

В серии:

Библиотека АЛТ

М. В. Сысоева, И. В. Сысоев

Программирование для «нормальных» с нуля на языке Python

В двух частях

Часть 1

Учебник

*Второе издание,
исправленное и дополненное*

Москва
Базальт СПО
МАКС Пресс
2023

УДК 004.43(075.8)
ББК 22.1я73
С95



<https://elibrary.ru/ngucay>

Рецензенты:

- В. И. Пономаренко* – доктор физико-математических наук, профессор,
ведущий научный сотрудник Саратовского филиала
Института радиотехники и электроники имени В. А. Котельникова РАН;
В. В. Матросов – доктор физико-математических наук, профессор,
декан радиофизического факультета Национального исследовательского
Нижегородского государственного университета им. Н. И. Лобачевского;
Г. В. Курячий – преподаватель факультета ВМК МГУ имени М. В. Ломоносова,
автор курсов по Python для ВУЗов и Вечерней математической школы,
разработчик компании «Базальт СПО»

Сысоева М. В., Сысоев И. В.

С95

Программирование для «нормальных» с нуля на языке Python : Учебник.
В двух частях. Часть 1 / М.В. Сысоева, И.В. Сысоев ; отв. ред. В. Л. Черный. –
2-е изд., испр. и доп. – Москва : Базальт СПО; МАКС Пресс, 2023. – 184 с.
[+ 4 с. вкл.] : ил. – (Библиотека ALT).

ISBN 978-5-317-06945-2

ISBN 978-5-317-06946-9 (Часть 1)

Книга – учебник, задачник и самоучитель по алгоритмизации и программированию на языке 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 экспериментов.

Ключевые слова: программирование; численные методы; алгоритмы; графики; Python; pipru.

УДК 004.43(075.8)

ББК 22.1я73

Материалы, составляющие данную книгу, распространяются на условиях лицензии GNU FDL. Книга содержит следующий текст, помещаемый на первую страницу обложки: «В серии “Библиотека ALT”». Название: «Программирование для «нормальных» с нуля на языке Python. В двух частях. Часть 1». Книга не содержит неизменяемых разделов. Linux – торговая марка Линуса Торвальдса. Прочие встречающиеся названия могут являться торговыми марками соответствующих владельцев.

Сайт книги: <http://www.altlinux.org/Books:Python-sysoeva-ed2>

ISBN 978-5-317-06946-9 (Часть 1)
ISBN 978-5-317-06945-2

© Сысоева М. В., Сысоев И. В., 2018
© Сысоева М. В., Сысоев И. В., 2023, с изменениями
© Basealt, 2023
© Оформление. ООО «МАКС Пресс», 2023

Оглавление

Предисловие	5
Глава 1. Введение	7
1.1 Языки программирования	7
1.2 Парадигмы программирования	12
1.3 Типизация в языках программирования	14
1.4 Области программирования	20
1.5 Области применения Python	23
1.6 Первая программа. Среда разработки	27
Глава 2. Основные типы данных	32
2.1 Числа. Арифметические операции с числами. Модуль <code>math</code>	32
2.2 Строки	38
2.3 Условия и логические операции	42
2.4 Списки	48
2.5 Кортежи	53
2.6 Словари	55
2.7 Примеры решения заданий	56
2.8 Задания на работу с основными типами данных	59
Глава 3. Циклы	67
3.1 Цикл с условием (<code>while</code>)	67
3.2 Цикл обхода последовательности (<code>for</code>)	70
3.3 Некоторые основные алгоритмические приёмы	73
3.4 Отладка программы	78
3.5 Примеры решения заданий	85
3.6 Задания на циклы	87
Глава 4. Массивы. Модуль <code>numpy</code>	93
4.1 Создание и индексация массивов	94
4.2 Арифметические операции и функции с массивами	101
4.3 Двумерные массивы, форма массивов	107
4.4 Примеры решения заданий	111
4.5 Задания на массивы, модуль <code>numpy</code>	114
Глава 5. Графики. Модуль <code>matplotlib</code>	116

5.1	Простые графики	116
5.2	Заголовок, подписи, сетка, легенда	120
5.3	Несколько графиков на одном полотне	124
5.4	Гистограммы, диаграммы-столбцы	129
5.5	Круговые и контурные диаграммы	132
5.6	Трёхмерные графики	133
5.7	Учёт ошибок	135
5.8	Примеры построения графиков	136
5.9	Задания на построение графиков	139
Глава 6. Файлы		143
6.1	Открытие файла	143
6.2	Запись в текстовый файл	145
6.3	Чтение из текстового файла	148
6.4	Чтение из файла в двоичном режиме. Модуль <code>struct</code>	154
6.5	Модуль <code>pickle</code>	157
6.6	Работа с операционной системой. Модули <code>os</code> и <code>os.path</code>	160
6.7	Примеры решения заданий	162
6.8	Задания на работу с файлами и с операционной системой	166
Глава 7. Библиотеки, встроенные в <code>numpy</code>		169
7.1	Элементы линейной алгебры	169
7.2	Быстрое преобразование Фурье	173
7.3	Генерация случайных чисел	176
7.4	Примеры решения заданий	177
7.5	Задания на использование встроенных библиотек <code>numpy</code>	180

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, доцент кафедры «Радиоэлектроника и телекоммуникации» Саратовского государственного технического университета имени Гагарина Ю.А.
- Сысоев Илья Вячеславович — доктор физико-математических наук, профессор кафедры системного анализа и автоматического управления Сара-

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

Сведения о рецензентах

- Пономаренко Владимир Иванович — доктор физико-математических наук, профессор, ведущий научный сотрудник Саратовского филиала Института радиотехники и электроники имени В. А. Котельникова РАН.
- Матросов Валерий Владимирович — доктор физико-математических наук, профессор, декан радиофизического факультета Национального исследовательского Нижегородского государственного университета им. Н. И. Лобачевского.
- 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 которой 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. Айкенем. 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 (Планкалькуль). 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 Fortran 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 выполнении 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 не критично, 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 поколение	Транслируемые языки (ассемблеры)
III поколение	Компилируемые языки
IV поколение	Интерпретируемые языки
V поколение	Языки виртуальных машин

¹Многие 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 «классической», 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 приходиться 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, внося 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 программно-го обеспечения, в основе которой лежит представление программы в виде иерархической структуры блоков. Языками-первопроходцами в этой парадигме были Fortran, Algol и В, позже их приемниками стали Pascal и С. 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(пицца)**.

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

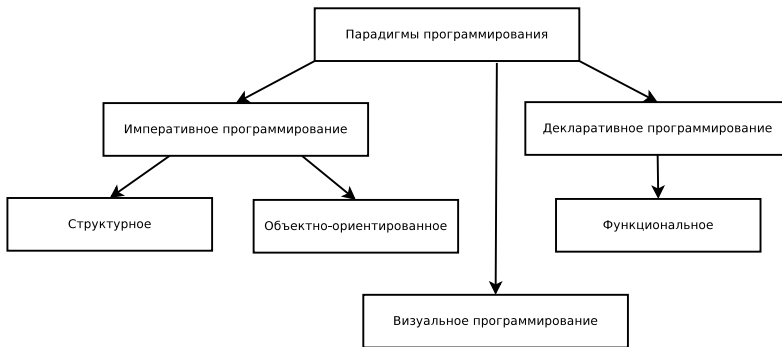


Рис. 1.1. Парадигмы программирования

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 мощь 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 договорённости 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 причиной 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 переключаются 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 интерпретируемые языки: Python, 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 явной типизацией в такой ситуации приходится писать много одинаковых функций, отличающихся только типом аргументов и результата (это называется перегрузкой функций), 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 статической типизацией: C, Java, C#, Ada, C++, D, Pascal. Примеры языков с динамической 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 статической 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 слабой 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 которой 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 базой 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 работой, 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 Windows 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='Liberation_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

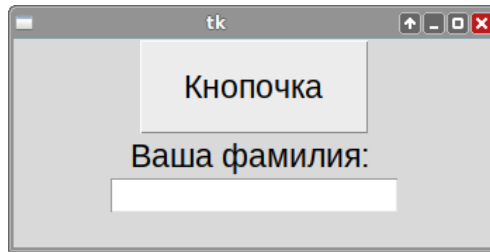


Рис. 1.2. Пример простейшего графического приложения, описанного выше.

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 `sqlite3` (`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

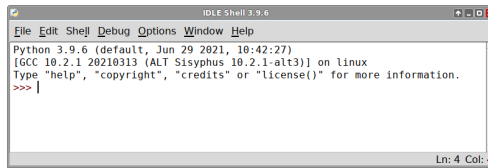
Базовая установка	Дополнительные модули	ОС
установщик с официального сайта https://python.org	встроенный механизм <code>pip</code> , любые	Linux, Windows, MacOS X
использование специальных сборок: WinPython, Pyzo, Anaconda и др.	частично встроены, расширение затруднительно	зависит от сборки

установка штатными средствами ОС	зависит от типа и версии ОС	Linux, MacOS X
----------------------------------	-----------------------------	-------------------

Все эти способы имеют свои преимущества и недостатки:

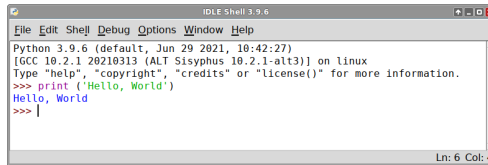
- При установке с официального сайта в Linux и MacOS X вам придётся столкнуться с тем, что у вас будут 2 частично пересекающихся интерпретатора Python в системе. Дело в том, что эти две ОС частично используют Python в своих целях и какой-то (часто не самый свежий) интерпретатор поставляется в комплекте. 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 штатно доступных пакетов невелико, а сами они часто очень древних версий. В Windows такой способ вообще невозможен.

Суммируя выше сказанное, мы будем рекомендовать пользователям Linux пользоваться своим штатным 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/>. В операционных системах семейства «Альт»



```
Python 3.9.6 (default, Jun 29 2021, 10:42:27)
[GCC 10.2.1 20210313 (ALT Sisyphus 10.2.1-alt3)] on linux
Type "help", "copyright", "credits" or "license()" for more information.
>>> |
```

Рис. 1.3. Интерактивный режим среды разработки IDLE.



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

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

штатным пакетным менеджером является Synaptic. «Альт Образование» — продукт, специально разрабатываемый для образовательных учреждений, в который уже включены все необходимые библиотеки для работы с Python, в том числе, среды Python IDLE и Spyder.

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-

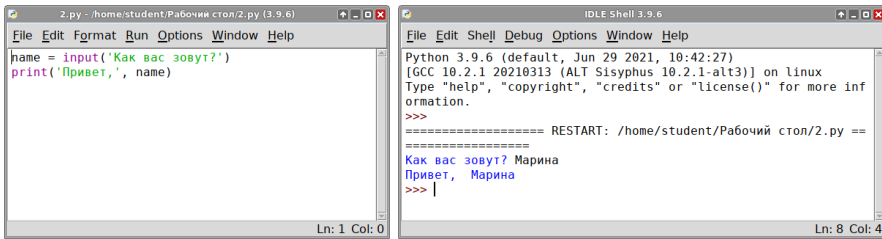


Рис. 1.5. Скриптовый и интерактивный режим: в интерактивном режиме можно видеть результаты выполнения скрипта.

пшло 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 запускает уже файл. Такой режим работы называется скриптовый. Файлы с кодом на Python обычно имеют расширение `py`.

Для того, чтобы создать новое окно для написания 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?» и ожидает, пока вы напечатаете что-нибудь и нажмёте `<Enter>`; введённое значение сохраняется 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`.

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 всё 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 (можно совместить первый и второй пункты в одной программе, запросив данные один раз, но обеспечив вывод обоими способами);
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), косую черту (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. Переменные¹ 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).

¹Строго 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 \rightarrow действительные \rightarrow 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 <code>float</code>
$x // y$	Деление x на y нацело: $11 // 4 = 2$, $11.8 // 4 = 2.0$, тип результата 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 : ($2 ** 3 = 8$)
<code>abs(x)</code>	Модуль числа x
<code>round(x)</code>	Округление (<code>round(11.3) = 11</code>)
<code>round(x, n)</code>	Округляет число x до n знаков после запятой: <code>round(12.34567, 3) = 12.346</code>
<code>pow(x, y)</code>	полный аналог записи $x ** y$
<code>divmod(x, y)</code>	выдаёт два числа: частное и остаток, обращаться следует так: $q, r = \text{divmod}(x, y)$

Отметим, что операции сложения, вычитания, умножения и возведения в степень выдают ответ типа `int` только если оба аргумента 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 операторы 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

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

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`

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

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

Первую строчку можно прочесть 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 старой кодировкой 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()` вычислит 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 вызывается. 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 $n2$ не включительно, если $n3$ присутствует (может не быть), то берутся не все символы, а с шагом $n3$.
$S2 \text{ in } S1$	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 одним аргументом, например метод `S1.endswith(S2)` требует 1 аргумент — строку — и проверяет, заканчивается ли строка $S1$ строкой $S2$. Бывают методы с двумя аргументами, например `S1.replace(S2, S3)`, который заменяет в исходной строке $S1$ содержащуюся в ней подстроку $S2$ новой подстрокой $S3$ и выдаёт новую строку, при этом $S1$ остаётся неизменной. 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 *любой* 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 двойные, 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 Алезе и говорить нечего, ему не быть первым.

Питер, к которому обратился Ник, возмущился: — Хиллу не видать первого места, а вот Алезе 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 Алезе.

Реплика Ника «Алезе 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: $v1 = \text{not } S \text{ and } H$;
- Ник: $v2 = S \text{ and not } A$;
- Питер: $v3 = \text{not } H$.

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

$$f = v1 \text{ and } v2 \text{ and not } v3 \text{ or } v1 \text{ and not } v2 \text{ and } v3 \setminus \\ \text{or not } v1 \text{ and } v2 \text{ and } v3$$

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 вид²:

```
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,

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

в том числе и `IDLE`, делают этот отступ автоматически. То, что выделено отступами, и есть *тело* оператора, а то, что до двоеточия, называется *заголовком*.

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', 't', 'h', '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
<code>L.append(x)</code>	<p>Добавление элемента со значением <code>x</code> в конец списка <code>L</code>:</p> <pre>>>> L = ['P', 'y', 't', 'h', 'o', 'n'] >>> L.append('3') >>> L ['P', 'y', 't', 'h', 'o', 'n', '3']</pre>
<code>L.extend(T)</code>	<p>Добавление списка или кортежа <code>T</code> в конец списка <code>L</code>. Похоже на объединение списков, но создание нового списка не происходит:</p> <pre>>>> 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']]</pre>

L.insert(i,x)	<p>Вставка элемента со значением x на позицию i в списке L:</p> <pre>>>> L = ['P', 'y', 't', 'h', 'o', 'n'] >>> L.insert(3, '!') >>> L ['P', 'y', 't', '!', 'h', 'o', 'n']</pre>
L.pop(i)	<p>Извлечение элемента с номером i из списка L, элемент удаляется и выдаётся в качестве результата:</p> <pre>>>> L = ['P', 'y', 't', 'h', 'o', 'n'] >>> x = L.pop(0) >>> L ['y', 't', 'h', 'o', 'n'] >>> x 'P'</pre> <p>Если использовать L.pop() без аргумента, то будет извлекаться последний элемент.</p>
L.remove(x)	<p>Удаление элемента со значением x из списка L:</p> <pre>>>> L = ['P', 'y', 't', '!', 'h', 'o', 'n'] >>> L.remove('!') >>> L ['P', 'y', 't', 'h', 'o', 'n']</pre> <p>Если в списке содержится несколько одинаковых элементов, удаляется тот, который имеет наименьший номер.</p>
L.count(x)	<p>Определение количества элементов, равных x, в списке L:</p> <pre>>>> L = [8, 1, 5, -7, 4, 9, -2, 6, 2, 5] >>> L.count(5) 2</pre>
L.index(x)	<p>Определение первой слева позиции элемента со значением x в списке L:</p> <pre>>>> L = [8, 1, 5, -7, 4, 9, -2, 6, 2, 5] >>> L.index(5) 2</pre>

L.reverse()	Переворачивание списка наоборот: <pre>>>> L = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] >>> L.reverse() >>> L [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]</pre>
L.sort()	Сортировка списка по возрастанию (в алфавитном порядке): <pre>>>> L = [10, 5, 2, 8, 1, 12] >>> L.sort() >>> L [1, 2, 5, 8, 10, 12]</pre>

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 изменение 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 создается 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 заданий

Пример задачи 2 (Арифметические операции) Сколько полных часов, минут и секунд содержится в A секундах? Разложите имеющееся количество секунд на сумму из x часов + y минут + z секунд.

Решение задачи 2 Напишем программу, используя операции деление нацело (`//`) и остаток от деления (`%`). Листинг программы:


```
A = int(input('Сколько секунд? '))
x = A//3600
s = A%3600
y = s//60
z = s%60
print (str(x)+' ч + '+str(y) +' мин + '+str(z)+' сек')
```

Вывод программы:

```
Сколько секунд? 5000
1 ч + 23 мин + 20 сек
```

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:

```
...йнёлез буд яьромокул У
```

Пример задачи 4 (Простое условие) Ответить на вопрос, истинно ли условие: $x^3 + y^3 \leq 9$. 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 на работу с основными типами данных

Задание 2 (Арифметические операции) 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, b — нечётное? a — нечётное, b — чётное?
5. N школьников делят K яблок поровну, неделяющийся остаток остается в корзине. Сколько яблок достанется каждому школьнику и сколько яблок останется в корзине?
6. Старинными английскими мерами длины являются: 1 миля = 1760 ярдов, 1 ярд = 3 фута, 1 фут = 12 дюймов. Сколько полных миль, ярдов и футов содержится в A дюймах. Разложите имеющееся количество дюймов на сумму из m миль + y ярдов + f футов + i дюймов.
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. В доме 9 этажей, на каждом этаже одного подъезда по 4 квартиры. В каком подъезде, и на каком этаже находится n -я квартира?
9. На территории Российской империи до введения метрической системы мер для измерения объёма жидкости применялись следующие меры: 1 бочка = 40 вёдер, 1 ведро = 10 штофов, 1 штоф = 10 чарок, 1 чарка = 2 шкалика. Какое полное количество вёдер, штофов, чарок и шкаликов можно наполнить из A бочек? Разлейте имеющееся количество некой жидкости на x полных вёдер + y полных штофов + z полных чарок + v шкаликов.
10. Старорусскими единицами измерения длины являлись: сажень — расстояние от конца пальцев одной руки до конца пальцев другой при расставленных в стороны руках, аршин — расстояние от кончика среднего пальца до плеча, пядь — расстояние между концами растянутых большого и указательного пальцев руки, вершок — длина основной фаланги указательного пальца. 1 сажень = 3 аршина = 12 пядей = 48 вершков. Какое полное количество сажень, аршин, пядей и вершков уместится в A вершках? Разложите имеющееся количество вершков на x сажень + y аршин + z пядей + v вершков.
11. 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?

12. Вы стоите на краю дороги и от вас до ближайшего фонарного столба x метров. Расстояние между столбами y метров. Длина вашего шага z метров. Мимо скольких столбов вы пройдёте, сделав n шагов?
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. $x^2 + y^2 \leq 4$;
2. $x^2 - y^2 \leq 4$;
3. $x \geq 0$ или $y^2 \neq 4$;
4. $x \geq 0$ 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. $x \geq 4$ или $y^2 \neq 4$;
10. $x \geq 4$ 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. Проверьте 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 $y^3 + py + q = 0$ 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 или 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 выполняйте 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 соединённого 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 образом¹:

```
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`.

¹В действительности, такое представление является частным и вместо `а 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 заикнется. 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.

Задача 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 $n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot n$.

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 ними².

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.

²В 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 эта 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 $1! + 2! + \dots + n!$

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 закликиванию: 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. Обобщённым 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 сложной 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*.

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 использования счётчика событий: пользователь вводит 10 чисел, необходимо определить, сколько из них являются одновременно чётными и положительными. 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 *обработать*, 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')
```

Нужно 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.

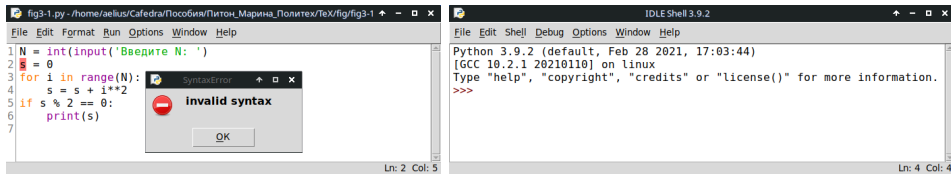


Рис. 3.1. Пример ошибки, выявляемой препроцессором.

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).

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 логикой 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 интерпретации `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 той, 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.

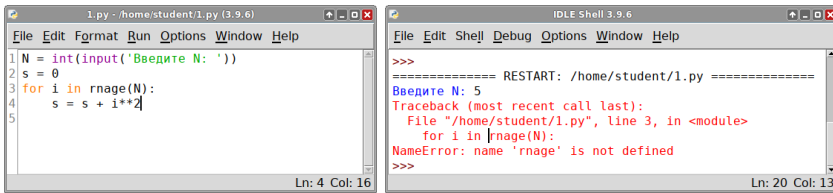


Рис. 3.2. Пример ошибки, выявляемой интерпретатором во время исполнения.

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 строке 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 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 ошибкой, 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.append(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.append(i**2)
AttributeError: 'list' object has no attribute 'append'
```

Интерпретатор указывает нам, что объект данного типа не имеет атрибута (метода или поля) `append`. 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 командой `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

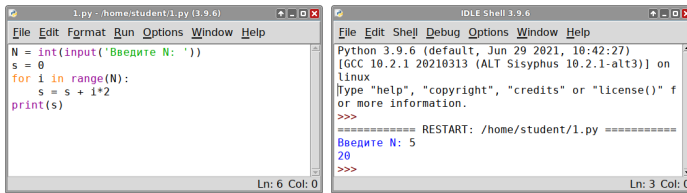


Рис. 3.3. Пример логической ошибки, которая может быть выявлена только разработчиком.

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:

```
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 интерпретируемых 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 несколько 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 Примеры решения заданий

Пример задачи 10 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, ..., 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 : $2p, 3p, 4p, \dots$).
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 10 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
```

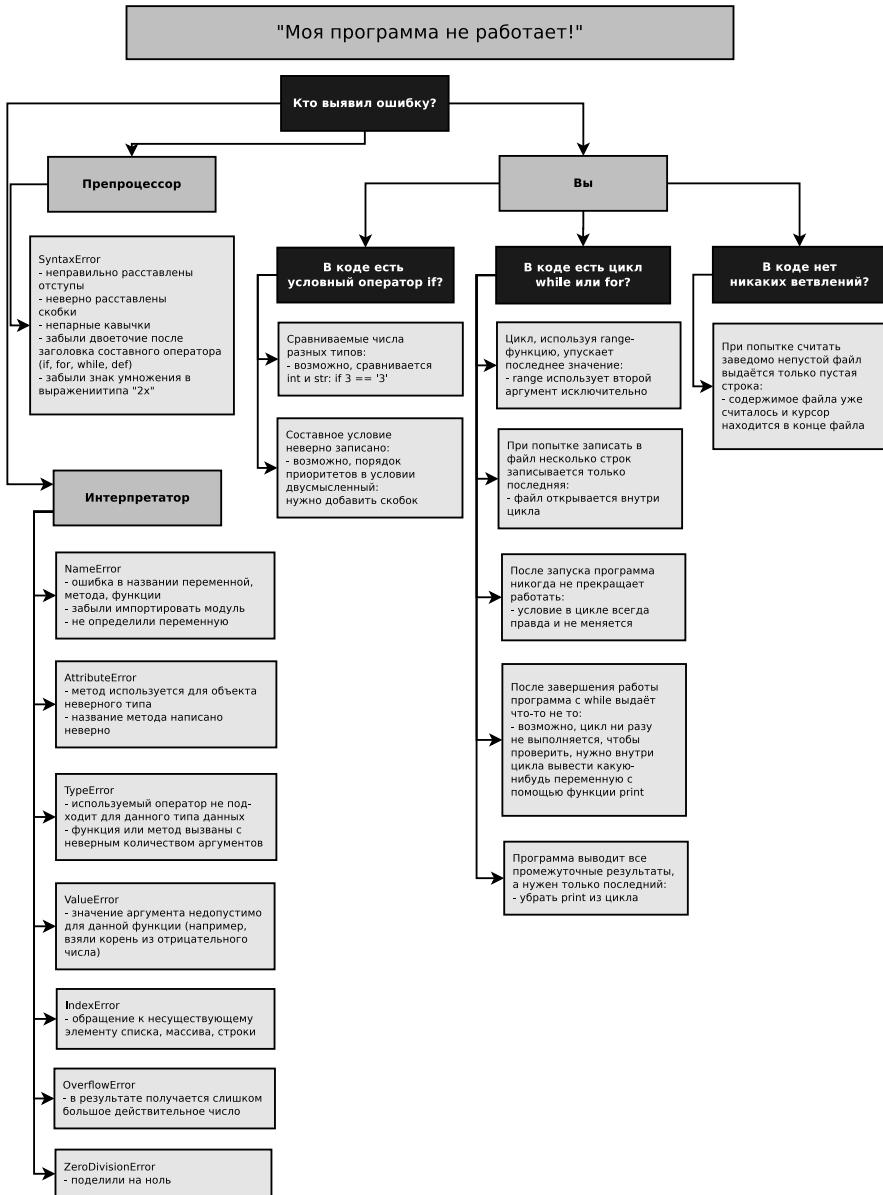


Рис. 3.4. Возможные источники ошибок и алгоритм их нахождения.

```

# 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]

3.6 Задания на циклы

Задание 7 (Задания на рекуррентные соотношения) Задания выполняйте все по порядку.

Придумайте рекуррентное соотношение, задающее следующие числовые последовательности:

- a) 1, 2, 3, 4, ...
- b) 0, 5, 10, 15, ...
- c) 1, 1, 1, 1, ...
- d) 1, -1, 1, -1, ...
- e) 1, -2, 3, -4, 5, -6 ...
- f) 2, 4, 8, 16, ...
- g) 2, 4, 16, 256, ...
- h) 0, 1, 2, 3, 0, 1, 2, 3, 0, ...
- i) 1!, 3!, 5!, 7!, ...

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
```

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 $1, 1 + \frac{1}{2}, 1 + \frac{1}{2} + \frac{1}{3}, \dots$ найдите первое, большее числа n .
8. Дано число a ($1 \leq a \leq 1.5$). Среди чисел $1 + \frac{1}{2}, 1 + \frac{1}{3}, 1 + \frac{1}{4}, \dots$ (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:

$$\begin{aligned}1 \times 5 &= 5 \\2 \times 5 &= 10 \\&\dots \\9 \times 5 &= 45\end{aligned}$$

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

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

$$\begin{aligned} 1 \times 9 &= 9 \\ 2 \times 9 &= 18 \\ \dots & \\ 9 \times 9 &= 81 \end{aligned}$$

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

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

$$\begin{aligned} 1 \times n &= \dots \\ 2 \times n &= \dots \\ \dots & \\ 9 \times n &= \dots \end{aligned}$$

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 круглый пирог. Пирог и его части можно делить только пополам, причём разрезы всегда делаются через центр пирога от края до края. 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 всё 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.

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

Глава 4

Массивы. Модуль `numpy`

Сам по себе «чистый» Python пригоден только для несложных вычислений. Наиболее привлекательная 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, добавляющий 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, `ruopenc1/rusuda` для вычисления на видеокартах и некоторые другие. Возможности `numpy` и `scipy` покрывают практически все потребности в математических алгоритмах.

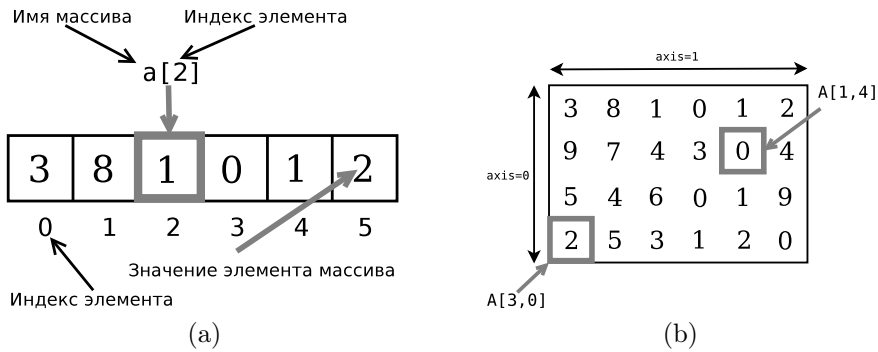


Рис. 4.1. Одномерный — (a) и двумерный — (b) массивы.

4.1 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 называются одномерными, с двумя — двумерными, и т. д. Одномерный массив («колонка», «столбец») примерно соответствует вектору в математике (на рис. 4.1(a) $a[4] == 56$, т. е. четвёртый элемент массива a равен 56); двумерный — матрице (на рис. 4.1(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 переписывания 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'>
```

Импорт из `numpy` «со звёздочкой» был ранее очень популярен. Он для простоты использовался в первом издании нашей книги. Но в настоящее время он признан нежелательным в первую очередь из-за того, что многие функции `numpy` дублируют и «затеняют» соответствующие стандартные функции, что явно не способствует лучшему пониманию и отлаживанию кода. Самые распространённые из них `sum`, `min` и `max`. Кроме того, большое число функций `numpy` также совпадают по названию с соответствующими функциями стандартного модуля `math`, что при импорте обоих модулей через звёздочку вызывает путаницу, какая функция используется в каждом конкретном случае. Поэтому далее мы везде будем импортировать и использовать `numpy` через укороченный псевдоним `np`, как это сделано в следующем примере.

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

```
import numpy as np
B = np.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 = np.array([[1, 2, 3], [4, 5, 6]], dtype = float)
print(C)
```

Вывод:

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

Можно создать массив из диапазона:

```
import numpy as np
L = np.range(5)
A = np.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 действительных 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 $\pi/6$:

```
import numpy as np
a1 = np.arange(0, 2*pi, pi/6)
s1 = np.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 не включительно, `linspace` — включительно. Это можно поправить с помощью опционального, четвёртого аргумента `linspace`, если установить его в `False`. Вот пример задания одинаковых массивов с помощью `arange` и `linspace`:

```
>>> import numpy as np
>>> a = np.arange(0, 2, 0.25)
>>> print(a)
[ 0.    0.25  0.5   0.75  1.    1.25  1.5   1.75]
>>> b = np.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. 4.2). 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:

```
import numpy as np
a1 = np.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 не включительно) элементов массива `a1`, массив `a3` — из всех чётных элементов `a1` (элементы от нулевого до последнего с шагом 2). Для обозначения

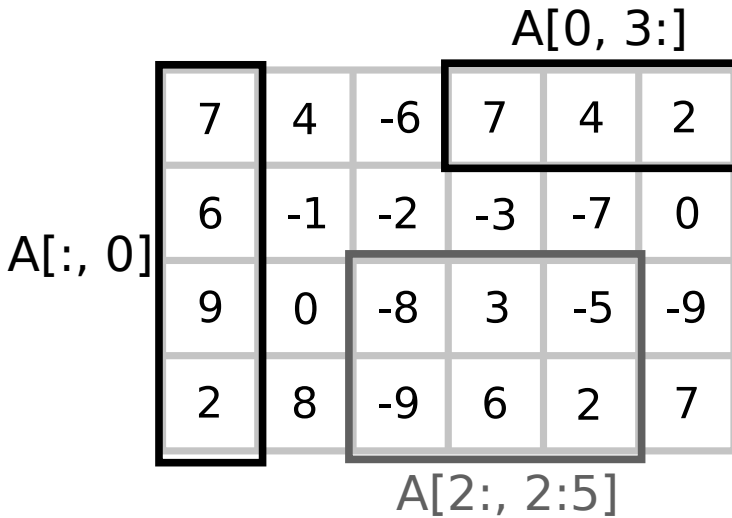


Рис. 4.2. Примеры срезов массивов: чёрным — срезы, в результате которых из двумерного массива получаются одномерные (из столбца и строки исходного массива соответственно); тёмносерым — срез, в результате которого получается двумерный массив меньшего размера.

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.

```
import numpy as np
a1 = np.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`:

```
import numpy as np
a1 = np.arange(0, 0.6, 0.1)
a2 = np.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 — `np.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. Четвёртая функция — `full` — может создавать массивы, заполненные произвольными одинаковыми числами, для чего принимает два аргумента: первый — число элементов, второй — значение. Вот пример работы этих функций:

```
import numpy as np
a1 = np.ones(5)
a2 = np.zeros(3)
a3 = np.empty(4)
a4 = np.full(6, 1.25)
print(a1)
print(a2)
print(a3)
print(a4)
```

Вывод:

```
[1.  1.  1.  1.  1.]
[0.  0.  0.]
[ 1.08168993e-316  0.00000000e+000  6.92979767e-310 -2.09463279e+220]
[1.25  1.25  1.25  1.25  1.25  1.25]
```

Видно, что в `a3` помещены 4 произвольных числа. В нашем случае три из них получились равными или близкими к нулю, а одно (последнее) — огромным.

Можно вручную задать тип данных:

```
a3 = np.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 = np.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`:

```
import numpy as np
import sys
A = np.arange(0, 3000, 1)
np.set_printoptions(threshold=sys.maxsize)
print(A)
```

Здесь также пришлось импортировать стандартный модуль `sys`, который хранит ряд системных переменных. В частности, `sys.maxsize` — максимальное целое число, доступное операционной системе нативно (вместо него можно было бы просто указать любое большое число, например, длину массива).

И вообще, с помощью этой функции можно настроить печать массивов «под себя». Функция `np.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` или `-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
```

```
Warning (from warnings module):
```

```
File "/home/aelius/Dropbox/Пособия/Питон_II_издание/test.py", line 4
    print(1./el)
```

```
RuntimeWarning: divide by zero encountered in double_scalars
```

```
inf
```

```
0.2
```

4.2 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 4.1. Пусть у нас есть массив вида `a=np.array([0.1, 0.25, -1.75, 0.4, -0.9])`:

Вот небольшая программа, показывающая, как эти функции работают:

```
import numpy as np
a = np.array([0.1, 0.25, -1.75, 0.4, -0.9])
print(a.sum())
```

Таблица 4.1. Некоторые методы массивов

Метод	Описание
<code>a.sum()</code>	Сумма элементов массива: $\sum_{i=0}^{\text{len}(a)} a_i = 0.1 + 0.25 - 1.75 + 0.4 - 0.9 = -1.9$
<code>a.mean()</code>	Среднее элементов массива: $\bar{a} = \frac{1}{\text{len}(a)} \sum_{i=0}^{\text{len}(a)} a_i = (0.1 + 0.25 - 1.75 + 0.4 - 0.9)/5 = -0.38$
<code>a.min()</code>	Минимальный элемент массива: -1.75
<code>a.max()</code>	Максимальный элемент массива: 0.4
<code>a.var()</code>	Дисперсия элементов массива: $\sigma_a^2 = \frac{1}{\text{len}(a)} \sum_{i=0}^{\text{len}(a)} (a_i - \bar{a})^2 = ((0.1 + 0.38)^2 + (0.25 + 0.38)^2 + (-1.75 + 0.38)^2 + (0.4 + 0.38)^2 + (-0.9 + 0.38)^2)/5 = 0.6766$
<code>a.std()</code>	Среднеквадратичное отклонение элементов массива от среднего: $\sigma_a = 0.822556988907$

```
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
0.8225569889071517
```

Все перечисленные методы имеют аналогичные функции `numpy`, например, вместо `a.min()` можно написать `np.min(a)`.

Обратите внимание на круглые скобки после имени каждой функции — они указывают, что вызывается и исполняется соответствующая функция. Поскольку Python язык очень глубоко объектный, сами по себе методы, как и их параметры и результаты, тоже являются объектами. Если скобки забыть, никаких вычислений не будет произведено, а вместо их результатов будут напечатаны строки документации соответствующих функций. То есть `a.min()` — это резуль-

тат вычисления — минимальный элемент массива `a`, а `a.min` — это сам метод вычисления как объект:

```
print(a.min)
```

Вывод:

```
<built-in method min of numpy.ndarray object at 0x00000000036B7710>
```

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 действительные, 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:

```
import numpy as np
a = np.array([0.1, 0.25, -1.75, 0.4, -0.9])
b = np.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:

```
import numpy as np
a = np.array([0.1, 0.25, -1.75, 0.4, -0.9])
b = np.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 (команды вводятся в интерактивном режиме):

```
>>> import numpy as np
>>> np.array([-1.75, 0.4]) * np.array([0.1, 0.25])
array([-0.175,  0.1  ])
```

в то время как в результате скалярного произведения должно было получиться: $-1.75 \cdot 0.1 + 0.4 \cdot 0.25 = -0.075$.

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:

```
import numpy as np
t = np.arange(0, np.pi, np.pi/6)
x = np.sin(t)
y = np.log(t)
print(t)
print(x)
print(y)
```

Вывод (вначале идёт предупреждение о делении на ноль, в результате чего одним из элементов будет `-inf`):

```
Warning (from warnings module):
  File "/home/aelius/Dropbox/Пособия/Питон_II_издание/test.py", line 4
    y = np.log(t)
RuntimeWarning: divide by zero encountered in log
[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 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, в ответе получилась бесконечность.

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 последние три строки с операторами `print` циклом:

```
for i in range(len(t)):
    print(t[i], x[i], y[i])
```

Вывод программы (предупреждение о делении на ноль мы опустили):

```
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 нескольких операций в интерактивном режиме:

```
>>> import numpy as np
>>> x = np.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 += np.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 выше):

```
>>> 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 бы сообщение об ошибке `NameError`, сразу локализовав проблему. Такие ошибки могут быть очень коварны в случае, если полученное на данном этапе значение используется в дальнейших вычислениях.

4.3 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.

```
import numpy as np
x = np.zeros((2, 3))
y = np.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 чисел (2, 3)); аналогично можно создать матрицу произвольного размера. Функция `eye` принимает единственный скалярный аргумент (число, а не кортеж), 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:

```
import numpy as np
a = np.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 каждой строкой отдельно.

Итерирование многомерных массивов начинается с нулевой оси, то есть в случае матриц элементом является строка:

```
import numpy as np
a = np.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, используя атрибут `flat` (для удобства числа выводятся в строку, используя дополнительный аргумент `end` функции `print`, значение которого по умолчанию — `'\n'`):

```
for element in a.flat:
    print(element, end= '␣')
```

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:

```
import numpy as np
a = np.array([[1, 2, 3], [4, 5, 6]])
b = np.array([[5, 6, 7], [8, 9, 10]])
print(np.vstack((a, b)))
print(np.hstack((a, b)))
```

Вывод:

```
[[ 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 `np.hsplitted()` 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]
```

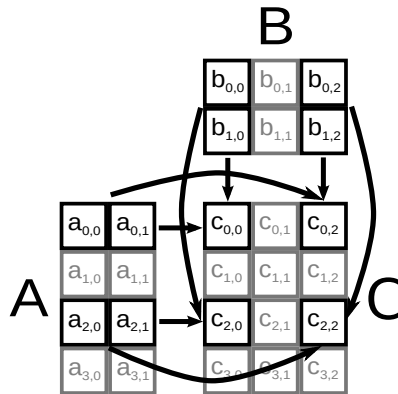


Рис. 4.3. Схема матричного умножения: $C = A \times B$.

Модуль `numpy` предоставляет множество различных функций для работы с многомерными массивами, одна из самых популярных — функция `np.dot`. Для матриц и одномерных массивов-векторов эта функция соответствует функции матричного умножения, см. рис. 4.3 (скалярное произведение для двух одномерных массивов):

```
import numpy as np
a = np.array([[1, 0], [0, 1]])
b = np.array([[4, 1], [2, 2]])
print(np.dot(a, b))
x = np.array([-4, 1, 3, 2])
y = np.array([-1, 8, 1, -2])
print(np.dot(x, y))
z = np.array([5, -2])
print(np.dot(z, b))
print(np.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, для обозначения матричного умножения вместо функции `dot` можно использовать символ `@`:

```
>>> import numpy as np
>>> a = np.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:

```
import numpy as np
a = np.eye(3)
b = np.arange(0, 10, 1)
np.savetxt('eye3.txt', a)
np.savetxt('Mas.txt', b)
a2 = np.loadtxt('eye3.txt')
print(a2)
b2 = np.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.

4.4 Примеры решения заданий

Пример задачи 11 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 11

```
import numpy as np
n = int(input('Размер матрицы: '))
x = np.zeros((n, n))
for i in range(n):
    x[i, :i] = -1
    x[i, i+1:] = 1
print(x)
np.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 12 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 12

```
import numpy as np
y = np.loadtxt('Mas.txt')
print(y.shape)
a = np.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 13 Протабулируйте (вычислите значения функций при изменении аргумента в некоторых пределах с заданным шагом) функцию $\sin(t)$ на отрезке $[-10; 10]$.

Решение задачи 13

```
import numpy as np
dt = 1
t = np.arange(-10, 10+dt, dt)
x = np.sin(t)
for i in range(len(t)):
    print("{:12.8f}\t{:12.8f}".format(t[i], x[i]))
```

Вывод программы:

```
-10.00000000  0.54402111
-9.00000000  -0.41211849
-8.00000000  -0.98935825
-7.00000000  -0.65698660
-6.00000000  0.27941550
-5.00000000  0.95892427
-4.00000000  0.75680250
-3.00000000  -0.14112001
-2.00000000  -0.90929743
-1.00000000  -0.84147098
 0.00000000  0.00000000
 1.00000000  0.84147098
 2.00000000  0.90929743
 3.00000000  0.14112001
 4.00000000  -0.75680250
 5.00000000  -0.95892427
 6.00000000  -0.27941550
 7.00000000  0.65698660
 8.00000000  0.98935825
 9.00000000  0.41211849
10.00000000  -0.54402111
```

4.5 Задания на массивы, модуль `numpy`

Задание 11 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.

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 12 Задания выполняйте все по порядку.

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 13 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.

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. $2 \cos(t - 2) + \sin(2t - 4)$ 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 5

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 называемые скрипты (маленькие программы на интерпретируемом языке программирования). 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, их сохранение в файлы, дают почти неограниченный контроль над точностью позиционирования и размером деталей.

Модуль `matplotlib` — специализированная библиотека для языка Python. Хотя основное её преимущество в простоте и скорости использования, она позволяет делать графики очень высокого типографского качества. Модуль `matplotlib` базируется на возможностях `numpy`, установка которого обязательна для функционирования `matplotlib`.

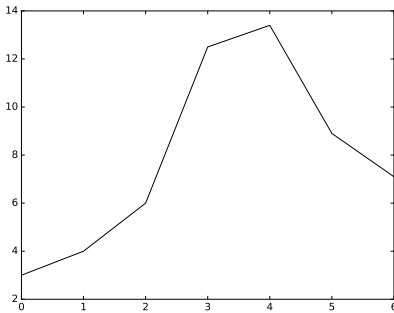
5.1 Простые графики

Непосредственно за построение двумерных графиков отвечает библиотека `pyplot` модуля `matplotlib`. Основная команда для построения графиков — команда `plot`. У команды `plot` существует множество вариантов синтаксиса, в

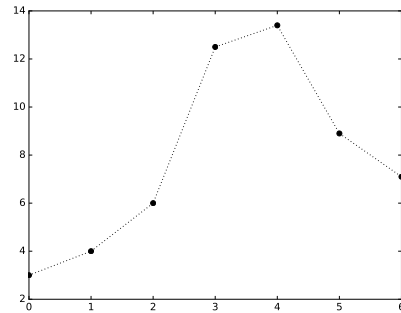
простейшем случае она требует единственный аргумент — массив, но подойдёт и список, кортеж или даже диапазон. Вот пример построения простейшего графика (см. рис. 5.1(a)):

```
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. 5.1. 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. 5.1(b)):

```
plot(x, 'o:')
show()
```

В результате, тот же график будет выглядеть немного иначе: на месте расположения данных появятся жирные синие кружки, соединённые тонкою пунктир-

Таблица 5.1. 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, повернутый на 45°
'p'	маркер пятиугольник
'h'	маркер шестиугольник
'*'	маркер звёздочка
'+'	маркер плюс
'x'	маркер крестик, как знак умножения ×
' '	маркер вертикальный отрезок
'_'	маркер горизонтальный отрезок

ною линий. Чтобы изменить цвет графика, достаточно указать дополнительный именованный параметр `color`:

```
plot(x, ':o', color='grey')
show()
```

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 кодировкой 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()
```

Здесь мы использовали частичный импорт только нужных функций из `matplotlib.pyplot` и `math`, что правильно, потому что позволяет не напутать и избежать одновременного импорта функций с одинаковым названием из разных модулей. Но при написании больших программ такой подход неудобен, потому что приходится писать слишком много всего. Поэтому в дальнейшем при импорте `matplotlib.pyplot` мы будем пользоваться общепринятым сокращением `plt`.

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 одной командой, например, синус и косинус разом (для этого нужно указывать абсциссы и ординаты попеременно):

```
import matplotlib.pyplot as plt
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]))
```

```
plt.plot(t, x, t, y)
plt.show()
```

На полученных графиках (см. цветную вкладку рис. 1(b)) синусоида получится бирюзового цвета, а косинусоида — оранжевого. 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:

```
plt.plot(t, x)
plt.plot(t, y)
plt.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 последние строки предыдущей программы так, чтобы она не только выводила график на экран, но и сохраняла его в файл `'sincos.png'`:

```
plt.plot(t, x)
plt.plot(t, y)
plt.savefig('sincos.png')
plt.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:

```
plt.plot(t, x)
plt.plot(t, y)
plt.savefig('sincos.png')
plt.savefig('sincos.pdf')
plt.show()
```

5.2 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` соответственно, принимающими единственный аргумент — строку:

```
import numpy as np
import matplotlib.pyplot as plt
t = np.arange(0, 4, 0.01)
x = np.sin(2*np.pi*t)
y = np.cos(2*np.pi*t)
plt.plot(t, x, t, y)
plt.title('Voltage over time')
plt.xlabel('time, s')
plt.ylabel('voltage, V')
plt.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 формирующих изображение команд 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'`. Это вызвано тем, что и в Python, и в `LATEX` символ `'\'` является специальным и вводит команды — вспомним, например, что в Python этим символом вводятся команды для «невидимых» символов табуляции и конца строки. Строка, начинающаяся с `'r'`, называется сырою, в такой строке все символы интерпретируются 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 `'$'`, с которого начинается формула `LATEX`. Собственно формулы следует вводить внутри пары из `'$'`, там можно использовать многие специальные обозначения: `'_'` для нижнего индекса (см. рис. 5.2(b)), `'^'` — для верхнего. Если в индекс входят несколько символов, следует заключить их в фигурные скобки, служащие в `LATEX` специальными символами наряду с `'\'`; причём сами фигурные скобки не отображаются — это ещё одна причина использовать сырые строки, так как в Python фигурные скобки тоже сами не отображаются, а служат для форматирования вывода, как мы видели в главе про массивы. 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 (\rightarrow , см. рис. 5.2(a)):

```
plt.xlabel(r'$t$, c', fontsize=18)
```

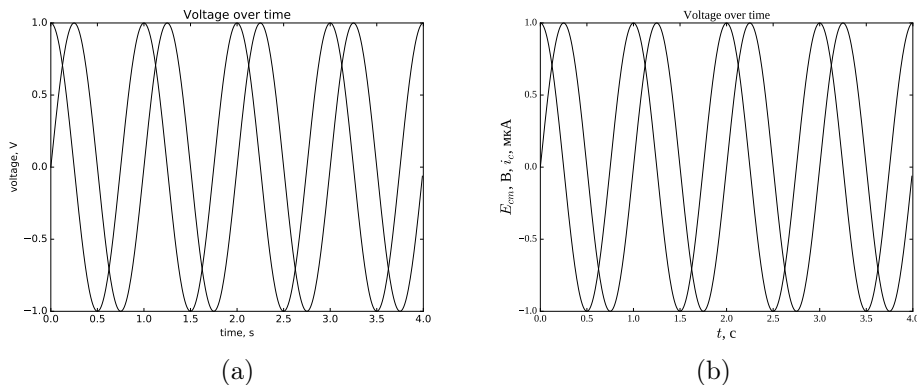


Рис. 5.2. Графики с подписями и названием.

```
plt.ylabel(r'$E_{cm}$, V, $t_c$, 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 латинские 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 присутствуют. Например, в Linux можно использовать шрифт `'Liberation Serif'`, а в Windows `'Arial'`, как это показано ниже. Кроме шрифтов можно также изменить весь стиль отрисовки, вернувшись к классическому виду, какой был в `matplotlib` до версии 2.0, для чего нужно кроме словаря `rcParams` импортировать модуль `style` и изменить стиль с помощью функции `use`.

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import rcParams, style
rcParams['font.sans-serif'] = ['Liberation_Serif', 'Arial']
style.use('classic')
```

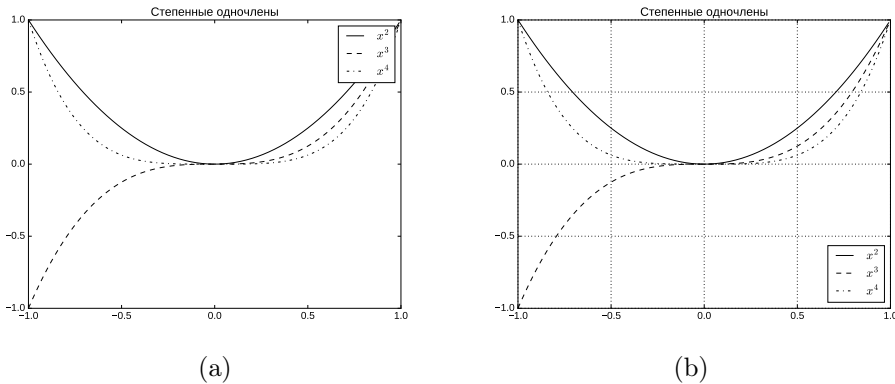


Рис. 5.3. Пример использования легенды: (a) — расположение по умолчанию, (b) — оптимальное расположение с дополнительно наложенной сеткой.

```
t = np.arange(-1, 1, 0.01)
x = t**2
y = t**3
z = t**4
plt.plot(t, x, label=r'$x^2$')
plt.plot(t, y, '--', label=r'$x^3$')
plt.plot(t, z, ':', label=r'$x^4$')
plt.legend()
plt.title('Степенные одночлены')
plt.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. 5.3(a). Чтобы исправить первый недостаток, нужно использовать параметр функции `legend` с именем `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'` — внизу посередине. Значение 0 соответствует строке `'best'` — Python сам пытается выбрать такое положение легенды, чтобы она не заслоняла график (см. рис. 5.3(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 добавить (как обычно, до `savefig` и `show`) в программу функцию `grid`. В итоге добавим в нашу программу ещё две строчки:

```
plt.legend(loc='best')
plt.grid(True)
```

У функции `grid` первый аргумент — логический. Если его значение правда — сетка будет, если ложь — нет, второй аргумент показывает

5.3 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.

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import rcParams, style
rcParams['font.sans-serif'] = ['Arial']
style.use('classic')
t = np.arange(0, 2, 0.01)
x = np.sqrt(t)
y = t
z = np.sqrt(t**3)
u = t**2
v = np.sqrt(t**5)
w = t**3

plt.subplot(3, 2, 1)
plt.plot(t, x, label='x**(1/2)')
plt.title(r'$\sqrt{x}$')
plt.ylim([0, 5])

plt.subplot(3, 2, 2)
plt.plot(t, y, label='x', color='red')
plt.title(r'$x$')
plt.ylim([0, 5])

plt.subplot(3, 2, 3)
plt.plot(t, z, label='x**(3/2)')
plt.title(r'$\sqrt{x^3}$')
plt.ylim([0, 5])
```

```
plt.subplot(3, 2, 4)
plt.plot(t, u, ':', label='x**2')
plt.title(r'$x^2$')
plt.ylim([0, 5])

plt.subplot(3, 2, 5)
plt.plot(t, v, label='x**(5/2)')
plt.title(r'$\sqrt{x^5}$')
plt.ylim([0, 5])

plt.subplot(3, 2, 6)
plt.plot(t, w, label=r'$x^3$')
plt.title(r'$x^3$')

plt.legend(loc=0)
plt.grid(True)
plt.tight_layout()
plt.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 умолчанию 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.

Чтобы придать нашим дальнейшим рассуждениям больше жизненности, приведём пример из радиопизики. 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:

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import rcParams, style
rcParams['font.sans-serif'] = ['Arial']
style.use('classic')
t = np.arange(0, 10, 0.001)
x = np.cos(2*np.pi*t)
y = np.cos(20*2*np.pi*t)
z = (1 + 0.7*x)*y
plt.subplot(2, 2, 1)
plt.plot(t[:2000], x[:2000], color='black')
plt.title('Модулирующий сигнал')
plt.xlabel('Time, \u03bc s')
plt.ylabel('Voltage, \u03bc V')
plt.subplot(2, 2, 2)
plt.plot(t[:2000], y[:2000], color='black')
plt.title('Несущий сигнал')
plt.xlabel('Time, \u03bc s')
plt.ylabel('Voltage, \u03bc V')
plt.subplot(2, 1, 2)
plt.plot(t, z, color='black')
plt.title('АМ-сигнал')
plt.xlabel('Time, \u03bc s')
plt.ylabel('Voltage, \u03bc V')
plt.tight_layout()
plt.show()
```

Здесь видно (см. рис. 5.4), 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.

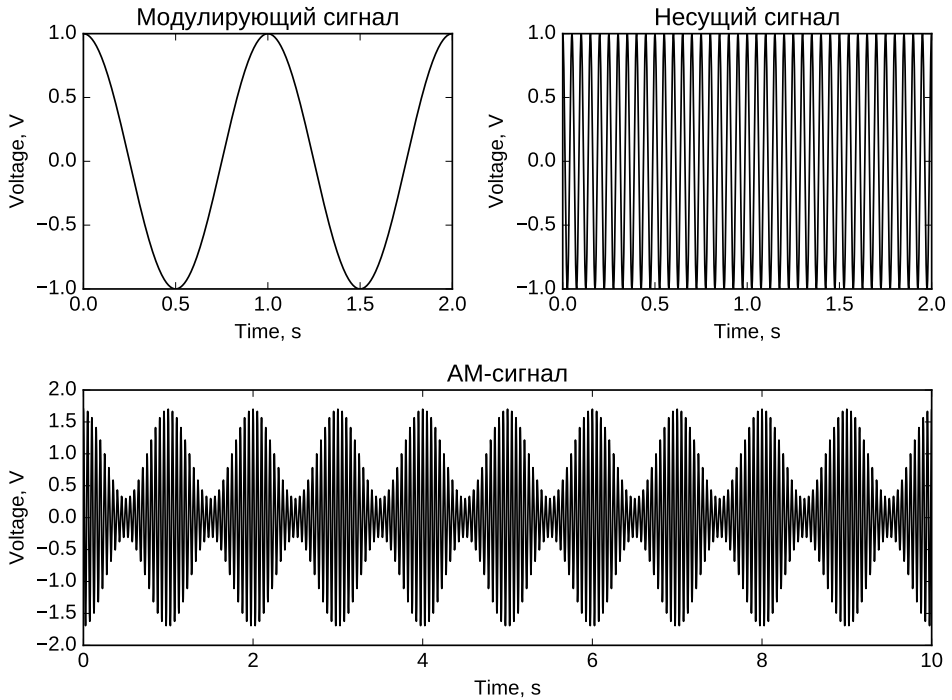


Рис. 5.4. Пример нескольких графиков различного размера на одном полотне.

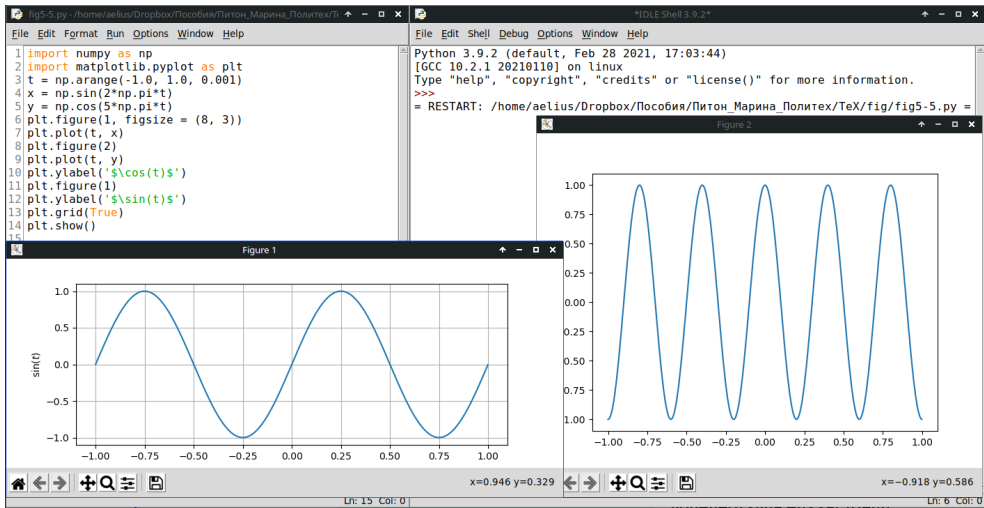
Часто бывает так, что требуется построить не несколько графиков в одном окне, а несколько окон с графиками; такая возможность в `matplotlib` предусмотрена. Например, вам нужно построить несколько разных рисунков и сохранить каждый в отдельный файл. Для этого нужно воспользоваться функцией `figure`. Вызов `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). Важно, что функция `tight_layout` будет действовать только в пределах полотна: раньше её действие распространялось на все графики (все `subplot`), теперь — только на последний. 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`:

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import rcParams, style
rcParams['font.sans-serif'] = ['Arial']
```

```
style.use('classic')
t = np.arange(0, 10, 0.001)
x = np.cos(2*np.pi*t)
y = np.cos(20*2*np.pi*t)
z = (1 + 0.7*x)*y
plt.figure(1)
plt.plot(t[:2000], x[:2000], color='black')
plt.title('Модулирующий сигнал')
plt.xlabel('Time,  $\mu$ s')
plt.ylabel('Voltage,  $\mu$ V')
plt.figure(2)
plt.plot(t[:2000], y[:2000], color='black')
plt.title('Несущий сигнал')
plt.xlabel('Time,  $\mu$ s')
plt.ylabel('Voltage,  $\mu$ V')
plt.figure(3)
plt.plot(t, z, color='black')
plt.title('AM-сигнал')
plt.xlabel('Time,  $\mu$ s')
plt.ylabel('Voltage,  $\mu$ V')
plt.tight_layout()
plt.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 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('$\sin(t)$')
plt.grid(True)
plt.show()
```


Рис. 5.5. Пример вызова нескольких полотен (`figure`).

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. 5.5).

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()`.

5.4 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`.

```

from random import uniform, normalvariate
import matplotlib.pyplot as plt

```

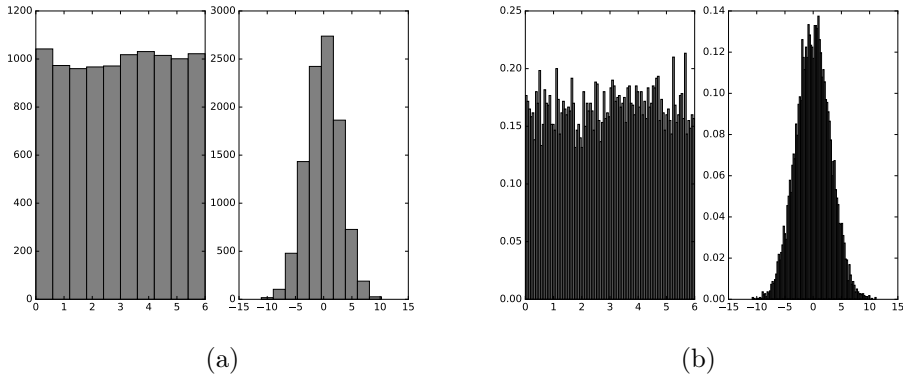


Рис. 5.6. Пример построения гистограмм: (а) — использованы значения параметров по умолчанию, в результате на обоих графиках наблюдаем по 10 бинов, а по вертикали отложено число попаданий в каждый из них; (б) — число бинов выставлено вручную и равно 100, по вертикали отложены значения плотности вероятности.

```
v = []
for i in range(10000):
    v.append(uniform(0, 6))
plt.subplot(1, 2, 1)
plt.hist(v)
w = []
for i in range(10000):
    w.append(normalvariate(0, 3))
plt.subplot(1, 2, 2)
plt.hist(w)
plt.show()
```

Представленная программа делит весь диапазон от минимального до максимального значения на 10 бинов и рисует частоты — число попаданий в каждый бин (см. рис. 5.6(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
import matplotlib.pyplot as plt
v = []
```

```
for i in range(10000):
    v.append(uniform(0, 6))
plt.subplot(1, 2, 1)
plt.hist(v, bins=100, density=True)
w = []
for i in range(10000):
    w.append(normalvariate(0, 3))
plt.subplot(1, 2, 2)
plt.hist(w, bins=100, density=True)
plt.show()
```

Теперь видим (см. рис. 5.6(b)), что по вертикали отложено уже знание плотности вероятности, причём график стал более изрезанным, поскольку увеличилось число бинов, а число значений 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:

```
import numpy as np
from random import normalvariate
import matplotlib.pyplot as plt
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 = np.arange(1, len(data1)+1)
width = 0.2
plt.bar(locs, data1, width=width, color='blue')
plt.bar(locs+width, data2, width=width, color='red')
plt.bar(locs+2*width, data3, width=width, color='green')
plt.xticks(locs+width*1.5, locs)
plt.show()
```

В данном примере `width` задаёт ширину прямоугольника, `color` задаёт его цвет. Опционно можно дописать `xerr`, `yerr`, которые позволяют устанавливать `error bars` — планки погрешностей. 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`.

5.5 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):

```
import matplotlib.pyplot as plt
data = [18, 15, 11, 9, 8, 6]
labels = ['Java', 'C', 'C++', 'PHP', 'Python', 'Ruby']
explode = [0, 0, 0, 0, 0.2, 0]
plt.pie(data, labels=labels, explode=explode,
        autopct='%1.1f%%', shadow=True)
plt.savefig('cirkla_abako.pdf')
plt.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 мира: 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 командой `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 :

```
import numpy as np
import matplotlib.pyplot as plt
x = np.arange(-3, 3, 0.01)
y = np.arange(-2, 2, 0.01)
X, Y = np.meshgrid(x, y)
Z = X**2 - 4*Y**2 + Y**4
plt.contour(X, Y, Z)
plt.show()
```

Представленная программа строит контурную диаграмму функции с помощью линий уровней, см. цветную вкладку на рис. 4(a). Можно всё залить цветом, как на физической карте мира. Для этого необходимо использовать функцию `contourf`, см. цветную вкладку на 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.

5.6 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 необязательные параметры, но пока мы их использовать не будем. Давайте нарисуем пустые оси (рис. 5.7):

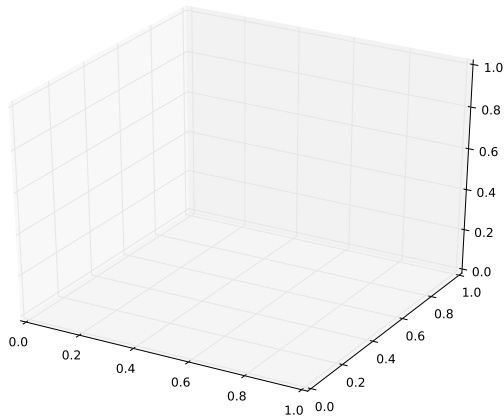


Рис. 5.7. 3D-оси.

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure()
Axes3D(fig)
plt.show()
```

Полученные оси вы можете вращать мышкой в интерактивном режиме.

А теперь нарисуем что-нибудь трёхмерное в полученных осях. Можно рисовать один каркас (`plot_wireframe`, см. цветную вкладку рис. 5(a)), а можно — поверхность (`plot_surface`, см. цветную вкладку рис. 5(b)):

```
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import Axes3D
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))
fig1 = plt.figure()
ax1 = Axes3D(fig1)
ax1.plot_wireframe(x, y, z, color='orange')
plt.savefig('3D_wireframe.pdf')
plt.show()
fig2 = plt.figure()
ax2 = Axes3D(fig2)
ax2.plot_surface(x, y, z)
```

```
plt.savefig('3D_surface.pdf')
plt.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 $(\text{stop}-\text{start})/(\text{num}-1)$, иначе оно не будет включено и расстояние между соседними значениями будет равно $(\text{stop}-\text{start})/\text{num}$.

5.7 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. 5.8). 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:

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import rcParams
rcParams['font.sans-serif'] = ['Arial']
x = np.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 = np.column_stack([y1, y2, y3])
plt.errorbar(x, y.mean(axis=1), yerr=[y.std(axis=1),
                                     y.std(axis=1)], marker='.', color='black')
plt.title('Вольт-амперная характеристика')
plt.xlabel('Напряжение, В');
plt.ylabel('Ток, мкА')
plt.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

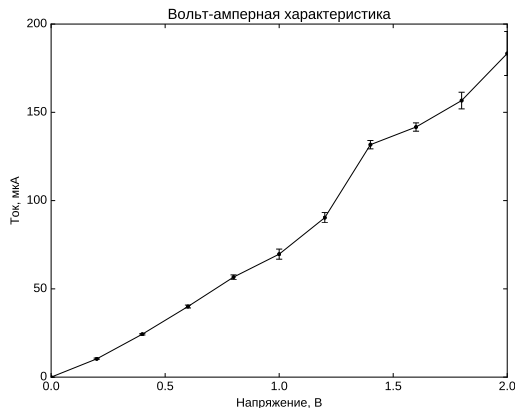


Рис. 5.8. ВАХ диода, усреднённая по трём экспериментам с отложенными планками погрешностей.

`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 качестве 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.

5.8 Примеры построения графиков

Пример задачи 14 (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 14

```
import numpy as np
import matplotlib.pyplot as plt
x = np.arange(0, 10, 0.1)
f = np.exp(-x) * np.sin(2*np.pi*x)
plt.plot(x, f)
```

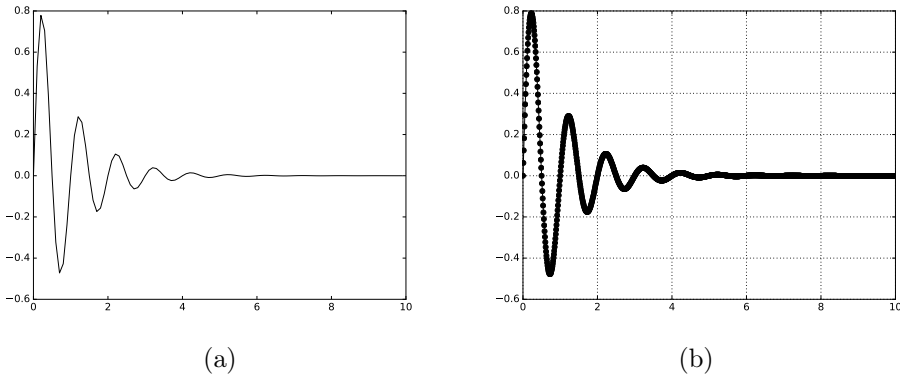



Рис. 5.9. Иллюстрация к задачам 14 — (a) и 15 — (b).

```
plt.show()
```

Пример задачи 15 (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 15

```
import numpy as np
import matplotlib.pyplot as plt
x = np.arange(0, 10, 0.1)
f = np.exp(-x) * np.sin(2*np.pi*x)
plt.plot(x, f, '-o', color='black')
plt.grid(True)
plt.savefig('plot.pdf')
plt.show()
```

Пример задачи 16 (Семейство затухающих синусоид) Постройте семейство функций $e^{-x} \sin(2\pi x + \phi_0)$ на одном графике различными типами линии при $\phi_0 = 0$, $\phi_0 = \pi/6$ и $\phi_0 = \pi/3$. Сделайте для графика легенду.

Решение задачи 16

```

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.pyplot import style
style.use('classic')
x = np.arange(0, 10, 0.01)
f1 = np.exp(-x) * np.sin(2*np.pi*x)
f2 = np.exp(-x) * np.sin(2*np.pi*x+np.pi/6)
f3 = np.exp(-x) * np.sin(2*np.pi*x+np.pi/3)
plt.plot(x, f1, '-', color='black', label=r'\sin(2\pi x)')
plt.plot(x, f2, '-', color='black', label=r'\sin(2\pi x + \pi /6)')
plt.plot(x, f3, ':', color='black', label=r'\sin(2\pi x + \pi /3)')
plt.legend(loc='best')
plt.grid(True)
plt.show()

```

Пример задачи 17 (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 17

```

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.pyplot import style
style.use('classic')
x = np.arange(0, 10, 0.01)
f1 = np.exp(-x) * np.sin(2*np.pi*x)
f2 = np.exp(-x) * np.sin(2*np.pi*x+np.pi/6)
f3 = np.exp(-x) * np.sin(2*np.pi*x+np.pi/3)
plt.subplot(3, 1, 1)
plt.title(r'\sin(2\pi x)')
plt.plot(x, f1, color='black')
plt.grid(True)
plt.subplot(3, 1, 2)
plt.title(r'\sin(2\pi x+\pi/6)')
plt.plot(x, f2, color='black')
plt.grid(True)
plt.subplot(3, 1, 3)
plt.title(r'\sin(2\pi x+\pi/3)')
plt.plot(x, f3, color='black')
plt.grid(True)
plt.tight_layout()
plt.show()

```

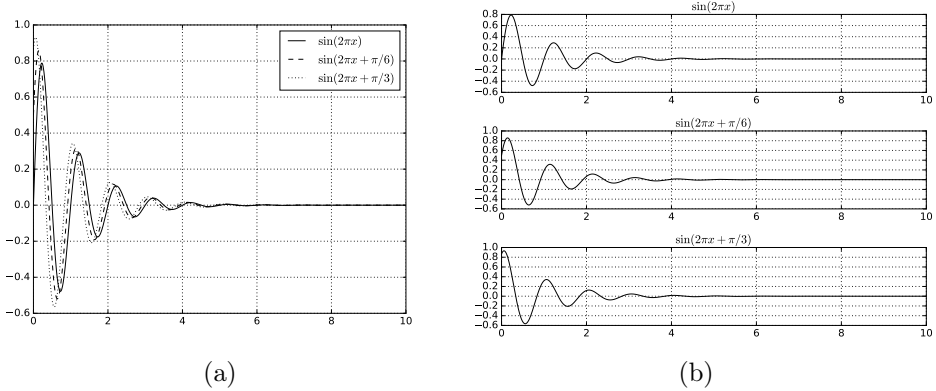


Рис. 5.10. Иллюстрации к задачам 16 – (a) и 17 – (b).

Пример задачи 18 Постройте закрашенную контурную диаграмму и трёхмерный график для функции двух переменных (5.1), определённой в прямоугольной области ($x \in [-3; 3]$, $y \in [-3; 3]$):

$$z = \frac{2xy}{x^2 + y^2} \quad (5.1)$$

```

Решение задачи 18 import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
x = np.arange(-3, 3, 0.01)
y = np.arange(-3, 3, 0.01)
X, Y = np.meshgrid(x, y)
Z = 2*X*Y/(X**2+Y**2)
plt.contourf(X, Y, Z)
plt.savefig('sedlo_a.pdf')
fig = plt.figure()
ax = Axes3D(fig)
ax.plot_surface(X, Y, Z)
plt.savefig('sedlo_b.pdf')
plt.show()

```

5.9 Задания на построение графиков

Задание 14 Выполнять одно задание с номером $(n - 1) \% t + 1$, где n – номер в списке группы, а t – число задач в задании.

Постройте графики следующих функций, используя шаг выборки данных по абсциссе из задания 13:

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. $2 \cos(t - 2) + \sin(2t - 4)$ 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 15 Выполнять одно задание с номером $(n - 1)\%t + 1$, где n — номер в списке группы, а t — число задач в задании.

Для построенного в рамках задания 14 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`, `jpg`, `pdf`, `eps`, `svg`, `svgz`. Откройте и просмотрите получившиеся графики. Сравните качество при масштабировании и размер в байтах одного и того же графика, сохранённого в разных форматах.

Задание 16 Выполнять одно задание с номером $(n - 1)\%t + 1$, где n — номер в списке группы, а t — число задач в задании.

Постройте семейство функций на одном графике различными цветами:

1. степенные одночлены с целыми степенями от 1 до 6 на отрезке $[-1; 1]$;
2. синусоиды $y = \sin(\omega t)$ с частотами $\omega = 2\pi, \omega = 3\pi, \dots, \omega = 8\pi$ на отрезке $t \in [-1; 1]$;
3. синусоиды $y = \sin(2\pi t + \phi_0)$ с начальными фазами $\phi_0 = 0, \phi_0 = \pi/6, \dots, \phi_0 = 5\pi/6$ 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 $\operatorname{sh}(x), \operatorname{ch}(x)$ и $\operatorname{th}(x)$ на отрезке $x \in [-10; 10]$, для их вычисления воспользуйтесь их выражением через экспоненту.

Задание 17 Выполнять одно задание с номером $(n - 1)\%m + 1$, где n — номер в списке группы, а m — число задач в задании.

Для построенного в задании 16 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 один 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 18 Выполнять одно задание с номером $(n - 1)\%m + 1$, где n — номер в списке группы, а m — число задач в задании.

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 19 Выполнять одно задание с номером $(n - 1) \% t + 1$, где n — номер в списке группы, а t — число задач в задании.

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. $z = \text{tg}(xy)$,
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 6

Файлы

В данной главе будут рассмотрены встроенные средства Python для работы с файлами: открытие и закрытие, чтение и запись, создание, удаление и поиск файлов на диске. До сего момента мы рассмотрели только один способ чтения текстового файла, основанный на использовании функции `loadtxt` из модуля `numpy`. Вообще говоря, этот способ совсем особый, поскольку для него не нужно напрямую обращаться к файловому объекту. Зато он очень простой и потому мы настоятельно рекомендуем не забывать его даже тогда, когда вы освоите более сложные алгоритмы чтения.

6.1 Открытие файла

Прежде, чем работать с файлом, его надо создать или открыть. С этим замечательно справится встроенная функция `open`:

```
f = open('file.txt', 'w')
```

У функции `open` много параметров, нам пока важны 2 из них. *Первый* — это имя файла. В приведённом выше при мере файл `file.txt` будет создан в текущем каталоге. Если программа не меняла его — это тот же каталог, откуда она была запущена (заметим, что программа может быть запущена и не оттуда, где лежит её файл).

В общем случае путь к файлу может быть относительным или абсолютным. Пример абсолютного пути: `«D:/User/Data/file.txt»`, здесь мы обращаемся к файлу `«file.txt»`, лежащему в папке (директории) `«Data»`, которая в свою очередь лежит в директории `«User»` на диске `«D»` в системе Windows. Аналогичный путь в Unix-подобных системах, к которым относятся MacOS, Android, Linux, BSD, может иметь вид, например `«/home/user/Data/file.txt»`. Будьте внимательны, хотя в Windows для разделения уровней вложенности папок и файлов принято использовать так называемый «обратный слэш» (символ `«\»`), при задании пути в Python вместо него используется прямой (символ `«/»`), как в MacOS и Linux. Это

сделано по двум причинам: во-первых, для универсальности, во-вторых, потому что символ «\» является специальным, контролирующим и в строке явным образом не отображается: с его помощью набираются другие специальные символы такие, как «\t» — символ табуляции, «\n» — символ перехода на новую строку.

Часто вместо абсолютного пути удобнее использовать относительный. Например, если файл лежит в папке «Results», которая расположена в той же папке, что и программа, можно указать путь «Results/file.txt». Если же программа лежит в папке «Programs», которая, как и папка «Results», лежит в папке «User», тогда нужно написать «../Results/file.txt», где «..» обозначают родительскую (лежащую на один уровень вверх) папку вне зависимости от её истинного имени (можно написать «../../» и это будет означать на две папки вверх).

Второй аргумент функции `open` — это режим, в котором мы будем открывать файл. Возможные режимы приведены в следующей таблице:

Таблица 6.1. Режимы открытия файлов

Режим	Описание
'r'	открытие на чтение (является значением по умолчанию)
'w'	открытие на запись: содержимое уже существующего файла удаляется, если файла не существует, создается новый
'x'	открытие на запись, если файла не существует, иначе исключение
'a'	открытие на дозапись, информация добавляется в конец файла
'+'	открытие на чтение и запись
'b'	открытие в двоичном режиме
't'	открытие в текстовом режиме (является значением по умолчанию)

Некоторые режимы могут быть объединены, то есть, к примеру, 'rb' — чтение в двоичном режиме. По умолчанию режим равен 'rt'.

Как только файл был открыт и у вас появился файловый объект, вы можете получить следующую информацию о нём:

1. `f.closed` имеет значение `True`, если файл был закрыт и `False`, если он всё ещё открыт.
2. `f.mode` означает режим доступа, в котором был открыт файл.
3. `f.name` — имя файла.

Например:

```
f = open("file.txt", "w")
print("Имя файла: ", f.name)
```



```
print("Файл закрыт: ", f.closed)
print("Режим, в котором файл открыт: ", f.mode)
```

Будет выведено:

```
Имя файла: file.txt
Файл закрыт: False
Режим, в котором файл открыт: w
```

6.2 Запись в текстовый файл

Попробуем записать в файл список из ряда Фибоначчи:

```
mylist = []
fib1 = 0
fib2 = 1
n = 10
summa = 0
for i in range(n):
    summa = fib1 + fib2
    mylist.append(summa)
    fib1 = fib2
    fib2 = summa
print(mylist)
```

Во-первых, откроем файл для записи, как описывалось выше:

```
f = open('file.txt', 'w')
```

Во-вторых, осуществляем запись в файл с помощью метода `write`:

```
for el in mylist:
    f.write(str(el)+'\n')
```

Метод `f.write()` записывает любую строку в открытый файл. Поскольку аргументом метода является именно строка, пришлось явно преобразовать числа в строки с помощью встроенной функции `str`. Метод `f.write()` не добавляет символ переноса строки (`'\n'`) в конец файла. Метод возвращает целое число – количество записанных символов. После окончания работы с файлом его обязательно нужно закрыть с помощью метода `f.close()`.

После исполнения метода `f.close()` работать с файлом (читать, записывать) уже нельзя. Надо признать, что Python автоматически закрывает файл, если файловый объект, к которому он привязан, присваивается другому файлу. Однако хорошей практикой будет вручную закрывать файл командой `f.close()`. Получившийся файл можно посмотреть на рис. 6.1.

Записать список строк в файл можно используя файловой метод `f.writelines()`. Но этот метод работает, только если список состоит из данных

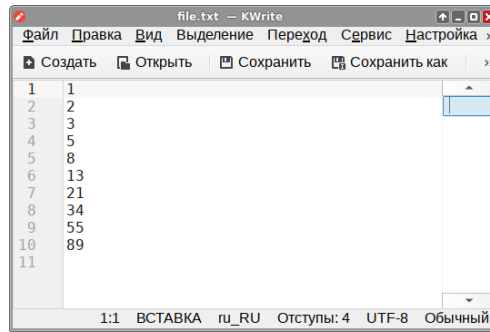


Рис. 6.1. Числа Фибоначчи, меньшие 100, записанные в текстовый файл.

строкового типа, и он не добавляет разделители строк автоматически. Фактически он просто в цикле вызывает метод `f.write()`. Вместо метода `f.writelines()` можно использовать следующую конструкцию `f.write('\n'.join(lines))`. Метод `join` объединяет список строк, ставя между элементами символ или строку, методом которого он является. Вместо символа конца строки `'\n'` можно использовать символы табуляции `'\t'` или пробелы `' '`. Пример работы этих двух методов показан на рис. 6.2.

```
f = open('file.txt', 'w')
lines = ['Мы', 'учимся', 'в', 'лучшем', 'вузе']
f.writelines(lines)
f.write('\n')
f.write(' '.join(lines))
f.write('\n')
f.write('\t'.join(lines))
f.close()
```

Если вы не доверяете своей памяти и боитесь забыть написать `f.close()`, то можно использовать оператор `with`, широко применяемый в программах на Python'e. Тогда программа записи данных в файл будет выглядеть следующим образом:

```
with open('file.txt', 'w') as f:
    for v in mylist:
        f.write(str(v)+'\n')
```

В главе про массивы кратко была рассмотрена функция сохранения данных в текстовые файлы `savetxt()`, лежащая в модуле `numpy`. Эту функцию разумно применять при сохранении числовых массивов. Функция `savetxt()` сама открывает и закрывает файл, сама производит форматирование: ставит символ конца строки, интервалы между столбцами, можно в качестве одного из аргументов

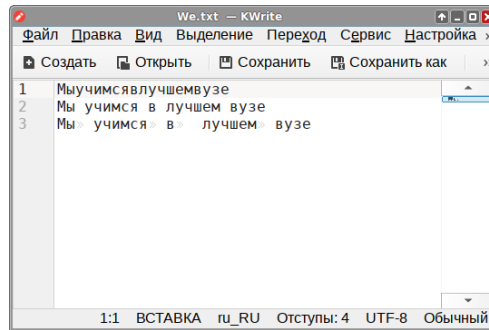


Рис. 6.2. Список строк, записанный в текстовый файл разными методами.

функции передавать формат для записываемых чисел. Но эта функция не позволяет записывать произвольные символы в произвольном порядке, нельзя записывать слова — только числа.

Функция `savetxt()` имеет много параметров, но нам хватит и первых пяти: `numpy.savetxt(f, X, delimiter=' ', newline='\n', fmt='%.18e')`. Первые два параметра обязательные — это путь к файлу или просто имя файла, в который будет производиться запись, и название числового одно- или двумерного массива, который нужно сохранить в текстовый файл. Далее идут три именованных необязательных параметра, которые передаются в строковом виде. Параметр `delimiter` задаёт разделитель между столбцами массива, например, табуляции `'\t'` или пробелы `' '`. Параметр `newline` задаёт разделитель между строками массива, как правило, это символ конца строки `'\n'`.

Параметр `fmt` определяет формат в котором записываются числовые элементы в массив. По умолчанию используется экспоненциальный формат `fmt='%.18e'`, означающий, что каждое число будет иметь 18 значащих цифр в дробной части, а целая часть будет менее 10 (представлена только 1 символом). Если представляемое число больше 10 или меньше 0.1, оно будет домножено на такую степень 10, чтобы влезть в данное представление, а соответствующий множитель будет записан после дробной части. То есть число 121 будет выглядеть как `1.210000000000000000e+02`, а число 0.003 — как `3.000000000000000062e-03` (заметьте «паразитные» 62 в конце числа, вызванные неточностью представления дробей, знаменатель которых не является степенью двойки). При таком представлении не происходит потеря данных, поскольку оно максимально соответствует представлению числе с плавающей запятой в компьютере. Но при чтении человеком такой формат очень неудобен. Вместо этого можно использовать формат с фиксированным числом значений после запятой вида `fmt='%6.3f'`, где 6 означает общее число символов, отводимых на представление данного числа, а 3 — число символов, выделенное на представление дробной части.

Экспоненциальный формат особо бессмысленен для целочисленных по типу массивов. В таком случае следует использовать формат, специально предназначенный для целых чисел: `fmt='%i'`. Минус такого представления в том, что числа будут занимать разное число символов, в результате форматирование столбцов «поедет». Если мы знаем максимальное число символов, например, 3, можно написать `fmt='%3i'` и все числа будут занимать минимум по 3 символа (при необходимости в начале добавляются пробелы), надо только всегда помнить, что для отрицательных чисел нужно зарезервировать ещё один символ под знак минус. Вместо пробелов можно заполнить пустое место нулями, используя формат с нулём перед числом символов — `fmt='%03i'`, в таком случае число 1 будет представлено как 001, а число -4 — как -04.

6.3 Чтение из текстового файла

Теперь попробуем прочитать этот список из получившегося файла. Откроем файл на чтение:

```
f = open('file.txt', 'r')
```

Для того, чтобы прочитать из этого файла информацию, есть несколько способов, но большого интереса заслуживают следующие три.

Первый, самый популярный метод чтения файла основан на том, что файловая переменная представляет собою итерируемый объект — итератор. Поэтому её содержимое можно получить с помощью цикла `for`, как это показано ниже. Попробуем считать файл, где в столбик записаны квадраты последовательных целых чисел:

```
f = open('file.txt')
for i in f:
    print(i)
```

Вывод программы:

```
1
2
3
5
8
13
21
```

34

55

89

Интересно, что вывод строк из файла перемежается пустыми строками. Это вызвано тем, что символы конца строки также считываются из файла и приводят к переходу на новую строку при печати, но функция `print` добавляет дополнительный перевод строки.

Рассмотрим более сложный пример с записью содержимого файла в список:

```
f = open('file.txt', 'r')
newlist = []
for line in f:
    newlist.append(int(line))
f.close()
print(newlist)
```

Вывод программы:

```
[1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

В этом примере переменная `line` (её можно назвать как угодно) последовательно принимает значения каждой строки с сохранением символа конца строки. Стандартная функция `int` преобразует строку к целому числу, причём пробелы, табуляции и символы конца строки игнорируются. Затем это число помещается в заранее созданный список `mylist`.

Умение читать числовые данные, записанные с столбик, из файлов и писать их в столбик в файл уже даёт вам возможность составлять реально полезные программы. Однако без существенного роста сложности можно рассмотреть более общий случай: в тестовый файл данные записаны в виде нескольких столбцов, разделенных пробелом или табуляцией, например (рис. 6.3):

```
9.600000 18.799999 26.799999 13.300000 7.600000
6.400000 17.200001 25.600000 10.900000 5.500000
3.700000 15.200000 24.200001 8.800000 4.300000
1.600000 12.700000 22.700001 7.000000 4.500000
...
```

Попробуем прочитать данные, записанные в третий столбец (помните, что в программировании отсчёт почти всегда идёт с нуля):

```
f = open('EEG.txt')
cannal = []
for line in f:
    mylist = line.split()
    cannal.append(float(mylist[3]))
```

The screenshot shows a text editor window titled 'EEG.txt - Kate'. The main area contains a table of numerical data with 18 rows and 4 columns. The first four columns are visible, showing values for each of the 16 channels across 18 samples. The values are floating-point numbers, some positive and some negative, representing the amplitude of the EEG signal.

1	9.600000	18.799999	26.799999	13.300000
2	6.400000	17.200001	25.600000	10.900000
3	3.700000	15.200000	24.200001	8.800000
4	1.600000	12.700000	22.700001	7.000000
5	-0.200000	10.000000	20.500000	5.700000
6	-1.600000	7.000000	18.400000	5.500000
7	-3.300000	3.700000	15.600000	5.700000
8	-5.100000	0.200000	12.700000	6.800000
9	-7.200000	-3.700000	9.600000	7.600000
10	-8.600000	-7.600000	6.400000	8.400000
11	-9.400000	-11.500000	3.700000	8.000000
12	-9.400000	-14.500000	1.400000	7.400000
13	-9.200000	-17.000000	-0.600000	6.200000
14	-8.800000	-18.400000	-1.600000	5.700000
15	-8.400000	-18.799999	-2.100000	5.900000
16	-7.400000	-18.200001	-2.100000	6.800000
17	-6.200000	-16.799999	-2.000000	8.200000
18	-4.900000	-15.000000	-1.600000	9.200000

Рис. 6.3. 16-канальная электроэнцефалограмма, записанная в 16 столбцов (на рисунке видны первые 4).

```
f.close()
for i in cannal[:100]:
    print(i)
```

Неполный вывод программы:

```
13.3
10.9
8.8
...
-11.1
-12.1
-12.9
```

Метод `split` разрезает строку на подстроки, возвращая список строк. В качестве разделителей используется любое число пробелов и табуляций, стоящих подряд. В данном случае переменная `mylist` содержит список строк: `['9.600000', '18.799999', '26.799999', '13.300000', '7.600000']`.

Второй способ чтения информации из файла — метод `read`, читающий весь файл целиком, если был вызван без аргументов, и n байт, если был вызван с аргументом (целым числом n):

```
b1 = f.read()
```

Попробуем считать этим способом наш файл с рядом Фибоначчи. Если затем напечатать переменную `b1` из интерактивного режима, выведется следующее:

```
'1\n2\n3\n5\n8\n13\n21\n34\n55\n89\n'
```

Аналогичный результат можно получить, если в файле программы использовать стандартную функцию `repr` внутри `print`, что приведёт к выводу объекта без форматирования (символы конца строки `\n` будут интерпретированы как есть):

```
print(repr(b1))
```

Обратите внимание, что после вызова метода `read()` на файловом объекте, если вы повторно вызовете `read()`, то увидите лишь пустую строку. Это происходит потому, что после первого прочтения указатель позиции чтения находится в конце файла. Для того, чтобы узнать позицию указателя в байтах, можно использовать метод `tell()`. Например:

```
f = open("file.txt", "r")
f.read(10)
print("Я на позиции: ", f.tell())
```

Будет выведено:

```
Я на позиции: 10
```

Метод `tell()` сообщает в скольких байтах от начала файла мы сейчас находимся (непечатные символы: пробелы, табуляции, символы конца строки — тоже считаются). Его антипод — метод `seek` предназначен для перемещения по файлу. Синтаксис метода таков: `f.seek(offset, from)`, где аргумент `offset` указывает, на сколько байт перейти, опциональный аргумент `from` указывает на позицию, с которой начинается движение: 0 означает начало файла, 1 — нынешняя позиция, 2 — конец файла.

К сожалению, тут есть важный подводный камень: `read` считает в символах, а `tell` и `seek` — в байтах. В случае, если один символ занимает один байт, всё хорошо. И тут всё начинает зависеть от кодировки. Вспомним, что на самом деле никакие символы в памяти не хранятся, а хранятся на самом деле целые числа — их коды. В зависимости от договорённости те или иные числа могут интерпретироваться при выводе на экран или на печать как одни или другие символы. Такая договорённость и называется кодировкой. Первая известная кодировка — ASCII — имела вообще 7-битное представление (старший бит в байте не использовался), в ней умещались латинские буквы, цифры, знаки пунктуации, некоторые специальные и непечатные символы. Чуть позже вторая половина бита была использована для отражения символов национальных алфавитов. Так возникли в том числе три популярные кодировки русского языка: `cp1251` (основная кодировка Windows), `IBM866` — «старая» кодировка DOS, до сих пор используемая в консоли Windows и `koï8r` — кодировка Unix. Однобайтные кодировки тем хороши, что в них, во-первых, символы занимают минимально доступный адресуемый объём памяти — 1 байт, во-вторых, длина символа постоянная, а значит, по номеру символа легко найти его сам в тексте. К сожалению, для международных требований однобайтные кодировки не подходят: уместить в один байт не удаётся даже несколько европейских алфавитов вместе. Поэтому, например, при

использовании однобайтных кодировок напечатать русско-немецкий словарь не получится: в одном и том же тексте могут встречаться только $2^8 = 256$ различных символов. Не говоря уже о многих языках со слоговой и иероглифической письменностями: там даже для символов одного языка не хватит одного байта, даже если выбросить все латинские буквы, цифры и знаки препинания.

Для решения проблемы множества языков и символов были придуманы многобайтные кодировки. Самые распространённые из них сейчас — UTF8 и UTF32. В кодировке UTF32 каждый символ занимает ровно 4 байта (32 бита, то есть $2^{32} = 4294967296$ вариантов). Такой способ позволяет, во-первых, представить все известные в мире символы всех языков в рамках одной кодировки (и остаётся ещё очень много места на будущее), во-вторых, обеспечивает высокую скорость поиска символов в строке, так как они по-прежнему занимают одинаковое число байтов. Следовательно, для поиска n -ного символа достаточно перейти к $4n$ -ному байту. Основной минус UTF32 состоит в том, что она очень неэкономична с точки зрения требований к памяти. Ведь наиболее употребимые символы — это символы ASCII, а для большинства национальных алфавитов хватит двух байт. Поэтому была придумана кодировка UTF8, в которой длина символа переменная: стандартные ASCII символы представляются одним байтом, русские буквы (символы кириллицы) и ряд значков, например, **№**, — двумя, а ещё более редкие символы (иероглифы, например) — тремя и более. По умолчанию Python считает, что файл записан именно в этой кодировке. Однако использование UTF8 осложняет навигацию по файлу, так как разные символы занимают разное число байт. Методы `tell` и `seek` оперируют байтами, а не символами, поэтому их прямое использование в текстовом режиме ненадёжно.

Из-за того, что файл может быть записан в одной кодировке, например, в популярной у нас `cp1251`, а мы читаем его в другой (по умолчанию это UTF8), его чтение будет происходить неверно. В таком случае нужно при открытии файла прямо указать кодировку:

```
f = open('file.txt', 'r', encoding='cp1251')
```

Третий способ состоит в использовании методов `readline()` — читает в строковую переменную текущую строку целиком, включая символ конца строки, и `readlines()` — читает весь оставшийся файл в список строк, где каждый элемент списка — одна строка файла, символы конца строки также будут присутствовать. Пример использования метода `readline()`:

```
f = open('file.txt', 'r')
print(f.readline())
```

Вывод программы:

```
1
```

Пример использования метода `readlines()`:

```
f = open('file.txt', 'r')
print(f.readlines())
```


Вывод программы:

```
['1\n', '2\n', '3\n', '5\n', '8\n', '13\n', '21\n', '34\n', '55\n', '89\n']
```

Наконец, есть ещё один — *четвёртый* способ чтения текстового файла, основанный на использовании функции `loadtxt()` из модуля `numpy`. Он был рассмотрен ранее в главе про массивы. Вообще говоря, этот способ совсем особенный, поскольку для него не нужно напрямую обращаться к файловому объекту и даже создавать его, функция делает всё «за вас». При этом есть определённые ограничения: `loadtxt` предназначена исключительно для считывания одномерных и двумерных массивов. Это значит, что с её помощью нельзя читать смешанные файлы, где наряду с числами есть строки (например, имена), файлы с переменным числом чисел в строке, файлы, где в качестве десятичного разделителя в числах используется запятая или числа разделены знаками, отличными от общепринятых пробельных символов `,` `\t` и `\n` или их комбинаций, например, запятыми или двоеточиями. Некоторые ограничения `loadtxt` можно преодолеть, используя дополнительные параметры. Функция `loadtxt`, как и функция `savetxt`, имеет много параметров, но мы остановимся на шести из них. Разберём следующий пример:

```
a = numpy.loadtxt(f, dtype='float', delimiter='\t',
                 skiprows=0, usecols=3, unpack=False)
```

У этой функции один обязательный параметр `f` — файл или строка с именем файла. Далее идут необязательные именованные параметры: `dtype` — строковый тип — определяет тип данных выходного массива, значение по умолчанию — `float`, возможны варианты: `'int'`, `'bool'`, `'complex'`. Параметр `delimiter` — строковый тип — строка-разделитель между значениями в текстовом файле, по умолчанию используется пробел. Параметр `skiprows` — целочисленный тип — указывает количество строк, которые необходимо пропустить, по умолчанию равно 0. Часто используется при работе с реальными данными. Например, практически все приборы, оснащённые АЦП, вначале записывают в файл служебную информацию (название прибора, частоту дискретизации, количество уровней квантования, название каналов измерения и др.), а уже потом — сами данные. Параметр `usecols` — целочисленный тип — указывает, какие столбцы будут считаны. Столбцы нумеруются с 0. По умолчанию `usecols=None`, что соответствует чтению всех столбцов в файле. Задание целого числа `usecols=3` или кортеж с одним значением `usecols=(3,)` позволит считать только третий столбец из файла. Если надо считать несколько столбцов, можно передать кортеж с их номерами. Параметр `unpack` логического типа по умолчанию равен — если этот параметр равен `False`. Если установить `unpack=True`, то все столбцы текстового файла могут быть разделены и присвоены отдельным переменным. Например, если написать `a, b, c = loadtxt(...)`, то три столбца из файла будут помещены в массивы `a`, `b` и `c`. Если число считываемых столбцов не совпадает с числом присваиваемых переменных, у вас будут проблемы.

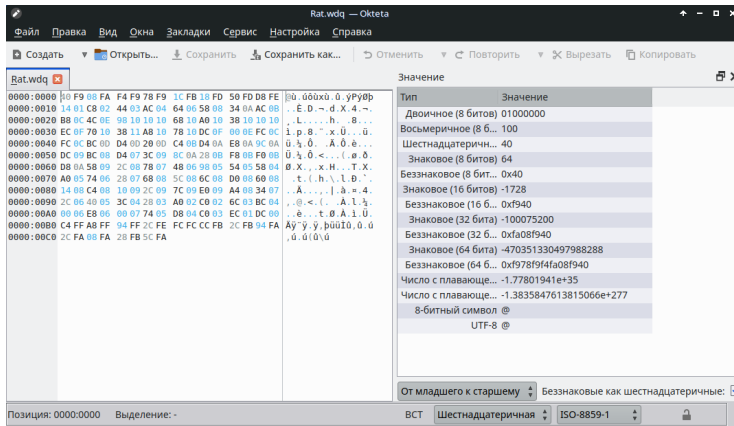


Рис. 6.4. Бинарный файл, открытый в бесплатном просмотрщике с открытым исходным кодом Okteta.

И вновь попробуем прочитать данные, записанные в третий столбец файла “EEG.txt”:

```
import numpy as np
X = np.loadtxt('EEG.txt', delimiter='\t', usecols=3)
print(X)
```

Вывод программы – наш искомый третий столбец:

```
[13.3    10.9    8.8    ... 50.799999 51.    50.]
```

6.4 Чтение из файла в двоичном режиме. Модуль struct

Хранение числовой информации в текстовом режиме часто неудобно и сильно избыточно, потому что в таком случае данные занимают гораздо больше памяти и их приходится каждый раз преобразовывать из строкового представления в числа и наоборот. К тому же многие приборы, оснащённые аналого-цифровыми преобразователями, выдают результат в целых числах. Для чтения данных в виде чисел в большинстве языков программирования есть двоичный режим открытия файла.

Попробуем прочитать с помощью метода `read` файл в двоичном режиме. Для сопоставления того что мы видим в Python и того, что лежит в файле полезно воспользоваться каким-нибудь шестнадцатеричным редактором (мы использовали Okteta из KDE, но вы можете взять и любой другой) — см. рис. 6.4:

```
40F908FAF4F978F91CFB18FD50FDD8FE...
```

В данном примере в файле были записаны целые числа размером в 2 байта, т. е. 4 16-ричные цифры, но догадаться об этом только по самому файлу нельзя, нужна именно дополнительная информация. С точки зрения файла — это просто поток байтов. Чтобы разобраться с этим примером, вначале переведём первое число 40F9 в двоичную форму: 1111 1001 0100 0000, переставив местами байты, поскольку в большинстве современных систем принято хранить числа наоборот: от младших байтов к старшим. Так как старший бит 1, то число — отрицательное. Теперь нужно узнать его модуль. Для этого инвертируем все биты (0000 0110 1011 1111) и прибавляем единицу (0000 0110 1100 0000). В итоге получается число 1728, или, с учётом знака —1728. Если первый бит 0, число положительное. Его не нужно инвертировать, можно просто перевести в десятичную систему счисления. С переводом второго числа можете потренироваться самостоятельно.

Теперь попробуем в Python прочитать этот файл в двоичном режиме. Для этого нам понадобится функция `unpack` из стандартного модуля `struct`. Модуль `struct` служит для конвертирования данных из байтов в типы данных языка Python и может быть полезен не только при чтении из файла, но также и при взаимодействии с программами на других языках, отправляющими вашей программе нетипизированную информацию. Функции `pack` и `unpack` позволяют «упаковывать» и «распаковывать» данные из байтов в другие типы на основе специальной форматной строки, состоящей из кодов форматирования.

Начнём с чтения первого числа, оно будет автоматически упаковано в кортеж. Не забываем, что для чтения в двоичном режиме файл необходимо открывать с аргументом `'rb'`:

```
from struct import unpack
f = open('Rat.wdq', 'rb')
buffer = f.read(2)
A = unpack('h', buffer)
print (A[0])
f.close()
```

Вывод программы:

-1728

Функция `unpack` принимает два аргумента. Первый — строка, описывающая формат из которого преобразуются данные. Основные форматы рассмотрены в таблице 6.2.

Таблица 6.2. Форматы считывания бинарных файлов. Для целочисленных форматов маленькая буква соответствует знаку, большая — беззнаковому.

Формат	Тип данных	Размер, байт
?	логический	1
b и B	целый	1

h и H	целый	2
i и I	целый	4
q и Q	целый	8
f	действительный	4
d	действительный	8
s	строковый	

Теперь попробуем прочитать сразу три первых числа:

```
from struct import unpack

f = open('Rat.wdq', 'rb')
buffer = f.read(3*2)
A = unpack('hhh', buffer)
print(A)
f.close()
```

Вывод программы в виде кортежа из трёх чисел:

```
(-1728, -1528, -1548)
```

Далее читаем все числа в список с помощью цикла `for`:

```
from struct import unpack
f = open('Rat.wdq', 'rb')
mylist = []
for i in range(10):
    buffer = f.read(2)
    A = unpack('h', buffer)
    mylist.append(A[0])
print(mylist)
f.close()
```

Вывод программы в виде списка из ста чисел:

```
[-1728, -1528, -1548, -1672, -1252, -744, -688, -296, 276, 712]
```

У большинства файлов, хранящих числовую или иную типизированную информацию, нет строчек в привычном смысле, поэтому читать командой `for i in f:` не очень разумно, хотя в ряде случаев и получится. Но можно узнать размер файла в байтах (как и любого другого) с помощью функции из модуля `os`. Он, кстати, часто нужен при работе с файлами, директориями, операционной системой. Для этого нужно написать следующее:

```
LengthFile = os.path.getsize('Rat.wdq')
```

Зная, что одно число занимает 2 байта, модифицируем программу следующим образом:

```
import struct
import os
NameFile = 'Rat.wdq'
f = open(NameFile, 'rb')
mylist = []
LengthFile = os.path.getsize(NameFile)
for i in range(LengthFile//2):
    buffer = f.read(2)
    A = struct.unpack('h', buffer)
    mylist.append(A[0])
print(mylist)
f.close()
```

6.5 Модуль pickle

Бывает так, что необходимо сохранить в виде файла некую довольно большую и сложную структуру данных, передать её или использовать в другой программе. Например, словарь, где ключи — кортежи чисел, а элементы — списки, содержащие разнотипные объекты. Записывать такой файл в виде текста, а потом читать — плохая идея. Сделать это можно, но придётся потратить много времени. К тому же нужно договориться с автором второй программы, которая будет читать ваш файл, о формате данных. Конечно, существуют специальные форматы разметки вроде `xml` или `json`, но, во-первых, не всякий объект легко конвертируется из Python в эти форматы, во-вторых, их нужно специально учить.

В Python есть встроенный механизм для быстрого и простого сохранения и чтения любых данных встроенных типов, реализованный с помощью стандартного модуля `pickle`. Это касается всех стандартных типов вроде `int`, `float`, `str`, `list` и др., а также любых их комбинаций. Кроме того, многие широко используемые модули вроде `numpy` также предоставляют свой функционал для поддержки типов данных, введённых в этих модулях. Поэтому часто можно видеть, что с использованием `pickle` читаются и сохраняются сложные объекты, составленные из переменных нестандартных типов. Вообще говоря, есть рекомендация, что любой новый модуль, предоставляющий новый класс объектов, должен обеспечить для них поддержку методов модуля `pickle`.

Представим себе, что имеется база данных студентов и она неполная в том смысле, что часть данных для некоторых может отсутствовать. К тому же она пополняется в процессе: в неё заносятся сданные за каждый семестр дисциплины с оценками. Это значит, что число семестров у разных студентов различно, как и число и состав дисциплин в семестре, а некоторые дисциплины могут появляться в разных семестрах, например, курс мат. анализа длится с 1-го по третий. Здорово, что организовать такую структуру в Питоне можно стандартными

средствами, используя комбинацию из вложенных словарей и списков¹. При этом записать всю эту структуру в текстовый файл ещё можно, а вот чтобы считать и перевести её обратно в приемлемый вид, потребуется много сил.

```
import pickle
a = [{'№': 20190601, 'фамилия': 'Иванов',
      1: {'мат. анализ': 4, 'геометрия': 5, 'история': 4}},
     {'№': 20180586, 'фамилия': 'Кузнецова', 'имя': 'Мария',
      1: {'мат. анализ': 3, 'геометрия': 5, 'история': 4},
      2: {'мат. анализ': 4, 'физиология': 4, 'философия': 5},
      3: {'мат. анализ': 4, 'материаловедение': 5,
          'численные методы': 4, 'биофизика': 5}}]
f = open('бд_студенты.pickle', 'wb')
pickle.dump(a, f)
f.close()
```

Обратите внимание, что перед записью файл необходимо открыть на запись как бинарный и обязательно не забыть закрыть! В данном примере мы придумали нестандартное расширение `.pickle`, операционная система почти наверняка не знает, как открывать и просматривать такие файлы. Вообще говоря, файл можно назвать как угодно, даже вовсе без расширения, потому что просмотреть его с помощью чего-нибудь отличного, от чтения средствами самого Python, скорее всего не получится. В этом — основной минус модуля `pickle`.

В действительности из всего модуля `pickle` вам пригодятся всего две функции: уже использованная выше `dump` для записи в файл и её партнёр `load` для чтения. Поэтому чаще всего эти функции импортируют напрямую, как это сделано ниже, чтобы не писать каждый раз имя модуля при обращении к ним. Рассмотрим пример программы для записанного выше файла:

```
from pickle import load
f = open('бд_студенты.pickle', 'rb')
a = load(f)
f.close()
for val in a:
    print(val)
```

Получится вот такой вывод:

```
{'№': 20190601, 'фамилия': 'Иванов',
 1: {'мат. анализ': 4, 'геометрия': 5, 'история': 4}}
{'№': 20180586, 'фамилия': 'Кузнецова', 'имя': 'Мария',
 1: {'мат. анализ': 3, 'геометрия': 5, 'история': 4},
 2: {'мат. анализ': 4, 'физиология': 4, 'философия': 5},
 3: {'мат. анализ': 4, 'материаловедение': 5, 'численные методы':
 4, 'биофизика': 5}}
```

¹Как сделать это ещё двумя способами, используя классы и базы данных, будет показано далее

Заметьте, что открывать файл на чтение нужно тоже в бинарном формате. В конце мы написали цикл по всем элементам списка, чтобы напечатать их построчно. Интерпретация результатов чтения файлов с помощью функции `load` из модуля `pickle` основана на важном предположении, что вы *действительно понимаете*, что лежит в файле. Если структура того, что там находится, для вас неведома, разобрать её будет отдельной задачей. В таком случае полезно начинать с вывода типа прочитанного объекта, далее, если это сложный тип вроде словаря или списка, — его длины и т. п. К счастью, при использовании `pickle` программисты, как правило, договариваются между собою, что и в какой последовательности будут передавать. Важно только оформить эту договорённость в виде справочного файла или хотя бы комментария, чтобы через год или два, когда вам срочно понадобятся сохранённые при помощи `pickle` данные, не запутаться. Ведь просмотреть какими-либо стандартными средствами файл типа `pickle` не получится.

В рассмотренном примере в файл был сохранён единственный объект — список. Никто не запрещает сохранять в файл несколько объектов, например, запись для каждого студента:

```
f = open('бд_студенты.pickle', 'wb')
dump(len(a), f)
for key in a:
    dump((key, a[key]), f)
f.close()
```

Теперь при чтении до цикла отдельно считываем число записей про студентов, а уже затем в цикле, зная, сколько объектов было записано в файл, их сами:

```
from pickle import load
f = open('бд_студенты.pickle', 'rb')
num = load(f)
for i in range(num):
    student = load(f)
    print(student)
f.close()
```

В приведённом примере мы явно знали, что в первой записи в файле содержится число объектов, а далее сами эти объекты. Эти априорные знания позволили написать правильную программу. При работе с объектами, записанными с использованием модуля `pickle`, подобные априорные знания обязательны!

Если попробовать считать из файла больше объектов, чем там есть (мы для примера написали в заголовке считывающего цикла `range(1+1)` вместо `range(1)`), будет выдана ошибка. В сообщении об ошибке указано, что ошибка произошла в строке 5 и сама эта строка, а также тип ошибки — достигнут конец файла `EOFError` и мы вышли за пределы:

```
File "load_multi.py", line 5, in <module>
student = load(f)
```

`EOFError: Ran out of input`

Важно! Так как ошибка возникла далеко не на первой итерации цикла, программа фактически сделала почти всю работу и даже вывела, что нужно, на экран. Следует помнить, что при работе с циклом ошибки часто возникают не на первой итерации, то есть цикл успевает сработать несколько раз. Это значит, что цикл надо как-то подправить, разобравшись, а не удалять целиком.

В качестве заключения: есть ещё две причины, почему иногда следует использовать `pickle` вместо текстовых файлов помимо удобства. Во-первых, данные в двоичном представлении занимают существенно меньше места на диске, чем в текстовом, что может быть важно при работе с объектами большого размера. Во-вторых, бинарные данные гораздо быстрее считываются и записываются, чем текстовые, так как при чтении и записи последних происходит разбор символов, а затем преобразование типов, при чтении бинарников же просто определённым последовательностям байтов в программе назначается тип и всё.

6.6 Работа с операционной системой. Модули `os` и `os.path`

Модуль `os` предоставляет множество функций для работы с операционной системой, причём их поведение, как правило, не зависит от ОС, поэтому программы остаются переносимыми. Здесь будут приведены наиболее часто используемые из них.

Будьте внимательны: некоторые функции из этого модуля поддерживаются не всеми ОС. Поэтому для начала можно выяснить тип вашей операционной системы: `os.name` — имя операционной системы. Доступные варианты: `'posix'` (Linux), `'nt'` (Windows), `'mac'` (Mac OS X), `'os2'` (OS/2), `'ce'`, `'java'` (виртуальная машина Java).

Теперь посмотрим, в какой папке мы находимся в текущий момент: `os.getcwd()` — текущая рабочая директория. Должно получиться что-то подобное: `'/home/marina/Python/Учёба'`.

Так же полезными будут следующие команды:

- `os.chdir(path)` — смена текущей директории.
- `os.listdir(path=".")` — выдаёт список файлов и директорий в текущей директории (`"."` можно заменить на любой другой корректный путь к директории — будет выдан список директорий и файлов, лежащих по данному пути).
- `os.mkdir(path)` — создаёт директорию. Выдаёт `OSError`, если директория уже существует.

Посмотрим, как это работает:

```
import os
way = '../..//BOOKS' # Поднимаемся на две папки вверх и перейдём в BOOKS
```



```
os.chdir(way)
for f in os.listdir('Учебное пособие Часть II'):
    print(f)
```

Результат:

```
TeX Часть II
Оглавление.odt
Программы Часть II
Распределения
Рисунки Часть II
Тестовые системы
```

Переменной `way` присвоили путь к директории. Затем с помощью функции `chdir` перешли в эту директорию, и с помощью функции `listdir` получили список всех файлов, лежащих в ней. Можно попробовать написать ту же программу с использованием функции `makedirs`:

```
import os
way = '../..//BOOKS'
os.chdir(way)
os.makedirs('Учебное пособие Часть II')
```

но в данном случае получим ошибку:

```
Traceback (most recent call last):
File "/home/marina/BOOKS/УЧЕБНОЕ ПОСОБИЕ Часть II/
Программы Часть II/os_1.py", line 4, in <module>
os.makedirs('Учебное пособие Часть II')
FileExistsError: [Error 17] File exists: 'Учебное пособие Часть II'
```

Для того, чтобы программа не выдавала такую ошибку, нужно использовать функцию `exists`, которая лежит в модуле `path`, который в свою очередь является вложенным модулем в модуль `os` и реализует некоторые полезные функции для работы с путями. В частности, `os.path.exists(путь)` возвращает `True`, если путь указывает на существующий путь.

Тогда программу грамотнее писать следующим образом:

```
import os
way = '/home/marina/BOOK/Методичка по Python'
os.chdir(way)
if not os.path.exists('Учебное пособие Часть II'):
    os.makedirs('Учебное пособие Часть II')
for f in os.listdir('Учебное пособие Часть II'):
    print(f)
```

В модуле `os.path` есть и другие полезные функции:

- `os.path.isfile(путь)` проверяет, является ли путь файлом (возвращает `True` если является, иначе — `False`).

- `os.path.isdir(путь)` проверяет, является ли путь директорией (возвращает `True` если является, иначе — `False`).
- `os.path.join(путь1[, путь2[, ...]])` — соединяет пути слешами с учётом особенностей операционной системы, путей может быть сколько угодно.
- `os.path.split(путь)` — разбивает путь на кортеж (голова, хвост), где хвост — последний компонент пути, а голова — всё остальное. Хвост никогда не начинается со слеша (если путь заканчивается слешем, то хвост пустой). Если слешей в пути нет, то пустой будет голова.
- `os.path.splitext(путь)` — разбивает путь на пару (путиям, расширение), где расширение начинается с последней точки.

Посмотрим, как это работает:

```
import os
way = '/home/marina/BOOKS/Методичка по Python/Учебное пособие Часть II'
os.chdir(way)
ListNameFile = []
for f in os.listdir('Распределения'):
    ListNameFile.append(os.path.splitext(f)[0])
print(ListNameFile)
```

Вывод в виде списка:

```
['Хи-квадрат с 2 степенями свободы', 'Хи-квадрат с 4 степенями свободы',
'Хи-квадрат с 6 степенями свободы', 'Хи-квадрат с 8 степенями свободы']
```

Теперь элементы этого списка можно использовать для записи результатов вычислений в отдельный файл с новым путём:

```
os.path.join(path, 'Results' + ListNameFile[0]+'_hist.png')
```

6.7 Примеры решения заданий

Пример задачи 19 (Запись в текстовый файл) Создайте (откройте на запись) текстовый файл «Телефоны.txt». Запишите в него информацию в следующем формате: <<Улица Дом Квартгиря Номер телефона>>. Данные вводите с клавиатуры. Запрашивайте у пользователя необходимую информацию, пока не введёте 10 номеров. При записи в файл в качестве разделителя между столбцами используйте табуляцию ('\t') или пробел (' ').

Решение задачи 19 Решение можно записать, например, следующим образом:

```
import struct
import os
file = open('Телефоны.txt', 'w')
for i in range(3):
    a = input('Улица: ')
    b = input('Дом: ')
    c = input('Квартира: ')
    d = input('Номер телефона: ')
    file.write(a+'\t'+b+'\t'+c+'\t'+d+'\n')
file.close()
```

При сложении строк здесь полезно также вспомнить о встроенном методе `join`, так что предпоследняя строка переписется в виде:

```
file.write('\t'.join([a, b, c, d])+ '\n')
```

Хотя сильного сокращения записи добиться в данном случае не удалось, у такого способа есть 3 значимых преимущества: во-первых, если мы захотим изменить символ-соединитель, менять нужно будет только 1 раз, во-вторых, теперь легко добавлять и удалять соединяемые элементы, в-третьих, начиная уже с 2-3 операндов такой способ работает быстрее, чем прямое сложение и, хотя вам сейчас это не заметно, есть множество ситуаций, например, при веб-программировании, когда даже такой сравнительно короткий отрезок кода, будучи выполнен миллионы раз, существенно тормозит программу.

Пример задачи 20 (Считывание из текстового файла) Считайте текстовый файл «Телефоны.txt». Найдите в нём любой номер телефона (вводится с клавиатуры) и выведите адрес, по которому расположен этот телефон, на экран.

Решение задачи 20 Решение можно представить следующим образом:

```
file = open('Телефоны.txt', 'r')
Dic = {} # создаём словарь
for i in file:
    mylist = i.split()
    Address = 'ул.'+mylist[0]+'', д.'+mylist[1]+'', кв.'+mylist[2]
    Tel = mylist[3]
    Dic[Tel] = Address # в словаре номер телефона - ключ, а адрес - значение
Number = input('Номер телефона: ')
if Number in Dic:
    print(Dic[Number])
else:
    print('Такого номера не существует')
file.close()
```

Пример задачи 21 (Считывание из текстового файла) Файл EEG.txt (возьмите у преподавателя) содержит запись ЭЭГ (электроэнцефалограммы) человека, включающую сигналы с 16 электродов (отведений). Каждому отведению соответствует столбец чисел. Столбцы разделены символом табуляции. Создайте программу, выделяющую в отдельный текстовый файл отведение с заданным номером. Название текстового файла составьте из названия исходного файла без расширения, номера отведения и расширения '.txt'.

Решение задачи 21 Вот пример решения:

```
fr = open('EEG.txt', 'r')
NumberChannel = int(input('Номер отведения: '))
cannal = []
for i in fr:
    mylist = i.split()
    cannal.append(mylist[NumberChannel])
fr.close()
fw = open('EEG'+str(NumberChannel)+'.txt', 'w')
for can in cannal:
    fw.write(can+'\n')
fw.close()
```

Заметим, что здесь мы не стали преобразовывать считанные данные в числа, а оставили их в строковом представлении, поскольку производить с ними вычисления не требовалось. Интересно, что метод `join` может и здесь помочь вам обойтись без цикла:

```
fw.write('\n'.join(cannal))
```

Пример задачи 22 (Считывание из бинарного файла) Файл Rat.hex содержит запись внутричерепной ЭЭГ крысы из 8 отведений в бинарном представлении. Отведения записаны последовательно. Числа беззнаковые однобайтные. Первые 230 байт содержат служебную информацию, её нужно пропустить. Создайте программу, выводящую информацию из одного из отведений с заданным номером в отдельный текстовый (.txt) файл. Запишите первые 1000 измерений. Название текстового файла составьте из названия исходного файла без расширения '.hex', номера отведения и расширения '.txt'.

Решение задачи 22

```
from struct import unpack
import os.path
NumberChannel = int(input('Номер отведения: '))
```

```

length = 1000 # количество измерений
Rat = 'Rat.hex'
fr = open(Rat, 'rb')
fr.seek(230) # пропускаем служебную информацию
buffer = fr.read(length*1)
mylist = unpack(length*'B', buffer)
# Т.к. 8 отведений записаны последовательно, делаем переывборку:
channel = mylist[NumberChannel::8]
file = os.path.join(os.getcwd(), Rat) # указываем полный путь к файлу
fw = open(os.path.splitext(file)[0] + str(NumberChannel) + '.txt', 'w')
for i in range(len(channel)):
    fw.write(str(channel[i])+'\n')
fw.close()

```

Пример задачи 23 (Работа с операционной системой) Напишите две независимые программы. Первую программу, которая создаёт папку *Распределения*, если она ещё не создана. Далее записывает в неё четыре текстовых файла *.txt*, содержащих последовательность величин, распределённых по закону χ^2 с 2, 4, 6 и 8 степенями свободы, соответственно.

Затем напишите вторую программу, которая переходит в папку *Распределения*, находит в ней все текстовые файлы, строит для них гистограммы распределений, сохраняет графики в файлы с таким же названием, как исходный текстовый файл, но другим расширением (*.png*, *.jpg* или *.pdf*).

Решение задачи 23

Первая программа:

```

from numpy import random
import os

def chi(s):
    x = 165 + random.chisquare(s, 1000)
    f = open('Chi-квадрат с '+str(s)+' степенями свободы.txt', 'w')
    for v in x:
        f.write(str(v)+'\n')
    f.close()

os.chdir('.')
if not os.path.exists('Распределения'):
    os.mkdir('Распределения')

os.chdir('Распределения')
chi(2)
chi(4)

```

```
chi(6)
chi(8)
```

Вторая программа:

```
import matplotlib.pyplot as plt
import os
plt.rcParams['font.sans-serif'] = ['Arial']

def grapher (NF):
    f = open(NF, 'r')
    x = []
    for line in f:
        x.append(float(line))
    f.close()

    Name = os.path.splitext(NF)[0]

    plt.figure(figsize=(5,5))
    plt.title(Name)
    plt.ylim([0, 0.3])
    plt.xlim([160, 200])
    plt.hist(x, bins=10, ec='black', density=True, color='grey')

    plt.savefig(Name+'.png')

os.chdir('../Распределения')
for NameFile in os.listdir('.'):
    if os.path.splitext(NameFile)[1]=='.txt':
        grapher(NameFile)
```

6.8 Задания на работу с файлами и с операционной системой

Задание 20 Работа с текстовыми файлами. Задания выполняйте все по порядку.

1. Запишите в файл с помощью функции `write()` три столбца: в первый — целые числа от 1 до 100, во второй — их квадраты, в третий — их кубы. Столбцы можно разделить табуляциями `'\t'` или пробелами `' '`.
2. Запишите в файл с помощью функции `write()` три столбца: в первый числа x от 0 до 2π с маленьким шагом (например, $\pi/24$), во второй — значения $\sin(x)$, в третий — значения $\cos(x)$. Столбцы можно разделить табуляциями `'\t'` или пробелами `' '`.
3. Создайте (откройте на запись) текстовый файл «Зачёт.txt». Запишите в него информацию о своей группе в следующем формате: «Фамилия Имя

Отчество Оценка». Данные вводите с клавиатуры. Вначале введите количество учеников в вашей группе. Далее запрашивайте у пользователя необходимую информацию, пока не введёте всех учеников. При записи в файл в качестве разделителя между столбцами используйте табуляцию (`'\t'`) или пробел (`' '`).

4. Читайте текстовый файл «Зачёт.txt». Найдите в нём оценку любого учащегося (фамилия вводится с клавиатуры) и выведите её на экран.
5. Файл `EEG.txt` (возьмите у преподавателя) содержит запись ЭЭГ (электроэнцефалограммы) человека, включающую сигналы с 16 электродов (отведений). Каждому отведению соответствует столбец чисел. Столбцы разделены символом табуляции. Создайте программу, выделяющую в отдельный текстовый файл отведение с заданным номером. Название текстового файла составьте из названия исходного файла без расширения, номера отведения и расширения `' .txt'`.

Задание 21 Работа с бинарными файлами. Задания выполняйте все по порядку.

1. Файл `Rat.wdq` содержит бинарное представление записи с 4 отведений внутричерепной ЭЭГ крыс. Отведения записаны последовательно. Числа знаковые двухбайтные. Создайте программу, выводящую информацию из одного из отведений с заданным номером в отдельный текстовый (`.txt`) файл. Запишите первые 100 измерений. Название текстового файла составьте из названия исходного файла без расширения `' .wdq'`, номера отведения и расширения `' .txt'`.
2. Файл `rat_01_02m1.wdq` содержит реальные данные от крысы генетической линии `WAG/Rij`, страдающей эпилепсией. Файл организован следующим образом: первые 5296 байт — служебная информация, которую при считывании можно пропустить методом `seek`, затем записаны целые двухбайтные знаковые числа, каждые 4 последовательных числа (8 байт) представляют собою записи 4 каналов за один и тот же момент времени. Читайте файл до конца, пропустив служебную информацию. Раскодируйте данные, используя модуль `struct`, функцию `unpack`. Чтобы декодировать большие объёмы данных полезно использовать умножение числа на символ. Постройте графики для всех четырёх каналов один под другим. Чтобы выбрать один канал полезно использовать срезы списков с шагом. По оси абсцисс отложите время: интервал между последовательными значениями времени составляет для данного файла 0.00195 с, время можно сгенерировать с помощью функций `arange` или `linspace` из модуля `numpy`.

Задание 22 Работа с операционной системой. Задания выполняйте все по порядку.

Напишите две независимые программы. Первую программу, которая создаёт папку **Распределения**, если она ещё не создана. Далее записывает в неё три текстовых файла `.txt`, содержащих последовательность величин, распределённых по равномерному закону, нормальному закону и закону χ^2 с 5 степенями свободы. Необходимые функции ищите в Главе 7, пункт 7.3.

Затем напишите вторую программу, которая переходит в папку **Распределения**, находит в ней все текстовые файлы, строит для них гистограммы распределений, сохраняет графики в файлы с таким же названием, как исходный текстовый файл, но другим расширением (`.png`, `.jpg` или `.pdf`).

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`:

```
import numpy as np
A = np.array([[2, 3], [-1, 5]])
b = np.array([-7, -16])
x = np.linalg.solve(A, b)
print(x)
D = np.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. Для этого следует использовать функцию (`np.linalg.eig(a)`), возвращающую кортеж из одномерного массива собственных чисел и матрицы собственных векторов. Есть функция `linalg.norm(x)` для расчёта нормы вектора или оператора, которая может работать с набором векторов в виде матрицы, если указать параметр `axis` аналогично тому, как он используется для функций и методов типа `min`, `sum` и др. Уже реализованы все основные методы разложения матриц:

- `np.linalg.cholesky` — разложение Холецкого;
- `np.linalg.qr` — QR разложение;
- `np.linalg.svd` — сингулярное разложение;

Есть функция для реализации метода наименьших квадратов — `np.linalg.lstsq(a, b)`.

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**) некоторую функцию **ток(напряжение)**.

Предположим, что у нас есть результаты измерения тока стока i_c (список **y**) при изменении напряжения на затворе $U_{3И}$ (массив **x**). Данная вольт-амперная характеристика (ВАХ) показана на рис. 7.1. Программа для её отображения будет иметь следующий вид:

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import rcParams
rcParams['font.sans-serif'] = ['Liberation_Sans', 'Arial']
x = np.arange(0, -4.5, -0.5)
y = [50, 35, 22, 11, 4, 0, 0, 0]
plt.plot(x, y, 'o', color='grey')
plt.title('Сток-затворная характеристика')
plt.xlabel(r'$U_{3И}$, В$', fontsize=18)
plt.ylabel(r'$i_{c}$, мкА$', fontsize=18)
plt.xlim(-5, 0.5)
plt.ylim(-10, 60)
plt.show()
```

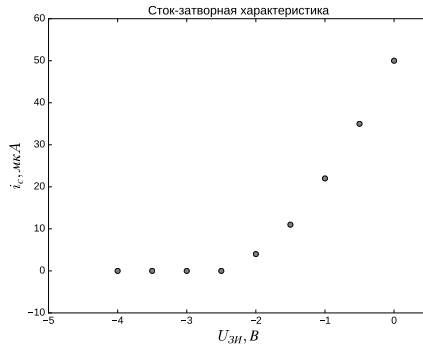


Рис. 7.1. Сток-затворная характеристика полевого транзистора (ВАХ) — результаты эксперимента.

Кусочно-линейная аппроксимация заменяет ВАХ транзистора двумя отрезками:

$$i_c = \begin{cases} \alpha_0 + \alpha_1 U_{зи} & \text{при } U_{зи} > U_{отс} \\ 0 & \text{при } U_{зи} < U_{отс} \end{cases} \quad (7.1)$$

Используя метод наименьших квадратов, произведём аппроксимацию этих двух участков ВАХ прямыми линиями, см. 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 функцией `show()`:

```
import numpy as np
import matplotlib.pyplot as plt
x = np.arange(0, -4.5, -0.5)
y = [50, 35, 22, 11, 4, 0, 0, 0, 0]
b = 0; e = 4; r = e - b
a = np.ones((r, 2))
a[:, 1] = x[1:r+1]
result = np.linalg.lstsq(a, y[b:e])
ya1 = a @ result[0]
plt.plot(x[b:e], ya1, color='black')
print(result[0])
b = 5; e = 9; r = e - b
a = np.ones((r, 2))
a[:, 1] = x[1:r+1]
result = np.linalg.lstsq(a, y[b:e])
ya1 = a @ result[0]
plt.plot(x[b:e], ya1, color='black')
```

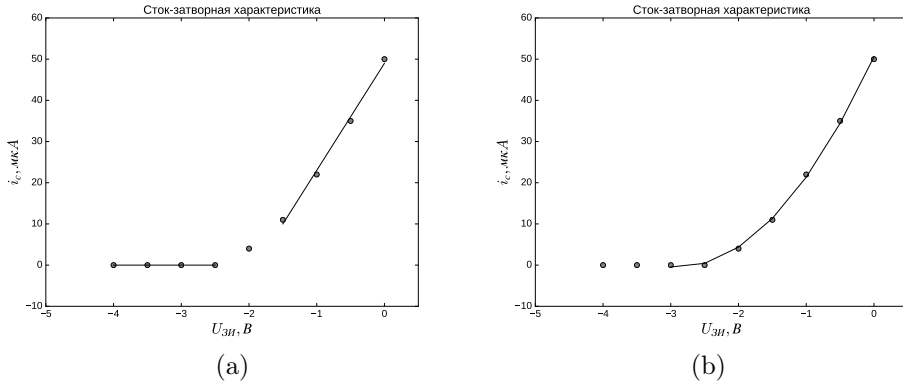


Рис. 7.2. Различные подходы к аппроксимации вольт-амперной характеристики полевого транзистора: (а) — кусочно-линейная, (б) — параболическая.

```
print(result[0])
```

Обратите внимание на использование оператора матричного умножения `@` в коде. 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 ($b = 0$; $e = 7$), см. рис. 7.2(b).

$$i_c = \alpha_0 + \alpha_1 U_{зи} + \alpha_2 U_{зи}^2 \quad (7.2)$$

Для получения параболической аппроксимации (см. рис. 7.2(b)) нужно заменить код из предыдущего листинга на следующий:

```
b = 0; e = 7; r = e - b
a = np.ones((r, 3))
a[:, 1] = x[1:r+1]
a[:, 2] = x[1:r+1]**2
result = np.linalg.lstsq(a, y[b:e])
ya1 = a @ result[0]
plt.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 длиной 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` — `rfft` и `irfft` позволяют произвести прямое Фурье-преобразование над действительными 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:

```
import numpy as np
t = np.arange(0, 1, 0.1)
x = np.sin(2*np.pi*t)
fx = np.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 -3.88578059e-16-5.00000000e+00j
 -8.70502608e-17+1.64296685e-16j  1.98072563e-16-7.23881735e-16j
  4.99600361e-16+4.44089210e-16j -1.11022302e-16+0.00000000e+00j]
```

Все остальные значения порядка 10^{-15} или ещё меньше следует признать нулями с точностью вычислений.

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 $\pi/2$ 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]
```

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(a)):

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import rcParams
rcParams['font.sans-serif'] = ['Liberation_Sans', 'Arial']
dt = 0.1
t = np.arange(0, 100, dt)
x = 2.5 * np.sin(2*np.pi*t)
fx = np.fft.rfft(x) / (len(x)/2) # нормировка на половину длины
ряда
fn = 1/(2*dt) # частота Найквиста
freq = np.linspace(0, fn, len(fx)) # массив частот
plt.plot(freq, abs(fx), color='black')
plt.title('Амплитудный спектр')
plt.xlabel('Частота, Гц');
plt.ylabel('Напряжение, В')
plt.ylim([0, 3])
plt.savefig('FFT.png')
plt.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.

Одинокая синусоида единичной амплитуды — не очень интересный пример. Рассчитаем и построим график бигармонического сигнала, состоящего из двух синусоид разных амплитуд (рис. 7.3(b)):

```
import numpy as np
t = arange(0, 2, 0.1)
x = 2*sin(2*pi*t) + 3*cos(pi*t)
```

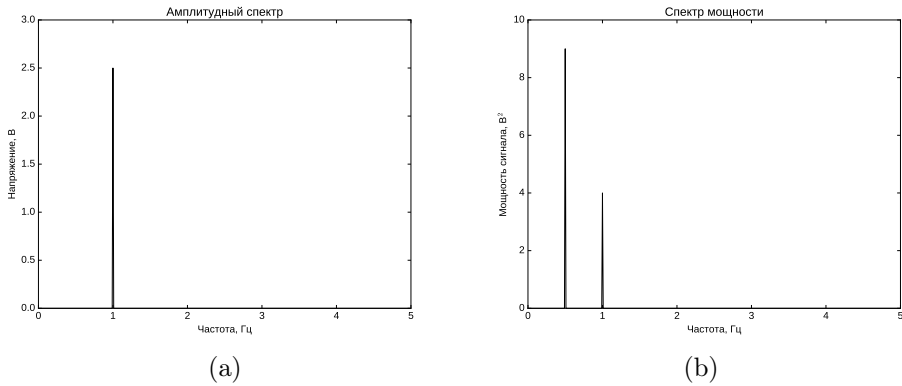


Рис. 7.3. Амплитудный спектр моногармонического сигнала — (а) и спектр мощности бигармонического сигнала — (б).

```
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 1.79958894e-31
 1.67632942e-31 1.28189897e-31 3.09627905e-31 1.79958894e-31
 1.14877869e-31 1.43967115e-31 1.97215226e-33]
```

Строим спектр мощности, увеличив длину ряда:

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import rcParams
rcParams['font.sans-serif'] = ['Liberation_Sans', 'Arial']
dt = 0.1
t = np.arange(0, 100, dt)
x = 2*np.sin(2*np.pi*t) + 3*np.cos(np.pi*t)
Px = np.abs(np.fft.rfft(x)/(0.5*len(t)))**2
fn = 1/(2*dt)
plt.plot(np.linspace(0, fn, len(Px)), Px, color='black')
plt.title('Спектр мощности')
plt.xlabel('Частота, Гц');
plt.ylabel(r'Мощность сигнала, В\$\wedge\{2}\$');
plt.ylim([0, 10])
plt.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 .

```
import numpy as np
x = np.random.uniform(-2, 1, 10)
print(x)
y = np.random.normal(5, 0.5, 10)
print(y)
```

Вывод программы:

```
[-0.0277742 -1.15596933 -1.87993463 -0.67060949 -1.86110957 0.47038121
-0.68869176 -1.15116482 -1.76515744 -0.48517902]
[4.77757507 4.57808828 4.99160443 5.07122687 4.65301469 5.06715848
5.44171544 4.73569554 4.55952495 4.90044623]
```

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`:

```
import numpy as np
x = np.random.randint(-10, 10, 10)
print(x)
y = np.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 24 (Определитель матрицы) Найдите определитель матрицы. Матрицу возьмите из текстового файла, созданного при выполнении задания №11.

Решение задачи 24

```
import numpy as np
M = np.loadtxt('Matrix.txt')
print(np.linalg.det(M))
```

Пример задачи 25 (Система линейных уравнений) Решите систему линейных уравнений, матрицу коэффициентов и столбец свободных членов прочитайте из текстовых файлов, созданных в задании №11. Запишите в новый текстовый файл полученные корни.

Решение задачи 25

```
import numpy as np
M = np.loadtxt('Matrix.txt')
V = np.loadtxt('Vector.txt')
print(np.linalg.det(M))
```

Пример задачи 26 (Аппроксимация параболой) 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(a)).

Решение задачи 26

```
import numpy as np
import matplotlib.pyplot as plt
x = np.arange(-6, 6, 0.1) # диапазон
y = x**2 - x - 6 # парабола
r = np.random.normal(0, 2, len(x)) # белый шум
z = y + r
a = np.ones((len(x), 3))
a[:, 1] = x
a[:, 2] = x**2
result = np.linalg.lstsq(a, z)
za = np.dot(a, result[0]) # аппроксимирующая кривая
```

```
plt.plot(x, z, 'o', color='red')
plt.plot(x, za, color='blue')
print(result[1]/len(x)) # ошибка аппроксимации
plt.show()
```

Значение ошибки аппроксимации должно быть порядка квадрата стандартного отклонения шума.

Пример задачи 27 (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 27

```
import numpy as np
import matplotlib.pyplot as plt
x = np.arange(-6, 6, 0.1)
y = x**2-x-6
r = np.random.normal(0, 2, len(x))
y1 = y + r
r = np.random.normal(0, 2, len(x))
y2 = y + r
r = np.random.normal(0, 2, len(x))
y3 = y + r
y = np.column_stack([y1, y2, y3])
plt.errorbar(x, y.mean(axis=1), yerr=y.std(axis=1),
             marker='.', color = 'green', ecolor='blue')
plt.savefig('errorbars.png')
plt.show()
```

Пример задачи 28 (Генерация массивов случайных чисел)

Сгенерируйте случайные векторы из n действительных значений с равномерным и нормальным распределением, а также из n целых чисел.

Решение задачи 28

```
from numpy import random
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 29 (Случайные перестановки) Сгенерируйте массив-лесенку длины n . Случайно перемешайте его. На экран выведите изначальный и перемешанный массивы.

Решение задачи 29

```
import numpy as np
n = int(input('Размер массива: '))
x = np.arange(n, 0, -1)
print(x)
y = np.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 30 (Периодограмма синуса) Рассчитайте и постройте периодограмму для функции $\sin(x)$ на отрезке $x \in [\pi; \pi]$.

Решение задачи 30 Результат можно увидеть на рис. 8(a). Грубость графика обусловлена малым объёмом данных.

```
import numpy as np
from math import pi
import matplotlib.pyplot as plt
t = np.arange(-pi, pi, pi/12)
x = np.sin(2*pi*t)
Px = np.abs(np.fft.rfft(x)/(0.5*len(t)))**2
fn = 1/(2*pi/12)
freq = np.linspace(0, fn, len(Px))
plt.grid(True); plt.plot(freq, Px, color='red')
plt.show()
```

Пример задачи 31 (Сложный шум) Сгенерируйте случайный процесс, представляющий собою сумму равномерно распределённых на отрезке $[-10; 10]$ случайных величин и нормально распределённых случайных величин с параметрами $(0; 1)$, длиной в 10000 значений. Постройте гистограмму его распределения.

Решение задачи 31 Возможный результат можно увидеть на рис. 8(b). От запуска к запуску картинка будет несколько варьировать.

```
import numpy as np
import matplotlib.pyplot as plt
n = 10000
x = np.random.uniform(-10, 10, n)
y = np.random.normal(0, 1, n)
z = x + y
plt.hist(z, bins=100, density=True, color='green')
plt.show()
```

7.5 Задания на использование встроенных библиотек numpy

Задание 23 Найдите определитель матрицы. Матрицу возьмите из текстового файла, созданного ранее, либо у преподавателя.

Задание 24 Решите систему линейных уравнений. Матрицу коэффициентов и столбец свободных членов прочитайте из текстовых файлов, созданных ранее. Запишите в новый текстовый файл полученные корни.

Задание 25 Выполнять одно задание с номером $(n - 1)\%m + 1$, где n — номер в списке группы, а m — число задач в задании.

Сгенерируйте набор значений заданной функции с шумом. Аппроксимируйте его полиномом второй степени. Оцените ошибку аппроксимации. Постройте график. Функции:

1. парабола $y = x^2 - x - 6$ на отрезке $[-4; 4]$ с белым шумом, распределённым по нормальному закону с параметрами $(0; 1)$;
2. парабола $y = x^2 - x - 6$ на отрезке $[-4; 4]$ с белым шумом, распределённым по равномерному закону на отрезке $[-10; 10]$;
3. парабола $y = x^2 - x - 6$ 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 26 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 27 Выполнять одно задание с номером $(n - 1)\%m + 1$, где n — номер в списке группы, а m — число задач в задании.

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 28 Выполнять одно задание с номером $(n - 1)\%m + 1$, где n — номер в списке группы, а m — число задач в задании.

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 29 Выполнять одно задание с номером $(n - 1)\%m + 1$, где n — номер в списке группы, а m — число задач в задании.

Рассчитайте и постройте периодограмму — оценку спектра мощности:

1. сигнала $y(t)$, полученного по формуле $4 \sin(\pi t + \pi/8) - 1$ на отрезке $t \in [-10; 10]$ с шагом 0.05;
2. сигнала $y(t)$, полученного по формуле $2 \cos(t - 2) + \sin(2t - 4)$, на отрезке $t \in [-20\pi; 10\pi]$ с шагом $\pi/20$;
3. нормального шума, параметры выберите сами;
4. равномерного шума, параметры выберите сами.

Задание 30 Выполнять одно задание с номером $(n - 1)\%m + 1$, где n — номер в списке группы, а m — число задач в задании.

Сгенерируйте случайный процесс длиной 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 единичной 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 АЛТ»

СЫСОЕВА Марина Вячеславовна
СЫСОЕВ Илья Вячеславович

**ПРОГРАММИРОВАНИЕ ДЛЯ «НОРМАЛЬНЫХ»
С НУЛЯ НА ЯЗЫКЕ PYTHON**

В двух частях
Часть 1

Второе издание, исправленное и дополненное

Учебник

Ответственный редактор: В.Л. Черный
Оформление обложки: А.С. Осмоловская
Вёрстка: Сысоева М.В., Сысоев И.В.

ООО «Базальт СПО»
Адрес для переписки: 127015, Москва, а/я 21
Телефон: (495)123-47-99. E-mail: sales@basealt.ru
<http://basealt.ru>

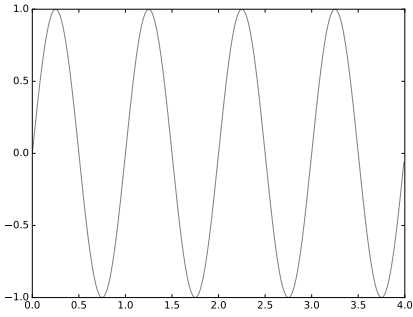
Напечатано с готового оригинал-макета

Подписано в печать 21.04.2023. Формат 70x100/16.
Гарнитура Computer Modern. Печать офсетная. Бумага офсетная.
Усл. печ. л. 14,95 [+0.33]. Тираж 999 экз. Изд. номер 024.

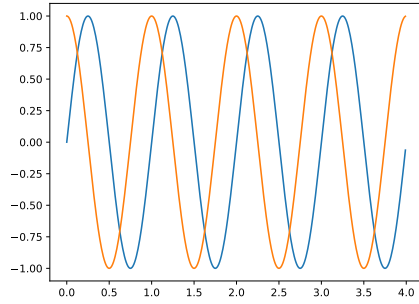
Издательство ООО «МАКС Пресс» Лицензия ИД N 00510 от 01.12.99 г.
119992, ГСП-2, Москва, Ленинские горы, МГУ им. М.В. Ломоносова,
2-й учебный корпус, 527 к.
Тел. 8(495)939-3890/91. Тел./Факс 8(495)939-3891.

Отпечатано в полном соответствии с качеством
предоставленных материалов в ООО «Фотоэксперт»
109316, г. Москва, Волгоградский проспект, д. 42,
корп. 5, эт. 1, пом. I, ком. 6.3-23Н

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 (бирюзовой) и косинусоиды (оранжевою) линиями — (b).

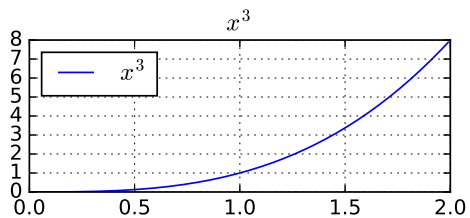
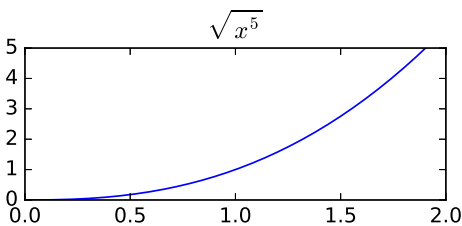
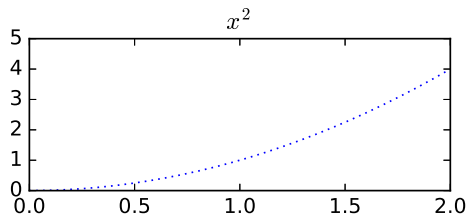
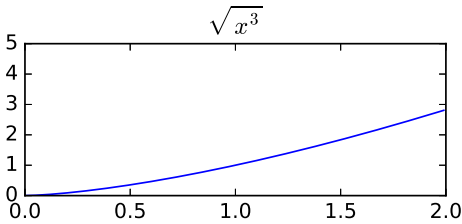
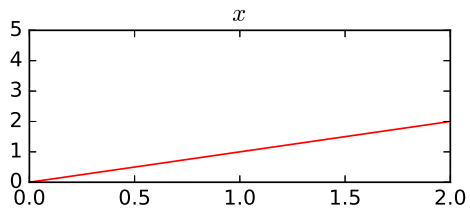
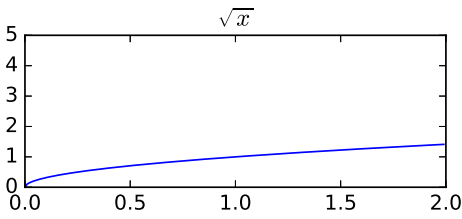
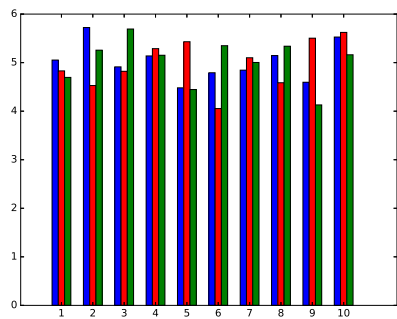
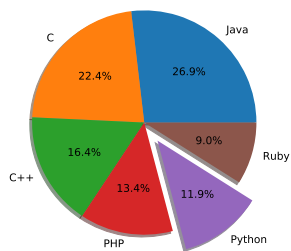


Рис. 2. Пример построения нескольких графиков на одном полотне.

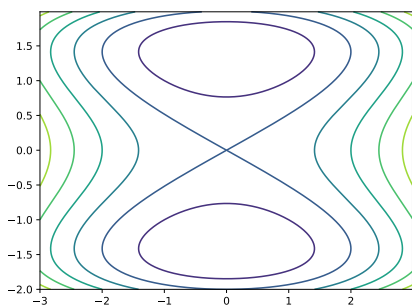


(a)

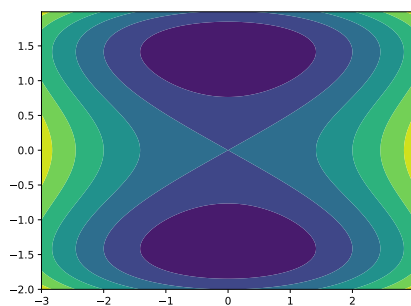


(b)

Рис. 3. Пример столбцовой (a) и круговой (b) диаграмм.



(a)



(b)

Рис. 4. Пример построения контурных диаграмм: (a) — использованы линии, (b) — использована заливка.

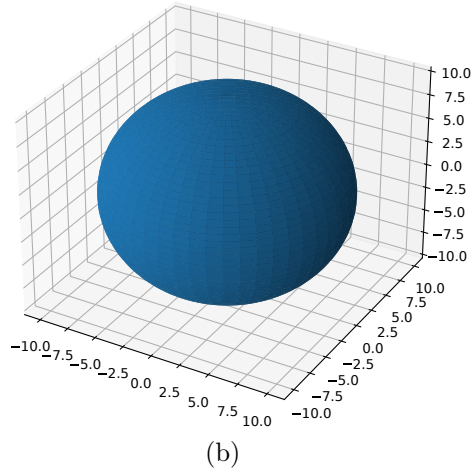
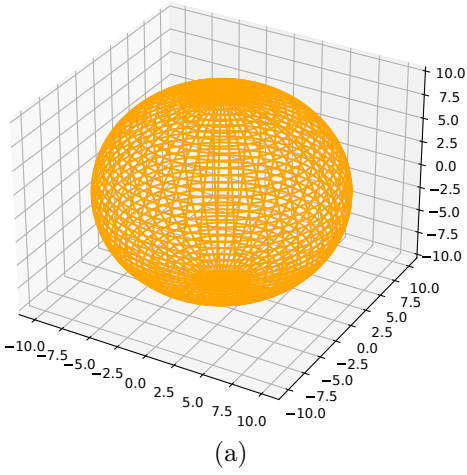


Рис. 5. 3D-каркас (а) и 3D-поверхность (б).

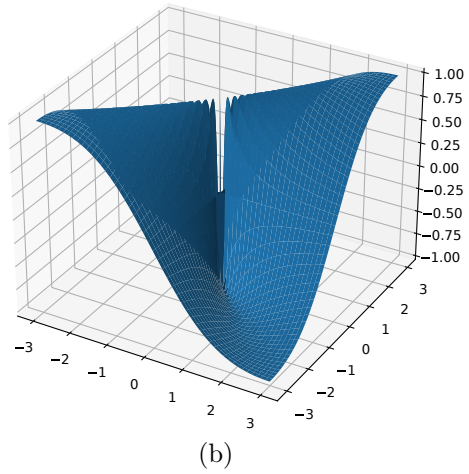
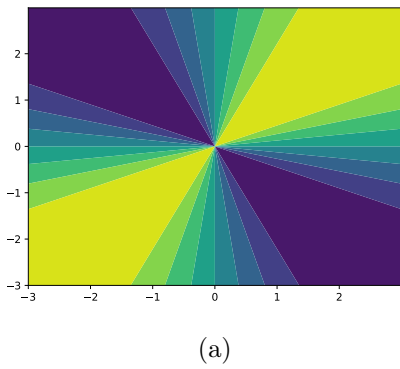
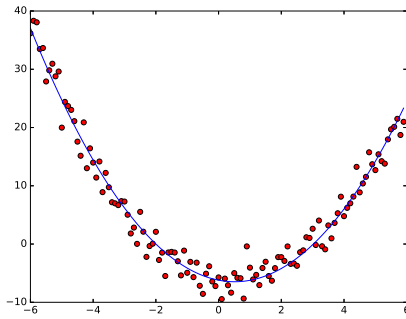
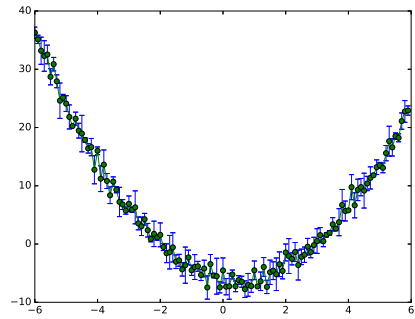


Рис. 6. Иллюстрация к задаче 18.

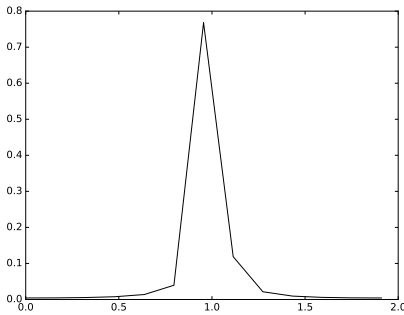


(a)

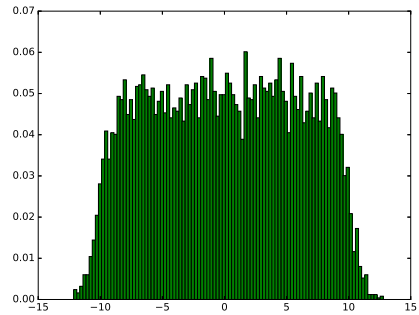


(b)

Рис. 7. Иллюстрации к задачам 26 — (а) и 27 — (b). На рис. (а) 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 $\pi/12$ — (а) и гистограмма сложного (суммы равномерного и нормального) распределения — (b).