                       ['Potassium', 'K', 19, 39, 0.932581],
...:                       ['Potassium', 'K', 19, 40, 1.17e-4],
...:                       ['Potassium', 'K', 19, 41, 0.067302]],
...:                      columns=['Element', 'Symbol', 'Z', 'A', 'Abundance'])
In [x]: df
Out[x]:
     Element Symbol   Z   A  Abundance
0    Lithium     Li   3   6   0.075900
1    Lithium     Li   3   7   0.924100
2     Sodium     Na  11  23   1.000000
3  Potassium      K  19  39   0.932581
4  Potassium      K  19  40   0.000117
5  Potassium      K  19  41   0.067302

In [x]: df.drop_duplicates(['Symbol'])
Out[x]:
     Element Symbol   Z   A  Abundance
0    Lithium     Li   3   6   0.075900
2     Sodium     Na  11  23   1.000000
3  Potassium      K  19  39   0.932581

In [x]: df.drop_duplicates(['Symbol', 'Z'], keep='last')
Out[x]:
     Element Symbol   Z   A  Abundance
1    Lithium     Li   3   7   0.924100
2     Sodium     Na  11  23   1.000000
5  Potassium      K  19  41   0.067302
```

9.4.3 Статистическая группировка данных

Часто возникает необходимость в статистической группировке (биннинге) больших объемов данных с целью сокращения до разумного управляемого размера или для категоризации на основе значений. В библиотеке pandas функцию `cut` можно использовать для выполнения такой задачи способом, похожим на действие функции `histogram` из библиотеки NumPy (см. раздел 6.3.3):

```
In [x]: marks = [67, 80, 34, 55, 77, 66, 59, 52, 70, 67, 58, 63, 49, 72]
In [x]: bins = [0, 40, 60, 70, 80, 100]
In [x]: dist = pd.cut(marks, bins)
In [x]: dist
```

```
[(60, 70], (70, 80], (0, 40], (40, 60], ..., (60, 70], (40, 60], (70, 80]]
Length: 14
Categories (5, interval): [(0, 40] < (40, 60] < (60, 70] < (70, 80] < (80, 100]]
```

Каждая оценка (из списка `marks`) помещена в свою группу с границами, определенными последовательностью значений `bins`. Количество значений в каждой группе возвращает метод `value_counts`:

```
In [x]: pd.value_counts(dist)
Out[x]:
(60, 70]     5
(40, 60]     5
(70, 80]     3
(0, 40]      1
(80, 100]    0
dtype: int64
```

По умолчанию правая граница каждого интервала закрыта (значения, равные этой границе, включаются в группу; это обозначено символом `']'`), а левая граница открыта (обозначена символом `'('`). Граничные условия можно поменять местами, установив значение аргумента `right=False`:

```
In [x]: pd.value_counts(pd.cut(marks, bins, right=False))
Out[x]:
[40, 60)     5
[60, 70)     4
[70, 80)     3
[80, 100)    1
[0, 40)      1
dtype: int64
```

Кроме того, для групп можно определять имена, передавая последовательность строк в аргументе `labels`:

```
In [x]: dist = pd.cut(marks, bins, labels=list(reversed('ABCDE')), right=False)
In [x]: dist
[C, A, E, D, B, ..., C, D, C, D, B]
Length: 14
Categories (5, object): [E < D < C < B < A]

In [x]: pd.value_counts(dist)
Out[x]:
D    5
C    4
B    3
A    1
E    1
dtype: int64
```

Следует отметить, что категории не обязательно должны иметь какой-то определенный порядок. Чтобы расположить счетчики оценок в возрастающем порядке, можно просто отсортировать индекс `Series`:

```
In [x]: pd.value_counts(dist).sort_index(ascending=False)
Out[x]:
A    1
B    3
C    4
D    5
E    1
dtype: int64
```

9.4.4 Обработка промахов

Обнаружение и фильтрация промахов (выбросов), как и обработка отсутствующих значений или некорректных данных, – процесс, требующий осторожного и внимательного отношения, поэтому должен выполняться с учетом предварительных предположений об ожидаемом внутреннем распределении значений данных. Но часто ожидаются и значения-промахи (выбросы) из-за критических сбоев измерительного оборудования или ПО («залипших» пикселов, космических лучей и т. п.), явных ошибок или общеизвестных исключительных случаев.

Например, рассмотрим имитацию поселка, в котором 200 домов, цены которых соответствуют нормальному распределению (μ = 250 000 долл., σ = 55 000 долл), за исключением пары владений, стоимость которых в несколько раз выше средней стоимости дома:

```
In [x]: nhouses = 200
In [x]: mu, sigma = 250, 55       # Среднее значение, стандартное отклонение в тыс. долл.
In [x]: prices = np.clip(np.random.randn(nhouses)*sigma + mu, 0, None).astype(int)      ❶
In [x]: prices[-2] = 1.e3
In [x]: prices[-1] = 2.e3
In [x]: df = pd.DataFrame(prices, columns=['price, $1000s'])
In [x]: df.tail()
Out[x]:
     price, $1000s
195            247
196            218
197            236
198           1000
199           2000
```

❶ Метод `np.clip(arr, min, max)` ограничивает значения массива `arr` границами min и max. Здесь это сделано для исключения отрицательных цен домов, генерируемых в случайной выборке. Это уже сам по себе один из типов фильтрации промахов (выбросов).

Эти промахи (высокая цена двух владений) искажают среднее значение и (особенно) стандартное отклонение распределения цен на дома:

```
In [x]: df.median()      # Срединное значение (медиана) - надежная мера среднего значения.
Out[x]:
price, $1000s    247.8
dtype: float64

In [x]: df.mean()        # На среднее значение в большей степени влияют промахи.
Out[x]:
```

```
price, $1000s    258.775
dtype: float64

In [x]: df.std()         # На стандартное отклонение промахи влияют особенно сильно.
Out[x]:
price, $1000s    145.796907
dtype: float64
```

Вероятнее всего, нам более интересен анализ цен на «обычные» дома в поселке без учета дорогих владений. Один из способов сделать это – определение этих двух владений как отклонений от средней цены домов, например на три значения стандартного отклонения, и установка вместо их стоимости специального значения NaN:

```
In [x]: df[df > 3*df.std()+df.mean()] = np.nan
In [x]: df.tail()
     price, $1000s
195          247.0
196          218.0
197          236.0
198            NaN
199            NaN
```

Теперь найдем значения, более близкие к распределению цен на дома (без учета дорогих владений):

```
In [x]: df.mean()
Out[x]:
price, $1000s    246.237374
dtype: float64

In [x]: df.std()
Out[x]:
price, $1000s    55.995279
dtype: float64
```

Пример П9.11. Знаменитые эксперименты Роберта Милликена (Robert Millikan) с заряженными каплями масла были проведены в Чикагском университете в 1910 г. (по другим данным Милликен начал эти эксперименты еще в 1908 г.) для определения величины электрического заряда электрона[128]. В одном эксперименте наблюдалось падение электрически заряженной капли масла, проходящей известное расстояние *d* между двумя незаряженными пластинами с предельной скоростью v_g: при известном времени падения t_g можно логически определить радиус капли *a*. Далее на пластины подавалось напряжение, создающее между ними электрическое поле. Поскольку капля поднималась под воздействием равнодействующей всех приложенных сил, измеренное время t_e, затраченное на возвращение капли с прохождением того же расстояния *d*, можно использовать для логического вывода ее полного заряда *q*, который был экспериментально определен как целое число, умноженное на некоторое базовое значение *e*, т. е. $q = Ne$.

[128] В мае 2019 г. величина заряда электрона была определена более точно и установлена равной $1.602176634 \times 10^{-19}$ Кл.

Для первой части эксперимента со свободным падением капли[129], т. е. капля падает с постоянной предельной скоростью $v_g = -d/t_g$ и на нее не действует равнодействующая сил, сумма силы тяжести и силы сопротивления воздуха равна нулю:

$$Fg + Fd = 0 \Rightarrow -m'g - 6\pi\eta a v_g = 0,$$

где $m' = 4/3\pi a^3\rho' = 4/3\pi a^3(\rho_{oil} - \rho_{air})$ – эффективная масса капли (с учетом массы вытесняемого ей воздуха), $g = 9.803$ м/с – ускорение свободного падения в Чикаго, $\eta = 1.859 \times 10^{-5}$ кг/м·с – вязкость воздуха в условиях эксперимента (температура, влажность и т. д.). После необходимых преобразований получим следующую формулу для вычисления радиуса капли:

$$a = \sqrt{(-9\eta v_g/2\rho'g)}.$$

Когда на пластины подается правильно подобранное напряжение и капля перемещается вверх с постоянной скоростью $v_e = d/t_e$, сила электрического поля уравновешивает силу тяжести и силу сопротивления воздуха (при этом новом значении скорости):

$$F_e + F_g + F'_d = 0 \Rightarrow qE + 6\pi\eta a v_g - 6\pi\eta a v_e = 0 \Rightarrow$$
$$\Rightarrow q = 6\pi\eta a(v_e - v_g)/E = 6\pi\eta ad/E\,(1/t_g + 1/t_e).$$

Каждая капля (капли помечались буквами A–H) наблюдалась в трех экспериментах для каждого различного значения заряда q и подвергалась облучению рентгеновскими лучами (до семи экспериментов для каждой капли).

В данных, которые можно скачать по адресу https://scipython.com/eg/bam, представлены измерения времени для ряда описанных выше экспериментов, выполненных с маслом, плотность которого $\rho_{oil} = 917.3$ кг/м^3, в день, когда плотность воздуха равнялась $\rho_{air} = 1.17$ кг/м^3. Величина электрического поля была установлена равной $E = 322.1$ кН/Кл, а расстояние, проходимое каплей, $d = 11.09$ мм. Можно использовать эти данные для приблизительной оценки значения e (предполагая, что это значение не является фиксированным по определению), как показано ниже.

```
drop expt    tg        te        tg        te        tg        te
A       1   13.102    46.822    12.941    46.896    13.086    46.681
A       2   12.938    86.767    13.032    86.952    13.086    86.746
A       3   13.023    61.082    12.958    60.826    12.998    60.860
A       4   12.943    86.747    12.922    86.840    13.054    86.899
B       1   11.434    56.305    11.350    56.097    11.246    56.282
B       2   11.402    75.823    11.584    75.819    11.487    76.063
B       3   11.591    44.717    11.397    44.851    11.364    44.776
B       4   11.443    75.905    11.368    75.975    11.457    76.041
B       5   11.434    75.939    11.414    75.880    11.444    75.929
```

[129] В этом примере принимается система координат, в которой положение капли по вертикали z увеличивается по направлению «вверх».

```
B       6   11.559    75.892    11.414    75.924    11.292    75.985
B       7   11.394    44.716    11.589    44.753    11.401    44.794
C       1   16.197   100.458    16.010   100.486    16.329   100.461
C       2   16.241    47.727    16.106    47.714    16.177    47.625
C       3   16.133    37.879    16.267    37.746    16.203    37.709
C       4   16.170    64.765    16.136    64.649    16.229    64.508
D       1   16.176    38.017    16.127    37.910    16.282    38.020
D       2   16.275    38.280    16.092    38.208    16.133    38.092
D       3   16.422    48.327    16.073    48.284    16.212    48.184
D       4   16.134    38.202    16.258    38.270    16.105    38.229
D       5   16.164   102.562    16.217   102.673    16.194   102.696
E       1   12.275    55.020    12.116    54.962    12.307    54.978
E       2   12.157    54.772    12.183    54.967    12.046    55.219
E       3   12.146    55.004    12.118    54.938    12.346    54.869
E       4   12.319    43.635    12.243    43.552    12.073    43.582
F       1   14.172    61.946    14.174    61.970    14.069    61.959
F       2   14.145    90.718    13.955    90.707    14.075    90.866
F       3   14.070    62.147    14.074    61.961    14.247    61.892
F       4   14.017    61.968    14.101    61.921    14.106    62.174
G       1    9.723    50.375     9.527    50.482     9.502    50.508
G       2    9.463    63.755     9.670    63.853     9.509    63.827
G       3    9.448    63.804     9.407    63.899     9.563    63.768
G       4    9.327    63.855     9.518    63.967     9.533    63.824
H       1   13.192    73.375    13.167    73.338    13.316    73.449
H       2   13.042    42.642    13.387    42.428    13.334    42.459
H       3   13.389    42.379    13.244    42.373    13.055    42.610
H       4   13.114    73.161    13.226    73.384    13.207    73.257
H       5   13.030    73.295    13.022    73.419    13.438    73.512
```

Сначала определим необходимые параметры:

```
eta = 1.859e-5                  # Вязкость воздуха, кг/м.с.
rho_air = 1.17                  # Плотность воздуха, кг/м3.
rho_oil = 917.3                 # Плотность масла, кг/м3.
rhop = rho_oil - rho_air
g = 9.803                       # Ускорение свободного падения, м/с2
d = 11.09e-3                    # Расстояние подъема/падения, м.
E = -322.1e3                    # Вектор электрического поля (направлен вниз).
```

Далее данные считываются с присваиванием первых двух столбцов объекту `MultiIndex`:

```
In [x]: import pandas as pd
In [x]: df = pd.read_csv('eg10-millikan-data.txt', delim_whitespace=True,
                         index_col=[0, 1])
In [x]: df.head()
Out[x]:
               tg        te      tg.1      te.1      tg.2      te.2
drop expt
A    1     13.102    46.822    12.941    46.896    13.086    46.681
     2     12.938    86.767    13.032    86.952    13.086    86.746
     3     13.023    61.082    12.958    60.826    12.998    60.860
     4     12.943    86.747    12.922    86.840    13.054    86.899
B    1     11.434    56.305    11.350    56.097    11.246    56.282
```

Обратите внимание: библиотека pandas добавила целочисленный счетчик к именам столбцов, чтобы они отличались друг от друга.

Начнем обработку с одной капли, преобразовав ее данные:

```
In [x]: dropA = df.loc['A'].T
In [x]: dropA
Out[x]:
expt       1       2       3        4
tg    13.102  12.938  13.023   12.943
te    46.822  86.767  61.082   86.747
tg.1   12.941  13.032  12.958  12.922
te.1   46.896  86.952  60.826  86.840
tg.2   13.086  13.086  12.998  13.054
te.2   46.681  86.746  60.860  86.899
```

Можно изменить метки каждой строки на упрощенные `'tg'` и `'te'`:

```
In [x]: dropA.index = dropA.index.str.slice(0, 2)
In [x]: dropA
Out[x]:
expt       1       2       3        4
tg    13.102  12.938  13.023   12.943
te    46.822  86.767  61.082   86.747
tg    12.941  13.032  12.958   12.922
te    46.896  86.952  60.826   86.840
tg    13.086  13.086  12.998   13.054
te    46.681  86.746  60.860   86.899
```

Требуется среднее значение по всем значениям t_g (при отсутствии электрического поля капля при падении на расстояние d затрачивает одинаковое время) и среднее значение t_e для каждого столбца (в каждом эксперименте заряд капли может быть различным, но время падения/подъема измеряется три раза для каждого эксперимента):

```
In [x]: tg = dropA.loc['tg'].values.mean()
In [x]: te = dropA.loc['te'].mean()
In [x]: tg
Out[x]: 13.006916666666667
In [x]: te
Out[x]:
expt
1    46.799667
2    86.821667
3    60.922667
4    86.828667
dtype: float64
```

Теперь используем значение t_g для вычисления радиуса капли:

```
In [x]: a = np.sqrt(9*eta*d/tg/2/rhop/g)
In [x]: a
Out[x]: 2.8181654881967875e-06
```

т. е. приблизительно 2.82 мкм. Заряд, определяемый в каждом эксперименте:

```
In [x]: q = 6 * np.pi * eta * a * d / E * (1/tg + 1/te)
In [x]: q
Out[x]:
expt
1   -3.340563e-18
2   -3.005663e-18
3   -3.172143e-18
4   -3.005631e-18
dtype: float64
```

Повторяя этот процесс для всех капель, можно добавить столбец `q` в объект `DataFrame df`:

```
for drop in df.index.levels[0]:
    drop_df = df.loc[drop].T
    drop_df.index = drop_df.index.str.slice(0, 2)
    tg = drop_df.loc['tg'].values.mean()
    te = drop_df.loc['te'].mean()
    a = np.sqrt(9*eta*d/tg/2/rhop/g)
    q = 6 * np.pi * eta * a * d / E * (1/tg + 1/te)
    df.loc[drop, 'q'] = q.values
```

Теперь для удобства отсортируем заряды капель по величине и построим график по отсортированному массиву и соседним разностям (см. рис. 9.5):

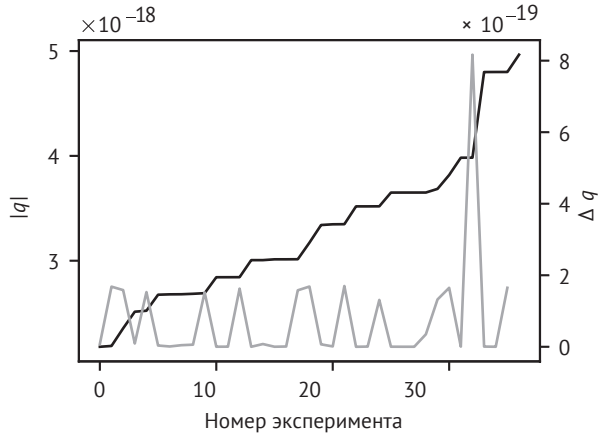


Рис. 9.5. Отсортированные значения зарядов q и соседние разности Δq

```
In [x]: sorted_q = sorted(-df.loc[:, 'q'])
In [x]: plt.plot(sorted_q)
In [x]: plt.ylabel('$|q|$')
In [x]: plt.twinx()
In [x]: dq = np.diff(sorted_q)
In [x]: plt.plot(dq)
In [x]: plt.ylabel(r'$\Delta q$')
In [x]: plt.show()
```

Выглядит вполне возможным предположение о том, что заряд капли всегда является результатом умножения на некоторое значение, находящееся между 1×10^{-19} Кл и 2×10^{-19} Кл. Следовательно, можно дать приблизительную оценку значения $|e|$:

```
In [x]: e_estimate = dq[(dq >1.e-19) & (dq <2.e-19)].mean()
In [x]: e_estimate
Out[x]: 1.5697150510604604e-19
```

Теперь можно добавить в объект `df` столбец, содержащий число элементарных зарядов, гипотетически предполагаемых в каждом эксперименте:

```
In [x]: df['N'] = (df['q'] / e_estimate).astype(int)
```

Дальнейшая обработка всех данных дает нам приблизительную оценку величины заряда электрона:

```
In [x]: (df['q']/df['N']).mean()
Out[x]: 1.5923552150386455e-19
```

Погрешность вычисленного здесь значения находится в пределах 1 % по сравнению со значением по определению.

9.4.5 Упражнения

Задачи

39.4.1. Использовать метод `cut` (pandas) для классификации звезд в наборе данных из задачи 39.2.3 по их температуре с распределением звезд по группам M, K, G, F, A, B и O с левыми границами (в K) 2400, 3700, 5200, 6000, 7500, 10 000 и 30 000.

Изменить исходный код решения этой задачи для отображения звезд на графике цветом, соответствующим их температуре, используя следующее отображение:

```
color_mapping = {'M': '#FFB56C', 'K': '#FFDAB5', 'G': '#FFEDE3', 'F': '#F9F5FF',
                 'A': '#D5E0FF', 'B': '#A2C0FF', 'O': '#92B5FF'}
```

Совет: pandas предоставляет метод `map` для отображения значений, вводимых из существующего столбца, в значения, выводимые в новый столбец, с использованием словаря.

39.4.2. Выполнить повторный анализ данных из примера П9.11, в котором рассматривался эксперимент Милликена с каплями масла, с использованием более точного приближения значения эффективной вязкости воздуха:

$$\eta = \eta_0 / (1 + b/ap),$$

где p = 100.82 кПа – давление воздуха, $\eta_0 = 1.859 \times 10^{-5}$ кг/м·с, $b = 7.88 \times 10^{-3}$ Па·м, a – радиус капли.

39.4.3. Группа цифровых технологий Кембриджского университета (Cambridge University Digital Technology Group) ведет записи погодных условий, наблюдаемых с крыши собственного здания, начиная с 1995 г., и предоставляет эти данные для свободного скачивания здесь: www.cl.cam.ac.uk/research/dtg/weather/.

Прочитать весь этот набор данных и выполнить его парсинг средствами библиотеки pandas, чтобы определить: а) наиболее частое направление ветра, б) самую высокую измеренную скорость ветра, в) год с самым солнечным июнем, г) день с наибольшим количеством осадков, д) самую низкую измеренную температуру. Следует отметить, что в этом наборе данных встречаются отсутствующие и некорректные точки данных.

39.4.4. Набор данных, доступный для скачивания по адресу https://scipython.com/ex/baq, содержит список следующих числовых значений, отслеживаемых по времени, в долл. США: а) цена золота, б) индекс фондового рынка США S&P 500, в) цена криптовалюты биткойн. Сравнить эффективность этих индексов за период 2010–2020 гг. с учетом регулярных инвестиций в размере 100 долл. в месяц.

9.5 Группирование и агрегация данных

9.5.1 Группирование DataFrame с помощью метода groupby

Мощный метод `groupby` (pandas) можно использовать для анализа данных в объектах `Series` или `DataFrame` на основе их категоризации в соответствии с некоторой ключевой строкой (или ключевым столбцом) значений. Термин «разделение–обработка–объединение» (split-apply-combine) кратко описывает этот процесс: сначала данные разделяются в соответствии с их категоризацией, далее требуемая методика анализа или статистический метод (например, суммирование значений либо поиск их среднего значения) применяется к разделенным на группы данным, наконец, результаты анализа объединяются в итоговый объект. На рис. 9.6 показан простой пример такого процесса.

Пример П9.12. Рассмотрим следующую таблицу результатов получения трех (химических) соединений A, B и C, полученных в ходе эксперимента по синтезу тремя студентами Anu, Jenny и Tom.

```
In [x]: data = [['Anu', 'A', 5.4], ['Anu', 'B', 6.7], ['Anu', 'C', 10.1],
...:            ['Jenny', 'A', 6.5], ['Jenny', 'B', 5.9], ['Jenny', 'C', 12.2],
...:            ['Tom', 'A', 4.0], ['Tom', 'B', None], ['Tom', 'C', 9.5]
...:           ]
In [x]: df = pd.DataFrame(data, columns=['Student', 'Compound', 'Yield /g'])
```

```
In [x]: print(df)
  Student Compound  Yield /g
0     Anu        A       5.4
1     Anu        B       6.7
2     Anu        C      10.1
3   Jenny        A       6.5
4   Jenny        B       5.9
5   Jenny        C      12.2
6     Tom        A       4.0
7     Tom        B       NaN
8     Tom        C       9.5
```

Один из способов анализа этих данных – группирование их по соединениям («разделение» на структуры данных, в каждой из которых общим будет значение столбца `'Compound'`), затем применение некоторой операции (например, поиск среднего значения) к каждой группе перед объединением в одном объекте `DataFrame`, как показано на рис. 9.6.

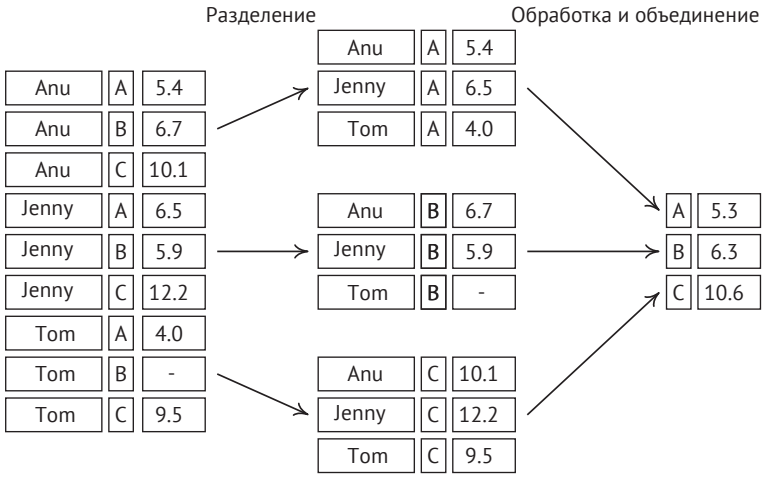


Рис. 9.6. Графическая схема методики (парадигмы) «разделение–обработка–объединение» для анализа данных, разделенных на группы средствами pandas: объект DataFrame разделяется на группы по химическим соединениям ('Compound') (A, B и C). Функция mean применяется к этим группам. Полученные значения объединяются в возвращаемом объекте

```
In [x]: grouped = df.groupby('Compound')
In [x]: grouped.mean()
Out[x]:
          Yield /g
Compound
A              5.3
B              6.3
C             10.6
```

Здесь столбец 'Student' исключается как так называемый «конфликтующий» столбец: не существует приемлемого способа получения среднего значения по текстовым строкам. Тем не менее функции max() и min() учитывают лексикографическое упорядочение строк:

```
In [x]: grouped.max()
Out[x]:
          Student  Yield /g
Compound
A             Tom       6.5
B             Tom       6.7
C             Tom      12.2
```

Обратите внимание: функция max() вернула значение 'Tom' в каждой строке, поскольку это имя лексикографически последнее («наибольшее») в столбце 'Student'. Столбец 'Yield /g' состоит из максимальных значений количества каждого полученного соединения для всех студентов. Чтобы применить функцию только к подмножеству столбцов (это может оказаться необходимым для весьма больших объектов DataFrame), следует выбрать их перед вызовом функции, например:

```
In [x]: grouped['Yield /g'].min()
Out[x]:
Compound
A    4.0
B    5.9
C    9.5
Name: Yield /g, dtype: float64
```

По содержимому объекта, возвращаемого методом groupby(), можно выполнить итеративный проход:

```
In [x]: for compound, group in grouped:
...:        print('Compound:', compound)
...:        print(group)
...:
Compound: A
  Student Compound  Yield /g
0     Anu        A       5.4
3   Jenny        A       6.5
6     Tom        A       4.0
Compound: B
  Student Compound  Yield /g
1     Anu        B       6.7
4   Jenny        B       5.9
7     Tom        B       NaN
Compound: C
  Student Compound  Yield /g
2     Anu        C      10.1
5   Jenny        C      12.2
8     Tom        C       9.5
```

Также можно выполнить группирование по столбцу 'Student':

```
In [x]: grouped = df.groupby('Student')

In [x]: grouped.mean()
Out[x]:
         Yield /g
Student
Anu          7.40
Jenny        8.20
Tom          6.75
```

Еще одной мощной функциональной возможностью является группирование на основе заданного отображения, представленного, например, словарем. Предположим, что все студенты принимают участие в различных программах обучения:

```
In [x]: degree_programmes = {'Anu': 'Chemistry',
                             'Jenny': 'Chemistry',
                             'Tom': 'Pharmacology'}
```

Сначала нужно преобразовать столбец 'Student' в Index, затем сгруппировать данные, но не по самому индексу, а с использованием предоставленного отображения:

```
In [x]: df.set_index('Student', inplace=True)
In [x]: df.groupby(degree_programmes).mean()
Out[x]:
              Yield /g
Chemistry         7.80
Pharmacology      6.75
```

Таким образом, среднее количество вещества, полученное студентами, специализирующимися в химии, равно 7.8 г, тогда как студенты, специализирующиеся в фармакологии, получили в среднем 6.75 г вещества.

9.5.2. Упражнения

Задачи

39.5.1. Организация экономического сотрудничества и развития (ОЭСР) в рамках своей Международной программы по оценке образовательных достижений учащихся (Programme for International Student Assessment – PISA) публикует оценку образовательных систем по всему миру, проводя раз в три года тест, оценивающий функциональную грамотность школьников в разных странах мира и умение применять знания на практике. В тесте участвуют подростки в возрасте 15 лет. Оценка качества образования проводится по трем основным направлениям: грамотность чтения, математическая грамотность и естественно-научная грамотность.

Данные PISA в хронологической последовательности (по годам) можно скачать здесь: https://scipython.com/ex/bza. Прочитать эти данные в объект DataFrame (pandas) и использовать функциональные средства группирования для определения и визуализации: а) общей функциональной грамотности во

всех исследуемых странах по времени, б) несоответствия между полами (если оно наблюдается) по каждому исследуемому направлению: чтение, математика и естественно-научная грамотность, в) корреляции между этими исследуемыми направлениями по всем странам.

39.5.2. Прочитать данные, расположенные по адресу https://scipython.com/ex/bar и содержащие результаты последних сезонов гонок Формула 1 Гран-при. Составить рейтинг: а) пилотов по количеству побед на этапах, б) конструкторов по количеству побед на этапах, в) этапов (трасс) по усредненному самому быстрому кругу (по времени) в гонке.

9.6 Примеры

В приведенных ниже примерах демонстрируется практическое использование библиотеки pandas в двух исследованиях, предполагающих анализ и визуализацию реальных данных.

Пример П9.13. Файл *nuclear-explosion-data.csv*, доступный для скачивания по адресу https://scipython.com/eg/ban, содержит данные обо всех ядерных взрывах с 1945 по 1998 г.[130] Воспользуемся средствами библиотеки pandas для анализа различными способами.

При просмотре файла в текстовом редакторе выясняется, что в нем содержится строка заголовка с именами столбцов, поэтому содержимое можно загрузить непосредственно с помощью метода `pd.read_csv` и исследовать основные свойства и особенности данных:

```
In [x]: import pandas as pd
In [x]: df = pd.read_csv('nuclear-explosion-data.csv')
In [x]: df.head()
```

```
Out[x]:
       date      time      id country  ... yield_upper purpose        name      type
0  19450716  123000.0   45001     USA  ...        21.0      WR     TRINITY    TOWER
1  19450805  231500.0   45002     USA  ...        15.0  COMBAT   LITTLEBOY  AIRDROP
2  19450809   15800.0   45003     USA  ...        21.0  COMBAT      FATMAN  AIRDROP
3  19460630  220100.0   46001     USA  ...        21.0      WE        ABLE  AIRDROP
4  19460724  213500.0   46002     USA  ...        21.0      WE       BAKER       UW

[5 rows x 16 columns]
```

```
In [x]: df.index
Out[x]: RangeIndex(start=0, stop=2051, step=1)
```

```
In [x]: df.columns
Out[x]:
Index(['date', 'time', 'id', 'country', 'region', 'source', 'lat', 'long',
       'mb', 'Ms', 'depth', 'yield_lower', 'yield_upper', 'purpose', 'name', 'type'],
      dtype='object')
```

[130] По материалам N.-O. Bergkvist and R. Ferm, Nuclear Explosions 1945–1998, Swedish Defence Research Establishment/SIPRI, Stockholm, July 2000.

В файле 16 столбцов, здесь мы будем рассматривать только описанные в табл. 9.2.

Таблица 9.2. Наиболее важные столбцы данных о ядерных взрывах в файле nuclear-explosion-data.csv

Столбец	Описание
`date`	Дата взрыва в формате `YYYYMMDD`
`time`	Время взрыва в формате `HHMMSS.Z`, где `Z` представляет десятые доли секунды
`country`	Страна, которая произвела взрыв
`lat`	Широта места взрыва в градусах относительно экватора
`long`	Долгота места взрыва в градусах относительно нулевого меридиана
`yield_lower`	Нижняя оценка мощности взрыва в килотоннах (кт) в тротиловом (TNT) эквиваленте
`yield_upper`	Верхняя оценка мощности взрыва в килотоннах (кт) в тротиловом (TNT) эквиваленте
`type`	Метод дислоцирования и развертывания ядерного устройства

Вполне естественным будет назначение даты и времени взрыва для индекса объекта `DataFrame`. Для этого определяются и применяются некоторые вспомогательные функции:

```
from datetime import datetime
def parse_time(t):
    hour, t = divmod(t, 10000)
    minute, t = divmod(t, 100)
    return int(hour), int(minute), int(t)

def parse_datetime(date, time):
    date_and_time = datetime.strptime(str(date), '%Y%m%d')
    hour, minute, second = parse_time(time)
    return date_and_time.replace(hour=hour, minute=minute, second=second)

df.index = pd.DatetimeIndex([parse_datetime(date, time) for date, time in
                             zip(df['date'], df['time'])])
```

Можно сформировать график количества взрывов в каждом году, группируя данные по `index.year`, и определить размер каждой группы. Затем создается обычная столбиковая диаграмма библиотеки Matplotlib:

```
explosion_number = df.groupby(df.index.year).size()

import matplotlib.pyplot as plt
fig, ax = plt.subplots()
ax.bar(explosion_number.index, explosion_number.values)
ax.set_xlabel('Year')
ax.set_ylabel('Number of nuclear explosions')
plt.show()
```

На рис. 9.7 показан итоговый график.

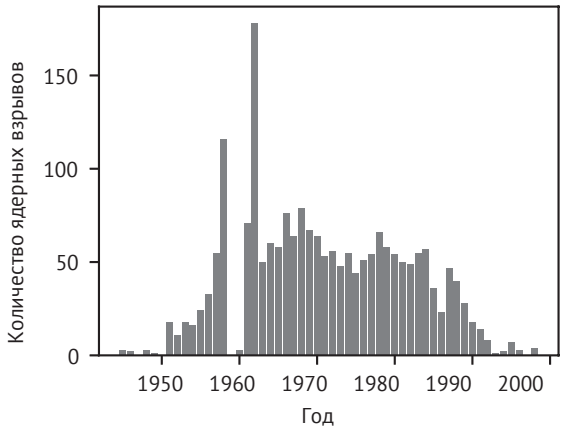


Рис. 9.7. Столбиковая диаграмма количества ядерных взрывов в период с 1948 по 1998 г.

Составная столбиковая диаграмма может оказать помощь в разделении ежегодного количества взрывов по странам. Сначала выполняется группирование по году и по стране, затем определяется счетчик взрывов для результатов этого группирования с помощью метода `size()`:

```
df2 = df.groupby([df.index.year, df.country])
explosions_by_country = df2.size()
print(explosions_by_country.head(7))
```

```
       country
1945   USA         3
1946   USA         2
1948   USA         3
1949   USSR        1
1951   USA        16
       USSR        2
1952   UK          1
dtype: int64
```

Далее второй индекс разделяется на столбцы с помощью метода `unstack` с заполнением пустых элементов нулями:

```
explosions_by_country = explosions_by_country.unstack().fillna(0)
print(explosions_by_country.head(7))
```

```
country   CHINA   FRANCE   INDIA   PAKISTAN    UK    USA    USSR
1945        0.0      0.0     0.0        0.0   0.0    3.0     0.0
1946        0.0      0.0     0.0        0.0   0.0    2.0     0.0
1948        0.0      0.0     0.0        0.0   0.0    3.0     0.0
1949        0.0      0.0     0.0        0.0   0.0    0.0     1.0
1951        0.0      0.0     0.0        0.0   0.0   16.0     2.0
1952        0.0      0.0     0.0        0.0   1.0   10.0     0.0
1953        0.0      0.0     0.0        0.0   2.0   11.0     5.0
```

Затем каждая строка в этом объекте DataFrame может быть изображена как составная столбиковая диаграмма на графике Matplotlib:

```
countries = ['USA', 'USSR', 'UK', 'FRANCE', 'CHINA', 'INDIA', 'PAKISTAN']
bottom = np.zeros(len(explosions_by_country))
fig, ax = plt.subplots()
for country in countries:
    ax.bar(explosions_by_country.index, explosions_by_country[country],
           bottom=bottom, label=country)
    bottom += explosions_by_country[country].values

ax.set_xlabel('Year')
ax.set_ylabel('Number of nuclear explosions')
ax.legend()
plt.show()
```

На рис. 9.8 показана полученная составная столбиковая диаграмма.

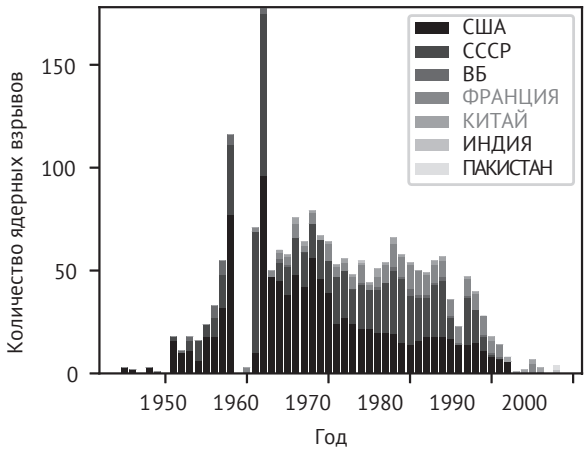


Рис. 9.8. Составная столбиковая диаграмма, отображающая ежегодное количество ядерных взрывов, произведенных в различных странах с 1945 по 1998 г.

Пакет geopandas предоставляет удобный способ изображения полученных данных на карте мира. Полное описание геоинформационных систем (GIS) не относится к тематике этой книги, но пакет geopandas относительно прост в использовании и самодостаточен. Сначала считывается объект DataFrame для карты мира в низком разрешении (включенной в комплект geopandas) и создается график в объекте Matplotlib Axes. В рассматриваемом здесь примере используется принятая по умолчанию равнопромежуточная (прямоугольная) проекция, но с определением свойств границ и заполнения географических областей серым цветом:

```
import geopandas
world = geopandas.read_file(geopandas.datasets.get_path('naturalearth_lowres'))
fig, ax = plt.subplots()
world.plot(ax=ax, color="0.8", edgecolor='black', linewidth=0.5)
```

В данных представлены нижняя и верхняя оценки мощности взрыва, поэтому вычисляется среднее значение и в график добавляются круги, как на точечной диаграмме, в местах с соответствующей широтой и долготой произведенного взрыва. Динамический диапазон мощностей достаточно велик: от нескольких килотонн в тротиловом эквиваленте до 58.6 млн тонн термоядерной бомбы «Царь-бомба», испытанной в 1961 г., поэтому устанавливается минимальный размер кружков, чтобы взрывы малой мощности также были видны на карте:

```
df['yield_estimate'] = df[['yield_lower','yield_upper']].mean(axis=1)
sizes = (df['yield_estimate'] / 120).clip(10)
ax.scatter(df['long'], df['lat'], s=sizes , fc='r', ec='none', alpha=0.5)
ax.set_ylim(-60, 90)
plt.axis('off')
plt.show()
```

Полученный график показан на рис. 9.9.

Рис. 9.9. Карта произведенных ядерных взрывов с отображением их мощности за период с 1945 по 1998 г.

Пример П9.14. Файл *volcanic-eruptions.csv*, доступный для скачивания по адресу https://scipython.com/eg/bap, содержит данные о 822 самых мощных вулканических явлениях на Земле за период с 1750 г. до н. э. по 2020 г. н. э. Источник данных: Национальные центры экологической информации США (NCEI)[131]. Информация о каждом вулканическом явлении размещена в полях, разделенных запятыми, и включает дату, название вулкана, место его расположения, тип, приблизительное (оценочное) количество человеческих жертв и показатель силы извержения по шкале вулканической активности (Volcanic Explosivity Index – VEI).

Эти данные легко поддаются парсингу с последующим размещением в объекте `DataFrame`:

```
In [x]: df = pd.read_csv('volcanic-eruptions.csv', index_col=0)
```

[131] https://www.ngdc.noaa.gov/hazard/volcano.shtml.

В этой базе данных самым разрушительным и опасным для людей является извержение вулкана Илопанго, произошедшее приблизительно в середине V в. н. э.:

```
In [x]: df.loc[df['Deaths'].idxmax()]
Out[x]:
Year                 450
Month                NaN
Day                  NaN
Name             Ilopango
Location     El Salvador
Country      El Salvador
Latitude          13.672
Longitude        -89.053
Elevation            450
Type             Caldera
VEI                    6
Deaths             30000
Name: 25, dtype: object
```

Может оказаться полезным наличие столбца с указанием дня, месяца и года извержения с соответствующим преобразованием (парсингом) в строку. Определим вспомогательную функцию `get_date`:

```
def get_date(year, month, day):
    if year < 0:
        s_year = f'{-year} BCE'
    else:
        s_year = str(year)
    if pd.isnull(month):
        return s_year
    s_date = f'{int(month)}/{s_year}'
    if pd.isnull(day):
        return s_date
    return f'{int(day)}/{s_date}'
```

и применим ее к объекту `DataFrame`:

```
In [x]: df['date'] = [get_date(year, month, day) for year, month, day in
                                    zip(df['Year'], df['Month'], df['Day'])]
```

Простая фильтрация может предоставить нам список извержений с показателем силы извержения по шкале вулканической активности (VEI) не менее 6 с начала XIX в.:

```
In [x]: df[(df['VEI'] >= 6) & (df['Year'] >= 1800)]
Out[x]:
     Year  Month   Day         Name  ...           Type  VEI   Deaths        date
218  1815    4.0  10.0      Tambora  ...  Stratovolcano  7.0  11000.0   10/4/1815
322  1883    8.0  27.0     Krakatau  ...        Caldera  6.0   2000.0   27/8/1883
365  1902   10.0  25.0  Santa Maria  ...  Stratovolcano  6.0   2500.0  25/10/1902
386  1912    9.0   6.0    Novarupta  ...        Caldera  6.0      2.0    6/9/1912
650  1991    6.0  15.0     Pinatubo  ...  Stratovolcano  6.0    350.0   15/6/1991
```

Чтобы определить 10 самых мощных взрывных извержений, можно отфильтровать данные с исключением записей с неизвестными значениями VEI перед сортировкой:

```
In [x]: df[pd.notnull(df['VEI'])].sort_values('VEI').tail(10)[
...:                 ['date', 'Name', 'Type', 'Country', 'VEI']]
Out[x]:
          date             Name            Type           Country    VEI
29         653         Dakataua         Caldera  Papua New Guinea    6.0
25         450          Ilopango         Caldera       El Salvador    6.0
22         240           Ksudach   Stratovolcano            Russia    6.0
21         230             Taupo         Caldera       New Zealand    6.0
18          60   Bona-Churchill   Stratovolcano     United States    6.0
99   19/2/1600     Huaynaputina   Stratovolcano              Peru    6.0
1     1750 BCE       Veniaminof   Stratovolcano     United States    6.0
40        1000     Changbaishan   Stratovolcano       North Korea    7.0
218  10/4/1815          Tambora   Stratovolcano         Indonesia    7.0
3     1610 BCE        Santorini  Shield volcano            Greece    7.0
```

Но существует много записей со значением VEI, равным 6, и их порядок здесь не вполне понятен. Более удачным подходом может оказаться предварительная сортировка сначала по значению VEI, потом по числу жертв. Необходимо установить параметр `na_position='first'`, чтобы все нулевые (неизвестные) значения располагались перед числовыми значениями (это позволит эффективно определить ранг по минимальным значениям):

```
In [x]: df.sort_values(['VEI', 'Deaths'], na_position='first').tail(10)[
...:                 ['date', 'Name', 'Type', 'Country', 'VEI', 'Deaths']]
Out[x]:
           date          Name             Type           Country  VEI   Deaths
386    6/9/1912     Novarupta          Caldera     United States  6.0      2.0
650   15/6/1991      Pinatubo    Stratovolcano       Philippines  6.0    350.0
99    19/2/1600  Huaynaputina    Stratovolcano              Peru  6.0   1500.0
120        1660   Long Island  Complex volcano  Papua New Guinea  6.0   2000.0
322   27/8/1883      Krakatau          Caldera         Indonesia  6.0   2000.0
365  25/10/1902   Santa Maria    Stratovolcano         Guatemala  6.0   2500.0
25          450      Ilopango          Caldera       El Salvador  6.0  30000.0
3      1610 BCE     Santorini   Shield volcano            Greece  7.0      NaN
40         1000  Changbaishan    Stratovolcano       North Korea  7.0      NaN
218   10/4/1815       Tambora    Stratovolcano         Indonesia  7.0  11000.0
```

Также можно создать несколько гистограмм с объединенными данными из нескольких столбцов (см. рис. 9.10):

```
fig, axes = plt.subplots(nrows=2, ncols=2)
df['Day'].hist(bins=31, ax=axes[0][0], grid=False)
axes[0][0].set_xlabel('(a) Day')
df['Month'].hist(bins=np.arange(1, 14) - 0.5, ax=axes[0][1], grid=False)
axes[0][1].set_xticks(range(1, 13))
axes[0][1].set_xlabel('(b) Month')
df[df['Year']>1600]['Year'].hist(ax=axes[1][0], grid=False)
axes[1][0].set_xlabel('(c) Year')
df['Elevation'].hist(ax=axes[1][1], grid=False)
```

```
axes[1][1].set_xlabel('(d) Elevation /m')
plt.tight_layout()
plt.show()
```

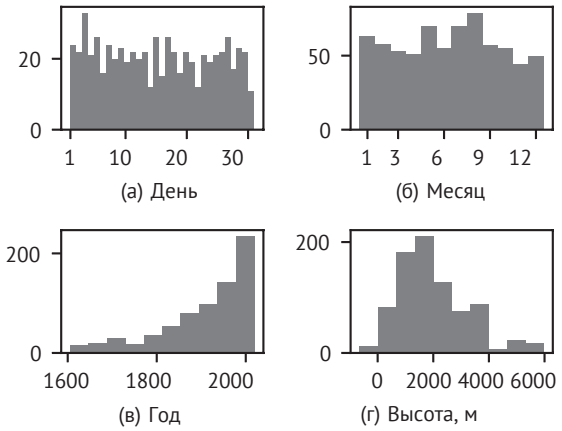


Рис. 9.10. Диаграммы, объединяющие данные об извержениях вулканов из нескольких столбцов: а) по дню месяца; б) по месяцу года; в) по частоте по годам с 1600 г. – можно надеяться, что точность записей об извержениях вулканов постоянно улучшалась с 1600 г. и частота извержений в действительности не увеличивалась; г) высота вулканов

Глава 10

Общие положения научного программирования

10.1 Арифметика с плавающей точкой

10.1.1 Представление действительных чисел

Действительные числа, такие как 1.2, –0.36, π, 4 и 13256.625, можно мысленно представить как точки на непрерывной бесконечной числовой прямой[132]. Некоторые действительные числа (включая сами целые числа) можно представить в виде отношения двух целых чисел, например 5/8 и 1/3. Такие числа называются рациональными. Другие числа, такие как π, *e* и √2, невозможно представить в таком виде, поэтому они называются иррациональными.

Существует несколько возможных способов записи действительных чисел в зависимости от того, к какой категории они относятся, но не все способы способны точно отображать числовое значение (ручки и карандаши имеют ограниченный ресурс использования). Например, рациональное число 5/8 можно точно записать как разложение в десятичную дробь:

$$5/8 = 6/10 + 2/100 + 5/1000,$$

но число 1/3 невозможно записать с помощью конечного количества знаков после запятой в аналогичном разложении:

$$1/3 = 3/10 + 3/100 + 3/1000 + \ldots = 0.333\ldots$$

При записи 1/3 в виде разложения в десятичную дробь непременно приходится в каком-то месте обрывать бесконечную последовательность троек.

Иррациональные числа могут быть точно описаны (с помощью некоторых предполагаемых геометрических или каких-либо других знаний), например число π – это отношение длины окружности к ее диаметру, √2 – длина гипотенузы прямоугольного треугольника с длинами катетов 1. Но для численного

[132] Очевидно, что целое число, такое как 4, представляет собой всего лишь особый случай действительного числа.

представления или сохранения таких значений необходим определенный уровень приближения. Например, дробь 355/113 представляет собой общеизвестное рациональное приближенное значение числа π. Десятичная дробь, более точно представляющая π: 3.14159265358979. Но при разложении в десятичную дробь[133] требуется бесконечное количество знаков после запятой, чтобы точно выразить значение π, точно так же, как необходимо бесконечное количество троек при разложении в десятичную дробь числа 1/3.

Компьютеры хранят числа в двоичной (бинарной) форме, и некоторые условия, ограничивающие точность десятичного представления действительного числа, применяются и к его двоичному представлению.

Например, числу 5/8 соответствует точное двоичное представление с помощью конечного числа битов:

$$5/8 = 1/2 + 0/4 + 1/8 = 0.101_2,$$

но двоичное представление

$$1/10 = 0.0001100110011001100110011\ldots_2$$

является бесконечно повторяющейся последовательностью. Можно сохранить только конечное число знаков после десятичной точки, поэтому такая усеченная последовательность битов после обратного преобразования в десятичную дробь дает результат

$$1/10 \approx 0.1000000000000000006$$

при использовании так называемого стандарта двойной точности (double-precision), общепринятого для большинства языков программирования в большинстве операционных систем. Это наиболее близкое (точное) числовое представление дроби 1/10.

Формат представления чисел с плавающей точкой двойной точности определяется международным стандартом IEEE-754. Представление числа, хранящееся в 64 битах (8 байтах), разделяется на три части: один знаковый бит, 11-битовый показатель степени и 52-битовая мантисса (значащая часть числа). Это становится более понятным на конкретном примере: десятичную дробь 13 256.625 можно записать в научном формате:

$$13\,256.625 = +1.3256625 \times 10^4$$

и сохранить со знаковым битом, соответствующим знаку +, как мантиссу, равную `13256625` (где подразумевается, что десятичная точка расположена после первого знака), и показатель степени 4. Такая форма записи называется «с пла-

[133] Следует отметить, что разложение в десятичную дробь – это просто рациональное число, в знаменателе которого записывается некоторая степень числа 10: 3.14159265358979 = 314 159 265 358 979 / 100 000 000 000 000.

вающей точкой» (floating point), потому что десятичная точка[134] смещается на количество знаков, определяемое показателем степени.

Представление чисел с плавающей точкой в двоичном виде работает аналогичным образом, но используются, разумеется, только цифры 0 и 1. Это позволяет воспользоваться хитроумным приемом: двоичная точка числа (равнозначная десятичной точке в системе счисления с основанием 10) сдвигается так, чтобы мантисса не содержала начальных нулей, т. е. начиналась с 1. Поскольку все мантиссы, нормализованные таким способом, непременно начинаются с 1, нет необходимости хранить эту начальную единицу, так что в 52-битовой мантиссе можно сохранить число с эффективной точностью 53 бита[135]. Опущенный бит иногда называют скрытым битом (hidden bit).

В рассматриваемом здесь примере число 13 256.625 в действительности можно точно представить как следующее двоичное число:

$$13256.62510 \equiv 11001111001000.101_2.$$

Следовательно, нормализованная форма мантиссы: `1100111100100010`1, а показатель степени равен 13, таким образом:

$$11001111001000.101_2 = 1.1001111001000101 \times 2^{13}.$$

В соответствии с описанным выше приемом первый знак нормализованной мантиссы, который всегда равен 1, опускается, и мантисса сохраняется в следующем виде:

```
10011110010001010000000000000000000000000000000000
```

Для обеспечения возможности использования чисел с отрицательными степенями (чисел, меньших 1) показатель степени сохраняется с так называемым сдвигом (смещением – bias): к действительному показателю степени прибавляется 1023. Таким образом, реальные показатели степени в диапазоне от –1023 до +1023 сохраняются как числа от 1 до 2046. В рассматриваемом здесь примере 11-битовое поле показателя степени содержит значение 13 + 1023 = 1036:

```
10000001100
```

Здесь знаковый бит равен `0`, обозначающему положительное число. Полное 64-битовое представление числа с плавающей точкой 13 256.625 (для удобства части представления разделены пробелами):

```
0 10000001100 1001111001000101000000000000000000000000000000000000
```

и это точное представление десятичной дроби. Но представление дроби 0.1

```
0 01111111011 1001100110011001100110011001100110011001100110011010
```

[134] Более общее название: точка (запятая) в позиционном представлении чисел (radix point) в системах счисления, отличающихся от привычной системы с основанием 10.

[135] Следует отметить, что этот прием работает только в двоичной системе.

не является точным (из-за описанного выше усечения и округления бесконечно повторяющейся последовательности 0011...) – после обратного преобразования в десятичную дробь получается число

```
0.1000000000000000005551115123126
```

В общем случае 53 бита (включая скрытый бит) мантиссы обеспечивают точность, приблизительно равную 15 значащим цифрам: $\log_{10}(2^{53}) = 15.95$. Любое вычисление с бóльшим количеством значащих цифр дает результат с ошибкой округления. Верхняя граница относительной ошибки из-за округления называется машинным эпсилоном (не путать с машинным нулем) ε. В языке Python:

```
In [x]: import sys
In [x]: eps = sys.float_info.epsilon
In [x]: eps
Out[x]: 2.220446049250313e-16
```

Можно показать, что максимальный интервал между двумя нормализованными числами с плавающей точкой равен 2ε. Таким образом, условное выражение `x + 2*eps == x` всегда вычисляется с неизменным результатом `False`.

10.1.2 Сравнение чисел с плавающей точкой

Из-за ограниченной точности представления чисел с плавающей точкой для (большинства) действительных чисел весьма опасно сравнивать два объекта типа `float` на равенство. Например, рассмотрим операцию возведения в квадрат числа 0.1:

```
In [x]: (0.1)**2
Out[x]: 0.010000000000000002
```

Как нам уже известно, результат не равен в точности 0.01, но полученный результат даже не является наиболее близким представлением числа 0.01, потому что в действительности квадратом этого числа являлось значение 0.100000000000000006. Печальным следствием этого факта становится неудачное сравнение:

```
In [x]: (0.1)**2 == 0.01
Out[x]: False
```

Библиотека NumPy предоставляет методы `isclose` и `allclose` (см. раздел 6.1.12) для сравнения двух чисел с плавающей точкой или массивов таких чисел с заданным или установленным по умолчанию пределом допустимой погрешности:

```
In [x]: np.isclose(0.1**2, 0.01)
Out[x]: True
```

Следует также отметить, что сложение чисел с плавающей точкой не всегда является ассоциативным:

```
In [x]: a, b, c = 1e14, 25.44, 0.74
In [x]: (a + b) + c
Out[x]: 100000000000026.17

In [x]: a + (b + c)
Out[x]: 100000000000026.19
```

Кроме того, в общем случае умножение чисел с плавающей точкой не всегда является дистрибутивным относительно сложения:

```
In [x]: a, b, c = 100, 0.1, 0.2

In [x]: a*b + a*c
Out[x]: 30.0

In [x]: a * (b + c)
Out[x]: 30.000000000000004
```

10.1.3 Потеря значащих разрядов

Большинство операций с числами с плавающей точкой (такие как сложение и вычитание) приводят к потере значащих разрядов. Это означает, что количество значащих разрядов (цифр) в полученном результате может быть меньшим, чем в исходных числах (операндах), используемых при вычислении. Для наглядной демонстрации рассмотрим предполагаемое представление числа с плавающей точкой, используемое в десятичном виде с шестью значащими разрядами мантиссы, и выполним следующее вычисление, записанное в точной форме:

$$1.2345432 - 1.23451 = 0.0000332.$$

Наша воображаемая система не способна сохранить первый операнд с его полной точностью, но может обеспечить приближение в виде числа 1.23454. Тогда вычитание чисел с плавающей точкой даст результат:

$$1.23454 - 1.23451 = 0.00003.$$

Исходные числа имели точность до шестой значащей цифры, но результат обеспечивает точность только в первой значащей цифре. Следует отметить, что это не тот случай, когда точный результат невозможно представить во всех доступных разрядах нашей воображаемой архитектуры с плавающей точкой: число $0.0000332 \equiv 3.32 \times 10^{-5}$ содержит только три значащие цифры, т. е. не выходит за предел шести доступных разрядов. Критическая потеря значащих разрядов произошла исключительно из-за чрезвычайно малой разности между двумя исходными числами. Об этом явлении, которое иногда называют «катастрофическим взаимоуничтожением» (catastrophic cancellation), следует всегда помнить при вычитании двух чисел с близкими значениями.

Аналогичная потеря значащих разрядов может происходить при вычитании или сложении малого числа со значительно бóльшим числом:

12 345.6 + 0.123456 = 12 345.72345 (точный результат),
12 345.6 + 0.123456 = 12 345.7 (в системе с мантиссой с шестью значащими разрядами).

Даже несмотря на то, что 15 значащих разрядов представления числа с плавающей точкой двойной точности могут показаться обеспечивающими вполне достаточную точность при одном отдельном вычислении, необходимо всегда помнить о том, что многократное выполнение таких вычислений может увеличивать неизбежную ошибку округления до критической величины, если используемые числа невозможно представить точно. Рассмотрим следующий пример:

```
In [x]: for i in range(10000000):
....:       a += 0.1
....:

In [x]: a
Out[x]: 999999.9998389754
```

Разность между вычисленным приближенным значением и известным точным значением 1 000 000 составляет более 1.61×10^{-4}.

В модуле Python `math` есть функция `fsum`, использующая методику под названием алгоритм (компенсационного суммирования) Шевчука для компенсации ошибок округления и потери значащих разрядов. Сравним две реализации ранее вычисленной в простом цикле суммы с использованием выражения генератора:

```
In [x]: sum((0.1 for i in range (10000000)))
Out[x]: 999999.9998389754
In [x]: math.fsum((0.1 for i in range (10000000)))
Out[x]: 1000000.0
```

10.1.4 Обращение в машинный ноль и переполнение

Еще одним последствием способа компьютерной обработки чисел с плавающей запятой является ограничение минимальной и максимальной величин чисел, которые могут быть сохранены. Например, при байесовском анализе часто требуется перемножение весьма малых значений вероятностей, поскольку каждое значение вероятности представлено числом от 0 до 1. При большом количестве таких вероятностей их произведение может достигать значения, которое слишком мало для представления, в результате чего происходит обращение в машинный ноль (underflow):

```
In [x]: P = 1
In [x]: for i in range(101):
....:       print(P)
....:       P *= 5.e-4

1
0.0005
2.5e-07
1.25e-10
```

```
6.250000000000001e-14
...
1.0097419586828971e-307
5.0487097934146e-311         # Начинается денормализация.
2.5243548965e-314
1.2621776e-317
6.31e-321
5e-324
0.0                          # Обращение в машинный ноль.
0.0
```

Ниже этого значения Python начинает жертвовать некоторой степенью точности и работает с измененным представлением числа (денормализованное или субнормальное (по существу – ошибочное) число). Такой процесс называется постепенное обращение в машинный ноль. Как бы то ни было, в итоге число, представляемое машинным нулем, становится неотличимым от настоящего нуля. Минимальным числом, которое можно представить полностью по стандарту IEEE-754 с двойной точностью, является

```
In [x]: import sys
In [x]: sys.float_info.min
Out[x]: 2.2250738585072014e-308
```

Существует несколько возможных методик решения проблемы обращения в машинный ноль (помимо использования чисел с большей степенью точности, например `np.float128`, если это доступно). В приведенном выше примере часто применяется вычисление суммы логарифмов вероятностей, которые имеют более приемлемые величины, вместо непосредственного вычисления произведения вероятностей. Другой вариант: в приведенном выше коде нужно начать со значения `P = 1.e100` и обрабатывать полученные в результате числа с учетом этого постоянного множителя.

Переполнение (overflow) чисел с плавающей точкой – это проблема, возникающая на другом конце числовой шкалы: наибольшее число с двойной точностью, которое можно представить в Python:

```
In [x]: sys.float_info.max
Out[x]: 1.7976931348623157e+308
```

В библиотеке NumPy для чисел, вышедших за границу переполнения, устанавливаются специальные значения `inf` или `-inf` в зависимости от знака:

```
In [x]: f = 1
In [x]: for x in range(1, 40, 4):
...:        print('exp({}) = {}'.format(x**2, np.exp(x**2)))
...:
exp(1) = 2.718281828459045
exp(25) = 72004899337.38588
exp(81) = 1.5060973145850306e+35
exp(169) = 2.487524928317743e+73
exp(289) = 3.2441824460394912e+125
exp(441) = 3.340923407659982e+191
```

```
exp(625) = 2.7167594696637367e+271
exp(841) = inf
exp(1089) = inf
exp(1369) = inf
```

Это может приводить к некоторым странным отношениям между числами, которые слишком велики для корректного представления:

```
In [x]: a, b = 1.e500, 1.e1000
In [x]: a == b
Out[x]: True
In [x]: a, b
Out[x]: (inf, inf)
```

Существует еще одно специальное значение `nan` (NaN – Not a Number – не число), которое возвращают некоторые операции, работающие с «переполненными» числами:

```
In [x]: a / b
Out[x]: nan
```

В библиотеке NumPy также реализованы собственные специальные значения `numpy.nan` и `numpy.inf` (см. раздел 6.1.4).

Никогда не проверяйте объект на значение nan с помощью оператора `==`, потому что `nan` не равно даже самому себе[136]:

```
In [x]: c = a / b
In [x]: c == c
Out[x]: False
```

Для объектов типа `int` в языке Python не существует проблемы переполнения, так как Python автоматически выделяет память для их хранения с полной точностью (ограничение существует только по доступной оперативной памяти). Но массивы NumPy целых чисел, которые отображаются во внутренние структуры данных языка C, сохраняются в фиксированном количестве байтов (см. табл. 6.2), поэтому переполнение возможно. Например:

```
In [x]: a = np.zeros(3, dtype=np.int16)
In [x]: a[:] = -30000, 30000, 40000
In [x]: a
Out[x]: array([-30000, 30000, -25536], dtype=int16)

In [x]: b = np.zeros(3, dtype=np.uint16)
In [x]: b[:] = -30000, 40000, 70000
In [x]: b
Out[x]: array([35536, 40000, 4464], dtype=uint16)
```

Для знаковых 16-битных целых чисел определен диапазон от −32 768 до 32 767, т. е. от -2^{15} до $(2^{15} - 1)$. Из-за такого способа хранения в приведенном

[136] Это означает, что оператор == не может быть отношением равенства для чисел с плавающей точкой, так как он не является рефлексивным (возвратным).

выше примере попытка присваивания элементу массива `a[2]` числа 40 000 привела к присваиванию вместо этого значения $40\,000 - 2^{16} = -25\,536$. Аналогично беззнаковые 16-битовые целые числа ограничены значениями от 0 до 65 535, т. е. от 0 до $(2^{16} - 1)$. Отрицательные числа вообще не могут быть каклибо представлены, поэтому при попытке присваивания `b[0] = -30000` происходит преобразование в $-30\,000 \bmod 2^{16} = 35536$, а присваивание `b[2] = 70000` приводит к переполнению, и в итоге получается значение $70\,000 \bmod 2^{16} = 4464$.

10.1.5 Материалы для дальнейшего изучения

- Документация по языку Python: Floating-Point Arithmetic: Issues and Limitations. Доступно здесь: https://docs.python.org/tutorial/floatingpoint.html.
- Статья «What Every Computer Scientist Should Know About Floating-Point Arithmetic» Дэвида Голдберга (David Goldberg) (Computing Surveys, March 1991) уже стала классической. В ней представлена строгая методика использования арифметики с плавающей точкой. Настоятельно рекомендуется для изучения. Статья доступна здесь: https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html.
- *S. Oliveira and D. Stewart*. Writing Scientific Software: A Guide to Good Style, Cambridge University Press, Cambridge (2006).
- *N. J. Higham*. Accuracy and Stability of Numerical Algorithms, 2nd edn., Society for Industrial and Applied Mathematics, Philadelphia, PA (2002).
- Модуль Standard Library `decimal` поддерживает операции с десятичными числами с фиксированной точкой и с корректно округленными числами с плавающей точкой, но эти вычисления в общем случае выполняются намного медленнее, чем вычисления со встроенным типом данных `float`. Более подробную информацию см. здесь: https://docs.python.org/3/library/decimal.html.

10.1.6 Упражнения

Вопросы

B10.1.1. Десятичное представление некоторых действительных чисел не является однозначным. Например, докажите математически, что $0.\dot{9} \equiv 0.9999... \equiv 1$.

B10.1.2. Выражение $\sqrt{tg(\pi)} = 0$ строго определено математически, но тогда почему при выполнении следующего вычисления возникает критический сбой из-за ошибки в области действия модуля `math`?

```
In [x]: math.sqrt(math.tan(math.pi))
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-135-7bfdceeef434> in <module>()
----> 1 math.sqrt(math.tan(math.pi))
```

B10.1.3. Великая теорема Ферма утверждает, что не существует трех положительных целых чисел x, y и z, являющихся корнями уравнения $x^n + y^n - z^n = 0$ для любого целого $n > 2$. Объясните результат выполнения примера, противоречащего этой теореме:

```
In [x]: 844487.**5 + 1288439.**5 - 1318202.**5
Out[x]: 0.0
```

B10.1.4. Функции $f(x) = (1 - \cos^2 x)/x^2$ и $g(x) = \sin^2 x/x^2$ неразличимы с математической точки зрения, но при построении их графиков с использованием Python в области $-0/001 \leqslant x \leqslant 0.001$ наблюдается существенное различие. Объяснить причину этого различия.

B10.1.5. Как можно определить, представлено ли число с плавающей точкой значением `nan` или нет, без использования методов `math.isnan` и `numpy.isnan`?

B10.1.6. Определить и объяснить результаты вычисления следующих инструкций:

а) `1e1001 > 1e1000`
б) `1e350/1.e100 == 1e250`
в) `1e250 * 1.e-250 == 1e150 * 1.e-150`
г) `1e350 * 1.e-350 == 1e450 * 1.e-450`
д) `1 / 1e250 == 1e-250`
е) `1 / 1e350 == 1e-350`
ж) `1e450/1e350 != 1e450 * 1e-350`
з) `1e250/1e375 == 1e-125`
и) `1e35 / (1e1000 - 1e1000) == 1 / (1e1000 - 1e1000)`
к) `1e1001 > 1e1000 or 1e1001 < 1e1000`
л) `1e1001 > 1e1000 or 1e1001 <= 1e1000`

Задачи

310.1.1. Формула Герона для вычисления площади треугольника (используемая в примере П2.3):

$$A = \sqrt{s(s-a)(s-b)(s-c)}, \text{ где } s = 1/2(a+b+c),$$

неточна, если одна из сторон намного меньше двух других (для «игловидных» треугольников). Почему? Показать, что приведенная ниже модификация формулы Герона позволяет получить более точный результат для подобного случая: рассматривается треугольник со сторонами $(10^{-13}, 1, 1)$, площадь которого равна 5×10^{-14137}:

$$A = 1/4\sqrt{(a+(b+c))(c-(a-b))(c+(a-b))(a+(b-c))},$$

где стороны обозначены в определенном порядке, таком что $a \geqslant b \geqslant c$.

Что произойдет, если переписать сомножители в этой формуле так, чтобы исключить внутренние круглые скобки? Почему?

310.1.2. Написать функцию для определения машинного эпсилона для любого числового типа данных (`float`, `np.float128`, `int` и т. д.).

[137] Автор этой формулы – канадский математик Уильям Кэхэн (William Kahan), один из разработчиков международного стандарта IEEE-754, определяющего представление чисел с плавающей точкой.

10.2 Стабильность и обусловленность алгоритма

10.2.1 Стабильность алгоритма

Стабильностью алгоритма можно считать его способность обрабатывать погрешности приближенных вычислений во время работы алгоритма или при вводе данных. Подобные погрешности обычно возникают из-за экспериментальных неопределенностей (из-за несовершенства измерений, формирующих исходные данные) или из-за некоторого типа аппроксимаций значений с плавающей точкой, вводимых в вычисления алгоритмом, описанных в предыдущем разделе. Еще одним частым источником ошибок и погрешностей являются аппроксимации, позволяющие «дискретизировать» задачу: например, требуется представить значения непрерывной функции $y = f(x)$ на дискретной «сетке» точек $y_i = f(x_i)$. Алгоритм называют численно стабильным (устойчивым), если он не увеличивает погрешности такого рода, и нестабильным, если его применение приводит к росту погрешностей.

Пример П10.1. Рассмотрим дифференциальное уравнение

$$dy/dx = -\alpha y$$

при $\alpha > 0$ с учетом граничного условия $y(0) = 1$. Эту простую задачу можно решить аналитически:

$$y = e^{-\alpha x},$$

но предположим, что необходимо решить ее в численном виде. Самым простым является явный одношаговый метод Эйлера первого порядка точности: выбирается размер шага h, определяющий сетку значений x, т. е. $x_i = x_{i-1} + h$, затем выполняется приближенное вычисление соответствующих значений y по формуле:

$$yi = y_{i-1} + h|dy/dx|_{x_{i-1}} = y_{i-1} - h\alpha y_{i-1} = y_{i-1}(1 - \alpha h).$$

Возникает вопрос: какое значение необходимо выбрать для h? Малое значение h минимизирует погрешность, возникающую при аппроксимации, описанной выше, при которой значения y просто соединяются отрезками прямой[138], но если значение h слишком мало, то возникает погрешность с потерей значащих разрядов («катастрофическое взаимоуничтожение») из-за ограниченной точности представления чисел, используемых при вычислении[139].

Код в листинге 10.1 реализует явный одношаговый алгоритм Эйлера для решения приведенного выше дифференциального уравнения. Наибольшее значение h (здесь $h = \alpha/2 = 1$) явно делает алгоритм нестабильным (см. рис. 10.1).

---

[138] То есть это ряд Тейлора, усеченный в окрестности точки y_{i-1} по линейному члену h.

[139] В экстремальном случае, если выбранное значение h меньше машинного эпсилона, обычно приблизительно равного 2×10^{-16}, то получаем $x_i = x_{i-1}$, следовательно, сетка точек вообще отсутствует.

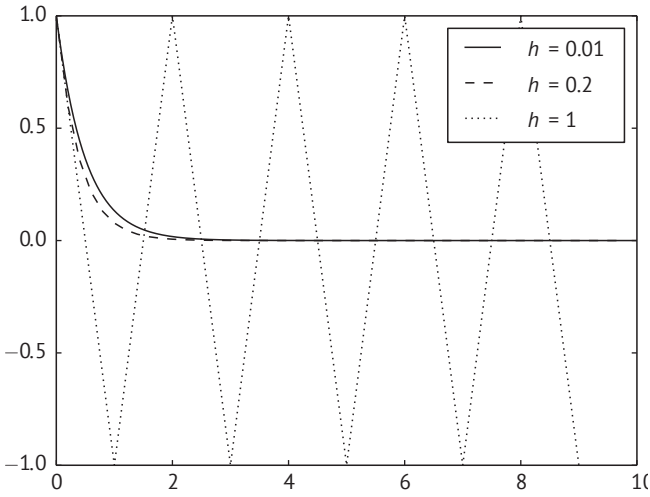


Рис. 10.1. Нестабильность явного одношагового решения Эйлера для дифференциального уравнения $dy/dx = -\alpha y$ при большом размере шага h

Листинг 10.1. Сравнение различных размеров шагов h для численного решения дифференциального уравнения $y' = -\alpha y$ явным одношаговым методом Эйлера

```
import numpy as np
import matplotlib.pyplot as plt

alpha, y0, xmax = 2, 1, 10

def euler_solve(h, n):
    """ Solve dy/dx = -alpha.y by forward Euler method for step size h."""
    """ Решение ДУ dy/dx = -alpha.y явным одношаговым методом Эйлера при размере шага h."""
    y = np.zeros(n)
    y[0] = y0
    for i in range(1, n):
        y[i] = (1 - alpha * h) * y[i-1]
    return y

def plot_solution(h):
    x = np.arange(0, xmax, h)
    y = euler_solve(h, len(x))
    plt.plot(x, y, label='$h={}$'.format(h))

for h in (0.01, 0.2, 1):
    plot_solution(h)

plt.legend()
plt.show()
```

Пример П10.2. Для интеграла

$$I_n = \int_0^1 x^n e^x dx \text{ при } n = 0, 1, 2, \ldots$$

предлагается рекурсивное отношение, полученное при интегрировании по частям:

$$I_n = [x^n e^x]_0^1 - n \int_0^1 x^n e^x dx = e - nI_{n-1},$$

завершающееся при $I_0 = e - 1$. Но этот алгоритм, применяемый «в направлении вперед», т. е. при увеличении значения n, является численно нестабильным, так как малые погрешности (такие как погрешности округления чисел с плавающей точкой) увеличиваются на каждом шаге вычислений: если погрешность для I_n равна ε_n, так что оцениваемое значение $I'_n = I_n + \varepsilon_n$, то

$$\varepsilon_n = I'_n - I_n = (e - nI'_{n-1}) - (e - nI_{n-1}) = n(I_{n-1} - I'_{n-1}) = -n\varepsilon_{n-1},$$

следовательно, $|\varepsilon_n| = n!\varepsilon_0$. Даже если значение погрешности в ε_0 мало, то в ε_n погрешность увеличивается, умножаясь на $n!$, т. е. результат может стать огромным.

В этом случае численно стабильным решением является применение рекурсии в обратном порядке, т. е. в сторону уменьшения n:

$$I_{n-1} = 1/n(e - I_n) \quad \Rightarrow \quad \varepsilon_{n-1} = -\varepsilon_n/n.$$

Таким образом, погрешности в I_n уменьшаются на каждом шаге рекурсии. Можно даже начать выполнение алгоритма при $I'_N = 0$, и при обеспечении достаточного количества шагов между N и требуемым n достигается сходимость к верному значению I_n.

Листинг 10.2. Сравнение стабильности алгоритма при вычислении интеграла $I_n = \int_0^1 x^n e^x dx$

```
# eg9-integral-stability.py
import numpy as np
import matplotlib.pyplot as plt

def Iforward(n):
    if n == 0:
        return np.e - 1
    return np.e - n * Iforward(n-1)

def Ibackward(n):
    if n >= 99:
        return 0
    return (np.e - Ibackward(n+1)) / (n+1)

N = 35
Iforward = [np.e - 1]
for n in range(1, N+1):
    Iforward.append(np.e - n * Iforward[n-1])
```

```
Ibackward = [0] * (N+1)
for n in range(N-1,-1,-1):
    Ibackward[n] = (np.e - Ibackward[n+1]) / (n+1)

n = range(N+1)
plt.plot(n, Iforward, label='Forward algorithm')
plt.plot(n, Ibackward, label='Backward algorithm')
plt.ylim(-0.5, 2)
plt.xlabel('$n$')
plt.ylabel('$I(n)$')
plt.legend()
plt.show()
```

На рис. 10.2 показано, что алгоритм, применяемый в прямом направлении, становится чрезвычайно нестабильным при $n > 16$ и начинает колебаться между весьма большими положительными и отрицательными значениями. В противоположность ему алгоритм, применяемый в обратном направлении, ведет себя корректно.

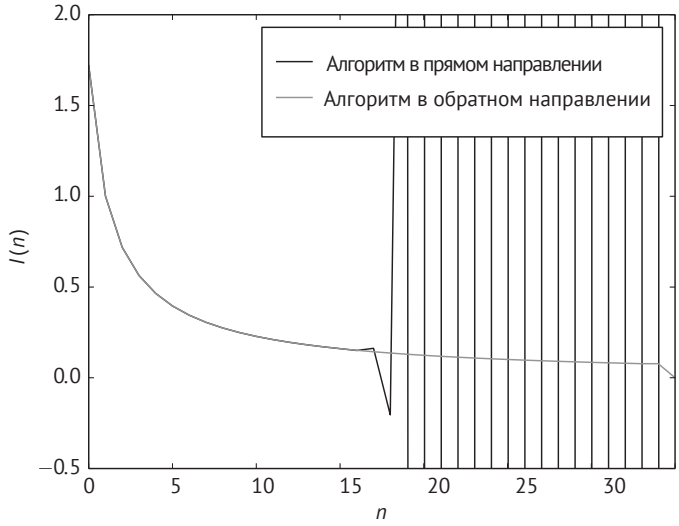


Рис. 10.2. Нестабильность явного одношагового алгоритма Эйлера, применяемого в прямом направлении для вычисления интеграла $I_n = \int_0^1 x^n e^x dx$

10.2.2 Хорошо обусловленные и плохо обусловленные задачи

В численном анализе существует еще одно различие между задачами, которые хорошо или плохо обусловлены. Хорошо обусловленной является задача, для которой малые относительные погрешности в исходных данных приводят к малым относительным погрешностям в решении. Плохо обусловленной яв-

ляется задача, для которой малые погрешности в исходных данных приводят к большим погрешностям в решении. Обусловленность (число обусловленности) является свойством самой задачи, а не алгоритма, и в этом заключается отличие от проблемы стабильности: вполне возможно использовать нестабильный алгоритм для решения хорошо обусловленной задачи и получить ошибочные результаты.

Пример П10.3. Рассмотрим две прямые, заданные уравнениями:

$$y = x,$$
$$y = mx + c.$$

Эти прямые пересекаются в точке $(x_\star, y_\star) = (c/(1 - m), c/(1 - m))$. Поиск точки пересечения представляет собой плохо обусловленную задачу при $m \approx 1$ (т. е. прямые почти параллельны).

Например, прямые $y = x$ и $y = (1.01)x + 2$ пересекаются в точке $(x_\star, y_\star) = (-200, -200)$. Если немного изменить значение m на величину $\delta m = 0.001$, т. е. $m' = m + \delta m = 1.011$, то точкой пересечения станет $(x'_\star, y'_\star) = (-181.8182, -181.8182)$. Таким образом, относительная погрешность $\delta m/m \approx 0.001$ в значении m привела к относительной погрешности результата $|(x'_\star - x_\star)/x_\star| \approx 0.091$, что почти в 100 раз больше.

Напротив, если прямые имеют значительно отличающиеся коэффициенты угла наклона, то задача является хорошо обусловленной. Например, примем $m = -1$ (перпендикулярные прямые), тогда точка их пересечения (1, 1) при том же небольшом изменении $m' = m + \delta m = -0.999$ становится точкой (1.0005, 1.0005), т. е. приводит к относительной погрешности 0.0005, которая действительно меньше относительной погрешности значения m.

Пример П10.4. Общеизвестно, что задача поиска корней многочлена плохо обусловлена. Одним из широко известных примеров является многочлен Уилкинсона:

$$P(x) = \prod_{i=1}^{20}(x - i) = (x - 1)(x - 2)\ldots(x - 20) =$$
$$= x^{20} - 210x^{19} + 20\,615x^{18} + \ldots + 2\,432\,902\,008\,176\,640\,000.$$

Вполне очевидно, что корнями являются числа 1, 2, …, 20. Но Уилкинсон показал, что небольшое уменьшение коэффициента при x^{19} с -210 до $-210 - 2^{-23} \approx -210.000000119209$ оказывает критическое воздействие на многие из корней, причем некоторые корни становятся комплексными. Например, корень $x = 20$ смещается к $x = 20.8$ – изменение на 4 % при весьма малом отклонении единственного коэффициента, всего лишь на одну миллиардную долю (также см. задачу 310.2.2).

10.2.3 Упражнения

Задачи

310.2.1. Самый простой (и наименее точный) способ вычисления первой производной функции просто использует определение самой производной:

$$f'(x) = \lim_{h \to 0} (f(x + h) - f(x) / h).$$

Если принять h как некоторое весьма малое (бесконечно малое) значение, то получим приближенную формулу:

$$f'(x) \approx (f(x + h) - f(x)) / h.$$

Если рассматривать функцию $f(x) = e^x$, то при каком значении h (при использовании арифметики с двойной точностью наиболее близкое к степени 10) получается самое точное приближение к $f'(1) = e$?

310.2.2. Использовать класс `Polynomial` из библиотеки NumPy (см. раздел 6.4) для генерации объекта, представляющего многочлен Уилкинсона, по его корням с доступной численной точностью. Затем найти корни этого представления многочлена Уилкинсона.

10.3 Методики программирования и разработка программного обеспечения

10.3.1 Общие замечания

Комментирование исходного кода

В этой книге я попытался комментировать примеры исходного кода и решения упражнений с наибольшей пользой. Это правильный подход даже для коротких скриптов, но эффективное использование комментариев – это совсем не простая задача. Ниже приводятся некоторые общие советы:

- в общем случае размещайте комментарии на отдельных строках, а не в конце строк кода (т. е. до или после строки комментируемого кода):

```
# Объем додекаэдра с длиной стороны a.
V = (15 + 7 * np.sqrt(5)) / 4 * a**3
```

Это лучше, чем:

```
V = (15 + 7 * np.sqrt(5)) / 4 * a**3 # Объем додекаэдра с длиной стороны a.
```

Более длинные комментарии должны быть законченными предложениями, начинающимися с заглавной (строчной) буквы (если это возможно, т. е. если первое слово не идентификатор, начинающийся с буквы нижнего регистра) и заканчивающимися точкой.

В этой книге используются комментарии в конце строк кода для объяснения особенностей синтаксиса, поэтому они, вероятнее всего, не нужны в «настоящем промышленном» исходном коде Python;

- необходимо не просто описывать, что делает код, но и объяснять, почему он это делает. Предположим, что некто читает ваш код и уже хорошо знает синтаксис языка. В таком случае

```
# Увеличение i на 10:
i += 10
```

является абсолютно бесполезным комментарием, ничего не добавляющим к смыслу строки кода, которую он намеревается объяснить. С другой стороны,

```
# Пропуск следующих 10 точек данных.
i += 10
```

по крайней мере обеспечивает некоторое понимание причины использования этой команды;

- необходимо поддерживать актуальность комментариев и соответствие их коду, который они описывают. Слишком часто встречаются случаи изменения исходного кода без соответствующего исправления комментариев. Это может привести к ситуации, в которой наличие неактуального комментария гораздо хуже, чем полное его отсутствие:

```
# Пропуск следующих 10 точек данных.
i += 20
```

Что здесь написано правильно? Правильный комментарий, объясняющий намерение программиста, а строка кода ошибочна, или строка кода была изменена по какой-либо причине, а комментарий остался неисправленным? Если в вашем коде весьма часто требуются подобные изменения, то следует рассмотреть вариант с определением отдельной дополнительной переменной, содержащей значение изменения для переменной `i`:

```
DATA_SKIP = 10
...
# Пропуск следующих DATA_SKIP точек данных.
i += DATA_SKIP
```

На практике некоторые программисты являются приверженцами минимизации количества комментариев при тщательном выборе осмысленных имен идентификаторов. Например, если переименовать индексную переменную, то, возможно, комментарий вообще не потребуется:

```
data_index += DATA_SKIP
```

- подробно описывайте определяемые функции с использованием строк документации docstring. В языке Python все функции имеют атрибут `__doc__`, которому присваивается строка docstring, записанная в определении функции (см. раздел 2.7.1). Строка документации docstring – это обычно текст на нескольких строках в тройных кавычках, представляющий объяснение того, что делает функция, какие аргументы принимает, а также объяснение смысла возвращаемого значения (или нескольких значений), если это необходимо. В интерактивной командной оболочке при вводе команды `help(`*имя_функции*`)` выводится более подробная информация об использовании указанной функции, включающая также и строку документации docstring.

Пример П10.5. Здесь показан пример правильно комментированной функции (вычисляющей объем тетраэдра).

Листинг 10.3. Функция, вычисляющая объем тетраэдра

```
# eg9-tetrahedron.py
import numpy as np

def tetrahedron_volume(vertexes=None, sides=None):
    """
    Return the volume of the tetrahedron with given vertexes or side lengths.
    If vertexes are given they must be in an array with shape (4, 3): the
    position vectors of the four vertexes in three dimensions; if the six sides are
    given, they must be an array of length 6. If both are given, the sides
    will be used in the calculation.
    Raises a ValueError if the vertexes do not form a tetrahedron (e.g.
    because they are coplanar, colinear or coincident).
    """
    """
    Возвращает объем тетраэдра при заданных вершинах или длинах сторон.
    Если заданы вершины, то они должны передаваться в массиве формы (4, 3):
    положения векторов четырех вершин в трех измерениях. Если заданы шесть сторон,
    то они должны передаваться в массиве длиной 6. Если определены оба аргумента,
    то при вычислении будут использоваться длины сторон.
    Генерирует исключение ValueError, если заданные вершины не образуют тетраэдр
    (например, если они лежат в одной плоскости, на одной прямой или совпадают).
    """

    # Этот метод реализует формулу Тартальи, используя определитель Кэли-Менгера:
    #           |0  1    1    1    1  |
    #           |1  0   s1^2 s2^2 s3^2|
    # 288 V^2 = |1 s1^2  0   s4^2 s5^2|
    #           |1 s2^2 s4^2  0   s6^2|
    #           |1 s3^2 s5^2 s6^2  0  |
    # где s1, s2, ..., s6 - длины сторон тетраэдра.
    # Внимание: этот алгоритм не проверялся на численную стабильность.

    # Индексы строк в массиве вершин соответствуют всем
    # возможным парам вершин.
    vertex_pair_indexes = np.array(((0, 1), (0, 2), (0, 3),
                                    (1, 2), (1, 3), (2, 3)))
```

```
    if sides is None:
        # Если длины сторон не заданы, то их необходимо вычислить по вершинам.
        vertexes = np.asarray(vertexes)                                    ❶
        if vertexes.shape != (4, 3):
            raise TypeError('vertexes must be a numpy array with shape (4, 3)')
        # Получить все квадраты всех длин сторон из разностей между шестью
        # различными парами положений вершин.
        vertex1, vertex2 = vertex_pair_indexes.T
        sides_squared = np.sum((vertexes[vertex1] - vertexes[vertex2])**2, axis=-1)
    else:
        # Проверка: длины сторон представлены как корректный массив.
        # Вычислить квадраты длин сторон.
        sides = np.asarray(sides)
        if sides.shape != (6,):
            raise TypeError('sides must be an array with shape (6,)')
        sides_squared = sides**2

# Создать определитель Кэли-Менгера.
M = np.zeros((5, 5))
# Заполнить верхнюю треугольную область матрицы.
M[0, 1:] = 1
# Элементы, содержащие квадраты длин сторон, можно проиндексировать, используя индексы пар
# вершин (сравнить с определителем, показанным выше).
M[tuple(zip(*(vertex_pair_indexes + 1)))] = sides_squared

# Эта матрица симметрична, поэтому можно заполнить нижнюю треугольную область,
# применив операцию транспонирования.
M = M + M.T

# Вычислить определитель и проверить, получено ли положительное значение (отрицательное
# значение или ноль означает, что вершины не образуют тетраэдр).
det = np.linalg.det(M)
if det <= 0:
    raise ValueError('Provided vertexes do not form a tetrahedron')
return np.sqrt(det / 288)
```

❶ Метод `np.asarray` используется для преобразования `vertexes` в массив NumPy, если `vertexes` пока еще не является массивом NumPy. Это позволяет функции работать с любым совместимым объектом (например, со списком списков (`list` of `lists`)).

Магические числа

С точки зрения стиля программирования магическое число (magic number) – это любое постоянное числовое значение, используемое непосредственно в программе, без присваивания ему осмысленного имени переменной. Вообще говоря, отказ от использования магических чисел позволяет писать более удобный для чтения, гибкий и легко сопровождаемый код, несмотря на некоторое увеличение объема кода. Если число появляется в коде без какого-либо описания (возможно, за исключением совсем очевидных случаев, таких как инициализация переменной нулем или увеличение значения на единицу), то необходимо рассмотреть возможность присваивания этого числа переменной. Если число встречается несколько раз, то почти всегда лучше сделать так, как показано в примере П10.6.

Пример П10.6. Программа в листинге 10.4 оценивает вероятность выпадения различных сумм при бросках двух игральных кубиков.

Листинг 10.4. Программа имитации бросков двух игральных кубиков, содержащая магические числа

```
import random

# Инициализация словаря бросков нулевым значением счетчика для каждого возможного исхода.
rolls = dict.fromkeys(range(2, 13), 0)

# Имитация: бросок двух кубиков 100 000 раз.
for j in range(100000):
    roll_total = random.randint(1, 6) + random.randint(1, 6)
    rolls[roll_total] += 1

# Вывод результатов имитации.
for i in range(2, 13):
    P = rolls[i] / 100000
    print(f'P({i}) = {P:.5f}')
```

В этой программе встречается несколько магических чисел: количество очков на каждой грани кубика случайно выбирается из диапазона целых чисел 1–6, суммарное количество каждого из возможных исходов 2–12 сохраняется в словаре `rolls`, ключи которого генерируются при вызове функции `range(2, 13)`, и количество имитируемых бросков жестко закодировано как число 100 000.

Следует отметить, что если потребуется, например, изменить количество бросков, то необходимо отредактировать исходный код в трех местах: в цикле имитации, в комментарии к циклу имитации и в инструкции вычисления вероятности. Сопровождение и редактирование подобного кода в более длинной программе, вероятнее всего, потребует больше времени и повысит вероятность внесения ошибок (опечаток).

Небольшое размышление о том, как присвоить эти постоянные магические числа переменным, к тому же обеспечивает способ сделать этот код более гибким, как показано ниже. Общепринятой (но не обязательной) практикой, как и во многих других языках, является запись имени константы буквами в верхнем регистре, что позволяет быстро отличать константы от переменных.

Листинг 10.5. Код имитации бросков двух игральных кубиков, улучшенный посредством использования именованных констант

```
import random

NDICE = 2
NFACES_PER_DIE = 6
NROLLS = 100000

# Вычисление всех возможных сумм очков при бросках.
min_roll, max_roll = NDICE, NDICE * NFACES_PER_DIE
roll_total_range = range(min_roll, max_roll+1)
```

```
# Инициализация словаря бросков нулевыми значениями счетчиков для каждого возможного исхода.
rolls = dict.fromkeys(roll_total_range, 0)

# Имитация: бросок NDICE кубиков NROLLS раз.
for j in range(NROLLS):
    roll_total = 0
    for i in range(NDICE):
        roll_total += random.randint(1, NFACES_PER_DIE)
    rolls[roll_total] += 1

# Вывод результатов имитации.
for i in roll_total_range:
    P = rolls[i] / NROLLS
    print(f'P({i}) = {P:.5f}')
```

С помощью этой программы можно имитировать произвольное количество бросков любого количества кубиков с любым числом сторон. Для этого нужно изменить только лишь в одном месте кода значения `NDICE`, `NFACES_PER_DIE` и `NROLLS`.

Краткое руководство по стилю написания кода Python

Официально рекомендованные соглашения по стилю кодирования для языка Python представлены в документе под названием PEP8 (документ доступен здесь: www.python.org/dev/peps/pep-0008). Известно, что эти соглашения не всегда соблюдаются в полном объеме, и все же программисты на Python в целом согласны, что они в высшей мере способствуют хорошему пониманию и удобному сопровождению исходного кода. Главное внимание уделено логической целостности, удобству чтения и минимизации вероятности возникновения ошибок и опечаток, которые трудно обнаружить. Ниже перечислены основные положения этих соглашений:

- использование четырех пробелов для смещения каждого уровня кода (табуляции не используются никогда)[140];
- в инструкциях присваивания знак = необходимо выделять пробелами с обеих сторон, например `a = 10`, но не `a=10`;
- максимальная длина строки должна равняться 79 символам, а если необходимо разделить строку кода на несколько строк, то рекомендуется соблюдение следующих правил:
  - более предпочтительным является неявное продолжение строки, заключенной в круглые скобки, нежели явное использование символа обратного слеша \ (см. раздел 2.3.1);
  - в арифметических выражениях следует разделять строку после бинарных операторов так, чтобы новая строка начиналась после оператора;
  - во всех случаях, когда это возможно, рекомендуется размещать строки кода так, чтобы выражения в скобках на разных строках были выровнены соответствующим образом.

[140] Хороший текстовый редактор можно сконфигурировать так, чтобы он автоматически заменял символы табуляции на заданное количество пробелов.

Ниже показан пример неудачно выбранного стиля:

```
lengthy_calculation = margin*margin_px + (border*border_px\
                            + padding*padding_px)
```

который можно улучшить следующим образом:

```
lengthy_calculation = (margin*margin_px + (border*border_px +
                                           padding*padding_px))
```

- определения функций самого высокого уровня и классов выделяются двумя пустыми строками. Внутри класса определения выделяются одной пустой строкой;
- для исходного кода рекомендуется использовать кодировку UTF-8 (в Python 3 эта кодировка принята по умолчанию);
- следует избегать импорта с использованием символов шаблонов (`from foo import *`), так как подобная операция вводит (и, возможно, перезаписывает) имена из импортируемого модуля в локальное пространство имен (см. вопрос B2.2.5);
- необходимо отделять операторы от операндов одним символом пробела, за исключением тех случаев, когда в одном выражении объединены операторы с различными приоритетами, например `x = x + 5`, но `r2 = x**2 + y**2`;
- не рекомендуется использовать пробелы, окружающие знак равенства, в именованных аргументах, например вызов функции записывается как `foo(b=4.5)`, но не `foo(b = 4.5)`;
- следует избегать записи нескольких инструкций на одной строке с разделением их точкой с запятой, например вместо `a = 1; b = 2` лучше написать `a, b = 1, 2` (см. раздел 4.3.1);
- функции, модули и пакеты должны иметь короткие имена, записанные символами (буквами) нижнего регистра. Символ подчеркивания при необходимости можно использовать в именах функций и модулей, но следует избегать его использования в именах пакетов;
- имена классов должны записываться в «стиле верблюда» (CamelCase) с первой буквой в верхнем регистре, также известном под названием CapWords, например AminoAcid, но не amino_acid (см. раздел 4.6.2);
- рекомендуется определять для констант[141] имена, состоящие только из букв верхнего регистра с символами подчеркивания, разделяющими слова, например `MAX_LINE_LENGTH`.

10.3.2 Текстовые редакторы

Выбор текстового редактора для написания исходного кода – это до известной степени личное дело каждого, но большинство программистов предпочитают редакторы с подсветкой синтаксиса и возможностью определения макрокоманд для быстрого выполнения повторяющихся задач. Наиболее распространенные варианты выбора перечислены ниже.

[141] Следует отметить, что в Python в действительности нет констант в прямом смысле, как, например, в C.

- Visual Studio Code – широко известный, бесплатный редактор с открытым исходным кодом, разработанный Microsoft для Windows, Linux и macOS.
- Sublime Text – коммерческий редактор с однопользовательской лицензией и возможностью временного бесплатного (пробного) использования.
- Vim – широко распространенный кросс-платформенный редактор с клавиатурными командами, с относительно высокой сложностью обучения, но обладающий мощными функциональными возможностями. Более простой редактор vi устанавливается почти во всех операционных системах Linux и Unix.
- Emacs – широко распространенная альтернатива Vim.
- Notepad++ – бесплатный редактор только для Windows.
- SciTE – быстрый и простой в использовании редактор исходного кода.
- Atom – еще один бесплатный кросс-платформенный редактор с открытым исходным кодом.

Кроме простых редакторов, существуют также полнофункциональные интегрированные среды разработки (Integrated Development Environment – IDE), предоставляющие еще и возможности отладки, выполнения кода, интеллектуального дополнения кода и доступа к сервисам операционных систем. Некоторые из таких IDE перечислены ниже.

- Eclipse с подключаемым модулем PyDev – широко распространенная бесплатная IDE (www.eclipse.org/ide/).
- JupyterLab – IDE на основе браузера с открытым исходным кодом для научных исследований в области обработки данных и создания прочих приложений на Python (https://jupyter.org/).
- PyCharm – кросс-платформенная IDE в коммерческой и бесплатной версиях (www.jetbrains.com/pycharm/).
- PythonAnywhere – онлайновая среда Python с бесплатным и оплачиваемым вариантами доступа (www.pythonanywhere.com/).
- Spyder – IDE с открытым исходным кодом для научного программирования на Python с интегрированными в нее библиотеками NumPy, SciPy, Matplotlib, а также интерактивной оболочкой IPython (www.spyder-ide.org/).

10.3.3 Системы управления версиями

Без правильно организованного управления крупные программные проекты (на практике это проект, состоящий из более одного файла исходного кода) зачастую быстро превращаются в запутанный клубок измененных версий, экспериментального кода, функциональных возможностей, добавляемых «с листа», и временных файлов. Управление изменениями в файлах, составляющих программный проект, называется системой управления версиями (version control или revision control).

В простейшем варианте система управления версиями может включать простое сохранение исходного кода в нескольких параллельных каталогах (папках), пронумерованных в хронологическом порядке в соответствии с развитием программного проекта. Этот подход может работать, но если небольшое изменение в большом объеме кода приводит к появлению новой версии, этот способ становится неэффективным (огромный объем неизмененного

кода полностью копируется в новый каталог). Если новая версия создается только при значительных изменениях в коде, то возникает вероятность массового беспорядка в коде, написанном между версиями.

Для устранения подобных проблем существует несколько доступных пакетов систем управления версиями программного обеспечения (ПО), некоторые из них перечислены ниже. Большинство систем управления версиями работают как отдельные независимые приложения в операционной системе и могут вызываться из командной строки или использоваться с графическим пользовательским интерфейсом. Некоторые преимущества систем управления версиями перечислены ниже:

- над одним проектом могут одновременно работать несколько разработчиков;
- разделение на ветви (branching): возможность параллельной разработки двух версий ПО одновременно, например для тестирования новых функциональных возможностей;
- система тегов или меток (tagging или labeling): способ обозначений (ссылок) мгновенных снимков проекта в определенном состоянии;
- возможность отката (rollback) файла в проекте к предыдущей версии;
- клонирование (cloning): средства распространения (дистрибуции) программного проекта вместе с полной хронологией его изменений;
- некоторые системы управления версиями объединены с онлайновыми репозиториями для хранения и совместного использования исходного кода. Наиболее известным является GitHub (https://github.com/).

Здесь не описывается подробно работа с системами управления версиями (синтаксис команд индивидуален в каждой системе, и существуют подробные руководства, документация и даже целые книги по таким системам). Можно лишь дать краткие рекомендации по выбору системы управления версиями.

Git – наиболее распространенная гибкая система управления версиями. Git работает на распределенной (децентрализованной) основе, позволяя разработчикам принять участие в проекте без совместного использования общей сетевой среды или централизованного репозитория эталонного кода. Проекты с открытым исходным кодом можно хранить бесплатно на онлайновых сервисах, таких как GitHub и Bitbucket (https://git-scm.com/).

Mercurial – другая распределенная система управления версиями (https://www.mercurial-scm.org/).

Subversion (SVN) – централизованный вариант с бесплатным (для проектов с открытым исходным кодом) хранением на сайте SourceForge (https://sourceforge.net/). При постоянном росте популярности Git система SVN уже не так широко распространена, как раньше (https://subversion.apache.org/).

10.3.4 Модульное тестирование

Модульное тестирование – это способ проверки работоспособности ПО при сосредоточении внимания на тестировании отдельных модулей исходного кода. Для Python как языка объектно-ориентированного программирования это обычно означает, что отдельные классы (иногда даже отдельные функции) проверяются на наборах тестовых (испытательных) данных, при этом некоторые

данные могут быть преднамеренно некорректными или искаженными. Цель таких тестов – обнаружение всех ошибок, приводящих к критически неверной интерпретации данных. Комплект модульных тестов также служит в качестве документированного и проверяемого подтверждения того, что код делает именно то, что от него требуется. В соответствии с некоторыми парадигмами разработки исходного кода модульные тесты создаются раньше, чем сам исходный код[142].

Важным аспектом модульного тестирования является «регрессионное тестирование»: оно предоставляет средства, позволяющие убедиться в том, что последующие изменения в коде (возможно, с добавлением новой функциональности) не нарушают его работоспособность – обновленный код должен успешно проходить те же модульные тесты, которые ранее прошел предыдущий вариант кода.

Модульное тестирование исходного кода в небольшом проекте требует определенной дисциплины. Сами по себе тесты – это машинный код (и, возможно, связанные с ним необходимые данные), поэтому необходимо тщательно продумать их написание. Разработка правильных модульных тестов часто заставляет программиста более глубоко задуматься о реализации исходного кода и даже может помочь определить возможные ошибки перед началом разработки.

Собственная рабочая среда (фреймворк) модульного тестирования Python основана на модуле `unittest` – простое приложение рассматривается ниже в примере П10.7. Другим вариантом является внешняя рабочая среда `pytest`, которая активно поддерживается и распространяется.

Пример П10.7. Предположим, что необходимо написать функцию преобразования температуры в градусах Фаренгейта, Цельсия и в кельвинах (соответственно обозначенных буквами `'F'`, `'C'` и `'K'`). Написать исходный код для шести нужных формул не составляет труда, но, возможно, потребуется тщательная обработка пары условий, возникающих при использовании этой функции: физически невозможные температуры (< 0 K) или буквы, не соответствующие допустимым единицам измерения `'F'`, `'C'` или `'K'`.

Эта функция сначала выполняет преобразование в кельвины, затем в требуемые единицы измерения. Если по какой-то причине входные и выходные единицы измерения заданы одинаковыми, то необходимо вернуть без изменений исходное значение. Функция `convert_temperature` определена в файле *temperature_utils.py*.

Листинг 10.6. Функция для преобразования различных единиц измерения температуры

```
# temperature_utils.py

def convert_temperature(value, from_unit, to_unit):
    """ Convert and return the temperature value from from_unit to to_unit. """
    """ Преобразование и возврат значения температуры: из from_unit в to_unit. """
```

[142] В особенности при использовании так называемого «экстремального» программирования.

```
    # Словарь функций преобразования из различных единиц измерения *в* K.
    toK = {'K': lambda val: val,
           'C': lambda val: val + 273.15,
           'F': lambda val: (val + 459.67)*5/9,
          }
    # Словарь функций преобразования *из* K в другие единицы измерения.
    fromK = {'K': lambda val: val,
             'C': lambda val: val - 273.15,
             'F': lambda val: val*9/5 - 459.67,
            }

    # Сначала выполняется преобразование из from_unit в K.
    try:
        T = toK[from_unit](value)
    except KeyError:
        raise ValueError('Unrecognized temperature unit: {}'.format(from_unit))

    if T < 0:
        raise ValueError('Invalid temperature: {} {} is less than 0 K'
                                 .format(value, from_unit))

    if from_unit == to_unit:
        # Преобразование не требуется.
        return value

    # Теперь выполняется преобразование из K в to_unit и возвращается полученное значение.
    try:
        return fromK[to_unit](T)
    except KeyError:
        raise ValueError('Unrecognized temperature unit: {}'.format(to_unit))
```

Для использования модуля `unittest` с целью выполнения модульных тестов для функции `convert_temperature` напишем новый скрипт Python, определяющий класс `TestTemperatureConversion`, производный от базового класса `TestCase`. В этом классе определены методы, работающие как тесты для функции `convert_temperature`. Тестовые методы должны вызывать одну из функций с утверждениями (assertion function) базового класса для проверки того, что значение, возвращаемое функцией `convert_temperature`, соответствует ожидаемому. Например:

```
self.assertEqual(<returned value>, <expected value>)
```

возвращает значение `True`, если оба значения абсолютно равны, в противном случае возвращается значение `False`. Другая функция с утверждением предназначена для проверки факта генерации заданного исключения (например, при передаче некорректных аргументов) или того, что возвращено значение `True`, `False`, `None` и т. п. В листинге 10.7 приведен код модульного теста для функции `convert_temperature`.

Листинг 10.7. Модульные тесты для функции преобразования температуры

```
from temperature_utils import convert_temperature
import unittest

class TestTemperatureConversion(unittest.TestCase):

    def test_invalid(self):
        """
        There's no such temperature as -280 C, so convert_temperature should
        raise a ValueError.
        """
        """
        Такие значения температуры, как -280 C, недопустимы, поэтому функция
        convert_temperature должна генерировать исключение ValueError.
        """

        self.assertRaises(ValueError, convert_temperature, -280, 'C', 'F')          ❶

    def test_valid(self):
        """ A series of valid temperature conversions to test. """
        """ Последовательность корректных преобразований температур для теста. """

        test_cases = [((273.16, 'K',), (0.01, 'C')),
                      ((-40, 'C'), (-40, 'F')),
                      ((450, 'F'), (505.3722222222222, 'K'))]

        for test_case in test_cases:
            ((from_val, from_unit), (to_val, to_unit)) = test_case
            result = convert_temperature(from_val, from_unit, to_unit)
            self.assertAlmostEqual(to_val, result)                                  ❷

    def test_no_conversion(self):
        """
        Ensure that if the from-units and to-units are the same the
        temperature is returned exactly as it was passed and not converted
        to and from kelvins, which may cause loss of precision.
        """
        """
        Проверка: если единицы from-units и to-units одинаковы, то возвращается
        в точности то значение температуры, которое было передано, без преобразования
        в кельвины и обратно, потому что при этом может произойти потеря значащих разрядов.
        """

        T = 56.67
        result = convert_temperature(T, 'C', 'C')
        self.assertTrue(result is T)                                                ❸

    def test_bad_units(self):
        """ Check that ValueError is raised if invalid units are passed. """
        """ Проверка: исключение ValueError генерируется, если переданы некорректные единицы. """

        self.assertRaises(ValueError, convert_temperature, 0, 'C', 'R')
        self.assertRaises(ValueError, convert_temperature, 0, 'N', 'K')

unittest.main()
```

❶ Метод `assertRaises` проверяет, сгенерировано ли заданное исключение методом `convert_temperature`. Необходимые аргументы передаются после самого объекта метода.
❷ Здесь необходим метод `assertAlmostEqual`, потому что при использовании арифметики с плавающей точкой вероятна потеря значащих разрядов из-за ошибок округления.
❸ Здесь используется метод `assertTrue`, чтобы убедиться в том, что возвращаемое значение температуры является тем же объектом, который был передан, без преобразования в кельвины и обратно.

Выполнение этого скрипта показывает, что функция `convert_temperature` успешно прошла все модульные тесты:

```
$ python eg9-temperature -conversion -unittest.py
...
----------------------------------------------------------------------
Ran 4 tests in 0.000s

OK
```

10.3.5 Материалы для дальнейшего изучения

- *F. Brooks*. The Mythical Man-Month, Addison-Wesley, Boston, MA (1975, 1995). Почти легендарная монография о разработке программного обеспечения, объясняющая, почему «добавление рабочих ресурсов на последней стадии проекта разработки ПО приводит к срыву его сроков»[143].
- *J. Loeliger and M. McCullough*. Version Control with Git, O'Reilly, Sebastopol, CA (2012).
- *S. McConnell*. Code Complete: A Practical Handbook of Software Construction, Microsoft Press, Redmond, WA (2004)[144].
- *A. Hunt and D. Thomas*. The Pragmatic Programmer, Addison-Wesley, Boston, MA (1999).

[143] Существует перевод этой книги на русский язык: *Фредерик Брукс.* Мифический человеко-месяц, или Как создаются программные системы. Сер.: Профессионально. СПб.: Символ-Плюс, 2000.

[144] Существует перевод этой книги на русский язык: *Стив Макконнелл.* Совершенный код. СПб.: Питер, 2005.

Приложение А. Решения

Здесь приведены ответы на некоторые вопросы из раздела «Упражнения» в конце глав. Дополнительные упражнения и их решения см. здесь: https://scipython.com.

В2.2.5. Этот вопрос демонстрирует опасность импорта с использованием символов шаблонов: значение переменной `e = 2` заменяется определением постоянной `e` из модуля `math`. Поэтому выражение `d ** e` вместо возведения в квадрат с результатом 8 дает результат возведения в степень e = 2.71828….

В2.2.7. Используются операторы Python:

```
>>> a = 2
>>> b = 6
>>> 3 * (a**3*b - a*b**3) % 7
3
>>> a = 3
>>> b = 5
>>> 3 * (a**3*b - a*b**3) % 7
1
```

В2.2.8. Толщина листа бумаги, сложенного пополам n раз, равна $2^n t$, поэтому потребуется $2^n t \geqslant d \Rightarrow n_{\min} = \lceil \log_2(d/t) \rceil$:

```
>>> d = 384_400 * 1.e3         # Расстояние до Луны, м
>>> t = 1.e-4                  # Толщина листа бумаги, м
>>> math.log(d / t, 2)         # Логарифм по основанию 2.
41.805745474760016
```

Следовательно, лист бумаги необходимо сложить пополам 42 раза, чтобы его толщина стала равна расстоянию до Луны ($\lceil x \rceil$ обозначает округление с избытком (в большую сторону) значения x: наименьшее целое число, не меньшее x).

В2.2.10. Оператор ^ не возводит число в заданную степень (это делает оператор **). Это оператор «побитовое исключающее или», поэтому в двоичной записи `10^2` выглядит как `1010 xor 0010 = 1000`, т. е. 8 в десятичной системе счисления.

В2.3.1. Необходимо выполнить вырезание из строки `s = 'seehemewe'`, как показано ниже (в некоторых случаях возможны и другие решения):

а) `s[:3]`
б) `s[3:5]`
в) `s[5:7]`

г) s[7:]
д) s[3:6]
е) s[5:2:-1]
ж) s[-2::-3]

B2.3.2. Нужно просто выполнить вырезание строки в обратном направлении и сравнить результат с исходной строкой:

```
>>> s = 'banana'
>>> s == s[::-1]
False
>>> s = 'deified'
>>> s == s[::-1]
True
```

B2.3.5. Это некорректный способ проверки на равенство строки s значению 'ham' или 'eggs'. Выражение ('eggs' or 'ham') логическое, в нем оба аргумента, являющиеся непустыми строками, вычисляются как значения True. Выражение вычисляется по укороченной схеме до первого эквивалента значения True, и возвращается соответствующий операнд (см. раздел 2.2.4), т. е. выражение ('eggs' or 'ham') возвращает 'eggs'. Поскольку s, несомненно, содержит строку ‘eggs’, сравнение на равенство возвращает значение True. Но если поменять местами операнды, то логический оператор or снова выполнит вычисление по укороченной схеме до первого эквивалента значения True, которым теперь является 'ham', и вернет эту строку. Сравнение на равенство с s завершается неудачей, и возвращается результат False.

Существуют два корректных способа проверки того, что s является одной из двух и более строк:

```
>>> s = 'eggs'
>>> s == 'ham' or s == 'eggs'
True
>>> s in ('ham', 'eggs')
True
```

(Более подробное описание синтаксиса второй команды см. в разделе 2.4.2.)

B2.4.2. Проблема заключается в том, что enumerate по умолчанию возвращает индексы и элементы переданного массива, а индексы нумеруются, начиная с 0. Передаваемый массив является срезом P[1:] = [5, 0, 2], а enumerate, в свою очередь, генерирует кортежи (0, 5), (1, 0) и (2, 2). Но для вычисляемой здесь производной необходимы индексы из исходного списка P, т. е. (1, 5), (2, 0) и (3, 2). Здесь возможны два варианта: передача дополнительного аргумента start=1 в enumerate или прибавление 1 к индексу по умолчанию:

```
>>> P = [4, 5, 0, 2]
>>> dPdx = []
>>> for i, c in enumerate(P[1:], start=1):
... dPdx.append(i*c)
```

```
>>> dPdx
[5, 0, 6]

>>> P = [4, 5, 0, 2]
>>> dPdx = []
>>> for i, c in enumerate(P[1:]):
... dPdx.append((i+1)*c)
>>> dPdx
[5, 0, 6]
```

B2.4.3. Одно из возможных решений:

```
>>> scores = [87, 75, 75, 50, 32, 32]
>>> ranks = []
>>> for score in scores:
...     ranks.append(scores.index(score) + 1)
...
>>> ranks
[1, 2, 2, 4, 5, 5]
```

B2.4.4. Ниже показано вычисление значения π до 10 знаков после (десятичной) запятой.

```
>>> import math

>>> pi = 0
>>> for k in range(20):
...     pi += pow(-3, -k) / (2*k+1)              ❶
...
>>> pi *= math.sqrt(12)
>>> print('pi = ', pi)
pi = 3.1415926535714034
>>> print('error = ', abs(pi - math.pi))
error =  1.8389734179891093e-11
```

❶ Встроенная функция `pow(x, j)` равнозначна выражению `(x)**j`.

B2.4.5. Результатом выражения `any(x) and not all(x)` является `True`, если хотя бы один элемент в `x` равнозначен `True`, но не все элементы:

```
>>> x1, x2, x3 = [False, False], [1, 2, 3, 4], [1, 2, 3, 0]
>>> any(x1) and not all(x1)
False
>>> any(x2) and not all(x2)
False
>>> any(x3) and not all(x3)
True
```

B2.4.6. Необходимо вспомнить о том, что оператор * распаковывает кортеж в список позиционных аргументов, передаваемых в функцию. Поэтому если `z = zip(a, b)` является (итерируемой) последовательностью: `(a0, b0)`, `(a1, b1)`, `(a2, b2)`, ..., то распаковка этой последовательности при вызове `zip(*z)` равнозначна вызову `zip` с этими кортежами как аргументами:

```
zip((a0, b0), (a1, b1), (a2, b2), ...)
```

Метод `zip` по очереди берет первый и второй элементы из каждого кортежа, воспроизводя исходные последовательности:

```
(a0, a1, a2, ...), (b0, b1, b2, ...)
```

B2.4.7. Необходимо просто применить метод `zip` к спискам значений солнечных часов и имен месяцев, затем отсортировать полученный список кортежей в обратном порядке:

```
>>> months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun',
...           'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec']
>>> sun = [44.7, 65.4, 101.7, 148.3, 170.9, 171.4,
...        176.7, 186.1, 133.9, 105.4, 59.6, 45.8]
>>> for s, m in sorted(zip(sun, months), reverse=True):
...     print('{}: {:.1f} hrs'.format(m, s))
...
Aug: 186.1 hrs
Jul: 176.7 hrs
Jun: 171.4 hrs
May: 170.9 hrs
Apr: 148.3 hrs
Sep: 133.9 hrs
Oct: 105.4 hrs
Mar: 101.7 hrs
Feb: 65.4 hrs
Nov: 59.6 hrs
Dec: 45.8 hrs
Jan: 44.7 hrs
```

B2.5.1. Необходимо нормализовать список:

```
>>> a = [2, 4, 10, 6, 8, 4]
>>> amin, amax = min(a), max(a)
>>> for i, val in enumerate(a):
...     a[i] = (val - amin) / (amax - amin)
...
>>> a
[0.0, 0.25, 1.0, 0.5, 0.75, 0.25]
```

B2.5.2. Приведенный ниже код вычисляет постоянную Гаусса до 14 знаков после запятой.

```
>>> import math
>>> tol = 1.e-14
>>> an, bn = 1., math.sqrt(2)
>>> while abs(an - bn) > tol:
...     an, bn = (an + bn) / 2, math.sqrt(an * bn)
...
>>> print('G = {:.14f}'.format(1/an))
G = 0.83462684167407
```

B2.5.3. Приведенный ниже код вычисляет первые 100 чисел «Fizzbuzz».

```
nmax = 100
for n in range(1, nmax + 1):
    message = ''
    if not n % 3:
        message = 'fizz'
    if not n % 5:
       message += 'buzz'
print(message or n)                ❶
```

❶ Обратите внимание: если `n` не делится ни на 3, ни на 5, то сообщением `message` будет пустая строка, которая вычисляется как `False` в этом логическом выражении, поэтому вместо сообщения выводится значение `n`.

B2.5.4. Здесь показано одно из решений, использующее в качестве примера `stoich = 'C8H18'`:

Листинг А.1. Структурная формула алкана с прямой (неразветвленной) цепью

```
# qn2-5-c-alkane-a.py

stoich = 'C8H18'

fragments = stoich.split('H')
nC = int(fragments [0][1:])
nH = int(fragments[1])
if nH != 2*nC + 2:
    print('{} is not an alkane!'.format(stoich))
else:
    print('H3C', end='')
    for i in range(nC -2):
    print('-CH2', end='')
    print('-CH3')
```

Вывод результата:

```
H3C-CH2-CH2-CH2-CH2-CH2-CH2-CH3
```

B2.7.1. Только варианты б) и е) ведут себя так, как предполагалось:

а) При отсутствии явной инструкции `return` функция `line` возвращает значение `None`. Поскольку `None` невозможно присоединить к строке, возникает ошибка:

```
my_sum = '\n'.join([' 56', ' +44', line , ' 100', line])
...
TypeError: sequence item 2: expected str instance, NoneType found
```

б) Код работает правильно.

в) Функция `line` возвращает строку, как и требуется, но она не вызывается как `line()` – без скобок `line` ссылается непосредственно на объект функции, который невозможно присоединить к строке, поэтому возникает ошибка:

```
my_sum = '\n'.join([' 56', ' +44', line , ' 100', line])
...
TypeError: sequence item 2: expected str instance, function found
```

г) В этом коде ошибка не возникает, но выводится строка представления (объекта) функции вместо строки, возвращаемой при правильном вызове этой функции:

```
   56
  +44
<function line at 0x103d9e9e0 >
  100
<function line at 0x103d9e9e0 >
```

д) Этот код выводит неожидаемое значение `None`:

```
   56
  +44
-----
None
  100
-----
None
```

Это происходит потому, что инструкция `print(line())` вызывает функцию `line`, которая выводит строку дефисов, но, кроме того, выводит свое возвращаемое значение (т. е. `None`, так как явно не возвращает что-либо другое).

е) Код работает правильно.

B2.7.2. Проблема возникает внутри функции `add_interest`:

```
def add_interest(balance, rate):
    balance += balance * rate / 100
```

Здесь создается новый объект типа `float` `balance`, который является локальным в этой функции и не зависит от исходного объекта `balance`. При выходе из функции локальный объект `balance` уничтожается, а исходный объект `balance` не обновляется никогда. Один из способов устранения этой проблемы – явный возврат из функции обновленного значения баланса:

```
>>> balance = 100
>>> def add_interest(balance, rate):
...     balance += balance * rate / 100
...     return balance
...
>>> for year in range(4):
```

```
...     balance = add_interest(balance, 5)
...     print('Balance after year {}: ${:.2f}'.format(year + 1, balance))
...
Balance after year 1: $105.00
Balance after year 2: $110.25
Balance after year 3: $115.76
Balance after year 4: $121.55
```

B2.7.3. Проблема возникает из-за того, что функция `digit_sum` не возвращает сумму цифр `n`, которая была вычислена. При отсутствии явной инструкции `return` функция Python возвращает значение `None`, но это недопустимый объект для использования при вычислении остатка от деления, поэтому генерируется исключение `TypeError`.

Проблему легко устранить, добавив нужную инструкцию `return dsum`:

```
def digit_sum(n):
    """ Find and return the sum of the digits of integer n. """
    """ Вычисление и возврат суммы цифр целого числа n. """

    s_digits = list(str(n))
    dsum = 0
    for s_digit in s_digits:
        dsum += int(s_digit)
    return dsum

def is_harshad(n):
    return not n % digit_sum(n)
```

Теперь код работает, как предполагается:

```
>>> is_harshad(21)
True
```

B2.7.4. Этот код выводит:

```
[1, 2, 'a']
[1, 2, 'a']
```

потому что новый список создается только один раз при определении функции, и это тот список, который добавляется и возвращается при каждом вызове функции. Таким образом, `lst1` и `lst2` – это один и тот же объект, в чем можно убедиться:

```
print(lst1 is lst2)
True
```

B4.1.1. Правильный практический прием – сохранять размер блока `try` настолько малым, насколько это возможно, чтобы предотвратить захват лишних исключений, отличающихся от требуемого. Например, в примере П4.5 предполагается чтение файла после его открытия в одном блоке `try`:

```
try:
    fi = open(filename, 'r')
    lines = fi.readlines()
except IOError:
...
```

В этом случае имеем две потенциальные ошибки, которые могут привести к генерации исключения `IOError`: критический сбой при открытии файла и критический сбой при чтении его строк. Ветвь `except` предназначена для обработки только первого случая, но будет также выполняться и во втором случае, хотя более правильным решением была бы отдельная обработка критических сбоев при чтении строк (или другой вариант: оставить этот случай без обработки, тогда программа просто прекратит выполнение).

B4.1.2. Проблема в блоке `finally` в примере П4.5 заключается в том, что инструкции в этом блоке начинают выполняться до того, как происходит возврат из функции. Если бы строка

```
print('    Done with file {}'.format(filename))
```

была перемещена в позицию после блока `try`, то она бы не выполнялась, если было бы сгенерировано исключение `IOError` (потому что возврат из функции в вызвавший ее блок кода произошел бы до перехода к этой инструкции `print`).

B4.2.1. Это легко сделать с помощью метода `set`. Вызов из строки `s`:

```
set(s.lower()) >= set('abcdefghijklmnopqrstuvwxyz')
```

возвращает результат `True`, если строка является панграммой. Например:

```
>>> s = 'The quick brown fox jumps over the lazy dog'
>>> set(s.lower()) >= set('abcdefghijklmnopqrstuvwxyz')
True
>>> s = 'The quick brown fox jumped over the lazy dog'
>>> set(s.lower()) >= set('abcdefghijklmnopqrstuvwxyz')
False
```

B4.2.2. Эту функцию можно использовать для удаления повторяющихся элементов из упорядоченного списка:

```
>>> def remove_dupes(l):
...     return sorted(set(l))
...
>>> remove_dupes([1, 1, 2, 3, 4, 4, 4, 5, 7, 8, 8, 9])
[1, 2, 3, 4, 5, 7, 8, 9]
```

Следует отметить, что хотя множества не имеют какого-либо определенного порядка, они являются итерируемыми объектами и могут передаваться во встроенный метод `sorted()`, который возвращает список (`list`).

B4.2.3. Необходимо выполнить следующие команды в интерактивном интерпретаторе Python:

```
>>> set('hellohellohello')
{'h', 'o', 'l', 'e'}
>>> set(['hellohellohello'])
{'hellohellohello'}
>>> set(('hellohellohello'))
{'h', 'o', 'l', 'e'}
>>> set(('hellohellohello',))
{'hellohellohello'}
>>> set(('hello', 'hello', 'hello'))
{'hello'}
>>> set(('hello', ('hello', 'hello')))
{'hello', ('hello', 'hello')}
>>> set(('hello', ['hello', 'hello']))
Traceback (most recent call last):
  File "<stdin >", line 1, in <module >
TypeError: unhashable type: 'list'
```

Обратите особое внимание на различие между инициализацией множества `set` списком (`list`) объектов и попыткой добавления списка как объекта в множество.

B4.2.4. Обратите внимание: инструкция

```
>> a |= {2, 3, 4, 5}
```

не изменяет `frozenset`, а создает новый объект, объединяя старое множество и `set {2, 3, 4, 5}`. (То же самое наблюдалось, когда после создания объекта `int i` присваивание `i = i + 1` заново связывало метку (имя) `i` с новым целочисленным объектом со значением `i + 1`, а не изменяло значение неизменяемого объекта типа `int`, ранее связанного с `i`).

B4.2.5. Приведенный ниже фрагмент кода должен быть добавлен после определения `text` – не забудьте импортировать `defaultdict` из модуля `collections`.

```
words_by_length = defaultdict(list)
for word in text.split():
    words_by_length[len(word)].append(word)

for length in sorted(words_by_length.keys()):
    print(f'{length}: {words_by_length[length]}')
```

Вывод результата:

```
1: ['a']
2: ['on', 'in', 'to']
3: ['and', 'ago', 'our', 'new', 'and', 'the', 'all', 'men', 'are']
4: ['four', 'this', 'that']
5: ['score', 'seven', 'years', 'forth', 'equal']
6: ['nation']
```

```
7: ['fathers', 'brought', 'liberty', 'created']
9: ['continent', 'conceived', 'dedicated']
11: ['proposition']
```

B4.3.1. Генератор списков

```
>>> flist = [lambda x, i=i: x**i for i in range(4)]
```

создает такой же список анонимных функций, как в примере П4.11.

Следует отметить, что необходимо передавать каждое значение `i` в лямбда-функцию явно, иначе правила замыкания Python приведут к тому, что каждая лямбда-функция станет равнозначной выражению `x**3` (3 – это последнее значение `i` в цикле).

B4.3.2. Приведенный в вопросе фрагмент кода выводит первые `nmax+1` строк треугольника Паскаля:

```
[1]
[1, 1]
[1, 2, 1]
[1, 3, 3, 1]
[1, 4, 6, 4, 1]
[1, 5, 10, 10, 5, 1]
```

В присваивании генератора списков

```
x = [([0] + x)[i] + (x + [0])[i] for i in range(n+1)]
```

добавляются элементы двух списков. Эти два списка формируются из списка, представляющего предыдущую строку: в первом случае добавлением `0` в начало списка, во втором случае добавлением `0` в конец списка. При таком подходе сумма вычисляется по соседним парам чисел, а начальное и конечное числа остаются неизменными. Например, если `x` содержит `[1, 3, 3, 1]`, то следующая строка формируется с помощью суммирования элементов в списках

```
[0, 1, 3, 3, 1]
[1, 3, 3, 1, 0]
```

что в итоге дает требуемую строку `[1, 4, 6, 4, 1]`.

B4.3.3.

а) Индекс элементов `a`, использующий элементы `b`:

```
>>> [a[x] for x in b]
['E', 'C', 'G', 'B', 'F', 'A', 'D']
```

б) Индекс элементов `a`, использующий отсортированные элементы `b`. В этом случае возвращаемый список – это просто (копия) `a`:

```
>>> [a[x] for x in sorted(b)]
['A', 'B', 'C', 'D', 'E', 'F', 'G']
```

в) Индекс элементов a, использующий элементы b, проиндексированный по элементам b(!):

```
>>> [a[b[x]] for x in b]
['F', 'G', 'D', 'C', 'A', 'E', 'B']
```

г) Каждый элемент b связывается с соответствующим элементом a в последовательности кортежей `[(4, 'A'), (2, 'B'), (6, 'C'), ...]`, которая затем сортируется, – этот метод используется для возврата элементов a, соответствующих упорядоченным элементам b.

```
>>> [x for (y,x) in sorted(zip(b,a))]
['F', 'D', 'B', 'G', 'A', 'E', 'C']
```

B4.3.4. Для возвращения отсортированного списка пар (ключ, значение) из словаря:

```
>>> d = {'five': 5, 'one': 1, 'four': 4, 'two': 2, 'three': 3}
>>> d
{'four': 4, 'one': 1, 'five': 5, 'two': 2, 'three': 3}
>>> sorted([(k, v) for k, v in d.items()])
[('five', 5), ('four', 4), ('one', 1), ('three', 3), ('two', 2)]
```

Обратите внимание: сортировка списка кортежей (ключ, значение) требует, чтобы все ключи имели типы данных, которые можно осмысленно упорядочить. Такой подход не работает, например, если в ключах объединены целые значения и строки, так как (в Python 3) не существует определенного порядка для совместной сортировки этих типов – будет сгенерировано исключение `TypeError: unorderable types: int() < str()`.

Для сортировки по значению можно отсортировать список кортежей (значение, ключ), но для сохранения возвращаемого списка с парами (ключ, значение) использовать следующий код:

```
>>> sorted([(k, v) for k, v in d.items()], key=lambda item: item[1])
[('one', 1), ('two', 2), ('three', 3), ('four', 4), ('five', 5)]
```

Аргумент `key` для метода `sorted` определяет, как интерпретировать каждый элемент в списке для упорядочения: здесь требуется упорядочение по второму элементу (`item[1]`) в каждом кортеже (`k, v`) для упорядочения по значению.

B4.3.5. В приведенном ниже коде шифруется (и расшифровывается) номер телефона, хранящийся как строка с использованием метода «прыжок через 5».

```
''.join(['5987604321'[int(i)] if i.isdigit() else '-' for i in '555-867-5309'])
```

В4.3.6. Одно из решений: создать кортеж из двух элементов для каждого элемента в сортируемом списке: первый элемент кортежа – логическое значение, обозначающее, является ли элемент списка значением `None` или нет, второй элемент кортежа – само значение из списка. При сортировке этих кортежей первый элемент дает результат `False` для всех чисел (которые могут сравниваться как второй элемент) и `True` для всех значений `None`. Поскольку `False` при сравнении всегда считается значением, «меньшим, чем» `True`, здесь нет необходимости сравнивать различные типы:

```
In [x]: lst = [4, 2, 6, None, 0, 8, None, 3]
In [x]: lst.sort(key=lambda e: (e is None, e))
In [x]: lst
Out[x]: [0, 2, 3, 4, 6, 8, None, None]
```

В4.3.7. Здесь можно предложить два решения:

а) использование выражения присваивания для кортежа:

```
>>> t = 1, 1
>>> while (t := (t[0] + t[1], t[0])) < (5000, 0):
...     continue
...
>>> t[0]
6765
```

б) в условие цикла `while` включается выражение присваивания, содержащее вызов метода `input`:

```
>>> while (s := input("> ").lower()) != "exit":
...     print(s)
...
> hello
hello
> bye
bye
> quit
quit
> :q
:q
> exit
>>>
```

В6.1.1. Массив `np.ndarray` – это класс NumPy для представления многомерных массивов в Python. В этой книге мы часто ссылаемся на экземпляры данного класса просто как на объекты массива. `np.array` – это функция, создающая такие объекты из передаваемых в нее аргументов (обычно это последовательность значений).

В6.1.2. Для создания двумерного массива в метод `array()` необходимо передать последовательность последовательностей как один аргумент, а в приведенном в вопросе вызове вместо этого передаются как отдельные аргументы три последовательности. Правильный вызов:

```
>>> np.array( ((1, 0, 0), (0, 1, 0), (0, 0, 1)) , dtype=float)
```

B6.1.3. `np.array([0, 0, 0])` создает одномерный массив с тремя элементами. `np.array([[0, 0, 0]])` создает двумерный массив 1×3 (т. е. `a[0]` – это одномерный массив, созданный в первом примере).

B6.1.4. Изменение типа массива с помощью непосредственной установки `dtype` не изменяет данные на уровне байтов, а влияет только на интерпретацию данных как чисел, строк и т. д. Известно, что байтовое представление нуля одинаково для целых чисел (`int64`) и чисел с плавающей точкой (`float64`), поэтому результат установки типа `dtype` будет ожидаемым. Но 8 байт, представляющие `1.0`, преобразуются в целое значение `4602678819172646912`. Для корректного преобразования типа данных необходимо использовать метод `astype()`, который возвращает новый массив (с собственными данными):

```
In [x]: a = np.ones((3,))
In [x]: a
Out[x]: array([ 1., 1., 1.])

In [x]: a.astype('int')
In [x]: a
Out[x]: array([1, 1, 1])
```

B6.1.5. Индексация и вырезание массива NumPy:

а) `a[1, 0, 3]`
б) `a[0, 2, :]` (или просто `a[0, 2]`)
в) `a[2, ...]` (или `a[2, :, :]`, или `a[2]`)
г) `a[:, 1, :2]`
д) `a[2, :, :1:-1]` («в третьем блоке для каждой строки взять (в обратном порядке) элементы из всех столбцов, кроме первого»).
е) `a[:, ::-1, 0]` («для каждого блока пройти по строкам в обратном направлении и взять элемент из первого столбца каждой строки»).
ж) Определить три индексных массива 2×2 для блоков, строк и столбцов, позиционирующих заданные элементы, как показано ниже:

```
ia = np.array([[0, 0], [2, 2]])
ja = np.array([[0, 0], [3, 3]])
ka = np.array([[0, 3], [0, 3]])
```

`a[ia, ja, ka]` возвращает требуемый результат.

B6.1.6. Например:

```
In [x]: a = np.array([0, -1, 4.5, 0.5, -0.2, 1.1])
In [x]: a[abs(a) <= 1]
Out[x]: array([ 0. , -1. , 0.5, -0.2])
```

B6.1.7. В приведенном ниже коде:

```
In [x]: a, b = -2.00231930436153, -2.0023193043615
In [x]: np.isclose(a, b, atol=1.e-14)
Out[x]: True
```

`np.isclose()` возвращает `True`, поскольку, несмотря на то что абсолютная разность между этими двумя числами больше 10^{-14}, она все же (значительно) меньше, чем `rtol * abs(b)`, т. е. доли относительной разности, принятой по умолчанию. Чтобы получить ожидаемое поведение, необходимо установить для `rtol` значение 0:

```
In [x]: np.isclose(-2.00231930436153, -2.0023193043615, atol=1.e-14, rtol=0)
Out[x]: False
```

B6.1.8. Здесь различное поведение наблюдается из-за ограниченной точности хранения действительных чисел: числа с плавающей точкой двойной точности представлены только 15 разрядами, поэтому в пределах такой ограниченной точности при сравнении двух заданных чисел они определяются как равные:

```
In [x]: 3.1415926535897932 - 3.141592653589793
Out[x]: 0.0
```

B6.1.9. Например:

```
In [x]: N = 5
In [x]: Nsq = N**2
In [x]: np.allclose(np.sort(magic_square.flatten()),
                    np.linspace(1, Nsq, Nsq).astype(int))
Out[x]: True

In [x]: Nsum = N * (N**2 + 1) // 2
In [x]: np.allclose(np.sum(magic_square, axis=0), Nsum)
Out[x]: True

In [x]: np.allclose(np.sum(magic_square, axis=1), Nsum)
Out[x]: True

In [x]: n.allclose(np.diag(magic_square), Nsum)
Out[x]: True

In [x]: n.allclose(np.diag(np.fliplr(magic_square)), Nsum)                    ❶
Out[x]: True
```

❶ Метод `np.fliplr` «переворачивает» содержимое массива в направлении слева направо. Еще один способ получения подобной «другой» диагонали – использование `a.ravel()[N-1:-N+1:N-1]`.

B6.1.10. Приведенная ниже инструкция определяет, является последовательность `a` возрастающей или нет:

```
np.all(np.diff(a) > 0)
```

B6.1.11. В первом случае создается один объект требуемого типа `dtype` и умножается на скаляр (обычный тип `int` Python). Python выполняет «приведение типа с повышением» для возврата результата типа `dtype`, который может содержать это значение:

```
In [x]: x = np.uint8(250)
In [x]: type(x*2)
Out[x]: numpy.int64
```

Но поскольку для `ndarray` жестко задан размер в байтах, для него нельзя выполнить приведение типа с повышением таким способом: его собственный `dtype` имеет приоритет над типом скаляра, на который умножается массив, поэтому умножение выполняется по модулю 256.

Сравните с результатом умножения двух скалярных значений одинакового типа `dtype`:

```
In [x]: np.uint8(250) * np.uint8(2)
Out[x]: 244             # Тип np.uint8.
```

(Возможно, будет выведено предупреждающее сообщение: `RuntimeWarning: overflow encountered in ubyte_scalars`.)

B6.4.1. Метод `deriv` класса `Polynomial` возвращает объект `Polynomial` (в рассматриваемом здесь случае с одним элементом – коэффициентом x^0, равным 18). Этот объект не равен объекту целого числа со значением 18.

B6.4.2. Используйте объект `numpy.polynomial.Polynomial`:

```
In [x]: p1 = Polynomial([-11, 1, 1])
In [x]: p2 = Polynomial([-7, 1, 1])
In [x]: p = p1**2 + p2**2
In [x]: dp = p.deriv()              # Первая производная.
In [x]: stationary_points = dp.roots()
In [x]: ddp = dp.deriv()            # Вторая производная.
In [x]: minima = stationary_points[ddp(stationary_points) > 0]
In [x]: maxima = stationary_points[ddp(stationary_points) < 0]
In [x]: inflections = stationary_points[np.isclose(ddp(stationary_points),0)]
In [x]: print(np.array((minima, p(minima))).T)
[[-3.54138127  8.        ]
 [ 2.54138127  8.        ]]
In [x]: print(np.array((maxima, p(maxima))).T)
[[  -0.5  ,   179.125]]
In [x]: print(np.array((inflections, p(inflections))).T)
[]
```

Таким образом, функция имеет два минимума:

$f(-3.54138127) = 8,$
$f(2.54138127) = 8$

и один максимум:

$f(-0.5) = 179.125$

при отсутствии точек перегиба/волнообразности.

B6.5.1. Без излишнего усложнения:

```
In [x]: pauli_matrices = np.array((
                                   ((0, 1), (1, 0)),
                                   ((0, -1j), (1j, 0)),
                                   ((1, 0), (0, -1))
                                  ))
In [x]: I2 = np.eye(2)
In [x]: for sigma in pauli_matrices:
...:        print(np.allclose(sigma.T.conj().dot(sigma), I2))
True
True
True
```

B6.5.2. Код в листинге А.2 подбирает коэффициенты заданного квадратного уравнения. Следует отметить, что это линейная подгонка методом наименьших квадратов, даже притом, что сама функция является нелинейной во времени, потому что она линейная по отношению к коэффициентам.

Листинг А.2. Подгонка методом наименьших квадратов к функции $x = x_0 + v_0t + 1/2gt^2$

```
# qn6-9-b-quadratic-fit-a.py
import numpy as np
import matplotlib.pyplot as plt
Polynomial = np.polynomial.Polynomial

x = np.array([1.3, 6.0, 20.2, 43.9, 77.0, 119.6, 171.7, 233.2, 304.2,
              384.7, 474.7, 574.1, 683.0, 801.3, 929.2, 1066.4, 1213.2,
              1369.4, 1535.1, 1710.3, 1894.9])
dt, n = 0.1, len(x)
tmax = dt * (n-1)
t = np.linspace(0, tmax, n)

A = np.vstack((np.ones(n), t, t**2)).T
coefs, resid, _, _ = np.linalg.lstsq(A, x)

# Начальное положение (см) и скорость (см/с), ускорение свободного падения (м/с2).
x0, v0, g = coefs[0], coefs[1], coefs[2] * 2 / 100

print('x0 = {:.2f} cm, v0 = {:.2f} cm.s-1, g = {:.2f} m.s-2'.format(x0, v0, g))

xfit = Polynomial(coefs)(t)
plt.plot(t, x, 'ko')
plt.plot(t, xfit, 'r')
plt.xlabel('Time (sec)')
plt.ylabel('Distance (cm)')
plt.show()
```

Результат подгонки к заданной функции показан на рис. А.1.

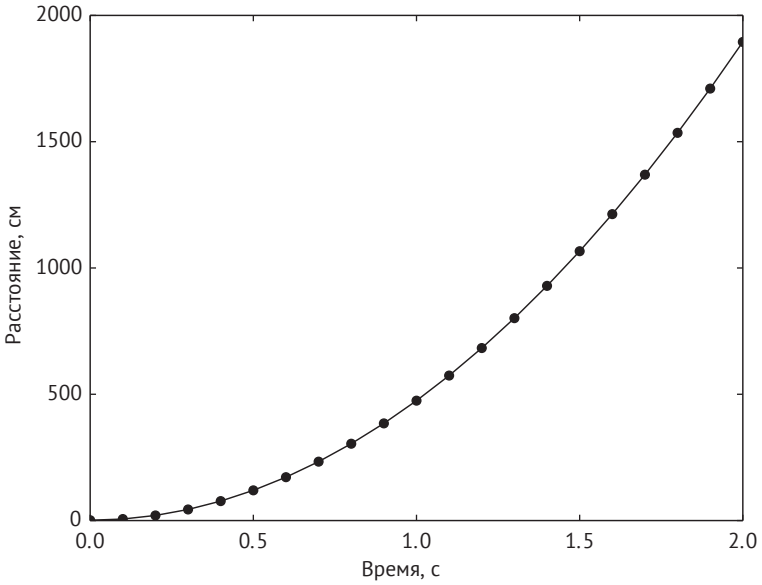


Рис. А.1. Подгонка методом наименьших квадратов к функции $x = x_0 + v_0t + 1/2gt^2$

В6.6.1. В первом случае

```
In [x]: a = np.array([6, 6, 6, 7, 7, 7, 7, 7, 7])
In [x]: a[np.random.randint(len(a), size=5)]
array([7, 7, 7, 6, 7])            # (Как пример).
```

выполняются случайные выборки из массива `a` с заменой: для каждого выбираемого элемента вероятность выбора 6 равна 1/3, а вероятность выбора 7 равна 2/3.

Во втором случае

```
In [x]: np.random.randint(6, 8, 5)
array([6, 6, 7, 7, 7])        # (Как пример).
```

числа выбираются из последовательности [6, 7] на равных условиях, поэтому вероятности выбора каждого числа равны 1/2.

В6.6.2. Функция `np.random.randint` выполняет равновероятную выборку из полуоткрытого интервала `[low, high)`, поэтому для получения поведения, равнозначного поведению метода `np.random.random_integers`, в примере П6.18 необходим следующий код:

```
In [x]: a, b, n = 0.5, 3.5, 4
In [x]: a + (b - a) * (np.random.randint(1, n + 1, size=10) - 1) / (n - 1)
Out[x]: array([ 0.5,  1.5,  0.5,  3.5,  1.5,  3.5,  2.5,  0.5,  1.5,  1.5])
```

B6.6.3. Вероятность выигрыша в одном из вариантов:

$$\binom{70}{5}\binom{25}{1} = \frac{70\cdot 69\cdot 68\cdot 67\cdot 66}{1\cdot 2\cdot 3\cdot 4\cdot 5}\cdot 25 = 302\,575\,350.$$

Исходный код для выбора пяти случайных чисел от 1 до 70 и одного числа от 1 до 25:

```
In [x]: (sorted(np.random.choice(np.arange(1, 71), 5, replace=False)),
         np.random.randint(25) + 1)
([23, 45, 51, 52, 67], 11)
```

B6.6.4. Здесь приводится более общее решение предложенной задачи. Определяется распределение опечаток в книге из биномиального распределения с использованием метода `np.random.binomial` и подсчитывается, на скольких страницах имеется более q опечаток. Для сравнения с распределением Пуассона по числу опечаток на странице X необходимо вычислить

$$\Pr(X >= q) = 1 - \Pr(X < q) = 1 - (\Pr(X = 0) + \Pr(X = 1) + \ldots + \Pr(X = q - 1)):$$

Листинг А.3. Вычисление вероятности нахождения q и более опечаток на заданной странице книги

```
# qn6-7-d-misprints-a.py
import numpy as np

n, m = 500, 400
q = 2
ntrials = 100
errors_per_page = np.random.binomial(m, 1/n, (ntrials , n))
av_ge_q = np.sum(errors_per_page >=q) / n / ntrials
print('Probability of {} or more misprints on a given page'.format(q))
print('Result from {} trials using binomial distribution: {:.6f}'
             .format(ntrials, av_ge_q))

# Теперь вычисляется то же самое количество с использованием распределения Пуассона:
# Pr(X>=q) = 1 - exp(-lam)[1 + lam + lam^2/2! + ... + lam^(q-1}/(q-1)!]
lam = m/n
poisson = 1
term = 1
for k in range(1, q):
    term *= lam/k
    poisson += term
poisson = 1 - np.exp(-lam) * poisson
print('Result from Poisson distribution: {:.6f}'.format(poisson))
```

Пример вывода результата:

```
Probability of 2 or more misprints on a given page
Result from 100 trials using binomial distribution: 0.190200
Result from Poisson distribution: 0.191208
```

B6.7.1. Для этих двух методов вычисления дискретного преобразования Фурье (ДПФ) можно измерить время их выполнения, используя магическую функцию IPython `%timeit`:

```
In [x]: import numpy as np
In [x]: n = 512
In [x]: # Функция ввода - это просто случайные числа.
In [x]: f = np.random.rand(n)

In [x]: # Время выполнения алгоритма NumPy ДПФ (алгоритм Кули-Тьюки).
In [x]: %timeit np.fft.fft(f)
100000 loops, best of 3: 13.1 us per loop

In [x]: # Теперь вычисляется ДПФ прямым суммированием.
In [x]: k = np.arange(n)
In [x]: m = k.reshape((n, 1))
In [x]: w = np.exp(-2j * np.pi * m * k / n)
In [x]: %timeit np.dot(w, f)
1000 loops, best of 3: 354 us per loop

In [x]: # Проверка совпадения результатов, полученных этими двумя методами.
In [x]: ftfast = np.fft.fft(f)
In [x]: ftslow = np.dot(w, f)
In [x]: np.allclose(ftfast, ftslow)
Out[x]: True
```

Здесь можно видеть, что алгоритм Кули–Тьюки почти в 30 раз быстрее, чем метод прямого суммирования. В действительности этот алгоритм характеризуется сложностью вычислений $O(n \log n)$ по сравнению со сложностью $O(n^2)$ метода прямого суммирования.

B8.1.1. Необходимо просто заменить строку

```
for rec in constants[-10:]:
```

на строку

```
for rec in constants[constants['rel_unc'] > 0][:10]:
```

а также спецификатор формата в строке вывода на `':g'` (так как значения неопределенностей меньше, чем 1 на миллион). Константа с наиболее точным известным значением – g-фактор электрона.

```
1.74797e-07 ppm: electron g factor = -2.00232
1.79792e-07 ppm: electron mag. mom. to Bohr magneton ratio = -1.00116
1.90811e-06 ppm: hertz-hartree relationship = 1.51983e-16 E_h
1.91096e-06 ppm: Rydberg constant times hc in eV = 13.6057 eV
...
```

B8.1.2. Вычисление $N/V = p/k_B T$ при заданных условиях можно выполнить, используя только константы из модуля `scipy.constants`:

```
In [x]: scipy.constants.atm / scipy.constants.k / scipy.constants.zero_Celsius
Out[x]: 2.686780501003883e+25
```

Это постоянная Лошмидта, определяемая стандартами 2010 CODATA и включенная в модуль `scipy.constants` (более подробно см. документацию):

```
In [x]: from scipy import constants
In [x]: constants.value('Loschmidt constant (273.15 K, 101.325 kPa)')
Out[x]: 2.6867805e+25
```

B8.2.1. При численном интегрировании подразумевается результат 3:

```
In [x]: from scipy.integrate import quad
In [x]: import numpy as np
In [x]: func = lambda x: np.floor(x) - 2*np.floor(x/2)
In [x]: quad(func, 0, 6)
Out[x]: (2.999964948683555,0.0009520766614606472)
```

B8.2.2. В приведенных ниже фрагментах кода подразумевается выполнение следующих инструкций импорта:

```
In [x]: import numpy as np
In [x]: from scipy.integrate import quad
```

а)
```
In [x]: f1 = lambda x: x**4 * (1 - x)**4/(1 + x**2)

In [x]: quad(f1, 0, 1)
Out[x]: (0.0012644892673496185, 1.1126990906558069e-14)

In [x]: 22/7 - np.pi
Out[x]: 0.0012644892673496777
```

б)
```
In [x]: f2 = lambda x: x**3/(np.exp(x) - 1)

In [x]: quad(f2, 0, np.inf)
Out[x]: (6.49393940226683, 2.628470028924825e-09)

In [x]: np.pi**4 / 15
Out[x]: 6.493939402266828
```

в)
```
In [x]: f3 = lambda x: x**-x

In [x]: quad(f3, 0, 1)
Out[x]: (1.2912859970626633, 3.668398917966442e-11)

In [x]: np.sum(n**-n for n in range(1, 20))
Out[x]: 1.2912859970626636
```

г)
```
In [x]: from scipy.misc import factorial
In [x]: f4 = lambda x, p: np.log(1/x)**p
In [x]: for p in range(10):
 ...:        print(quad(f4, 0, 1, args=(p,))[0], factorial(p))
 ...:
```

```
    1.0 1.0
    0.9999999999999999 1.0
    1.9999999999999991 2.0
    6.000000000000064 6.0
    24.000000000000014 24.0
    119.9999999999327 120.0
    719.9999999989705 720.0
    5039.99999945767 5040.0
    40320.00000363255 40320.0
    362880.00027390465 362880.0
```

д)
```
In [x]: from scipy.special import i0
In [x]: z = np.linspace(0, 2, 100)
In [x]: y1 = 2 * np.pi * i0(z)
In [x]: f5 = lambda theta, z: np.exp(z*np.cos(theta))
In [x]: y2 = np.array([quad(f5, 0, 2*np.pi, args=(zz,))[0] for zz in z])
In [x]: np.max(abs(y2-y1))
Out[x]: 2.1863399979338283e-11
```

В8.2.3. Для приближенного вычисления значения π с помощью интегрирования постоянной функции $f(x, y) = 4$ в четверти круга с единичным радиусом в квадранте $x > 0, y > 0$:

```
In [x]: from scipy.integrate import dblquad
In [x]: dblquad(lambda y, x: 4, 0, 1, lambda x: 0, lambda x: np.sqrt(1 - x**2))
Out[x]: (3.1415926535897922, 3.533564552071766e-10)
```

В8.2.4. Вычисляемый интеграл:

$$\int_0^1 \int_0^{2\pi} r \, \mathrm{d}\theta \, \mathrm{d}r = \pi.$$

Обратите внимание: внутренний интеграл вычисляется по θ, а внешний – по r. Следовательно, при вызове метода `dblquad` должна вызываться функция $f(r, \theta) = r$ как `lambda theta, r: r` (обратите особое внимание на порядок аргументов).

```
In [x]: dblquad(lambda theta, r: r, 0, 1, lambda r: 0, lambda r: 2*np.pi)
Out[x]: (3.141592653589793, 3.487868498008632e-14)
```

Другой вариант с изменением порядка интегрирования:

```
dblquad(lambda r, theta: r, 0, 2*np.pi, lambda theta: 0, lambda theta: 1)
(3.141592653589793, 3.487868498008632e-14)
```

В8.4.1. Необходимо переписать уравнение в следующем виде:

$$f(x) = x + 1 + (x - 3)^{-3} = 0.$$

Для этой функции легко построить график, а искомые корни можно ограничить интервалами (−2, −0.5) и (0, 2.99) (избегая сингулярности в точке $x = 3$).

```
In [x]: f = lambda x: x + 1 + (x-3)**-3
In [x]: brentq(f, -2, -0.5), brentq(f, 0, 2.99)
Out[x]: ( -0.984188231211512, 2.3303684533047426)
```

В8.4.2. Некоторые примеры поиска корней уравнений, для которых не подходит алгоритм Ньютона–Рафсона, и другие способы их решения.

а)
```
In [x]: newton(lambda x: x**3 - 5*x, 1, lambda x: 3*x**2 - 5)
...
RuntimeError: Failed to converge after 50 iterations, value is 1.0
```

Здесь алгоритм Ньютона–Рафсона входит в бесконечный цикл перебора повторяющихся значений x:

$$x_0 = 1 \;:\; x_1 = x_0 - f(x_0)/f'(x_0) = -1,$$
$$x_1 = -1 \;:\; x_2 = x_1 - f(x_1)/f'(x_1) = 1,$$
$$x_2 = 1 \;:\; x_3 = x_2 - f(x_2)/f'(x_2) = -1.$$
$$\ldots$$

Выбор других начальных точек обеспечивает сходимость к корню. Даже весьма малое смещение от $x = 0$ обеспечит сходимость:

```
In [x]: newton(lambda x: x**3 - 5*x, 1.0001, lambda x: 3*x**2 - 5)
Out[x]: 2.23606797749979
In [x]: newton(lambda x: x**3 - 5*x, 1.1, lambda x: 3*x**2 - 5)
Out[x]: -2.23606797749979
In [x]: newton(lambda x: x**3 - 5*x, 0.5, lambda x: 3*x**2 - 5)
Out[x]: 0.0
```

б)
```
In [x]: f, fp = lambda x: x**3 - 3*x + 1, lambda x: 3*x**2 - 3
In [x]: newton(f, 1, fp)
Out[x]: 1.0
In [x]: f(1.0)
Out[x]: -1
```

Алгоритм сходится, но не к корню. К сожалению, градиентом этой функции является ноль в выбранной начальной точке, поэтому возникающая здесь ошибка округления не приводит к генерации исключения `ZeroDivisionError`. Чтобы найти истинные корни, необходимо выбрать другие начальные точки, например $f'(x0) \neq 0$, или воспользоваться другим методом после ограничения интервалов искомых корней и исследования графика функции:

```
In [x]: brentq(f, -0.5, 0.5), brentq(f, -2, -1.5), brentq(f, 1, 2)
Out[x]: (0.34729635533386066, -1.879385241571423, 1.532088886237956)
```

в) Функция $f(x) = 2 - x^5$ на графике имеет плоское плато в окрестности точки $f(0) = 2$, поэтому малый градиент в этом интервале приводит к медленной сходимости к корню:

```
In [x]: newton(f, 0.01, fp)
...
RuntimeError: Failed to converge after 50 iterations, value is ...
```

Для вычисления корня методом `newton` необходимо либо переместить начальную точку ближе к корню, либо увеличить максимальное число итераций:

```
In [x]: newton(f, 0.01, fp, maxiter=100)
Out[x]: 1.148698354997035
```

г) Еще один пример функции, которая порождает бесконечный цикл перебора значений при использовании метода Ньютона–Рафсона:

```
In [x]: f = lambda x: x**4 - 4.29 * x**2 - 5.29
In [x]: fp = lambda x: 4*x**3 - 8.58 * x
In [x]: newton(f, 0.8, fp)
...
RuntimeError: Failed to converge after 50 iterations, value is ...
```

В отличие от функции в случае а) область $0.6 \leqslant x_0 \leqslant 1.1$ вызывает такое зацикленное поведение, поэтому необходимо инициализировать алгоритм вне этого интервала, чтобы вычислить корни ± 2.3. Например:

```
In [x]: newton(f, 1.2, fp)
Out[x]: -2.3
```

B8.4.3. В общем случае существуют два (физически различных) возможных значения угла θ_0, соответствующих прохождению маятника через заданную точку $(x_1, y_1) = (5, 15)$ – при движении вверх и при движении вниз. Эти значения являются корнями в интервале $(0, \pi/2)$ для следующей функции:

$$f(\theta_0; x_1, z_1) = x_1 \operatorname{tg} \theta_0 - (g x_1^2) / (2 v_0^2 \cos^2 \theta_0) - z_1.$$

После ограничения интервала искомых корней с помощью приближенного графика функции $f(\theta_0)$ можно воспользоваться методом `brentq`:

```
In [x]: g = 9.81
In [x]: v0, x1, z1 = 25, 5, 15
In [x]: f = lambda theta0, x1, z1: x1 * np.tan(theta0) - g / 2\
                          * (x1 / v0 / np.cos(theta0))**2 - z1
In [x]: th1 = brentq(f, 1, 1.4, args=(x1,z1))
In [x]: th2 = brentq(f, 1.5, 1.6, args=(x1,z1))
In [x]: np.degrees(th1), np.degrees(th2)
Out[x]: (74.172740936822834, 87.392310240255171)
```

Таким образом, $\theta_0 = 74.2°$ или $\theta_0 = 87.4°$.

B10.1.1. Пусть $x = 0.9999\ldots$. Тогда

$$10x = 9.9999\ldots = 9 + x \;\Rightarrow\; 9x = 9 \;\Rightarrow\; x = 1.$$

B10.1.2. Это происходит, потому что `math.pi` – это только лишь приближенное значение (число с плавающей точкой двойной точности) π, поэтому тангенс этого приближенного значения (непредвиденно) становится отрицательным:

```
In [x]: math.tan(math.pi)
Out[x]: -1.2246467991473532e-16
```

Попытка извлечения квадратного корня из этого значения приводит к арифметической ошибке.

B10.1.3. Здесь проблема, разумеется, состоит в том, что выражение записано с использованием чисел с плавающей точкой двойной точности, а разность между суммой первых двух членов и третьим меньше предела точности этого представления. Использование точного представления в целочисленной арифметике

```
In [x]: 844487**5 + 1288439**5
Out[x]: 398024523518563901305561949 7406
In [x]: 1288439**5
Out[x]: 39802452351856390132909246 56032
```

дает разность

```
In [x]: 844487**5 + 1288439**5 - 1318202**5
Out[x]: -235305158626
```

Но ограниченная точность используемого ранее представления чисел с плавающей точкой отсекает знаки после десятичной точки до вычисления разности:

```
In [x]: 844487.**5 + 1288439.**5
Out[x]: 3.980245235185639e+30
In [x]: 1318202.**5
Out[x]: 3.980245235185639e+30
```

Это пример «катастрофического взаимоуничтожения» – потери значащих разрядов.

B10.1.4. Выражение `1 - np.cos(x)**2` становится некорректным из-за потери значащих разрядов («катастрофического взаимоуничтожения») при значении, близком к `x = 0`, приводящем к критической потере точности и весьма резким непредсказуемым колебаниям на графике функции $f(x)$ (см. рис. А.2). Например, рассмотрим значение `x = 1.e-9`: в этом случае разность `1 - np.cos(x)**2` неотличима от нуля (в представлении с двойной точностью), поэтому `f(x)` возвращает 0. В другом случае значение `np.sin(x)**2` неотличимо от `x**2`, поэтому `g(x)` корректно возвращает `1.0`.

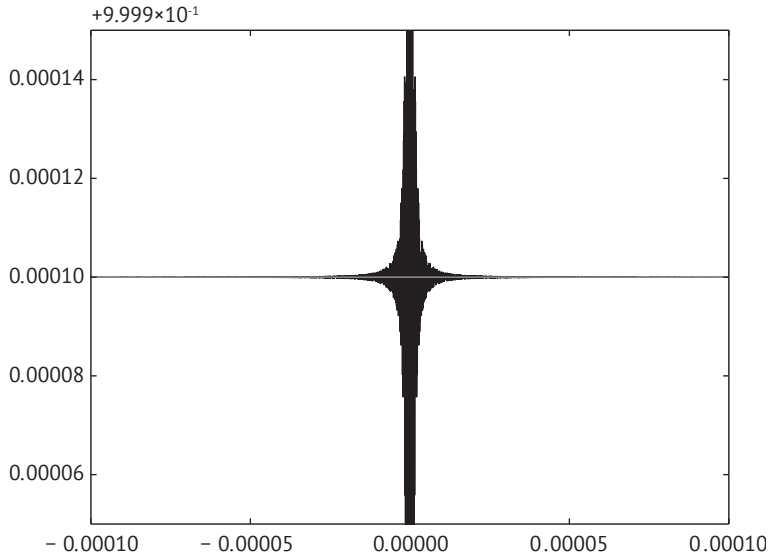


Рис. А.2. Сравнение поведения в численном представлении функций $f(x) = (1 - \cos 2x)/x^2$ и $g(x) = \sin 2x/x^2$ при значениях, близких к $x = 0$

Листинг А.4. Сравнение поведения в численном представлении функций $f(x) = (1 - \cos 2x)/x^2$ и $g(x) = \sin 2x/x^2$ при значениях, близких к $x = 0$

```
# qn9-1-c-cos-sin-a.py

import numpy as np
import matplotlib.pyplot as plt

f = lambda x: (1 - np.cos(x)**2)/x**2
g = lambda x: (np.sin(x)/x)**2

x = np.linspace(-0.0001, 0.0001, 10000)

plt.plot(x, f(x))
plt.plot(x, g(x))
plt.ylim(0.99995, 1.00005)
plt.show()
```

B10.1.5. В этом случае нельзя выполнять сравнение с помощью оператора ==, так как специальное значение `nan` не равно самому себе. Но это всего лишь число с плавающей точкой, которое не равно самому себе, поэтому вместо сравнения на равенство можно использовать оператор `!=`:

```
In [x]: c = 0 * 1.e1000      # 0 * inf дает результат nan.
In [x]: c != c
Out[x]: True                 # c не равно самому себе, поэтому его значением должно быть nan.
```

Приложение B. Различия между версиями Python 2 и 3

B.0.1 Целые числа и целочисленная арифметика

В версии Python 2 существовали два типа целых чисел: «простые» (системнозависимые, но обычно хранимые в 32 или 64 битах) и «длинные» (любого размера), обозначаемые суффиксом L. В версии Python 3 все проще: существует только один тип целого числа, величина которого может быть любой (ограничение только по доступной оперативной памяти на конкретном компьютере).

В Python 2 оператор деления / всегда должен был выполнять целочисленное деление (с округлением в сторону уменьшения), если оба операнда являлись целыми числами. В противоположность этому в Python 3 оператор / всегда возвращает число с плавающей точкой типа `float`. В обеих версиях можно использовать оператор // для принудительного выполнения целочисленного деления. Подводя итог, отметим следующие факты:

Python 2:

```
>>> 8 / 4
2            # Только в Python 2: результат int.
>>> 8 // 4
2
>>> 7.7 / 2
3.85
>>> 7.7 // 2
3.0          # Наибольшее целое, не превышающее 3.85.
>>> -7.7 // 2
-4.0         # Наибольшее целое, не превышающее -3.85.
```

Python 3:

```
>>> 8 / 4
2.0            # Результат типа float, даже если оба операнда 8 и 4 типа int.
>>> 8 // 4
2
>>> 7.7 / 2
3.85
>>> 7.7 // 2
3.0
>>> -7.7 // 2
-4.0
```

Встроенная функция `round()` работает немного по-разному в версиях 2 и 3. При округлении до нуля знаков после десятичной точки Python 3 применяет метод округления банкира: если число находится точно посередине между двумя целыми значениями, то возвращается четное целое типа `int`. В Python 2 функция `round()` округляет в сторону, противоположную направлению к нулю, и всегда возвращает значение типа `float`.

Python 2:

```
>>> round(-4.5)
-5.0
>>> round(6.5)
7.0
```

Python 3:

```
>>> round(-4.5)
-4
>>> round(6.5)
6
```

B.0.2 Операции сравнения

В Python 2 были разрешены сравнения объектов различных типов. При сравнении числового и нечислового типов числовой тип всегда был меньше нечислового. Другие объекты различных типов упорядочивались последовательно, но в произвольном порядке (таким образом, различные интерпретаторы Python могли создавать различные варианты упорядоченности, например в CPython объекты упорядочивались по имени типа, т. е. все объекты типа `dict` были «меньше, чем» объекты типа `str`):

```
>>> '2' > 5
True
>>> {} > 'a'       # (CPython).
False
```

В Python 3 не допускается сравнение объектов различных типов:

```
>>> '2' > 5
TypeError                                 Traceback (most recent call last)
...
----> 1 '2' > 5
TypeError: unorderable types: str() > int()
```

B.0.3 Ключевые слова

Зарезервированные ключевые слова Python, которые нельзя использовать как имена переменных (идентификаторы), приведены в табл. 2.4. В соответствующем списке для Python 2.7 исключены ключевые слова `async`, `await`, `nonlocal`, `True`, `False`, `None`, но включены слова `exec` и `print`.

Для новых пользователей одним из наиболее заметных различий между Python 2 и Python 3 является то, что в Python 2 `print` был оператором (ключевым словом), а в Python 3 `print()` – это (встроенная) функция:

Python 2:

```
>>> print '2 + 2 =', 4
2 + 2 = 4
>>> print >>fo, 'A string to write to file with open handle fo'
```

Python 3:

```
>>> print('2 + 2 =', 4)
2 + 2 = 4
>>> print('A string to write to file with open handle fo', file=fo)
```

Также обращает на себя внимание тот факт, что поскольку True и False не являлись ключевыми словами в Python 2, они могли использоваться как идентификаторы имен переменных. Из-за этого возникали «смысловые» проблемы, например:

Python 2:

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

B.0.4 Строки и Unicode

В Python 2 различались строки в кодировке Unicode (таким строковым литералам предшествовал символ `u`, например `u'El Niño'`) и строки в 8-битовой кодировке (содержащие только 7-битовые ASCII-символы). Это становилось источником многочисленных проблем, в том числе весьма неприятной ошибки `UnicodeEncodeError`, возникающей при смешивании строк в разных кодировках. Например, при попытке преобразования строки в кодировке Unicode в строку с 8-битовой кодировкой:

```
>>> s1 = 'I live in'
>>> s2 = u'Saint Étienne'
>>> '{} {}'.format(s1, s2)
----> 1 '{} {}'.format(s1, s2)

UnicodeEncodeError: 'ascii' codec can't encode characters in position 0-6:
ordinal not in range(128)
```

Все строки текста в Python 3 являются строками в кодировке Unicode и имеют тип `str`:

```
>>> s1 = 'I live in'
>>> s2 = 'Saint Étienne'
>>> '{} {}'.format(s1, s2)
'I live in Saint Étienne'
```

Если действительно необходимо использовать строку 8-битовых значений данных, то для этого существует строка байтов типа `bytes`. Бинарные литералы данных можно определить с помощью синтаксиса `b'…'` или выполнить явное приведение типа других совместимых объектов, как показано ниже:

```
>>> b1 = b'ABC'
>>> b2 = b'ABC\xff'
>>> b3 = bytes([65, 66, 67, 255])
>>> b4 = bytes([65, 66, 67, 0xff])
```

Последние три инструкции генерируют один и тот же объект типа `bytes` с конечным байтом, определяемым различными способами. При попытке присваивания объекту типа `bytes` значения, превышающего 255, возникает ошибка, так как байт по определению содержит 8 бит:

```
>>> b5 = bytes([65, 66, 67, 256])
----> 1 b5 = bytes([65, 66, 67, 256])

ValueError: bytes must be in range(0, 256)
```

B.0.5 Итераторы и списки

В Python 2 встроенная функция `range` возвращала список, а память выделялась для каждого элемента:

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Это создавало очевидные проблемы для весьма больших последовательностей, поэтому существовала отдельная встроенная функция `xrange`, возвращающая итерируемый объект, который должен был последовательно генерировать каждое значение арифметической прогрессии без сохранения всех элементов.

В Python 3 функция `range` возвращает итерируемый объект, подобно устаревшей функции `xrange`, а `xrange` больше не существует. В Python 3 `range` (но не `xrange` из Python 2) достаточно интеллектуально определяет, является ли целое число членом последовательности без итеративного прохода по ней, поэтому следующая инструкция выполняется чрезвычайно быстро:

```
>>> 999999999998 in range(0, 10**12, 2)       # Все четные числа меньше 1 триллиона.
True

>>> 999999999997 in range(0, 10**12, 2)
False
```

В Python 2 методы словарей `keys()`, `values()` и `items()` также возвращали списки соответствующих значений:

```
>>> d = {'a': 1, 'b': 2, 'c': 3}
>>> d.keys()
['a', 'c', 'b']
>>> d.items()
[('a', 1), ('c', 3), ('b', 2)]
```

В Python 3 возвращаются специальные итерируемые объекты. Если требуется список `list`, то необходимо обязательное явное приведение типа:

```
>>> d = {'a': 1, 'b': 2, 'c': 3}
>>> d.keys()
dict_keys(['a', 'b', 'c'])
>>> list(d.keys())
['a', 'b', 'c']     # Внимание: в Python 3.6+: словари выводятся в порядке вставки элементов.
```

Очевидно, что это более эффективно для больших словарей.

Следующий пример демонстрирует особенности использования встроенной функции `zip`.

Python 2:

```
>>> zip(['a', 'b', 'c'], [1, 2, 3])
[('a', 1), ('b', 2), ('c', 3)]        # Список кортежей.
```

Python 3:

```
>>> zip(['a','b','c'], [1, 2, 3])
<zip at 0x102bcf5c8 >            # Специальный итерируемый объект zip...
>>> list(zip(['a','b','c'], [1, 2, 3]))
[('a', 1), ('b', 2), ('c', 3)]   # …который можно преобразовать в список, если необходимо.
```

Приложение C. Механизм решения обыкновенных дифференциальных уравнений `odeint` в библиотеке SciPy

В разделе 8.2.3 описан метод `solve_ivp` из библиотеки SciPy для решения обыкновенных дифференциальных уравнений (ОДУ): эта функция предоставляет доступ к комплекту механизмов решения ОДУ через универсальный интерфейс и в настоящий момент является рекомендуемой методикой для большинства ситуаций, с которыми встречаются пользователи библиотеки SciPy. Более старая функция `scipy.integrate.odeint`[145] остается доступной в версии SciPy 1.4, так как от нее, вероятно, зависит весьма большой объем ранее написанного кода. В этом приложении описано практическое использование этой функции. Работа `odeint` основана на тщательно протестированной программе Fortran LSODA, которая может автоматически переключаться между алгоритмами решений жестких и нежестких систем ОДУ.

Одно обыкновенное дифференциальное уравнение первого порядка

Простейший вариант использования – решение одного ОДУ первого порядка:

$$dy/dt = f(y, t).$$

Функция `odeint` принимает три аргумента: объект функции, возвращающей dy/dt, начальное условие y_0 и последовательность значений t, в которых вычисляется решение $y(t)$.

Вернемся к примеру химической реакции первого порядка A → P с учетом концентрации реагента y = [A], которой соответствует дифференциальное уравнение

$$dy/dt = -ky.$$

Необходимо определить функцию:

```
def dydt(y, t):
    return -k * y
```

[145] Функция odeint является упрощенным интерфейсом к более продвинутому методу `scipy.integrate.ode`, который предоставляет набор разнообразных механизмов численного интегрирования, включая алгоритмы Рунге-Кутты и поддержку переменных с комплексными значениями.

Обратите внимание: порядок аргументов противоположен порядку, требуемому для метода `solve_ivp`. При использовании `odeint` можно предложить следующее решение, равнозначное решению, приведенному в разделе 8.2.3:

```
from scipy.integrate import odeint
# Постоянная скорости реакции первого порядка, 1/с.
k = 0.2
# Начальное условие для y: 100 % реагента имеется в момент времени t = 0.
y0 = 100
# Правильно подобранная сетка точек времени для этой реакции.
t = np.linspace(0, 20, 21)
y = odeint(dydt, y0, t)
```

Обратите внимание: по умолчанию `odeint` возвращает только решение *y* как функцию на заданной сетке точек времени[146].

Одно обыкновенное дифференциальное уравнение второго порядка

При решении ОДУ второго порядка, как и при решении задачи простого генератора гармонических колебаний из раздела 8.2.3, требуется разложение в систему ОДУ первого порядка:

```
# Частота гармонического осциллятора (1/с).
omega = 0.9
# Начальные условия для x1 = x и x2 = dx/dt в момент времени t = 0.
x0 = 3, 0 # cm, cm.s-1

def dxdt(x, t, omega):
    """ Return dx/dt = f(x,t) at time t for the harmonic oscillator."""
    """ Возвращает dx/dt = f(x,t) в момент времени t для генератора гармонических колебаний. """
    x1, x2 = x
    dx1dt = x2
    dx2dt = -omega**2 * x1
    return dx1dt, dx2dt

# Интегрирование дифференциальных уравнений с помощью odeint.
x1, x2 = odeint(dxdt, x0, t, args=(omega ,)).T
```

Функция `odeint` возвращает двумерный массив значений каждой зависимой переменной, расположенных по строкам: если необходимо распаковать этот массив в отдельные одномерные массивы координат положения и скорости, то потребуется транспонирование возвращенного массива.

[146] Для дополнительного аргумента `full_output` можно установить значение `True`, чтобы вывести еще и словарь `infodict` со статистическими данными о том, насколько хорошо (или плохо) было выполнено интегрирование.

Словарь терминов

IPython – интерактивная рабочая среда на основе интерфейса командной строки для Python, предоставляющая обширное разнообразие удобных функциональных возможностей, в том числе интроспекцию, хронологию команд, интерактивную визуализацию и автоматическое завершение ввода по клавише **Tab**.

Jupyter Notebook – интерактивная вычислительная блокнотная (виртуальная) среда на основе браузера для создания совместно используемых документов с включением в них мультимедийных фрагментов. В этой среде можно объединять исходный код, вывод результатов выполнения кода и текстовые описания в упорядоченной последовательности редактируемых ячеек.

Matplotlib – бесплатная библиотека с открытым исходным кодом, предназначенная для создания разнообразных двумерных и трехмерных графиков и диаграмм, удобная для визуализации данных, которые можно использовать в печатных изданиях и в режиме онлайн.

NumPy – бесплатная библиотека с открытым исходным кодом, поддерживающая быстрые численные операции с большими многомерными массивами, содержащими элементы одного типа данных.

pandas – бесплатная библиотека с открытым исходным кодом, предназначенная для обработки и анализа данных с поддержкой табличных данных разнородных типов, а также функциональность высокого уровня для группирования, агрегации и очистки наборов данных.

PEP – Python Enhancement Proposals – комплект рекомендаций по использованию, описаний стандартов и регулирующих процедур для разработчиков на языке Python. Новые функциональные возможности языка предлагаются и описываются в коротких документах, которые рецензируются координационным советом и помещаются в репозиторий www.python.org/dev/peps.

pip – менеджер пакетов для программной среды языка Python для установки пакетов из Python Package Index (https://pypi.org/), репозитория бесплатных программных библиотек для расширения функциональных возможностей языка.

SciPy – бесплатная библиотека с открытым исходным кодом, предназначенная для научных вычислений и включающая алгоритмы численного интегрирования, интерполяции, оптимизации и решения дифференциальных уравнений.

Unicode – международный стандарт для кодирования и представления текста, отображаемого почти во всех системах записи мира. Стандарт Unicode присваивает числовую кодовую точку (code point) каждому символу и определяет, как это числовое значение должно выражаться в байтах, с учетом

правил, относящихся к направленности текста, подобию (сопоставлению) символов и т. д.

Аргумент (argument) – значение, передаваемое в вызываемую функцию. Позиционные аргументы идентифицируются по их порядку в последовательности, заданной в определении функции (например, `complex(1, -2)`). Именованные аргументы связаны с конкретным идентификатором при вызове функции (например, `complex(real=1, imag=2)`).

Атрибут (attribute) – в Python объект (элемент данных, функция и т. д.), связанный («принадлежащий») с другим объектом и доступный через точку: `object.attribute`. Например, функция `upper` является атрибутом объекта `str`: при создании экземпляра строки, скажем `'python'`, атрибут становится доступным как `'python'.upper` и вызывается как `'python'.upper()`.

Байт-код (bytecode) – промежуточный язык, в который исходный код компилируется интерпретатором Python для выполнения виртуальной машиной Python как машинного кода, понятного процессору компьютера.

Векторизация (vectorization) – пакетное (комплексное) выполнение одной операции с массивом в целом без необходимости явного применения цикла Python – это улучшает и скорость, и удобство чтения. Библиотека NumPy поддерживает векторизацию для объекта типа `ndarray` посредством реализации многих из подобных операций в виде предварительно скомпилированного кода на языке C.

Встроенный (built-in) – любой из типов и функций, предварительно определенных интерпретатором Python и всегда доступных без явного импорта из какого-либо другого модуля. Примеры: `print()`, `float` и `range()`.

Выражение (expression) – синтаксически корректное сочетание допустимых в Python идентификаторов, литералов и операторов, вычисляющее некоторое значение. Например, (предполагается, что значение `x` определено) `x + 2` – корректное выражение Python, вычисляющее сумму значения `x` и целого числа 2.

Генератор псевдослучайных чисел (pseudorandom-number generator – PRNG) – алгоритм, генерирующий последовательность чисел, которая приближенно имитирует свойства случайных чисел. Генератор псевдослучайных чисел может инициализироваться (seeded) фиксированным значением для повторного воспроизведения последовательности (это детерминированная аппроксимация), которая в конце концов окажется повторяющейся (хотя и с достаточно длинным периодом).

Генератор списков (list comprehension) – синтаксическая конструкция Python для создания списка в одной лаконичной, но удобочитаемой строке кода без явного использования метода append, например `[x**2 for x in range(5)]`.

Динамическая проверка типа (dynamic type-checking) – определение соответствия объекта выполняемой над ним операции во время выполнения, а не перед выполнением программы (статическая проверка типа). Python – это язык с динамической типизацией.

Изменяемые и неизменяемые объекты (mutable and immutable objects) – неизменяемый объект не может быть изменен после его определения: объектам этого типа выделяется область памяти (связанная с некоторой идентификационной характеристикой, которую можно мысленно интерпретировать как «адрес» объекта в памяти), и содержимое этой области памяти невозможно изменить, пока существует объект (хотя имя, связанное с этим объектом, может быть «переприсвоено» другому объекту). Изменяемый объект можно изменять непосредственно: его свойства могут изменяться без изменения идентификационной характеристики. Примеры некоторых типов неизменяемых объектов: `int`, `float`, `str`, `tuple`. Объекты типа `list`, `dict` и `set` являются изменяемыми.

Имя-идентификатор (identifier name) – символьное имя, связанное с объектом, которое используется для обращения к этому объекту в исходном коде программы.

Инструкция (statement) – одна строка (или несколько строк) кода Python, состоящая из выражений, которые в общем случае оказывают воздействие на состояние выполняемой программы.

Интроспекция (introspection) – способность программы или сеанса интерактивной командной оболочки предоставлять информацию об объекте во время выполнения. Например, выражение `type(x)` возвращает тип объекта, идентифицируемого по имени `x`.

Исключение (exception) – ошибка, возникающая во время выполнения Python-программы. Если исключение не обрабатывается, то программа прекращает выполнение. Общепринятая практика: предвосхищать и перехватывать (catch) с целью обработки конкретные исключения в блоке `try…except` (см. описание парадигмы EAFP). Кроме того, можно использовать и генерировать исключения, определенные пользователем.

Итерируемый (объект) (iterable) – объект является итерируемым, если он содержит или порождает последовательность значений, по которой можно проходить поочередно при выполнении любого цикла. Примеры: списки (`list`), кортежи (`tuple`) и строки (`str`).

«Катастрофическое взаимоуничтожение» (catastrophic cancellation) – критическая потеря значимых разрядов при использовании арифметики с плавающей точкой, когда выполняется вычитание очень близких друг к другу значений.

Класс (class) – шаблон для создания (экземпляров) объектов, определяющий их атрибуты, интерфейсы и поведение. Классы могут быть связаны между собой механизмом наследования.

Командная оболочка (shell) – пользовательский интерфейс в компьютерной системе, точнее интерактивный интерфейс командной строки, в которой вводятся и выполняются команды.

«Легче попросить прощения, чем разрешения» («Easier to Ask Forgiveness than to seek Permission» – EAFP) – характерный для Python стиль напи-

сания кода, при котором пытаются выполнить обработку данных с намерением аккуратно обработать любые возникающие ошибки (исключения). Этот подход противоположен парадигме «Look Before You Leap» (посмотри, прежде чем прыгнуть; семь раз отмерь, один отрежь), принятой в других языках, где данные тщательно проверяются на соответствие их типа перед попыткой выполнения операции с ними.

Литерал (literal) – в Python литерал – это прямая спецификация объекта при синтаксическом разборе (парсинге) интерпретатором (в противоположность ссылке на объект по имени соответствующей переменной). Например, `'parrot'` – строковый литерал.

Лямбда-функция (lambda) – анонимная функция: не связанная с каким-либо именем-идентификатором. В Python лямбда-функции обычно определяются непосредственно в строке (inline) другого кода и могут содержать только одно выражение.

Магическое число (magic number) – постоянное числовое значение, используемое как литерал в программе вместо присваиваемого осмысленного имени переменной. Рекомендуется избегать использования магических чисел, чтобы исходный код оставался легко читаемым и удобно сопровождаемым.

Метод (method) – функция, связанная с объектом.

Методы с двойным подчеркиванием (double-underscore methods) – специальные методы (dunder-методы) обозначаются в Python именами, начинающимися и заканчивающимися двумя символами подчеркивания. Такие методы обычно реализуют операцию, вызываемую некоторыми специальными синтаксическими конструкциями или встроенными функциями. Например, индексирование списка `x[i]` – это по существу равнозначно вызову `x.__getitem__(i)`. В Python возможно переопределение этих методов аналогично перегрузке операторов.

Модуль (module) – комплект кодов Python (определений объектов, включая классы и функции, и выполняемого кода) в одном или нескольких файлах, который можно импортировать для многократного использования в других программах.

Наследование (inheritance) – реализация иерархических связей между классами в объектно-ориентированной программе. Класс, определяемый как основанный на другом классе (производный класс или подкласс), наследует атрибуты этого базового класса, которые можно изменять и добавлять новые. Наследование способствует многократному использованию исходного кода, а представляемая этим механизмом иерархия часто неотделима от внутренних концепций, реализованных в классах.

Не число (Not a Number – NaN) – специализированное значение для типа данных с плавающей точкой, представляющее неопределенные или непредставимые числа.

Область видимости (scope) – характеристика доступности переменной (имени) в каком-либо блоке кода. Поскольку одно имя-идентификатор может быть связано с различными объектами в разных частях программы, правила разрешения области видимости определяют, какой объект разрешено использовать с рассматриваемым идентификатором. Например, если происходит обращение к переменной внутри функции, то интерпретатор Python в первую очередь определяет, присвоено ли это имя объекту внутри функции (в локальном пространстве имен). Если объекта с таким именем нет, то рассматривается блок кода, включающий эту функцию, и т. д. – до глобального пространства имен программы (или модуля). Конечным является собственное пространство встроенных имен Python (содержащее предварительно определенные имена объектов, таких как функция `print`).

Обусловленность (численной задачи) (conditioning (of a numerical problem)) – задача, решение которой является относительно независимым от ошибок и неопределенностей в исходных данных, называется хорошо обусловленной (well-conditioned). Задача, в которой относительно малые погрешности в исходных данных значительно искажают результат, называется плохо обусловленной (ill-conditioned).

Объект (object) – абстрактная концепция сущности, реализованная в компьютерной программе как структура данных, которая может содержать и обрабатывать собственные данные (атрибуты), включая методы и другие объекты.

Объектно-ориентированное программирование (object-oriented programming – OOP) – парадигма программирования, в соответствии с которой компоненты системы идентифицируются как отдельные сущности, определяемые в коде объектами, описываемые шаблонами («схемами») – классами. Такой концептуальный подход может помочь при решении задач посредством разделения их на небольшие взаимосвязанные части с простым индивидуальным поведением, взаимодействующие определенным и управляемым способом.

Оптимизация (математическая) (optimization (mathematics)) – процесс получения наилучшей (по некоторому критерию) параметризации задачи (возможно, связанной с некоторыми ограничениями). Например, поиск значений x и y, при которых значение функции $f(x,y)$ минимально, – это задача оптимизации. Поиск максимального значения той же функции по существу является аналогичной задачей, так как она равнозначна минимизации $-f(x,y)$.

Пакет (package) – иерархически структурированный набор модулей, хранящихся в каталоге, который создает для них пространство имен и позволяет в крупных проектах определять способ их импортирования и использования.

Перегрузка оператора (operator overloading) – реализация оператора, используемого в одном классе, в другом классе: например, оператор + используется для сложения целых чисел (как в выражении `2 + 3`), но еще и для объединения объектов типа list: `[1, 2] + [3, 4, 5]` возвращает `[1, 2, 3, 4, 5]`. Это пример полиморфизма. В классах, определенных пользователем, можно перегрузить любой оператор, определив соответствующий ему метод с двойным подчеркиванием.

Переменная (variable) – символьное имя, связываемое с объектом, по которому происходит обращение к этому объекту в коде программы. В Python переменная в этом смысле иногда более формально называется именем-идентификатором (identifier name): один объект может иметь более одного такого идентификатора.

Переполнение и обращение в машинный ноль (overflow and underflow) – состояние, когда результат вычислений больше или меньше значения, которое может быть представлено в памяти для используемого типа данных. Например, вычисление `math.exp(1000)` приводит к переполнению при использовании арифметики с плавающей точкой двойной точности, потому что результат больше, чем наибольшее представимое значение типа `float`, использующее 64 бита (приблизительно 1.8×10^{308}).

Подгонка методом наименьших квадратов (least-squares fitting) – в анализе переопределенных систем (систем с количеством точек данных, бóльшим, чем количество неизвестных параметров рассматриваемой модели) это процесс получения набора параметров, минимизирующих сумму квадратов (наблюдаемых – моделируемых) остатков (невязок). При линейной подгонке методом наименьших квадратов функция модели зависит линейно от своих параметров.

Полиморфизм (polymorphism) – свойство функции (метода), позволяющее разумно выполнять некоторую операцию с данными различных типов. Например, оператор * является полиморфным, потому что может использоваться для умножения двух чисел (`2 * 1.2` возвращает `2.4`), но также для создания списка повторяющихся элементов (`[0] * 4` возвращает `[0, 0, 0, 0]`).

Порядок следования байтов (endianness) – зависимый от платформы порядок байтов, представляющих число. Системы с порядком байтов от старшего к младшему (с обратным порядком) (big-endian systems) первым размещают старший байт (по наименьшему адресу памяти). Системы с порядком байтов от младшего к старшему (с прямым порядком) (little-endian systems) записывают старший байт последним. Название порядка байтов происходит от фанатичной вражды лилипутов и блефусканцев по поводу разбивания вареных яиц с тупого или острого конца («тупоконечники» и «остроконечники»), описанной Джонатаном Свифтом в романе «Путешествия Гулливера» (1726).

Пространство имен (namespace) – отображение (словарь), устанавливаемое между именами-идентификаторами и объектами, с которыми связаны имена. Решение о выборе пространства имен для правильной идентификации имени используемого объекта определяется областью видимости (scope) имени.

Синтаксический сахар (syntactic sugar) – программный синтаксис и конструкции, которые не являются абсолютно необходимыми для обеспечения функциональности языка, но делают исходный код более простым, понятным и удобным для программирования, а иногда даже более эффективным (быстрым). Например, Python поддерживает комбинированное присваивание (допустим, `a += 1` – это синоним присваивания `a = a + 1`) и генератор списков.

Стабильность (устойчивость) алгоритма (stability of an algorithm) – алгоритм называют стабильным (устойчивым), если он относительно независим от ошибок (погрешностей) аппроксимации различного рода, которые могут возникать при его выполнении или при вводе данных. Если такие ошибки накапливаются (обычно это приводит к критическому отказу алгоритма, который не позволяет получить осмысленный результат), то алгоритм называется нестабильным (неустойчивым).

Стандартная библиотека (Standard Library) – большой набор модулей, содержащих методы выполнения общих задач, например математических, операций ввода-вывода или отладки. Стандартная библиотека устанавливается автоматически вместе с большинством дистрибутивов Python. Дополнительные пакеты и модули доступны в репозитории Python Package Index, их можно устанавливать с помощью утилиты pip.

Стек (программный) (stack (software)) – набор программных компонентов, которые работают совместно для создания платформы (или среды), в которой может выполняться некоторый класс компьютерных задач. Например, стеком SciPy иногда называют комплект, состоящий из интерпретатора Python, установленного вместе с библиотеками NumPy, SciPy и Matplotlib.

Строгая и слабая типизация (strongly and weakly typed) – язык определяется как слабо типизированный, если он незаметно для пользователя выполняет преобразование объектов в требуемый тип, чтобы позволить функции работать с ним. Языки со строгой типизацией позволяют выполнять только операции с объектами предварительно определенного набора типов (языки со статической типизацией) или с объектами, обладающими совместимыми свойствами (языки с динамической типизацией). Python является языком с относительно строгой типизацией: например, выражение `'hello' + 4` генерирует исключение (`TypeError`). Языки, подобные JavaScript, являются языками с относительно слабой типизацией – в них автоматически выполняется преобразование целого числа `4` в строку, чтобы вернуть литерал `'hello4'`.

Строка документации (docstring) – строковый литерал, записанный как первая инструкция в определении функции, класса или модуля. Становится атрибутом `__doc__` своего объекта и документирует его поведение и функциональность.

Управление версиями (version control) – система управления изменениями в процессе разработки программного обеспечения, в которой часто используются инструментальные средства, обеспечивающие совместную работу, систему меток (тегов) выпускаемых версий и разветвление на параллельно развивающиеся версии исходного кода в процессе разработки.

Утиная (неявная) типизация (duck typing) – определение принадлежности объекта к какому-либо типу с учетом выполняемой над ним операции по методам и свойствам, которыми он обладает, вместо явного формального объявления его типа данных. Python – это язык с утиной типизацией.

Функция (function) – набор инструкций, объединенных в блок программы для выполнения некоторой задачи при вызове. В функции можно передавать аргументы (данные в форме ссылок на объекты Python). Функция также может возвращать одно или несколько значений. Функции, принадлежащие какому-либо объекту, называются его методами. Но поскольку в Python все есть объект, между терминами «метод» и «функция» нет почти никакой разницы.

Хеш (hash) – объект Python является хешируемым (hashable), если он может возвращать из специального метода `__hash__` хеш-значение, которое никогда не изменяется в течение своего жизненного цикла, и может сравниваться с другими объектами на равенство. Это важно для таких структур данных, как словари, которые отображают хеш-значения в объекты для обеспечения эффективного хранения и высокопроизводительного доступа.

Число с плавающей точкой (floating-point number) – представление действительного числа, используемое в компьютерной арифметике, при котором число хранится в виде отдельных частей, представляющих значащие разряды и показатель степени, в фиксированном количестве битов, в общем случае определяющем ограниченную точность.

Экземпляр (instance) – специальное представление объекта, созданное (instantiated) из класса с собственными конкретными данными и именем-идентификатором. В дополнение к значениям атрибутов, принадлежащих конкретному экземпляру объекта, могут существовать данные, совместно используемые всеми экземплярами рассматриваемого класса (атрибуты класса).

Предметный указатель

O

P

R

S

В

Г

К

Л

M

H

P

C

Т

У

Ф

Книги издательства «ДМК Пресс» можно заказать
в торгово-издательском холдинге «Планета Альянс» наложенным платежом,
выслав открытку или письмо по почтовому адресу:
115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **www.a-planeta.ru**.
Оптовые закупки: тел. **(499) 782-38-89**.
Электронный адрес: **books@alians-kniga.ru**.

Кристиан Хилл

Научное программирование на Python

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Зам. главного редактора *Сенченкова Е. А.*
Перевод *Снастин А. В.*
Корректор *Синяева Г. И.*
Верстка *Луценко С. В.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.
Гарнитура «PT Serif». Печать цифровая.
Усл. печ. л. 52,49. Тираж 200 экз.

Веб-сайт издательства: www.dmkpress.com