В серии: Библиотека ALT

М. В. Сысоева, И. В. Сысоев

Программирование для «нормальных» с нуля на языке Python

В двух частях

Часть 1

Учебник

Москва
Базальт СПО
МАКС Пресс
2018

УДК 004.43
ББК 22.1
С95

Сысоева М. В., Сысоев И. В.
С95 Программирование для «нормальных» с нуля на языке Python: Учебник. В двух частях. Часть 1 / Ответственный редактор: В. Л. Черный : — М.: Базальт СПО; МАКС Пресс, 2018. — 176 с. [+4 с. вкл]: ил. — (Библиотека ALT).

ISBN 978-5-9908979-5-3
ISBN 978-5-317-05783-1
ISBN 978-5-317-05784-8 (часть 1)

Книга — учебник, задачник и самоучитель по алгоритмизации и программированию на Python. Она не требует предварительных знаний в области программирования и может использоваться для обучения «с нуля».

Издание адресовано студентам, аспирантам и преподавателям инженерных и естественно-научных специальностей вузов, школьникам старших классов и учителям информатики. Обучение языку в значительной степени строится на примерах решения задач обработки результатов радиофизического и биологического эксперимента.

Сайт книги: `http://www.altlinux.org/Books:Python-sysoeva`

Ключевые слова: программирование; численные методы; алгоритмы; графики; Python; numpy

УДК 004.43
ББК 22.1

ISBN 978-5-9908979-5-3
ISBN 978-5-317-05783-1
ISBN 978-5-317-05784-8 (часть 1)

Оглавление

Предисловие

Эта книга написана для инженеров, физиков, биологов и просто всех-всех, кто не изучал программирование прежде, но для кого оно может быть полезно как средство решения своих насущных задач, а не является самоцелью. Для них выбор правильного языка для обучения и работы очень важен: такой язык должен быть одновременно прост в освоении и использовании и логично организован, иметь много внешних модулей и расширений для решения реальных задач (то есть быть популярным), и быть хорошо доступен — свободно распространяться вместе со внешними модулями для всех основных операционных систем. Язык Python лучше всех других удовлетворяет всем этим требованиям и поэтому ныне используется во многих вузах и школах для обучения и одновременно бьёт рекорды по популярности среди учёных и инженеров.

Книга позволит вам, начав с нуля, быстро и качественно научиться делать нужные вещи: производить вычисления, читать, записывать и анализировать данные, строить графики, и при этом освоить основные принципы программирования: структуры данных, циклы, условия, подпрограммы, поиск ошибок и отладку. Подбирая материал, мы сознательно придерживались самых простых путей решения той иной задачи, даже если это несколько противоречило академически принятому порядку изложения или сложившимся традициям. В ряде случаев, например, при работе с текстовыми файлами и массивами, мы даже предпочли использование широко популярных внешних модулей встроенным средствам. Всё изложение построено так, чтобы быть полезным и применимым сразу, ведь усваивается то, что используется.

Книга может выступать как в качестве самоучителя, так и в качестве учебника для преподавания в школе или вузе, задачника или просто справочника.

Сведения об авторах

- Сысоева Марина Вячеславовна — кандидат физико-математических наук, ассистент кафедры «Радиоэлектроника и телекоммуникации» Саратовского государственного технического университета имени Гагарина Ю.А.

- Сысоев Илья Вячеславович — кандидат физико-математических наук, доцент кафедры динамического моделирования и биомедицинской инженерии

Саратовского национального исследовательского государственного университета имени Н.Г. Чернышевского.

Сведения о рецензентах

- Пономаренко Владимир Иванович — доктор физико-математических наук, ведущий научный сотрудник Саратовского филиала Института радиотехники и электроники имени В.А. Котельникова РАН.
- Огнева Марина Валентиновна — кандидат физико-математических наук, заведующая кафедрою информатики и программирования Саратовского национального исследовательского государственного университета имени Н. Г. Чернышевского.
- Курячий Георгий Владимирович — преподаватель факультета ВМК Московского Государственного Университета им. М. В. Ломоносова, автор курсов по Python для ВУЗов и Вечерней математической Школы, разработчик компании «Базальт СПО».

Глава 1

Введение

1.1 Языки программирования

Первый проект вычислительной машины был разработан Чарльзом Бэббиджем в 1833 году в Великобритании. Описание проекта сделала Августа Ада Кинг (единственная дочь знаменитого поэта лорда Байрона), она же ввела такие фундаментальные понятия, как «цикл», «рабочая ячейка», и потому считается первым в мире программистом; язык программирования Ада назван в её честь.

Машина Бэббиджа никогда не была реализована полностью, хотя на её реализацию ушло 17 тысяч фунтов стерлингов и 9 лет времени. Основная проблема, с которою столкнулся Бэббидж, — низкий уровень элементной базы: промышленность того времени не была готова производить детали нужного качества и в требуемые сроки. Тем не менее, его последователь Георг Шутц в 1850-ых построил несколько работающих «разностных машин».

Первая реальная электрическая вычислительная машина была построена немецким инженером-исследователем К. Цузе в 1938 году, аналогичные работы велись независимо от него в США Д. Штибитцем и Г. Айкеном. Базовые принципы архитектуры современных ЭВМ были сформулированы Джоном фон Нейманом в 1946 году в США, а в 1948 году в Англии была построена первая ЭВМ, основанная на этих принципах.

В СССР первая машина БЭСМ была спроектирована в 1951 году и уже в следующем году началась её практическая эксплуатация. Элементная база для первых ЭВМ 40-50-ых годов представляла собою вакуумные лампы. Переход на полупроводниковую элементную базу в 1960-ых позволил существенно повысить быстродействие, уменьшить размер и энергопотребление ЭВМ. Следующим этапом стал переход от отдельных транзисторов к интегральным логическим схемам.

Первые программы заключались в установке ключевых переключателей на передней панели вычислительного устройства. Очевидно, таким способом можно было составить только небольшие программы. Одну из первых попыток со-

здать полноценный язык программирования предпринял немецкий учёный Конрад Цузе, который в период с 1943 по 1945 год разработал язык Plankalkül (Планкалкюль). В переводе на русский язык это название соответствует выражению «планирующее исчисление». Это был очень перспективный язык, фактически являвшийся языком высокого уровня, однако из-за военных действий он не был доведён до практической реализации.

Неизвестно, насколько бы ускорилось развитие программирования, если бы наработки Цузе стали доступны другим учёным в конце 40-х годов, но на практике с развитием компьютерной техники сначала получил распространение *машинный язык*. Запись программы на нём состояла из единиц и нулей. Машинный язык принято считать языком программирования первого поколения (при этом разные машины разных производителей использовали различные коды, что требовало переписывать программу при переходе на другую ЭВМ).

Программа «Hello, world!» для процессора архитектуры x86 (ОС MS DOS, вывод при помощи BIOS прерывания int10h) выглядит следующим образом (в шестнадцатеричном представлении):

```
BB 11 01 B9 0D 00 B4 0E 8A 07 43 CD 10 E2 F9
CD 20 48 65 6C 6C 6F 2C 20 57 6F 72 6C 64 21
```

Вскоре на смену такому методу программирования пришло применение языков второго поколения, также ограниченных спецификациями конкретных машин, но более простых для использования человеком за счет использования мнемоник (символьных обозначений машинных команд) и возможности сопоставления имен адресам в машинной памяти. Они традиционно известны под наименованием языков ассемблера (*транслируемые языки*). Эти языки относятся к языкам низкого уровня, то есть близким к программированию непосредственно в машинных кодах.

Однако при использовании ассемблера становился необходимым процесс перевода программы на язык машинных двоичных кодов перед её выполнением, для чего были разработаны специальные программы, также получившие название ассемблеров (трансляторов). Сохранялись и проблемы с переносимостью программы с ЭВМ одной архитектуры на другую, и необходимость для программиста при решении задачи мыслить терминами «низкого уровня»: ячейка, адрес, команда.

Классическая программа на одном из диалектов Ассемблера:

```
.386 // тип процессора
.MODEL SMALL // модель памяти
.DATA // инициализированные данные:
    msg DB 'Hello, World', 13, 10, '$'
.CODE // исполняемый код:
START:
    mov ax, @DATA // Загрузка адреса сегмента в регистр ds
    mov ds, ax
```

```
    mov ax, 0900h
    lea dx, msg
    int21h
    mov ax, 4C00h
    int 21h
END START
```

Поскольку команды ассемблера всего лишь более удобно обозначенные инструкции в двоичных кодах, как правило, можно добиться взаимнооднозначного соответствия между программами в машинных кодах и программами на ассемблере, следовательно, они оказываются полностью взаимно заменяемыми. В этом кроется как ключевое преимущество, так и ключевой недостаток ассемблера. С одной стороны, программирование в двоичных кодах становится ненужным ровно с того момента, как написана программа-транслятор, при этом сохраняется полный доступ ко всем возможностям ЭВМ. С другой стороны, если изменятся инструкции или регистры, программа на ассемблере окажется бесполезна. Поскольку архитектура ЭВМ меняется часто, а одновременно сосуществуют вычислительные машины различных архитектур, получается, что программы на ассемблере приходится всё время переписывать.

Чтобы не переписывать каждый раз программу, необходимо было создать некоторый уровень абстракции: спрятать детали организации конкретного компьютера от программиста и позволить ему мыслить категориями более универсальными, чем категории конкретных инструкций конкретной машины. Такой шаг был впервые сделан в 1958 году, когда появился первый язык высокого уровня — FORTRAN (сокращение от FORmula TRANSlation). Хотя программы на Фортране работали существенно (в 2–4 раза) медленнее, чем программы на ассемблере, переход на языки высокого уровня стал огромным шагом вперёд, поскольку число способных к программированию людей резко увеличилось: стало не нужно помнить все регистры и инструкции процессора. Программировать начали не только профессиональные программисты, но и учёные и инженеры.

Программа «Привет мир» на Фортране (эта программа будет компилироваться только сравнительно современными компиляторами, поддерживающими стандарт Frotran 90 или более новые):

```
Program hello
  write(*,*) 'Hello, World!'
end
```

Конечно, программа на Фортране требовала перевода в двоичный код, который, в отличие от ассемблера, нельзя было сделать простым взаимно однозначным транслированием. Для этого была написана на ассемблере специальная программа — компилятор. Поэтому такие языки получили название *компилируемых*. Когда изменяется архитектура ЭВМ, компилятор для каждого языка приходится переписывать, ведь он всё равно написан на ассемблере. Компилируемые языки: Fortran (1958 г.), Algol (1960 г.), C (1970 г.) и его потомки (C++,

D, Vala), Pascal (1970 г.) и его потомки (Delphi, FreePascal/Lazarus) — основа современного программирования.

Программа «Привет мир» на Pascal:

```
program hello;
begin
  write('Hello, World');
end.
```

Программа «Привет мир» на C:

```
#include <stdio.h>
int main() {
  printf('Hello, World');
  return 0;
}
```

Со временем производительность компьютеров выросла настолько, что оказалось возможным не компилировать код программ в двоичный, а сразу исполнять его строчка за строчкою. Такой способ называется интерпретированием, программа, интерпретирующая код — интерпретатором, а такие языки — *интерпретируемые*. MATLAB (1978 г.), Perl (1987 г.), Python (1992 г.), PHP (1995 г.), Ruby (1995 г.), Javascript (1995 г.) — примеры популярных интерпретируемых языков. Интерпретатор может работать в двух режимах: интерактивном и выполнения скрипта.

Программа «Привет мир» на Perl:

```
#!/usr/bin/perl
print "Hello, World\n"
```

Программа «Привет мир» на Python:

```
print('Hello, World')
```

На PHP:

```
<?='Hello, World'?>
```

На Ruby:

```
{puts "Hello, World"}
```

На JavaScript:

```
<script type="application/javascript">
        Alert('Hello, World');
</script>
```

Интерпретируемые языки проще в освоении и использовании, но их область применения ограничена, поскольку программы на них не могут взаимодействовать с процессором напрямую, а производительность существенно ниже, чем у

компилируемых. Они используются там, где либо время исполнения программы некритично, либо в случае, когда программа пишется на один раз, поскольку тогда относительно большое время исполнения компенсируется существенно меньшим временем написания. Так, Perl появился как язык для обработки текстов, PHP — пример удачного языка для создания сайтов.

Промежуточное положение между компилируемыми и интерпретируемыми языками занимают *языки виртуальных машин*, самые распространённые из которых Java (компилируется в машинный код виртуальной машины Java Virtual Machine) и C# (компилируется в машинный код виртуальной машины Common Language Runtime — основы для всех языков семейства .NET).

Для них компиляция происходит не в двоичный код данного конкретного процессора, а в двоичный код специальной виртуальной машины (иногда его называют байткод). Таким образом, достигаются два существенных плюса: во-первых, можно не перекомпилировать программу под каждый новый процессор, во-вторых, компилятор, имея возможность анализировать всю программу целиком, всё-таки может произвести ряд оптимизаций, увеличивая таким образом скорость исполнения по сравнению с простым пошаговым интерпретированием.

Хотя языки виртуальных машин ближе к компилируемым, чем интерпретируемые языки, они появились позже и их условно можно назвать пятым поколением языков программирования.[1] Некоторые языки могут и компилироваться, и интерпретироваться, и компилироваться в байткод, например, OCaml.

Программа «Привет мир» на Java:

```
class HelloWorld {
  public static void main (String args []) {
    System.out.println("Hello␣World");
  }
}
```

Таблица 1.1. Поколения языков программирования

I поколение	Машинные языки
II поколтение	Транслируемые языки (ассемблеры)
III поколение	Компилируемые языки
IV поколение	Интерпертируемые языки
V поколение	Языки виртуальных машин

[1] Многие авторы до сих пор выделяют только 3 поколения языков программирования, совмещая компилируемые, интерпретируемые и компилируемые в байткод языки в рамках одного последнего. Хотя такая классификация является «классическою», в наше время с нею трудно согласиться, так как большинство программистов никогда не имели дело с первыми двумя поколениями и, следовательно, для них от такой классификации вовсе нет толка.

С начала 90-х на смену обычным языкам программирования в области вычислений стали приходить различные специализированные *математические пакеты*. В настоящее время наибольшею популярностью пользуется MatLab. Кроме него часто используются также другие коммерческие пакеты: Mathematica, MathCad, STATISTICA, а также свободные аналоги: SciLab и, особенно, статистический пакет R. Пакеты существенно упростили разработку приложений, внеся два ключевых усовершенствования:

- большая доступная встроенная библиотека алгоритмов, которая может быть расширена средствами, как самого пакета, так и с подключением модулей на Fortran и C;
- встроенные средства для построения графиков, позволяющие визуализировать данные на экране компьютера в интерактивном режиме и сохранять результаты построения в файлы основных форматов.

1.2 Парадигмы программирования

Парадигма программирования — это совокупность идей и понятий, определяющих стиль написания компьютерных программ (подход к программированию).

Важно отметить, что парадигма программирования не определяется однозначно языком программирования; практически все современные языки программирования в той или иной мере допускают использование различных парадигм (*мультипарадигмальное программирование*). Также важно отметить, что существующие парадигмы зачастую пересекаются друг с другом в деталях, поэтому можно встретить ситуации, когда разные авторы употребляют названия из разных парадигм, говоря при этом, по сути, об одном и том же явлении. Считается, что все парадигмы делятся на две большие части: императивное и декларативное программирование. *Императивное программирование* — это парадигма программирования, которая описывает процесс вычисления в виде инструкций, изменяющих состояние данных. Подразделы императивного программирования — *структурное* и *объектно-ориентированное*. *Декларативное программирование* — это парадигма программирования, в которой вместо пошагового алгоритма решения задачи (что делает императивное программирование, описывающее как решить задачу) задаётся спецификация решения задачи, т. е. описывается, что собой представляет проблема и что требуется получить в качестве результата. Декларативные программы не используют понятия состояния и, в частности, не содержат переменных и операторов присваивания. К декларативной парадигме относится *функциональное* программирование.

Структурное программирование — методология разработки программного обеспечения, в основе которой лежит представление программы в виде иерархической структуры блоков. Языками-первопроходцами в этой парадигме были Fortran, Algol и B, позже их приемниками стали Pascal и C. В соответствии с данной методологией любая программа состоит из трёх базовых управляющих

структур: последовательность, ветвление, цикл; кроме того, используются подпрограммы. При этом разработка программы ведётся пошагово, методом «сверху вниз».

Следование принципам структурного программирования сделало тексты программ, даже довольно крупных, нормально читаемыми. Серьёзно облегчилось понимание программ, появилась возможность разработки программ в нормальном промышленном режиме, когда программу может без особых затруднений понять не только её автор, но и другие программисты. Python использует эту парадигму как вспомогательную.

Объектно-ориентированное программирование (ООП) — это методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования. В основе концепции объектно-ориентированного программирования лежит понятие объекта — некой сущности, которая объединяет в себе поля (данные) и методы (выполняемые объектом действия). Например, объект `ЧЕЛОВЕК` может иметь поля `ИМЯ`, `ФАМИЛИЯ` и методы `КУШАТЬ`, `СПАТЬ`. Соответственно, в программе можем использовать операторы `ЧЕЛОВЕК.ИМЯ:='Иван'` и `ЧЕЛОВЕК.КУШАТЬ(пища)`.

С самого начала Python проектировался как объектно-ориентированный язык программирования: все данные представляются объектами Python, программа является набором взаимодействующих объектов, посылающих друг другу сообщения, каждый объект имеет собственную часть памяти и может иметь в составе другие объекты, каждый объект имеет тип, объекты одного типа могут принимать одни и те же сообщения (и выполнять одни и те же действия).

Функциональное программирование — парадигма программирования, в которой процесс вычисления трактуется как вычисление значений функций в математическом понимании последних (в отличие от функций как подпрограмм в структурном программировании). Наиболее известные LISP, Haskell, семейство языков ML.

Противопоставляется парадигме императивного программирования, которая описывает процесс вычислений как последовательное изменение состояний (в значении, подобном таковому в теории автоматов). При необходимости, в функциональном программировании вся совокупность последовательных состояний вычислительного процесса представляется явным образом, например, как список.

Функциональное программирование предполагает обходиться вычислением результатов функций от исходных данных и результатов других функций, и не предполагает явного хранения состояния программы. Соответственно, не предполагает оно и изменяемость этого состояния (в отличие от императивного, где одной из базовых концепций является переменная, хранящая своё значение и позволяющая менять его по мере выполнения алгоритма).

Функциональное программирование является одной из парадигм, поддерживаемых языком программирования Python. Основными предпосылками для полноценного функционального программирования в Python являются: функции

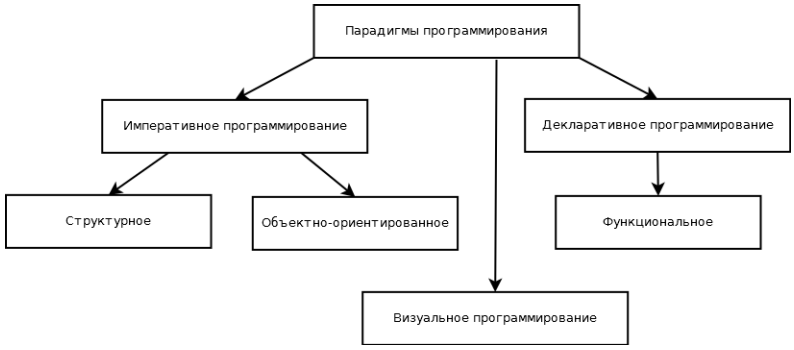


Рис. 1.1. Парадигмы программирования

высших порядков, развитые средства обработки списков, рекурсия, возможность организации ленивых вычислений. Элементы функционального программирования в Python могут быть полезны любому программисту, так как позволяют гармонично сочетать выразительную мощность этого подхода с другими подходами.

Визуальное программирование — способ создания программы для ЭВМ путём манипулирования графическими объектами вместо написания её текста. Визуальное программирование часто представляют как следующий этап развития текстовых языков программирования. Наглядным примером может служить среда разработки Delphi, сделанная для языка Object Pascal, где редактируются графические объекты: форма, кнопки, метки. В последнее время визуальному программированию стали уделять больше внимания в связи с развитием мобильных сенсорных устройств (КПК, планшеты).

С Python поставляется библиотека tkinter на основе Tcl/Tk для создания кроссплатформенных программ с графическим интерфейсом. Существуют расширения, позволяющие использовать все основные библиотеки графических интерфейсов: wxPython, основанное на библиотеке wxWidgets, PyGTK для Gtk, PyQt и PySide для Qt и другие. Некоторые из них также предоставляют широкие возможности по работе с базами данных, графикой и сетями, используя все возможности библиотеки, на которой основаны.

1.3 Типизация в языках программирования

Все данные в компьютере хранятся в виде последовательностей нулей и единиц подряд. Для удобства эти последовательности группируют по 8 цифр подряд и такую группу называют байтом (два байта называются машинным словом). Однако оперировать последовательностями битов напрямую при написании больших программ неудобно, поэтому вводят дополнительные договорёности о спо-

собе интерпретации отдельных байтов в памяти. Эти договорённости и можно назвать типами данных. Все языки программирования можно разделить на:

- нетипизированные (бестиповые),
- типизированные.

Нетипизированными являются языки ассемблера, а также язык программирования встраиваемых устройств Forth. По сути, бестиповые — это наиболее низкоуровневые языки, предоставляющие прямой доступ к манипулированию отдельными битами прямо в регистрах процессора. Все компилируемые и интерпретируемые широко используемые языки, такие как Pascal, C, Python, PHP и другие, являются типизированными.

У отсутствия типизации есть некоторые преимущества:

- Полный контроль над действиями компьютера. Компилятор или интерпретатор не будет мешать какими-либо проверками типов, запрещая те или иные действия.
- Получаемый код обычно имеет высокую эффективность, которая, правда, зависит в первую очередь от квалификации программиста.
- Прозрачность инструкций. При знании языка обычно нет сомнений, что из себя представляет тот или иной код.

Недостатки отсутствия типизации:

- Сложность написания программы быстро растёт с ростом необходимой абстракции и общности. Даже операции с такими, казалось бы, несложными объектами, как списки, строки или записи, уже требуют существенных усилий.
- Отсутствие проверок и как следствие огромное число трудноуловимых ошибок на этапе компиляции. Любые бессмысленные действия, например вычитание указателя на массив из символа будут считаться совершенно нормальными.
- Высокие требования к квалификации программиста и фактическим знаниям об архитектуре целевой ЭВМ.

Типизированные языки делятся ещё на несколько пересекающихся категорий.

1. *Сильная/слабая* типизация (также иногда говорят строгая / нестрогая). Сильная типизация означает, что язык не позволяет смешивать в выражениях различные типы и не выполняет автоматические неявные преобразования, например, нельзя вычесть из строки множество. Языки со слабой типизацией выполняют множество неявных преобразований автоматически,

даже если может произойти потеря точности или преобразование неоднозначно. В действительности почти все популярные языки: C, Java, Python, Pascal и другие имеют условно сильную типизацию, позволяя некоторые автоматические преобразования типов. Самые распространённые примеры: автоматическое приведение целых чисел к действительным и действительных к комплексным, а также символов к строкам. Крайний случай слабой типизации — отсутствие типов вообще.

Преимущества сильной типизации:

- Надежность — вместо неправильного поведения вы получите исключение или ошибку компиляции.
- Скорость — преобразования типов могут быть довольно затратными, с сильной типизацией необходимо писать их явно, что заставляет программиста как минимум знать, что этот участок кода может быть медленным, или избегать их.
- Понимание работы программы — опять же, вместо неявного приведения типов программист пишет все сам, а, значит, примерно понимает, что сравнение строки и числа происходит не само собой и не по волшебству, а использовать действительнозначную переменную в качестве счётчика цикла опасно из-за ошибок округления.
- Определенность — когда вы пишете преобразования вручную, вы точно знаете, что вы преобразуете и во что. Также вы всегда будете понимать, что такие преобразования могут привести к потере точности, затратам машинного времени или стать причиною логической ошибки.

Преимущества слабой типизации:

- Удобство использования смешанных выражений (например, комбинирование целых и вещественных чисел).
- Скорость разработки: не нужно тратить время на написание большого числа явных преобразований типов.
- Краткость записи.

2. ***Явная/неявная*** типизация. Явно-типизированные языки отличаются тем, что тип новых переменных/функций/их аргументов нужно писать явно. Соответственно, языки с неявной типизацией перекладывают эту задачу на компилятор/интерпретатор, такой способ называется автоматическим выведением типов. Все компилируемые языки — наследники ALGOL 60 – имеют явную типизацию. Это C, C++, D, Java, C#, Pascal, Modula 2, Ada и другие. Напротив, языки семейства ML (Standard ML и Ocaml), Haskell, почти все интепретируемые языки: Pyhton, Ruby, Perl, PHP, JavaScript, Lua имеют неявную.

   Преимущества *явной* типизации:

- Многие логические ошибки, ведущие к неверному приведению типов, можно отловить на этапе компиляции. Либо эти ошибки вовсе не возникают, поскольку попытка выписать тип выражения приводит к мысли, что само выражение неверно.
- Знание того, какого типа значения могут храниться в конкретной переменной, снимает необходимость помнить это при отладке и дальнейшей модификации программы.
- Существенно упрощается написание компиляторов, поскольку компилятор не должен уметь определять тип переменной. Как следствие, часто можно произвести ряд дополнительных оптимизаций уже на этапе компиляции автоматически.

Преимущества *неявной* типизации:

- Сокращение записи (сравните Python и Pascal):
```
def add(x, y):
function add(x: real; y: integer): real;
```
- Полиморфизм (универсальность). Одна и та же функция может быть написана для переменных разных типов, если используемые в ней операции определены для них. В языках с явною типизацией в такой ситуации приходится писать много одинаковых функций, отличающихся только типом аргументов и результата (это называется перегрузкою функции), либо эмулировать неявную типизацию за счёт шаблонов и генериков.

3. ***Статическая/динамическая*** типизация. Статическая типизация определяется тем, что конечные типы переменных и функций устанавливаются на этапе компиляции. Т.е. уже компилятор на 100% уверен, какой тип, где находится. В динамической типизации все типы выясняются уже во время выполнения программы. Примеры языков со статическою типизацией: C, Java, C#, Ada, C++, D, Pascal. Примеры языков с динамическою типизацией: Python, JavaScript, Ruby, PHP, Perl, JavaScript, Lisp.

   При *статической* типизации параметр подпрограммы и возвращаемое значение функции связывается с типом в момент объявления и тип не может быть изменён позже (переменная или параметр будут принимать, а функция — возвращать значения только этого типа). Некоторые статически типизированные языки позже получили возможность также использовать динамическую типизацию при помощи специальных подсистем. Например, тип `Variant` в Delphi, `Data.Dynamic` в Haskell, C# поддерживает псевдо-тип `dynamic`.

   Преимущества *статической* типизации:

   - Статическая типизация даёт самый простой машинный код.

- Многие ошибки исключаются уже на стадии компиляции.
- Статическая типизация хороша для написания сложного, но быстрого кода.
- В интегрированной среде разработки осуществимо автодополнение (среда разработки сама догадывается и дописывает часть кода за программиста), особенно если типизация — строгая статическая: множество вариантов можно отбросить как не подходящие по типу.
- Чем больше и сложнее проект, тем большее преимущество дает статическая типизация, и наоборот.

Недостатки *статической* типизации:

- Языки с недостаточно проработанной математической базой оказываются довольно многословными: каждый раз надо указывать, какой тип будет иметь переменная. В некоторых языках есть автоматическое выведение типа, однако оно может привести к трудноуловимым ошибкам.
- Тяжело работать с данными из внешних источников (например, в реляционных СУБД/ десериализация данных).

При *динамической* типизации переменная связывается с типом в момент присваивания значения, а не в момент её объявления (как правило, она вообще не объявляется нигде до момента первого использования). Таким образом, в различных участках программы одна и та же переменная может принимать значения разных типов.

Преимущества *динамической* типизации:

- Упрощается написание несложных программ.
- Облегчается работа прикладного программиста с СУБД, которые принципиально возвращают информацию в «динамически типизированном» виде. Поэтому динамические языки ценны, например, для программирования веб-служб.
- Иногда требуется работать с данными переменного типа. Например, может понадобиться выдать массив или одно число, или вернуть специальное значение типа «ничто». В языке со статическою типизацией такое поведение приходится эмулировать: одно число заменять массивом размером в 1 элемент; при возможности появления особого значения — вводить так называемые «вариантные типы», как сделано в OCaml.

Недостатки *динамической* типизации:

- Статическая типизация позволяет уже при компиляции заметить простые ошибки «по недосмотру». Для динамической типизации требуется как минимум выполнить данный участок кода.

- Особенно коварны в динамическом языке программирования опечатки: разработчик может несколько раз просмотреть неработающий код и ничего не увидеть, пока наконец не найдёт набранный с ошибкой идентификатор.
- Не действует либо действует с ограничениями автодополнение в среде разработки: трудно или невозможно понять, к какому типу относится переменная, и вывести набор её полей и методов.
- Низкая скорость, связанная с динамической проверкой типа, и большие расходы памяти на переменные, которые могут хранить «что угодно». К тому же большинство языков с динамической типизацией интерпретируемые, а не компилируемые.
- Невозможна перегрузка процедур и функций по типу данных — только по количеству операндов (правда, она, как правило, и не требуется).

В действительности, практически все языки, имеющие сильную типизацию, допускают некоторые послабления. Например, Pascal и D допускают смешивание в одном выражении целых и действительных чисел (но результат обязан быть действительным), строк и символов (результат обязан быть строкою), то есть допускают сведение типа к более общему. Аналогично C хотя и относят к языкам со слабою в целом типизацией (можно смешивать в одном выражении логические переменные, числа, указатели и строки), всё же не лишён ряда проверок типов.

Таблица 1.2. Типизация в языках программирования

JavaScript	Динамическая	Слабая	Неявная
Ruby	Динамическая	Сильная	Неявная
Python	Динамическая	Сильная	Неявная
Java, C#	Статическая	Сильная	Явная
PHP	Динамическая	Слабая	Неявная
C, C++, Objective-C	Статическая	Слабая	Явная
Perl	Динамическая	Слабая	Явная
Haskell, Ocaml, Standard ML	Статическая	Сильная	Неявная
Lisp	Динамическая	Сильная	Неявная
D	Статическая	Сильная	Явная
Fortran 90/95/2003/2008	Статическая	Сильная	Явная
Pascal, ObjectPascal/Delphi, Ada	Статическая	Сильная	Явная

1.4 Области программирования

Следует понимать, что в сложившейся в наши дни программной индустрии различные языки программирования заняли разные ниши. Некоторые из них достигли успеха именно благодаря специализации, яркие примеры: JavaScript и PHP. Другие, как Python и Java — существенно более универсальны и получили признание и распространение за счёт возможности сходными средствами решать разные задачи. Но ни один современный язык, в том числе широко рекламируемые C# и C++, не может эффективно использоваться для решения любых задач.

В настоящее время программирование применяется в самых различных областях человеческой деятельности, таких как:

1. *Системное программирование*
   Написание операционных систем, компиляторов, интерпретаторов, виртуальных машин. В этой области требования к быстродействию и потреблению памяти очень велики, а создание переносимых программ затруднено необходимостью тесно и напрямую взаимодействовать с конкретным оборудованием («железом»). Основные языки программирования в этой области: ассемблер, а также компилируемые языки, компиляторы которых написаны на них самих методом постепенной самораскрутки (всегда имеют платформозависимое ассемблерное ядро): C, C++, Objective C, Pascal, Ada.

2. *Программирование встраиваемых устройств*
   Создание операционных систем и прикладных программ для разных «малых» вычислительных машин: станков с программным управлением, сетевых маршрутизаторов, модемов, автомобильной и авиационной электроники. По сути, эта область примыкает к системному программированию и потому здесь используются примерно те же средства: ассемблер, Forth, некоторые компилируемые языки.

3. *Программирование видеокарт*
   Видеоускорители имеют весьма специфические аппаратные особенности: они не могут работать напрямую с устройствами ввода/вывода, не могут сами динамически выделять память, часто способны работать эффективно только с действительными числами одинарной точности (4 байта), эффективно могут выполнять одинаковые инструкции над разными данными, но очень теряют в производительности при необходимости глобальной проверки условий и частой синхронизации потоков. Поэтому для них созданы специализированные языки: OpenCL и CUDA.

4. *Программирование высоко нагруженных серверов*
   Задача состоит в управлении большим числом (часто 10 тысяч и более в секунду) запросов, поступающих как локально с этого же компьютера, так и,

главным образом, по сети. По запросам необходимо производить некоторые вычисления и/или поиск в базах данных. На первом месте в таких задачах стоит надёжность: сбой работы над одним из запросов не должен приводить к краху исполнения всех остальных или полной остановке сервера. На втором месте — производительность, в том числе способность, не снижая существенно производительности на 1 поток, обрабатывать одновременно много потоков с использованием нескольких вычислительных ядер или даже нескольких физически разнесённых ЭВМ (это свойство называется масштабируемостью). Основные языки здесь: Java, C#, Erlang, то есть языки, использующие виртуальные машины и имеющие достаточно высокие возможности абстрагирования (ООП), что позволяет локализовать многие ошибки времени исполнения. Реже используется C++, поскольку, несмотря на высокую производительность и широкие возможности, программы на C++ часто приводят к некорректной работе с памятью. В последнее время популярны Scala и Go в качестве замены Java, поскольку Scala позволяет писать более лаконичный и сложный код частично в функциональном стиле, а Go прост, поддерживает очень эффективную модель многопоточных вычислений и эффективно компилируется в машинный код.

5. *Программы для работы с базами данных*
   Эта область частично пересекается с предыдущей, но затрагивает также клиентские программы, где требования к скорости и надёжности работы не такие жёсткие. Программы в этой области, как правило, сочетают в себе две части. На одном языке написана высокоуровневая обёртка, с которою взаимодействует пользователь. Для её написания часто используются 1C, C#, Delphi, а также многие интерпретируемые языки, в первую очередь Python и Ruby. Вторая часть отвечает за непосредственное взаимодействие с базою данных и написана на одном из диалектов языка запросов SQL.

6. *Системное администрирование*
   Задача системного администратора — автоматизация основных работ по обслуживанию серверов. Это резервное копирование данных, установка обновлений, а также новых программ и библиотек, восстановление после сбоя, синхронизация разных серверов в кластере, запуск различных задач разных пользователей и их распределение по отдельным процессорным ядрам. Персональному компьютеру системный администратор почти не нужен, все основные действия по поддержанию компьютера в работоспособном состоянии производит сам пользователь. Долгое время основным языком системных администраторов был shell script, но в настоящее время языки общего применения, в первую очередь Python, также стали активно применяться, поскольку позволяют, владея на высоком уровне одним языком, совмещать работу системного администратора с работою, например, веб-программиста или программиста баз данных.

7. *Написание графических интерфейсов пользователя*
   В этой области очень большое распространение получила парадигма ООП и парадигма визуального программирования. Пишут на многих языках, как компилируемых: C++, Object Pascal, Vala, так и интерпретируемых: Python, Tcl, Ruby. Java и C# также иногда используются в данной области.

8. *Веб-программирование*
   Написание программ, работающих в браузере, начиная от простых сайтов и заканчивая сложными компьютерными играми, имеет определённую специфику. В настоящее время здесь используются все основные скриптовые языки: PHP, Python, Ruby (на платформе Rails). Наибольшую популярность имеет JavaScript, поскольку его виртуальная машина хорошо оптимизирована по производительности и потреблению памяти во всех популярных браузерах.

9. *Компьютерные игры*
   Уже долгое время индустрия компьютерных игр является локомотивом развития как аппаратных средств: центральных процессоров и особенно видеокарт, так и концепций и языков программирования. Первоначально игры писались на системных языках и мало отличались от прочих программ, но впоследствии именно в игростроении наибольшее распространение получила концепция объектно-ориентированного программирования. В настоящее время только самые критичные для производительности части пишутся на высокопроизводительных языках вроде C++, большая же часть программной логики и управляющих скриптов, графический интерфейс пользователя, и даже многие базовые части пишут на интерпретируемых языках, самым популярным из которых здесь является Python. Основная причина этого — необходимость соблюдать сроки: времени на разработку игр нужно много, но самая лучшая и надёжная игра потерпит фиаско на рынке, если опоздает даже на 2–3 года.

10. *Научное программирование*
    Учёные долгое время были одними из основных потребителей ЭВМ. Для них был создан первый компилируемый язык — Fortran, который и в настоящее время используется в случае, когда производительность программ имеет ключевое значение. Однако возможности современных компьютеров оказались столь велики, что избыточны для решения большинства задач с точки зрения производительности и объёма памяти. В результате наибольшее признание в последние 20 лет получили языки интерпретируемого типа, глубоко интегрированные со средствами разработки, библиотеками алгоритмов и средствами построения графиков. Такие интегрированные системы условно называют «пакетами». Наиболее известными примерами таких систем являются коммерческие MATLAB, Mathematica, Stasistica, а также бесплатные/свободные R, SciLab, GNU Octave. Единственный язык общего назначения, в настоящее время не только сохранивший свою привле-

кательность, но и успешно теснящий математические пакеты, в том числе и коммерческие, — это Python. Произошло это благодаря простоте и понятности языка с одной стороны, и наличию очень хороших и высокопроизводительных библиотек алгоритмов и средств для построения графиков. Есть и проекты создания специализированного научного языка, самым популярным и развитым из которых является Julia.

1.5 Области применения Python

Будучи удачно спроектированным языком программирования, Python прекрасно подходит для решения ежедневных реальных задач. Он имеет самый широкий спектр применений: как инструмент управления другими программными компонентами и для реализации самостоятельных программ. Фактически, круг ролей, которые может играть Python как многоцелевой язык программирования, не включает только области встроенных устройств и системного программирования, где ограничения на использование памяти и требования к скорости исполнения настолько велики, что время и удобство написания программы не играют существенной роли, причём можно нанять программистов сколь угодно высокой квалификации.

За счёт чего Python получил столь широкое распространение? Python имеет огромное количество высококачественных уже готовых модулей, распространяемых бесплатно, которые вы можете использовать в любой части программы. В модуле уже реализованы многие нужные вам детали программы. Написание программы с использованием уже готовых модулей можно сравнить со строительством сборного каркасного дома: отдельные детали: фундамент, стены, крыша, коммуникации уже сделаны до вас, вам нужно только выбрать подходящие детали и собрать вместе. Модули подключаются при помощи команды `import`, которая присутствует в начале каждого примера.

Все широко используемые модули делятся на две основные части: модули стандартной библиотеки, поставляемые вместе с интерпретатором Python (эти модули «всегда с вами»), и внешние модули, для которых существуют средства установки.

Установка внешних модулей может быть осуществлена разными путями: в Linux все популярные модули доступны для установки штатными средствами (например, через «Центр установки и обновления программ» в Ubuntu), для Window и MacOS X доступны скомпилированные установочные файлы (например, exe или msi для Windows). Можно также использовать возможности штатного установщика внешних модулей pip, входящий в состав стандартных модулей. С его помощью можно установить почти любой, даже редко используемый внешний модуль, для которого нет скомпилированных пакетов под Linux или установщиков под Windows. Недостатком последнего подхода является то, что для установки модулей, написанных на других языках, например, C или Fortran, pip требует наличия в системе компилятора этих языков, причём не абы какого, а совместимого с тем, что использовал разработчик.

1.5.1 Системное администрирование

Встроенные в Python интерфейсы доступа к службам операционных систем (например, половина графического интерфейса Ubuntu написана на Python) делают его идеальным инструментом для создания переносимых программ и утилит системного администрирования (иногда они называются инструментами командной оболочки). Программы на языке Python могут:

- Создавать, удалять, отыскивать, сортировать, перебирать файлы и каталоги в любой системе. Например, в Linux и MacOS разделительным знаком при записи пути к файлу является «/», а в Windows — «\». Программа на Python будет работать и там, так как умеет заменять слэши. Так же в Linux и MacOS есть один главный диск, а в Windows их может быть много (C, D, E). Python автоматически подставляет над этими логическими дисками один общий корень. Для этого используется стандартный модуль `os`.

  Пример:

  ```
  import os # загружаем модуль
  # Создаём список всех файлов и папок в текущей папке:
  filesdirs = os.listdir(".")
  # Печатаем имена только файлов:
  for fd in filesdirs:
      if os.path.isfile(fd):
          print(fd,'это файл')
  # Проверяем, есть ли в папке folder1
  if not os.path.exists('Folder1'):
      # Если её нет, создаём её
      os.mkdir('Folder1')
  ```

- Запускать другие программы. Например, автоочистку корзины или автоустановку программ. Для этого используются стандартные модули sys, os, subprocess. Пример, в котором из Python запускается популярный бесплатный редактор изображений Gimp, причём команда запуска выбирается в зависимости от типа операционной системы:

```
import sys
import subprocess
if sys.platform == 'win32':
    subprocess.call(['C:/Program␣Files/GIMP␣2/bin/gimp-2.8.exe'])
elif sys.platform == 'linux':
    subprocess.call(['gimp'])
```

- Производить параллельные вычисления с использованием нескольких процессов и потоков, для чего используется стандартный модуль `multiprocessing`.

- Осуществлять проверку имён пользователей и паролей на соблюдение политики безопасности и делать многое другое.

При этом стандартная библиотека Python поддерживает все типичные инструменты операционных систем: переменные окружения, файлы, сокеты, каналы, процессы, многопоточную модель выполнения, поиск по шаблону с использованием регулярных выражений, аргументы командной строки, стандартные интерфейсы доступа к потокам данных, запуск команд оболочки, дополнение имен файлов и многое другое.

1.5.2 Написание графических интерфейсов пользователя

Простота Python и высокая скорость разработки делают его отличным средством разработки графического интерфейса. В состав Python входит стандартный модуль `tkinter`, позволяющий программам на языке Python реализовать переносимый графический интерфейс с внешним видом, присущим операционной системе. Графические интерфейсы на базе `Python/tkinter` без изменений могут использоваться в MS Windows, X Window (в операционных системах UNIX и Linux) и Mac OS (как в классической версии, так и в OS X).

Напишем простенькую программу для создания графического интерфейса с кнопкой, надписью и полем ввода (рис. 1.2):

```
from tkinter import * # подключение модуля tkinter
root = Tk() # создание главного окна
btn = Button(root, text = 'Кнопочка', width=10, height=2,
        bg='white',fg='black', font='Arial␣14') # создание кнопки
lab = Label(root, text='Ваша фамилия:',
        font='Arial␣14') # создание надписи
Edit = Entry (root, width=20) # создание поля ввода
btn.pack() # размещение кнопки на форме
lab.pack() # размещение надписи на форме
Edit.pack() # размещение поля ввода на форме
root.mainloop() # отображение главного окна
```

1.5.3 Веб-программирование

Python традиционно используется для написания сложных сайтов. Самым популярным средством для этого служит веб-фреймворк (большой набор модулей) Django. С его помощью написаны некоторые очень известные сайты, включая Instagram и сайт сообщества Mozilla. Django представляет множество различных функций, включая средства для автоматического создания баз данных.

1.5.4 Программы для работы с базами данных

В языке Python имеются интерфейсы доступа ко всем основным реляционным базам данных: Sybase, Oracle, Informix, ODBC, MySQL, PostgreSQL, SQLite и

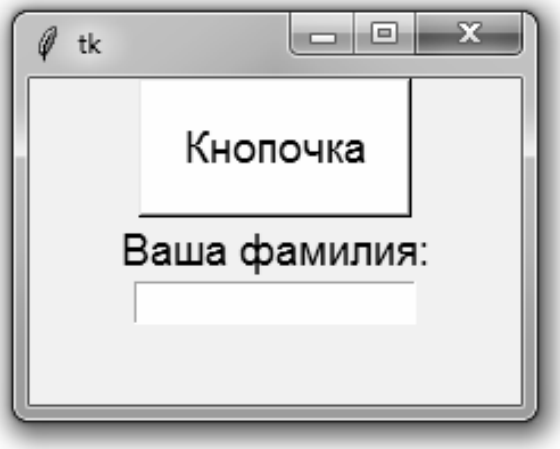


Рис. 1.2. Пример простейшего графического приложения, описанного выше.

многим другим. В мире Python существует также переносимый прикладной программный интерфейс баз данных, предназначенный для доступа к базам данных SQL из сценариев на языке Python, который унифицирует доступ к различным базам данных.

Например, для базы данных SQLite необходимо подключить модуль `sqllite3` (`import sqlite3`). Вот небольшая программа, которая создаёт соединение с базой данных, если БД не существует, то она будет создана, иначе файл будет открыт:

```
import sqlite3
conn = sqlite3.connect('data.db')
cr = conn.cursor()
cr.execute("""CREATE TABLE IF NOT EXISTS 'romanus'
            ('numerus' INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
             'nomen' TEXT NOT NULL, 'praenomen' TEXT,
             'cognomen' TEXT)""")
cr.execute("""INSERT INTO romanus VALUES
            (1, 'Claudius','Tiberius','Nero')""")
conn.commit()
conn.close()
```

В базу заносится одна запись.

1.5.5 Игры, искусственный интеллект

Python используется для разработки многих популярных игр. Ещё в первой половине 2000-ых Python стал основным средством для написания внутренней логики четвёртой игры серии Civilization. Сейчас число игр, содержащих в себе интерпретатор Python и использующих его для реализации программной логики, редакторов сценариев и искусственного интеллекта, исчисляется сотнями. Многие простые игры, браузерные игры и игры для мобильных устройств используют модуль `pygame`, предоставляющий простой и удобный доступ к библиотекам трёхмерной графики OpenGL и управления звуком OpenAL.

1.5.6 Программирование математических и научных вычислений

Python представляет собою удачный компромисс между языком общего назначения и математическими пакетами. Сам по себе «чистый» Python пригоден только для несложных вычислений.

Ключевая особенность Python — его расширяемость. Это, пожалуй, самый расширяемый язык из получивших широкое распространение. Как следствие этого, для Python не только написаны и приспособлены многочисленные библиотеки алгоритмов на C и Fortran, но и имеются возможности использования других программных средств и пакетов, в частности, R и SciLab, а также графопостроителей, например, Gnuplot и PLPlot.

Ключевыми модулями для превращения Python в математический пакет являются `numpy`, `matplotlib` и `scipy`. Кроме них популярностью пользуются `sympy` для символьных вычислений, `ffnet` для построения искусственных нейронных сетей, `pyopencl`/`pycuda` для вычисления на видеокартах и некоторые другие. Возможности `numpy` и `scipy` покрывают практически все потребности в математических алгоритмах.

Одним из важнейших преимуществ Python является то, что все известные его реализации, дополнительные специальные модули, в том числе `numpy`, `scipy` и `matplotlib`, а также большинство сред разработки распространяются свободно. Это означает возможность всегда иметь любимое средство разработки под рукою.

1.6 Первая программа. Среда разработки. Интерактивный и скриптовый режим. Примеры решения заданий

1.6.1 Установка Python

Установка Python на компьютер зависит от используемой операционной системы. Существуют несколько основных подходов. Нужно понимать, что следует различать базовую установку, включающую интерпретатор, среду разработки IDLE, а также стандартную библиотеку, и установку дополнительных модулей, которых для Python написано очень много.

Таблица 1.3. Способы установки Python

Базовая установка	**Дополнительные модули**	**ОС**
установщик с официального сайта `https://python.org`	встроенный механизм pip, любые	Linux, Windows, MacOS X
использование специальных сборок: WinPython, Pyzo, Anakonda и др.	частично встроены, расширение затруднительно	зависит от сборки
установка штатными средствами ОС	зависит от типа и версии ОС	Linux, MacOS X

Все эти способы имеют свои преимущества и недостатки:

- При установке с официального сайта в Linux и MacOS X вам придётся столкнуться с тем, что у вас будут 2 частично пересекающихся интерпретатора Python в системе. Дело в том, что эти две ОС частично используют Python в своих целях и какой-то (часто не самый свежий) интерпретатор поставляется в комплекте. В результате, замена его свежим интерпретатором с официального сайта может частично нарушить работу ОС (конечно, этого можно не допустить или поправить, но для новичка такой подход может стать фатальным). В Windows нет своего Python по умолчанию, поэтому установка базового функционала пройдёт штатно, но есть другая проблема: многие полезные модули содержат код на других языках: в первую очередь это Fortran и C, а также могут зависеть от внешних библиотек, также написанных на других языках, например, библиотеки линейной алгебры Lapack или графической библиотеки QT. В Linux есть штатная возможность установить все нужные компиляторы и библиотеки средствами самой ОС, в меньшей степени эта же возможность есть в Mac OSX, но Windows здесь не предоставляет почти ничего, всё придётся искать и ставить своими руками.

- При использовании специализированных сборок вы получаете готовую и настроенную среду программирования со множеством установленных модулей помимо стандартной библиотеки. Но если вы захотите что-то сверх того, что там есть, вам придётся сильно помучаться. Часть сборок, например WinPython, ориентированы на определённую ОС.

- Установка штатными средствами ОС (через менеджер пакетов, например Synaptic в Debian/Ubuntu/AltLinux) — лучший выбор пользователя Linux, так как все устанавливаемые таким образом модули будут работать штатно почти наверняка, все необходимые библиотеки и компиляторы будут автоматически установлены и правильных версий. Редкие недостающие пакеты, как правило, можно доставить через pip. Но в MacOS X такой способ сложно рекомендовать, поскольку число шатано доступных пакетов невелико, а

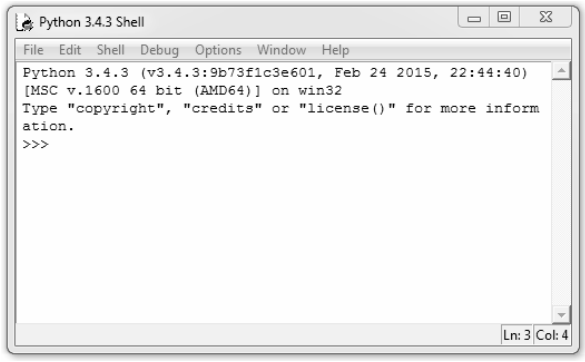


Рис. 1.3. Интерактивный режим среды разработки IDLE.

сами они часто очень древних версий. В Windows такой способ вовсе невозможен.

Суммируя выше сказанное, мы будем рекомендовать пользователям Linux пользоваться своим шатаным менеджером пакетов, а пользователям Windows — использовать стандартную сборку WinPython, включающую модули для математических и инженерных расчетов, построения графиков и многие другие, с сайта `http://winpython.github.io/`.

Помните, что Python и модули к нему — свободное программное обеспечение. Вы можете выбирать способ установки и нужные вам модули наиболее удобным для вас способом и не думать ни о какой плате и лицензионных отчислениях.

1.6.2 Интерактивный режим и первая программа

После загрузки и установки Python открываем IDLE (среда разработки на языке Python, поставляемая вместе с дистрибутивом). Запускаем IDLE (изначально запускается в интерактивном режиме). Далее последует приглашение к вводу (`>>>`).

Теперь можно начинать писать первую программу. Традиционно, первой программой у нас будет «Hello, world». Чтобы написать «Hello, world» на Python, достаточно всего одной строки:

```
>>> print('Hello, World')
```

Функция `print` выводит данные на экран.

Интерпретатор выполняет команды построчно: пишешь строку, нажимаешь `<Enter>`, интерпретатор выполняет ее, наблюдаешь результат. Это очень удобно, когда человек только изучает программирование или тестирует какую-нибудь небольшую часть кода. Ведь если работать на компилируемом языке, то при-

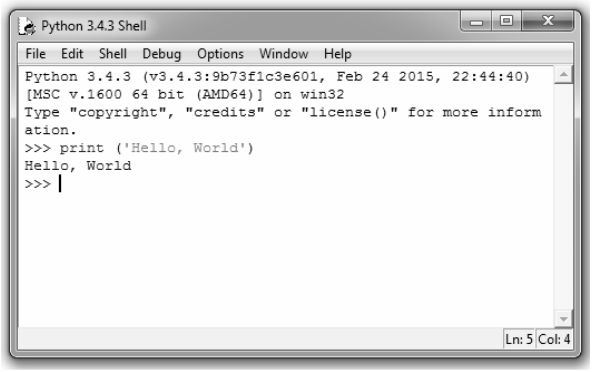


Рис. 1.4. Вывод надписи «Hello, world» в интерактивном режиме в среде разработки IDLE.

шлось бы сначала написать код на исходном языке программирования, затем скомпилировать, и уже потом запустить исполняемый файл на выполнение.

Кстати, если интерпретатору Python дать команду `import this` (импортировать «сам объект» в себя), то выведется так называемый «Дзен Python», иллюстрирующий идеологию и особенности данного языка. Считается, что глубокое понимание этого дзена приходит тем, кто сможет освоить язык Python в полной мере и приобретет опыт практического программирования.

Хотя интерактивный режим будет вам ещё не раз полезен при написании и отладке программ и тестировании возможностей языка, всё же он не является основным. В большинстве случаев программист сохраняет код в файл, и запускать уже файл. Такой режим работы называется скриптовый. Файлы с кодом на Python обычно имеют расширение py.

Для того, чтобы создать новое окно для написании скрипта, в интерактивном режиме IDLE выберите `File` → `New File` (или нажмите `<Ctrl> + N`). В открывшемся окне попробуйте ввести следующий код:

```
name = input ('Как вас зовут? ')
print ('Привет,', name)
```

Функция `input` считывает данные, введённые с клавиатуры, и записывает их в переменную `name`.

Первая строка печатает вопрос («Как вас зовут?»), ожидает, пока вы напечатаете что-нибудь и нажмёте `<Enter>` и сохраняет введённое значение в переменной `name`.

Во второй строке используется функция print для вывода текста на экран, в данном случае для вывода `"Привет, "` и того, что хранится в переменной `name`.

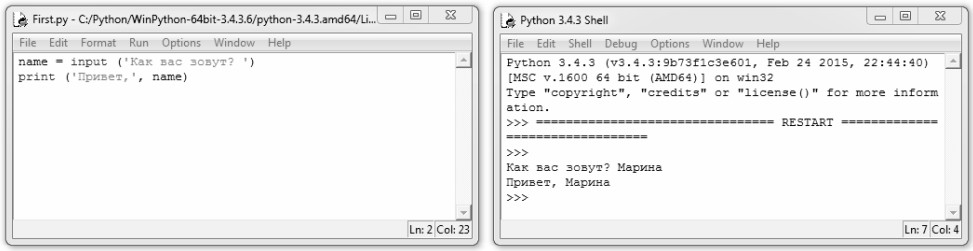


Рис. 1.5. Скриптовый и интерактивный режим: в интерактивном режиме можно видеть результаты выполнения скрипта.

Теперь нажмём F5 (или выберем в меню IDLE `Run` → `Run Module`) и убедимся, что написанное работает. Перед запуском IDLE предложит нам сохранить файл. Сохраним туда, куда вам будет удобно, после чего программа запустится.

Для «сложения» строк можно также воспользоваться оператором `+`:

```
print ('Привет, ' + str(name) + '!').
```

1.6.3 Задания

Задание 1 (Ввод/вывод данных) Здесь собраны задачи по сути учебные, но полезные. Их выполнение позволит вам «набить руку» и выработать необходимые навыки. Задания необходимо выполнить в интерактивном и скриптовом режимах. *Выполнять следует все и по порядку.*

Напишите программу:

1. последовательно запрашивающую ваши фамилию, имя, отчество и выводящую их одной строкой в последовательности: фамилия → имя → отчество;

2. последовательно запрашивающую ваши фамилию, имя, отчество и выводящую их одной строкой в последовательности: имя → отчество → фамилия;

3. преобразующую простую русскую фамилию в мужском роде в женский род (Петров → Петрова; Путин → Путина);

4. последовательно введите число, месяц, год рождения; выведите дату своего рождения через точки (01.01.2000), слэши (01/01/2000), пробелы (01 01 2000), тире (01-01-2000).

Глава 2

Основные типы данных

В Python имеется множество встроенных типов данных. Все типы делятся на простые и составные. Переменные[1] простых типов состоят из единственного значения, к ним относятся числа (всего 3 типа: целые, действительные и комплексные) и логические переменные. Переменные составных типов состоят из набора значений и, в свою очередь, делятся на неизменяемые, для которых нельзя изменять значения элементов, добавлять или изымать элементы, и изменяемые, для которых всё это можно делать. К неизменяемым типам относятся строки и кортежи, к изменяемым — списки, словари и множества.

Чтобы получить корректно работающую программу, важно понимать, к какому типу относится переменная. В Python тип переменной не объявляется, а автоматически определяется при присвоении ей значения.

2.1 Числа. Арифметические операции с числами. Модуль `math`

Числа в Python делятся на:

- целые,
- действительные (с плавающей точкой размером в 64 бита),
- комплексные (с плавающей точкой размером в 128 бит).

[1] Строго говоря, «переменных» в классическом смысле в Python нет, а есть *имена*, связанные с некоторыми *объектами* в памяти. Но использование термина «переменная» удобно и принято в программировании. Под переменными мы будем понимать именно сами объекты. Называть переменными имена неудобно, так как на протяжении работы программы одно и то же имя может быть сопоставлено разным объектам, поэтому получилось бы, что переменные имеют переменный тип или не имеют его вовсе, что создаёт путаницу, так как типы активно используются в Python. Для краткости, однако, мы будем иногда употреблять выражения типа «переменная `a`», подразумевая объект, имеющий в данный момент имя `a`, в тех случаях, когда это не создаёт путаницы.

Python поддерживает динамическую типизацию, то есть тип переменной определяется только во время исполнения. Переменная может быть переопределена, при этом её тип изменится:

```
>>> a=2
>>> a
2
>>> a=2.0
>>> a
2.0
>>> a=2+3j
>>> a
(2+3j)
```

Инструкция `a = 2` создаёт числовой объект-переменную с целочисленным значением 2 и присваивает ему имя `a`. Далее инструкция `a = 2.0` создаёт новый числовой объект с действительным значением 2.0 и присваивает уже ему имя `a`. А объект со значением 2 удаляется из памяти автоматически сборщиком мусора (англ. garbage collector, специальный процесс, периодически освобождающий память, удаляя объекты, которые уже не будут востребованы приложениями), т. к. была потеряна последняя ссылка на этот объект. Затем инструкция `a=2+3j` создаёт числовой объект с комплексным значением и всё с тем же именем, а объект с действительным значением удаляется.

В интерактивном режиме есть возможность быстро вызвать предыдущую команду сочетанием `<Alt> + p`. Поэтому легко можно поправить `a=2` на `a=2.0`.

При операциях с числами существует общее правило: результат будет того же типа, что и операнды, если операнды разных типов, то результат будет приведён к наиболее общему из всех имеющихся. Порядок общности естественный: `целые` → `действительные` → `комплексные`. Следующий пример иллюстрирует это правило:

```
>>> a=2
>>> b=2+3j
>>> a+b
(4+3j)
```

Так же полезно запомнить, что для проверки типа любого значения и любой переменной можно использовать функцию `type()`:

```
>>> a=5
>>> b=17.1
>>> c=4+2j
>>> type(a); type(b); type(c)
<class 'int'>
<class 'float'>
<class 'complex'>
```

Приведённая программа имеет особенность: в четвёртой строке записано сразу 3 оператора. Так можно делать, но при этом для разделения операторов нужно использовать «;» (после последнего она не обязательна). Пробелы в большинстве случаев необязательны.

Можно явно преобразовывать значение любого типа с помощью соответствующих функций `int()`, `float()` или `complex()`:

```
>>> a=5
>>> int(a)
5
>>> float(a)
5.0
>>> complex(a)
(5+0j)
```

Важно помнить, что комплексное число нельзя преобразовать с помощью функций `int()` и `float()` к целому или действительному. Функция `int()` отбрасывает дробную часть числа, а не округляет его:

```
>>> a=4.3
>>> int(a)
4
>>> b=-4.3
>>> int(b)
-4
```

А теперь про подводные камни и отличие третьего Python от второго при работе в скриптовом (текстовом) режиме. Напишем в текстовом режиме в Python 3.x простую программу, которая должна складывать два введённых числа и выводить результат на экран:

```
a = input('Введите первое число: ')
b = input('Введите второе число: ')
print(a + b)
```

Ход работы программы:

```
Введите первое число: 10
Введите второе число: 4
104
```

Получили совсем не то, что ожидали. Python версий 2.x поддерживал автоматическое определение типа переменной при вводе с клавиатуры. Однако это часто вызывало различные ошибки или сложности. Поэтому при проектировании версии 3.0 было решено отказаться от автоматического определения типа и всегда считывать данные как строку. Поэтому, если мы хотим действительно сложить два числа, программу придется переписать:

```
a = int(input('Введите первое число: '))
b = int(input('Введите второе число: '))
print(a + b)
```

Вывод программы:

```
Введите первое число: 10
Введите второе число: 4
14
```

Python поддерживает все основные арифметические операторы (см. табл. 2.1).

Таблица 2.1. Арифметические операции с числами и встроенные функции с одним и двумя аргументами

`x + y`	Сложение — сумма x и y (`2 + 3 = 5`)
`x - y`	Вычитание — разность x и y (`5 - 2 = 3`)
`x * y`	Умножение — произведение x и y (`2 * 3 = 6`)
`x / y`	Деление x на y: `4 / 1.6 = 2.5`, `10 / 5 = 2.0`, результат всегда типа `float`
`x // y`	Деление x на y нацело: `11 // 4 = 2`, `11.8 // 4 = 2.0`, результат целый, только если оба аргумента целые
`x % y`	Остаток от деления: `11 % 4 = 3`, `11.8 % 4 = 3.8000000000000007` (присутствует ошибка, связанная с неточностью представления данных в компьютере)
`x ** y`	Возведение x в степень y: (`2 ** 3 = 8`)
`abs(x)`	Модуль числа x
`round(x)`	Округление (`round(11.3) = 11`)
`round(x, n)`	Округляет число x до n знаков после запятой: `round(12.34567, 3) = 12.346`
`pow(x, y)`	полный аналог записи `x ** y`
`divmod(x, y)`	выдаёт два числа: частное и остаток, обращаться следует так: `q, r = divmod(x, y)`

Отметим, что операции сложения, вычитания, умножения и возведения в степень выдают ответ типа `int` только если оба аргмента целые, типа `float`, если один из аргументов действительный, а другой — целый или действительный, и типа `complex`, если хотя бы один аргумент комплексный. Операция возведения в степень также может выдать комплексный результат при возведении отрицательных чисел в степень кроме случая, когда эта степень целая и нечётная. То есть, эти опраторы в Python подчиняются общеупотребительным правилам преобразования типов.

Оператор деления традиционно является «проблемным»: результат его работы в разных языках программирования определяется разными правилами. Для Python версии 3.x деление «/» всегда действительного типа. В Python версии 2.x деление подчинялось другому правилу: если оба операнда целые — результат будет целый, иначе — действительный. Приведём небольшую программу–пример для иллюстрации работы операторов «/», «//» и «%» на Python 3.4.3:

```
>>> a = 5; b = 98
>>> c1 = b/a; c2 = b//a; c3 = b%a
>>> c1; c2; c3
19.6
19
3
```

Таблица 2.2. Встроенные функции с последовательностями или произвольным числом аргументов

`max(a,b,...)`	Максимальное число из двух или более: `max([2, 6, 3]) = 6`
`min(a,b,...)`	Минимальное число из двух или более: `min([2, 6, 3]) = 2`
`max(seq)`	Максимальный элемент последовательности: `max([2,6,3]) =6`
`min(seq)`	Минимальный элемент последовательности: `min([2, 6, 3])=2`
`sum(seq)`	Сумма элементов последовательности, например, кортежа `sum((2, 6, 3)) = 11` или списка `sum([2, 6, 3]) = 11`
`sorted(seq)`	Отсортированный список: `sorted([3, 2, 5, 1, 4]) = [1, 2, 3, 4, 5]`

То, что описанные функции являются встроенными, означает, что они доступны без всяких дополнительных действий. Однако встроенных функций немного, гораздо больше функций находится в *стандартной библиотеке* — наборе *модулей*, поставляемых всегда вместе с интерпретатором Python. Функции для работы с числами находятся в модулях `math` для целых и действительных чисел и `cmath` для комплексных. Сделано это потому, что комплексные числа нужны гораздо реже целых и действительных, а Python часто используется в качестве языка сценариев и в других приложениях, где память нужно экономить. Самые употребительные функции модуля `math` описаны в таблице 2.3.

Загрузка модулей в Python осуществляется с помощью оператора `import`. Самый простой способ его использования — загрузить всё содержимое модуля глобально:

```
from math import *
t = sin(pi/6)
```

Таблица 2.3. Наиболее употребительные функции и константы модуля math

trunc(X)	Усечение значения X до целого
sqrt(X)	Квадратный корень из X
exp(X)	Экспонента числа X
log(X), log2(X), log10(X)	Натуральный, двоичный и десятичный логарифмы X
log(X, n)	Логарифм X по основанию n
sin(X), cos(X), tan(X)	Синус, косинус и тангенс X, X указывается в радианах
asin(X), acos(X), atan(X)	Арксинус, арккосинус и арктангенс X
atan2(X, Y)	арктангенс отношения $\frac{X}{Y}$ с учётом квадранта
degrees(X)	Конвертирует радианы в градусы
radians(X)	Конвертирует градусы в радианы
sinh(X), cosh(X), tanh(X)	Гиперболические синус, косинус и тангенс X
asinh(X), acosh(X), atanh(X)	Обратный гиперболический синус, косинус и тангенс X
hypot(X, Y)	Гипотенуза треугольника с катетами X и Y
factorial(X)	Факториал числа X
gamma(X)	Гамма-функция X
pi	Выдаётся число π
e	Выдаётся число e

```
v = log(e)
print(t, v)
```

Первую строчку можно прочитать дословно: из модуля math импортируем всё («*» означает всё). Такая запись позволяет в теле программы просто обращаться к функциям, лежащим в math, без сложной записи: math.sin.

Но такой способ подойдёт только для первых двух-трёх занятий, на которых будет использоваться один модуль math. При подключении двух, трёх и более модулей может возникнуть такая ситуация, когда в разных модулях лежат функции с одинаковыми названиями (например, open), но делают они разные действия, да и аргументы вызываются в разном порядке. В такой ситуации, да и просто академически правильнее, писать следующим образом:

```
import math
t = math.sin(math.pi/6)
v = math.log(math.e)
print(t, v)
```

Если название модуля слишком длинное, или оно вам не нравится по каким-то другим причинам, то для него можно создать псевдоним с помощью ключевого слова as:

```
import matplotlib.pyplot as plt
plt.plot(x)
```

2.2 Строки

Строки в Python — упорядоченные последовательности символов, используемые для хранения и представления текстовой информации, поэтому с помощью строк можно работать со всем, что может быть представлено в текстовой форме. При этом отдельного символьного типа в Python нет, символ — это строка длины 1. Более того, символы как элементы строки тоже являются строками.

Работа со строками в Python очень удобна. Существует несколько вариантов написания строк:

```
>>> S1 = 'Alice␣said:␣"Hi,␣Anne!"'
>>> S2 = "Anne␣answered:␣'Hi,␣Alice'"
```

Строки в апострофах и в кавычках (иногда говорят «одинарных» и «двойных» кавычках) — это одно и то же. Причина наличия двух вариантов в том, чтобы позволить вставлять в литералы строк символы кавычек или апострофов.

Строки можно писать в тройных кавычках или апострофах. Главное достоинство строк в тройных кавычках в том, что их можно использовать для записи многострочных блоков текста:

```
>>> S = '''Это
длинная
строка'''
>>> S
'Это\nдлинная\nстрока'
>>> print(S)
Это
длинная
строка
```

Внутри такой строки возможно присутствие кавычек и апострофов, главное, чтобы не было трёх кавычек подряд. Символ '\n' означает перевод строки на одну вниз (кнопка <Enter>) и разделяет строки текстовых файлов. Заметим, что в Windows принято использовать 2 разделительных символа подряд: '\r\n', а в Mac OS X — только '\r', но почти все современные редакторы (за исключением «Блокнота» Windows) без труда справляются с файлами, записанными с использованием «чужих» разделителей.

Все строки в Python являются юникодом, то есть разрешено использование любых символов национальных алфавитов, которые вы сможете набрать (и даже

многих, для которых нет соответствующих клавиш на клавиатуре). При этом используется внутреннее представление UTF32, то есть все символы имеют длину 4 байта, что экономит процессорное время, а запись в файл и чтение из файла происходят в кодировке UTF8, что обеспечивает совместимость со старою кодировкою ASCII и уменьшает потребление памяти.

Здесь уместно упомянуть о том, как в Python при написании кода программы делать комментарии. Однострочные комментарии начинаются со знака решетки «#», многострочные — начинаются и заканчиваются тремя двойными кавычками «"""».

Числа могут быть преобразованы в строки с помощью функции `str()`. Например, `str(123)` даст строку `'123'`. Если строка является последовательностью знаков-цифр, то она может быть преобразована в целое число в помощью функции `int()`: `int('123')` даст в результате число 123, а в вещественное с помощью функции `float()`: `float('12.34')` даст в результате число 12.34. Для любого символа можно узнать его номер (код символа) с помощью функции `ord()`, например, `ord('s')` даст результат 115. И наоборот, получить символ по числовому коду можно с помощью функции `chr()`, например `chr(100)` даст результат `'d'`.

2.2.1 Базовые операции над строками

Существуют несколько различных подходов к операциям над строками.

- «Арифметические операции». Для строк подобно числам определены операторы сложения + и умножения ∗. В результате сложения содержимое двух строк записывается подряд в новую строку, например:

```
>>> S1 = 'Py'
>>> S2 = 'thon'
>>> S3 = S1 + S2
>>> print(S3)
Python
```

  Можно складывать несколько строк подряд.

  Умножение определено для строки и целого положительного числа, в результате получается новая строка, повторяющая исходную столько раз, каково было значение числа (возьмём строку `S3` из прошлого примера):

```
>>> S3 * 4
'PythonPythonPythonPython'
>>> 2 * S3
'PythonPython'
```

- Функция `len()` вычислят длину строки, результат имеет целочисленный тип. Например, `len('Python')` выдаст 6.

- Доступ по *индексу*. Можно обратиться к любому элементу (символу) строки по его номеру, нумерация начинается с 0 (первый элемент строки `S` имеет номер 0, последний — `len(S)-1`. Разрешается использовать отрицательные индексы, в этом случае нумерация происходит *с конца*, что можно также интерпретировать как правило: к отрицательным индексам всегда добавляется длина строки, например последний элемент строки чаще всего обозначают как `-1`):

```
>>> S = 'Python'
>>> S[0]
'P'
>>> S[-1]
'n'
```

  Обращение к символу с несуществующим номером порождает ошибку: «`IndexError: string index out of range`».

  При использовании индексов необходимо помнить, что строки в Python относятся к категории неизменяемых последовательностей: нельзя поменять значение того или иного символа, а можно лишь создать новую строку.

```
>>> S = 'Ура'
>>> S[1] = 'x'
Traceback (most recent call last):
  File "<pyshell#68>", line 1, in <module>
    S[1] = 'x'
TypeError: 'str' object does not support item assignment
>>> S = S[0]+'x'+S[2]
>>> S
'Уха'
```

- *Срезы* позволяют скопировать или использовать в выражениях часть строки. Оператор извлечения среза из строки выглядит так: `S[n1:n2]`. `n1` — это индекс начала среза, а `n2` — его окончания, причем символ с номером `n2` в срез уже не входит! Если указан отрицательный индекс, это значит, что любой индекс `-n` аналогичен записи `len(s)-n`. Если отсутствует первый индекс, то срез берётся от начала до второго индекса; при отсутствии второго индекса срез берётся от первого индекса до конца строки:

```
>>> Day = 'morning, afternoon, evening'
>>> Day[0:7]
'morning'
>>> Day[9:-9]
'afternoon'
>>> Day[-7:]
'evening'
```

Можно извлекать символы не подряд, а через определённое количество. В таком случае оператор индексирования выглядит так: `[n1:n2:n3]`; `n3` — это шаг, через который осуществляется выбор элементов:

```
>>> flag = 'Красный Голубой Белый'
>>> flag[::8]
'КГБ'
```

Обратите внимание, что в срезе строки `s` могут быть пропущены и первый, и второй индексы одновременно: вместо них подставляются `0` и `len(s)` соответственно.

- Оператор `in` позволяет узнать, принадлежит ли подстрока в строке. Оператор возвращает логическое значение: `True`, если элемент в составе строки встречается и `False`, если нет:

```
>>> S = 'Python'
>>> SubS = 'th'
>>> SubS in S
True
```

- Функции `min` и `max` применимы также и к строкам: `max(s)` определяет и выводит (возвращает) символ с наименьшим кодом — номером в кодовой таблице. Например:

```
>>> S = 'Python'
>>> min(S)
'P'
```

Возвращает символ с наибольшим значением (кодом). Например:

```
>>> S = 'Python'
>>> max(S)
'y'
```

2.2.2 Методы строк

Кроме операторов, функций и срезов значительное количество операций над строками доступно в виде *методов*. Основное различие методов и функций — синтаксическое; так, большинство методов ранее являлись функциями стандартного модуля `string`, в котором теперь остались почти только различные константы. Обратите внимание, как записываются методы объекта: `объект.метод()`, например, `S.isdigit()`. Методы — это по сути функции, у которых в качестве первого аргумента выступает сам объект, метод которого вызвается. Например, вызов метода `S.isdigit()` выдаст логическое значение: `True`, если все символы строки `S` и `False` иначе.

Таблица 2.4. Базовые операции над строками

Операция	Описание
`S1 + S2`	Объединение двух или более строк в новую строку.
`S * n`	Умножение строки на целое число `n` — многократное повторение строки.
`len(S)`	Функция, вычисляющая длину строки `S`.
`S[n]`	Доступ по индексу (номеру) к любому символу строки.
`S[n1:n2:n3]`	Срез — новая строка, являющаяся частью исходной и содержащая символы с номерами от `n1` включительно до `n2` невключительно, если `n3` присутствует (может не быть), то берутся не все символы, а с шагом `n3`.
`S2 in S1`	Логический оператор, проверяющий, является ли строка `S2` частью строки `S1`.
`min(S)`	Функция, вычисляющая символ строки `S` с наименьшим кодом.
`max(S)`	Функция, вычисляющая символ строки `S` с наибольшим кодом.

Бывают методы, как описанный выше, не требующие вовсе никаких аргументов, бывают с одни аргументом, например метод `S1.endswith(S2)` требует 1 аргумент — строку — и проверяет, заканчивается ли строка `S1` строкою `S2`. Бывают методы с двумя аргументами, например `S1.replace(S2, S3)`, который заменяет в исходной строке `S1` содержащуюся в ней подстроку `S2` новою подстрокою `S3` и выдаёт новую строку, при этом `S1` остаётся неизменною. Более полную информацию о строковых методах можно получить, введя в интерактивном режиме команду `help(str)`.

2.3 Условия и логические операции

2.3.1 Логический (булевский) тип. Операторы сравнения. Логические операторы

Логический (булевский) тип может принимать одно из двух значений `True` (истина) или `False` (ложь). В языке Python булевский тип данных обозначается как `bool`, для приведения других типов данных к булевскому существует функция `bool()`, работающая по следующим соглашениям:

- строки: пустая строка — ложь, непустая строка — истина.
- числа: нулевое число — ложь, ненулевое число (в том числе и меньшее единицы) — истина.

- функции — всегда истина.

Для работы с алгеброй логики в Python кроме логического типа данных предусмотрены операторы сравнения:

- «>» больше,
- «<» меньше,
- «==» равно (одиночное «=» зарезервировано за оператором присваивания),
- «! =» не равно,
- «>=» больше или равно,
- «<=» меньше или равно.

Вот простенькая программа, вычисляющая различные логические выражения:

```
x = 12 - 5
h1 = x == 4
h2 = x == 7
h3 = x != 7
h4 = x != 4
h5 = x > 5
h6 = x < 5
print (h1, h2, h3, h4, h5, h6)
```

Её вывод:

```
False True False True True False
```

Как видим, сравнивать особенно по равенству/неравенству можно всё, что угодно, включая типы данных. Обратите внимание, что оператор присваивания имеет самый низкий приоритет, поэтому расстановка скобок вокруг логических операторов и операторов сравнения не требуется.

Из логических переменных и выражений можно строить более сложные (составные) логические выражения с помощью логических операторов: `not` (отрицание, логическое НЕ), `or` (логическое ИЛИ) и `and` (логическое И):

- `x and y` — логическое «И» (умножение). Принимает значение `True` (истина), только когда `x = True` и `y = True`. Принимает значение `False` (ложь), если хотя бы одна из переменных равна `False`, или обе переменные `False`.
- `x or y` — логическое «ИЛИ» (сложение). Принимает значение `True` (истина), если хотя бы одна из переменных равна `True`, или обе переменные `True`. Принимает значение `False` (ложь), если `x == y == False`.

A	not A
True	False
False	True

A	B	A and B	A or B
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

Таблица 2.5. Таблицы истинности логических функций для логического типа.

- `not x` — логическое «НЕ» (отрицание). Принимает значение `True` (истина), если `x == False`. Принимает значение `False` (ложь), если `x == True`.

Правила работы логических операторов можно также задать с помощью таблиц истинности, в которых указывается истинность составного выражения, в зависимости от значений исходных простых выражений.

Следует отметить, что логические операции в Python определены для объектов *любых типов*, но результаты таких операций для операторов `and` и `or` не всегда легко понятны (оператор `not` работает достаточно просто, он приводит аргумент к логическому значению по описанным в начале этого раздела правилам и выдаёт всегда только логическое значение). Вот пример:

```
>>> [1, 2] and [1] and 13
13
>>> [1, 2] and [1] or 13
[1]
```

Здесь все три объекта: `[1, 2]`, `[1]` и `13` будут интерпретироваться как истина, поскольку списки не пустые, а число не равно нулю. Но в первом случае в результате вычисления выражения получится число, а во втором — один из списков. Поэтому мы рекомендуем программистам не использовать логические операторы в выражениях с нелогическими объектами, либо преобразовывать эти объекты к логическому типу напрямую с помощью функции `bool()`:

```
>>> bool([1, 2]) and bool([1]) or bool(13)
True
>>> bool([1, 2]) and bool([1]) and bool(13)
True
```

Логические операторы `and` и `or` в Python используются существенно реже, чем во многих других популярных языках программирования, например Java, C/C++ или Pascal/Delphi, потому что в Python можно делать любые двойное, тройные и т. д. сравнения, например, `a < x < b` эквивалентно `(x > a) and (x < b)`.

Рассмотрим пример, в котором используются логические операторы и функции.

Пример задачи 1 Трое друзей, болельщиков автогонок «Формула–1», спорили о результатах предстоящего этапа гонок.

— Вот увидишь, Шумахер не придет первым, — сказал Джон. — Первым будет Хилл.

— Да нет же, победителем будет, как всегда, Шумахер, — воскликнул Ник. — А об Алези и говорить нечего, ему не быть первым.

Питер, к которому обратился Ник, возмутился: — Хиллу не видать первого места, а вот Алези пилотирует самую мощную машину.

По завершении этапа гонок оказалось, что предположения двух друзей подтвердились, а предположения одного из трёх неверны. Кто выиграл этап гонки?

Решение задачи 1 Введем обозначения для логических высказываний: `S` — победит Шумахер; `H` — победит Хилл; `A` — победит Алези.

Реплика Ника «Алези пилотирует самую мощную машину» не содержит никакого утверждения о месте, которое займёт этот гонщик, поэтому в дальнейших рассуждениях не учитывается.

Зафиксируем высказывания каждого из друзей:

- Джон: `v1 = not S and H`;
- Ник: `v2 = S and not A`;
- Питер: `v3 = not H`.

Учитывая то, что предположения двух друзей подтвердились, а предположения одного из трёх неверны, запишем логическую функцию:

```
f = v1 and v2 and not v3 or v1 and not v2 and v3 \
    or not v1 and v2 and v3
```

В алгебре логики существует возможность доказательства утверждения методом перебора. Утверждение истинно, если при подстановке любых значений переменных оно превращается в верное тождество. Этот метод перебора не слишком трудоемок, поскольку переменные могут принимать только значения `False` и `True`.

Логическая функция от n аргументов может быть задана таблицей, в которой перечислены все возможные наборы из `False` и `True` длины `n` и для каждого из них рассчитано значение функции. Пусть эту таблицу нам автоматически составит программа на Python:

```
for S in (False, True):
    for H in (False, True):
        for A in (False, True):
            v1 = not S and H
            v2 = S and not A
            v3 = not H
```

```
f = v1 and v2 and not v3 or \
    v1 and not v2 and v3 or \
    not v1 and v2 and v3
print(S, H, A, f)
```

Здесь использован оператор цикла `for`, подробнее изложенный в следующей главе. Оператор \ позволяет перенести часть кода на следующую строчку, число начальных пробелов в которой неважно.

Вывод программы:

```
False False False False
False False True False
False True False False
False True True False
True False False True
True False True False
True True False False
True True True False
```

Из таблицы видно, что заданное утверждение истинно (`True` в четвёртом столбце) только при `S==True`, `H==False`, `A==False`. Значит ответ на задачу: победил Шумахер.

Обратите внимание на отступы, Python к ним чрезвычайно чувствителен. Дело в том, что в Python фактически нет операторных скобок типа `begin/end` (как в Pascal) или «{}» (как в Си-подобных языках), их роль выполняют отступы (роль открывающейся скобки в некотором смысле выполняет «:»). Если последующая строчка сдвинута по отношению к предыдущей вправо — значит, то, что на ней написано, представляет собою блок (часть кода, которая сгруппирована и воспринимается как единое целое). Принято и очень рекомендуется делать по 4 пробела на каждый уровень вложенности. При работе в IDLE и Geany редактор сам поставит нужный отступ, если вы не забудете «:» в конце предыдущей строки, клавиша `<Backspace>` позволит вернуться на один уровень назад.

Форматирование — очень важный момент при программировании на Python. Если вы поставите хотя бы один лишний пробел в начале строки, программа вообще не запустится, выдав `Indentation Error` — ошибку расстановки отступов, указав на первую строку, где с точки зрения интерпретатора возникла ошибка.

2.3.2 Условный оператор `if`

Поведение реальных программ должно зависеть от входных данных. Например, в рассмотренной ранее задаче о преобразовании мужской формы фамилии в женскую добавление символа «а» вовсе не единственный вариант: если мужская форма заканчивается на «ой», «ый» или «ий», нужно это окончание отбросить и добавить соответственно «ая» или «яя». Чтобы программа могла осуществить такой выбор, она должна уметь проверять условия, для чего во всех языках программирования есть *условный оператор*.

В Python простейшая форма условного оператора имеет вид[2]:

```
if <логическое выражение>:
    <действия, выполняемые, когда логическое
    выражение принимает значение True>
```

В такой форме действия после двоеточия выполняются, если логическое выражение истинно. Если же оно ложно, программа ничего не делает и переходит к оператору, следующему за `if`. Когда нужно выполнить различные действия, если условие истинно и если оно ложно, используется следующая более полная форма:

```
if <логическое выражение>:
    <действия, выполняемые, когда логическое
    выражение принимает значение True>
else:
    <действия, выполняемые, когда логическое
    выражение принимает значение False>
```

Наконец, если нужно последовательно проверить несколько условий, используется форма с дополнительным оператором `elif` (сокращение от `else if`):

```
if <логическое выражение>:
    <действия, выполняемые, если логическое
    выражение принимает значение True>
elif <второе логическое выражение>:
    <действия, выполняемые, если второе логическое
    выражение принимает значение True>
elif <третье логическое выражение>:
    <действия, выполняемые, если третье логическое
    выражение принимает значение True>
...
else:
    <действия, выполняемые, если ни одно из
    логических выражений не принимает значение True>
```

Дополнительных условий и связанных с ними блоков `elif` может быть сколько угодно, но важно отметить, что в такой сложной конструкции будет выполнен всегда только один блок кода. Другими словами, как только некоторое условие оказалось истинным, соответствующий блок кода выполняется, и дальнейшие условия не проверяются.

Обратите внимание, что после двоеточия в конструкциях типа `if`, `else`, `elif` всегда идёт блок, выделенный отступом вправо. Большинство редакторов кода, в том числе и IDLE, делают этот отступ автоматически. То, что выделено отступами, и есть *тело* оператора, а то, что до двоеточия, называется *заголовком*.

[2]Вообще говоря, выражение может быть вовсе не логическим, в этом случае оно будет приведено к логическому типу по ранее приведённым правилам.

Приведём простой пример. Следующая простая программа проверяет, делится ли первое введённое число на второе нацело:

```
a = int(input('Введите первое число: '))
b = int(input('Введите второе число: '))
if a % b == 0:
    print("Yes")
else:
    print("No")
```

2.4 Списки

Любой язык программирования обязан поддерживать составные типы данных, где одна переменная может содержать как контейнер несколько — в лучшем случае произвольно много — единиц информации. Для этой цели в Python существуют несколько типов данных, самым базовым из которых является список.

Списки в языке программирования Python, как и строки, являются упорядоченными последовательностями значений. Однако, в отличие от строк, списки состоят не из символов, а из различных объектов (значений, данных), и заключаются не в кавычки, а в квадратные скобки `[ ]`. Объекты отделяются друг от друга с помощью запятой.

Списки могут состоять из различных объектов: чисел, строк и даже других списков. В последнем случае, списки называют вложенными. Вот некоторые примеры списков:

```
[159, 152, 140, 128, 113]           #список целых чисел
[15.9, 15.2, 14.0, 128., 11.3]      #список вещественных чисел
['Даша', 'Катя', 'Ксюша']           #список строк
['Саратов', 'Астраханская', 104, 18]  #смешанный список
[[1, 0, 0], [0, 1, 0], [0, 0, 1]]   #список списков
```

Как и над строками, над списками можно выполнять операции соединения и повторения:

```
>>> [6, 'октябрь', 2015]+[16, 'декабрь', 2015]
[6, 'октябрь', 2015, 16, 'декабрь', 2015]
>>> [2, 3, 4]*2
[2, 3, 4, 2, 3, 4]
```

По аналогии с символами (элементами) строки можно получать доступ к элементам списка по их индексам, складывать их, извлекать срезы, измерять длину списка, узнавать тип данных:

```
>>> list1 = ['P', 'y', 'th', 'o', 'n', 3.4]
>>> len(list1)
6
```

```
>>> list1[0] + list1[1]
'Py'
>>> list1[0]
'P'
>>> list1[0:5]
['P', 'y', 'th', 'o', 'n']
>>> list1[5:]
[3.4]
>>> type (list1)
<class 'list'>
```

Обратите внимание, что нумерация элементов всегда начинается с нуля, поэтому нулевой элемент это 'P'.

В отличие от строк, списки — это изменяемые последовательности. Если представить строку как объект в памяти, то когда над ней выполняются операции конкатенации и повторения, то эта строка не меняется, а в результате операции создаётся другая строка в другом месте памяти. В строку нельзя добавить новый символ или удалить существующий, не создав при этом новой строки. Со списком дело обстоит иначе. При выполнении операций новые списки могут не создаваться, а будет изменяться непосредственно оригинал. Из списков можно удалять элементы, добавлять новые. При этом следует помнить, многое зависит от того, как вы распоряжаетесь переменными.

Символ в строке изменить нельзя, элемент списка — можно:

```
>>> mystr = 'Python'
>>> mylist = ['P', 'y', 't', 'h', 'o', 'n']
>>> mystr[1] = 'i'
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    mystr[1] = 'i'
TypeError: 'str' object does not support item assignment
>>> mylist[1] = 'i'
>>> mylist
['P', 'i', 't', 'h', 'o', 'n']
```

В списке можно заменить целый срез:

```
>>> mylist[:3] = ['Y', 'e', 's']
>>> mylist
['Y', 'e', 's', 'h', 'o', 'n']
```

Для списка можно создавать его копию:

```
>>> list1 = ['P','y','t','h','o','n']
>>> list2 = list1.copy() #Создание копии списка
>>> list2[1] = 'i'
>>> list2, list1
(['P', 'i', 't', 'h', 'o', 'n'], ['P', 'y', 't', 'h', 'o', 'n'])
```

Список list2 изменился, а список list1 — нет.

Для списка можно создать вторую ссылку на список. Внимание! При создании второй ссылки данные не копируются, просто эти данные теперь имеют два имени, поэтому изменение list1 будет приводить к изменению list2:

```
>>> list2 = list1 #Создание второй ссылки, а не копии
>>> list2[1] = 'i'
>>> list2, list1
(['P','i','t','h','o','n'], ['P','i','t','h','o','n'])
```

Изменились оба списка. Для создания копии предусмотрен более простой синтаксис, нежели использование стандартного метода copy: достаточно взять срез списка от начала и до конца: list3 = list1[:] эквивалентно тому, что мы написали бы list3 = list1.copy().

Таблица 2.6. Методы списка

Метод	Описание

L.append(x)	Добавление элемента со значением x в конец списка L:
	`>>> L = ['P', 'y', 't', 'h', 'o', 'n']`
	`>>> L.append('3')`
	`>>> L`
	`['P', 'y', 't', 'h', 'o', 'n', '3']`

L.extend(T)	Добавление списка или кортежа T в конец списка L. Похоже на объединение списков, но создание нового списка не происходит:
	`>>> L = ['P', 'y', 't', 'h', 'o', 'n']`
	`>>> T = ['3', '.', '4']`
	`>>> L.extend(T)`
	`>>> L`
	`['P', 'y', 't', 'h', 'o', 'n', '3', '.', '4']`
	`>>> L = ['P', 'y', 't', 'h', 'o', 'n']`
	`>>> L.append(T)`
	`>>> L`
	`['P', 'y', 't', 'h', 'o', 'n', ['3', '.', '4']]`

L.insert(i,x)	Вставка элемента со значением x на позицию i в списке L:
	`>>> L = ['P', 'y', 't', 'h', 'o', 'n']`
	`>>> L.insert(3, '!')`
	`>>> L`
	`['P', 'y', 't', '!', 'h', 'o', 'n']`

`L.pop(i)`	Извлечение элемента с номером `i` из списка `L`, элемент удаляется и выдаётся в качестве результата:
	`>>> L = ['P', 'y', 't', 'h', 'o', 'n']`
	`>>> x = L.pop(0)`
	`>>> L`
	`['y', 't', 'h', 'o', 'n']`
	`>>> x`
	`'P'`
	Если использовать `L.pop()` без аргумента, то будет извлекаться последний элемент.

`L.remove(x)`	Удаление элемента со значением `x` из списка `L`:
	`>>> L = ['P', 'y', 't', '!', 'h', 'o', 'n']`
	`>>> L.remove('!')`
	`>>> L`
	`['P', 'y', 't', 'h', 'o', 'n']`
	Если в списке содержится несколько одинаковых элементов, удаляется тот, который имеет наименьший номер.

`L.count(x)`	Определение количества элементов, равных `x`, в списке `L`:
	`>>> L = [8, 1, 5, -7, 4, 9, -2, 6, 2, 5]`
	`>>> L.count(5)`
	`2`

`L.index(x)`	Определение первой слева позиции элемента со значением `x` в списке `L`:
	`>>> L = [8, 1, 5, -7, 4, 9, -2, 6, 2, 5]`
	`>>> L.index(5)`
	`2`

`L.reverse()`	Переворачивание списка наоборот:
	`>>> L = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`
	`>>> L.reverse()`
	`>>> L`
	`[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]`

L.sort()	Сортировка списка по возрастанию (в алфавитном порядке):
	`>>> L = [10, 5, 2, 8, 1, 12]`
	`>>> L.sort()`
	`>>> L`
	`[1, 2, 5, 8, 10, 12]`

К спискам применимы некоторые стандартные функции, например, знакомая нам `len`, к которой обращаться нужно следующим образом: `len(mylist)`. Функция `sum` подсчитывает сумму элементов списка, если все они числового типа. Функция `range` позволяет сформировать диапазон значений целых чисел. В самом общем случае `range` принимает 3 аргумента: начало диапазона, конец (всегда берётся не включительно) и шаг. Обратите внимание, что в Python 3.x эта функция не выдаёт список, а производит специальный объект-диапазон. Поэтому, чтобы получить список чисел, нужно обязательно явно преобразовать результат с помощью функции `list`. Есть ещё функция `sorted()`, которая возвращает новый список, отсортированный по убыванию. Функции `min` и `max` находят максимальный и минимальный элементы списка. Вот небольшая программа с использованием этих функций:

```
>>> A = list(range(0, 10, 1))
>>> A
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> sum(A)
45
>>> min(A)
0
>>> max(A)
9
```

В данном примере объект `A` формируется с помощью итератора `range`, а потом явно преобразуется к типу `list`.

К спискам, как и к строкам, применим оператор `in`, позволяющий узнать, принадлежит ли элемент списку. Напомним, что оператор возвращает логическое значение: `True`, если элемент в списке содержится и `False`, если нет. Вот программа, использующая этот оператор:

```
>>> mylist1 = ['P','y','t','h','o','n']
>>> mylist2 = [6, 10, 2015]
>>> 'y' in mylist1
True
>>> 30 in mylist2
False
>>> mylist2.append(['Программирование', 11.30])
```

```
>>> mylist2
[6, 10, 2015, ['Программирование', 11.3]]
>>> mylist2[-1]
['Программирование', 11.3]
```

Во второй список специально был добавлен ещё один список, чтобы показать, что списки могут быть вложенными. Также в последней приведённой программе была использована возможность индексировать списки с конца: минус первый элемент списка — это его последний элемент. Таким образом, `mylist2[-1]` — это обращение к последнему (первому с конца) элементу списка, являющемуся тоже списком.

2.5 Кортежи

Список так же может быть неизменяемым, как и строка, в этом случае он называется кортеж (`tuple`). Кортеж использует меньше памяти, чем список. При задании кортежа вместо квадратных скобок используются круглые (хотя можно и совсем без скобок). Кортеж не допускает изменений, в него нельзя добавить новый элемент, удалить или заменить существующие элементы, но он может содержать изменяемые объекты, например, списки:

```
>>> ll = []
>>> A = (1, 2, 3, ll)
>>> A
(1, 2, 3, [])
>>> A[1] = 4
Traceback (most recent call last):
  File "<pyshell#22>", line 1, in <module>
    A[1] = 4
TypeError: 'tuple' object does not support item assignment
>>> A[3].append(3)
>>> print(A, ll)
(1, 2, 3, [3]) [3]
```

Видно, что прямая замена элемента кортежа недопустима — вызывает `TypeError`, так как тип `tuple` не поддерживает изменеине элементов, но если использовать втроенный метод `append` у списка, являющегося элементом кортежа, этот список можно изменить.

Функция `tuple()` берет в качестве аргумента строку или список и превращает его в кортеж, а функция `list()` переводит кортеж в список:

```
>>> B = list(A)
>>> B
[1, 2, 3]
>>> C = tuple(B)
>>> C
```

```
(1, 2, 3)
```

Основное различие между кортежами и списками состоит в том, что кортежи не могут быть изменены. На практике это означает, что у них нет методов, которые бы позволили их изменить: `append()`, `extend()`, `insert()`, `remove()`, `pop()`. Но можно взять срез от кортежа, так как при этом создастся новый кортеж.

Кортежи в некоторых случаях быстрее, чем списки. Но такие оптимизации в каждом конкретном случае требуют дополнительных исследований. Кортежи делают код безопаснее в том случае, если у вас есть «защищенные от записи» данные, которые не должны изменяться. Некоторые кортежи могут использоваться в качестве элементов множества и ключей словаря (конкретно, кортежи, содержащие неизменяемые значения, например, строки, числа и другие кортежи). Словари будут рассмотрены в следующем разделе. Списки никогда не могут использоваться в качестве ключей словаря, потому что списки — изменяемые объекты.

В Python можно использовать кортежи, чтобы присваивать значение нескольким переменным сразу:

```
>>> v = ('f', 5, True)
>>> (x, y, z) = v
>>> x
'f'
>>> y
5
>>> z
True
>>>
```

Это не единственный способ использования. Предположим, что вы хотите присвоить имена диапазону значений. Вы можете использовать встроенную функцию `range()` для быстрого присвоения сразу нескольких последовательных значений.

```
>>> (MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
 FRIDAY, SATURDAY, SUNDAY) = range(1, 8)
>>> MONDAY
1
>>> SUNDAY
7
>>>
```

Заметим, что при вводе длинных списков, кортежей и словарей как в интерактивном, так и в скриптовом режиме можно перейти на следующую строчку после любой запятой, разделяющей элементы. Это позволяет в большинстве случаев избежать использования символа переноса строки «\».

2.6 Словари

Одним из сложных типов данных наряду со строками и списками в языке программирования Python являются словари. Словарь — это изменяемый (как список) неупорядоченный (в отличие от строк и списков) набор пар `'ключ:значение'`. Словари оказываются очень удобными объектами для хранения данных и, по сути, являются своеобразной заменой базе данных.

Чтобы представление о словаре стало более понятным, можно провести аналогию с обычным словарём, например, англо-русским. На каждое английское слово в таком словаре есть русское слово перевод: cat — кошка, dog — собака, bird — птица и т.д. Если англо-русский словарь описывать с помощью Python, то английские слова будут ключами, а русские — их значениями:

```
>>> animal = {'cat':'кошка', 'dog':'пёс', 'bird':'птица',
'mouse':'мышь'}
>>> animal
{'mouse': 'мышь', 'cat': 'кошка', 'dog': 'пёс', 'bird': 'птица'}
>>> type(animal)
<class 'dict'>
```

Обратите внимание на фигурные скобки, именно с их помощью определяется словарь. Такой тип данных в Python называется `dict`. Если создать словарь в интерпретаторе Python (как и было сделано), то после нажатия `<Enter>` можно наблюдать, что последовательность вывода пар `'ключ:значение'` не совпадёт с тем, как было введено. Дело в том, что в словаре абсолютно не важен порядок пар, и интерпретатор выводит их в случайном порядке. Тогда как же получить доступ к определённому элементу, если индексация невозможна в принципе? В словаре доступ к значениям осуществляется по ключам, которые заключаются в квадратные скобки (по аналогии с индексами строк и списков):

```
>>> animal={'cat':'кошка','dog':'пёс','bird':'птица','mouse':'мышь'}
>>> animal['cat']
'кошка'
```

Словари, как и списки, являются изменяемым типом данных: можно изменять, добавлять и удалять элементы — пары `'ключ:значение'`. Изначально словарь можно создать пустым, например, `dic = {}` и лишь потом заполнить его элементами.

Добавление и изменение имеет одинаковый синтаксис: `словарь[ключ] = значение`. Ключ может быть, как уже существующим (тогда происходит изменение значения), так и новым (происходит добавление элемента словаря). Удаление элемента словаря осуществляется с помощью функции `del(dic[key])` или метода `pop(key)`:

```
>>> dic = {'cat':'кошка','dog':'пёс','bird':'птица','mouse':'мышь'}
>>> dic['cat'] = 'кот'
>>> dic
{'mouse': 'мышь', 'cat': 'кот', 'dog': 'пёс', 'bird': 'птица'}
```

```
>>> dic['fox'] = 'лиса'
>>> dic
{'fox': 'лиса', 'mouse': 'мышь', 'cat': 'кот', 'dog': 'пёс',
'bird': 'птица'}
>>> del(dic['mouse'])
>>> dic
{'fox': 'лиса', 'cat': 'кот', 'dog': 'пёс', 'bird': 'птица'}
>>> dic.pop('fox')
'лиса'
>>> dic
{'bird': 'птица', 'cat': 'кот', 'dog': 'пёс'}
```

Тип данных ключей и значений словарей не обязательно должен быть строковым:

```
>>> DicProg = {1:'Pascal', 2:'Python', 3:'C', 4:'Java'}
```

Словари — это широко используемый тип данных языка Python. Для работы с ними существует ряд встроенных методов и функций. Метод `keys()` для словаря возвращает последовательность всех используемых ключей в произвольном порядке. Для определения наличия определенного ключа раньше был метод `has_key()`, но в версии 3.0 вместо него есть знакомый нам оператор `in`:

```
>>> DicProg.keys()
dict_keys([1, 2, 3, 4])
>>> 1 in DicProg
True
>>> 'Pascal' in DicProg
False
```

2.7 Примеры решения задач

Пример задачи 2 (Арифметические операции) *Напишите программу (необходимые данные вводятся с клавиатуры) для вычисления всех трёх сторон прямоугольного треугольника, если даны один из острых углов и площадь.*

Решение задачи 2 Обозначим катеты прямоугольного треугольника a и b, а гипотенузу — c. Площадь треугольника обозначим S, один из острых углов — α. Воспользуемся формулой площади прямоугольного треугольника $S = \frac{ab}{2}$ и формулой тангенса $\operatorname{tg}\alpha = \frac{a}{c}$. Отсюда можно получить выражение для одного из катетов: $a = \sqrt{\frac{2S}{\operatorname{tg}\alpha}}$. Теперь легко вычислить оставшийся катет и гипотенузу: $b = \frac{2S}{a}$ и $c = \sqrt{a^2 + b^2}$.

```
from math import *
S = int(input('Площадь трегольника = '))
alpha = int(input('Острый угол (в градусах) = '))
```

```
a = sqrt(2*S/tan(radians(alpha)))
b = 2*S/a
c = (a**2+b**2)**(1/2)
print (a, b, c)
```

Вывод программы:

```
Площадь трегольника = 8
Острый угол (в градусах) = 45
4.0 4.0 5.656854249492381
```

Пример задачи 3 (Строки) *Свяжите любую переменную со строкой: «У Лукоморья дуб зелёный...». Выведите все символы этой строки в обратном порядке.*

Решение задачи 3 Листинг программы:

```
S = 'У Лукоморья дуб зелёный...'
print(S[-1::-1])
```

Вывод программы:

```
...йынёлез буд яьромокуЛ У
```

Пример задачи 4 (Простое условие) *Ответить на вопрос, истинно ли условие:* $x^3 + y^3 \leqslant 9$. *Значения переменных* x *и* y *вводятся с клавиатуры.*

Решение задачи 4 Листинг программы:

```
x = float(input('x␣=␣'))
y = float(input('y␣=␣'))
print (x**3 + y**3 <= 9)
```

Вывод программы:

```
x = 1
y = 3
False
```

Пример задачи 5 (Сложное условие) *Записать условие (составить логическое выражение), которое является истинным, когда число* X *чётное и меньше* 100.

Решение задачи 5 Листинг программы:

```
X = float(input('x␣=␣'))
print ((X % 2 == 0) and (X < 100))
```

Вывод программы:

```
>>>
x = 50
True
>>> ================ RESTART ==================
>>>
x = 3
False
>>> ================ RESTART ==================
>>>
x = 102
False
```

Пример задачи 6 (Условный оператор) Приведём пример множественного ветвления с использованием `elif`, где разберём перебор вариантов. Задача такая: пользователь вводит количество денег в рублях, в магазине можно купить хлеб за 20 руб. и сыр за 100 руб. Если хватает на то и другое, покупаем всё, если только на сыр или только на хлеб, берём что-то одно, если не хватает ни на что — уходим.

Решение задачи 6 Листинг программы:

```
a = int(input("Input␣amount␣of␣money:␣"))
if a >= 120:
    print ("Bread␣and␣cheese")
elif a >= 100:
    print ("Cheese␣only")
elif a >= 20:
    print ("Bread␣only")
else:
    print ("Nothing␣:(")
```

Как видим, проверять все условия в каждом случае, например, для хлеба условие, что денег меньше 100, нет смысла: если первое условие выполняется, то проверка прочих никогда не происходит, иначе управление передаётся на следующий `elif` и так далее, если не выполнилось ни одно из условий, выполняются операторы в блоке `else`, если таковой присутствует.

Пример задачи 7 (Списки) Создайте список в диапазоне (0, 100) с шагом 1. Свяжите его с переменной. Извлеките из него срез с 20 по 30 элемент включительно.

Решение задачи 7 Листинг программы:

```
A = list((0, 100, 1))
print(A[20:31])
```

Вывод программы:

```
[20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30]
```

Пример задачи 8 (Кортежи) Создайте кортеж в диапазоне (0, 20) с шагом 1. Свяжите его с переменной. Выведите эту переменную на экран.

Решение задачи 8 Листинг программы:

```
A = tuple(range(0, 20, 1))
print(A)
```

Вывод программы:

```
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19)
```

Пример задачи 9 (Словари) Создайте словарь, который будет содержать значения параметров функции $y = A\cos(\omega t + f)$. А затем по ключу запросите значения каждого из них.

Решение задачи 9 Листинг программы

```
print ('y␣=␣A␣cos(wt+f)')
Parameters = {'A':10, 'w':1, 'f':0}
Key = str(input('Какой параметр? '))
print(Parameters[Key])
```

Вывод программы:

```
y = A cos(wt+f)
Какой параметр? A
10
```

2.8 Задания

Задание 2 Выполнять три задания в зависимости от номера в списке. Чтобы узнать номера ваших заданий, необходимо решить задачку: требуется сделать задания № m, № $m+5$, № $m+10$, где $m = (n-1)\%5+1$, n — порядковый номер студента в списке группы по алфавиту.

Используя арифметические операторы ($+$, $-$, $*$, $/$, $//$, $\%$), напишите программу (необходимая информация запрашивается у пользователя с клавиатуры).

1. Составьте арифметическое выражение и вычислите n-е чётное число (первым считается 2, вторым 4 и т.д.).

2. Составьте арифметическое выражение и вычислите n-е нечётное число (первое — 1, второе — 3 и т.д.).

3. Сколько человек находится между i-м и k-м в очереди?

4. Сколько нечётных чисел на отрезке $[a; b]$, если a и b — чётные? a и b — нечётные? a — чётное, a — нечётное?

5. Сколько полных часов, минут и секунд содержится в x секундах? Разложите имеющееся количество секунд на сумму из x часов + y минут + z секунд.

6. В доме 9 этажей, на каждом этаже одного подъезда по 4 квартиры. В каком подъезде, и на каком этаже находится n-я квартира?

7. Старинными русскими денежными единицами являются: 1 рубль = 100 копеек, 1 гривна = 10 копеек, 1 алтын = 3 копейки, 1 полушка = 0,25 копейки. Имеется A копеек. Разложите имеющуюся сумму в копейках на сумму из x рублей + y гривен + z алтынов + v полушек.

8. Стрелка прибора вращается с постоянной скоростью, совершая w оборотов в секунду (не обязательно стрелка прибора, может быть это волчок в игре «Что? Где? Когда?» и т.п.) Угол поворота стрелки в нулевой момент времени примем за 0. Каков будет угол поворота через t секунд?

9. Вы стоите на краю дороги и от вас до ближайшего фонарного столба x метров. Расстояние между столбами y метров. На каком расстоянии от вас находится n-й столб?

10. Та же ситуация, что и в предыдущей задаче. Длина вашего шага z метров. Мимо скольких столбов вы пройдете, сделав n шагов?

11. x — вещественное число. Запишите выражение, позволяющее выделить его дробную часть.

12. x — вещественное число. Запишите выражение, которое округлит его до сотых долей (останется только два знака после запятой).

13. От бревна длиной L отпиливают куски длиной x. Сколько целых полноразмерных кусков максимально удастся отпилить?

14. Бревно длиной L распилили в n местах. Какова средняя длина получившихся кусков?

15. Резиновое кольцо диаметром d разрезали в n местах. Какова средняя длина получившихся кусков?

Задание 3 (Строки) Задания выполняйте все по порядку. Свяжите любую переменную со строкой: «Мы обязательно научимся программировать!». Извлеките из неё следующие срезы:

1. выведите третий символ этой строки;
2. выведите предпоследний символ этой строки;
3. выведите первые пять символов этой строки;
4. выведите всю строку, кроме последних двух символов;
5. выведите все символы с чётными индексами (считая, что индексация начинается с 0);
6. выведите все символы с нечётными индексами, то есть, начиная с первого символа строки;
7. выведите четыре символа из центра строки;
8. выведите символы с индексами, кратными трём;
9. выведите все символы в обратном порядке;
10. выведите все символы строки через один в обратном порядке, начиная с последнего;
11. удалите второе слово из строки;
12. замените второе слово на строку «никогда не»;
13. добавьте в конец строки «на Python»;
14. поставьте последнее слово первым в строке;
15. выведите длину данной строки.

Задание 4 (Логический тип данных. Логические операторы) В каждой группе выполнять по одному заданию в зависимости от номера в списке группы: $(n-1)\%10+1$, где n — номер в списке.

Вычислить значение логического выражения. Значения переменных x и y вбиваются с клавиатуры.

1. $x^2 + x^2 \leqslant 4$;
2. $x^2 - x^2 \leqslant 4$;
3. $x \geqslant 0$ или $y^2 \neq 4$;
4. $x \geqslant 0$ и $y^2 \neq 4$;

5. $x \cdot y \neq 0$ или $y > x$;
6. $x \cdot y \neq 0$ и $y > x$;
7. не $x \cdot y < 0$ или $y > x$;
8. не $x \cdot y < 0$ и $y > x$;
9. $x \geqslant 4$ или $y^2 \neq 4$;
10. $x \geqslant 4$ и $y^2 \neq 4$.

Вычислить значение логического выражения при всех возможных значениях логических величин X, Y и Z (для образца можно взять задачку про Шумахера):

1. не (X или не Y и Z);
2. Y или (X и не Y или Z);
3. не (не X и Y или Z);
4. не (X или не Y и Z) или Z;
5. не (X и не Y или Z) и Y;
6. не (не X или Y и Z) или X;
7. не (Y или не X и Z) или Z;
8. X и не (не Y или Z) или Y;
9. не (X или Y и Z) или не X;
10. не (X и Y) и (не X или не Z).

Записать условие (составить логическое выражение), которое является истинным, когда:

1. число X делится нацело на 13 и меньше 100;
2. число X больше 10 и меньше 20;
3. каждое из чисел X и Y больше 25;
4. каждое из чисел X и Y нечетное;
5. только одно из чисел X и Y четное;
6. хотя бы одно из чисел X и Y положительно;
7. каждое из чисел X, Y, Z кратно пяти;
8. только одно из чисел X, Y, Z кратно трем;

9. только одно из чисел X, Y, Z меньше 10;

10. хотя бы одно из чисел X, Y, Z отрицательно.

Задание 5 (Условный оператор) *Выполнять три задания в зависимости от номера в списке. Необходимо сделать задания № m, № $m+5$, № $m+10$, где $m = (n-1)\%5+1$, n — номер студента в списке группы в алфавитном порядке.*

1. Напишите программу, которая запрашивает значение x, а затем выводит значение следующей функции от x (она называется по латыни «signum», что значит «знак»):
$$y(x) = \begin{cases} 1, & x > 0, \\ 0, & x = 0, \\ -1, & x < 0 \end{cases}$$

2. Напишите программу, которая запрашивает значение x, а затем выводит значение следующей функции от x:
$$y(x) = \begin{cases} \sin^2(x), & x > 0, \\ 0, & x = 0, \\ 1 + 2\sin(x^2), & x < 0 \end{cases}$$

3. Напишите программу, которая запрашивает значение x, а затем выводит значение следующей функции от x:
$$y(x) = \begin{cases} \cos^2(x), & x > 0, \\ 0, & x = 0, \\ 1 - 2\sin(x^2), & x < 0 \end{cases}$$

4. Запросите у пользователя два числа. Далее:
   - если первое больше второго, то вычислить их разницу и вывести данные на печать;
   - если второе число больше первого, то вычислить их сумму и вывести на печать;
   - если оба числа равны, то вывести это значение на печать.

5. Запросите у пользователя два целых числа m и n. Если целое число m делится нацело на целое число n, то вывести на экран частное от деления, в противном случае вывести сообщение «m на n нацело не делится».

6. Напишите программу для решения квадратного уравнения $ax^2 + bx + c = 0$. Значения коэффициентов a, b, c вводятся с клавиатуры. Вычисление квадратного корня можно организовать либо путём возведения в степень

0.5, либо с помощью функции `sqrt` из математического модуля. Проверяйте значение дискриминанта: если оно меньше нуля, корней нет, если равно нулю, значит, корень 1, если больше нуля — корней два. Для этого можно использовать конструкцию вида `if elif else`.

7. Напишите программу, решающую кубическое уравнение вида $y^3+px+q=0$ с помощью формулы Кардано. Значения коэффициентов p и q вводятся с клавиатуры. Найдите корни уравнения. Помните, что Python может работать с комплексными числами, но модуль math использовать для их возведения в степень нельзя. Будьте внимательны с кубическим корнем: кубический корень от отрицательного числа превращается в комплексное число.

8. Напишите программу, которая запрашивает у пользователя его возраст (целое число лет) и в зависимости от значения введённого числа выводит:

   - от 0 до 7 — «Вам в детский сад»;
   - от 7 до 18 — «Вам в школу»;
   - от 18 до 25 — «Вам в профессиональное учебное заведение»;
   - от 25 до 60 — «Вам на работу»;
   - от 60 до 120 — «Вам предоставляется выбор»;
   - меньше 0 и больше 120 — пять раз подряд: «Ошибка! Это программа для людей!».

9. Напишите программу, которая поможет вам оптимизировать путешествие на автомобиле. Пусть программа запрашивает у пользователя следующие данные:

   - Сколько километров хотите проехать на автомобиле?
   - Сколько литров топлива расходует автомобиль на 100 километров?
   - Сколько литров топлива в вашем баке?

   Далее в зависимости от введённых значений программа должна выдать вердикт: проедете вы желаемое расстояние или нет;

10. Пользователь вводит три действительных числа: длины сторон треугольника. Программа должна сообщить пользователю:

    - является ли треугольник равносторонним;
    - является ли треугольник равнобедренным;
    - является ли треугольник разносторонним;
    - является ли треугольник прямоугольным;
    - существует ли вообще такой треугольник (такого треугольника не может быть, если длина хотя бы одной стороны больше или равна сумме длин двух других).

11. Известен вес боксёра-любителя. Он таков, что боксёр может быть отнесен к одной из трех весовых категорий:

    - легкий вес — до 60 кг;
    - первый полусредний вес — до 64 кг;
    - полусредний вес — до 69 кг;

    Определить, в какой категории будет выступать данный боксер.

12. В чемпионате по футболу команде за выигрыш дается 3 очка, за проигрыш — 0, за ничью — 1. Известно количество очков, полученных командой за игру. Определить словесный результат игры (выигрыш, проигрыш или ничья).

13. Составить программу, которая в зависимости от порядкового номера дня недели (от 1 до 7) выводит на экран его название (понедельник, вторник, ..., воскресенье).

14. Составить программу, которая в зависимости от порядкового номера месяца (1, 2, ..., 12) выводит на экран его название (январь, февраль, ..., декабрь).

15. Составить программу, которая в зависимости от порядкового номера месяца (1, 2, ..., 12) выводит на экран время года, к которому относится этот месяц.

Задание 6 (Списки. Кортежи. Словари) *Задания выполнять все по порядку.*

1. *Списки*

    a) Создайте два списка в диапазоне (0, 100) с шагом 10. Присвойте некоторым переменным значения этих списков.

    b) Извлеките из первого списка второй элемент.

    c) Измените во втором списке последний объект на число «200». Выведите список на экран.

    d) Соедините оба списка в один, присвоив результат новой переменной. Выведите получившийся список на экран.

    e) Возьмите срез из соединённого списка так, чтобы туда попали некоторые части обоих первых списков. Срез свяжите с очередной новой переменной. Выведите значение этой переменной.

    f) Добавьте в список-срез два новых элемента и снова выведите его.

    g) С помощью функций `min()` и `max()` найдите и выведите элементы объединённого списка с максимальным и минимальным значением.

2. *Кортежи*

   a) Создайте два кортежа: один из чисел в диапазоне (1, количество учеников в группе) с шагом 1, второй — из фамилий учеников вашей группы. Пусть они соответствуют друг другу;

   b) Посмотрите, какая фамилия у студента с номером 5.

   c) А теперь посмотрите, что записано во второй кортеж под номером 5.

   d) Объедините два кортежа в один, присвоив результат новой переменной. Выведите получившийся список на экран.

   e) Возьмите срез из соединенного кортежа так, чтобы туда попали некоторые части обоих первых кортежей. Срез свяжите с очередной новой переменной. Выведите значение этой переменной.

3. *Словари*

   a) Создайте словарь, связав его с переменной `School`, и наполните его данными, которые бы отражали количество учащихся в пяти разных классах (например, 1а, 1б, 2в и т. д.); выведите содержимое словаря на экран.

   b) Узнайте сколько человек в каком-нибудь классе. Класс запрашивается у пользователя с клавиатуры, если такого запрашиваемого класса в школе нет, то выдаётся сообщение: «Такого класса на существует».

   c) В школе произошли изменения, внесите их в словарь: в трёх классах изменилось количество учащихся; результат выведите на экран.

   d) В школе появилось два новых класса, новый словарь выведите на экран.

   e) В школе расформировали один из классов, выведите содержимое нового словаря на экран.

Глава 3

Циклы

Циклы — это инструкции, выполняющие одну и ту же последовательность действий многократно.

В реальной жизни мы довольно часто сталкиваемся с циклами. Например, ходьба человека — вполне циклическое явление: шаг левой, шаг правой, снова левой-правой и т. д., пока не будет достигнута определенная цель (например, университет или кафе). В компьютерных программах наряду с инструкциями ветвления (т.е. выбором пути действия, конструкция `if-else`) также существуют инструкции циклов (повторения действия). Если бы инструкций цикла не существовало, пришлось вставлять в программу один и тот же код подряд столько раз, сколько нужно выполнить одинаковую последовательность действий.

3.1 Цикл с условием (`while`)

Универсальным организатором цикла в языке программирования Python (как и во многих других языках) является цикл с условием (конструкция `while`). Слово «while» с английского языка переводится как «пока» (пока логическое выражение возвращает истину, выполнять определенные операции). Конструкция `while` на языке Python может выглядеть следующим образом[1]:

```
a = начальное значение
while a оператор сравнения b:
    действия
    изменение a
    действия
```

Эта схема сильно неполная, так как логическое выражение в заголовке цикла может быть более сложным, а изменяться может переменная (или выражение) `b`.

[1] В действительности, такое представление является частным и вместо `a оператор сравнения b` может стоять любое логическое выражение и даже нелогическое выражение, которое может быть интерпретировано как логическое путём неявных преобразований типов.

Может возникнуть вопрос: «Зачем изменять a или b?». Когда выполнение программного кода доходит до цикла while, выполняется логическое выражение в заголовке, и, если было получено True, выполняются вложенные выражения. После поток выполнения программы снова возвращается в заголовок цикла while, и снова проверяется условие. Внимание! Если условие никогда не будет ложным, то не будет причин для остановки цикла, и программа зациклится. Простейший способ создать такую ситуацию:

```
while True:
    print('У попа была собака, он её любил.')
    print('Она съела кусок мяса - он её убил.')
    print('Вырыл ямку, закопал и на камне написал:')
```

Чтобы такого не произошло в обычной программе, необходимо предусмотреть возможность выхода из цикла — ложность выражения в заголовке. Таким образом, изменяя значение переменной в теле цикла, можно довести логическое выражение до ложности. Эту изменяемую переменную, которая используется в заголовке цикла while, обычно называют счётчиком. Как и всякой переменной, ей можно давать произвольные имена, однако очень часто используются буквы i и j.

Пример использования цикла while: вывод первых n чисел Фибоначчи. Ряд Фибоначчи — ряд чисел, в котором каждое последующее число равно сумме двух предыдущих: 0, 1, 1, 2, 3, 5, 8, 13 и т. д. Выведем первые 10 чисел:

```
fib1 = 0
fib2 = 1
print(fib1)
print(fib2)
n = 10
i = 2
summa = 0
while i <= n:
    summa = fib1 + fib2
    print(summa)
    fib1 = fib2
    fib2 = summa
    i = i + 1
```

Как работает эта программа? Вводятся две переменные (fib1 и fib2), которым присваиваются начальные значения. Присваиваются начальные значения переменным n и summa, а также счетчику i. Внутри цикла переменной summa присваивается сумма двух предыдущих членов ряда, и ее же значение выводится на экран. Далее изменяются значения fib1 и fib2 (первому присваивается второе, второму — сумма), а также увеличивается значение счетчика.

Задачка из жизни. В багажник автомобиля грузят овощи и фрукты с дачи: картофель, капусту, морковь, яблоки, груши и др. Объем багажника равен

350 л. Продукты кладут последовательно, объём каждого груза известен в литрах. Нужно сказать в какой момент (назвать номер груза) багажник переполнится. Программа выглядит следующим образом:

```
s = 0
n = 0
while s < 350:
    x = int(input())
    s = s + x
    n = n + 1
print(n)
```

Здесь переменная `s` хранит суммарный объём уже накопленных грузов, в переменную `x` считывается объём очередного груза, а `n` считает номер груза.

В обоих примерах был применен важный *приём накопления суммы*. Данный алгоритмический приём используется, когда надо просуммировать большое количество чисел. Для этого переменной, в которую будет записываться сумма, в начале присваивается нулевое значение, затем делается цикл, где на каждом шаге к этой переменной добавляется очередное число.

Очень важная, фундаментальная идея, использованная в данном приёме, состоит в том, что результат выполнения каждого шага цикла зависит от значения переменной, вычисленного на предыдущем. Таким образом, вместо тривиального повторения одного и того же мы на каждом шаге получаем новый результат.

В приведенном примере очередное число добавляется к значению переменной `s`, полученному на предыдущем шаге. А к чему добавляется очередное число на самом первом? Чтобы было к чему добавлять, перед циклом обязательно должна присутствовать инициализация (присваивание начального значения) переменной, в которой накапливается сумма. Чаще всего требуется присвоить ей начальное значение 0.

Программистский анекдот в тему. Буратино подарили три яблока. Два он съел. Сколько яблок осталось у Буратино? Ответ «одно» — неправильный. В действительности, неизвестно, сколько осталось, так как не сказано, сколько яблок было у него до того, как ему подарили три новых. Мораль: не забывайте обнулять (и вообще инициализировать) переменные!

Аналогично накоплению суммы можно в отдельной переменной накапливать произведение. Переменной, в которой производится накопление, присваивается начальное значение 1. Для примера вычислим факториал некоторого числа. Факториалом целого числа n называется произведение всех целых чисел от 1 до n. Обозначается $n!$, то есть $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$.

Вычисляющая факториал программа выглядит так:

```
n = int(input('Сколько факториалов будем суммировать? '))
i = 2
p = 1
while i <= n:
```

```
        p = p * i
        i = i + 1
print(p)
```

3.2 Цикл обхода последовательности (for)

Цикл `while` не единственный способ организации повторения группы выражений. Также широко применяется цикл `for`, который представляет собой цикл обхода заданного множества элементов (символов строки, объектов списка или словаря) и выполнения в своем теле различных операций над ними[2].

Как правило, циклы `for` используются либо для повторения какой-либо последовательности действий заданное число раз, либо для изменения значения переменной в цикле от некоторого начального значения до некоторого конечного.

Для повторения цикла некоторое заданное число раз n можно использовать цикл `for` вместе с функцией `range`:

```
for i in range(n):
    Тело цикла
```

В качестве `n` может использоваться числовая константа, переменная или произвольное арифметическое выражение (например, `2**10`). Если значение `n` равно нулю или отрицательное, то тело цикла не выполнится ни разу.

Если задать цикл таким образом:

```
for i in range(a, b):
    Тело цикла
```

то индексная переменная `i` будет принимать значения от `a` до `b - 1` включительно, то есть первый параметр функции `range`, вызываемой с двумя параметрами, задает начальное значение индексной переменной, а второй параметр — значение, которое индексная переменная принимать не будет. Например, для того, чтобы просуммировать значения чисел от 1 до n, можно воспользоваться следующей программой:

```
summa = 0
for i in range(1, n+1):
    summa = summa + i
```

В этом примере переменная `i` принимает значения 1, 2, ..., n, и значение переменной `summa` последовательно увеличивается на указанные значения. Здесь опять видим прием накопления суммы.

[2] В большинстве языков программирования под циклом `for` принято понимать цикл со счётчиков. Такого цикла в Python нет, хотя существующий `for` может исполнять его функции. Цикл `for` языка Python фактически представляет собою цикл `foreach` — для каждого. Такой цикл существует во многих языках программирования, созданных в последние два десятилетия, например, в D.

Наконец, чтобы организовать цикл, в котором индексная переменная будет уменьшаться (в Pascal цикл с `downto`, цикл с отрицательным приращением), необходимо использовать функцию `range` с тремя параметрами. Первый параметр задает начальное значение индексной переменной, второй параметр — значение, до которого будет изменяться индексная переменная (не включая его!), а третий параметр — величину изменения индексной переменной. Например, сделать цикл по всем нечетным числам от 1 до 99 можно при помощи функции `range(1, 100, 2)`, а сделать цикл по всем числам от 100 до 1 можно при помощи `range(100, 0, -1)`.

Более формально, цикл `for i in range(a, b, d)` при `d > 0` задаёт значения индексной переменной `i = a`, `i = a + d`, `i = a + 2 * d` и так для всех значений, для которых `i < b`. Если же `d < 0`, то переменная цикла принимает все значения `i > b`.

Но в языке программирования Python цикл `for` имеет зачастую несколько иное применение. Например, список в Python относится к итерируемым объектам. Это значит, что его элементы можно обойти циклом `for`, причём переменная-счётчик будет на каждом шаге принимать значение очередного элемента цикла:

```
mylist = [12, 17.9, True, -8, False]
for j in mylist:
    print(j)
```

Программа выведет все элементы списка `mylist` в столбик:

```
12
17.9
True
-8
False
```

Приведённый способ можно назвать обходом по значению, поскольку автоматически создаваемая переменная `j` на каждом шаге принимает значение очередного элемента списка. Есть ещё один способ обойти список — по индексам, когда такая же переменная будет принимать номер очередного элемента:

```
mylist = [12, 17.9, True, -8, False]
for j in range(0, len(mylist), 1):
    print(j)
```

Вывод будет совсем другой:

```
0
1
2
3
4
```

Если написать:

```
mylist = [12, 17.9, True, -8, False]
for j in range(0, len(mylist), 1):
    print(mylist[j])
```

то вывод будет такой же, как в первом примере.

На самом деле, механизм обоих подходов один и тот же, потому что во втором варианте фактически неявно создаётся новая последовательность `range(0, len(mylist), 1)`, содержащая номера всех элементов списка `mylist` — диапазон от нуля до длины `mylist` с шагом 1, и этот новая последовательность обходится по значению.

Напишем с помощью цикла `for` вывод ряда Фибоначчи:

```
fib1 = 0
fib2 = 1
n = 10
summa = 0
for i in range(n):
        summa = fib1 + fib2
        print(summa)
        fib1 = fib2
        fib2 = summa
```

В результате будет выведено следующее:

```
1
2
3
5
8
13
21
34
55
89
```

С помощью цикла `for` можно перебирать строки, если не пытаться их при этом изменять:

```
str1 = 'Привет'
for i in str1:
    print(i, end='␣')
```

Будет выведено:

```
П р и в е т
```

Здесь можно видеть, что у функции `print` есть параметр `end`. По умолчанию `end = '\n'` — неотображаемому символу новой строки. В предыдущем примере параметру `end` был присвоен символ пробел.

Цикл `for` используется и для работы со словарями:

```
dic = {'cat':'кошка', 'dog':'пёс',
       'bird':'птица', 'mouse':'мышь'}
for i in dic:
   dic[i] = dic[i] + '_ru'
print(dic)
```

Вывод программы:

```
{'bird': 'птица_ru', 'cat': 'кошка_ru', 'mouse': 'мышь_ru', 'dog': 'пёс_ru'}
```

На практике часто не важно, каким образом вы решите задачу. Искать сразу оптимальное решение не следует, достаточно найти просто правильное (а их может быть множество). Как писал Дональд Кнут в своём фундаментальном труде «Искусство программирования», «преждевременная оптимизация — корень многих зол».

3.3 Некоторые основные алгоритмические приёмы

3.3.1 Приёмы накопления суммы и произведения. Их комбинация

В разделе про цикл `while` рассматривались примеры с накоплением суммы и произведения. Эти же приёмы можно и нужно применять при работе с циклом `for`. Так же их можно комбинировать. Рассмотрим следующий пример: необходимо вычислить значение выражения $1! + 2! + \cdots + n!$

Решение в лоб состоит в том, чтобы в теле цикла, осуществляющего суммирование, производить вычисление факториала:

```
n = int(input('Сколько факториалов будем суммировать? '))
s = 0
for i in range(1, n+1):
    # Вычисление факториала от i:
    p = 1
    for k in range(1, i+1):
        p = p * k;
    # Добавление вычисленного факториала к сумме:
    s = s + p
print(s)
```

Циклы позволяют повторять выполнение любого набора операторов. В частности, можно повторять много раз выполнение другого цикла. Такие циклы называются *вложенными*. В приведённом выше примере один цикл `for` вложен в другой цикл `for`.

Типичная ошибка, когда в качестве счётчиков вложенных циклов (`i` и `k` в приведённом примере) используется одна и та же переменная. То есть, нельзя в каждом из циклов использовать одну переменную `i`. Ваша программа запустится, но делать будет вовсе не то, что вы от неё ждёте. В приведённом примере, если допустить ошибку, заменив переменную `k` на `i`, внешний цикл выполнится всего 1 раз вместо 4-х. Возможна также ситуация, когда такая ошибка приведет к зацикливанию: внешний цикл будет выполняться бесконечно долго — программа зависнет.

Заметим, что при вычислении факториала на каждом шаге получается факториал все большего целого числа. Эти «промежуточные» результаты однократного вычисления факториала и можно суммировать:

```
n = int(input('Сколько факториалов суммировать? '))
s = 0
p = 1
for i in range(1, n+1):
    p = p * i
    s = s + p
print(s)
```

Стоит отметить, что в основе рассмотренных ранее алгоритмических приёмов накопления суммы и произведения лежит фундаментальная идея о том, что результат вычислений на каждом шаге цикла должен зависеть от результата вычислений на предыдущем шаге. Обобщенным математическим выражением этой идеи являются *рекуррентные соотношения*.

В наших примерах в качестве рекуррентных соотношений выступали, например, формулы `p = p * i` и `s = s + p`. Причём, последнее выражение (`s = s + p`) является сложною рекурсией, когда значение `s` зависит не только от своего прошлого значения, но и от значения `p` на прошлом шаге.

Для лучшего понимания решим задачу: пусть дано рекуррентное соотношение $x_{n+1} = 1 - \lambda x_n^2$. В нелинейной динамике это соотношение называют логистическим отображением. Оно, например, может использоваться для приближённого описания изменения численности популяций некоторых животных во времени. Пусть начальное значение $x_0 = 1$. Параметр $\lambda = 0.75$. Необходимо найти x_5.

```
x = 1
for i in range(1, 6):
    x = 1 - 0.75*x**2
print(x)
```

Вывод программы:

```
0.35989018693629404
```

Однократное вычисление следующих значений по предыдущим посредством рекуррентных соотношений называется *итерацией*. А процесс вычислений с помощью рекуррентных соотношений — *итерированием*.

Задание 7 Придумайте рекуррентное соотношение, задающее следующие числовые последовательности:

a) $1, 2, 3, 4, \ldots$

b) $0, 5, 10, 15, \ldots$

c) $1, 1, 1, 1, \ldots$

d) $1, -1, 1, -1, \ldots$

e) $1, -2, 3, -4, 5, -6 \ldots$

f) $2, 4, 8, 16, \ldots$

g) $2, 4, 16, 256, \ldots$

h) $0, 1, 2, 3, 0, 1, 2, 3, 0, \ldots$

i) $1!, 3!, 5!, 7!, \ldots$

Важно!!! Если в написанной вами формуле вам встречается значок суммы $\sum_{i=0}^{N-1} x_i$, то вы сразу должны представлять себе цикл `for` с накоплением суммы внутри:

```
x = 0
for i in range(0, N):
    x = x + i
```

Аналогично, если в задании вам встречается значок произведения $\prod_{i=0}^{N-1} x_i$, то вы сразу должны представлять себе цикл `for` с накоплением произведения внутри:

```
x = 1
for i in range(0, N):
    x = x * i
```

3.3.2 Счётчик событий

Часто требуется подсчитать, сколько раз во время вычислений наступает то или иное событие (выполняется то или иное условие). Для этого вводится вспомогательная переменная, которой в начале присваивается нулевое значение, а после каждого наступления события она увеличивается на единицу.

Пример задачи 10 (Счётчик событий) Пользователь вводит 10 чисел. Определите, сколько из них являются одновременно чётными и положительными.

Решение задачи 10 Решение можно записать следующим образом:

```
Counter = 0 # Обнуляем переменную-счётчик
for i in range(0, 10):
    x = int(input('Введите число: '))
    if (x%2 == 0) and (x > 0):
        Counter = Counter + 1
print(Counter)
```

Вывод программы:

```
Введите число: 2
Введите число: 3
Введите число: 4
Введите число: -2
Введите число: -4
Введите число: 3
Введите число: 5
Введите число: 7
Введите число: 6
Введите число: 3
3
```

3.3.3 Досрочное завершение цикла

Отметим два простых оператора `break` и `continue`, с помощью которых можно управлять ходом выполнения цикла. Оператор `break` прерывает выполнение цикла, управление передается операторам, следующим за оператором цикла. Оператор `continue` прерывает выполнение очередного шага цикла и возвращает управление в начало цикла, начиная следующий шаг.

```
for n in range(10):
    if n%2 == 0:
        continue
    if n == 7:
        break
    print(n)
```

Данная программа будет печатать только нечётные числа из-за срабатывания `continue`. Цикл прекратит выполняться, когда `n` станет равно 7. В итоге вывод программы таков:

```
1
3
5
```

3.3.4 Поиск первого вхождения

Ранее мы подсчитывали количество положительных чётных чисел в последовательности ввода. Зачастую нужен не подсчёт, а только проверка, произошло ли за время вычислений некоторое событие. Например, необходимо проверить, *содержится ли* в некоторой последовательности хотя бы одно отрицательное число. Для того, чтобы утверждать, что отрицательных чисел в последовательности нет, необходимо просмотреть её всю. Если же такое число в ней есть, достаточно добраться до него, после чего цикл можно закончить. Получается цикл `for` с проверкой и оператором `break` внутри.

```
seq = (12, 54, 0, -7, 22, -11, 54, 0, -7)
for x in seq:
    if x < 0:
        print(x)
        break
```

В этом коде пока нет места для действий на случай, если отрицательное число не найдено. В самом деле, и после `break`, и после «естественного» завершения цикла программа продолжит работу с одной и той же строки. В Python на этот счёт предусмотрена конструкция `else`, относящаяся *к циклу*. Работает она так, как и ожидается: только если цикл завершился «естественным путём» — потому что проверка условия в `while` оказалась ложной или последовательность в `for` закончилась. Если же выход из цикла произошёл по `break`, блок операторов внутри `else` не выполняется. Так что для того, чтобы вывести какое-нибудь сообщение, если отрицательных чисел в последовательности нет, соответствующий `print()` надо добавить в такую конструкцию.

```
seq = (12, 54, 0, 7, 22, 11, 54, 0, 7)
for x in seq:
    if x < 0:
        print(x)
        break
else:
    print("Отрицательных чисел нет")
```

3.3.5 Обработка исключений

Исключения (exceptions) — ещё один тип данных в Python. Часто в работе программы возникают ошибки, препятствующие её дальнейшему выполнению.

Вот простой пример такой ошибки:

```
>>> 10/0
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    10/0
ZeroDivisionError: division by zero
```

В данном случае интерпретатор сообщил нам об ошибке `ZeroDivisionError`, то есть о делении на ноль. Также возможны и другие исключения, например, несовпадающие типы:

```
>>> 1 + 'a'
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    1 + 'a'
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Во всех таких случаях интерпретатор прерывает работу программы, поскольку либо не может понять очередную инструкцию, либо предполагает, что полученное в её результате значение (например, при делении на ноль или взятии логарифма отрицательного числа) недопустимо. Это считается правильным, поскольку указывает программисту на наличие ошибки. Однако иногда в программе могут возникать ошибки, которые невозможно быстро поправить, а работу программы останавливать нельзя. В таком случае принято говорить об исключении. Такие исключения можно *обрабатывать*, для чего используется конструкция `try`–`except`. Пример применения этой конструкции:

```
a = int(input('Введите делимое = '))
b = int(input('Введите делитель = '))
try:
    print(a/b)
except ZeroDivisionError:
    print('Деление на ноль')
```

Надо понимать, что обработка исключений — это *крайняя мера*, которая используется, либо если иначе починить программу без существенного переписывания быстро нельзя, либо если программа зависит от сторонних модулей, которые не могут быть исправлены, но способны вызвать ошибку. Злоупотребление конструкцией `try`–`except` быстро приводит программу в неработоспособное состояние, поскольку эта конструкция в действительности *ничего не исправляет*, а просто помогает игнорировать проблему в данном месте. В большинстве случаев вместо `try`–`except` достаточно добавить просто проверку условия.

3.4 Отладка программы

В большинстве случаев многие даже несложные программы, будучи написаны, работают не так, как предполагал автор. Возможно, вы уже убедились в этом при написании простеньких программ из предыдущей главы. Как минимум, у половины из вас появлялись синтаксические ошибки (забыли поставить скобочку или кавычку). В этом разделе после изучения таких сравнительно сложных конструкций, как циклы, количество ваших ошибок резко увеличится. Но расстраиваться не стоит. Нужно всегда помнить, что процесс написания программы состоит из двух этапов: кодирование (написание кода программы, занимает менее трети времени) и отладки (занимает более двух третей времени).

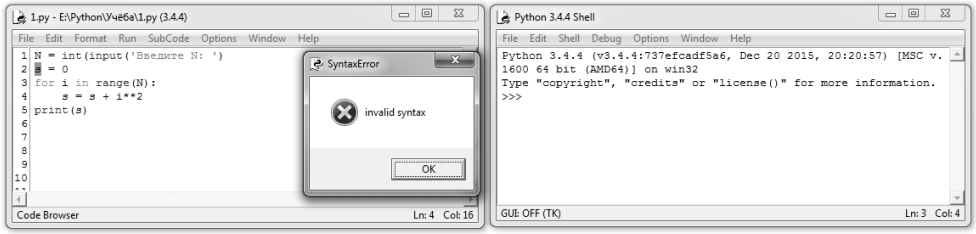


Рис. 3.1. Пример ошибки, выявляемой препроцессором.

Все ошибки можно условно разделить на следующие три категории, которые разберём на примере программы, считающей сумму квадратов целых чисел от 0 до N.

3.4.1 Ошибки, выявляемые препроцессором

В интерпретатор Python встроена специальная программа — препроцессор. У препроцессора несколько функций, в частности, он переводит текст программы в специальный байт-код, понятный для интерпретатора. В процессе перевода текста в байт-код препроцессор вынужден анализировать синтаксис вашей программы, для чего используется *синтаксический анализатор*, проверяющий ваш текст с целью понять, похож ли он на текст программы на Python по ряду формальных признаков. Если препроцессор не может понять смысл тех или иных символов в вашем тексте, он чаще всего указывает вам на ошибку типа (`SyntaxError`). При синтаксической ошибке возникает диалоговое окно, которое предотвращает запуск интерпретатора (рис. 3.1), так как нет смысла запускать то, что непохоже на программу.

Самый важный вид синтаксической ошибки с точки зрения препроцессора — это ошибка расстановки отступов, поскольку она нарушает всю структуру программы. Если вы попытаетесь запустить на исполнение вот такой код:

```
N = int(input('Введите N: '))
s = 0
 for i in range(N):
    s = s + i**2
  if s % 2 == 0:
     print(s)
```

то ничего не выйдет: вы получите сообщение `unexpected indent` — неожиданный отступ, и пробел перед ключевым словом `for` будет подсвечен красным. Исправить такую ошибку совсем несложно: нужно просто нормально расставить отступы в соответствии с логикою программы.

Ещё одна популярная ошибка на примере того же кода:

```
N = int(input('Введите N: '))
s = 0
for i in range(N):
s = s + i**2
if s % 2 == 0:
print(s)
```

Интерпретатор выдаст: `expected an indented block`: нужен отступ для тех команд, которые лежат внутри цикла `for` и условного оператора `if`.

Бывает, что в результате опечаток возникают недопустимые с точки зрения интерпретатора выражения. Например, можно допустить следующую ошибку:

```
s = s + 2i
```

С точки зрения правил Python выражение `2i` никогда не может возникнуть: имя переменной не может начинаться с цифры, а для интепретации `2` и `i` как разных сущностей между ними должен быть знак какой-нибудь арифметической или логической операции (чаще всего забывают знак умножения `*`, поскольку в математических выражениях он обычно опускается).

Чуть сложнее разобраться с другим подвидом синтаксических ошибок, вызванных неверною расстановкою скобок:

```
N = int(input('Введите N: ')
s = 0
for i in range(N):
    s = s + i**2
print(s)
```

Такой пример вызовет ошибку `invalid syntax`, причём укажет на символ `s` в начале второй строки, что может сбить вас с толку. На самом деле проблема в несоответствии числа открывающихся и закрывающихся скобок в предыдущей строке. Интерпретатор в поисках второй закрывающейся скобки дошёл до строки, следующей за тою, где совершена ошибка, и, поняв, что искать дальше бессмысленно (на новой строке по правилам её уже не может быть), выдал ошибку. При неверном числе скобок интерпретатор всегда выдаёт ошибку в начале следующей строки.

3.4.2 Ошибки, выявляемые интерпретатором

Если вы успешно справились с синтаксисом, другие ошибки за вас может выявить интерпретатор во время исполнения программы. Интерпретатор даже напишет, что это за ошибка и в какой она строчке кода (рис. 3.2).

Ошибки, выявляемые интерпретатором, также называются *ошибками времени исполнения*. Самые распространённые из них — *ошибки пространства имён*. Это такие ошибки, когда имя функции, метода или введённой вами же переменной написано неверно. Кроме них часто возникают ошибки неверной типизации и

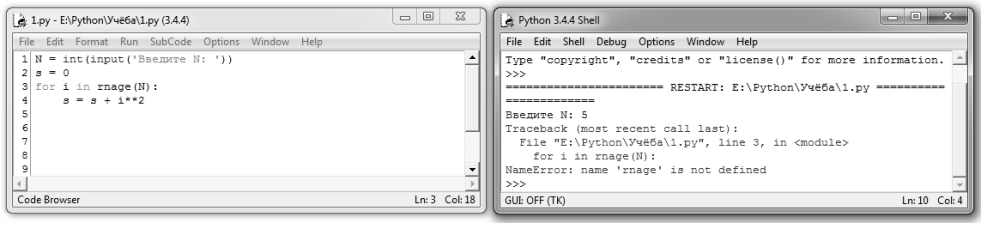


Рис. 3.2. Пример ошибки, выявляемой интерпретатором во время исполнения.

ошибки, связанные с недопустимыми операциями с памятью компьютера. Далее основные ошибки разобраны более подробно:

1. `NameError` — ошибка в имени. Вот пример неправильно написанного имени стандартной функции `range`:

   ```
   N = int(input('Введите N: '))
   s = 0
   for i in rnage(N):
       s = s + i**2
   ```

   При попытке выполнить этот код получится следующее:

   ```
   Traceback (most recent call last):
     File "/home/paelius/test_error.py", line 3, in <module>
       for i in rnage(N):
   NameError: name 'rnage' is not defined
   ```

   Как видим, интерпретатор, дойдя до строчки с ошибкою, указал нам, что имя `rnage` ему неизвестно (`NameError: name 'rnage' is not defined`). Найти и исправить такие ошибки обычно довольно просто, в том числе, благодаря тому, что все встроенные функции (`range`, `len`, `sorted`, `sum`, `int` и другие) выделяются цветом (в IDLE это фиолетовый). Поэтому вы можете контролировать себя уже на этапе написания кода: если `range` не подсветилось, значит, вы написали что-то неверно. Аналогично другим — жёлтым — цветом выделяются встроенные операторы и их части: `in`, `for`, `while`, `if`, `else`, `from`, `import`, `as`, `with`, `break`, `continue`, а также встроенные значения: `True`, `False` и `None`.

2. `AttributeError` — ошибочный атрибут. `NameError` — не единственная лексическая ошибка. Перепишем задачу так, что сначала положим все квадраты чисел в список, а затем воспользуемся стандартной функцией `sum`:

   ```
   l = []
   for i in range(N):
   ```

```
    l.apend(i**2)
print(sum(l))
```

В этой программе есть одна трудно уловимая ошибка: в методе `append` пропущена одна буква `p`. В результате мы получим:

```
AttributeError:
Traceback (most recent call last):
  File "/home/paelius/test_error.py", line 4, in <module>
    l.apend(i**2)
AttributeError: 'list' object has no attribute 'apend'
```

Интерпретатор указывает нам, что объект данного типа не имеет атрибута (метода или поля) `apend`. Поскольку методы даже стандартных объектов таких, как список, никак не подсвечиваются, обнаружить эту ошибку заранее сложно. Плюс в том, что исправление подобной ошибки не составит труда. Есть, однако, один способ снизить вероятность их появления: для длинных методов, имя которых вы плохо помните, лучше пользоваться автодополнением.

3. `TypeError` — ошибка типов. Всегда следует помнить, что в третьей версии Python функция `input()` возвращает строковую переменную. Если попробовать написать что-то подобное:

```
a = input()
b = input()
print(a/b)
```

то получим ошибку:

```
Traceback (most recent call last):
  File "E:/Python/1.py", line 3, in <module>
    print(a/b)
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

Для выявления подобных ошибок полезно выводить на экран тип переменной командою `print(type(a))`.

4. `ValueError` — ошибка значения, являющаяся ещё одним видом ошибок, связанных с типами данных. Она возникает, например, при попытке извлечь корень из отрицательного числа. Причём интересно, что ошибка будет выдана только при использовании функции `sqrt` из модуля `math`, а при возведении в степень стандартным образом с помощью оператора `**` число будет просто конвертировано в комплексное:

```
>>> (-3)**(1/2)
(1.0605752387249068e-16+1.7320508075688772j)
>>> import math
>>> math.sqrt(-3)
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    math.sqrt(-3)
ValueError: math domain error
```

5. `IndexError` — ошибка индекса. Появляется при обращении к несуществующему элементу строки или списка:

```
>>> L = list(range(10))
>>> L
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> L[10]
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    L[10]
IndexError: list index out of range
```

6. `OverflowError` — ошибка переполнения. Возникает, когда в результате вычислений получается слишком большое действительное число:

```
p = 1.5
for i in range(2, 100):
    p = p**i
print(p)
```

В результате будет выдана ошибка:

```
Traceback (most recent call last):
  File "E:/Python/1.py", line 3, in <module>
    p = p**i
OverflowError: (34, 'Result␣too␣large')
```

3.4.3 Ошибки, выявляемые разработчиком

Их ещё можно назвать логическими. Это такие ошибки, когда ваша программа работает, но выдаёт что-то не то. Это наиболее сложный тип ошибок, потому что их нужно не только устранять, но и выявлять самостоятельно, а для этого необходимо думать.

Вернёмся к нашей программе расчёта суммы квадратов последовательных целых чисел:

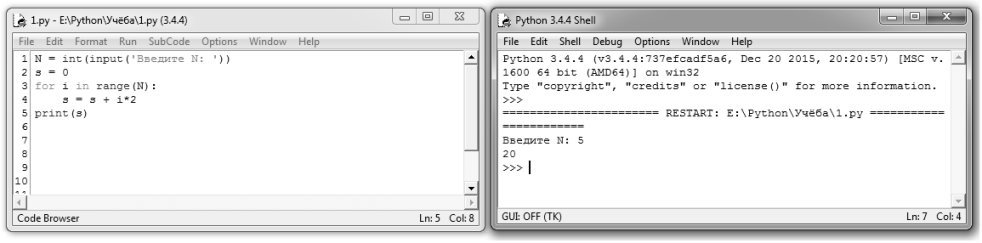


Рис. 3.3. Пример логической ошибки, которая может быть выявлена только разработчиком.

```
N = int(input('Введите N: '))
s = 0
for i in range(N):
    s = s + i*2
print(s)
```

В окне интерпретатора вы увидите:

```
Введите N: 5
20
```

Казалось бы, всё неплохо: программа работает и выдаёт что-то разумное. Но не спешите радоваться, ведь на самом деле $0^2 + 1^2 + 2^2 + 3^2 + 4^2 = 30$, а вовсе не 20, как выдала наша программа (рис. 3.3). В чём же проблема?

Для выявления логических ошибок применяется такой приём, как тестирование. Вы уже неосознанно прибегали к нему ранее. Тестирование — это составление входных данных для программы, для которых вы можете сами составить выходные. Совокупность набора входных данных и соответствующих им выходных данных называется «тестом». В нашем случае весь тест — это два числа: входу 5 соответствует выход 30. Для сложных программ тесты будут сложнее и больше, и их понадобится не один, а много, чтобы охватить как можно больше разных вариантов поведения программы.

Когда факт наличия ошибки установлен, нужно найти конкретное место, где она возникла и устранить её. Для этого используется отладка. Отладка — это пошаговое исполнение программы с выводом промежуточных результатов, которое позволяет определить, в промежутке между какими операторами произошла логическая ошибка. В компилируемых языках программирования таких, например, как Pascal и C, для отладки применяется специализированная программа-отладчик. В интепретируемых языках, в частности в Python, в этом нет большой необходимости, поскольку программа и так выполняется пошагово, и вы можете вывести любые данные в любой момент с помощью стандартной функции print. При отладке важно суметь сформулировать гипотезу: «что нужно проверить»?

Иногда это удаётся не с первого раза. Попробуем рассмотреть это на нашем примере. Итак, первая гипотеза: счётчик `i` принимает не те значения, попробуем выводить его значения в цикле:

```
N = int(input('Введите N: '))
s = 0
for i in range(N):
    print(i)
    s = s + i*2
print(s)
```

Получим:

```
Введите N: 5
0
1
2
3
4
20
```

Как видим, со счётчиком всё в порядке. Тогда проверим, всё ли в порядке с очередным элементом суммы:

```
N = int(input('Введите N: '))
s = 0
for i in range(N):
    s = s + i*2
    print(i*2)
print(s)
```

Получим:

```
0
2
4
6
8
20
```

Мы получили последовательность 0, 2, 4, 6, 8, в то время как должны были получить 0, 1, 4, 9, 16, значит, наше предположение подтвердилось: очередной элемент суммы вычисляется неверно. Честно говоря, уже на этапе написания отладочного вывода можно было заметить, что `i*2` это не совсем то, что нужно, ведь вместо возведения в степень мы написали умножение (забыли одну звёздочку).

В более сложных программах, однако, вам часто придётся проверять по нескольку гипотез и выводить значительное число отладочной информации, чтобы обнаружить точную причину ошибки. Помните: отладка — это мощный и эффективный способ борьбы с логическими ошибками, но работает он только тогда, когда вы способны внятно сформулировать гипотезу, т. е. определить, что проверять. Если начать выводить всё подряд, вы быстро потеряетесь в отладочной информации и ничего не сможете найти.

Некоторые наиболее распространённые ошибки были проклассифицированы нами в виде схемы, приведённой на рис. 3.4. Схема не претендует на полноту, но будет полезна для начинающих во многих типичных случаях.

3.5 Задания на циклы

Пример задачи 11 Решим популярную задачу по нахождению всех простых чисел до некоторого целого числа n. Классический алгоритм решения этой задачи носит название «Решето Эратосфена». Для нахождения всех простых чисел не больше заданного числа n, следуя методу Эратосфена, нужно выполнить следующие шаги:

1. Выписать подряд все целые числа от двух до n ($2, 3, 4, \ldots, n$).
2. Пусть переменная `p` изначально равна двум — первому простому числу.
3. Зачеркнуть в списке числа от $2p$ до n, считая шагами по p (это будут числа кратные p: $2p, 3p, 4p, \ldots$).
4. Найти первое незачёркнутое число в списке, большее, чем `p`, и присвоить значению переменной p это число.
5. Повторять шаги 3 и 4, пока возможно.
6. Теперь все незачёркнутые числа в списке — это все простые числа от 2 до n.

Решение задачи 11 На практике алгоритм можно улучшить следующим образом. На шаге №3 числа можно зачеркивать, начиная сразу с числа p^2, потому что все составные числа меньше него уже будут зачеркнуты к этому времени. И, соответственно, останавливать алгоритм можно, когда p^2 станет больше, чем n.

```
from math import sqrt
n = int(input("вывод простых чисел до ... "))
a = list(range(n)) # создаём список из n элементов
# Вторым элементом является единица, которую не
# считают простым числом. Забиваем её нулем:
a[1] = 0
```

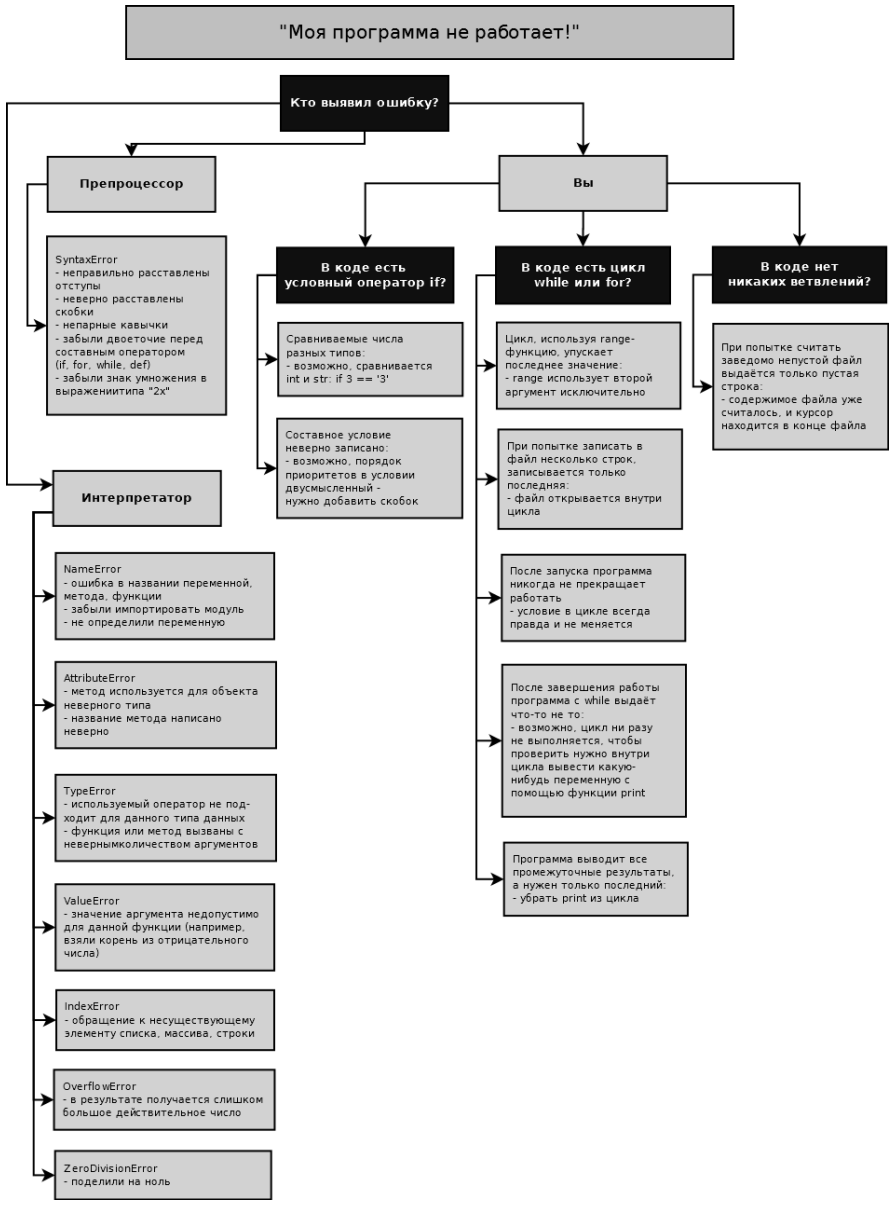


Рис. 3.4. Возможные источники ошибок и алгоритм их нахождения.

```
# Перебор всех элементов до заданного числа:
for p in range(2, int(sqrt(n))+1):
    if a[p] != 0: # если он не равен нулю, то
        j = p ** 2 # удвоить: текущий элемент простое число
        while j < n:
            a[j] = 0 # заменить на 0
            j = j + p # перейти в позицию на m больше
# Вывод простых чисел на экран:
b = []
for i in a:
    if a[i] != 0:
        b.append(a[i])
print(b)
```

Вывод программы:

```
вывод простых чисел до числа ... 70
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67]
```

Задание 8 (Задания на цикл с условием) Выполнять три задания в зависимости от номера в списке группы в алфавитном порядке. Необходимо сделать задания №m, №m+5, №m+10, m=(n-1)%5+1, где n — номер в списке группы.

1. Напишите программу, которая будет суммировать вводимые с клавиатуры числа до тех пор, пока они положительны.
2. Напишите программу, которая будет суммировать вводимые с клавиатуры числа до тех пор, пока они отрицательны.
3. Напишите программу, которая будет суммировать вводимые с клавиатуры числа до тех пор, пока они не равны нулю.
4. Напишите программу, которая будет суммировать вводимые с клавиатуры числа до тех пор, пока они чётные.
5. Дано число n. Напечатать те натуральные числа, квадрат которых не превышает n.
6. Дано число n. Найти первое натуральное число, квадрат которого больше n.
7. Дано число n. Среди чисел 1, $1+\frac{1}{2}$, $1+\frac{1}{2}+\frac{1}{3}$, $\ldots$ найдите первое, большее числа n.
8. Дано число a ($1 \leqslant a \leqslant 1.5$). Среди чисел $1+\frac{1}{2}$, $1+\frac{1}{3}$, $1+\frac{1}{4}$, $\ldots$ (заметим, что каждое следующее число в последовательности меньше предыдущего) найдите первое, меньшее a.

9. Напишите программу, которая запрашивает у пользователя числа до тех пор, пока каждое следующее число больше предыдущего. В конце программа сообщает, сколько чисел было введено.

10. Напишите программу, которая запрашивает у пользователя числа до тех пор, пока каждое следующее число меньше предыдущего. В конце программа сообщает, сколько чисел было введено.

11. Напишите программу, которая запрашивает у пользователя числа до тех пор, пока каждое следующее число целое. В конце программа сообщает, сколько чисел было введено.

12. Напишите программу, которая запрашивает у пользователя числа до тех пор, пока каждое следующее число меньше 10. В конце программа сообщает, сколько чисел было введено.

13. Дано натуральное число, в котором все цифры различны. Определить порядковый номер его максимальной цифры, считая номера: от конца числа; от начала числа.

14. Дано натуральное число, в котором все цифры различны. Определить порядковый номер его минимальной цифры, считая номера: от конца числа; от начала числа.

15. Дано натуральное число. Определить, сколько раз в нем встречается максимальная цифра (например, для числа 132233 ответ равен 3, для числа 46336 — двум, для числа 12345 — одному).

Задание 9 (Задания на цикл со счётчиком) *Выполнять три задания в зависимости от номера в списке группы в алфавитном порядке. Необходимо сделать задания* №m, №m+5, №m+10, m=(n-1)%5+1, *где* n *— номер в списке группы.*

1. Напишите программу, вычисляющую сумму всех чётных чисел в диапазоне от 1 до 90 включительно.

2. Напишите программу, вычисляющую сумму всех чётных чисел в диапазоне от a до b включительно (вводятся с клавиатуры).

3. Напишите программу, вычисляющую сумму всех нечётных чисел в диапазоне от 1 до 90 включительно.

4. Напишите программу, вычисляющую сумму всех нечётных чисел в диапазоне от a до b включительно (вводятся с клавиатуры).

5. Напечатайте таблицу умножения на 5, желательно печатать в виде:

$$1 \times 5 = 5$$
$$2 \times 5 = 10$$
$$\dots$$
$$9 \times 5 = 45$$

   Вместо знака умножения $\times$ можно использовать строчную латинскую букву «x».

6. Напечатайте таблицу умножения на 9, желательно печатать в виде:

$$1 \times 9 = 9$$
$$2 \times 9 = 18$$
$$\dots$$
$$9 \times 9 = 81$$

   Вместо знака умножения $\times$ можно использовать строчную латинскую букву «x».

7. Напечатайте таблицу умножения на целое число n, n вводится с клавиатуры ($2 \leqslant n \leqslant 9$), желательно печатать в виде:

$$1 \times n = \dots$$
$$2 \times n = \dots$$
$$\dots$$
$$9 \times n = \dots$$

   Вместо знака умножения $\times$ можно использовать строчную латинскую букву «x». Внимание! Не нужно печатать символ `n`, вместо этого нужно печатать введённое значение.

8. Напечатать таблицу стоимости 50, 100, 150, ..., 1000 г сыра (стоимость 1 кг сыра вводится с клавиатуры).

9. Напечатать таблицу стоимости 100, 200, 300, ..., 2000 г конфет (стоимость 1 кг конфет вводится с клавиатуры).

10. Найти сумму всех целых чисел от 10 до 100;

11. Найти сумму всех целых чисел от a до 100 включительно (значение a вводится с клавиатуры).

12. Найти сумму всех целых чисел от 10 до b включительно (значение b вводится с клавиатуры).

13. Найти сумму всех целых чисел от a до b включительно (значения a и b вводятся с клавиатуры).

14. Найти произведение всех целых чисел от 10 до 100 включительно. Обратите внимание, что Python может работать с целыми числами неограниченного размера!

15. Найти произведение всех целых чисел от a до b включительно (значения a и b вводятся с клавиатуры).

Задание 10 (Задания на комбинацию циклов со счётчиком и условием) *Выполнять одно задание с номером* `(n-1)%8+1` *в зависимости от номера* n *в списке группы в алфавитном порядке.*

1. За столом сидят n гостей (вводится с клавиатуры), перед которыми стоит пирог. Пирог и его части можно делить только пополам. Определите, сколько раз нужно делить пирог на ещё более мелкие части, чтобы:

   - каждому из гостей достался хотя бы 1 кусок;
   - как минимум половине гостей досталось по 2 куска;
   - каждому гостю досталось по 1 куску и при этом ещё хотя бы 10 кусков осталось в запасе.

2. Ученик 4-го класса Василий время от времени начинает прогуливать школу. Первый раз он прогуливает 2 дня в конце первого месяца, через месяц — 3 дня, ещё через месяц — 4 дня и так далее. За каждый день прогулов Василию ставят по 2 двойки, плюс ещё по 3 двойки он получает в месяц на занятиях. Сколько раз Василий может прогуливать школу (сколько раз уйти в «загул») и сколько дней прогуляет, чтобы не быть отчисленным, если отчисление грозит ему за 70 двоек? Продолжительность учебного года — 9 месяцев, выйти из «загула» досрочно (не прогуляв положенное число дней) Василий не в состоянии, каникулами пренебречь.

3. В детском садике n детей играют в следующую игру. Перед ними гора из m кубиков, первый ребёнок вынимает из кучи 1 кубик, каждый последующий ребёнок — в два раза больше предыдущего и так по кругу. Если число кубиков, которые нужно вынуть, превышает 25, из него вычитается 25 и отсчёт идёт от уменьшенного числа, например, вместо 32 кубиков будет вынуто 7, затем 14 и т. д. Проигравшим считается тот, кто не смог вытащить нужное число кубиков (в куче осталось недостаточно). Определите проигравшего.

4. Последовательность Фибоначчи определяется рекуррентным соотношением $x_{n+1} = x_n + x_{n-1}$, где $x_0 = 1$ и $x_1 = 1$. Найти первое число в последовательности Фибоначчи, которое больше 1000.

5. Для n-го члена в последовательности Фибоначчи существует явная формула:

$$x_n = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^{n+1} - \left(\frac{1-\sqrt{5}}{2} \right)^{n+1} \right)$$

Поскольку операции с вещественными числами происходят с конечной точностью, то с ростом n, результат вычисления по этой формуле будет все больше отличаться от настоящего числа Фибоначчи. Найдите n, начиная с которого, отличие от истинного значения превысит 0.001.

6. Создайте программу, играющую с пользователем в орлянку. Программа должна спрашивать у пользователя: орёл или решка. Если пользователь вводит 0, то выбирает орла, 1 — решку, любое другое число — конец игры. Программа должна вести учёт выигрышей и проигрышей и после каждого раунда сообщать пользователю о состоянии его счёта. Пусть вначале на счету 3 рубля и ставка в каждом коне 1 рубль. Если денег у пользователя не осталось — игра прекращается.[3]

7. Гражданин 1 марта открыл счет в банке, вложив 1000 руб. Через каждый месяц размер вклада увеличивается на 2% от имеющейся суммы. Определить:

   - за какой месяц величина ежемесячного увеличения вклада превысит 30 рублей;
   - через сколько месяцев размер вклада превысит 1200 руб.

8. Начав тренировки, лыжник в первый день пробежал 10 км. Каждый следующий день он увеличивал пробег на 10% от пробега предыдущего дня. Определить:

   - в какой день он пробежит больше 20 км;
   - в какой день суммарный пробег за все дни превысит 100 км.

[3] Выпал орёл или решка, программа определяет с помощью функции `randint(a, b)` из стандартного модуля `random`, которая возвращает случайное целое число n, $a \leqslant n \leqslant b$.

Глава 4

Функции

4.1 Функции в программировании

Функции в программировании можно представить как изолированный блок кода, обращение к которому в процессе выполнения программы может быть многократным. Зачем нужны такие блоки инструкций? В первую очередь, чтобы сократить объем исходного кода: рационально вынести часто повторяющиеся выражения в отдельный блок и затем по мере надобности обращаться к нему.

Для того, чтобы в полной мере осознать необходимость использования функций, приведём сложный, но чрезвычайно полезный пример вычисления корня нелинейного уравнению с помощью метода деления отрезка пополам.

Пусть на интервале $[a; b]$ имеется ровно 1 корень уравнения $f(x) = 0$. Значит, $f(a)$ и $f(b)$ имеют разные знаки. Используем этот факт. Найдём $f(c)$, где $c = \frac{a+b}{2}$. Если $f(c)$ того же знака, что и $f(a)$, значит корень расположен между c и b, иначе — между a и c.

Пусть теперь начало нового интервала (будь то a или c в зависимости от того, где находится корень) обозначается a_1 (что означает после первой итерации), а конец, соответственно, b_1. Исходные начало и конец также будем обозначать a_0 и b_0 для общности. В результате нам удасться снизить неопределённость того, где находится корень, в два раза.

Такой процесс можно повторять сколько угодно раз (при расчёте на компьютере столько, сколько позволяет точность представления данных ЭВМ), последовательно заужая интервал, в котором находится корень. Обычно процесс заканчивают по достижении на n-ой итерации интервалом $[a_n; b_n]$ величины менее некоторого заранее заданного ε.

Итак, напишем программу для вычисления корня нелинейного уравнения $f(x) = x^2 - 4$:

```
from math import *
a = -10
b = 10
```

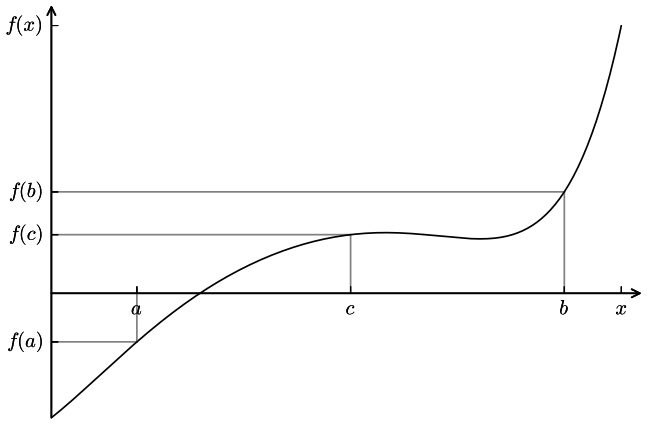


Рис. 4.1. Иллюстрация к методу деления отрезка пополам.

```
while (b-a) > 10**(-10):
    c = (a+b)/2
    f_a = a**2-4
    f_b = b**2-4
    f_c = c**2-4
    if f_a*f_c > 0:
        a = c
    else:
        b = c
print((a+b)/2)
```

Если мы захотим вычислить корень другой нелинейной функции, например, $f(x) = x^2 + 4x + 4$, то придётся переделывать выражения сразу в трёх строчках. Кажется логичным выделить наше нелинейное уравнение в отдельный блок программы. Перепишем программу с помощью функции:

```
from math import *

def funkcija(x):
    f = x**2+4*x+4
    return f

a = -10
b = 10
while (b-a) > 10**(-10):
```

```
    c = (a+b)/2
    f_a = funkcija(a)
    f_b = funkcija(b)
    f_c = funkcija(c)
    if f_a*f_c > 0:
        a = c
    else:
        b = c
print((a+b)/2)
```

Программный код подпрограммы описывается единожды перед телом основной программы, затем из основной программы можно им пользоваться многократно. Обращение к этому программному коду из тела основной программы осуществляется по его имени (имени подпрограммы).

Инструкция `def` — это команда языка программирования Python, позволяющая создавать функцию; `funkcija` — это имя функции, которое (так же как и имена переменных) может быть почти любым, но желательно осмысленным. После в скобках перечисляются параметры функции. Если их нет, то скобки остаются пустыми. Далее идет двоеточие, обозначающее окончание заголовка функции (аналогично с условиями и циклами). После заголовка с новой строки и с отступом следуют выражения тела функции. В конце тела функции обычно присутствует инструкция `return`, после которой идёт значение или выражение, являющееся результатом работы функции. Именно оно будет подставлено в главной (вызывающей) программе на место функции. Принято говорить, что функция «возвращает значение», в данном случае — результат вычисления выражения $x^2 + 4x + 4$ в конкретной точке x. В других языках программирования такая инструкция называется также **функцией**, а инструкция, которая ничего не возвращает, а только производит какие-то действия, называется **процедурой**[1], например:

```
def procedura(x):
    f = x**2+4*x+4
    print(f)
```

Те же самые инструкции можно переписать в виде функции `Bisection`, которой передаются данные из основной программы, и которая возвращается корень уравнения с заданной точностью:

```
from math import *
def function(x):
    f = x**2-4
```

[1]На самом деле никакого разделения на функции и процедуры в Python нет. Просто те функции, в которых возвращаемое значение не указано, возвращают специальное значение `None`. Более того, даже если функция какой-то результат возвращает, вы можете его проигнорировать и использовать её как процедуру (в этом случае она, конечно, должна делать что-то полезное кроме вычисления возвращаемого значения).

```
    return f
def Bisection(a, b, e):
    while (b-a) > e:
        c = (a+b)/2
        f_a = function(a)
        f_b = function(b)
        f_c = function(c)
        if f_a*f_c > 0:
            a = c
        else:
            b = c
    return ((a+b)/2)
A = -10
B = 10
E = 10**(-15)
print(Bisection(A, B, E))
```

Вывод программы:

```
-2.0000000000000004
```

В Python результатом функции может быть только одно значение. Если необходимо в качестве результата выдать значения сразу нескольких переменных, используют кортеж. Продемонстрируем это, дополнив программу вычислением количества шагов, за которые достигается значения корня с заданной точностью:

```
from math import *
def function(x):
    f = x**2-4
    return f
def Bisection(a, b, e):
    n = 0
    while (b-a) > e:
        n = n + 1
        c = (a+b)/2
        f_a = function(a)
        f_b = function(b)
        f_c = function(c)
        if f_a*f_c > 0:
            a = c
        else:
            b = c
    return ((a+b)/2, n)
A = -10
B = 10
E = 10**(-15)
```

```
print(Bisection(A, B, E))
```

В качестве результата выдаётся кортеж из двух чисел: значения корня и количества шагов, за которое был найден этот корень:

```
(-2.0000000000000004, 55)
```

Выражения тела функции выполняются лишь тогда, когда она вызывается в основной ветке программы. Так, например, если функция присутствует в исходном коде, но нигде не вызывается, то содержащиеся в ней инструкции не будут выполнены ни разу.

4.2 Параметры и аргументы функций

Часто функция используется для обработки данных, полученных из внешней для нее среды (из основной ветки программы). Данные передаются функции при её вызове в скобках и называются аргументами. Однако чтобы функция могла «взять» передаваемые ей данные, необходимо при её создании описать параметры (в скобках после имени функции), представляющие собой переменные.

Пример:

```
def summa(a,b):
    c = a + b
    return c
num1 = int(input('Введите первое число: '))
num2 = int(input('Введите второе число: '))
summa(num1, num2)
```

Здесь `num1` и `num2` суть *аргументы* функции, `a` и `b` суть *параметры* функции.

В качестве аргументов могут выступать как числовые или строковые константы вроде `12.3`, `-9` или `'Hello'`, так и переменные или выражения, например, `a+2*i`.

4.2.1 Обязательные и необязательные аргументы

Аргументы функции могут быть обязательными или необязательными. Для всех необязательных аргументов необходимо указать значение по умолчанию.

Рассмотрим функцию, возводящую один свой параметр в степень, заданную другим:

```
def Degree(x, a):
    f = x**a
    return f
```

Если мы попробуем вызвать эту функцию с одним аргументом вместо двух, получим ошибку:

```
»> Degree(3)
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    Degree(3)
TypeError: Degree() missing 1 required positional argument: 'a'
```

Интерпретатор недоволен тем, что параметру с именем `a` не сопоставили ни одного значения. Если мы хотим, чтобы по умолчанию функция возводила `x` в квадрат, можно переопределить её следующим образом:

```
def Degree(x, a=2):
    f = x**a
    return f
```

Тогда наш вызов с одним аргументом станет корректным:

```
>>> Degree(3)
9
```

При этом мы не теряем возможность использовать функцию `Degree` для возведения в произвольную степень:

```
>>> Degree(3, 4)
81
```

В принципе, все параметры функции могут иметь значение по умолчанию, хотя часто это лишено смысла. Параметры, значение по умолчанию для которых не задано, называются *обязательными*. Важно помнить, что обязательный параметр не может стоять после параметра, имеющего значение по умолчанию. Попытка написать функцию, не удовлетворяющую этому требованию, приведёт к синтаксической ошибке:

```
>>> def Degree(x=2, a):
    f = x**a
    return f
SyntaxError: non-default argument follows default argument
```

4.2.2 Именованные аргументы

Бывает, что у функции много или очень много параметров. Или вы забыли порядок, в котором они расположены, но помните их смысл. Тогда можно обратиться к функции, используя имена параметров как ключи. Пусть у нас есть следующая функция:

```
def Clothing (Dress, ColorDress, Shoes, ColorShoes):
    S = 'Сегодня я надену ' + ColorDress + '␣'  \
      + Dress + '␣и '+ ColorShoes + '␣' + Shoes
    return S
```

Теперь вызовем нашу функцию, с аргументами не по порядку:

```
print(Clothing(ColorDress='красное', Dress='платье',
               ColorShoes='чёрные', Shoes='туфли'))
```

Будет выведено:

```
Сегодня я надену красное платье и чёрные туфли
```

Как видим, результат получился верный, хотя аргументы перечислены не в том порядке, что при определении функции. Это происходит потому, что мы явно указали, какие параметры соответствуют каким аргументам.

Следует отметить, что часто программисты на Python путают параметры со значением по умолчанию и вызов функции с именованными аргументами. Это происходит оттого, что синтаксически они плохо различимы. Однако важно знать, что наличие значения по умолчанию не обязывает вас использовать имя параметра при обращении к нему. Также и отсутствие значения по умолчанию не означает, что к параметру нельзя обращаться по имени. Например, для описанной выше функции `Degree` все следующие вызовы будут корректными и приведут к одинаковому результату:

```
>>> Degree(3)
9
>>> Degree(3, 2)
9
>>> Degree(3, a=2)
9
>>> Degree(x=3, a=2)
9
>>> Degree(a=2, x=3)
9
```

Чего нельзя делать, так это ставить обязательные аргументы после необязательных, если имена параметров не указаны:

```
>>> Degree(a=2, 3)
SyntaxError: non-keyword arg after keyword arg
```

4.2.3 Произвольное количество аргументов

Иногда возникает ситуация, когда вы заранее не знаете, какое количество аргументов будет необходимо принять функции. Для такого случая есть специальный синтаксис: все параметры обозначаются одним именем (обычно используется имя `args`) и перед ним ставится звёздочка `*`. Например:

```
def unknown(*args):
    for argument in args:
```

```
        print (argument)
unknown('Что ', 'происходит', '?')
unknown('Не знаю!')
```

Вывод программы:

```
Что
происходит
?
Не знаю!
```

При этом тип значения `args` — кортеж, содержащий все переданные аргументы по порядку.

4.3 Локальные и глобальные переменные

Если записать в IDLE приведённую ниже функцию, и затем попробовать вывести значения переменных, то обнаружится, что некоторые из них почему-то не существуют:

```
>>> def mathem(a, b):
    a = a/2
    b = b+10
    print(a+b)
>>> num1 = 100
>>> num2 = 12
>>> mathem(num1, num2)
72.0
>>> num1
>>>100
>>> num2
12
>>> a
Traceback (most recent call last):
File "<pyshell#10>", line 1, in <module>
a
NameError: name 'a' is not defined
>>> b
Traceback (most recent call last):
File "<pyshell#11>", line 1, in <module>
b
NameError: name 'b' is not defined
>>>
```

Переменные `num1` и `num2` не изменили своих первоначальных значений. Дело в том, что в функцию передаются копии значений. Прежние значения из основной ветки программы остались связанны с их переменными.

А вот переменных a и b, оказывается, нет и в помине (ошибка `name 'b' is not defined` переводится как `"переменная b не определена"`). Эти переменные существуют лишь в момент выполнения функции и называются *локальными*. В противовес им, переменные num1 и num2 видны не только во внешней ветке, но и внутри функции:

```
>>> def mathem2():
    print(num1+num2)
>>> mathem2()
112
>>>
```

Переменные, определённые в основной ветке программы, являются *глобальными*. Итак, в Python две базовых области видимости переменных:

1. глобальные переменные,
2. локальные переменные.

Переменные, объявленные внутри тела функции, имеют локальную область видимости, те, что объявлены вне какой-либо функции, имеют глобальную область видимости. Это означает, что доступ к локальным переменным имеют только те функции, в которых они были объявлены, в то время как доступ к глобальным переменным можно получить по всей программе в любой функции.

Например:

```
Place = 'Солнечная система' # Глобальная переменная
def global_Position():
    print(Place)

def local_Position():
    Place = 'Земля' # Локальная переменная
    print(Place)

S = input()
if S == 'система':
    global_Position()
else:
    local_Position()
```

Вывод программы при двух последовательных запусках:

```
система
Солнечная система
>>>
========== RESTART: /home/paelius/Python/1.py ==========
планета
Земля
```

Важно помнить, что для того чтобы получить доступ к глобальной переменной на чтение, достаточно лишь указать её имя. Однако если перед нами стоит задача изменить глобальную переменную внутри функции, необходимо использовать ключевое слово `global`. Например:

```
Number = 10
def change():
    global Number
    Number = 20
print(Number)
change()
print(Number)
```

Вывод программы:

```
10
20
```

Если забыть написать строчку `global Number`, то интерпретатор выдаст следующее:

```
10
10
```

4.4 Программирование сверху вниз

Вряд ли стоило бы уделять много внимания функциям, если бы за ними не скрывались важные и основополагающие идеи. В действительности, функции оказывают решающее влияние на стиль и качество работы программиста. Функция — это не только способ сокращения текста, но что более важно, средство разложения программы на логически связанные, замкнутые компоненты, определяющие её структуру.

Представьте себе программу, содержащую, например, 1000 строк кода (это ещё очень маленькая программа). Обозреть такое количество строк и понять, что делает программа, было бы практически невозможно без функций.

Большие программы строятся методом последовательных уточнений. На первом этапе внимание обращено на глобальные проблемы, и в первом эскизном проекте упускаются из виду многие детали. По мере продвижения процесса создания программы глобальные задачи разбиваются на некоторое число подзадач. Те, в свою очередь, на более мелкие подзадачи и т.д., пока решать каждую подзадачу не станет достаточно просто. Такая декомпозиция и одновременная детализация программы называется *нисходящим методом* программирования или *программированием сверху вниз*.

Концепция функций позволяет выделить отдельную подзадачу как отдельную подпрограмму. Тогда на каждом этапе можно придумать имена функций для подзадач, вписывать в раздел описаний их заголовки и, ещё не добавляя к

ним тело функции, уже использовать их вызовы для создания каркаса программы так, будто они уже написаны.

Например, на численных методах студенты решают задачу сравнения двух методов поиска корня уравнения: уже расписанного выше метода деления отрезка пополам и метода Ньютона. Необходимо понять, какой из методов быстрее сходится к корню с заданной точностью.

Концепция программирования «сверху вниз» предусматривает, что вначале студенты напишут «скелет» программы, а уже потом начнут разбираться, как какая функция функционирует в отдельности:

```
def Function(X) #Наше нелинейное уравнение
def Newton(a, b, E)      #Метод Ньютона
def Bisection(a, b, E)   #Метод деления отрезка пополам

A = ?
B = ?
E = ?
Bisection(A, B, E) #Вызов функции метода деления отрезка пополам#
Newton(A, B, E) #Вызов функции метода Ньютона#
```

Какие именно инструкции будут выполняться функциями `Bisection` и `Newton` пока не сказано, но уже известно, что они принимают на вход и что должны возвращать. Это следующий этап написания программы. Потом нам известно, что наше нелинейное уравнение, корень которого необходимо найти, желательно оформить в виде отдельной функции. Так появляется функция `Function`, которая будет вызываться внутри функций `Bisection` и `Newton`.

Когда каркас создан, остаётся только написать тела функций. Преимущество такого подхода в том, что, создавая тело каждой из функций, можно не думать об остальных функциях, сосредоточившись на одной подзадаче. Кроме того, когда каждая из функций имеет понятный смысл, гораздо легче, взглянув на программу, понять, что она делает. Это в свою очередь позволит: (а) допускать меньше логических ошибок и (б) организовать совместную работу нескольких программистов над большой программой.

Важной идеей при таком подходе становится использование локальных переменных. В глобальную переменную можно было бы записать результат работы функции, однако это будет стилистической ошибкой. Весь обмен информацией функция должна вести исключительно через параметры. Такой подход позволяет сделать решение каждой подзадачи максимально независимым от остальных подзадач, упрощает и упорядочивает структуру программы.

4.5 Рекурсивный вызов функции

Рекурсия в программировании — это вызов функции из неё же самой непосредственно (простая рекурсия) или через другие функции (сложная или косвенная рекурсия), например, функция `A` вызывает функцию `B`, а функция `B` —

функцию A. Количество вложенных вызовов функции или процедуры называется глубиной рекурсии. Рекурсивная программа позволяет описать повторяющееся или даже потенциально бесконечное вычисление, причём без явных повторений частей программы и использования циклов.

Принцип работы рекурсивной функции удобнее всего объяснять на примере матрёшек (рис. 4.2). Пусть есть набор нераскрашенных матрёшек. Нужно узнать, в какие цвета покрасить самую большую матрёшку. При этом раскрашена только самая маленькая матрёшка, которая не раскрывается. Необходимо последовательно раскрывать матрёшки, пока не дойдём до той, которую раскрыть не получается — это раскрашенная матрёшка. Далее можно раскрашивать каждую следующую матрёшку по возрастанию, держа перед собою предыдущую как пример и затем вкладывать все меньшие, раскрашенные ранее, во вновь раскрашенную.

От матрёшек можно перейти к классическому примеру рекурсии, которым может послужить функция вычисления факториала числа:

```
def fact(num):
    if num == 0:
        return 1
    else:
        return num * fact(num - 1)
n = int(input('Введите число '))
print(fact(n))
```

Структурно рекурсивная функция на верхнем уровне всегда представляет собой команду ветвления (выбор одной из двух или более альтернатив в зависимости от условия (условий), которое в данном случае уместно назвать «условием прекращения рекурсии», имеющей две или более альтернативные ветви, из которых хотя бы одна является рекурсивной и хотя бы одна — *терминальной*.

В вышеприведённых примерах с матрёшками и с вычислением факториала условия «если матрёшка не раскрывается» и `if num==0` являются командами ветвления, ветвь `return 1` (или самая маленькая раскрашенная матрёшка) является терминальной, а ветвь `return num*fact(num-1)` (постепенное раскрашивание и закрывание матрёшек) — *рекурсивной*.

Рекурсивная ветвь выполняется, когда условие прекращения рекурсии ложно, и содержит хотя бы один рекурсивный вызов — прямой или опосредованный вызов функцией самой себя. Терминальная ветвь выполняется, когда условие прекращения рекурсии истинно; она возвращает некоторое значение, не выполняя рекурсивного вызова. Правильно написанная рекурсивная функция должна гарантировать, что через конечное число рекурсивных вызовов будет достигнуто выполнение условия прекращения рекурсии, в результате чего цепочка последовательных рекурсивных вызовов прервётся и выполнится возврат.

Помимо функций, выполняющих один рекурсивный вызов в каждой рекурсивной ветви, бывают случаи «параллельной рекурсии», когда на одной рекурсивной ветви делается два или более рекурсивных вызова. Параллельная ре-

Рис. 4.2. Наглядное представление принципа работы рекурсивной функции.

курсия типична при обработке сложных структур данных, таких как деревья. Простейший пример параллельно-рекурсивной функции – вычисление ряда Фибоначчи, где для получения значения n-го члена необходимо вычислить $(n-1)$-й и $(n-2)$-й:

```
def fib(n):
    if n<3:
        return 1
    return fib(n-1) + fib(n-2)

n = int(input('Введите число: '))
print(fib(10))
```

Реализация рекурсивных вызовов функций в практически применяемых языках и средах программирования, как правило, опирается на механизм стека вызовов — адрес возврата и локальные переменные функции записываются в стек, благодаря чему каждый следующий рекурсивный вызов этой функции пользуется своим набором локальных переменных и за счёт этого работает корректно. Оборотной стороной этого довольно простого по структуре механизма является то, что на каждый рекурсивный вызов требуется некоторое количество оперативной памяти компьютера, и при чрезмерно большой глубине рекурсии может наступить переполнение стека вызовов.

Вопрос о желательности использования рекурсивных функций в программировании неоднозначен: с одной стороны, рекурсивная форма может быть структурно проще и нагляднее, в особенности, когда сам реализуемый алгоритм, по сути, рекурсивен. Кроме того, в некоторых декларативных или чисто функциональных языках (таких, как Пролог или Haskell) просто нет синтаксических средств для организации циклов, и рекурсия в них — единственный доступный механизм организации повторяющихся вычислений. С другой стороны, обычно рекомендуется избегать рекурсивных программ, которые приводят (или в некоторых условиях могут приводить) к слишком большой глубине рекурсии. Так, широко распространённый в учебной литературе пример рекурсивного вычисления факториала является, скорее, примером того, как не надо применять рекурсию, так как приводит к достаточно большой глубине рекурсии и имеет очевидную реализацию в виде обычного циклического алгоритма.

4.6 Примеры решения заданий

Пример задачи 12 *Напишите функцию, вычисляющую значения экспоненты по рекуррентной формуле* $e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots = \sum_{n=0}^{\infty} \frac{x^n}{n!}$. *Реализуйте контроль точности вычислений с помощью дополнительного параметра* ε *со значением по умолчанию (следует остановить вычисления, когда очередное приближение будет отличаться от предыдущего менее, чем на* 10^{-10}*).*

Реализуйте вызов функции различными способами:

- с одним позиционным параметром (при этом будет использовано значение по умолчанию);
- с двумя позиционными параметрами (значение точности будет передано как второй аргумент);
- передав значение как именованный параметр.

Решение задачи 12

```
def EXPONENTA(x, eps=10**(-10)):
    ex = 1 # будущий результат
    dx = x # приращение
    i = 2  # номер приращения
    while abs(dx)>eps:
        ex = ex + dx
        dx = dx * x / i
        i = i + 1
    return ex
#Основная программа
A = float(input('Введите показатель экспоненты: '))
print(EXPONENTA(A))
print(EXPONENTA(A, 10**(-4)))
print(EXPONENTA(x = A)
```

Пример задачи 13 Сделайте из функции процедуру (вместо того, чтобы вернуть результат с помощью оператора `return`, выведите его внутри функции с помощью функции `print`).

Решение задачи 13

```
def EXPONENTA(x, eps=10**(-10)):
    ex = 1 # будущий результат
    dx = x # приращение
    i = 2  # номер приращения
    while abs(dx)>eps:
        ex = ex + dx
        dx = dx * x / i
        i = i + 1
    print(ex)
#Основная программа
A = float(input('Введите показатель экспоненты: '))
EXPONENTA(A)
```

4.7 Задания на функции

Задание 11 Выполнять одно задание в зависимости от номера в списке: `(n-1)%10+1`, где `n` — номер в списке. Напишите функцию, вычисляющую значения одной из следующих специальных функций по рекуррентной формуле. Реализуйте контроль точности вычислений с помощью дополнительного параметра ε со значением по умолчанию (следует остановить вычисления, когда очередное приближение будет отличаться от предыдущего менее, чем на 10^{-10}). Реализуйте вызов функции различными способами:

- с одним позиционным параметром (при этом будет использовано значение по умолчанию);
- с двумя позиционными параметрами (значение точности будет передано как второй аргумент);
- передав значение как именованный параметр.

Сделайте из функции процедуру (вместо того, чтобы вернуть результат с помощью оператора `return`, выведите его внутри функции с помощью функции `print`).

1. Косинус $\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \cdots = \sum_{n=0}^{\infty}(-1)^n \frac{x^{2n}}{(2n)!}$. Формула хорошо работает для $-2\pi \leqslant x \leqslant 2\pi$, поскольку получена разложением в ряд Тейлора возле ноля. Для прочих значений x следует воспользоваться свойствами периодичности косинуса: $\cos(x) = \cos(2 + 2\pi n)$, где n есть любое целое число, тогда `cos(x) = cos(x%(2*math.pi))`. Для проверки использовать функцию `math.cos(x)`.

2. Синус $\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} + \cdots = \sum_{n=0}^{\infty}(-1)^n \frac{x^{2n+1}}{(2n+1)!}$. Формула хорошо работает для $-2\pi \leqslant x \leqslant 2\pi$, поскольку получена разложением в ряд Тейлора возле ноля. Для прочих значений x следует воспользоваться свойствами периодичности косинуса: $\sin(x) = \sin(2 + 2\pi n)$, где n есть любое целое число, тогда `sin(x) = sin(x%(2*math.pi))`. Для проверки использовать функцию `math.sin(x)`.

3. Гиперболический косинус $\mathrm{ch}(x) = 1 + \frac{x^2}{2!} + \frac{x^4}{4!} + \cdots = \sum_{n=0}^{\infty} \frac{x^{2n}}{(2n)!}$. Для проверки использовать функцию `math.cosh(x)`.

4. Гиперболический косинус по формуле для экспоненты, оставляя только слагаемые с чётными n. Для проверки использовать функцию `math.cosh(x)`.

5. Гиперболический синус $\mathrm{sh}(x) = x + \frac{x^3}{3!} + \frac{x^5}{5!} + \cdots = \sum_{n=0}^{\infty} \frac{x^{2n+1}}{(2n+1)!}$. Для проверки использовать функцию `math.sinh(x)`.

6. Гиперболический синус по формуле для экспоненты, оставляя только слагаемые с нечётными n. Для проверки использовать функцию `math.sinh(x)`.

7. Натуральный логарифм (формула работает при $0 < x \leqslant 2$):

$$\ln(x) = (x-1) - \frac{(x-1)^2}{2} + \frac{(x-1)^3}{3} - \frac{(x-1)^4}{4} + \cdots = \sum_{n=1}^{\infty} (-1)^{n-1} \frac{(x-1)^n}{n}$$

   Чтобы найти логарифм для $x > 2$, необходимо представить его в виде $\ln(x) = \ln(y \cdot 2^p) = p\ln(2) + \ln(y)$, где $y < 2$, а p натуральное число. Чтобы найти p и y, нужно в цикле делить x на 2 до тех пор, пока результат больше 2. Когда очередной результат деления станет меньше 2, этот результат и есть y, а число делений, за которое он достигнут – это p. Для проверки использовать функцию `math.log(x)`.

8. Гамма-функция $\Gamma(x)$ по формуле Эйлера:

$$\Gamma(x) = \frac{1}{x} \prod_{n=1}^{\infty} \frac{\left(1 + \frac{1}{n}\right)^x}{1 + \frac{x}{n}}$$

   Формула справедлива для $x \notin \{0, -1, -2, \dots\}$. Для проверки можно использовать `math.gamma(x)`. Также, поскольку $\Gamma(x+1) = x!$ для натуральных x, то для проверки можно использовать функцию `math.factorial(x)`.

9. Функция ошибок, также известная как интеграл ошибок, интеграл вероятности, или функция Лапласа:

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt = \frac{2}{\sqrt{\pi}} \sum_{n=0}^{\infty} \frac{x}{2n+1} \prod_{i=1}^{n} \frac{-x^2}{i}$$

   Для проверки использовать функцию `scipy.special.erf(x)`.

10. Дополнительная функция ошибок:

$$\operatorname{erfc}(x) = 1 - \operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_x^{\infty} e^{-t^2} dt = \frac{e^{-x^2}}{x\sqrt{\pi}} \sum_{n=0}^{\infty} (-1)^n \frac{(2n)!}{n!(2x)^{2n}}$$

    Для проверки использовать функцию `scipy.special.erf(x)`.

Задание 12 (Танцы) Выполнять в каждом разделе по одному заданию в зависимости от номера в списке группы: `(n-1)%5+1`, где n — номер в списке.

1. два списка имён: первый — мальчиков, второй — девочек;

2. один список имён и число мальчиков, все имена мальчиков стоят в начале списка;

3. один список имён и число девочек, все имена девочек стоят в начале списка;
4. словарь, в котором в качестве ключа используется имя, а в качестве значения символ «м» для мальчиков и символ «ж» для девочек;
5. словарь, в котором в качестве ключей выступают символы «м» и «ж», а в качестве соответствующих им значений — списки мальчиков и девочек соответственно.

Проверьте работу функции на разных примерах:

- когда мальчиков и девочек поровну,
- когда мальчиков больше, чем девочек, и наоборот,
- когда есть ровно 1 мальчик или ровно 1 девочка,
- когда либо мальчиков, либо девочек нет вовсе.

Модифицируйте функцию так, чтобы она принимала второй необязательный параметр — список уже составленных пар, участников которых для составления пар использовать нельзя. В качестве значения по умолчанию для этого аргумента используйте пустой список. Проверьте работу функции, обратившись к ней:

- как и ранее (с одним аргументом), в этом случае результат должен совпасть с ранее полученным;
- передав все аргументы позиционно без имён;
- передав последний аргумент (список уже составленных пар) по имени;
- передав все аргументы по имени в произвольном порядке.

Задание 13 (Создание списков) Напишите функцию, принимающую от 1 до 3 параметров — целых чисел (как стандартная функция `range`). Единственный обязательный аргумент — последнее число. Если поданы 2 аргумента, то первый интерпретируется как начальное число, второй — как конечное (не включительно). Если поданы 3 аргумента, то третий аргумент интерпретируется как шаг. Функция должна выдавать один из следующих списков:

1. квадратов чисел;
2. кубов чисел;
3. квадратных корней чисел;

4. логарифмов чисел;

5. чисел последовательности Фибоначчи с номерами в указанных пределах.

Запускайте вашу функцию со всеми возможными вариантами по числу параметров: от 1 до 3.

Подсказка: проблему переменного числа параметров, из которых необязательным является в том числе первый, можно решить 2-мя способами. Во-первых, можно сопоставить всем параметрам нечисловые значения по умолчанию, обычно для этого используют специальное значение `None`. Тогда используя условный оператор можно определить, сколько параметров реально заданы (не равны `None`). В зависимости от этого следует интерпретировать значение первого аргумента как: конец последовательности, если зада только 1 параметр; как начало, если заданы 2 или 3. Во-вторых, можно проделать то же, используя синтаксис функции с произвольным числом параметров; в таком случае задавать значения по умолчанию не нужно, а полезно использовать стандартную функцию `len`, которая выдаст количество реально используемых параметров.

Задание 14 (Интернет-магазин) Решите задачу об интернет-торговле. Несколько покупателей в течении года делали покупки в интернет-магазине. При каждой покупке фиксировались имя покупателя (строка) и потраченная сумма (действительное число). Напишите функцию, рассчитывающую для каждого покупателя и выдающую в виде словаря по всем покупателям (вида имя:значение) один из следующих параметров:

1. число покупок;

2. среднюю сумму покупки;

3. максимальную сумму покупки;

4. минимальную сумму покупки;

5. общую сумму всех покупок.

На вход функции передаётся:

- либо 2 списка, в первом из которых имена покупателей (могут повторяться), во втором – суммы покупок;

- либо 1 список, состоящий из пар вида `(имя, сумма)`;

- либо словарь, в котором в качестве ключей используются имена, а в качестве значений — списки с суммами.

Глава 5

Массивы. Модуль numpy

Сам по себе «чистый» Python пригоден только для несложных вычислений. Ключевая особенность Python — его расширяемость. Это, пожалуй, самый расширяемый язык из получивших широкое распространение. Как следствие этого для Python не только написаны и приспособлены многочисленные библиотеки алгоритмов на C и Fortran, но и имеются возможности использования других программных средств и математических пакетов, в частности, R и SciLab, а также графопостроителей, например, Gnuplot и PLPlot.

Ключевыми модулями для превращения Python в математический пакет являются `numpy` и `matplotlib`.

`numpy` — это модуль (в действительности, набор модулей) языка Python, добавляющая поддержку больших многомерных массивов и матриц, вместе с большим набором высокоуровневых (и очень быстрых) математических функций для операций с этими массивами.

`matplotlib` — модуль (в действительности, набор модулей) на языке программирования Python для визуализации данных двумерной (2D) графикой (3D графика также поддерживается). Получаемые изображения могут быть использованы в качестве иллюстраций в публикациях.

Кроме `numpy` и `matplotlib` популярностью пользуются `scipy` для специализированных математических вычислений (поиск минимума и корней функции многих переменных, аппроксимация сплайнами, вейвлет-преобразования), `sympy` для символьных вычислений (аналитическое взятие интегралов, упрощение математических выражений), `ffnet` для построения искусственных нейронных сетей, `pyopencl`/`pycuda` для вычисления на видеокартах и некоторые другие. Возможности `numpy` и `scipy` покрывают практически все потребности в математических алгоритмах.

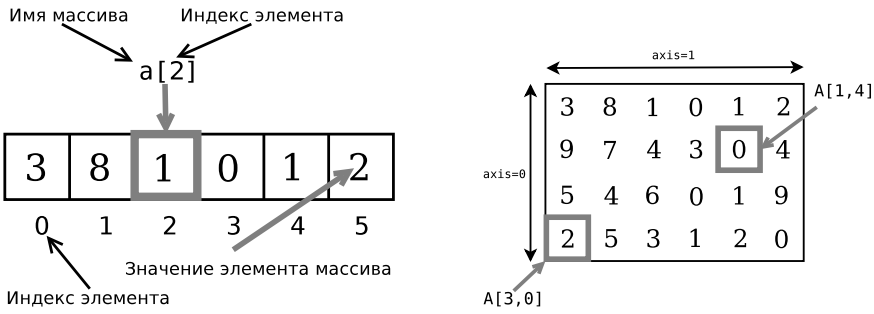


Рис. 5.1. Одномерный — (a) и двумерный — (b) массивы.

5.1 Создание и индексация массивов

Массив — упорядоченный набор значений одного типа, расположенных в памяти непосредственно друг за другом. При этом доступ к отдельным элементам массива осуществляется с помощью индексации, то есть через ссылку на массив с указанием номера (индекса) нужного элемента. Это возможно потому, что все значения имеют один и тот же тип, занимая одно и то же количество байт памяти; таким образом, зная ссылку и номер элемента, можно вычислить, где он расположен в памяти. Количество используемых индексов массива может быть различным: массивы с одним индексом называют одномерными, с двумя — двумерными, и т. д. Одномерный массив («колонка», «столбец») примерно соответствует вектору в математике (на рис. 5.1(a) `a[4] == 56`, т.е. четвёртый элемент массива a равен 56); двумерный — матрице (на рис. 5.1(b) можно писать `A[1][6] == 22`, можно `A[1, 6] == 22`). Чаще всего применяются массивы с одним или двумя индексами; реже — с тремя; ещё большее количество индексов встречается крайне редко.

Как правило, в программировании массивы делятся на *статические* и *динамические*. *Статический массив* — массив, размер которого определяется на момент компиляции программы. В языках с динамической типизацией таких, как Python, они не применяются. *Динамический массив* — массив, размер которого задаётся во время работы программы. То есть при запуске программы этот массив не занимает нисколько памяти компьютера (может быть, за исключением памяти, необходимой для хранения ссылки). Динамические массивы могут поддерживать и не поддерживать изменение размера в процессе исполнения программы. Массивы в Python не поддерживают явное изменение размера: у них, в отличие от списков, нет методов `append` и `extend`, позволяющих добавлять элементы, и методов `pop` и `remove`, позволяющих их удалять. Если нужно изменить размер массива, это можно сделать путём переприсваивания имени переменной,

обозначающей массив, нового значения, соответствующего новой области памяти, больше или меньше прежней разными способами.

Базовый оператор создания массива называется `array`. С его помощью можно создать новый массив с нуля или превратить в массив уже существующий список. Вот пример:

```
from numpy import *
A = array([0.1, 0.4, 3, -11.2, 9])
print(A)
print(type(A))
```

В первой строке из модуля `numpy` импортируются все функции и константы. Вывод программы следующий:

```
[0.1   0.4   3.  -11.2   9. ]
<type 'numpy.ndarray'>
```

Функция `array()` трансформирует вложенные последовательности в многомерные массивы. Тип элементов массива зависит от типа элементов исходной последовательности:

```
B = array([[1, 2, 3], [4, 5, 6]])
print(B)
```

Вывод:

```
[[1 2 3]
 [4 5 6]]
```

Тип элементов массива можно определить в момент создания с помощью именованного аргумента `dtype`. Модуль `numpy` предоставляет выбор из следующих встроенных типов: `bool` (логическое значение), `character` (символ), `int8`, `int16`, `int32`, `int64` (знаковые целые числа размеров в 8, 16, 32 и 64 бита соответственно), `uint8`, `uint16`, `uint32`, `uint64` (беззнаковые целые числа размеров в 8, 16, 32 и 64 бита соответственно), `float32` и `float64` (действительные числа одинарной и двойной точности), `complex64` и `complex128` (комплексные числа одинарной и двойной точности), а также возможность определить собственные типы данных, в том числе и составные.

```
C = array([[1, 2, 3], [4, 5, 6]], dtype = float)
print(C)
```

Вывод:

```
[[ 1.  2.  3.]
 [ 4.  5.  6.]]
```

Можно создать массив из диапазона:

```
from numpy import *
L = range(5)
A = array(L)
print(A)
```

Вывод:

```
[0 1 2 3 4]
```

Отсюда видно, что в первом случае массив состоит из действительных чисел, так как при определении часть его значений задана в виде десятичных дробей. В `numpy` есть несколько действительнозначных типов, базовый тип соответствует действительным числам Python и называется `float64`. Второй массив состоит из целых чисел, потому что создан из диапазона `range`, в который входят только целые числа. На 64-битных операционных системах элементы такого списка будут автоматически приведены к целому типу `int64`, на 32-битных — к типу `int32`.

В `numpy` есть функция `arange`, позволяющая сразу создать массив-диапазон, причём можно использовать дробный шаг. Вот программа, рассчитывающая значения синуса на одном периоде с шагом $\pi/6$:

```
from numpy import *
a1 = arange(0, 2*pi, pi/6)
s1 = sin(a1)
for j in range(len(a1)):
    print(a1[j], s1[j])
```

А вот её вывод:

```
0.0 0.0
0.523598775598 0.5
1.0471975512 0.866025403784
1.57079632679 1.0
2.09439510239 0.866025403784
2.61799387799 0.5
3.14159265359 1.22464679915e-16
3.66519142919 -0.5
4.18879020479 -0.866025403784
4.71238898038 -1.0
5.23598775598 -0.866025403784
5.75958653158 -0.5
```

Кроме `arange` есть ещё функция `linspace`, тоже создающая массив-диапазон. Её первые два аргумента те же, что и у `arange`: начало и конец диапазона, а третий аргумент — количество элементов в диапазоне. К сожалению, по умолчанию эти функции по разному обрабатывают второй аргумент: `arange` берёт его невключительно, `linspace` — включительно. Это можно поправить с помощью

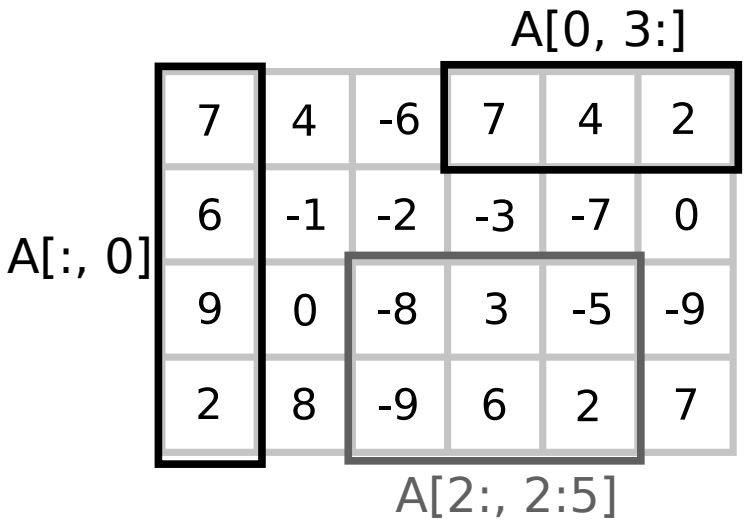


Рис. 5.2. Примеры срезов массивов: чёрным — срезы, в результате которых из двумерного массива получаются одномерные (из столбца и строки исходного массива соответственно); тёмносерым — срез, в результате которого получается двумерный массив меньшего размера.

опционного, четвёртого аргумента `linspace`, если установить его в `False`. Вот пример задания одинаковых массивов с помощью `arange` и `linspace`:

```
>>> a = numpy.arange(0, 2, 0.25)
>>> print(a)
[ 0.    0.25  0.5   0.75  1.    1.25  1.5   1.75]
>>> b = numpy.linspace(0, 2, 8, False)
>>> print(a)
[ 0.    0.25  0.5   0.75  1.    1.25  1.5   1.75]
```

Массивы построены на базе списков и сохраняют некоторые их черты. В частности, массивы можно обходить циклом любым из изложенных выше способов, а функция `len` возвращает длину массива. Важным свойством массивов является то, что над ними можно производить операции как над простыми числами, при этом операция, в нашем случае вычисление синуса, будет произведена с каждым элементом массива.

Модуль `numpy` дублирует и расширяет функционал модуля `math`, он содержит все основные тригонометрические, логарифмические и прочие функции, а также константы, поэтому при работе с `numpy` надобности в `math` обычно не возникает.

Одна из важнейших особенностей массивов — возможность делать срезы (см. рис. 5.2). Срез представляет собою массив из части элементов исходного массива,

взятых по некоторому простому правилу, например, каждый второй или первые десять. Вот небольшая программа, иллюстрирующая создание срезов массивов:

```
from numpy import *
a1 = arange(0, 4, 0.5)
print(a1)
a2 = a1[0:4]
print(a2)
a3 = a1[0:len(a1):2]
print(a3)
```

Массив a2 состоит из первых четырёх (от нулевого включительно до четвёртого невключительно) элементов массива a1, массив a3 — из всех чётных элементов a1 (элементы от нулевого до последнего с шагом 2). Для обозначения индексации используются двоеточия, сначала ставится номер начального индекса, потом конечного, потом шаг, все они — целые числа (точно в том же порядке, как и в функциях range и arange). Если шаг и стоящее перед ним двоеточие опустить, шаг будет автоматически единичным. Вот вывод программы:

```
[ 0.   0.5  1.   1.5  2.   2.5  3.   3.5]
[ 0.   0.5  1.   1.5]
[ 0.  1.  2.  3.]
```

Нужно отметить, что массив-срез делит с исходным массивом память. Это значит, что копирование элементов в новый массив не происходит, вместо этого просто делается сложная ссылка. Поэтому изменение элементов исходного массива приводит к изменению элементов массива-среза и наоборот. Чтобы скопировать элементы, нужно воспользоваться функцией copy. Здесь у внимательных читателей должен возникнуть эффект déjà vu, потому что нечто подобное уже было в разделе про списки.

```
from numpy import *
a1 = arange(0, 0.6, 0.1)
a2 = a1[:len(a1)/2]
print(a1, a2)
a1[0] = 10
a1[-1] = a1[-1] + 3
print(a1, a2)
a2[1] = 0
print(a1, a2)
```

Вывод программы:

```
[0.  0.1  0.2  0.3  0.4  0.5] [ 0.  0.1  0.2]
[10.  0.1  0.2  0.3  0.4  3.5] [10.  0.1  0.2]
[10.  0.  0.2  0.3  0.4  3.5] [10.  0.  0.2]
```

В приведённом примере мы поменяли нулевой и последний элементы массива a1, в результате в массиве a2 нулевой элемент тоже изменился, затем мы поменяли первый элемент a2, и первый элемент a1 также изменился. Теперь тот же пример, но с использованием функции copy:

```
from numpy import *
a1 = arange(0, 0.6, 0.1)
a2 = copy(a1[:len(a1)/2])
print(a1, a2)
a1[0] = 10
a1[-1] = a1[-1] + 3
print(a1, a2)
a2[1] = 0
print(a1, a2)
```

Теперь в выводе видно, что изменение одного из массивов не приводит к изменению другого:

```
[0.  0.1  0.2  0.3  0.4  0.5] [0.  0.1  0.2]
[10.  0.1  0.2  0.3  0.4  3.5] [0.  0.1  0.2]
[10.  0.1  0.2  0.3  0.4  3.5] [0.  0.  0.2]
```

В приведённых примерах было использовано ещё одно свойство индексации: если номер начального элемента опустить, то автоматически подставляется ноль, если опустить номер конечного — длина массива.

Функция array() не единственная функция для создания массивов. Обычно элементы массива вначале неизвестны, а массив, в котором они будут храниться, уже нужен. Поэтому имеется несколько функций для того, чтобы создавать массивы с каким-то исходным содержимым (по умолчанию тип элементов создаваемого массива — float64): ones создаёт массив из единиц, zeros — массив из нулей, empty — массив, значения элементов которого могут быть какими угодно (берётся то, что лежало в ячейках памяти до того, как в них был размещён массив). Все эти функции принимают единственный параметр — длину вновь создаваемого массива. Вот пример их работы:

```
from numpy import *
a1 = ones(5)
a2 = zeros(3)
a3 = empty(4)
print(a1)
print(a2)
print(a3)
```

Вывод:

```
[1.  1.  1.  1.  1.]
[0.  0.  0.]
[4.39718425e-322  0.00000000e+000  0.00000000e+000  2.84236875e-316]
```

Видно, что в a3 помещены 4 произвольных числа. В нашем случае они получились равными или близкими к нулю, но рассчитывать на это в общем случае не приходится.

Можно вручную задать тип данных:

```
a3 = empty(5, dtype=int)
print(a3)
```

Вывод:

```
[500   0   0   0   0]
```

В данном примере как раз хорошо видно, что нулевой элемент массива далеко не равен нулю.

Если массив слишком большой, чтобы его печатать, numpy автоматически скрывает центральную часть массива и выводит только его уголки:

```
A = arange(0, 3000, 1)
print(A)
```

Вывод:

```
[   0    1    2 ..., 2997 2998 2999]
```

Если вам действительно нужно увидеть весь массив, используйте функцию numpy.set_printoptions:

```
A = arange(0, 3000, 1)
set_printoptions(threshold=nan)
print(A)
```

И вообще, с помощью этой функции можно настроить печать массивов «под себя». Функция numpy.set_printoptions принимает несколько аргументов? описание которых при необходимости можно с лёгкостью найти в официальной документации к numpy в интернете.

Действительные числа из numpy типа float64 обладают практически всеми теми же свойствами, что и встроенные числа Python, но могут принимать три дополнительных специальных значения: inf, -inf или nan, что можно перевести как бесконечность, минус бесконечность и неопределённость. Если поделить действительное число типа numpy.float64 на 0, получится inf -inf в зависимости от его знака, а вместо ошибки будет выдано только предупреждение, которое позволит продолжить вычисления. Программа:

```
a = np.array([2., 0., 5.])
for el in a:
    print(1./el)
```

выдаст:

```
0.5
/usr/bin/ipython3:2: RuntimeWarning: divide by zero encountered in
double_scalars
inf
0.2
```

5.2 Арифметические операции и функции с массивами

Python — объектно-ориентированный язык программирования. Каждая переменная Python — объект, хотя часто это совсем незаметно. Поэтому многие переменные Python имеют встроенные методы — присущие им функции — и свойства. Так, массивы имеют ряд очень простых, но полезных методов, сильно упрощающих жизнь при написании сложных программ. Собственно, именно поэтому в сложных программах удобнее использовать именно массивы, а не списки. Самые часто используемые занесены в таблицу 5.1. Пусть у нас есть массив вида `a=numpy.array([0.1, 0.25, -1.75, 0.4, -0.9])`:

Таблица 5.1. Некоторые методы массивов

Метод	**Описание**
`a.sum()`	Сумма элементов массива: $\sum_{i=0}^{\text{len}(a)} a_i = 0.1 + 0.25 - 1.75 + 0.4 - 0.9 = -1.9$
`a.mean()`	Среднее элементов массива: $\overline{a} = \frac{1}{\text{len}(a)} \sum_{i=0}^{\text{len}(a)} a_i = (0.1 + 0.25 - 1.75 + 0.4 - 0.9)/5 = -0.38$
`a.min()`	Минимальный элемент массива: -1.75
`a.min()`	Максимальный элемент массива: 0.4
`a.var()`	Дисперсия элементов массива: $\sigma_a^2 = \frac{1}{\text{len}(a)} \sum_{i=0}^{\text{len}(a)} (a_i - \overline{a}) = ((0.1 + 0.38)^2 + (0.25 + 0.38)^2(-1.78 + 0.38)^2 + (0.4 + 0.38)^2 + (-0.9 + 0.38)^2)/5 = 0.6766$
`a.std()`	Среднеквадратичное отклонение элементов массива от среднего: $\sigma_a = 0.822556988907$

Вот небольшая программа, показывающая, как эти функции работают:

```
from numpy import *
a = array([0.1, 0.25, -1.75, 0.4, -0.9])
print(a.sum())
print(a.mean())
print(a.min())
```

```
print(a.max())
print(a.var())
print(a.std())
```

Вывод программы:

```
-1.9
-0.38
-1.75
0.4
0.6766
0.822556988907
```

Обратите внимание на круглые скобки после имени каждой функции — они указывают, что вызывается и исполняется соответствующая функция. Если скобки забыть, никаких вычислений не будет произведено, а вместо их результатов будут напечатаны строки документации соответствующих функций. То есть `a.min()` — это результат вычисления, минимальный элемент массива `a`, но `a.min` — это сам метод вычисления:

```
print(a.min)
```

Вывод:

```
<built-in method min of numpy.ndarray object at 0x00000000036B7710>
```

Поскольку Python язык очень глубоко объектный, сами по себе методы, как и их параметры и результаты, тоже являются объектами. Тип результата встроенных функций определяется их природою: `min`, `max` и `sum` всегда того же типа, что и элементы массива, а вот `var`, `mean` и `std` всегда действительнозначные, поскольку для их вычисления выполняются деление и — для `std` — извлечение квадратного корня.

Библиотека `numpy` унаследовала от Fortran ряд особенностей работы с массивами, не присущих C, Pascal, Java, C# и прочим распространённым языкам общего назначения: массивы `numpy` можно складывать, как простые числа, прибавлять к ним число, умножать и делить друг на друга и на число. При этом все операции происходят поэлементно. Вот пример программы сложения двух массивов и умножения массива на число:

```
from numpy import *
a = array([0.1, 0.25, -1.75, 0.4, -0.9])
b = array([0.9, 1.75, -0.15, 0.4, 0.4])
c = a + b
print (c)
d = a + 1.5
print(d)
```

Программа выведет:

```
[ 1.   2.  -1.9  0.8 -0.5]
[ 1.6   1.75 -0.25  1.9   0.6 ]
```

Такие операции называются иногда «векторными» в том смысле, что одинаковые действия производятся сразу с большим набором данных. Складывать можно только массивы одинаковой длины. Аналогично можно перемножать массивы. Число можно прибавлять к любому массиву или умножать массив на число, в результате получается массив того же размера, что и исходный, а каждый его элемент есть результат сложения (или умножения) соответствующего элемента исходного массива и числа. Такой синтаксис операций позволяет существенно упростить запись сложных программ, избежав множества циклов, которые пришлось бы писать на C или Pascal.

В векторных операциях можно использовать как целые массивы, так и их срезы:

```
from numpy import *
a = array([0.1, 0.25, -1.75, 0.4, -0.9])
b = array([0.9, 1.75, -0.15, 0.4, 0.4])
c = a[:3] + b[1:4]
print (c)
d = a[2:4]*a[0:2]
print (d)
```

Вот вывод программы:

```
[ 1.85  0.1  -1.35]
[-0.175  0.1  ]
```

Обратите внимание ещё раз: при перемножении массивов действия производятся поэлементно, в результате чего получается новый массив той же длины, а вовсе не их векторное произведение, как это принято в некоторых математических пакетах. Вот пример:

```
>>> import numpy
>>> numpy.array([-1.75, 0.4]) * numpy.array([0.1, 0.25])
array([-0.175,  0.1  ])
```

в то время как в результате скалярного произведения должно было получиться: `-1.75*0.1 + 0.4*0.25 = -0.075`.

Кроме арифметических операций над массивами можно также векторно выполнять все заложенные в `numpy` функции, например, взять синус или логарифм от массива (если написать просто `log(X)`, то получится натуральный логарифм, иначе нужно написать основание логарифма вторым аргументом `log(X, 10)`). Вот пример такой программы:

```
from numpy import *
t = arange(0, pi, pi/6)
x = sin(t)
```

```
y = log(t)
print(t)
print(x)
print(y)
```

Вывод:

```
[0.  0.52359878  1.04719755  1.57079633  2.0943951  2.61799388]
[0.  0.5  0.8660254  1.  0.8660254  0.5]
[-inf  -0.64702958
0.0461176  0.45158271  0.73926478  0.96240833]
```

Здесь мы протабулировали (вычислили значения функций при изменении аргумента в некоторых пределах) две функции: $\sin(t)$ и $\ln(t)$ на полуинтервале $[0; \pi)$ с шагом $\pi/6$. Как видим, можно проделывать над массивами сколь угодно сложные векторные операции. Так как натуральный логарифм нуля равен минус бесконечности, то получился соответствующий ответ: `-inf`.

Массивы `numpy` можно обходить циклом, как списки. Чтобы сделать наше табулирование более удобным и привычным для чтения, модифицируем предыдущую программу, чтобы она выдавала результат своей работы в три столбца. Для этого заменим последнюю строку, где стоят операторы `print`, на две новые:

```
for i in range(len(t)):
    print(t[i], x[i], y[i])
```

Вывод программы:

```
0.0 0.0 -inf
0.523598775598 0.5 -0.647029583379
1.0471975512 0.866025403784 0.0461175971813
1.57079632679 1.0 0.451582705289
2.09439510239 0.866025403784 0.739264777741
2.61799387799 0.5 0.962408329055
```

Полученные столбцы чисел выглядят понятно, но не очень презентабельно, поскольку каждая запись выглядит по-своему — все они имеют разное число знаков после точки. Для получения стройных колонок можно воспользоваться встроенным методом форматирования, поменяв последнюю строку приведённой программы на следующую:

```
print("{:12.8f}\t{:12.8f}\t{:12.8f}".format(t[i],
      x[i], y[i]))
```

Здесь символ `'\t'` означает табуляцию, впереди идёт строка форматирования, где `:12.8f` означает, что на это место будет подставлено действительное число (на это указывает `f`) с 12 знаками, из них после десятичной точки 8. Три подставляемых числа взяты в скобки. Вывод программы:

```
 0.00000000   0.00000000        -inf
 0.52359878   0.50000000  -0.64702958
 1.04719755   0.86602540   0.04611760
 1.57079633   1.00000000   0.45158271
 2.09439510   0.86602540   0.73926478
 2.61799388   0.50000000   0.96240833
```

В Python есть специальные операторы +=, -=, которые увеличивают и уменьшают одно число на другое, а также операторы *= и /=, которые умножают и делят одно число на другое, например:

```
>>> a = 10
>>> a += 4
>>> print(a)
14
>>> a -= 8
>>> print(a)
6
```

По сути эти инструкции аналогичны инструкциям типа a = a + b или a = a - b, где b — число, на которое нужно увеличить или уменьшить a. Преимущество использования операторов +=, -= кроме краткости записи состоит в том, что при их использовании новый объект не создаётся, а только модифицируется исходный, в то время как запись a = a + b или a = a - b приводит к уничтожению старого объекта с именем a (его должен будет удалить из памяти сборщик мусора) и созданию нового с тем же именем. Поэтому использование операторов +=, -= наиболее актуально при работе со сложными типами вроде массивов numpy, так как на их создание и размещение в памяти, а также удаление тратится гораздо больше ресурсов. Вот пример:

```
>>> x = numpy.array([0., 2., 0.5, -1.7, 3.])
>>> x
array([ 0. ,  2. ,  0.5, -1.7,  3. ])
>>> x += 1
>>> print(x)
[ 1.   3.   1.5 -0.7  4. ]
>>> x -= 2.5
>>> print(x)
[-1.5  0.5 -1.  -3.2  1.5]
>>> x += numpy.linspace(0, 4, 5)
>>> print(x)
[-1.5  1.5  1.  -0.2  5.5]
```

Как видим, прибавлять и вычитать из массивов таким образом можно как числа, так и другие массивы (нужно, чтобы совпал размер).

Использование операторов +=, -= в ряде случаев позволяет избежать некоторых сложно уловимых ошибок с именами переменных и, иногда, с их типами.

Скажем, нужно увеличить переменную со сложным именем, а затем использовать её в дальнейших вычислениях. Вот пример, когда переменная со сложным названием `komputilo_de_nombro_de_eventoj` используется в качестве счётчика событий (нужно посчитать число положительных элементов массива `x`, описанного ранее):

```
>>> komputilo_de_nombro_de_eventoj = 0
>>> for ela in x:
        if ela > 0:
            komputilo_de_npmbroj_de_eventoj = \
            komputilo_de_nombro_de_eventoj + 1
>>> print(komputilo_de_nombro_de_eventoj)
0
```

Программа отработала без сообщений об ошибках, но результат получился неверный: 0 вместо 3. Всё дело в том, что мы ошиблись в имени переменной в строке внутри условного оператора. В результате на каждом шаге цикла, когда условие оказалось истинно, создавалась новая переменная `komputilo_de_npmbroj_de_eventoj`, которой каждый раз присваивалось одно и то же значение 1 (поскольку исходная переменная-счётчик была равна 0, а к ней прибавлялось 1), нужная же переменная `komputilo_de_nombro_de_eventoj` не изменялась. Эта ошибка была бы невозможна при использовании оператора `-=`:

```
>>> komputilo_de_nombro_de_eventoj = 0
>>> for ela in x:
        if ela > 0:
            komputilo_de_npmbroj_de_eventoj += 1


Traceback (most recent call last):
  File "<pyshell#92>", line 3, in <module>
    komputilo_de_npmbroj_de_eventoj += 1
NameError: name 'komputilo_de_npmbroj_de_eventoj' is not
defined
```

В результате мы получили сообщение об ошибке `NameError`, сразу локализовав возникшую проблему. Такие ошибки могут быть очень коварны в случае, если полученное на данном этапе значение используется в дальнейших вычислениях.

5.3 Двумерные массивы, форма массивов

Никакие сложные вычисления не могут обойтись без двумерных массивов, часто называемых матрицами. Модуль `numpy` позволяет оперировать с массивами произвольной размерности: одномерными, двумерными, трёхмерными и т. д. Чтобы работать с массивами произвольной размерности, удобно пользоваться

понятием формы (shape). Форма — совокупность длин массива по каждому измерению. Обычно форма задаётся в виде кортежа из нескольких чисел: двух для двумерного, трёх для трёхмерного и т. д. Форма — свойство массива и обозначается shape. Форму массива можно задать при его создании. Если массив создаётся при помощи какой-нибудь встроенной функции numpy, его форма определяется этой функцией. Для одномерных массивов форма — это просто его длина.

```
from numpy import *
x = zeros((2, 3))
y = eye(3)
print(x)
print(y)
print(x.shape)
print(len(x))
print(y.shape)
```

Вот вывод программы:

```
[[ 0.  0.  0.]
 [ 0.  0.  0.]]
[[ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]]
(2, 3)
2
(3, 3)
```

В приведённом примере мы создали матрицу размера 2×3 из нулей с помощью функции zeros (в качестве аргумента передаём кортеж из двух числе (2, 3)); аналогично можно создать матрицу произвольного размера. Функция eye принимает единственный скалярный аргумент, поскольку она создаёт единичную квадратную матрицу заданного размера. Видно, что длина массива (len) — это его первая размерность (в ранних версиях numpy длина соответствовала общему числу элементов, т. е. произведению размерностей).

Также к двумерным массивам применимы операции sum, mean, max и т. д. По умолчанию, эти операции применяются к массиву, как если бы он был линейным списком всех чисел, независимо от его формы. Однако, указав параметр axis, можно применить операцию для указанной оси массива:

```
from numpy import *
a = array([[1, 2, 3], [4, 5, 6]])
print(a.sum())
print(a.sum(axis=0))
print(a.sum(axis=1))
print(a.min())
print(a.min(axis=0))
print(a.min(axis=1))
```

Вывод программы:

```
21
[5 7 9]
[ 6 15]
1
[1 2 3]
[1 4]
```

Видно, что `axis=0` работает с каждым столбцом отдельно, а `axis=1` — с каждою строкою отдельно.

Итерирование многомерных массивов начинается с нулевой оси:

```
from numpy import *
a = array([[1, 2, 3], [4, 5, 6]])
for element in a:
    print(element)
```

Вывод:

```
[1 2 3]
[4 5 6]
```

Однако если нужно перебрать поэлементно весь массив, как если бы он был одномерным, для этого можно использовать атрибут `flat`:

```
for element in a.flat:
    print(element)
```

Вывод:

```
1
2
3
4
5
6
```

Несколько массивов могут быть объединены вместе вдоль разных осей с помощью функций `hstack` и `vstack`: `hstack()` объединяет массивы по начальному индексу, `vstack()` — по последнему:

```
from numpy import *
a = array([[1, 2, 3], [4, 5, 6]])
b = array([[5, 6, 7], [8, 9, 10]])
print(vstack((a, b)))
print(hstack((a, b)))
```

Вывод:

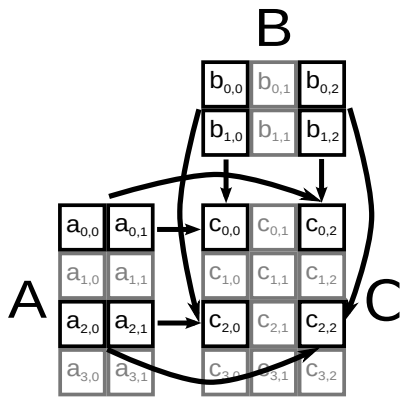


Рис. 5.3. Схема матричного умножения: $C = A \times B$.

```
[[ 1  2  3]
 [ 4  5  6]
 [ 5  6  7]
 [ 8  9 10]]
[[ 1  2  3  5  6  7]
 [ 4  5  6  8  9 10]]
```

Используя `hsplit()` вы можете разбить массив вдоль горизонтальной оси, указав либо число возвращаемых массивов одинаковой формы, либо номера столбцов, после которых массив разрезается «ножницами». Но лучше для этого использовать срезы. Например, выведем первый столбец массива `a`:

```
print (a[:, 1])
```

Вывод:

```
[2 5]
```

Модуль `numpy` предоставляет множество различных функций для работы с многомерными массивами, одна из самых популярных — функция `dot`. Для матриц и одномерных массивов-векторов эта функция соответствует функции матричного умножения, см. рис. 5.3 (скалярное произведение для двух одномерных массивов):

```
from numpy import *
a = array([[1, 0], [0, 1]])
b = array([[4, 1], [2, 2]])
print(dot(a, b))
x = array([-4, 1, 3, 2])
y = array([-1, 8, 1, -2])
```

```
print(dot(x, y))
z = array([5, -2])
print(dot(z, b))
print(dot(b, z))
```

Вывод программы показывает, что одномерный массив интерпретируется как вектор-столбец или вектор-строка в зависимости от контекста (об этом не нужно заботиться дополнительно):

```
[[4 1]
 [2 2]]
11
[16  1]
[18  6]
```

В версиях Python, начиная с 3.5 и новее для обозначения матричного умножения вместо функции `dot` можно использовать символ `@`:

```
>>> from numpy import *
>>> a = array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> a @ a
array([[ 30,  36,  42],
       [ 66,  81,  96],
       [102, 126, 150]])
```

Существуют две очень простые в использовании и чрезвычайно полезные функции в `numpy`, которые легко позволяют перейти от задач чисто учебных к задачам практически важным. Дело в том, что на практике все данные хранятся в файлах, чаще всего текстовых. Функция `loadtxt` позволяет загрузить данные из текстового файла в двумерный или одномерный, если в исходном файле данные представлены в 1 столбец или строку, массив. Функция `savetxt` в свою очередь позволяет записать массив в текстовый файл. Пример работы этих функций приведён ниже:

```
from numpy import *
a = eye(3)
b = arange(0, 10, 1)
savetxt('eye3.txt', a)
savetxt('Mas.txt', b)
a2 = loadtxt('eye3.txt')
print (a2)
b2 = loadtxt('Mas.txt')
print(b2)
```

По выводу программы видно, что чтение в переменную `a2` произошло успешно:

```
[[ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]]
[ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9.]
```

Наличие файлов 'eye3.txt' и 'Mas.txt' на диске и его содержимое вы можете проверить с помощью вашего файлового менеджера.

5.4 Примеры решения задач

Пример задачи 14 Создайте и выведите на экран квадратную матрицу размера $n \times n$, где на главной диагонали стоят нули, элементы выше неё — единицы, ниже — минус единицы. Затем сохраните получившийся массив в текстовый файл.

Решение задачи 14

```
from numpy import *
n=int(input('Размер матрицы: '))
x = zeros((n, n))
for i in range(n):
    x[i, :i] = -1
    x[i, i+1:] = 1
print (x)
savetxt('Mas.txt', x)
```

Вывод программы:

```
Размер матрицы: 3
[[ 0.  1.  1.]
 [-1.  0.  1.]
 [-1. -1.  0.]]
```

Заметим, что здесь мы воспользовались срезами, попутно сократив 1 цикл и убрав условный оператор.

Пример задачи 15 С помощью функции loadtxt загрузите из файла, созданного в предыдущем задании, данные в массив y. Убедитесь, что новый массив получился двумерным. Создайте одномерный массив-диапазон и прибавьте его к вашей матрице. Посмотрите, что получилось. Определите максимальный и минимальный элементы массива. Посчитайте сумму элементов по каждой строке массива. Запишите в два отдельных текстовых файла ваш массив-матрицу и массив-вектор.

Решение задачи 15

```
from numpy import *
y = loadtxt('Mas.txt')
print(y.shape)
a = arange(0, n, 1)
y = y + a
print(y)
print(y.min())
print(y.max())
print(y.sum(axis=1))
savetxt('Matrix.txt', y)
savetxt('Vector.txt', a)
```

Вывод программы:

```
(3, 3)
[[ 0.  2.  3.]
 [-1.  1.  3.]
 [-1.  0.  2.]]
-1.0
3.0
[ 5.  3.  1.]
```

Пример задачи 16 Протабулируйте (вычислите значения функций при изменении аргумента в некоторых пределах с заданным шагом) функцию на отрезке $[-\pi; \pi]$.

Решение задачи 16

```
from numpy import *
t = arange(-pi, pi, pi/6)
x = sin(t)
for i in range(len(t)):
    print("{:12.8f}\t{:12.8f}".format(t[i], x[i]))
```

Вывод программы:

```
-3.14159265  -0.00000000
-2.61799388  -0.50000000
-2.09439510  -0.86602540
-1.57079633  -1.00000000
-1.04719755  -0.86602540
-0.52359878  -0.50000000
-0.00000000  -0.00000000
 0.52359878   0.50000000
 1.04719755   0.86602540
```

```
1.57079633   1.00000000
2.09439510   0.86602540
2.61799388   0.50000000
```

5.5 Задания на массивы, модуль numpy

В заданиях 15 и 17 следует выполнить три пункта в зависимости от номера в списке группы в алфавитном порядке. Необходимо сделать задания №m, №$m+5$, №$m+10$, $m = (n-1)\%5+1$, где n — номер в списке группы.

Задание 15 *Создайте и выведите на экран массивы. Получившиеся матрицы сохраните в текстовые файлы.*

1. из нулей одномерные длины 10 и 55, матрицу размерами 3×4, трёхмерный массив формы $2 \times 4 \times 5$;
2. из единиц одномерные длины 10 и 55, матрицу размерами 3×4, трёхмерный массив формы $2 \times 4 \times 5$;
3. из девяток одномерные длины 10 и 55, матрицу размерами 3×4, трёхмерный массив формы $2 \times 4 \times 5$;
4. одномерные длины 10 и 55, матрицу размерами 3×4, трёхмерный массив формы $2 \times 4 \times 5$, все состоящие целиком из значений 0.25;
5. массив-диапазон от -10 до 10 с шагом 0.1;
6. массив-диапазон от $-e$ до e с шагом $e/50$;
7. массив-диапазон от -15π до 15π с шагом $pi/12$;
8. единичную матрицу размера 5×5;
9. диагональную матрицу размера 5×5, все значения на главной диагонали которой равны 0.5;
10. матрицу размера 5×5, где на главной диагонали стоят единицы, а прочие элементы равны 2;
11. матрицу размера 5×5, где в первом столбце стоят единицы, во втором — двойки, в третьем — тройки и т. д.
12. матрицу размера 5×5, где в первой строке стоят единицы, во втором — двойки, в третьем — тройки и т. д.
13. матрицу размера 5×5, где на главной диагонали стоят нули, элементы выше неё — единицы, ниже — минус единицы;

14. верхнюю треугольную матрицу 5×5, где все элементы выше главной диагонали равны -2, а на ней — единицы;
15. нижнюю треугольную матрицу 5×5, где все элементы ниже главной диагонали равны 2, а на ней — единицы.

Внимание: задания 10–15 требуют умения манипулировать с отдельными столбцами или строками двумерного массива!

Задание 16 Загрузите из файла, созданного в предыдущем задании, данные в массив. Убедитесь, что новый массив получился двумерный. Создайте одномерный массив-диапазон и прибавьте его к вашей матрице. Посмотрите, что получилось. Определите максимальный и минимальный элементы массива. Посчитайте сумму элементов по каждой строке массива. Запишите в два отдельных текстовых файла ваши массив-матрицу и массив-вектор.

Задание 17 Протабулируйте (вычислите значения функций при изменении аргумента в некоторых пределах с заданным шагом) функции:

1. x^2 на отрезке $x \in [-2; 2]$ с шагом 0.01, с шагом 0.1, с шагом 0.25;
2. x^3 на отрезке $x \in [-2; 2]$ с шагом 0.01, с шагом 0.1, с шагом 0.25;
3. x^4 на отрезке $x \in [-2; 2]$ с шагом 0.01, с шагом 0.1, с шагом 0.25;
4. $\cos(2\pi t)$ на отрезке $t \in [-10; 10]$ с шагом 1 и с шагом 0.25;
5. $\frac{1}{t}\cos(2\pi t)$ на отрезке $t \in [1; 10]$ с шагом 1 и с шагом 0.25;
6. $e^{-t}\cos(2\pi t)$ на отрезке $t \in [-10; 10]$ с шагом 1 и с шагом 0.25;
7. $4\sin(\pi t + \pi/8) - 1$ на отрезке $t \in [-10; 10]$ с шагом 1 и с шагом 0.25;
8. $2\cos(t - 2) + \sin(2 * t - 4)$ на отрезке $t \in [-20\pi; 10\pi]$ с шагом π и с шагом $\pi/12$;
9. $\ln(x + 1)$ на отрезке $x \in [0; e - 1]$ с шагом 0.01 и с шагом 0.001;
10. $\log_2(|x|)$ на отрезке $x \in [-4; 4]$ за исключением точки $x = 0$ с шагом 0.1 и с шагом 0.25;
11. 2^x на отрезке $x \in [-2; 2]$ с шагом 0.01, с шагом 0.1, с шагом 0.25;
12. e^x на отрезке $x \in [-2; 2]$ с шагом 0.01, с шагом 0.1, с шагом 0.25;
13. 2^{-x} на отрезке $x \in [-2; 2]$ с шагом 0.01, с шагом 0.1, с шагом 0.25;
14. $\sqrt[3]{x}$ на отрезке $x \in [1; 125]$ с шагом 1 и с шагом 5, но так, чтобы значения 1 и 5 присутствовали среди аргументов;
15. $\sqrt[5]{x}$ на отрезке $x \in [1; 32]$ с шагом 1 и с шагом 0.25.

Глава 6

Графики. Модуль `matplotlib`

Построение графиков — один из главных этапов обработки данных. Все современные компьютерные программы, предоставляющие функцию построения графиков, условно можно разделить на две категории: программы с визуальным интерфейсом, где построение и редактирование графика осуществляется средствами разного рода меню, полей ввода, лист-боксов, чек-боксов и других виджетов, и программы, где для построения графика необходимо писать команды, объединяемые в так называемы скрипты. К первой категории относятся, например, Origin, MS Excel, OpenOffice/LibreOffice Calc, Statistica, Grapher, ко второй — gnuplot, многие математические пакеты, например, MATLAB и SciLab и различные библиотеки вроде PGPlot и PLPlot, имеющие поддержку во многих языках программирования.

Основное преимущество скриптового способа построения графика в том, что вы можете встроить его без проблем в вашу программу, производящую вычисления. Кроме того, скрипты позволяют легко перестраивать графики с новыми данными, автоматизировать построение графиков, дают почти неограниченный контроль над точностью позиционирования и размером деталей.

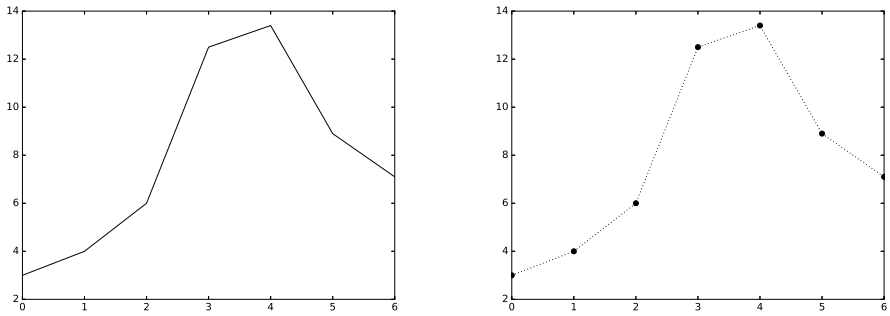
Модуль `matplotlib` — специализированная библиотека для языка Python. Хотя основное её преимущество в простоте и быстроте использования, она позволяет делать графики очень высокого типографского качества. Модуль `matplotlib` базируется на возможностях `numpy`, установка которого обязательна для функционирования `matplotlib`.

6.1 Простые графики

Непосредственно за построение графиков отвечает библиотека `pyplot` модуля `matplotlib`. Основная команда для построения графиков — команда `plot`. У команды `plot` существует множество вариантов синтаксиса, в простейшем случае она требует единственный аргумент — массив, но подойдёт и список, кортеж или даже диапазон. Вот пример построения простейшего графика (см. рис. 6.1(а)):

```
from matplotlib.pyplot import *
x = [3, 4, 6, 12.5, 13.4, 8.9, 7.1]
plot(x)
show()
```

Команда `show()` даётся без аргументов и нужна для того, чтобы вывести на экран содержимое графического буфера, куда заносятся результаты ваших действий. В приведённом примере получился график в виде ломаной, где по оси абсцисс отложены номера элементов списка `x`, а по оси ординат — сами эти элементы. Чтобы изменить тип линии или заменить её на маркеры, можно указать дополнительный аргумент в команде `plot` — строку форматирования. Например, строка из одного символа дефиса `'-'` соответствует поведению по умолчанию — сплошной линии, `'--'` — прерывистой линии, `':'` — пунктиру. При этом маркеры: кружочки, квадраты, ромбы будут располагаться только в тех местах, где есть данные.



(a) (b)

Рис. 6.1. Простейший график сплошными линиями — (a) и пунктиром и кружками одновременно — (b).

Маркеры и линии можно комбинировать. Изменим предпоследнюю строчку предыдущей программы следующим образом (результат на рис. 6.1(b)):

```
plot(x, ':o')
show()
```

В результате, тот же график будет выглядеть немного иначе: на месте расположения данных появятся жирные синие кружки, соединённые тонкою пунктирною линией. Чтобы изменить цвет графика, достаточно указать дополнительный именованный параметр `color`:

```
plot(x, ':o', color='grey')
show()
```

Таблица 6.1. Встроенные типы линий (`linestyle`) и маркеров (`markers`).

Строка форматирования	Описание
`'-'`	сплошная линия
`'-'`	прерывистая линия
`'-.'`	штрих-пунктир
`':'`	пунктирная линия
`'.'`	маркер точка
`','`	маркер пиксель
`'o'`	маркер кружочек
`'v'`	маркер треугольник углом вниз
`'^'`	маркер треугольник углом вверх
`'<'`	маркер треугольник углом влево
`'>'`	маркер треугольник углом вправо
`'s'`	маркер квадрат
`'d'`	маркер вытянутый ромб
`'D'`	маркер квадрат, повёрнутый на 45°
`'p'`	маркер пятиугольник
`'h'`	маркер шестиугольник
`'*'`	маркер звёздочка
`'+'`	маркер плюс
`'x'`	маркер крестик, как знак умножения ×
`'|'`	маркер вертикальный отрезок
`'_'`	маркер горизонтальный отрезок

В данном случае было использовано специальное именованное значение `'grey'`, соответствующее серому цвету. Существуют значения `'red'`, `'black'`, `'blue'`, `'yellow'`, `'green'`, `'magenta'`, `'cyan'` и некоторые другие. Если желаемый цвет не встречается среди именованных значений, можно воспользоваться кодировкою в стиле RGB — задать значения компонентов красного, зелёного и синего в долях от максимально возможного: 1 соответствует максимально возможной интенсивности каждого цвета, 0 — его нулевой интенсивности. Все три интенсивности ставятся в кортеж. Вот как получить темно-фиолетовый цвет:

```
plot(x, ':o', color=(0.5, 0, 0.5))
show()
```

Если вы хотите получить оттенок серого, вместо трёх чисел достаточно указать только одно, поставленное в кавычки:

```
plot(x, ':o', color='0.5')
show()
```

даст средне-серый цвет.

Рассмотренный выше пример прост, но недостаточно полон: часто оказывается необходимо отложить по оси абсцисс не номера точек, а какие-то осознанные значения. Это можно сделать, передав функции `plot` два списка. Рассмотрим этот случай на примере построения синусоиды:

```
from matplotlib.pyplot import plot, show
from math import pi, sin
t = []
x = []
for i in range(400):
    t.append(i*0.01)
    x.append(sin(2*pi*t[i]))
plot(t, x, color='grey')
show()
```

Команда `plot` всегда пытается рассматривать первые аргументы как массивы со значениями до тех пор, пока не встретит именованный аргумент или строку. В приведённом примере (см. цветную вкладку рис. 1(a)) в качестве аргументов выступали 2 списка, первый интерпретируется как абсциссы точек, второй — как их ординаты. Можно построить несколько кривых на одном поле одною командою, например, синус и косинус разом (для этого нужно указывать абсциссы и ординаты попеременно):

```
from matplotlib.pyplot import *
from math import *
t = []
x = []
y = []
for i in range(400):
    t.append(i*0.01)
    x.append(sin(2*pi*t[i]))
    y.append(cos(2*pi*t[i]))
plot(t, x, t, y)
show()
```

На полученных графиках (см. цветную вкладку рис. 1(b)) синусоида получится синего цвета, а косинусоида — зелёного; это потому, что matplotlib самостоятельно чередует цвета, если несколько кривых отображаются на одном графике. Последние две строчки в программе можно было бы поменять следующим образом без изменения результата:

```
plot(t, x)
plot(t, y)
show()
```

Кроме просмотра графиков на экране современные графопостроители обязаны поддерживать вывод изображения в файл. В matplotlib это делается очень просто с помощью команды savefig, единственным обязательным аргументом которой является имя файла. Изменим последнюю строку предыдущей программы так, чтобы она не только выводила график на экран, но и сохраняла его в файл 'sincos.png':

```
plot(t, x)
plot(t, y)
savefig('sincos.png')
show()
```

При необходимости, вывод на экран можно убрать, вывод в файл от этого не пострадает. Важно только помнить, что функция show не только выводит изображение на экран, но и высвобождает буфер так, что он остаётся пуст. Поэтому перестановка местами функций savefig и show не скажется на выводе на экран, но в файл будет записано пустое полотно.

Модуль matplotlib поддерживает несколько популярных форматов, в частности png для растровой графики, eps и pdf — для векторной. Можно сохранить график несколько раз, в том числе в файлы разных форматов:

```
plot(t, x)
plot(t, y)
savefig('sincos.png')
savefig('sincos.pdf')
show()
```

6.2 Заголовок, подписи, сетка, легенда

Если график строится просто для себя, и его не планируется вставлять, например, в научную статью или диплом, представленных выше возможностей matplotlib может показаться достаточным. Но чтобы показать его другим людям, да и самому не забыть, что нарисовано и что по какой оси отложено, полезно уметь ставить подписи. Простейшие подписи: заголовок и подписи к осям ставятся командами title, xlabel и ylabel, принимающими единственный аргумент — строку:

```
from numpy import *
from matplotlib.pyplot import *
t = arange(0, 4, 0.01)
x = sin(2*pi*t)
y = cos(2*pi*t)
plot(t, x, t, y)
title('Voltage␣over␣time')
xlabel('time,␣s')
```

```
ylabel('voltage, V')
show()
```

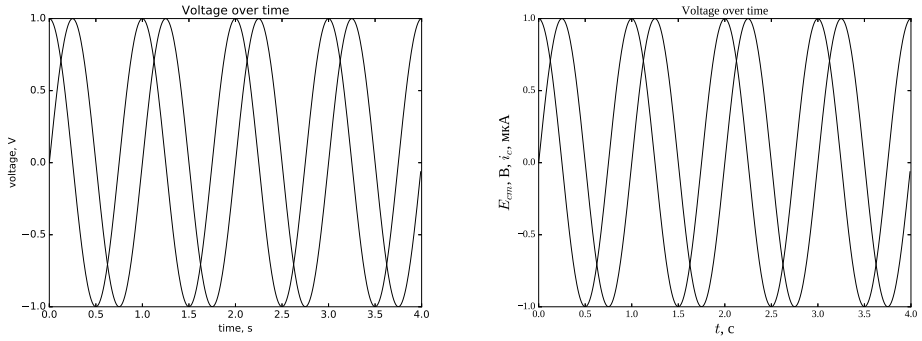
Заметим, что на практике всё же удобнее работать не со списками, а с массивами, что мы продемонстрировали здесь, заменив явный цикл операциями с массивами из модуля `numpy`.

Команды `plot`, `title`, `xlabel` и др. формируют изображение, поэтому они должны предшествовать командам вывода типа `savefig` или `show`; порядок друг относительно друга — произвольный.

Часто необходимо, чтобы в подписях на осях или легенде содержались верхние или нижние индексы, греческие буквы, различные значки и прочие математические символы. Для этого `matplotlib` имеет режим совместимости с командами LaTeX. Чтобы использовать его, нужно поставить перед строкою символ `'r'`, поскольку и в Python, и в LaTeXсимвол `'\'` является специальным и вводит команды. Строка, начинающаяся с `'r'`, называется сырою. В такой строке все символы интепретируются как они есть, в том числе `'\t'` и `'\n'` не будут означать табуляцию и конец строки, а просто будут парами символов. Внутри сырой строки нужно поставить символ `'$'`, с которого начинается и которым заканчивается формула LaTeX. Внутри пары из `'$'` можно использовать многие специальные обозначения: `'_'` для нижнего индекса (см. рис. 6.2(b)), `'^'` — для верхнего. Если в индекс входит несколько символов, следует заключить их в фигурные скобки, служащие в LaTeXспециальными символами наряду с `'\'`; причём сами фигурные скобки не отображаются. Кроме того, можно использовать почти все стандартные команды LaTeX, например, `'$\dot{x}$'` нарисует `'x'` с точкою наверху (будет выглядеть $\dot{x}$ — так обозначают производную), а `'$\to$'` нарисует красивую стрелку слева направо ($\to$, см. рис. 6.2(a)):

```
xlabel(r'$t$, c', fontsize=18)
ylabel(r'$E_{cm}$, B, $i_{c}$, мкА', fontsize=18)
```

Если на графике несколько кривых, возникает желание как-то подписать их. Такая подпись, помещённая на график, как правило называется «легендой». Чтобы сделать легенду средствами `matplotlib` нужно, во-первых, указать при построении каждой кривой требуемую подпись с помощью параметра с ключевым словом `label`, во-вторых, вызвать специальную функцию `legend`, рисующую легенду. Для того, чтобы можно было использовать в подписях нелатинские буквы, нужно установить шрифт, их поддерживающий. По умолчанию `matplotlib` использует шрифты без засечек (в типографии такие шрифты принято обозначать по-французски «Sans Serif»), список которых содержится в переменной `rcParams['font.sans-serif']`, где `rcParams` — словарь, а `'font.sans-serif'` — ключ. Если в вашей системе нужные шрифты не установлены, либо не имеют необходимых символов, вместо букв получатся «крякозябры» или просто квадратики. В таком случае нужно переопределить переменную `rcParams['font.sans-serif']`, записав туда список шрифтов, в которых нужные символы имеются. Например, в Windows можно использовать шрифт `'Arial'`, как это показано ниже.

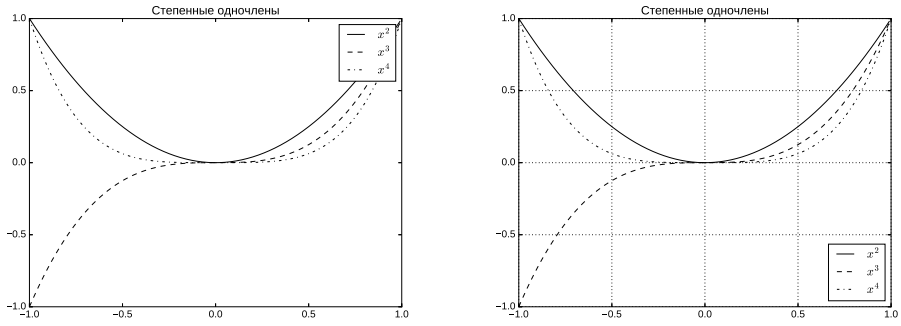


(a) (b)

Рис. 6.2. Графики с подписями и названием.

```
from numpy import *
from matplotlib.pyplot import *
from matplotlib import rcParams
rcParams['font.sans-serif']=['Arial']
t = arange(-1, 1, 0.01)
x = t**2
y = t**3
z = t**4
plot(t, x, label=r'$x^2$')
plot(t, y, '--', label=r'$x^3$')
plot(t, z, ':', label=r'$x^4$')
legend()
title('Степенные одночлены')
show()
```

Получившаяся программа строит легенду, но пока качество её нас не устраивает: во-первых, она налезает на графики, во-вторых, подписи выглядят не очень красиво — см. рис. 6.3(a). Чтобы исправить первый недостаток, нужно использовать параметр функции `legend` с названием `loc`, отвечающий за расположение. Параметр `loc` может принимать числовые значения с нуля по 10, либо соответствующие им строковые значения типа `'upper right'` — верхний правый угол, или `'lower center'` — посредине внизу. Значение 0 соответствует строке `'best'` — Python сам пытается выбрать такое положение легенды, чтобы она не заслоняла график (см. рис. 6.3(b)). Ещё один часто встречающийся элемент графиков — сетка. Сетка используется для того, чтобы лучше видеть относительное расположение далеко разнесённых значений. Для получения сетки нужно всего лишь



(a) (b)

Рис. 6.3. Пример использования легенды: (a) — расположение по умолчанию, (b) — оптимальное расположение с дополнительно наложенною сеткою.

добавить (как обычно, до `savefig` и `show`) в программу функцию `grid`. В итоге добавим в нашу программу ещё две строчки:

```
legend(loc='best')
grid(True)
```

У функции `grid` единственный логический аргумент. Если его значение правда — сетка будет, если ложь — нет.

6.3 Несколько графиков на одном полотне

Модуль `matplotlib` позволяет построить несколько графиков на одном полотне. Для этого существует команда `subplot`, определяющая, в какой части полотна расположен выводимый в настоящий момент график, и осуществляющая его масштабирование (по умолчанию, если `subplot` ни разу не использован, вывод осуществляется на всё полотно). Команда `subplot` имеет три обязательных аргумента — целых числа. Первое означает, на сколько частей будет разделено полотно по вертикали, второе — по горизонтали (получается своеобразная сетка), третье — номер элемента в этой сетке, нумерация идёт сначала по горизонтали слева направо, потом по вертикали сверху вниз. Чтобы лучше понять работу `subplot`, создадим программу, рисующую графики различных одночленов на разных графиках.

```
from numpy import *
from matplotlib.pyplot import *
t = arange(0, 2, 0.01)
x = sqrt(t)
```

```
y = t
z = sqrt(t**3)
u = t**2
v = sqrt(t**5)
w = t**3
subplot(3, 2, 1)
plot(t, x, label='x**(1/2)')
title(r'$\sqrt{x}$')
ylim([0, 5])

subplot(3, 2, 2)
plot(t, y, label='x', color='red')
title(r'$x$')
ylim([0, 5])

subplot(3, 2, 3)
plot(t, z, label='x**(3/2)')
title(r'$\sqrt{x^3}$')
ylim([0, 5])

subplot(3, 2, 4)
plot(t, u, ':', label='x**2')
title(r'$x^2$')
ylim([0, 5])

subplot(3, 2, 5)
plot(t, v, label='x**(5/2)')
title(r'$\sqrt{x^5}$')
ylim([0, 5])

subplot(3, 2, 6)
plot(t, w, label=r'$x^3$')
title(r'$x^3$')

legend(loc=0)
grid(True)
tight_layout()
show()
```

Из приведённого примера видно следующее. Во-первых, все команды построения такие, как `plot`, `xlabel`, `title` и др., включая рисование сетки `grid` и легенды `legend`, относятся только к текущему графику. Во-вторых, для каждого графика стиль и цвет линий выставляется независимо (см. цветную вкладку рис. 2), поэтому, изменив цвет второго и стиль линий четвёртого, мы получили

цвет и стиль линий по умолчания для всех остальных, идущих как до, так и после. С помощью команды `ylim([0, 5])` у всех графиков кроме последнего, задали фиксированные пределы по оси ординат. Функция `tight_layout()` обеспечивает расположение графиков на одном полотне без залезания друг на друга автоматически.

Можно сделать так, чтобы графики на полотне занимали разный размер, например, сверху два, снизу — один. Дело в том, что функция `subplot` просто определяет положение графика на полотне, она не задаёт никакой реальной сетки. Поэтому могут одновременно встречаться графики, для которых сетка задана по-разному.

Приведём пример из радиофизики. На выходах микрофона, передающей телекамеры и различных датчиков создаются низкочастотные сигналы с малой амплитудой. Таким сигналам свойственно большое затухание в пространстве и, следовательно, они не могут передавать информацию по каналам связи. Для эффективной передачи сигналов в произвольной среде необходимо перенести спектр данных сигналов из низкочастотной области в область высоких частот. Такой процесс называется модуляцией.

Модуляция — процесс изменения одного или нескольких параметров высокочастотного несущего колебания по закону низкочастотного информационного сигнала (сообщения). Будем использовать амплитудную модуляцию — вид модуляции, при которой изменяемым параметром несущего сигнала является его амплитуда. Передаваемая информация заложена в управляющем (модулирующем) сигнале (в данном примере $x = \cos(2\pi t)$), а роль переносчика информации выполняет высокочастотное колебание ($y = \cos(20 \cdot 2\pi t)$), называемое несущим. Модуляция, таким образом, представляет собой процесс «посадки» информационного колебания на заведомо известную несущую. Формула амплитудно-модулированного колебания выглядит следующим образом: $z = (1 + M \cdot x) \cdot y$, где M — глубина модуляции.

Вот пример генерации и отображения такого модулированного сигнала на компьютере:

```
from numpy import *
from matplotlib.pyplot import *
rcParams['font.sans-serif']=['Arial']
t = arange(0, 10, 0.001)
x = cos (2*pi*t)
y = cos (20*2*pi*t)
z = (1+0.7*x)*y
subplot(2, 2, 1)
plot(t[:2000], x[:2000], color='black')
title('Модулирующий сигнал')
xlabel('Time, s')
ylabel('Voltage, V')
subplot(2, 2, 2)
```

```
plot(t[:2000], y[:2000], color='black')
title('Несущий сигнал')
xlabel('Time, s')
ylabel('Voltage, V')
subplot(2, 1, 2)
plot(t, z, color='black')
title('AM-сигнал')
xlabel('Time, s')
ylabel('Voltage, V')
tight_layout()
show()
```

Здесь видно (см. рис. 6.4), что мы как бы «обманываем» `matplotlib`: для графиков y и z мы говорим, что будем размещать графики по схеме 2×2, а для графика x — по схеме 2×1. В результате третий график занимает всю нижнюю половину, как будто он второй из двух, а первые — каждый по четверти в верхней половине, как первый и второй из четырёх.

Часто бывает так, что требуется построить не несколько графиков в одном окне, а несколько окон с графиками; такая возможность в `matplotlib` предусмотрена. Например, вам нужно построить несколько разных рисунков и сохранить каждый в отдельный файл. Для этого нужно воспользоваться функцией `figure`. Вызов `figure` подобен вызову `subplot` в том смысле, что рисование начинается заново, а все ранее нарисованные объекты: сами графики, подписи к ним, сетка, легенда и проч. остаются на предыдущем полотне (или полотнах). Разница в том, что `subplot` размещает новый график в пределах всё того же полотна, а `figure` просто создаёт новое. Вот что будет, если переписать предыдущий пример с помощью `figure`:

```
from numpy import *
from matplotlib.pyplot import *
t = arange(0, 1, 0.001)
x = sin(10*pi*t)
y = t
z = sqrt(t**3)
figure(1)
plot(t, y)
figure(2)
plot(t, z)
figure(3)
plot(t, x)
show()
```

В использованном примере у функции `figure` только один аргумент — номер создаваемого полотна. Номер этот полезен тем, что в любой момент вы можете вернуться к уже начатому графику и что-то там дорисовать; для этого просто

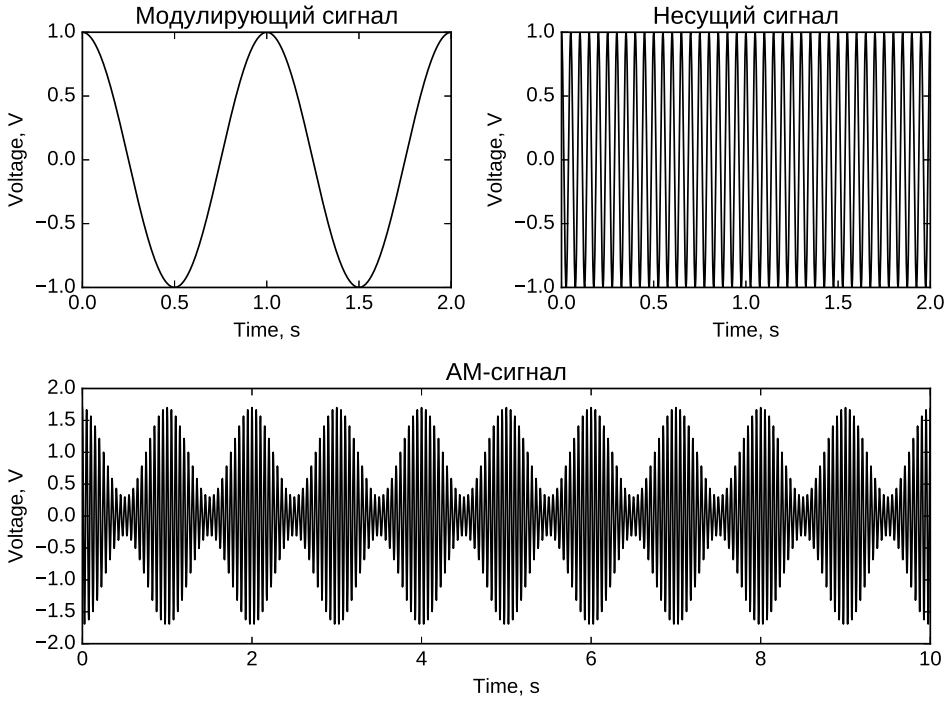


Рис. 6.4. Пример нескольких графиков различного размера на одном полотне.

необходимо сменить «фигуру» на ту, где вы уже рисовали. Так же полотну можно задать нужные размеры. Это делается с помощью именного аргумента `figsize` функции `figure`. Этот аргумент задаётся в виде кортежа из двух чисел — ширина и высота полотна. Вот пример:

```
import numpy as np
import matplotlib.pyplot as plt
t = np.arange(-1.0, 1.0, 0.001)
x = np.sin(2*np.pi*t)
y = np.cos(5*np.pi*t)
plt.figure(1, figsize = (8,3))
plt.plot(t, x)
plt.figure(2)
plt.plot(t, y)
plt.ylabel('cos(t)')
plt.figure(1)
plt.ylabel('sin(t)')
```

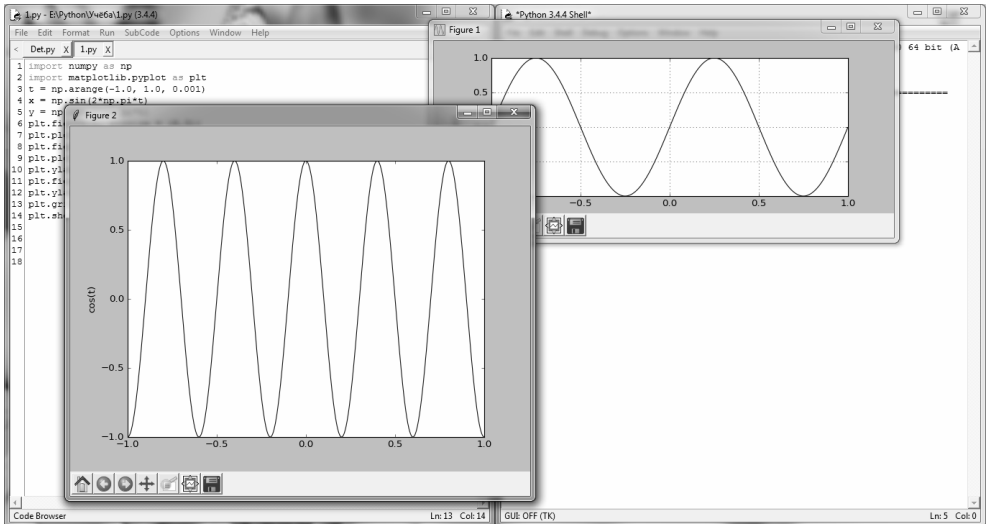


Рис. 6.5. Пример вызова нескольких полотен (`figure`).

```
plt.grid(True)
plt.show()
```

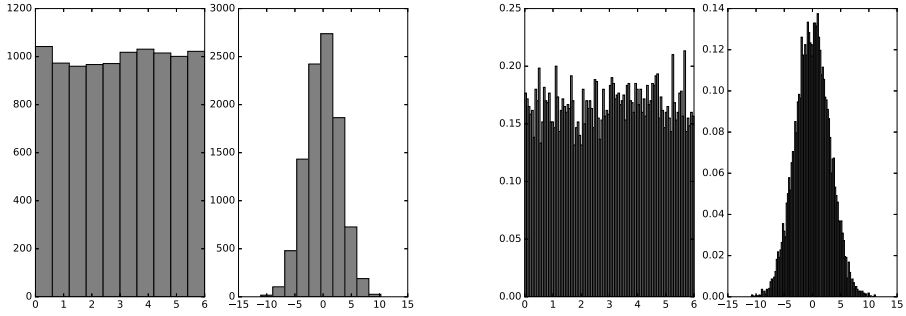
Здесь мы сначала нарисовали первую синусоиду в первом окне, потом вторую во втором, а затем вернулись к первому окну и доделали подписи к осям и сетку (рис. 6.5).

Создание каждого нового полотна потребляет оперативную память. Поэтому, если нет необходимости наблюдать несколько полотен одновременно, эффективнее один раз создавать полотно функцией `figure`, а потом на каждом шаге, например, цикла перерисовывать на нём новый рисунок, предварительно стирая предыдущий функцией `clf()`.

6.4 Гистограммы, диаграммы-столбцы

Кроме обычных графиков, отражающих зависимость одной величины от другой, бывает нужно построить графики другого типа, чаще всего это гистограммы. Гистограммы строят, чтобы следить за распределением некоторой величины. Если величина дискретна — каждому значению сопоставляют его частоту (число выпадений в данной реализации) или вероятность в процентах или долях единицы. Если величина изменяется непрерывно, её значения делят на диапазоны — бины, подсчитывая число попаданий в каждый диапазон.

Для примера построим гистограммы равномерно на отрезке $[0; 6]$ и нормально с параметрами $\mu = 0, \sigma = 3$ распределённых случайных величин, сгенерировав по 10000 значений в каждом случае. Воспользуемся стандартным модулем `random`.



(a) (b)

Рис. 6.6. Пример построения гистограмм: (a) — использованы значения параметров по умолчанию, в результате на обоих графиках наблюдаем по 10 бинов, а по вертикали отложено число попаданий в каждый из них; (b) — число бинов выставлено вручную и равно 100, по вертикали отложены значения плотности вероятности.

```
from random import uniform, normalvariate
from matplotlib.pyplot import *
v = []
for i in range(10000):
    v.append(uniform(0, 6))
subplot(1, 2, 1)
hist(v)
w = []
for i in range(10000):
    w.append(normalvariate(0, 3))
subplot(1, 2, 2)
hist(w)
show()
```

Представленная программа делит весь диапазон от минимального до максимального значения на 10 бинов и рисует частоты — число попаданий в каждый бин (см. рис. 6.6(a)). Для 10000 значений 10 бинов — маловато, поэтому используем необязательный параметр `bins`, принимающий целое число, равное желаемому количеству бинов. А вместо частот нужно получить плотность вероятности (т. е. площадь под графиком должна равняться единице), для чего используем ещё один необязательный параметр `normed`, принимающий логическое значение. Изменим программу, указав дополнительные параметры:

```
from random import uniform, normalvariate
```

```
from matplotlib.pyplot import *
v = []
for i in range(10000):
    v.append(uniform(0, 6))
subplot(1, 2, 1)
hist(v, bins=100, normed=True)
w = []
for i in range(10000):
    w.append(normalvariate(0, 3))
subplot(1, 2, 2)
hist(w, bins=100, normed=True)
show()
```

Теперь видим (см. рис. 6.6(b)), что по вертикали отложено уже знание плотности вероятности, причём график стал более изрезанным, поскольку увеличилось число бинов, а число значеинй в каждом бине уменьшилось.

Иногда полезными бывают диаграммы-столбцы. На таких диаграммах горизонтальный или вертикальный прямоугольник показывает своей длиной вклад, вносимый каждым участником. Главная его задача состоит в сравнении этих количественных показателей.

Для визуализации используется функция `bar()`, принимающая две последовательности координат: x, определяющих левый край столбца, и y, определяющих высоту. Ширина прямоугольников по умолчанию равна 0.8. Но этот и другие параметры можно менять за счёт необязательных именованных параметров:

```
from numpy import *
from random import *
from matplotlib.pyplot import *
data1 = []
data2 = []
data3 = []
for i in range(10):
    data1.append(normalvariate(5, 0.5))
    data2.append(normalvariate(5, 0.5))
    data3.append(normalvariate(5, 0.5))
locs = arange(1, len(data1)+1)
width = 0.2
bar(locs, data1, width=width, color = 'blue')
bar(locs+width, data2, width=width, color = 'red')
bar(locs+2*width, data3, width=width, color = 'green')
xticks(locs + width*1.5, locs)
show()
```

В данном примере `width` задает ширину прямоугольника, `color` задает цвет прямоугольника. Опционно можно дописать `xerr`, `yerr`, которые позволяют устанавливать `error bars`. Далее генерируем последовательности трёх видов данных

(data) для пяти точек. Задаем переменную, которая будет определять толщину столбцов. Первый аргумент bar() имеет такой вид для того, чтобы три столбца стояли вместе, впритык друг к другу. Также здесь применяется фокус с функцией xticks, позволяющей изменять засечки на оси абсцисс, и мы смещаемся так, чтобы аргумент, породивший три своих столбца, стоял посередине — рис. 3(a). Часто используют и горизонтальное расположение. Оно описывается практически также, но вместо функции bar() используется barh(). Более того, такого рода диаграмм и возможностей существует великое множество, и вы сами можете ознакомиться с ними по документации matplotlib.

6.5 Круговые и контурные диаграммы

Достаточно распространённым способом графического изображения структуры статистических совокупностей является секторная диаграмма, так как идея целого очень наглядно выражается кругом, который представляет всю совокупность. Относительная величина каждого значения изображается в виде сектора круга, площадь которого соответствует вкладу этого значения в сумму значений. Этот вид графиков удобно использовать, когда нужно показать долю каждой величины в общем объёме. Секторы могут изображаться как в общем круге, так и отдельно, расположенными на небольшом удалении друг от друга.

Круговая диаграмма сохраняет наглядность только в том случае, если количество частей совокупности диаграммы небольшое. Если частей диаграммы слишком много, её применение неэффективно по причине несущественного различия или малого размера сравниваемых структур. Недостаток круговых диаграмм — малая информационная ёмкость, невозможность отразить более широкий объём полезной информации.

Нарисуем круговую диаграмму средствами matplotlib'а — рис. 3(b):

```
from matplotlib.pyplot import *
data = [18, 15, 11, 9, 8, 6]
labels = ['Java', 'C', 'C++', 'PHP', 'Python', 'Ruby']
explode = [0, 0, 0, 0, 0.2, 0]
axes(aspect=1)
pie(data, labels=labels, explode=explode,
    autopct='%1.1f%%', shadow=True)
show()
```

Данная программа задаёт набор данных (data), добавляет на рисунок оси с соотношением сторон 1:1 (axes), строит график в виде круговой диаграммы (pie). Первым аргументом функция pie принимает последовательность данных, затем code задаёт имена, по одному на каждый элемент data, далее задаётся explode — маска, по которой «вырезается кусок пирога», autopct задаёт тип форматирования численных значений, shadow добавляет тень. Как обычно, цвет чередуется сам собою, порядок по умолчанию для matplotlib это 'blue', 'green', 'red', 'cyan', 'magenta', 'yellow', 'black'.

Ещё одним специализированным видом графиков являются *контурные диаграммы*. Проще всего понять, что это такое, если вспомнить физическую карту миры: там высоты и глубины обозначены цветом от тёмно-коричневого до тёмно-синего, а значения разделены на диапазоны, внутри которых все значения красятся единым цветом. Контурная диаграмма — способ представления трёхмерной поверхности как бы «сверху».

Чтобы построить контурную диаграмму с помощью `matplotlib`, нужно задать три двумерных массива одинаковой формы: массив значений координаты x для каждого узла сетки, массив координаты y и массив значений функции от них $z = f(x, y)$. Если функция f нам известна, нужно определить пределы изменения x и y и задать шаг их изменения по каждой из осей, т. е. сетку. Это можно сделать с помощью функции `arange`. Далее из одномерных массивов, задающих сетку, нужно получить двумерные, содержащие координаты x и y в каждом узле сетки; это можно сделать командою `meshgrid`, как показано в примере далее. Сама контурная диаграмма строится с помощью команды `contour`, которой в качестве параметров передаются все три массива: x, y и z:

```
from numpy import *
from matplotlib.pyplot import *
from matplotlib.mlab import *
x = arange(-3, 3, 0.01)
y = arange(-2, 2, 0.01)
X, Y = meshgrid(x, y)
Z = X**2-4*Y**2+Y**4
contour(X, Y, Z)
show()
```

Представленная программа строит контурную диаграмму функции (см. цветную вкладку рис. 4(a)) с помощью линий уровней. Можно всё залить цветом, как на физической карте мира. Для этого необходимо использовать функцию `contourf` (см. цветную вкладку рис. 4(b)).

Конечно, на практике значения массивов x, y и z будут, скорее всего, известны и прибегать к таким ухищрениям не придётся. Тем не менее, предложенный пример — хорошая иллюстрация того, как построить график функции двух переменных, не прибегая к изометрической проекции.

6.6 Трёхмерные графики

В `matplotlib` кроме двумерных существует возможность построения трёхмерных графиков. Для это используется модуль `mplot3d`. Для того, чтобы нарисовать трехмерный график, в первую очередь надо создать трехмерные оси. Они задаются как объект класса `mpl_toolkits.mplot3d.Axes3D`, конструктор которого ожидает как минимум один параметр — экземпляр класса `matplotlib.figure.Figure`. У конструктора класса `Axes3D` есть ещё и другие

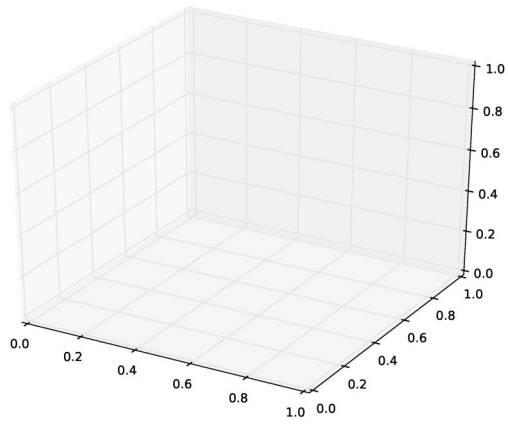


Рис. 6.7. 3D-оси.

необязательные параметры, но пока мы их использовать не будем. Давайте нарисуем пустые оси (рис. 6.7):

```
from matplotlib.pyplot import *
from mpl_toolkits.mplot3d import Axes3D
fig = figure()
Axes3D(fig)
show()
```

Полученные оси вы можете вращать мышкой в интерактивном режиме.

А теперь нарисуем что-нибудь трёхмерное в полученных осях. Можно рисовать один каркас (`wireframe`, см. цветную вкладку рис. 5(a)), а можно — поверхность (`surface`, см. цветную вкладку рис. 5(b)):

```
from matplotlib.pyplot import *
import numpy as np
from mpl_toolkits.mplot3d import Axes3D
fig = figure()
ax = Axes3D(fig)
u = np.linspace(0, 2 * np.pi, 100)
v = np.linspace(0, np.pi, 100)
x = 10 * np.outer(np.cos(u), np.sin(v))
y = 10 * np.outer(np.sin(u), np.sin(v))
z = 10 * np.outer(np.ones(np.size(u)), np.cos(v))
ax.plot_surface(x, y, z, color='grey')
savefig('3D.png')
show()
```

Функция `linspace` как и функция `arange` создаёт массив значений, каждое следующее из которых больше предыдущего на одну и ту же величину. Напомним, что `arange(start, stop, step)` принимает три основных аргумента: начало, конец, шаг; `linspace(start, stop, num, endpoint=True)` также принимает три обязательных аргумента: начало, конец и число значений, расположенных между ними, а кроме того у неё есть ещё необязательный четвёртый аргумент `endpoint`, который принимает логические значения. Если `endpoint=True` (это значение по умолчанию), то конечное значение `stop` будет включено в генерируемый диапазон и расстояние между соседними значениями будет равно `(stop-start)/(num-1)`, иначе оно не будет включено и расстояние между ними будет равно `(stop-start)/num`.

6.7 Учёт ошибок

В реальности эксперимент даже при максимальной точности измерений всегда вносит свою погрешность. Например, при снятии вольт-амперной характеристики (ВАХ) диода в трёх экспериментах получаются немного разные значения тока. С помощью `errorbar` можно построить среднее значение и отложить разброс (рис. 6.8). Для того, чтобы учесть это и указать возможный разброс вокруг значения, считаемого истинным, вводят планки погрешностей (функция `errorbar`), которые на кривой для полученных точек показывают своеобразный доверительный интервал:

```
from numpy import *
from matplotlib.pyplot import *
rcParams['font.sans-serif']=['Arial']
x = arange (0, 2.2, 0.2)
y1 = [0, 10, 24, 39, 55, 73, 87, 130, 140, 150, 200]
y2 = [0, 11, 25, 41, 58, 66, 94, 135, 140, 160, 170]
y3 = [0, 10, 24, 40, 57, 70, 90, 130, 145, 160, 180]
y = column_stack([y1, y2, y3])
errorbar(x, mean(y, axis=1), yerr=[std(y, axis=1),
         std(y, axis=1)], marker='.', color='black')
title('Вольт-амперная характеристика')
xlabel('Напряжение, В');
ylabel('Ток, мкА')
show()
```

В примере с помощью функции `arange()` был создан диапазон изменения напряжения от 0 В до 2 В с шагом 0.2 В. Далее результаты трёх серий измерений были представлены в виде трёх списков: `y1`, `y2`, `y3`. С помощью функции `column_stack` из нескольких одномерных массивов или списков одинаковой длины можно сделать двумерный массив. Далее вызывается функция `errorbar`. В качестве первого аргумента передаём диапазон изменения напряжения. В каче-

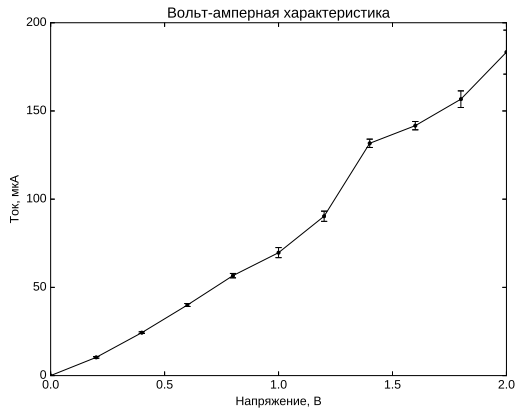


Рис. 6.8. ВАХ диода, усреднённая по трём экспериментам с отложенными планками погрешностей.

стве второго — среднее (`mean`) по трём измерениям значение тока. В качестве третьего (`yerr`) — разброс значений тока (`std`).

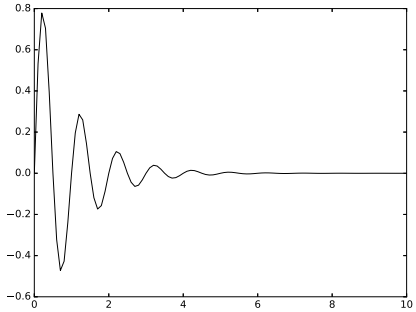
Рассмотренные в данном примере планки погрешностей симметричны относительно среднего значения, но существует возможность рисовать и несимметричные отклонения. Их можно задать в той же функции с помощью списка из двух разных последовательностей: первой для отрицательных отклонений, второй — для положительных.

6.8 Примеры построения графиков

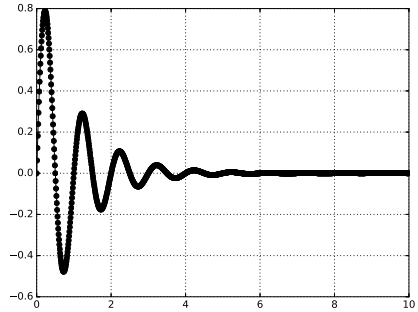
Пример задачи 17 (Затухающая синусоида, вариант 1) *Постройте график затухающей синусоиды* $e^{-x}\sin(2\pi x)$ *на отрезке* $[0; 10]$, *используя шаг по абсциссе, равный* 0.1.

Решение задачи 17

```
from numpy import *
from matplotlib.pyplot import *
x = arange(0, 10, 0.1)
f = exp(-x)*sin(2*pi*x)
plot(x, f)
show()
```



(a)



(b)

Рис. 6.9. Иллюстрация к задачам 17 — (a) и 18 — (b).

Пример задачи 18 (Затухающая синусоида, вариант 2) *Для построенного в предыдущем задании графика измените: цвет линии, тип линии и маркеров, шаг выборки данных, введите сетку и сохраните полученный график в файл.*

Решение задачи 18

```
from numpy import *
from matplotlib.pyplot import *
x = arange(0, 10, 0.01)
f = exp(-x)*sin(2*pi*x)
plot(x, f, '-o', color='black')
grid(True)
savefig('plot.png')
show()
```

Пример задачи 19 (Семейство затухающих синусоид) *Постройте семейство функций* $e^{-x}\sin(2\pi x+\phi_0)$ *на одном графике различными цветами при* $\phi_0 = 0$, $\phi_0 = \pi/6$ *и* $\phi_0 = \pi/3$. *Сделайте для графика легенду.*

Решение задачи 19

```
from numpy import *
from matplotlib.pyplot import *
x = arange(0, 10, 0.01)
```

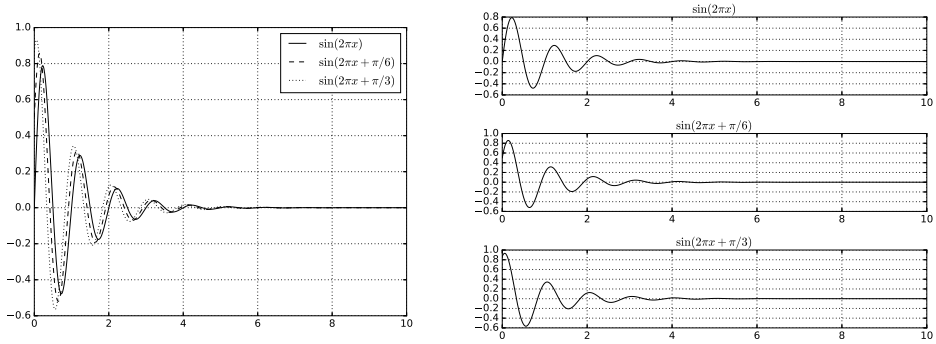
```
f1 = exp(-x)*sin(2*pi*x)
f2 = exp(-x)*sin(2*pi*x+pi/6)
f3 = exp(-x)*sin(2*pi*x+pi/3)
plot(x, f1, label=r'$\sin(2\pi␣x)$')
plot(x, f2, label=r'$\sin(2\pi␣x␣+␣\pi␣/6)$')
plot(x, f3, label=r'$\sin(2\pi␣x␣+␣\pi␣/3)$')
legend(loc='best')
grid(True)
show()
```

Пример задачи 20 (Семейство графиков с затухающими синусоидами) Перестройте графики так, чтобы каждая кривая располагалась на одном графике с помощью команды `subplot`, легенду уберите, а её текст переместите в название соответствующего графика. Графики расположите на полотне в один столбец.

Решение задачи 20

```
from numpy import *
from matplotlib.pyplot import *
x = arange(0, 10, 0.01)
f1 = exp(-x)*sin(2*pi*x)
f2 = exp(-x)*sin(2*pi*x+pi/6)
f3 = exp(-x)*sin(2*pi*x+pi/3)
subplot(3, 1, 1)
plot(x, f1)
title(r'$\sin(2\pi␣x)$')
grid(True)
subplot(3, 1, 2)
plot(x, f2)
title(r'$\sin(2\pi␣x␣+␣\pi␣/6)$')
grid(True)
subplot(3, 1, 3)
plot(x, f3)
title(r'$\sin(2\pi␣x␣+␣\pi␣/3)$')
grid(True)
tight_layout()
show()
```

Пример задачи 21 Постройте закрашенную контурную диаграмму и трёхмерный график для функции двух переменных (6.1), определённой в прямоугольной области ($x \in [-3; 3]$, $y \in [-3; 3]$):



(a) (b)

Рис. 6.10. Иллюстрация к задачам 19 — (a) и 20 — (b).

$$z = \frac{2xy}{x^2 + y^2} \tag{6.1}$$

Решение задачи 21

```
from numpy import *
from matplotlib.pyplot import *
from matplotlib.mlab import *
from mpl_toolkits.mplot3d import Axes3D
x = arange(-3, 3, 0.01)
y = arange(-3, 3, 0.01)
X, Y = meshgrid(x, y)
Z = 2*X*Y/(X**2+Y**2)
contourf(X, Y, Z)
savefig('plot.png')
fig = figure()
ax = Axes3D(fig)
ax.plot_surface(X, Y, Z)
savefig('3D.png')
show()
```

6.9 Задания на построение графиков

Для каждого из заданий данного раздела следует выполнить 1 вариант с номером $(n-1)\%m+1$, где n — номер в списке группы, а m — число заданий.

Задание 18 *Постройте графики следующих функций, используя шаг выборки данных по абсциссе из задания 17:*

1. x^2 на отрезке $x \in [-2; 2]$;
2. x^3 на отрезке $x \in [-2; 2]$;
3. x^4 на отрезке $x \in [-2; 2]$;
4. $\cos(2\pi t)$ на отрезке $t \in [-10; 10]$;
5. $\frac{1}{t}\cos(2\pi t)$ на отрезке $t \in [1; 10]$;
6. $e^{-t}\cos(2\pi t)$ на отрезке $t \in [-10; 10]$;
7. $4\sin(\pi t + \pi/8) - 1$ на отрезке $t \in [-10; 10]$;
8. $2\cos(t - 2) + \sin(2 * t - 4)$ на отрезке $t \in [-20\pi; 10\pi]$;
9. $\ln(x + 1)$ на отрезке $x \in [0; e - 1]$;
10. $\log_2(|x|)$ на отрезке $x \in [-4; 4]$ за исключением точки $x = 0$;
11. 2^x на отрезке $x \in [-2; 2]$;
12. e^x на отрезке $x \in [-2; 2]$;
13. 2^{-x} на отрезке $x \in [-2; 2]$;
14. $\sqrt[3]{x}$ на отрезке $x \in [1; 125]$;
15. $\sqrt[5]{x}$ на отрезке $x \in [1; 32]$.

Задание 19 Для построенного в рамках задания 18 графика измените:

- цвет линии;
- тип линии и маркеров;
- шаг выборки данных.

Далее введите сетку. Сохраните полученный график в файл, опробуйте сохранять файл в разных форматах: `png`, `pdf`, `jpg`, `eps`.

Задание 20 Постройте семейство функций на одном графике различными цветами:

1. степенные многочлены с целыми степенями от 1 до 6 на отрезке $[-1; 1]$;
2. синусоиды $y = \sin(\omega t)$ с частотами $\omega = 2\pi$, $\omega = 3\pi$, ..., $\omega = 8\pi$ на отрезке $t \in [-1; 1]$;
3. синусоиды $y = \sin(2\pi t + \phi_0)$ с начальными фазами $\phi_0 = 0$, $\phi_0 = \pi/6$, ..., $\phi_0 = 5\pi/6$ на отрезке $t \in [-1; 1]$;

4. логарифмические функции $\log_2(x)$, $\ln(x)$ и $\log_{10}(x)$ на отрезке $x \in [1; 10]$;
5. гиперболические функции $\mathrm{sh}(x)$, $\mathrm{ch}(x)$ и $\mathrm{th}(x)$ на отрезке $x \in [-10; 10]$, для их вычисления воспользуйтесь их выражением через экспоненту.

Задание 21 Для построенного в задании 20 графика сделайте сетку и легенду. Перестройте графики так, чтобы каждая кривая располагалась на одном графике с помощью команды `subplot`, легенду уберите, а её текст переместите в название соответствующего графика. Графики расположите на полотне:

- в одни столбец;
- в два столбца;
- в 3 столбца;
- в одну строку.

Перестройте графики из задания каждый в своём окне. Сделайте так, чтобы эти графики автоматически сохранялись каждый в свой файл.

Задание 22 Постройте круговую диаграмму, которая показывала бы доли от общего числа студентов вашей группы, сдавших сессию на:

1. одни пятёрки,
2. пятёрки и четвёрки,
3. с тройками, но без задолжностей,
4. с задолжностями, сумевших в итоге пересдать,
5. несдавших и отчисленных (если такие имеются).

Задание 23 Постройте закрашенную контурную диаграмму и трёхмерный график для следующих функций двух переменных, определённых в прямоугольной области $x \in [-3; 3]$, $y \in [-3; 3]$:

1. $z = x^2 + y^2$,
2. $z = x^2 - y^2$,
3. $z = x^3 + y^3$,
4. $z = x^3 - y^3$,
5. $z = x^2 - y^2 + x$,
6. $z = x^2 - y^2 + y$,

7. $z = x^2 + y^2 + x$,
8. $z = x^2 + y^2 + y$,
9. $z = \sin(xy)$,
10. $z = \cos(xy)$,
11. $z = \mathrm{tg}(xy)$,
12. $z = xy$,
13. $z = x - \sin(xy)$,
14. $z = x + \cos(xy)$,
15. $z = \sqrt{x^2 + y^2}$.

Построенные графики сохраните в файлы с расширением `png`.

Глава 7

Библиотеки, встроенные в numpy

Исторически `numpy` имеет в своём составе 3 библиотеки численных методов: `linalg` для задач линейной алгебры, `fft` для выполнения быстрого Фурье-преобразования и `random` для генерации случайных чисел. Хотя основным модулем для научных и инженерных вычислений стал `scipy`, построенный на базе `numpy`, поддержка этих трёх библиотек сохраняется из соображений обратной совместимости со старыми программами. Освоение возможностей этих библиотек позволяет писать много полезных прикладных программ.

7.1 Элементы линейной алгебры

Библиотека `linalg` даёт возможность вычислять определители матриц, решать системы линейных уравнений и задачу наименьших квадратов, производить QR и SVD разложения. Вот пример простой программы, решающей систему линейных уравнений с помощью функции `solve` и вычисляющей определитель матрицы с помощью функции `det` из `linalg`:

```
from numpy import *
A = array([[2, 3], [-1, 5]])
b = array([-7, -16])
x = linalg.solve(A, b)
print(x)
D = linalg.det(A)
print(D)
```

Результаты её работы легко проверить вручную и получить искомые решения:

```
[ 1. -3.]
13.0
```

Синтаксис функций `solve` и `det` простой и очевидный для тех, кто привычен к записи систем уравнений в матричной форме вида $\hat{A}\mathbf{x} = \mathbf{b}$. Функция `solve` требует 2 параметра: матрицу коэффициентов $\hat{A}$ и вектор свободных членов (правая

часть системы уравнений) **b**. Результат выполнения функции — вектор искомых значений **x**. Функция `det` требует один параметр — матрицу, определитель которой следует отыскать.

Кроме определителя для матриц можно вычислить собственные значения и собственные векторы матрицы (`linalg.eig(a)`), а также норму вектора или оператора (`linalg.norm(x[, ord, axis])`). Уже реализованы все основные методы разложения матриц:

- `inalg.cholesky(a)` — разложение Холецкого; `linalg.qr(a[, mode])` — QR разложение; `linalg.svd(a[, full_matrices, compute_uv])` — сингулярное разложение; `linalg.lstsq(a, b)` — метод наименьших квадратов.

Используя возможности встроенной в модуль `numpy` библиотеки `linalg`, попробуем решить популярную задачу аппроксимации сток-затворной характеристики полевого транзистора полиномиальной (параболой) и кусочно-линейной функцией. В данном случае, под аппроксимацией будем понимать замену измеренных пар значений `(напряжение, ток)` некоторою функцией `ток(напряжение)`.

Предположим, что у нас есть результаты измерения тока стока i_c (список `y`) при изменении напряжения на затворе $U_{\text{ЗИ}}$ (массив `x`). Данная вольт-амперная характеристика (ВАХ) показана на рис. 7.1. Программа для её отображения будет иметь следующий вид:

```
from numpy import *
from matplotlib.pyplot import *
from matplotlib import rcParams
rcParams['font.sans-serif'] = 'Liberation␣Sans'
x = arange(0, -4.5, -0.5)
y = [50, 35, 22, 11, 4, 0, 0, 0, 0]
plot(x, y, 'o', color='grey')
title('Сток-затворная характеристика')
xlabel(r'$U_{ЗИ}, В$', fontsize=18)
ylabel(r'$i_{c}, мкА$', fontsize=18)
xlim(-5, 0.5)
ylim(-10, 60)
show()
```

Кусочно-линейная аппроксимация заменяет ВАХ транзистора двумя отрезками:

$$i_{\text{с}} = \begin{cases} \alpha_0 + \alpha_1 U_{\text{ЗИ}} \text{ при } U_{\text{ЗИ}} > U_{\text{отс}} \\ 0 \text{ при } U_{\text{ЗИ}} < U_{\text{отс}} \end{cases} \tag{7.1}$$

Используя метод наименьших квадратов (`lstsq`) и функцию матричного умножения (`dot`), произведём аппроксимацию этих двух участков ВАХ прямыми линиями (рис. 7.2(а)). Разделим все измерения на два отрезка от 0 В до -1.5 В ($b = 0$; $e = 4$) и от -2.5 В до -4 В ($b = 5$; $e = 9$). Выведем значения коэффициентов аппроксимации, для чего добавим следующий код перед выводом изображения на экран функцей `show()`:

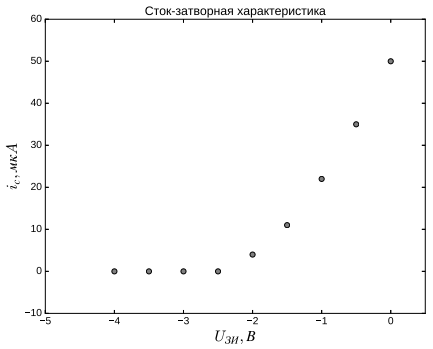


Рис. 7.1. Сток-затворная характеристика полевого транзистора (ВАХ) — результаты эксперимента.

```
b=0; e=4; r=e-b
a=ones((r, 2))
a [:, 1] = x[1:r+1]
result=linalg.lstsq(a, y[b:e])
ya1=dot(a, result[0])
plot (x[b:e], ya1, color='black')
print (result[0])
b=5; e=9; r=e-b
a=ones((r, 2))
a [:, 1] = x[1:r+1]
result=linalg.lstsq(a, y[b:e])
ya1=dot(a, result[0])
plot (x[b:e], ya1, color='black')
print (result[0])
```

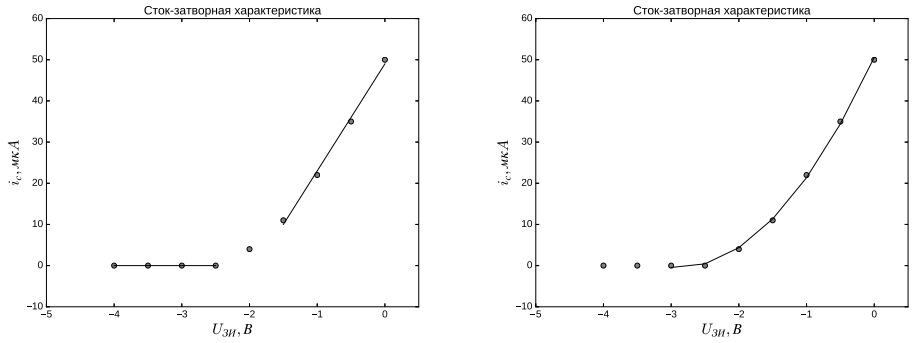
Результаты:

```
[ 62.  26.]
[ 0.  0.]
```

Функция `lstsq` возвращает кортеж, нулевой элемент которого — коэффициенты модели, первый элемент — сумма квадратов ошибок аппроксимации (разниц между реальными и аппроксимированными значениями). Теперь, используя те же функции из модуля `numpy`, произведём аппроксимацию полиномом второго порядка (7.2) для участка от 0 В до −3 В (`b=0; e=7`) (рис. 7.2(b)).

$$i_c = \alpha_0 + \alpha_1 U_{ЗИ} + \alpha_2 U_{ЗИ}^2 \tag{7.2}$$

Для получения параболической аппроксимации (см. рис. 7.2(b)) нужно заменить код из предыдущего листинга на следующий:



(a) (b)

Рис. 7.2. Различные подходы к аппроксимации вольт-амперной характеристики полевого транзистора: (a) — кусочно-линейная, (b) — параболическая.

```
b=0; e=7; r=e-b
a=ones((r, 3))
a [:, 1] = x[1:r+1]
a [:, 2] = x[1:r+1]**2
result=linalg.lstsq(a, y[b:e])
ya1=dot(a, result[0])
plot(x[b:e], ya1, color='black')
print(result[0])
```

Результат:

```
[ 69.71428571  41.38095238   6.0952381 ]
```

7.2 Быстрое преобразование Фурье

Библиотека `fft` позволяет быстро и легко делать все возможные варианты преобразования Фурье над массивами. Многие реализации преобразования Фурье до сих пор поддерживают только массивы длиною в 2^N значений, где N — целое число, иначе массив либо обрезается до ближайшей степени двойки, либо удлиняется нулями или периодически. Функции библиотеки `fft` поддерживают длины массивов, являющиеся степенями 2, 3, 5 и 7 или произвольными их произведениями. Самые востребованные методы библиотеки `fft` — `rttf` и `irfft` позволяют произвести прямое Фурье-преобразование над действительнозначными данными и обратное Фурье-преобразование над комплексными данными, для которых половина значений является комплексно-сопряжёнными, причём сопряжённые значения не включаются в обрабатываемый массив. Далее приведём пример расчёта Фурье-преобразования синусоиды:

```
from numpy import *
t = arange(0, 1, 0.1)
x = sin(2*pi*t)
fx = fft.rfft(x)
print (fx)
```

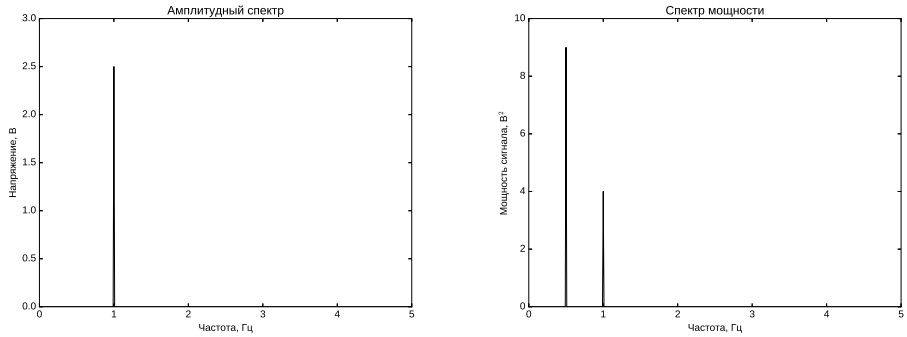
Как и положено нормальной синусоиде, Фурье-образ которой рассчитан на целом числе периодов, наша имеет только одну существенно отличную от нуля компоненту:

```
[ -3.33066907e-16 +0.00000000e+00j  -1.72084569e-15
-5.00000000e+00j 7.35173425e-16 +4.55540506e-16j
-6.24151122e-16 -1.76490554e-15j 1.83186799e-15
+4.44089210e-16j  -1.11022302e-16 +0.00000000e+00j]
```

Все остальные значения порядка 10^{-15} или ещё меньше следует признать нулями с точностью вычислений.

Часто требуется построить спектр мощности или амплитудный спектр сигнала. С помощью numpy эта задача легко решается. Проще всего построить так называемую *периодограмму* — неусреднённую оценку спектра по одной реализации. Для получения периодограммы мощности достаточно взять квадрат модуля Фурье-образа. Чтобы мощность соответствовала коэффициентам при гармониках в исходном сигнале, нужно учесть нормировку. Дело в том, что большинство библиотечных функций Фурье-преобразования производят необходимое для алгоритма суммирование, но не нормируют результат, поскольку в ряде случаев (например, при реализации преобразования Гильберта, для фильтрации, для расчёта функции когерентности) это излишне, а вычислительные ресурсы экономятся. Поэтому нормировку необходимо произвести вручную. Для реализации Фурье-преобразования в numpy необходимая нормировка — половина длины исходного ряда. Для начала можно построить амплитудный спектр моногармонического сигнала (рис. 7.3(а)):

```
from numpy import *
from matplotlib.pyplot import *
rcParams['font.sans-serif']=['Liberation␣Sans']
dt = 0.1
t = arange(0, 100, dt)
x = 2.5*sin(2*pi*t)
fx = fft.rfft(x) / (len(x)/2)  # нормировка на половину длины ряда
fn = 1/(2*dt)  # частота Найквиста
freq = linspace(0, fn, len(fx)) # массив частот
plot(freq, abs(fx), color='black')
title('Амплитудный спектр')
xlabel('Частота, Гц');
ylabel('Напряжение, В')
ylim([0, 3])
```



(a) (b)

Рис. 7.3. Амплитудный спектр моногармонического сигнала — (a) и спектр мощности бигармонического сигнала — (b).

```
savefig('FFT.png')
show()
```

Здесь частота Найквиста — максимально разрешимая частота в спектре, равная половине частоты выборки. Чтобы построить спектр, нужно рассчитать массив частот, для которых с помощью преобразования Фурье получены значения амплитуд гармоник. Это несложно сделать с помощью функции `linspace` из `numpy`, если учесть, что минимальная частота равна 0, максимальная — частота Найквиста, а число частот равно числу амплитуд.

Если умножить Фурье образ синусоиды на $e^{i\varphi}$, а затем сделать обратное преобразование, получится сигнал, сдвинутый по фазе относительно исходного на φ. Чаще всего оказывается нужно сдвинуть сигнал на $\pi/2$ или $-\pi/2$. По правилу Эйлера $e^{-i\pi/2} = \cos(-\pi/2) + i\sin(-\pi/2) = -i$, то есть мы сдвинули фазу синусоиды на и получили вместо синуса минус косинус:

```
fy = fx*-1j
y = fft.irfft(fy)
print(y)
```

Вывод:

```
[-1.  -0.80901699 -0.30901699  0.30901699
0.80901699  1.  0.80901699  0.30901699 -0.30901699 -0.80901699]
```

Одинокая синусоида единичной амплитуды — не очень интересный пример. Рассчитаем и построим график бигармонического сигнала, состоящего из двух синусоид разных амплитуд (рис. 7.3(b)):

```
from numpy import *
```

```
t = arange(0, 2, 0.1)
x = 2*sin(2*pi*t) + 3*cos(pi*t)
Px = abs(fft.rfft(x)/(0.5*len(t)))**2
print(Px)
```

Получаем коэффициент 9 при 2 члене, что соответствует квадрату амплитуды косинуса, и 4 при третьем — квадрат амплитуды синуса, остальные коэффициенты — нули:

```
[1.97215226e-33   9.00000000e+00   4.00000000e+00   7.29696337e-31
 2.30741815e-31   1.28189897e-31   1.04524070e-31   3.82597539e-31
 3.86541844e-31   1.28189897e-31   1.97215226e-33]
```

Строим спектр мощности, увеличив длину ряда:

```
from numpy import *
from matplotlib.pyplot import *
rcParams['font.sans-serif']=['Liberation␣Sans']
dt = 0.1
t = arange(0, 100, dt)
x = 2*sin(2*pi*t) + 3*cos(pi*t)
Px = abs(fft.rfft(x)/(0.5*len(t)))**2
fn = 1/(2*dt)
plot(linspace(0, fn, len(Px)), Px, color='black')
title('Спектр мощности')
xlabel('Частота, Гц');
ylabel(r'Мощность сигнала, B$^2$')
ylim([0, 10])
show()
```

7.3 Генерация случайных чисел

Кроме стандартного модуля `random` есть библиотека `random` из модуля `numpy`, которая позволяет генерировать псевдослучайные числа с самыми различными распределениями. Самыми распространёнными являются равномерное, которому соответствует функция `uniform`, и нормальное — функция `normal`. Обе эти функции имеют одинаковый синтаксис: сначала идут параметры распределения, потом — число значений, которое нужно сгенерировать. Если это число не указать, получится не массив, а одно случайное число. Для равномерного распределения на отрезке $[a; b]$ параметрами являются величины a и b, для нормального со средним μ и дисперсией σ^2 — величины μ и σ.

```
from numpy import *
x = random.uniform(-2, 1, 10)
print(x)
y = random.normal(5, 0.5, 10)
```

```
print(y)
```

Вывод программы:

```
[-0.41038517  0.33622363 -0.24998561  0.58409914  0.96982331
-1.11356063 -0.43092171 -0.92418957 -0.49456604 -0.61474565]
[ 4.89630265  3.68213872  4.97692322  4.4682948   5.19711419
4.87308781  5.51078962  5.68588492  4.65777752  5.66324449]
```

Кроме генерации непрерывно распределённых чисел библиотека `random` поддерживает также множество дискретных распределений. Самое простое — равномерное дискретное, когда вероятности всех событий равны, задаётся с помощью функции `randint`:

```
from numpy import *
x = random.randint(-10, 10, 10)
print(x)
y = random.permutation(x)
print(y)
```

Синтаксис `randint` полностью повторяет таковой у `uniform`. В приведённом примере использована ещё одна весьма полезная на практике функция `permutation`. Она случайно тасует элементы массива, но не меняет оригинальный массив, как это видно из сравнения первой и второй строк вывода приведённой программы, а вместо этого создаёт новый:

```
[-9 -1 -7  2  8 -8 -4  5 -9  5]
[-9 -9  2 -4  5 -7 -1  5  8 -8]
```

В заключение раздела хотелось бы сказать, что `numpy` обладает гораздо более мощными и гибкими возможностями, чем это можно проиллюстрировать в рамках приведённого краткого ознакомительного курса. Но красота `numpy` и его удобство становятся очевидными только по мере использования.

7.4 Примеры решения заданий

Пример задачи 22 (Определитель матрицы) *Найдите определитель матрицы. Матрицу возьмите из текстового файла, созданного при выполнении задания №15.*

Решение задачи 22

```
from numpy import *
M = loadtxt('Matrix.txt')
print(linalg.det(M))
```

Пример задачи 23 (Система линейных уравнений) Решите систему линейных уравнений, матрицу коэффициентов и столбец свободных членов прочитайте из текстовых файлов, созданных в задании №15. Запишите в новый текстовый файл полученные корни.

Решение задачи 23

```
from numpy import *
M = loadtxt('Matrix.txt')
V = loadtxt('Vector.txt')
print(linalg.solve(M, V))
```

Пример задачи 24 (Аппроксимация параболою) Сгенерируйте параболу $y = x^2 - x - 6$ на отрезке $[-6; 6]$. Прибавьте к ней белый шум с параметрами $(0; 2)$. Аппроксимируйте её полиномом второй степени. Оцените ошибку аппроксимации. Постройте график (рис. 7(а)).

Решение задачи 24

```
from numpy import *
from matplotlib.pyplot import *
x = arange(-6, 6, 0.1) # диапазон
y = x**2 - x - 6 # парабола
r = random.normal(0, 2, len(x)) # белый шум
z = y + r
a = ones((len(x), 3))
a[:, 1] = x
a[:, 2] = x**2
result = linalg.lstsq(a, z)
za = dot(a, result[0]) # аппроксимирующая кривая
plot(x, z, 'o', color='red')
plot(x, za, color='blue')
print(result[1]/len(x)) # ошибка аппроксимации
show()
```

Значение ошибки аппроксимации должно быть порядка квадрата стандартного отклонения шума.

Пример задачи 25 (Погрешности аппроксимации) Сгенерируйте 3 ряда y, как это описано в предыдущем задании, пусть ряды отличаются реализациями шума. Для каждого x таким образом будет доступно по 3

значения y. По этим значениям рассчитайте для каждого x среднее значение $\bar{y}$ и среднеквадратичное отклонение от среднего σ_x. С использованием полученных рядов и постройте график погрешностей результатов (errorbar) (рис. 7(b)).

Решение задачи 25

```
from numpy import *
from matplotlib.pyplot import *
x = arange(-6, 6, 0.1)
y = x**2-x-6
r = random.normal(0, 2, len(x))
y1 = y + r
r = random.normal(0, 2, len(x))
y2 = y + r
r = random.normal(0, 2, len(x))
y3 = y + r
y = column_stack([y1, y2, y3])
errorbar(x, mean(y, axis=1), yerr=[std(y, axis=1),
         std(y, axis=1)], marker='.', color = 'green',
                                        ecolor='blue')
savefig('plot.png')
show()
```

Пример задачи 26 (Генерация массивов случайных чисел) Сгенерируйте случайные векторы из n действительных значений с равномерным и нормальным распределением, а также из n целых чисел.

Решение задачи 26

```
n = int(input('Введите количество значений: '))
a = input('Введите диапазон равномерного распределения: ').split()
x = random.uniform(float(a[0]), float(a[1]), n)
print(x)
b = input('Введите параметры нормального распределения: ').split()
y = random.normal(float(b[0]), float(b[1])**0.5, n)
print(y)
c = input('Введите диапазон для дискретного распределения: ').split()
z = random.randint(int(c[0]), int(c[1]), n)
print(z)
```

Возможный вывод программы (при каждом запуске будут различные случайные числа):

```
Введите количество значений: 8
Введите диапазон равномерного распределения: 1 10
[ 4.59727241  5.25897018  7.05872105  6.53497311
2.79842143  6.9183058  5.47890825  8.6876828 ]
Введите параметры нормального распределения: 3 9
[ 9.2115671   2.97903395  2.91634906  3.67303643
0.29596935  6.45306148  3.50505498  3.44637582]
Введите диапазон дискретного распределения: -10 10
[ 7  2 -2 -2  6 -8  3  1]
```

Чтобы вводить сразу несколько значений с клавиатуры, в данном примере был использован метод `split`, разделяющий строку на подстроки. При выполнении задания можно просто каждый необходимый параметр считывать новым `input()`.

Пример задачи 27 (Случайные перестановки) *Сгенерируйте массив-лесенку длины n. Случайно перемешайте его. На экран выведите изначальный и перемешанный массивы.*

Решение задачи 27

```
from numpy import *
n = int(input('Размер массива: '))
x = arange(n, 0, -1)
print(x)
y = random.permutation(x)
print(y)
```

Возможный вывод программы (третья строка будет меняться от запуска к запуску):

```
Размер массива: 10
[10  9  8  7  6  5  4  3  2  1]
[ 7  5  6  1  2  8 10  4  3  9]
```

Пример задачи 28 (Периодограмма синуса) *Рассчитайте и постройте периодограмму для функции $\sin(x)$ на отрезке $x \in [\pi; \pi]$.*

Решение задачи 28 Результат можно увидеть на рис. 8(а). Грубость графика обусловлена малым объёмом данных.

```
from numpy import *
from matplotlib.pyplot import *
t = arange(-pi, pi, pi/12)
x = sin(2*pi*t)
```

```
Px = abs(fft.rfft(x)/(0.5*len(t)))**2
fn = 1/(2*pi/12)
freq = linspace(0, fn, len(Px))
grid(True); plot(freq, Px, color='red')
show()
```

Пример задачи 29 (Сложный шум) Сгенерируйте случайный процесс, представляющий собою сумму равномерно распределённых на отрезке $[-10; 10]$ случайных величин и нормально распределённых случайных величин с параметрами $(0; 1)$, длиною в 10000 значений. Постройте гистограмму его распределения.

Решение задачи 29 Возможный результат можно увидеть на рис. 8(b). От запуска к запуску картинка будет несколько варьировать.

```
from numpy import *
from matplotlib.pyplot import *
n = 10000
x = random.uniform(-10, 10, n)
y = random.normal(0, 1, n)
z = x + y
hist(z, bins=100, normed=True, color='green')
show()
```

7.5 Задания на использование встроенных библиотек numpy

Для заданий, содержащих большое число вариантов (от 4 и более) следует выполнить 1 вариант с номером $(n-1)\%m+1$, где n — номер в списке группы, а m — число заданий.

Задание 24 Найдите определитель матрицы. Матрицу возьмите из текстового файла, созданного ранее, либо у преподавателя.

Задание 25 Решите систему линейных уравнений. Матрицу коэффициентов и столбец свободных членов прочитайте из текстовых файлов, созданных ранее. Запишите в новый текстовый файл полученные корни.

Задание 26 Сгенерируйте набор значений заданной функции с шумом. Аппроксимируйте его полиномом второй степени. Оцените ошибку аппроксимации. Постройте график. Функции:

1. парабола $y = x2 - x - 6$ на отрезке $[-4; 4]$ с белым шумом, распределённым по нормальному закону с параметрами $(0; 1)$;

2. парабола $y = x2 - x - 6$ на отрезке $[-4; 4]$ с белым шумом, распределённым по равномерному закону на отрезке $[-10; 10]$;

3. парабола $y = x2 - x - 6$ на отрезке $[-4; 4]$ с белым шумом, распределённым по закону χ^2 (`random.chisquare`) с параметрами $(6; n)$, где 6 — количество степеней свободы шума, а n — длина ряда y;

4. парабола $y = 5x^2 - 4x + 1$ на отрезке $[-6; 6]$ с белым шумом, распределённым по нормальному закону с параметрами $(0; 3)$;

5. парабола $y = 5x^2 - 4x + 1$ на отрезке $[-6; 6]$ с белым шумом, распределённым по равномерному закону на отрезке $[-1; 1]$;

6. парабола $y = 5x^2 - 4x + 1$ на отрезке $[-6; 6]$ с белым шумом, распределённым по закону χ^2 (`random.chisquare`) с параметрами $(6; n)$, где 6 — количество степеней свободы шума, а n — длина ряда y;

7. парабола $y = x^2 + 5$ на отрезке $[-10; 10]$ с белым шумом, распределённым по нормальному закону с параметрами $(0; 1)$;

8. парабола $y = x^2 + 5$ на отрезке $[-10; 10]$ с белым шумом, распределённым по равномерному закону на отрезке $[-10; 10]$.

Задание 27 Сгенерируйте 5 рядов y, как это описано в предыдущем задании, пусть ряды отличаются реализациями шума. Для каждого x таким образом будет доступно по 5 значений y. По этим значениям рассчитайте для каждого x соответствующее ему среднее значение $\bar{y}$ и среднеквадратичное отклонение от среднего σ_y. С использованием полученных рядов $\bar{y}(x)$ и $\sigma_y(x)$ постройте график средних с планками погрешностей (`errorbar`).

Задание 28 Сгенерируйте случайные векторы из 10, 30 и 200 значений:

1. с равномерным распределением на отрезке $[-0.5; 0.5]$;

2. с нормальным распределением с параметрами $\mu = 1$, $\sigma = 0.5$;

3. из целых чисел в диапазоне $[0; 10]$.

Задание 29 Сгенерируйте и случайно перемешайте:

1. массив-диапазон, покрывающий полуинтервал $[0; 10)$ с шагом 0.5;

2. массив-диапазон из целых чисел от 0 до 19;

3. массив из 10 чисел, первые 5 из которых нули, вторые 5 — единицы;

4. массив длины 10, в котором изначально в начале и в конце было по 2 тройки. А середине — пятёрки;

5. массив из 4 нулей, 4 единиц и 4 двоек;
6. массив из 15 нулей и 1 единицы;
7. массив-пирамиду длины 11 из целых чисел, где среднее число — самое большое, стоящие рядом с ним на 1 меньше, следующие по очереди от середины ещё на 1 меньше и т. д., значение среднего числа задайте сами;
8. массив, полученный в результате табулирования синусоиды.

Выведите на экран сначала неизменённый массив, потом — перемешанный.

Задание 30 Рассчитайте и постройте периодограмму — оценку спектра мощности:

1. сигнала $y(x)$, полученного по формуле $4\sin(\pi x + \pi/8) - 1$ на отрезке $[-10; 10]$ с шагом 0.05;
2. сигнала $y(x)$, полученного по формуле $2\cos(x - 2) + \sin(2x - 4)$, на отрезке $[-20\pi; 10\pi]$ с шагом $\pi/20$;
3. нормального шума, параметры выберите сами;
4. равномерного шума, параметры выберите сами.

Задание 31 Сгенерируйте случайный процесс длиною в 10000 значений и постройте гистограмму его распределения для следующих рядов:

1. равномерный шум с параметрами $(0, 1)$;
2. равномерный шум с параметрами $(-4, 10)$;
3. равномерный шум с параметрами $(0.5, 0.6)$;
4. равномерный шум с параметрами $(-a, a)$, где a — случайное равномерно распределённое число из диапазона $[0; 1]$;
5. равномерный шум с параметрами $(-a, 2a)$, где a — случайное равномерно распределённое число из диапазона $[1; 10]$;
6. нормальный (гауссов) шум со стандартными параметрами: $(0, 1)$;
7. нормальный шум с параметрами $(-2, 0.25)$;
8. нормальный шум с параметрами $(1, 2.5)$;
9. нормальный шум с нулевым средним и среднеквадратичным отклонением σ, где σ — число, равномерно распределённое в диапазоне $[0; 1]$;

10. нормальный шум с параметрами μ, σ, где μ есть нормально распределённое число со стандартными параметрами $(0, 1)$, а σ — число, равномерно распределённое в диапазоне $[1, 10]$;

11. процесс, представляющий собою сумму двух независимых величин, распределённых равномерно на интервале $[-1; 1]$;

12. процесс, представляющий собою сумму 3 независимых величин, равномерно распределённых на интервале $[-1; 1]$;

13. процесс, представляющий собою сумму двух нормально распределённых случайных величин с единичною дисперсией, первое из которых имеет среднее $-e$, а второе имеет среднее e;

14. процесс, представляющий собою сумму большого числа (например, 30) равномерно распределённых на отрезке $[-0.1; 0.1]$ случайных величин;

15. процесс, представляющий собою сумму большого числа (например, 30) нормально распределённых случайных величин с параметрами $\mu = 0$, $\sigma = 0.1$.

Учебное издание

Серия «Библиотека ALT»

Сысоева Марина Вячеславовна
Сысоев Илья Вячеславович

Программирование для «нормальных» с нуля на языке Python
В двух частях
Часть 1

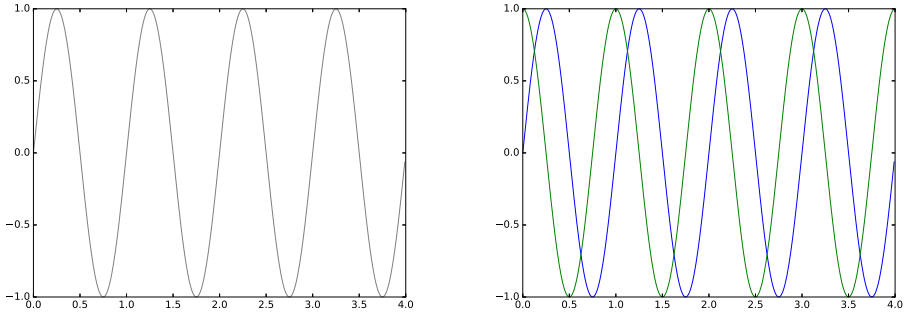
Ответственный редактор: В. Л. Черный
Оформление обложки: А. С. Осмоловская
Вёрстка: В. Л. Черный

Издание доступно в РИНЦ по адресу: `https://elibrary.ru`

ООО «Базальт СПО»
Адрес для переписки: 127015, Москва, а/я 21
Телефон: (495) 123-47-99. E-mail: sales@basealt.ru
`http://basealt.ru`

Подписано в печать 02.03.18. Формат 70x100/16.
Гарнитура Computer Modern. Печать офсетная. Бумага офсетная.
Усл. печ. л. 14.3[+0.33]. Уч.-изд. л. 11.77 Тираж 999 экз. Изд. номер. 044
Издательство ООО «МАКС Пресс»
Лицензия ИД N 00510 от 01.12.99 г.
119992, ГСП-2, Москва, Ленинские горы, МГУ имени М. В. Ломоносова,
2-й учебный корпус, 527 к.
Тел. 8(495)939-3890/91. Тел./Факс 8(495)939-3891.

По вопросам приобретения обращаться: ООО «Базальт СПО»
(495)123-47-99 E-mail: sales@basealt.ru http://basealt.ru



(a) (b)

Рис. 1. График синусоиды серыми линиями — (a) и синусоиды (синими) и косинусоиды (зелёными) линиями — (b).

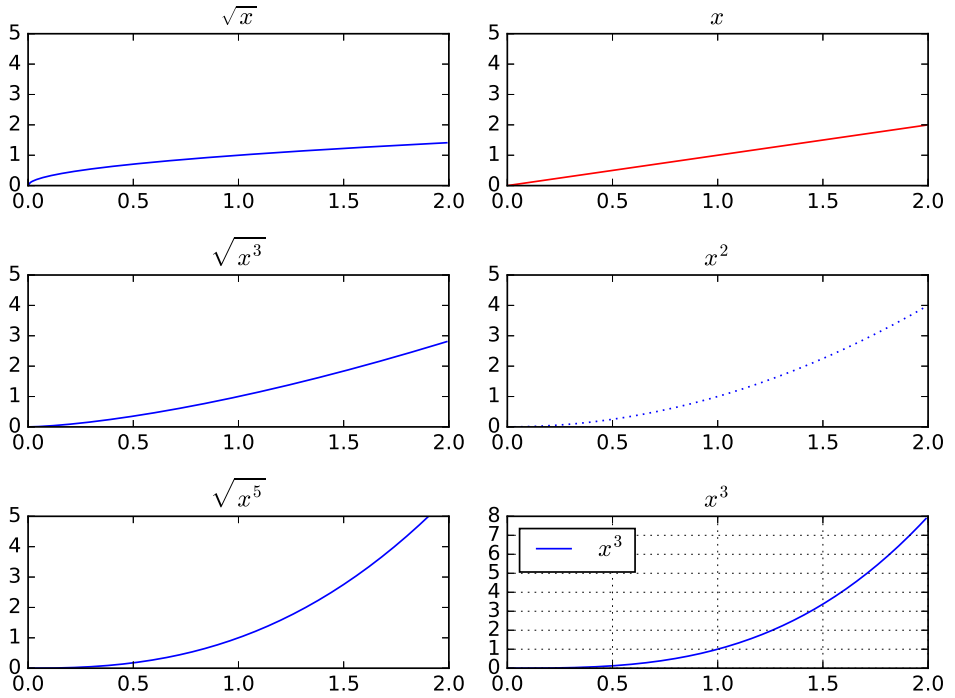
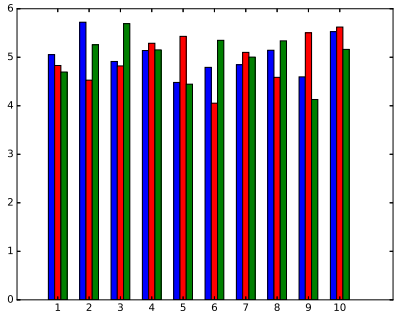
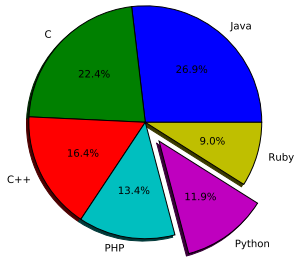


Рис. 2. Пример построения нескольких графиков на одном полотне.

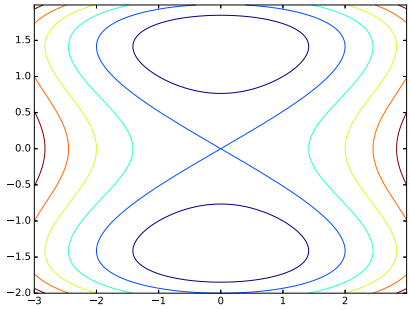


(a)

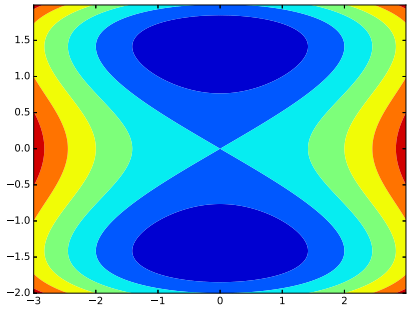


(b)

Рис. 3. Пример столбцовой (a) и круговой (b) диаграмм.



(a)



(b)

Рис. 4. Пример построения контурных диаграмм: (a) — использованы линии, (b) — использована заливка.

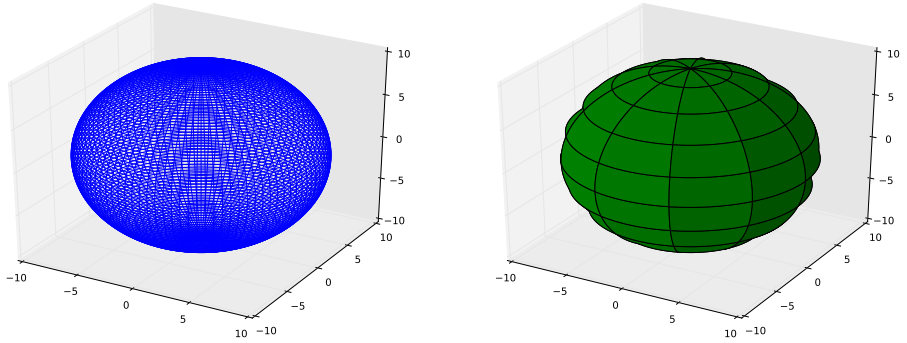


Рис. 5. 3D-каркас (a) и 3D-поверхность (b).

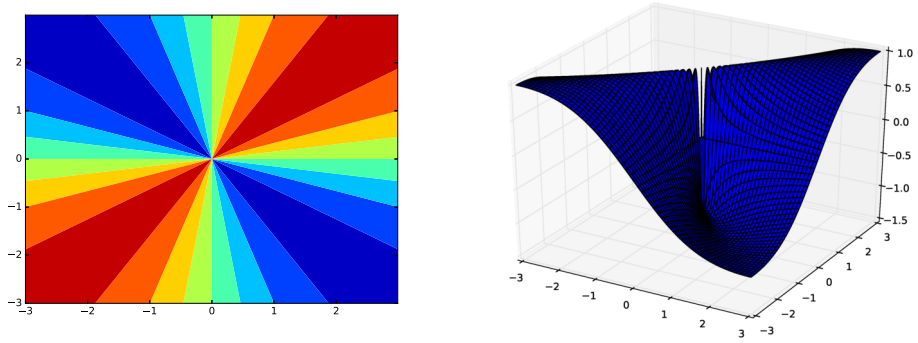
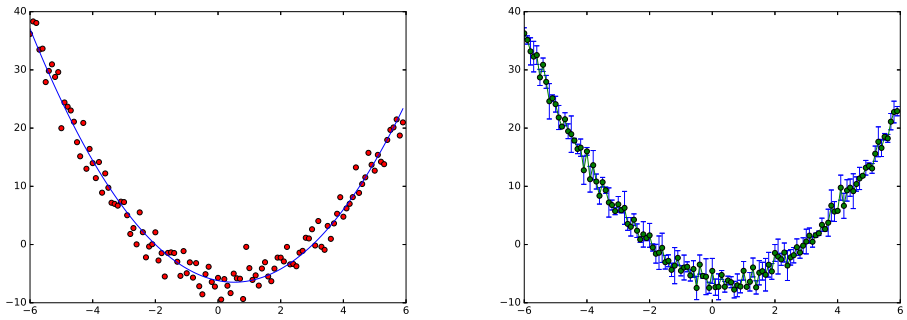
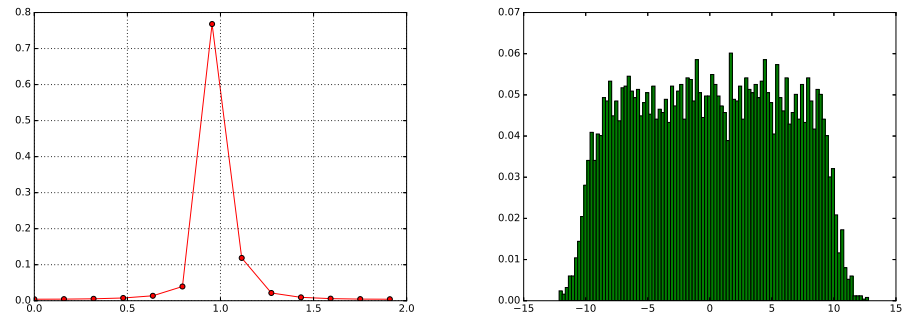


Рис. 6. Иллюстрация к задаче 21.



(a) (b)

Рис. 7. Иллюстрации к задачам 24 — (a) и 25 — (b). На рис. (a) представлена зависимость $y_i = x_i^2 - x_i - 6 + \xi_i$ (серые точки), где ξ_i — нормально распределённые случайные числа (шум) с нулевым средним и среднеквадратичным отклонением 2, и её аппроксимирующая функция (чёрная кривая), рассчитанная методом наименьших квадратов. На рис. (b) — зависимость $\langle y_k(x) \rangle_{k=1,2,3}$ с разбросом ошибок, построенная по 3 экспериментам (k — номер эксперимента), исходная зависимость сгенерирована по формуле $y_i = x_i^2 - x_i - 6 + \xi_i$, причём для каждого эксперимента реализация шума ξ своя.



(a) (b)

Рис. 8. Спектр мощности синусоиды, построенный по 1 периоду при шаге выборки $\pi/12$ — (a) и гистограмма сложного (суммы равномерного и нормального) распределения — (b).