Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Челябинский государственный университет»

С. А. Хайбрахманов

ОСНОВЫ НАУЧНЫХ РАСЧЁТОВ НА ЯЗЫКЕ ПРОГРАММИРОВАНИЯ PYTHON

Учебное пособие

Челябинск
Издательство Челябинского государственного университета
2019

УДК 004(075.8)
ББК 3973я7
Х154

Печатается по решению редакционно-издательского совета
Челябинского государственного университета

Р е ц е н з е н т ы:
кафедра вычислительной механики
факультета математики, механики и компьютерных технологий
Южно-Уральского государственного университета (НИУ);

К. Л. Маланчев, кандидат физико-математических наук,
научный сотрудник отдела релятивистской астрофизики
Государственного астрономического института имени П. К. Штернберга
Московского государственного университета имени М. В. Ломоносова

Хайбрахманов, С. А.

Х154 Основы научных расчётов на языке программирования Python : учеб. пособие / С. А. Хайбрахманов. — Челябинск : Изд-во Челяб. гос. ун-та, 2019. — 96 с.

ISBN 978-5-7271-1629-6

Излагаются основы научных расчётов на языке программирования Python. Приводятся и демонстрируются базовые возможности библиотек NumPy для работы с многомерными массивами, Matplotlib для визуализации данных и SciPy для научных расчётов и анализа данных. Рассматриваются примеры решения некоторых математических и физических задач с помощью библиотек NumPy, Matplotlib и SciPy.

Предназначено для практических занятий и самостоятельной работы по дисциплине «Компьютерные технологии в науке и образовании» для студентов, обучающихся по направлениям 03.04.02 Физика, 03.04.03 Радиофизика.

Ил. 11. Библиогр.: 6 назв.

УДК 004.432.2(075.8)
ББК 3973.22я73-1

ISBN 978-5-7271-1629-6

© Хайбрахманов С. А., 2019
© Челябинский государственный университет, 2019

Оглавление

Предисловие 5

Используемые обозначения 8

Г л а в а 1. Основы программирования на языке Python 9

1.1. Базовые сведения о Python 9
1.2. Установка Python 10
1.3. Интерактивный режим работы 12
1.4. Типы данных 13
1.5. Операции и логические операторы 20
1.6. Ввод/вывод 21
1.7. Управляемые блоки кода в Python 23
1.8. Функции 27
1.9. Работа с библиотеками 30
1.10. Написание файлов исходного кода 32
1.11. Создание и использование собственных модулей . . . 35
Примеры решения задач 37
Задания для самостоятельной работы 40
Вопросы для самоконтроля 41
Рекомендуемая литература 41

Г л а в а 2. Библиотека NumPy для работы с многомерными массивами 43

2.1. Общие сведения 43
2.2. Создание массивов NumPy 44
2.3. Доступ к элементам массива 46
2.4. Функции для работы с массивами 50
2.5. Универсальные функции 52
2.6. Работа с файлами 53
Задание для самостоятельной работы 55
Вопросы для самоконтроля 57
Рекомендуемая литература 57

Глава 3. Библиотека визуализации Matplotlib 58
3.1. Общие сведения 58
3.2. Создание простого рисунка с параметрами по умолчанию 59
3.3. Создание рисунка с настраиваемыми параметрами . . 61
3.4. Добавление нескольких панелей на рисунок 68
3.5. Двумерные графики 69
3.6. Визуализация данных из текстовых файлов 76
3.7. Математические символы на рисунках 78
Задание для самостоятельной работы 78
Вопросы для самоконтроля 79
Рекомендуемая литература 79

Глава 4. Библиотека SciPy для научных и инженерных расчётов 80
4.1. Общие сведения 80
4.2. Численное интегрирование 82
4.3. Решение обыкновенных дифференциальных уравнений 83
4.4. Интерполяция 88
Варианты заданий для самостоятельной работы . . . 91
Вопросы для самоконтроля 92
Рекомендуемая литература 93

Заключение 94

Список литературы 96

Предисловие

Учебное пособие посвящено основам программирования на языке Python и работы с библиотеками NumPy, Matplotlib и SciPy для проведения научных расчётов.

Язык Python является интерпретируемым высокоуровневым языком программирования, поддерживающим разработку программ с использованием структурного, функционального, объектно-ориентированного и др. подходов. Отличительными чертами языка являются кроссплатформенность, динамическая типизация, автоматическое управление памятью, интроспекция, высокоуровневые структуры данных. Синтаксис Python легко читаем. С помощью Python создаётся как системное, так и прикладное программное обеспечение, программы с графическим интерфейсом, веб-приложения, программы для научных расчётов и т. д. Python имеет богатую стандартную библиотеку. Кроме того, для Python написано большое количество прикладных библиотек, в том числе для научных расчётов, которые позволяют решать ряд математических задач без необходимости самостоятельной разработки алгоритмов.

Проведение как теоретических, так и практических научных исследований включает в себя решение различных математических задач, таких как: нахождение корней алгебраических уравнений, задачи линейной алгебры, вычисление интегралов, решение систем обыкновенных дифференциальных уравнений, аппроксимация и интерполяция данных и т. д. Важным элементом анализа научных результатов является визуализация данных. Найти точное решение математических задач в реальных приложениях зачастую не удаётся, что обусловливает необходимость использования алгоритмов численного решения.

К настоящему времени разработано, реализовано и протестировано большое количество библиотек и программных пакетов как для аналитического, так и для численного решения научных задач. Использование таких пакетов избавляет учёных

от необходимости тратить время на разработку и реализацию алгоритмов решения стандартных математических задач и позволяет сконцентрироваться на анализе получаемых решений.

Наиболее известными коммерческими пакетами для проведения научных и инженерных расчётов являются пакет программ MATLAB, системы компьютерной алгебры Mathcad, Maple и Mathematica. Указанные программные пакеты имеют продвинутые и удобные графические интерфейсы, собственные высокоуровневые языки программирования и позволяют решать большое количество математических и инженерных задач, а также визуализировать данные и получать рисунки высокого качества. Однако возможность использования данных пакетов индивидуальными исследователями и небольшими организациями затруднена по причине высокой стоимости их лицензий.

Существует ряд бесплатных программных пакетов и библиотек для проведения научных расчётов, анализа и визуализации данных. Среди них можно выделить пакеты для численных расчётов Scilab (URL: https://www.scilab.org/) и Octave (URL: https://www.gnu.org/software/octave/), систему компьютерной алгебры Maxima (URL: http://maxima.sourceforge.net/ru/index.html), систему для статистических расчётов и визуализации данных на основе языка программирования R (URL: https://www.r-project.org/), а также библиотеки NumPy для работы с многомерными массивами, Matplotlib для визуализации данных и SciPy для научных и инженерных расчётов на языке программирования Python. Совместные возможности библиотек NumPy, Matplotlib и SciPy близки к возможностям MATLAB. Данные библиотеки в настоящее время активно развиваются и являются одним из лидирующих бесплатных инструментов для решения стандартных математических задач и анализа данных.

Целью данного пособия является ознакомление студентов с основными возможностями языка Python и библиотек NumPy, Matplotlib и SciPy. Изложенный материал основан на многолетнем опыте автора по использованию указанных библиотек для проведения расчётов, анализа и визуализации данных. В пособии изложен необходимый минимум знаний для написания простейших программ на Python и применения библиотек NumPy, Matplotlib и SciPy. Для чтения пособия не требуется предварительных зна-

ний языка Python, однако необходимы базовые знания в области структурного, функционального и объектно-ориентированного программирования. Для лучшего понимания материала приводится сравнение основных понятий и синтаксиса Python с распространёнными языками программирования C и Pascal. Для получения дополнительной информации в конце каждой главы приведён список рекомендуемой литературы.

Учебное пособие организовано следующим образом. В первой главе рассматриваются основы программирования на Python: описывается интерактивный режим работы и написание файлов исходного кода, основные типы данных и операторы, работа с функциями, библиотеками и модулями. Во второй главе описываются возможности библиотеки NumPy по работе с многомерными массивами. В третьей главе приводятся основные способы визуализации данных с помощью библиотеки Matplotlib: построение одномерных и двумерных графиков, настройка подписей и внешнего вида рисунков, сохранение рисунков в файлы, визуализация данных из файлов. В четвёртой главе рассматриваются некоторые возможности библиотеки SciPy: численное интегрирование, решение систем обыкновенных дифференциальных уравнений, интерполяция.

Используемые обозначения

Будем выделять названия библиотек, модулей, функций, типов данных, имена файлов и зарезервированные слова жирным шрифтом Times New Roman, например: **Matplotlib**, **pyplot**, **plot()**, **float**, **test.py**, **for** и т. д.

Переменные в тексте обозначаются наклонным шрифтом Times New Roman: *x*, *y*, *'string'* и т. д.

Строки программного кода в тексте записываются шрифтом Consolas чёрного цвета и сопровождаются линией по левому полю:

```
import matplotlib.pyplot as plt
```

Комментарии и строки в тексте программ выделяются шрифтом Consolas серого цвета и сопровождаются линией по левому полю:

```
# однострочный комментарий

"""
Строка, разделённая
На несколько строчек
"""
```

Ожидание ввода команды в окне интерактивной оболочки IPython:

```
In    [N]: |
```

где N — натуральное число (1, 2, 3 ...), указывающее текущий номер команды и номер результата от момента начала работы с оболочкой.

Результат выполнения предыдущей команды:

```
Out [N]: <Ans>
```

где N — натуральное число (1, 2, 3 ...), указывающее номер результата, <Ans> — результат выполнения команды.

Глава 1
Основы программирования на языке Python

В данной главе рассматриваются основы программирования на языке Python и базовые функции интегрированной среды разработки Spyder (URL: https://www.spyder-ide.org/). Теоретический материал сопровождается примерами интерактивной работы в IDE Spyder и написания файлов исходного кода. Приводятся примеры программ. В завершение приводится список заданий для самостоятельной работы и вопросы для самоконтроля.

1.1. Базовые сведения о Python

Python является высокоуровневым языком программирования (URL: https://www.python.org). Основные особенности языка:

- Python поддерживает структурное, функциональное и объектно-ориентированное программирование.
- Python — интерпретируемый язык программирования. Это означает, что программный код на языке Python не компилируется перед его исполнением в отличие от таких языков, как C и Fortran. Код выполняется специальной программой-интерпретатором. Интерпретатор считывает высокоуровневую программу — исходный код — и, напрямую взаимодействуя с операционной системой, выполняет программу. Преобразование и выполнение программы осуществляется построчно.
- Python поддерживает динамическую типизацию — связывание переменной с типом в момент присваивания ей значения.
- В Python реализовано автоматическое управление памятью. Это означает, что пользователю нет необходимости вручную выделять и удалять память для динамических переменных.
- Python поддерживает интерактивный режим работы.

- Для Python написано большое количество свободно распространяемых библиотек, в том числе библиотек для научных расчётов.
- Синтаксис языка Python является легко читаемым и неперегруженным.
- Интерпретатор Python является свободно распространяемым программным обеспечением с открытым исходным кодом. Эталонной реализацией интерпретатора является CPython (URL: https://www.python.org).
- Кроссплатформенность: Python портирован на большинство существующих платформ, в том числе Windows, Linux и Mac OS X. Программный код, написанный на Python, может выполняться на любой платформе, на которой установлен интерпретатор Python.

1.2. Установка Python

В данном учебном пособии будут рассмотрены примеры программирования на Python на платформе Windows. Для установки всех необходимых инструментов для программирования удобно воспользоваться одним из наиболее развивающихся на данный момент дистрибутивов — Anaconda (URL: https://www.anaconda.com/download/). Он включает в себя интерпретатор языка Python, стандартные библиотеки Python, интегрированную среду разработки (IDE) Spyder с интерактивной оболочкой IPython, менеджер пакетов conda, а также набор библиотек для научных и инженерных расчётов: **NumPy** [1] (URL: https://numpy.org/), **SciPy** [2–4] (URL: https://www.scipy.org/), **Matplotlib** [6] (URL: https://matplotlib.org/) и др.

В данном учебном пособии рассмотрены примеры интерактивной работы в оболочке IPython (URL: http://ipython.org/) и примеры написания программ на языке Python. В обоих случаях для написания и запуска программ будет использована интегрированная среда разработки Spyder. При написании пособия использовался Python версии 3.6, IDE Spyder версии 3.0.

По умолчанию рабочее пространство IDE Spyder имеет три основных области (рис. 1):

• Текстовый редактор (отображается в левой части рабочего пространства). Предназначен для написания программ на языке Python. В редакторе реализованы подсветка синтаксиса, динамическая интроспекция кода (автодополнение вводимых команд, переход к определению объекта по щелчку мыши) и нахождение ошибок на лету.

• Окно справки (отображается справа сверху).

• Окно интерактивной оболочки IPython (отображается справа снизу), которое также называется «консоль». Консоль предназначена для работы в интерактивном режиме, то есть в режиме диалога, когда введённые пользователем команды сразу же выполняются и результат выводится в консоли на экран.

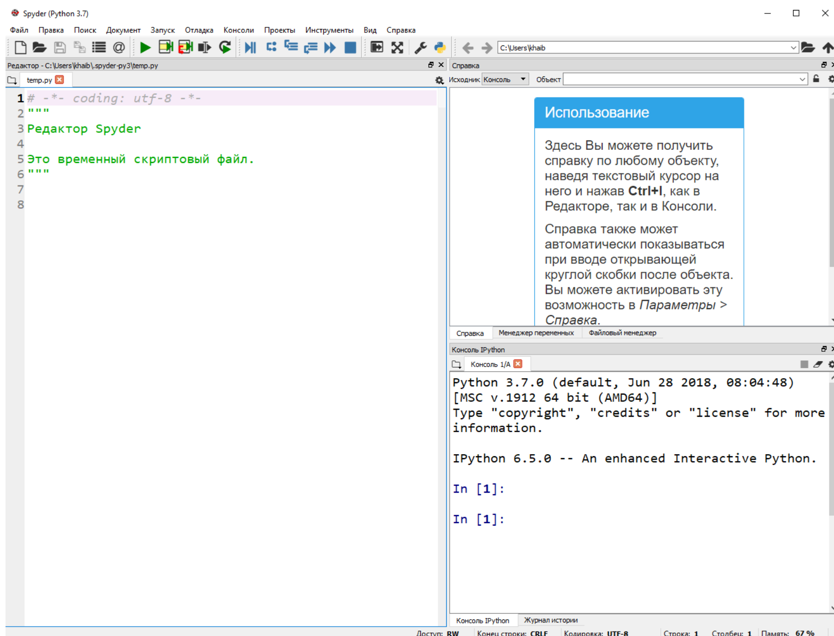

Рис. 1. *Внешний вид и основные элементы интегрированной среды разработки Spyder*

1.3. Интерактивный режим работы

В интерактивном режиме работы введённые пользователем с клавиатуры команды сразу же выполняются интерпретатором, а результат выводится на экран в окне консоли. Работа с интерпретатором осуществляется в окне оболочки IPython. Каждая строка ввода/вывода в IPython последовательно нумеруется. Например, запустим Spyder и введём в командной строке интерактивной оболочки IPython

```
In  [1]: 20 + 1
```

После нажатия клавиши Enter интерпретатор выполняет введённую пользователем команду и выводит результат её работы на экран:

```
Out [1]: 21
```

Порядковый номер, указываемый в квадратных скобках после слова Out, позволяет обращаться к ранее полученному результату. Например, чтобы вычесть из числа 3 результат, вычисленный в предыдущей команде, напишем

```
In  [2]: 3 - Out[1]
```

После нажатия клавиши Enter на экран будет выведен результат:

```
Out [2]: -18
```

Справку на английском языке по любому объекту можно получить, набрав в окне IPython имя объекта со знаком ? Например, справка по функции `type()`

```
In  [3]: type?
Docstring:
type(object) -> the object’s type
type(name, bases, dict) -> a new type
Type:      type
```

Для вывода на экран значения переменной в интерактивном режиме достаточно написать в консоли имя переменной и нажать

Enter, например:

```
In  [4]: Out[2]
Out [4]: -18
```

Клавиши стрелок «вверх» и «вниз» на клавиатуре позволяют просматривать ранее введённые команды и выполнять их заново.

в IPython реализовано автоматическое дополнение имён вводимых пользователем ключевых слов, имён модулей функций, переменных и имён файлов в текущей директории. Автодополнение имён делается с помощью клавиши TAB на клавиатуре. Например, если ввести в консоли символы ty и нажать клавишу TAB, интерпретатор автоматически дополнит вводимую пользователем конструкцию до имени **type**.

1.4. Типы данных

Типы данных в Python можно разделить на два класса: атомарные и структурные.

Атомарные типы данных: число с плавающей точкой (**float**), целое число (**int**), логический тип (**bool**), комплексное число (**complex**). При присваивании переменных атомарного типа копируется значение. Вещественные числа типа **float** в Python реализованы с двойной точностью, аналогично числам типа **double** в языке С. Переменные типа **bool** принимают всего два значения: **True** (истина) и **False** (ложь).

Структурные типы данных: списки (в частности, строки **str**), кортежи (англ., **tuple**), словари, классы, функции и т. д. При присваивании переменных структурных типов копируется указатель на объект.

Особенностью Python является ***динамическая типизация*** — переменная связывается с типом в момент присваивания значения, а не в момент объявления переменной. В таком подходе нет необходимости явно объявлять тип переменной до того, как ей будет присвоено какое-либо значение. Например, команда в консоли IPython

```
In  [1]: a = 20
```

создаёт переменную *a* и присваивает ей целое значение 20. В процессе дальнейшей работы в интерактивном режиме переменная *a* трактуется как целое число (число типа **int**). Информация о типе хранится вместе со значением и доступна во время исполнения. Например, тип переменной можно узнать с помощь команды **type()**

```
In  [2]: type(a)
Out [2]: int
```

Присвоим в интерактивном режиме *той же* переменной строковое значение:

```
In  [3]: a = "a string"
In  [4]: type(a)
Out [4]: str
```

Затем запишем в переменную *a* вещественное число:

```
In  [5]: a = 32.5
In  [6]: type(a)
Out [6]: float
```

или комплексное число:

```
In  [7]: a = 1.5 + 0.5j
In  [8]: type(a)
Out [8]: complex
```

Мнимая часть комплексного числа в Python указывается с помощью буквы **j**.

Язык Python использует принципы ***объектно-ориентированного программирования*** (ООП). В указанной парадигме программирования данные и методы для работы с ними объединяются внутри одной сущности — так называемого объекта. Класс в ООП — это тип, описывающий устройство и поведение объектов. Объект — это конкретный экземпляр класса.

Все сущности в Python являются объектами — и числа, и строки, и операции, и т. д. Обращение к свойствам (методам и полям) объектов осуществляется через знак точка (.) после имени соответствующего объекта.

Например, все переменные типа **complex** — которые также являются объектами — имеют поля **real** и **imag**, содержащие соответ-

ственно действительную и мнимую части комплексного числа. Для комплексных чисел реализован метод **conjugate()**, вычисляющий число, комплексно сопряжённое к данному. Для переменной *a*, введённой в команде 6 (см. выше):

```
In  [9]: a.real
Out [9]: 1.5
In  [10]: a.imag
Out [10]: 0.5
In  [11]: a.conjugate()
Out [11]: 1.5-0.5j
```

Рассмотрим некоторые структурные типы данных в Python.

Строки

Строки — это наборы символов. Для записи строк в Python используются как символы апострофа, так и кавычки:

‘label’, “label”

Это сделано для того, чтобы можно было использовать внутри строк символы кавычек или апострофов, например: `‘ letter “A” ’` или `“ letter ‘A’ ”`.

В интерпретаторе Python начиная с версии 3 все строки хранятся в кодировке Unicode. Это позволяет, например, использовать в строках символы кириллицы.

Списки

Список — упорядоченная последовательность элементов, пронумерованных с 0. Элементы списка перечисляются в квадратных скобках через запятую и могут иметь разные типы. Например, создадим в интерактивной оболочке IPython список с именем *my_list*, состоящий из трёх элементов, первый из которых целое число 1, второй — символ s, третий — число с плавающей точкой 3.14:

```
In  [1]: my_list = [1, ‘s’, 3.14]
```

Обращение к элементу списка осуществляется с помощью указания номера искомого элемента в квадратных скобках после имени списка:

```
In  [2]: my_list[0]
Out [2]: 1
In  [3]: my_list[1]
Out [3]: 's'
```

Функция **len**(<*список*>) из стандартной библиотеки Python возвращает длину (количество элементов) списка <*список*>:

```
In  [4]: len(my_list)
Out [4]: 3
```

Одной из наиболее распространённых операций со списками является их объединение — ***конкатенация***. В Python это осуществляется с помощью оператора сложения +. Например, создадим в интерактивном режиме списки *A* и *B*, а затем выполним их объединение:

```
In  [5]: A = [1, 2, 3]
In  [6]: B = [2, 3]
In  [7]: C = A + B
In  [8]: C
Out [8]: [1, 2, 3, 2, 3]
```

Приведём некоторые полезные операции для со списками. Все операции реализованы как методы класса «список».

- ***Добавление элемента*** в конец списка — метод **append**(<*v*>), где <*v*> — значение добавляемого элемента. Например, добавим в конец списка *my_list*, созданного в команде 1 (см. выше), целое число *2* и выведем список на экран:

```
In  [9]: my_list.append(2)
In  [10]: my_list
Out [10]: [1, 's', 3.14, 2]
```

- ***Удаление последнего элемента*** из списка — метод **pop**(). Значение удаляемого элемента выводится на экран. Например, удалим из списка *my_list* последний элемент и выведем список на экран:

```
In  [11]: my_list.pop()
Out [11]: 2
In  [12]: my_list
Out [12]: [1, ‘s’, 3.14]
```

Удаление элемента с индексом *index* осуществляется методом **pop**(*index*).

• ***Перестановка элементов*** в обратном порядке — метод **reverse()**. В применении к списку *my_list*, созданному выше:

```
In  [13]: my_list.reverse()
In  [14]: my_list
Out [14]: [3.14, ‘s’, 1]
```

Примечание. Строки являются неизменяемыми списками символов, поэтому к ним можно применять операции, реализованные для списков, но не изменяющие список. Например, операция взятия элемента по индексу (квадратные скобки) позволяет обращаться к конкретному символу из строки:

```
In  [15]: a = "a string"
In  [16]: s[0]
Out [16]: ‘a’
In  [17]: s[2]
Out [17]: ‘s’
```

Кортежи

Кортеж — это неизменяемый список. Кортежи объявляются так же, как списки, только с использованием круглых скобок. Например, объявим в окне интерактивной оболочки кортеж *my_tuple* из трёх элементов:

```
In  [1]: my_tuple = (1, ‘s’, 3.14)
```

Обращение к элементам кортежа осуществляется так же, как обращение к элементам списка: через указание индекса искомого элемента в квадратных скобках,

```
In  [2]: my_tuple[0]
Out [2]: 1
In  [3]: my_tuple[1]
Out [3]: 's'
```

Для кортежей определены все операции, указанные для списков, но не изменяющие список.

П р и м е ч а н и е. Удобство использования кортежей заключается в том, что их невозможно случайно изменить. Кроме того, кортежи занимают в оперативной памяти меньше места, чем списки той же длины.

Приведём пример создания кортежа с именем *my_smallest_tuple*, состоящего из одного элемента:

```
In  [4]: my_smallest_tuple = (1,)
```

Использование запятой после первого и единственного в рассматриваемом примере элемента обязательно:

```
In  [5]: my_smallest_tuple
Out [5]: (1,)
```

Словари

Словарь — это неупорядоченный набор объектов, записанных в виде пары «ключ: значение». Словари аналогичны базе данных, в которой чтение или запись значения некоторого объекта осуществляется не по его порядковому номеру в списке, а по уникальному идентификатору объекта — *ключу*. Ключи и значения в Python могут быть объектами любых типов. Начиная с интерпретатора Python версии 3.6 гарантируется, что элементы словаря перебираются в порядке их добавления.

Словари в Python записываются с использованием фигурных скобок, внутри которых через запятую перечисляются пары «ключ: значение». Например, создадим в интерактивном режиме словарь *student_dict*, состоящий из двух записей об имени и возрасте студента:

```
In  [1]: student_dict = {'name': 'Tom', 'age': 20.0}
```

В данном словаре строки *'name'* (имя) и *'age'* (возраст) являются ключами, а строка *'Tom'* и число *20.0* — соответствующими значениями. Конструкция вида

```
d = dict.fromkeys([list], default_value)
```

где *[list]* — некоторый список, *default_value* — некоторое значение, создаёт словарь *d*, в котором значения ключей установлены в соответствии со значениям элементов списка *[list]*, а в качестве значения всех записей по умолчанию используется *default_value*.

Доступ к значениям конкретной записи в словаре осуществляется либо с помощью метода **get**(<*имя ключа*>), либо с помощью квадратных скобок после имени словаря с указанием имени ключа:

```
In  [2]: student_dict['name']
Out [2]: 'Tom'
In  [3]: student_dict['age']
Out [3]: 20.0
In  [4]: student_dict.get('age')
Out [4]: 20.0
```

Аналогично осуществляется запись в словарь:

```
In  [5]: student_dict['age'] = 18.0
In  [6]: student_dict
Out [6]: {'name': Tom, 'age': 18.0}
```

Если запись с указанным ключом в словаре уже существует, то соответствующее значение переписывается. Добавление записи в словарь можно также осуществлять с помощью метода **update()**, в котором в качестве аргумента указывается добавляемая запись. Например, добавим в словарь *student_dict* запись о том, закончил ли студент обучение:

```
In  [7]: student_dict.update{'graduated': False}
In  [8]: student_dict
Out [8]: {'name': Tom, 'age': 18.0, 'graduated': False}
```

Удаление записи из словаря осуществляется с помощью метода **pop()**. При вызове метод **pop()** возвращает удаляемое значение. Например, удалим запись о возрасте студента из словаря *student_dict*:

```
In  [9]: student_dict.pop('age')
Out [9]: 18.0
In  [10]: student_dict
Out [10]: {'name': Tom, 'graduated': False}
```

Методы **keys()** и **values()** отображают соответственно списки ключей и значений из словаря.

1.5. Операции и логические операторы

Рассмотрим некоторые наиболее часто используемые в Python операторы.

• ***Операция присваивания*** переменной значения делается с помощью знака `=`.

• *Арифметические операции* над числами: `+` (сложение), `–` (вычитание), `/` (деление), `*` (умножение).

Примечание 1. В применении к спискам оператор `+` выполняет их конкатенацию (см. параграф 1.4).

Примечание 2. В предыдущей версии Python (2.7) в применении к целым числам оператор `/` выполнял операцию целочисленного деления. В Python 3.6, оператор `/` выполняет обычное деление как в применении к числам с плавающей точкой, так и к целым числам. Выполнение целочисленного деления выполняется оператором `//`, нахождение остатка от деления — оператором `%`.

• ***Команда возведения в степень***: `**` (двойная звёздочка).

• ***Команды преобразования типов***:

int(*a*) — преобразование аргумента *a* к целому числу;

float(*a*) — преобразование аргумента *a* к числу с плавающей точкой;

str(*a*) — преобразование аргумента *a* к строке.

Оператор проверки равенства: `==`. Результатом действия операции является **True**, если объекты совпадают, и **False** — в противном случае. Например, проверим в окне интерактивной оболочки, равны ли друг другу числа 1 и 1:

```
In  [1]: 1==1
Out [1]: True
```

• ***Оператор* in** — проверка, содержится ли объект в списке, кортеже или словаре. Например, создадим список *my_list* и проверим, входит ли в него символ *‘s’*:

```
In  [2]: my_list = [1, ‘s’, 3.14]
In  [3]: ‘s’ in my_list
Out [3]: True
```

Логические операторы переменных типа **bool: and** (логическое И), **or** (логическое ИЛИ).

1.6. Ввод/вывод

Для вывода значений переменных на экран используется встроенная функция **print()**. Если передать в функцию **print()** несколько аргументов, разделённых запятыми, то она выведет на экран в строке последовательно значения всех аргументов, разделённые пробелами. Например, создадим переменные *r* и *m* и выведем на экран их значения:

```
In  [1]: r=1.2
In  [2]: m=10
In  [3]: print("radius =", r, "[cm], mass =", m, "kg")
radius = 1.2 [cm], mass = 10 [kg]
```

Для ввода данных с клавиатуры используется функция **input()** из стандартной библиотеки Python. После срабатывания команды **input()** поток выполнения программы останавливается в ожидании данных, которые пользователь должен ввести с помощью клавиатуры. После ввода данных и нажатия Enter, функция **input()** завершает своё выполнение и возвращает результат, который представляет собой строку символов, введённых пользователем:

```
In  [4]: x = input()
10
```

```
In  [5]: x
Out [5]: '10'
```

Функция **input()** может принимать в качестве аргумента строку, которая будет выведена на экран прежде, чем пользователь начнёт вводить данные с клавиатуры:

```
In  [4]: x = input("Hello! Enter a number. ")
Hello! Enter a number. 10
In  [5]: x
Out [5]: '10'
```

Форматированный ввод-вывод данных в Python осуществляется в стиле языка программирования С. Оператор `%` используется для форматирования набора переменных, заключённых в кортеж, вместе со строкой форматирования, которая содержит обычный текст вместе со спецификаторами формата — специальными символами, такими как **%s** и др., которые указывают тип и правила форматирования соответствующих переменных. Основные спецификаторы формата:

%s — строка;

%d — целое число;

%f — число с плавающей точкой (по умолчанию отображается с шестью значащими цифрами после запятой);

%.<N>f — число с плавающей точкой с фиксированным числом N значащих цифр после запятой;

%e — число с плавающей точкой, записанное в научном формате (пр., число записывается как 6.673e-8; по умолчанию отображается с шестью значащими цифрами после запятой).

Например:

```
In  [6]: name = "Tom"
In  [7]: age = 20
In  [8]: print("My name is %s, and I am %d years old" % (name, age))
My name is Tom, and I am 20 years old
In  [9]: pi = 3.1415926535
In  [10]: print("I know that the Pi number equals %f" % pi)
I know that the Pi number equals 3.141593
In  [11]: print("More precisely, Pi number equals %.8f" % pi)
```

```
More precisely, Pi number equals 3.14159265
In    [12]: G = 6.67259e-8
In  [13]: print("The gravitational constant equals %.3e" % G)
The gravitational constant equals 6.673e-08
```

1.7. Управляемые блоки кода в Python

Рассмотрим реализацию управляющих конструкций для блоков кода в Python.

Важной особенностью Python является то, что блоки кода выделяются с помощью *отступов*, а не фигурными скобками, как в Си, или операторами begin-end как в Pascal.

Вход в управляемый блок кода указывается с помощью знака двоеточие (:). После этого каждая строка кода, которая входит в управляемый блок, должна быть выделена с помощью отступа. IPython автоматически создаёт отступ после того, как пользователь напечатает знак двоеточие (:) и нажмёт клавишу Enter. Чтобы убрать отступ, воспользуйтесь клавишей Backspace. Для выхода из управляемого блока необходимо дважды нажать клавишу Enter. Аналогично осуществляется работа с отступами в текстовом редакторе, встроенном в IDE Spyder.

Условный оператор if

Данный оператор сообщает интерпретатору, что некоторый блок кода необходимо выполнять только при определённом условии. Синтаксис конструкции:

```
if <условие 1>:
    <блок операторов 1>
elif <условие 2>:
    <блок операторов 2>
…
elif <условие i>:
    <блок операторов i>
```

```
…
elif <условие n-1>:
    <блок операторов n-1>
…
else:
    <блок операторов n>
```

где *<условие 1>, <условие 2>* и т. д. — любые логические выражения. Зарезервированное слово **elif** является сокращением от **else if** и позволяет проверять дополнительное *<условие i>*, если не выполнено *<условие i-1>*, и выполнять соответствующий *<блок операторов i>* (здесь *i* — номер блока). Количество блоков кода, управляемых **elif**, не ограничено. Каждая строка в каждом блоке операторов, следующем после **if**, или **elif**, или **else**, должна быть выделена с помощью отступа. Если не выполнено ни одно из условий вплоть до условия *<условие n-1>*, то выполняется блок кода *<блок операторов n>,* указанный после команды **else** (здесь *n* — полное число блоков операторов). В конце каждой из строк, начинающихся со слов **if**, **elif**, **else**, должно стоять двоеточие.

Например, ниже в окне интерактивной оболочки IPython переменной *x* присваивается целочисленное значение 1, а затем создаётся конструкция, проверяющая, является ли число *x* положительным, нулевым или отрицательным, и выводящая соответствующее сообщение на экран:

```
In  [1]: x = 1
In  [2]: if x < 0:
      …:    print ("x is negative")
      …: elif x == 0:
      …:            print ("x is zero")
      …: else:
      …:    print ("x is positive")
x is positive
```

Оператор цикла for

Оператор цикла позволяет выполнять блок некоторых операций заданное число раз. Чаще всего используется перебор значений переменной цикла из заранее заданного списка значений. Синтаксис конструкции:

```
for <переменная цикла> in [список значений]:
    <блок операторов>
```

где *<переменная цикла>* — переменная, хранящая текущее значение из списка *[список значений]*, *<блок операторов>* — блок операторов, выполняемый столько раз, сколько элементов в списке *[список значений]*. Каждая строка в блоке кода внутри конструкции **for** должна быть выделена с помощью отступа. Например, в примере ниже с помощью цикла **for** перебираются и выводятся на экран значения элементов списка *my_list*:

```
In  [1]: my_list = [0, 1, 2, 3]
In  [2]: for i in my_list:
      ...:    print (“i = ”, i)
      ...:
i = 0
i = 1
i = 2
i = 3
```

Для перебора значений целочисленного индекса в диапазоне от *0* до *n-1* удобно использовать функцию **range**(*n*) из стандартной библиотеки Python. Эта функция создаёт список из *n* элементов множества целых чисел начиная с нуля. Например, создадим с помощью команды **range**() цикл, последовательно выводящий на экран значения чисел 0, 1, 2, 3:

```
In  [3]: for i in range(4):
       ...:   print ("i = ", i)
      ...:
i = 0
i = 1
i = 2
i = 3
```

Перебор значений переменной цикла можно осуществлять с помощью любой последовательности (строки, списка, словаря, строк в файле и т. п.). Например, в качестве множества значений переменной цикла можно использовать строку:

```
In  [4]: text = "today is a good day"
In  [5]: words_list = text.split()
In  [6]: words_list
Out [6]: ["today", "is", "a", "good", "day"]
In  [7]: for word in word_list:
      …:     print(word)
      …:
today
is
a
good
day
```

В данном примере использован метод **split**() объектов строкового типа. Данный метод разбивает строку на части, используя разделитель, и возвращает эти части списком. По умолчанию считается, что разделителем является пробельный символ.

Оператор цикла while

Данная управляющая конструкция позволяет выполнять некоторый блок кода до тех пор, пока является истинным некоторое заданное условие. Конструкция вида

```
while <условие>:
    <блок операторов>
```

выполняет *<блок операторов>* до тех пор, пока истинно заданное *<условие>*. Каждая строка в блоке кода внутри конструкции **while** должна быть выделена с помощью отступа. В следующем примере значения переменной x последовательно уменьшаются на 1 и выводятся на экран до тех пор, пока x остаётся положительным числом:

```
In  [1]: x=2
In  [2]: while x > 0:
     …:     print ("x=", x)
     …:     x = x - 1

x=2
x=1
```

1.8. Функции

Функции в программировании нужны для повторного использования некоторого блока кода. Синтаксис объявления функции в Python:

```
def <имя функции>(<список формальных аргументов>):
    <блок операторов>
    return <результат>
```

Объявление функции начинается с зарезервированного слова **def**. После него указывается имя функции *<имя функции>*. Формальные аргументы функции перечисляются в круглых скобках после её имени. После скобок ставится знак двоеточия. Тело функции может состоять из произвольного числа строк кода, каждая из которых выделяется с помощью отступа. В конце функции может располагаться оператор **return** с последующим указанием значения *<результат>*, которое будет возвращать функция при её вызове. Если в теле функции отсутствует строка, начинающаяся с **return**, то при вызове такая функция возвращает значение **None** (нулевой указатель).

Строка документации, описывающая то, как работает функция, оформляется с помощью тройных кавычек.

Вызов функции осуществляется указанием её имени со списком фактических аргументов в круглых скобках.

Например, определим функцию **my_pow(***arg1, arg2***)**, принимающую два аргумента, *arg1* и *arg2*, и вычисляющую сумму квадратов этих аргументов:

```
In  [1]: def my_pow(arg1, arg2):
 ...:            """ сумма квадратов аргументов """
 ...:            tmp = arg1**2 + arg2**2
 ...:            return tmp
 ...:
In  [2]: my_pow(2, 2)
Out [2]: 8
In  [3]: my_pow?
Signature: my_pow(arg1, arg2)
Docstring: calculates arg1^arg2
File:      c:\path\<ipython-input-36-5b9e115c0c78>
Type:      function
```

Примечания

1. Все переменные, объявленные внутри функции, являются локальными.

2. Тип формальных аргументов функции определяется исходя из типа фактических аргументов. При этом команды для работы с формальными аргументами в теле функции не должны противоречить подразумеваемому типу предполагаемых фактических аргументов.

Например, создадим функцию

```
In  [4]: def f1(x):
...:             return x
```

Подразумевается, что функция возвращает тот же объект, что принимает в качестве аргумента. В таком случае в качестве фактического аргумента в функцию **f1** можно передавать объект любого типа, будь то число, строка, список, массив и т. п. Например, для определённой выше функции

```
In  [5]: f1(1)
Out [5]: 1
In  [6]: f1('a')
Out [6]: 'a'
In  [7]: f1([0, 1, 'b'])
Out [7]: [0, 1, 'b']
```

Рассмотрим другой пример:

```
In  [8]: def f2(x):
...:             result1 = x[0]
...:             result2 = x[1]
...:             return [result1, result2]
```

Реализация функции **f2** подразумевает, что функция принимает в качестве аргумента объекты, к которым можно применять оператор **[***index***]** (получение элемента по индексу *index*). Такими объектами могут быть либо список, либо массив. Рассмотренное объявление функции запрещает передачу функции в качестве аргумента чисел типа **int**, **float**, **complex**, поскольку к переменным указанных типов невозможно применить операцию **[]**, использованную в реализации функции. В таком случае интерпретатор сообщит об ошибке:

```
In  [9]: f2(1)
Traceback (most recent call last):
File "<ipython-input-12-e94ca8346097>", line 1, in <module>
    f3(1)

File "<ipython-input-8-ff221084ea08>", line 2, in f3
    return [x[0], x[1]]
TypeError: 'int' object has no attribute '__getitem__'
```

В рассматриваемом примере продемонстрирована ещё одна удобная возможность функций в Python — они могут возвращать не одно значение, а целый список значений. В примере функция **f2** возвращает не одно значение, а список из двух значений, *x[0]* и *x[1]* (которые для наглядности обозначены внутри функции как *result1* и *result2*). Пример использования функции **f2** после того, как она была объявлена в окне интерактивной оболочки:

```
In  [10]: f2([1, 'b'])
Out [10]: [1, 'b']
```

Кроме того, в функцию **f2** можно передать список из более чем двух элементов. В таком случае использованы будут только первые два элемента списка.

1.9. Работа с библиотеками

Существует несколько альтернативных способов подключения внешних библиотек (модулей) в Python. Рассмотрим эти способы на примере математической библиотеки **math**. Данная библиотека содержит математические функции и константы, такие как: **fabs**(*x*) — модуль числа *x*; **exp**(*x*) — экспонента в степени *x*; **log**(*x*, *n*) — логарифм числа *x* по основанию *n*; **pow**(*x*, *y*) — возведение числа *x* в степень *y*; **sqrt**(*x*) — квадратный корень из числа *x*; **cos**(*x*), **sin**(*x*), **tan**(*x*) — косинус, синус и тангенс числа *x* в радианах соответственно; **pi** — число Пи; **e** — экспонента, и др. Описание полного списка функций библиотеки можно получить с помощью команды **help(«math»)** в консоли.

1. Подключение библиотеки осуществляется с помощью команды **import**. Например, подключение стандартной библиотеки математических функций **math**:

```
In  [1]: import math
```

В этом случае все функции из библиотеки **math** загружаются в текущее пространство имён интерпретатора так, что пользователь может получить доступ к каждой функции через префикс **math**. Например, применим функцию вычисления квадратного корня **sqrt()** из библиотеки **math**:

```
In  [2]: a = math.sqrt(4)
In  [3]: a
Out [3]: 2.0
```

2. Загрузка всех функций из некоторой библиотеки в текущее пространство имён интерпретатора делается с помощью знака звёздочка:

```
In  [4]: from math import *
```

В этом случае все функции из библиотеки **math** доступны без префикса. Например, найдём e^2 и $sin\ \pi$:

```
In  [5]: b = exp(2)
In  [6]: b
Out [6]: 7.3890560989306504
```

```
In  [7]: sin(pi)
Out [7]: 0
```

3. Загрузка в текущее пространство имён конкретных функций из заданной библиотеки:

```
In  [8]: from math import log10, fact
```

В рассмотренном примере, из библиотеки **math** загружаются только функции **log10** (десятичный логарифм) и **fact** (факториал). Загруженные функции можно использовать без префикса **math**. Например:

```
In  [9]: c = log10(10)
In  [10]: c
Out [10]: 1.0
In  [11]: factorial(5)
Out [11]: 120
```

4. Подключение библиотеки с псевдонимом.

Использование второго способа подключения библиотек кажется наиболее удобным, поскольку избавляет от необходимости писать префиксы функций. Однако при подключении таким способом нескольких библиотек может возникнуть конфликт имён, если в разных библиотеках имеются функции с одинаковым именем. Чтобы избежать конфликтов имён и сократить время написания префиксов в Python, реализована возможность подключения библиотеки с псевдонимом. Общий синтаксис конструкции

```
import <library> as <alias>
```

где *<library>* — название подключаемой библиотеки, *<alias>* — псевдоним, под которым предполагается использовать библиотеку в программе. В примере ниже библиотека **math** подключается с псевдонимом **m**:

```
In  [12]: import math as m
```

Имя-псевдоним выбирается пользователем самостоятельно. В дальнейшем обращение к функциям библиотеки **math** осуществляется с помощью префикса-псевдонима, а не полного имени библиотеки:

```
In  [13]: a = m.sqrt(4)
In  [14]: a
Out [14]: 2.0
In  [15]: b = m.exp(2)
In  [16]: b
Out [16]: 7.3890560989306504
In  [17]: m.sin(m.pi)
Out [17]: 0
```

1.10. Написание файлов исходного кода

Работа с Python в интерактивной оболочке удобна, так как позволяет быстро выполнить какую-либо операцию, протестировать небольшой блок кода и пр. В случае когда код требуется сохранить для дальнейшего использования, подготавливаются файлы исходного кода, которые затем передаются интерпретатору на исполнение.

Файлы исходного кода на языке Python имеют расширение **.py**. Такие файлы называют скриптами. ***Скрипт*** — файл, содержащий последовательность инструкций, повторяющуюся каждый раз, когда файл вызывается в интерпретаторе.

Нет никаких специальных требований к оформлению файлов исходного кода на Python. Достаточно перечислить в скрипте строчка за строчкой команды на синтаксисе Python.

Комментарий в текстовом редакторе начинается со знака решётки:

```
# комментарий
```

При создании нового файла текстовый редактор в IDE Spyder автоматически добавляет шапку-документацию с краткой информацией о файле. Шапка выделяется с помощью тройных кавычек, которые позволяют записывать строковые переменные в несколько строчек кода. В шапке указываются используемая кодировка и общая информация о файле:

```
# -*- coding: utf-8 -*-
"""
Created on Mon Oct  1 20:00:00 2018

@author: Sergey A. Khaibrakhmanov
"""
```

При необходимости изменить кодировку файла соответствующий комментарий (как в примере выше) обязательно пишется в самой первой строке файла исходного кода.

Создадим в папке **“Z:\”** файл **hello.py**, содержащий следующий код:

```
# -*- coding: utf-8 -*-
"""
Created on Mon Oct  1 20:00:00 2018

@author: Sergey A. Khaibrakhmanov
"""
# подключение математической библиотеки
import math as m
def distance(x, y):
    """
    Функция возвращает сумму квадратов двух
    аргументов, которые она принимает. Внутри функции
    выполняется преобразование аргументов к типу
    float.
    """
    return m.sqrt(float(x)**2 + float(y)**2)


# приветствие
print("Hello!")


# считывание значения координаты x с клавиатуры в
# переменную - строку str_x
str_x = input("Enter x-coordinate. ")
```

```
# считывание значения координаты y с клавиатуры в
# переменную - строку str_y
str_y = input("Enter y-coordinate. ")


# вывод значения расстояния до точки (x, y) на экран
print("The distance is", distance(str_x, str_y))
```

Запуск файлов в окне интерактивной оболочки IPython осуществляется с помощью команды **run** с последующим указанием в скобках имени скрипта. В первую очередь в интерактивной оболочке нужно перейти в папку, где находится файл программы, с помощью команды **cd** (change directory):

```
In  [1]: cd "Z:\"
Out [1]: ‘Z:\\’
In  [2]: run hello.py
Hello!

Enter x-coordinate. 3

Enter y-coordinate. 4
The distance is 5.0
```

Другой способ — запустить скрипт командой **runfile**, указав в качестве аргумента строку-путь к файлу. Например, если скрипт **hello.py** расположен в корне диска **Z**, то нужно написать

```
In  [3]: runfile(‘Z:/hello.py’)
```

Если вы работаете в среде разработки Spyder, то текущее окно интерактивной оболочки IPython, как правило, связано с текстовым редактором. Запуск программы в текущей консоли осуществляется нажатием клавиши F5.

Если файлы исходного кода Python ассоциированы в проводнике Windows с установленным интерпретатором Python, то запуск скриптов из проводника осуществляется двойным щелчком.

Для запуска скрипта Python из командной строки Windows или консоли Linux необходимо перейти в командной строке (консоли) в папку, где хранится скрипт, напечатать имя скрипта и нажать клавишу Enter.

1.11. Создание и использование собственных модулей

Пользовательские файлы исходного кода Python могут быть использованы как модули, подключаемые к другой программе на Python. Загрузка объектов (данных и функций) из внешнего модуля осуществляется так же, как в случае любых сторонних библиотек и модулей Python: с помощью команд **import**, описанных в параграфе 1.9.

Например, создадим файл-модуль **convert_distance.py**, в котором определены значения астрономической единицы (1 а.е. = $1{,}5 \cdot 10^{13}$ см) и парсека (1 пк = $3{,}086 \cdot 10^{18}$ см), а также функции для преобразования расстояния из сантиметров в а.е. и пк:

```
# -*- coding: utf-8 -*-
"""
Created on Mon Oct 1 20:00:00 2018
Модуль содержит константы и функции для преобразования рассто-
яний к единицам измерения, принятым в астрофизике
@author: С.А. Хайбрахманов
"""
# астрономическая единица в см
au = 1.5e13
# парсек в см
pc = 3.086e18

def cm2au(dist_cm):
    """
    Функция преобразования расстояния, измеренного в см,
    в астрономические единицы
    """
    return dist_cm / au

def cm2pc(dist_cm):
    """
    Функция преобразования расстояния, измеренного в см,
    в парсеки
    """
    return dist_cm / pc
```

Допустим, модуль **convert_distances.py** сохранён в папке **G:/python_modules/**. Ниже приведён листинг программы **test_distance_convert.py**, в которой подключается и используется модуль **convert_distances.py**. В программе сначала подключается модуль **sys**, содержащий определение списка *path* — списка имён папок, в которых интерпретатор Python выполняет поиск подключаемых модулей. В список **path** с помощью метода **append()** добавляется строка с именем папки, в которой сохранён модуль **convert_distances.py**. Затем программа предлагает пользователю ввести расстояние в см и последовательно использует определённые в модуле функции и переменные:

```
# -*- coding: utf-8 -*-
"""
Created on Fri Oct 11 14:16:51 2019

Тестирование модуля преобразования расстояний
@author: С.А. Хайбрахманов
"""
# подключение модуля sys для взаимодействия
# с интерпретатором python
import sys
# добавление в список path из модуля sys
# строки с именем папки (G:/), в которой
# расположен пользовательский модуль
sys.path.append("G:/")

# подключение пользовательского модуля
# с псевдонимом convert
import convert_distance as convert


print("Привет!")
dist = float(input("Введите расстояние в см:\n r = "))
print("Результат преобразования:")
print(" r = %.2e а.е." % convert.cm2au(dist))
print(" r = %.2e пк" % convert.cm2pc(dist))
print(" где 1 а.е. = %.2e см" % convert.au)
print(" где 1 пк   = %.2e см" % convert.pc)
```

Примеры решения задач

Задача 1. Создайте в окне интерактивной оболочки IPython функцию, выполняющую деление двух чисел.

Решение. Создадим функцию **my_division**, принимающую два аргумента, *arg1* и *arg2* (см. вставку кода ниже). С помощью управляющей конструкции **if-else** реализуем проверку того, не является ли второй аргумент нулём. Если нет, то функция возвращает результат деления *arg1/arg2*, если да — выводит сообщение об ошибке «division by zero!».

```
In  [1]: def my_division(arg1, arg2):
...:              if arg2 != 0:
...:                 return arg1 / arg2
...:              else:
...:                 print ("division by zero!")
...:
```

Проверим работоспособность функции:

```
In  [2]: my_division(2, 2)
Out [2]: 1.0
In  [3]: my_division(3.0, 0)
division by zero!
```

Задача 2. Напишите программу, которая предлагает пользователю ввести два числа a_1 и a_2, а затем вычисляет величину

$$\frac{1}{(a_1^2 + a_2^2)}.$$

Решение. Далее приведён листинг программы. Предложение пользователю ввести два числа и их считывание реализовано с помощью функции **input()** из стандартной библиотеки (см. параграф 1.6). Вычисление квадратного корня осуществляется с помощью функции **sqrt()** из библиотеки математических функций **math**, которая подключается с псевдонимом **m** (см. параграф 1.9). Операция вычисления требуемой величины оформлена в виде функции **f**(*x,y*). Внутри функции **f** выполняется преобразование аргументов *x* и *y* к вещественному типу **float**.

```
# -*- coding: utf-8 -*-
"""
Задача 2
"""
import math as m


def f(x, y):
    return 1 / m.sqrt(float(x)**2 + float(y)**2)


print("Hello!")
str_a1 = input("Enter a1: ")
str_a2 = input("Enter a2: ")
print("The result is", f(str_a1, str_a2))
```

Задача 3. Напишите программу «телефонный справочник». Программа предлагает пользователю ввести имя абонента, затем проверяет, есть ли абонент в заранее подготовленном справочнике. Если имя абонента найдено, то программа выводит на экран сообщение с соответствующим номером телефона, если нет — сообщает пользователю, что данный абонент не найден. Справочник имён и телефонов в программе должен быть реализован с помощью типа ‘список’.

Решение. Далее приведён листинг программы. В программе создаются списки *phones_list* и *names_list*, содержащие соответственно номера телефонов и имена, в строковом формате. Программа выводит на экран приветствие, затем считывает и записывает в переменную *str_name* введённую пользователем строку. Если введённая строка содержится в списке *names_list*, то с помощью метода **index(***str_name***)** вычисляется индекс соответствующего элемента в списке *names_list*, и затем на экран выводится строка с соответствующим номером телефона. Если введённого пользователем имени нет в списке *names_list*, то программа сообщает, что пользователь не найден, и завершает работу.

```
# -*- coding: utf-8 -*-
"""
Задача 3
"""
```

```
phones_list = ['7-351-00-00', '7-351-00-01', '7-351-00-02', '7-351-
00-03']
names_list = ['Ivan', 'Kate', 'Sergey', 'Fox']


print("Hello!")
str_name = input("name: \n")


if str_name in names_list:
   index = names_list.index(str_name)
   print(str_name, "phone is", phones_list[index])
else:
   print ("user not found")
```

Задача 4. Модифицируйте программу из задачи 2 так, чтобы она завершала свою работу по желанию пользователя.

Решение. Далее приведён листинг программы. Основной цикл выполнения программы реализован с помощью управляющей конструкции **while**. Программа выводит на экран приветствие, просит пользователя ввести два числа, выводит результат выполнения требуемой операции на экран до тех пор, пока значение логической переменной **work** истинно. Внутри основного цикла с помощью конструкции **while** реализован ещё один цикл, который выполняется до тех пор, пока значение логической переменной *ask_to_proceed* истинно. Внутри данного цикла программа выводит запрос о необходимости продолжить работу. Если пользователь вводит ответ *y* или *yes*, то логической переменной *work* присваивается значение **True**. Если пользователь ввёл *n* или *no*, то *work* присваивается значение **False**. В обоих случаях значение переменной *ask_to_proceed* устанавливается в **False** для того, чтобы выйти из данного цикла. Если ответ пользователя не содержит *y*, *yes*, *n* или *no*, то исходное значение (**True**) логической переменной *ask_to_proceed* не меняется и внутренний цикл повторяется.

```
# -*- coding: utf-8 -*-
"""
Задача 4
"""
```

```
import math as m

def f(x, y):
    return 1 / m.sqrt(float(x)**2 + float(y)**2)

work = True
while work:
   print("Hello!")
   str_a1 = input("Enter a1: \n")
   str_a2 = input("Enter a2: \n")
   print("The result is", f(str_a1, str_a2))

   ask_to_proceed = True
   while ask_to_proceed:
      str_ans = input("continue work? \n")
      if (str_ans in ["y", "n", "yes", "no"]):
         if (str_ans in ["y", "yes"]):
            work = True
         else:
            work = False
         ask_to_proceed = False
      else:
         print ("I didn’t understand your answer.
               Please, enter y, n, yes, or no.")
```

Задания для самостоятельной работы

1. Модифицируйте программу задачи 3 так, чтобы записи в телефонном справочнике были реализованы с помощью словарей, а не списков. Код, реализованный с помощью словарей, является более быстрым, чем код, написанный с использованием списков.

2. Напишите программу, реализующую возможности простого калькулятора. Программа ожидает ввода двух чисел с клавиатуры, затем предлагает выбрать требуемую арифметическую опера-

цию (+, –, /, ×), производит вычисления и выводит результат. Если результат вычисления равен числу больше 100 или меньше 0.01, то он должен выводиться на экран в научном формате с тремя значащими цифрами после запятой. Выход из программы осуществляется по желанию пользователя.

Вопросы для самоконтроля

1. Что такое интерпретатор?
2. В чём заключается интерактивный режим работы с интерпретатором?
3. Как в интерактивном режиме вывести на экран значение переменной?
4. Что означает динамическая типизация? Как определить тип переменной?
5. В чём отличие списка от кортежа?
6. Как в управляющих конструкциях Python выделяются блоки кода?
7. Что такое скрипт?
8. Как вывести на экран значение переменной в скрипте?
9. В чём отличие одинарных и двойных кавычек при оформлении строк в Python?
10. Как называется библиотека для работы с математическими функциями и константами? Воспользуйтесь функцией **help()** и выясните названия функций для вычисления обратных синуса, косинуса и тангенса, а также гиперболических функций.

Рекомендуемая литература

1. Фёдоров, Д. Ю. Программирование на языке высокого уровня python : учеб. пособие для сред. профес. образования / Д. Ю. Фёдоров. — 2-е изд. — М. : Юрайт, 2019. — 161 с. (Профессиональное образование). — URL: https://www.biblio-online.ru/bcode/446505 (дата обращения: 15.10.2019).

2. Буйначев, С. К. Основы программирования на языке Python / С. К. Буйначев, Н. Ю. Боклаг ; Мин-во образования и науки Рос. Федерации, Урал. федер. ун-т им. первого Президента России Б. Н. Ельцина. — Екатеринбург : Изд-во Урал. ун-та, 2014. — 92 с.— URL: http://biblioclub.ru/index.php?page=book&id=275962 (дата обращения: 15.10.2019).

3. Шелудько, В. М. Основы программирования на языке высокого уровня Python / В. М. Шелудько ; Мин-во науки и высш. образования РФ, Юж. федер. ун-т, Инженер.-технолог. акад. — Ростов н/Д. ; Таганрог : Изд-во Юж. федер. ун-та, 2017. — 147 с. — URL: http://biblioclub.ru/index.php?page=book&id=500056 (дата обращения: 15.10.2019).

4. Читлур, С. Укус Питона [Электронный ресурс] / С. Читлур ; пер. В. Смоляр. — 2013–2019. — URL: https://wombat.org.ua/AByteOfPython/frontpage.html (дата обращения: 16.10.2019)

Глава 2

Библиотека **NumPy** для работы с многомерными массивами

В данной главе описываются базовые возможности библиотеки **NumPy** для работы с многомерными массивами. Разбираются примеры создания массивов, доступа к элементам массивов, быстрых операций над массивами. В завершение приводится самостоятельное задание по работе с одномерными и двумерными массивами, а также список контрольных вопросов.

2.1. Общие сведения

Библиотека **NumPy** представляет собой набор инструментов для научных расчётов. Библиотека содержит описание специального типа данных **ndarray**, реализующего возможности многомерных массивов (массивов произвольной размерности). Кроме того, библиотека включает в себя функции для работы с многомерными массивами типа **ndarray**, а также имеет возможности для решения задач линейной алгебры, использования преобразования Фурье и работы со случайными числами.

Массив **ndarray** может содержать последовательности данных одного типа, таких, например, как:

- сетка экспериментально измеренных величин (пр., показания термометра в разные моменты времени);
- сетка расчётных величин (пр., координаты материальной точки в заданные моменты времени, полученные с помощью численного решения уравнения движения);
- пиксели изображения;
- прочие.

Общепринятый способ подключения библиотеки **NumPy**:

```
In  [1]: import numpy as np
```

2.2. Создание массивов NumPy

Существует несколько способов создания массивов с помощью библиотеки **NumPy**.

Ручное создание на основе списка Python

Функция **array()** из библиотеки **NumPy** принимает в качестве аргумента список и возвращает массив типа **ndarray**, содержащий те же значения, что и указанный список.

Например, создадим одномерный массив чисел от 0 до 4 из заранее определённого списка *my_list*:

```
In  [1]: my_list = [0, 1, 2, 3, 4]
In  [2]: a1 = np.array(my_list)
In  [3]: a1
Out [3]: array([ 0, 1, 2, 3, 4])
In  [4]: type(a1)
Out [4]: <type ‘numpy.ndarray’>
```

Создадим двумерный массив (матрицу) чисел с плавающей точкой. Матрицу можно представить как список, элементами которого являются списки чисел — строк матрицы. Чтобы указать интерпретатору тип элементов создаваемого массива, добавим после запятой в списке аргументов функции **array()** аргумент **dtype** со значением **float**,

```
In  [5]: a2 = np.array([[1, 2, 3], [4, 5, 6]], dtype=float)
In  [6]: a2
Out [6]:
array([[ 1., 2., 3.],
            [4., 5., 6.]])
```

*Создание массива с помощью встроенных функций из библиотеки **NumPy***

Функция **numpy.arange(***start, stop, step***)** создаёт массив чисел, распределённых в диапазоне [*start, stop*) с шагом *step*. Если аргументы *start* и *step* не указаны, то **numpy.arange()** создаёт массив чисел в диапазоне от *0* до *stop-1* с шагом 1. Например, массив из 5 чисел от 0 до 4:

```
In  [1]: a = np.arange(5, dtype=float)
Out [1]: array([ 0., 1., 2., 3., 4.])
```

В данном примере в функцию **numpy.arange()** передан дополнительный аргумент **dtype** со значением **float**, указывающий тип чисел в создаваемом массиве. Создадим массив чисел от 11 до 16 с шагом 2:

```
In  [2]: a = np.arange(11, 16, 2, dtype=float)
Out [2]: array([ 11., 13., 15.])
```

Функция **numpy.linspace(***start, stop, n***)** создаёт массив из *n* чисел, распределённых с одинаковым интервалом в диапазоне от *start* до *stop*. Функция также может принимать дополнительный аргумент **dtype** для указания типа чисел в создаваемом массиве. Например, создадим массив из 6 чисел с плавающей точкой в диапазоне от 0 до 1:

```
In  [3]: a = np.linspace(0, 1, 6, dtype=float)
Out [3]: array([ 0., 0.2, 0.4, 0.6, 0.8, 1.0])
```

Массивы, заполненные нулями или единицами

numpy.zeros(*n***)** —создаёт массив из *n* нулей:

```
In  [1]: a = np.zeros(7, dtype=int)
Out [1]: array([0, 0, 0, 0, 0, 0, 0])
```

numpy.ones(*n***)** — создаёт массив из *n* единиц:

```
In  [2]: a = np.ones(3, dtype=int)
Out [2]: array([1, 1, 1])
```

2.3. Доступ к элементам массива

Доступ к элементам массива для выполнения операций чтения и присваивания осуществляется так же, как в случае списков Python, — с помощью указания индекса искомого элемента в квадратных скобках:

```
In  [1]: a1 = np.array([1, 4, 5, 8])
In  [2]: a1[3]
Out [2]: 8.0
In  [3]: a1[0] = 5.
In  [4]: a1
Out [4]: array([ 5., 4., 5., 8.])
```

Рассмотрим двумерный массив *a2* из *n* строк и *m* столбцов. Данный массив представляет собой матрицу $n \times m$. Будем нумеровать строки массива индексом *i* (пробегает значения от *0* до *n-1*), а столбцы — индексом *j* (пробегает значения от *0* до *m-1*). Доступ к элементам двумерного массива осуществляется с помощью указания соответственно номеров строки и столбца в квадратных скобках через запятую. Например, *a2[i, j]* — это элемент из *i*-й строки и *j*-го столбца. Рассмотрим конкретный пример с массивом из двух строк и трёх столбцов:

```
In  [5]: a2 = np.array([[1, 2, 3], [4, 5, 6]], dtype=float)
In  [6]: a2[0, 1]
Out [6]: 2.0
```

Срезы массивов

Мощным и удобным инструментом работы с массивами в библиотеке **NumPy** являются так называемые срезы. Общий синтаксис среза некоторого одномерного массива **a**:

```
a[start:stop:step]
```

Данная конструкция возвращает подмножество («срез») элементов из массива *a* с индексами в диапазоне от *start* до *stop-1* с шагом по индексу *step*. Например, создадим массив из 9 чисел в диапазоне от 0 до 2:

```
In  [1]: a1 = np.linspace(0, 2, 9)
In  [2]: a1
Out [2]: array([ 0.0, 0.25, 0.5, 0.75, 1.0, 1.25, 1.5, 1.75, 2.0])
```

Выделим из данного массива подмножество, содержащее элементы массива *a1* начиная с первого и заканчивая пятым:

```
In  [3]: a1[1:5:1]
Out [3]: array([0.25, 0.5, 0.75, 1.0])
```

В данном примере *start=1, stop=5, step=1*.

Любой из компонентов среза (*start, stop, step*) не обязателен. Например, если не указать третий аргумент (*step*), то будет использовано его значение по умолчанию (*step=1*). Для созданного выше массива при *start=1*, *stop=3* будут возвращены первый и второй элементы:

```
In  [4]: a1[1:3]
Out [4]: array([0.25, 0.5])
```

Если не указаны первый (*start*) и/или второй (*stop*) аргументы, то будут использованы их значения по умолчанию: 0 для *start* и **array.size** для *stop*, где **array.size** — число элементов в данном массиве. Таким образом, конструкция вида array[::] возвращает все элементы массива **array**. В случае когда не указывается ни один из аргументов (*start, stop, stop*), один знак двоеточия может быть опущен:

```
In  [5]: a1[::]
Out [5]: array([0.0, 0.25, 0.5, 0.75, 1.0, 1.25, 1.5, 1.75, 2.0])
In  [6]: a1[:]
Out [6]: array([0.0, 0.25, 0.5, 0.75, 1.0, 1.25, 1.5, 1.75, 2.0])
```

Для примера выведем на экран каждый второй элемент массива *a1*:

```
In  [7]: a1[::2]
Out [7]: array([0. ,  0.5,  1. ,  1.5,  2.])
```

(*step=2, start* и *stop* не указаны).

Все элементы массива *a1* начиная с 4-го:

```
In  [8]: a1[4:]
Out [8]: array([1.   ,  1.25,  1.5 ,  1.75,  2.])
```

(указан *start=4*, *stop* и *step* не указаны).

Отрицательное значения индексов *start* и/или *stop* интерпретируются как значения $n + start$ и/или $n + stop$ соответственно, где n — число элементов в соответствующем измерении массива. Например:

```
In  [9]: a1[-3::]
Out [9]: array([1.5 ,  1.75,  2.])
In  [10]: a1[:-3:]
Out [10]: array([0.0, 0.25, 0.5, 0.75, 1.0, 1.25])
```

Отрицательное значение индекса *step* соответствует перебору элементов в выбранном диапазоне в обратном порядке, то есть от больших индексов к меньшим:

```
In  [11]: a1[::-1]
Out [11]: array([2., 1.75, 1.5, 1.25, 1., 0.75, 0.5, 0.25, 0.])
```

Аналогичным образом можно применять операции срезов для любого из измерений многомерных массивов. Общая конструкция среза для двумерного массива:

```
a2[start_row:stop_row:step_row, start_column:stop_column:step_
column]
```

возвращает двумерный массив-срез, содержащий строки исходного массива *a2* с номерами от *start_row* до *stop_row-1* с шагом *step_row* и столбцы с номерами от *start_column* до *stop_solumn-1* с шагом *step_column*. Как и в случае одномерных массивов, любой из трёх аргументов в каждом из измерений является необязательным. Например, конструкции вида

```
a2[::, ::]
```

и

```
a2[:, :]
```

возвращают элементы из всех строк и всех столбцов исходного массива *a2*, то есть весь массив *a2*. Конструкция

```
a2[:, j]
```

возвращает элементы из всех строк *j*-го столбца массива *a2* (проще говоря, *j*-й столбец). Конструкция

```
a2[i, :]
```

возвращает элементы из всех столбцов *i*-й строки массива *a2*, то есть *i*-ю строку.

Рассмотрим конкретный пример. Создадим двумерный массив *a2* — матрицу из трёх строк и четырёх столбцов:

```
In  [9]: a2 = np.array([[-1, 0, 2, 4], [10, 20, 40, 80], [7, 7, 7,
7]])
Out [9]:
array([[-1, 0, 2, 4],
           [10, 20, 40, 80],
           [7, 7, 7, 7]])
```

Нулевой столбец массива *a2*:

```
In  [10]: a2[:, 0]
Out [10]: array([-1, 10, 7])
```

Первый столбец массива *a2*:

```
In  [11]: a2[:, 1]
Out [11]: array([0, 20, 7])
```

Первая строка массива *a2*:

```
In  [12]: a2[1, :]
Out [12]: array([10, 20, 40, 80])
```

Вторая строка массива *a2*:

```
In  [13]: a2[2, :]
Out [13]: array([7, 7, 7, 7])
```

Срезы можно использовать и в операциях присваивания. Например, создадим одномерный массив *new_array*, элементы которого являются суммой соответствующих элементов из первой и второй строк двумерного массива *a2*:

```
In  [14]: new_array = a2[1, :] + a2[2, :]
In  [15]: new_array
Out [15]: array([17, 27, 47, 87])
```

Удобство срезов заключается в возможности проведения операций сразу над некоторой частью массива без необходимости

явного использования цикла по перебору элементов. Работа со срезами оптимизирована так, что она осуществляется быстрее, чем ручной перебор элементов с помощью циклов **for**.

Замечание. Для списков Python реализована аналогичная операция взятия срезов.

2.4. Функции для работы с массивами

Рассмотрим некоторые наиболее распространённые функции для работы с массивами. Напомним, что все сущности в Python являются объектами. Массивы **NumPy** являются объектами класса **ndarray**. Обращение к методам работы с данным объектом осуществляется через точку.

Сумма элементов массива — метод **sum()**:

```
In  [1]: a = np.array([2, 4, 3], float)
In  [2]: a.sum()
Out [2]: 9.0
```

Произведение элементов массива — метод **prod()**:

```
In  [3]: a.prod()
Out [3]: 24.0
```

Среднее арифметическое значение среди элементов массива — метод **mean()**:

```
In  [4]: a.mean()
Out [4]: 3.0
```

Минимальный элемент массива — метод **min()**:

```
In  [5]: a.min()
Out [5]: 2.0
```

Максимальный элемент массива — метод **max()**:

```
In  [6]: a.max()
Out [6]: 4.0
```

Индекс минимального элемента массива — метод **argmin()**:

```
In  [7]: a.argmin()
Out [7]: 0
```

Индекс максимального элемента массива — метод **argmax()**:

```
In  [8]: a.argmax()
Out [8]: 1
```

Рассмотрим также некоторые поля объектов типа **ndarray**. Обращение к полям объектов в Python осуществляется через точку без круглых скобок после имени поля.

Полное число элементов массива хранится в поле **size**:

```
In  [9]: a.size
Out [9]: 3
```

Размерность массива — поле **ndim**:

```
In  [10]: a.ndim
Out [10]: 1
```

Количество элементов в каждом из измерений массива содержится в поле **shape**. Данное поле является кортежем (см. параграф 1.4). Число элементов кортежа соответствует числу измерений. Каждый элемент кортежа — число элементов в данном измерении. Например, для одномерного массива *a*, состоящего из трёх элементов:

```
In  [11]: a.shape
Out [11]: (3,)
```

Для двумерного массива *a2*, состоящего из двух строк и трёх столбцов:

```
In  [12]: a2 = np.array([[0, 1, 2], [10, 20, 101]])
In  [13]: a2.shape
Out [13]: (2, 3)
```

2.5. Универсальные функции

Универсальные функции представляют собой оболочки для обычных математических функций, выполняющие обработку массивов типа **ndarray** поэлементно. Универсальные функции реализованы в виде скомпилированного кода на языке C, что обеспечивает быструю скорость выполнения операций.

Например, все арифметические операции, будучи применены к массиву **ndarray** целиком (без указания индексов элементов), применяются поэлементно. Данная процедура выполняется значительно быстрее, чем если бы операция выполнялась в обычном цикле **for**. Например, создадим массивы a и b из трёх чисел и применим к ним арифметические операции:

```
In  [1]: a = np.array([1,2,3], float)
In  [2]: b = np.array([5,2,6], float)
In  [3]: a + b
Out [3]: array([6., 4., 9.])
In  [4]: a – b
Out [4]: array([-4., 0., -3.])
In  [5]: a * b
Out [5]: array([5., 4., 18.])
In  [6]: b / a
Out [6]: array([5., 1., 2.])
```

Возведение в степень:

```
In  [7]: b**a
Out [7]: array([5., 4., 216.])
```

Матричное произведение массивов осуществляется с помощью функции **numpy.dot()**:

```
In  [8]: a = np.array([1, 2, 3], float)
In  [9]: b = np.array([0, 1, 1], float)
In  [10]: np.dot(a, b)
Out [10]: 5.0
```

Многие стандартные математические функции реализованы в **NumPy** в виде универсальных функций. Например, **numpy.sin()**,

numpy.cos(), **numpy.sqrt()**, **numpy.exp()** и т. д. При передаче массивов **ndarray** в универсальные математические функции, определённые в библиотеке **NumPy**, автоматически создаётся массив с соответствующими значениями функции от каждого элемента исходного массива. Например:

```
In  [11]: x = np.linspace(0, 9, 10)
In  [12]: x
Out [12]: array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.])
In  [13]: y = np.sin(x)
In  [14]: y
Out [14]: array([ 0.,   0.84147098,   0.90929743,   0.14112001,
-0.7568025,  -0.95892427,  -0.2794155,   0.6569866  ,   0.98935825,
0.41211849])
In  [15]: type(y)
Out [15]: numpy.ndarray
In  [16]: y2 = np.sqrt(x)
In  [17]: y2
Out [17]: array([ 0.,   1.,   1.41421356,   1.73205081,   2.,
2.23606798,  2.44948974,  2.64575131,  2.82842712,  3.])
```

Таким образом, **NumPy** позволяет осуществлять быстрые математические преобразования над массивами целиком без необходимости применения циклов для поэлементной обработки массивов.

2.6. Работа с файлами

Для чтения табулированных данных из текстовых файлов в библиотеке **NumPy** реализована функция **loadtxt()**. Её синтаксис:

```
numpy.loadtxt(fname, dtype=<type ‘float’>, comments=’#’, delimi
ter=None, skiprows=0, usecols=None)
```

где

- *fname* — строка-имя текстового файла (вместе с путём до него);
- **dtype** — тип, к которому необходимо преобразовать считываемые данные;

- **comments** — символы, которые использованы в текстовом файле для обозначения комментариев, то есть строк, которые не нужно считывать;
- **delimeter** — использованный в файле разделитель между столбцами. Разделителями по умолчанию считаются пробелы и знаки табуляции;
- **skiprows** — количество строк (начиная с первой), которые необходимо пропустить при считывании файла;
- **usecols** — кортеж, содержащий номера столбцов, которые необходимо считать из файла. Нумерация начинается с 0.

Функция **loadtxt()** возвращает массив типа **ndarray**, хранящий данные из файла, с тем же количеством строк и столбцов.

Для примера рассмотрим текстовый файл **data.txt**, расположенный в корне диска **D** и содержащий следующую информацию:

```
#данные расчёта: x, t, v, T
0       0       1       0
1       0.84    0.54    1
2       0.91    –0.42   4
3       0.94    –0.99   9
```

В файле первая строка является комментарием, сообщающем о содержимом файла: нулевой столбец — координата x, первый — время t, второй — скорость v, третий — температура T. Запишем первые три столбца файла в переменную *data:*

```
 In [1]: data = np.loadtxt("D:\data.txt", comments='#', usecols=(0,
1, 2))
 In   [2]: data
 Out  [2]:
 array([[ 0.   ,  0.   ,  1.   ],
        [ 1.   ,  0.84,  0.54],
        [ 2.   ,  0.91, -0.42],
        [ 3.   ,  0.94, -0.99]])
```

Выведем на экран значения координаты (нулевой столбец массива *data*):

```
In  [3]: data[:, 0]
Out [3]: array([0., 1., 2., 3.])
```

Выведем на экран значения скорости (второй столбец массива *data*):

```
In  [4]: data[:, 2]
Out [4]: array([ 1.   ,  0.54, -0.42, -0.99])
```

Для записи массива числовых данных в текстовый файл в библиотеке **NumPy** имеется функция **savetxt()** со следующим синтаксисом:

```
numpy.savetxt(fname, X, fmt='%.18e', delimiter=' ')
```

где

- *fname* — строка-имя текстового файла для сохранения данных (вместе с путём до него);
- *X* — одномерный или двумерный массив типа **ndarray**, который требуется сохранить в файл;
- **fmt** — формат для сохранения данных (см. описание спецификаторов формата в параграфе 1.6);
- **delimeter** — строка или символ, который следует использовать как разделитель между столбцами.

С полным списком аргументов функции **savetxt()** можно ознакомиться с помощью функции **help(numpy.loadtxt)** в консоли или на сайте библиотеки **NumPy**.

Задание для самостоятельной работы

Напишите программу, выполняющую следующие операции:

1. Поэлементное сложение двух одномерных массивов, *a1* и *a2*, заданного размера *N*. Элементы массива *a1*: числа, равнораспределённые в диапазоне от 0 до 100. Элементы массива *a2*: числа, равнораспределённые в интервале от 0 до 1.

2. Перемножение матриц, *m1* и *m2*, с заданным числом строк и столбцов. Элементы в каждом столбце матрицы *m1*: элементы из массива *a1*. Элементы в каждой строке матрицы *m2*: элементы из массива *a2*.

Все массивы должны иметь тип **ndarray.** Размеры массивов и матриц произвольны и задаются как параметры. Заполнение матриц *m1* и *m2* значениями следует сделать с использованием операции срезов массивов. Операции 1 и 2 необходимо реализовать двумя способами: непосредственно с помощью циклов **for** и используя возможности библиотеки **NumPy**, позволяющие выполнять операции над массивами целиком. Сравните время выполнения операций 1 и 2, выполненных обоими способами. Время выполнения некоторого участка кода можно вычислить, применяя функцию **default_timer()** из модуля **timeit**. Данная функция возвращает время в секундах, прошедшее с момента запуска интерпретатора. Ниже для примера приведён код для выполнения и анализа времени выполнения операции сложения массива *a1*, заполненного нулями, с массивом *a2*, заполненным единицами.

```
# загрузка функции-таймера
from timeit import default_timer as timer
# подключение библиотеки numpy
import numpy as np

N=1000 # размер одномерных массивов
# создание массива, заполненного нулями
a1 = np.zeros((N))
# создание массива, заполненного единицами
a2 = np.ones((N))
# массив для сохранения результата сложения
result = np.zeros((N))

# сложение созданных массивов напрямую в цикле
print('a) сложение массивов напрямую в цикле:')
start = timer() # запуск таймера
for i in range(N):
result[i] = a1[i] + a2[i]
```

```
end = timer()   # остановка таймера
print(u'  время выполнения: %.3e с' % (end - start))

# быстрое сложение
print('б) сложение с помощью numpy:')
start = timer() # запуск таймера
result = a1 + a2
end = timer()   # остановка таймера
print('  время выполнения: %.3e с' % (end - start))
```

Вопросы для самоконтроля

1. Объясните, в чём отличие функций **arange()** и **linspace()**?
2. Что такое срез массива?
3. Допустим, *m* — это двумерный массив. Что означают конструкции *m[:, j]* и *m[i, :]*?
4. Что означают отрицательные индексы в срезах массивов?
5. Как быстро определить номера минимальных и максимальных элементов в данном массиве?
6. Что такое универсальные функции?

Рекомендуемая литература

1. Нуньес-Иглесиас, Х. Элегантный SciPy / Х. Нуньес-Иглесиас, Ш. ван дер Уолт, Х. Дэшноу. — М. : ДМК Пресс, 2018. — 266 с.
2. The SciPy community. NumPy quickstart tutorial [Электронный ресурс] / The SciPy community. — 2008–2019. — URL: https://numpy.org/devdocs/user/quickstart.html (дата обращения: 16.10.2019)

Глава 3
Библиотека визуализации Matplotlib

В данной главе описываются базовые возможности библиотеки **Matplotlib**, предназначенной для визуализации данных. Разбираются примеры по рисованию графиков с параметрами по умолчанию, создания рисунков с настраиваемыми параметрами, рисования графиков по данным из текстовых файлов, рисования двумерных распределений, а также добавления на рисунки подписей с математическими символами. В завершение приводится задание для самостоятельного написания программы создания рисунка с заданным параметрами, а также вопросы для самоконтроля.

3.1. Общие сведения

Библиотека **Matplotlib** предназначена для создания двумерной и трёхмерной графики. В качестве основного типа данных используются массивы типа **ndarray**, определённые в библиотеке **NumPy**. Имеются функции для рисования графиков в прямоугольных и полярных координатах, диаграмм рассеяния (англ., scatter plot), графиков с указанием величин ошибок (англ., errorbar), гистограмм, круговых и столбчатых диаграмм (англ., pie chart и bar chart), двумерных распределений скалярных и векторных величин и т. д. Высокое качество получаемых изображений позволяет использовать их в научных публикациях. Библиотека может использоваться в скриптах Python, в интерактивной оболочке IPython, веб-приложениях, графическом интерфейсе пользователя.

Библиотека имеет иерархическую структуру и использует принципы объектно-ориентированного программирования. Базовые функции для создания графиков расположены в модуле

matplotlib.pyplot. Данный модуль позволяет создавать графику в стиле команд MATLAB.

3.2. Создание простого рисунка с параметрами по умолчанию

Рассмотрим простейший пример рисования графика с помощью функций модуля **pyplot**. Общепринятым способом подключения данного модуля является команда

```
import matplotlib.pyplot as plt
```

то есть модуль подключается с псевдонимом plt.

Создадим скрипт для рисования простого рисунка — графика зависимости $y = \sin x$. Для отображения набора пар значений x и y используется функция **plot()** из модуля **matplotlib.pyplot**. Синтаксис вызова функции:

```
matplotlib.pyplot.plot(x, y, args)
```

где x — массив значений, отображаемых по оси абсцисс (массив координат), y — массив значений, откладываемых по оси ординат (массив значений функции), *args* — перечисленные через запятую необязательные дополнительные аргументы для настройки графика. Если в функцию **plot()** передаётся только один аргумент-массив, то в качестве массива координат для рисования графика будут использованы индексы элементов этого массива.

Для изображения функциональной зависимости $y = f(x)$ необходимо создать массивы координат $x = [x_0, x_1, \ldots, x_N]$ и массивы значений функции $y = [y_0, y_1, \ldots, y_N]$ в соответствующих точках. Далее приведён текст программы для рисования графика зависимости $y = \sin(x)$. Массив координат создаётся с помощью функции **linspace()**, а массив значений функции — с помощью универсальной функции **sin()** из библиотеки **NumPy**. В конце программы вызывается функция **show()** из модуля **pyplot**, предназначенная для отображения рисунка на экране компьютера.

```
# подключение модуля pyplot под псевдонимом plt
import matplotlib.pyplot as plt
# подключение библиотеки numpy под псевдонимом np
import numpy as np

# массив координат - 30 точек, равнораспределенных
# в диапазоне от 0 до 10
x = np.linspace(0.0, 10.0, 30)
# массив значений функции в заданных координатах
y = np.sin(x)

# рисование графика функции с помощью функции plot
plt.plot(x, y)
# отображение рисунка на экране
plt.show()
```

Результат работы программы показан на рис. 2.

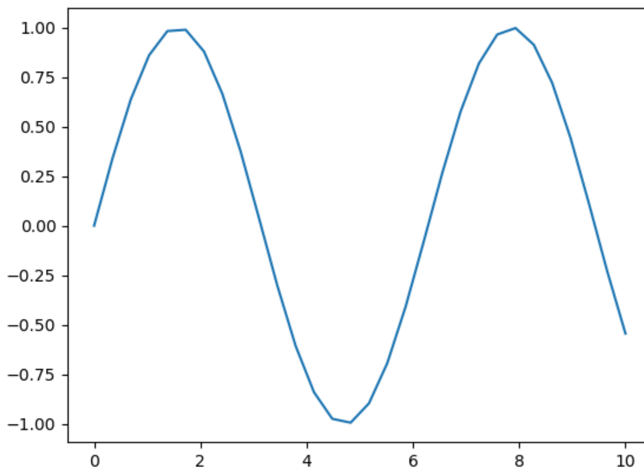

Рис. 2. *График функции sin(x), построенный с параметрами по умолчанию (см. пояснение в тексте параграфа 3.2)*

3.3. Создание рисунка с настраиваемыми параметрами

Все объекты в **Matplotlib** имеют иерархическую (древоподобную) структуру. Основными элементами являются *рисунок*, *панели* и *оси координат* (рис. 3).

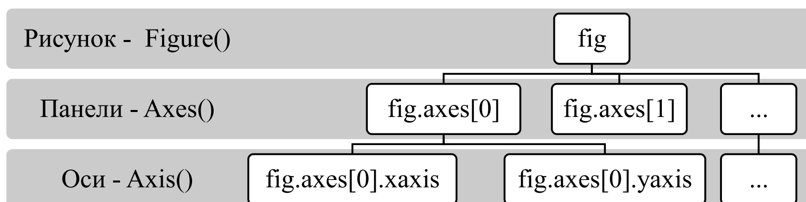

Рис. 3. *Схема иерархии объектов, служащих для создания рисунка в библиотеке* **Matplotlib**

На верхнем уровне иерархии находится окно рисунка, которое является контейнером для остальных элементов. *Рисунок* является объектом класса **Figure**.

На рисунке может располагаться несколько (одна или больше) *панелей* — независимых областей рисования. Панели являются объектами класса **Axes**.

Каждая из панелей содержит основные элементы рисунка: оси координат, метки на осях, текстовые метки, графики и т. д.

Алгоритм работы с рисунками можно записать в виде следующей последовательности действий:

1. Создание окна рисунка. Окно создаётся с помощью функции **matplotlib.pyplot.figure()** из модуля **pyplot**. Например, конструкция вида

```
fig = plt.figure(args)
```

создаёт окно рисунка и ассоциирует его с переменной *fig*. Данная переменная является объектом класса **Figure**. Настройки окна в дальнейшем осуществляются посредством обращения к полям и методам этого объекта. Функция **figure()** может принимать необязательные аргументы для настройки окна рисунка. Например, конструкция

```
fig = plt.figure(figsize=(width, height))
```

создаёт рисунок шириной *width* и высотой *height* (в дюймах).

Будем в дальнейшем использовать имя *fig* в качестве имени по умолчанию для создаваемого окна.

2. Добавление в окно панелей для рисования графиков. Одним из способов добавления панели на рисунок является использование метода **add_subplot(***nrows, ncols, index***)** объекта «рисунок». Три целочисленных аргумента указывают, что панель займёт на рисунке положение под номером *index* в таблице из *nrows* строк и *ncols* столбцов. Нумерация начинается с 1 в левом верхнем углу таблицы и увеличивается слева направо. Например, конструкция вида

```
ax = fig.add_subplot(1,1,1)
```

добавляет в ранее созданное окно *fig* одну панель и ассоциирует её с переменной *ax*. Запятые между целочисленными аргументами **add_subplot()** необязательны.

Настройки панели в дальнейшем осуществляются посредством обращения к полям и методам этого объекта. Рассмотрим основные свойства панели:

- **set_xlim(***a, b***)** — установка диапазона отображаемых по оси абсцисс значений от *a* до *b;*
- **set_ylim(***a, b***)** — установка диапазона отображаемых по оси ординат значений от *a* до *b*;
- **set_xlabel(***str***)** — установка строки *str* в качестве подписи оси абсцисс;
- **set_ylabel(***str***)** — установка строки *str* в качестве подписи оси ординат;
- **set_title(***str***)** — установка строки *str* в качестве заголовка панели;
- **set_xscale(***str***)** — установка масштаба по оси абсцисс, если *str=”lin”*, то будет использован линейный масштаб, если *str=”log”*, то логарифмический. По умолчанию все оси отображаются с линейным масштабом;
- **set_yscale(***str***)** — установка масштаба по оси ординат;
- **legend()** — отображение легенды на рисунке.

3. Рисование графиков. В **Matplotlib** реализовано большое количество функций для рисования графиков, например:

- **plot(***x, y, args***)** — рисование графика зависимости *y* от *x* с помощью линий или маркеров. Через запятую после *y* можно перечис-

лить необязательные дополнительные аргументы *args* для настройки свойств линий и маркеров (цвет, тип, толщина линии, тип маркера и т. п.);

- **errorbar(***x, y, xerr, yerr***)** — рисование графика зависимости *y* от *x* с указанием баров ошибок *xerr* для *x* и *yerr* для *y*;
- **hist(***x, bins***)** — построение гистограммы величины *x*, разбитой на число столбцов, равное целому числу *bins*;
- **contour(***X, Y, Z, [levels], args***)** — рисование контуров величины *Z*, представляющей собой двумерный массив чисел. Аргументы *X* и *Y* — массивы координат, в которых определены значения величины *Z*. Аргумент *[levels]* представляет собой список значений величины *Z*, которые необходимо изобразить контурами;
- **contourf(***X, Y, Z, [levels], [args]***)** — то же, что **contour()**, только пространство между контурами заполняется цветовой заливкой;
- прочие.

4. Отображение окна рисунка на экране или его сохранение в файл. Отображение рисунка на экране выполняется методом **show()**. Сохранение рисунка в файл — методом **savefig()**, принимающим в качестве аргументов строку-имя файла для сохранения, а также необязательный список дополнительных аргументов, указывающих параметры сохранения.

Рассмотрим пример скрипта, в котором реализованы основные возможности создания рисунка с графиками нескольких функций и ручной настройки его параметров: размера текста в подписях, диапазонов по осям, подписей осей, заголовка рисунка:

```
# подключение библиотеки numpy под псевдонимом np
import numpy as np

# подключение модуля pyplot из библиотеки matplotlib
# под псевдонимом plt
import matplotlib.pyplot as plt

# пользовательская переменная для хранения размера шрифта
fsize=12
```

```
# настройка типа шрифта на рисунке с помощью изменения
# записей в словаре rcParams из модуля pyplot
plt.rcParams['font.family'] = 'serif'
plt.rcParams['font.serif'] = 'Times New Roman'
# настройка размера шрифта в различных частях рисунка
# в заголовке:
plt.rcParams['axes.titlesize'] = fsize
# в подписях осей:
plt.rcParams['axes.labelsize'] = fsize
# в подписях меток на осях:
plt.rcParams['xtick.labelsize'] = fsize
plt.rcParams['ytick.labelsize'] = fsize
# в легенде рисунка:
plt.rcParams['legend.fontsize'] = fsize

# массив координат – 50 точек в диапазоне [0, 10]
x = np.linspace(0.0, 10.0, 50)

# создаём окно рисунка.
# Для дальнейшей работы рисунок ассоциируется с переменной
# fig
fig=plt.figure()

# добавляем панель (оси координат) с именем ax в окно fig.
# в дальнейшем настройка осей производится
# через обращение к переменной ax.
# аргументы 1, 1, 1 указывают, что на рисунке будет
# только одна панель для рисования графиков
ax=fig.add_subplot(1,1,1)
```

```
# график синуса:
# кружки (o), соединённые сплошной (-) чёрной линией.
# графику присваивается строковый идентификатор ‹1›
# для дальнейшего отображения в легенде

ax.plot(x, np.sin(x), 'ko-', label='1')


# график косинуса:
# квадратики (s, размером 3), соединённые сплошной (-)
# оранжевой линией толщиной 1.
# графику присваивается строковый идентификатор ‹2›
# для отображения в легенде

ax.plot(x, np.cos(x), 'ks-', color='orange', linewidth=1,
markersize=3.0, label='2')

# график синуса в квадрате:
# треугольники (^), соединённые сплошной (-) лиловой
# линией толщиной 1.
# графику присваивается строковый идентификатор ‹3›
# для отображения в легенде

ax.plot(x, (np.sin(x))**2.0, 'k^-', color='magenta', linewidth=1,
label='3')


# график функции f(x)=x^0.15:
# чёрная штриховая линия толщиной 1.
# графику присваивается строковый идентификатор ‹x^2'
# для отображения в легенде.
# символ r и знаки доллара внутри строки позволяют
# вводить математические символы с помощью команд TeX

ax.plot(x, (x)**0.15, 'k--', linewidth=1, label=r'$x^2$')


# легенда

ax.legend(loc='best')


# диапазон отображаемых значений по оси x

ax.set_xlim(-1.0, 11.0)

# диапазон отображаемых значений по оси y
```

```
ax.set_ylim(-1.5, 1.5)


# подпись по оси x
ax.set_xlabel(r'$x$')
# подпись по оси y
ax.set_ylabel(r'$f(x)$')


# заголовок рисунка
ax.set_title('Мой первый рисунок')
# сетка на рисунке
ax.grid()


# сохраняем в файл с именем fig1 типа PNG с разрешением
# 300 точек на дюйм
# (dpi - dots per inch), с альбомной ориентацией
fig.savefig("fig1.png", orientation='landscape', dpi=300)
```

Результат работы программы показан на рис. 4.

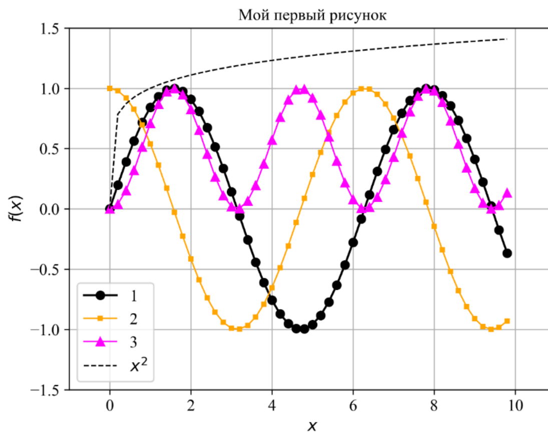

Рис. 4. *Демонстрация возможностей создания одномерных графиков с помощью* **pyplot**

Обсудим приведённую выше программу для рисования графиков.

Каждый график добавляется на текущую панель рисунка с помощью функции **plot()**.

График функции синус на рис. 1 изображён с помощью команды

```
ax.plot(x, np.sin(x), 'ko-', label='1')
```

Тип точек и соединительной линии может быть указан с помощью дополнительного строкового аргумента в команде **plot()** после первых двух обязательных аргументов — отображаемых массивов. Данная строка имеет следующий формат

```
'CDL'
```

где

C — буква-код, указывающая цвет линии. Возможные варианты: **k** (чёрный), **b** (blue — синий), **c** (cyan — голубой), **g** (green — зелёный), **m** (magenta — пурпурный), **r** (red — красный), **y** (yellow — жёлтый), **w** (white — белый).

• *D* — символ, указывающий тип соединительной точки (маркера). Варианты: **s** (square — квадратик), **d** (diamond — ромбик), **o** (кружок), * (звёздочка), ^ (треугольник вверх), **v** (треугольник вниз).

• *L* — тип линии. Варианты: - (сплошная линия), -- (штриховая линия), : (пунктирная линия), -. (штрихпунктирная линия).

Например, строка *'ko-'* в примере выше означает, что график нужно нарисовать чёрной сплошной линией, а соединительные точки отображать с помощью кружочков.

Каждый из указанных параметров можно настроить с помощью отдельного ключевого слова в списке аргументов функции **plot()**. Например, график косинуса на рис. 4 нарисован жёлтыми квадратиками размером 3 (**markersize** = 3.0), соединёнными сплошной оранжевой линией (**color** = 'orange') толщиной 1 (**linewidth** = 1),

```
ax.plot(x, np.cos(x), 'ks-', color='orange', linewidth=1,
markersize=3.0, label='2')
```

Аргумент **label** в команде **plot()** присваивает текущему графику имя-строку, которая будет отображена в легенде рисунка.

3.4. Добавление нескольких панелей на рисунок

Команда **add_subplot()** добавляет на текущий рисунок (*fig* в примере выше) несколько осей координат (не менее 1) для рисования графиков. Синтаксис этой функции:

```
ax_k = add_subplot(n, m, k)
```

Запятые могут быть опущены. Эта команда создаёт на рисунке *fig* таблицу из $n \times m$ ячеек (n строк и m столбцов) и помещает текущую панель в ячейку с индексом k. В таблице может быть не более $n \times m$ панелей, то есть число k может пробегать значения от 1 до $n \times m$. Текущая создаваемая командой **add_subplot()** панель ассоциируется с пользовательской переменной *ax_k*. На рис. 5 приведена диаграмма, на которой схематически показано, как можно расположить 6 различных панелей (осей координат) в таблице из трёх строк и двух столбцов на одном рисунке.

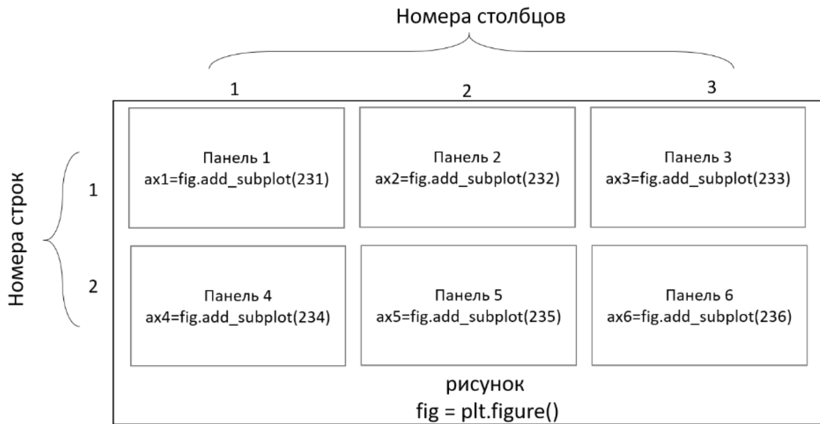

Рис. 5. *Диаграмма расположения шести панелей с именами ax1, … ax6 в три столбца и две строки на рисунке fig*

3.5. Двумерные графики

В **Matplotlib** реализовано несколько способов построения двумерных графиков. Рассмотрим способ графического отображения скалярного поля — функции $F(x, y)$, значения которой заданы на координатной плоскости (x, y) в области $x \in [x_L, x_R]$, $y \in [y_T, y_B]$. Распределения $F(x, y)$ можно изобразить с помощью изолиний, называемых также линиями уровня, — линий в плоскости (x, y), на которых функция F имеет определённое значение. Для построения изолиний в **Matplotlib** имеется две функции.

Функция

```
contour(x, y, f, levels=[values_list], args)
```

отображает изолинии величины, заданной в виде двумерного массива *f*. Аргументы *x* и *y* представляют собой массивы координат точек на плоскости, в которых задана величина *f*. Параметр *[values_list]* — список значений *f*, которые необходимо отобразить изолиниями. Если вместо списка уровней указывается целое число, то на графике будет отображено соответствующее число равнораспределённых изолиний. Функция может принимать дополнительные необязательные аргументы *args*, перечисляемые через запятую и предназначенные для настройки параметров отображения изолиний. Например, аргумент **cmap**=*'Name'* указывает, что для отображения изолиний следует использовать цветовую схему с именем *Name*. Полный список имеющихся цветовых схем можно просмотреть с помощью команды **matplotlib.pyplot.colormaps()**.

Величины *x* и *y* могут быть либо одномерными массивами координат в каждом из направлений, либо матрицами, задающими сетку этих координат. В первом случае длины массивов *x* и *y* должны соответствовать числу столбцов и строк матрицы *f* соответственно. Во втором случае размеры (количество строк и столбцов) массивов *x, y* и *f* должны быть одинаковы. Удобным способом создания матрицы координат является функция **meshgrid** из библиотеки **NumPy**. Конструкция вида

```
xx, yy = meshgrid(x, y)
```

создаёт на основе одномерных массивов координат *x* и *y* сет-

ки соответствующих координат в двумерных массивах *xx* и *yy*. На рис. 6 продемонстрирована работа функции **meshgrid**.

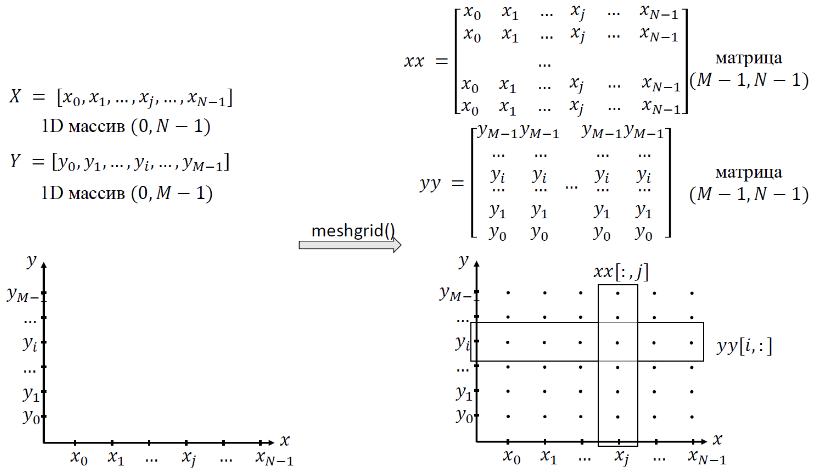

Рис. 6. *Схематическая иллюстрация работы функции meshgrid() из библиотеки* **NumPy**

Рассмотрим экспоненциальную функцию вида

$$F(x, y) = \exp(-x^2 - y^2).$$

Ниже приведён листинг программы для отображения 10 изолиний функции $F(x,y)$ в области $x \in [-2,\ 2]$, $y \in [-2,\ 2]$. Одномерные массивы координат создаются с помощью функции **linspace()** из библиотеки **NumPy**. Затем эти массивы преобразуются к матрицам координат *xx* и *yy* с помощью **meshgrid()**. Созданные матрицы координат используются для расчёта значений функции *F* на сетке.

```
# -*- coding: utf-8 -*-
"""
Created on Fri Oct 11 00:00:00 2018

Рисование двумерного распределения скалярной величины
с помощью изолиний (линий уровня)

@author: С.А. Хайбрахманов
```

```
"""
# подключение библиотеки numpy и модуля pyplot
import numpy as np
import matplotlib.pyplot as plt

# создание рисунка размером 15 на 15 см
# с одной панелью
inch = 2.54 # дюйм в см
fig1 = plt.figure(figsize=(15.0/inch, 15.0/inch))
ax1 = fig1.add_subplot(111)
# подписи осей на панели
ax1.set_xlabel(r'$x$')
ax1.set_ylabel(r'$y$')

# массив x-координат - 50 точек в диапазоне от -2 до 2
x = np.linspace(-2.0, 2.0, 50)
# массив y-координат - 50 точек в диапазоне от -2 до 2
y = np.linspace(-2.0, 2.0, 50)
# матрицы (сетка) координат
xx, yy = np.meshgrid(x, y)
# вычисление значений функции на сетке
F = np.exp(-xx**2 - yy**2)

# отображение 10 изолиний величины F.
# график изолиний ассоциируется с переменной CS1
CS1 = ax1.contour(xx, yy, F, 10)

# добавление подписей изолиний на графике CS1.
# с помощью обращения к полю levels (списку изолиний
# на графике CS1) подписи выводятся только
# для каждой второй линии
```

```
ax1.clabel(CS1, CS1.levels[::2])

fig1.show()
```

На рис. 7 показан результат работы программы.

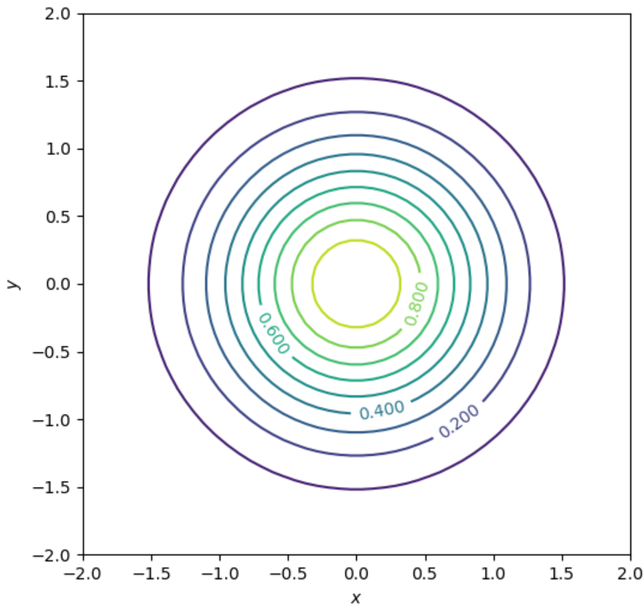

Рис. 7. *Пример использования функции contour()*

Для отображения распределения скалярной величины *f* на плоскости (*x, y*) можно также воспользоваться функцией

```
contourf(x, y, f, levels=[values_list], args)
```

которая действует аналогично **contour()**, но дополнительно выполняет заливку цветом областей между изолиниями.

Векторное поле на плоскости может быть изображено с помощью функции

```
quiver(x, y, Vx, Vy)
```

где x и y — матрицы координат на плоскости, Vx и Vy — матрицы соответствующих координат векторного поля. Легенда для векторного поля добавляется на рисунок командой

```
quiverkey(field_name, X=…, Y=…, U=…, label=…, labelpos=…)
```

где *field_name* — имя векторного поля, нарисованного командой **quiver()**, X и Y указывают координаты для размещения легенды на рисунке ($X=1$, $Y=1$ соответствует правому верхнему углу рисунка), с помощью параметра U указывается длина вектора, который будет отображён в легенде, параметр *label* указывает строку-подпись в легенде, параметр *labelpos* указывает, как будет располагаться текст подписи относительно стрелки в легенде (‘N’ — сверху, ‘W’ — слева, ‘S’ — снизу, ‘E’ — справа).

Далее приведён листинг программы с примером использования **contourf()** и **quiver()** для отображения поля $F(x,y)$ и соответствующего векторного поля $\vec{\nabla}F=(\partial F/\partial x,\partial F/\partial y)$.

```
# -*- coding: utf-8 -*-
"""
Created on Fri Oct 11 00:00:00 2018

Рисование двумерного распределения скалярной величины
с помощью изолиний (линий уровня) и её градиента с помощью век-
торов

@author: С.А. Хайбрахманов
"""
# подключение библиотеки numpy и модуля pyplot
import numpy as np
import matplotlib.pyplot as plt


# создание рисунка размером 15 на 15 см
# с одной панелью
inch = 2.54 # дюйм в см
fig1 = plt.figure(figsize=(18.0/inch, 15.0/inch))
ax1 = fig1.add_subplot(111)
# подписи осей на панели
```

```
ax1.set_xlabel(r'$x$')
ax1.set_ylabel(r'$y$')

# массив x-координат - 50 точек в диапазоне от -2 до 2
x = np.linspace(-2.0, 2.0, 50)
# массив y-координат - 50 точек в диапазоне от -2 до 2
y = np.linspace(-2.0, 2.0, 50)
# матрицы (сетка) координат
xx, yy = np.meshgrid(x, y)
# функция для отображения
def F(x, y):
    return np.exp(-x**2 - y**2)

# градиент функции
def gradF(x, y):
    # x-координата градиента
    gradF_x = -2 * x * F(x, y)
    # y-координата градиента
    gradF_y = -2 * y * F(x, y)
    return [gradF_x, gradF_y]

# вычисление значений функции на сетке
FF = F(xx, yy)
# отображение 20 изолиний величины F с заливкой.
# график изолиний ассоциируется с переменной CS
# аргумент cmap указывает используемую цветовую схему
CS = ax1.contourf(xx, yy, FF, 20, cmap='Blues')

# добавление на рисунок легенды,
# указывающей соответствие цветов заливки
```

```
# уровням величины F на графике CS
fig1.colorbar(CS, label=r'$F(x,y)$')

# вычисление градиента функции на сетке
grad = gradF(xx, yy)

# рисование поля градиента с помощью векторов
# векторное поле ассоциируется с переменной q
q = ax1.quiver(xx, yy, grad[0], grad[1])

# отображение легенды для векторного поля -
# стрелка длиной 1 для масштаба
ax1.quiverkey(q, X=0.5, Y=1.05, U = 1, label=r'$|\vec{\nabla}
F|=1$', labelpos='E')

fig1.show()
```

На рис. 8 показан результат действия программы.

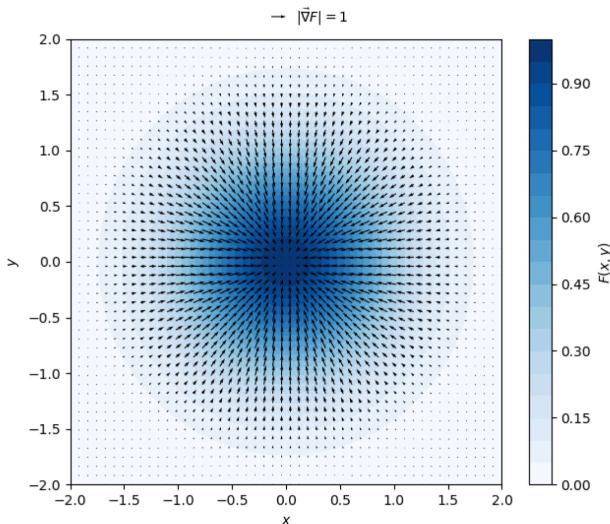

Рис. 8. *Пример использования функций contourf() и quiver()*

3.6. Визуализация данных из текстовых файлов

Для рисования графиков по данным из текстового файла удобно воспользоваться командой **loadtxt()** из библиотеки **NumPy**:

```
from matplotlib import pyplot as plt
import numpy as np

data = np.loadtxt("data.txt")

fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)

ax.set_xlim(-1.0, 11.0)
ax.set_ylim(-1.1, 1.1)
ax.set_xlabel(r'$x$')
ax.set_ylabel(r'$f(x)$')
ax.set_title(u'Данные из файла')

#рисование графика зависимости первого столбца от
#нулевого
ax.plot(data[:, 0], data[:, 1], 'ko-', color='grey', label=r'$d_1$')
#рисование графика зависимости второго столбца от
#нулевого
ax.plot(data[:, 0], data[:, 2], 'gs-', label=r'$d_2$')
```

```
ax.legend(loc='best')
fig.show()
```

Вывод данной программы показан на рис. 9.

Содержимое файла **data.txt**:

0	0	1
1	0.84	0.54
2	0.91	–0.42
3	0.14	–0.99
4	–0.76	–0.65
5	–0.96	0.28
6	–0.28	0.96
7	0.66	0.75
8	0.99	–0.15
9	0.41	–0.91
10	–0.54	–0.84

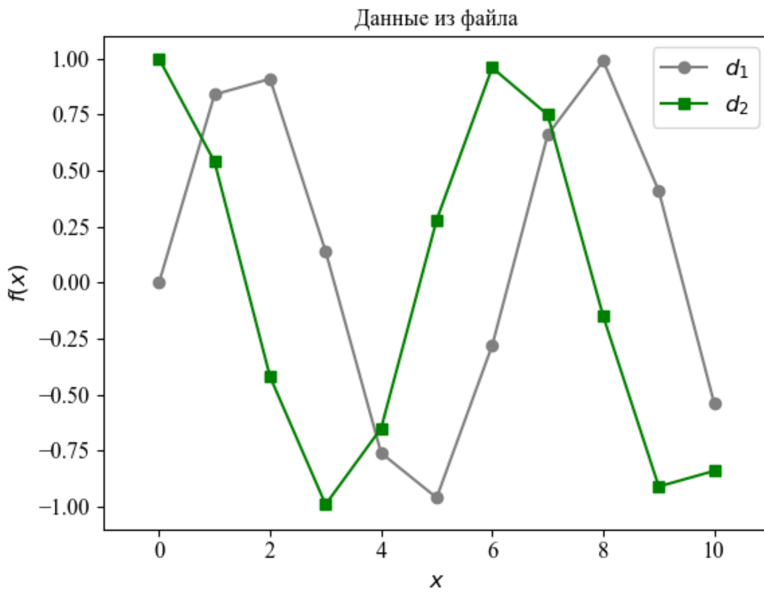

Рис. 9. *Графики данных из файла*

3.7. Математические символы на рисунках

На рисунки **matpotlib** можно добавлять математические надписи. Набор математических выражений осуществляется по правилам системы TeX. Перед строкой, содержащей математическое выражение, помещается буква **r**, а внутри кавычек используются знаки доллара. Например, строка

```
r'$\frac{1}{n}\sum\limits_{i=1}^n(x_i-\langle x\rangle)^2$'
```

будет отображена в соответствующем месте рисунка как

$$\frac{1}{n}\sum_{i=1}^{n}\left(x_i-\langle x\rangle\right)^2.$$

Задание для самостоятельной работы

Модифицируйте программу рисования графиков, приведённую в параграфе 3.3, в соответствии со следующими требованиями:

1. Уберите заголовок рисунка, сетку и легенду.

2. Нарисуйте график каждой из четырёх функций в отдельной панели на рисунке. Оси каждой панели должны быть подписаны. На каждой панели сверху по центру следует добавить текстовую метку: «(а)» для панели 1, «(б)» для панели 2, «(в)» для панели 3, «(г)» для панели 4.

3. График синуса должен быть нарисован сплошной чёрной линией, график косинуса — пунктирной чёрной линией, график квадрата косинуса — штриховой чёрной линией, график функции $x^{0.15}$ — сплошной серой линией.

4. Установите в каждой панели диапазон значений по оси абсцисс от 0 до 2π. Диапазон значений по оси ординат на каждой панели должен быть таким, чтобы график соответствующей функции был виден целиком.

Вопросы для самоконтроля

1. Опишите иерархию объектов на рисунке в **Matplotlib**.
2. Пусть *fig* — объект-рисунок, созданный функцией **figure()**. Поясните результат работы следующих команд:

```
ax1 = fig.add_subplot(2, 1, 1)
ax2 = fig.add_subplot(2, 1, 2)
```

Как будут располагаться панели на рисунке **fig**?

3. Ниже приведены команды рисования графиков зависимостей *y1(x), y2(x), y3(x)*. Какими линиями будут нарисованы графики 1, 2, и 3?

```
plt.plot(x, y1, ‘k-’, label=’1’)
plt.plot(x, y2, ‘g:’ , label=’2’)
plt.plot(x, y3, ‘--’, color=’orange’ , label=’3’)
```

4. В чём отличие функций **contour()** и **contourf()**?
5. Пусть функция *f* определена в некоторой области на плоскости *(x, y)* и имеет в этой области минимальное значение 0, максимальное значение 10. Чем будет отличаться вывод команд **contour**(*x, y, f, levels=[0, 5, 10]*) и **contour**(*x, y, f, 10*)?
6. Воспользуйтесь функцией **help()** и выясните, для чего предназначена функция **matplotlib.pyplot.imshow()**.

Рекомендуемая литература

1. Шабанов, П. А. Научная графика в python [Электронный ресурс] / П. А. Шабанов. — URL: https://github.com/whitehorn/Scientific_graphics_in_python (дата обращения: 16.10.2019).
2. Hunter, J. Matplotlib [Электронный ресурс] / J. Hunter, M. Droettboom. — URL: http://www.aosabook.org/en/matplotlib.html (дата обращения: 16.10.2019).
3. Hunter, J. Matplotlib [Электронный ресурс] / J. Hunter, D. Dale, E. Firing, M. Droettboom and the Matplotlib development team. — URL: https://matplotlib.org/index.html (дата обращения 16.10.2019)

Глава 4
Библиотека SciPy для научных и инженерных расчётов

В данной главе описываются базовые возможности библиотеки **SciPy,** предназначенной для научных и инженерных расчётов. Рассматриваются инструменты численного интегрирования, решения обыкновенных дифференциальных уравнений, а также задачи интерполяции. В завершение приводится самостоятельное задание и вопросы для самоконтроля.

4.1. Общие сведения

SciPy — это открытая библиотека научных инструментов для языка программирования Python. Инструменты **SciPy** предназначены для проведения стандартных научных расчётов. Функции, реализованные в **SciPy**, оптимизированы и протестированы. Они могут использоваться в качестве инструмента для решения различных научных задач без необходимости самостоятельной реализации стандартных методов вычислительной математики и физики. Библиотека **SciPy** опирается на работу с многомерными массивами типа **ndarray** из библиотеки **NumPy**.

Список модулей **SciPy**:

- **scipy.constants** — математические и физические константы;
- **scipy.special** — специальные функции (пр., функции Бесселя, эллиптические функции, гамма-функция, функция ошибок);
- **scipy.integrate** — численное интегрирование (методы трапеции, Симпсона, Ромберга и др.), решение обыкновенных дифференциальных уравнений (ОДУ);

- **scipy.optimize** — решение задач оптимизации, а именно: минимизация скалярных функций многих аргументов, глобальная оптимизация, аппроксимация кривых, анализ данных методом наименьших квадратов, поиск корней нелинейных алгебраических уравнений и систем линейных алгебраических уравнений;
- **scipy.linalg** — линейная алгебра;
- **scipy.sparse** — работа с разреженными матрицами;
- **scipy.interpolate** — интерполяция дискретных данных (одномерных и двумерных);
- **scipy.fftpack** — быстрое преобразование Фурье;
- **scipy.signal** — обработка сигналов;
- **scipy.stats** — статистические алгоритмы.

Рассмотрим пример подключения конкретного модуля **integrate** из библиотеки **SciPy**:

```
In  [1]: from scipy import integrate
```

Справку по целому модулю или по любой его функции можно получить с помощью функции **help()**,

```
In  [2]: help(integrate.quad)

Help on function quad in module scipy.integrate.quadpack:

quad(func, a, b, args=(), full_output=0, epsabs=1.49e-08,
epsrel=1.49e-08, limit=50, points=None, weight=None, wvar=None,
wopts=None, maxp1=50, limlst=50)

Compute a definite integral.

Integrate func from `a` to `b` (possibly infinite interval)
using a

technique from the Fortran library QUADPACK.
```

(Далее следует описание аргументов функции **quad** и возвращаемого значения)

4.2. Численное интегрирование

В модуле **scipy.integrate** реализовано несколько функций для вычисления определённых интегралов. Наиболее простой является функция **quad()**, которая имеет следующий синтаксис:

```
quad (func, a, b, …)
```

В **quad()** реализован численный метод из Fortran-библиотеки QUADPACK (http://nines.cs.kuleuven.be/software/QUADPACK/). Функция принимает как минимум три аргумента:

- *func* — имя подынтегральной функции. Функция должна быть определена по правилам языка Python (см. параграф 1.8). Если функция зависит от более чем одного аргумента, то интегрирование ведётся по первому аргументу;
- *a* — нижний предел интегрирования (число типа **float**);
- *b* — верхний предел интегрирования (число типа **float**).

Список и значение дополнительных аргументов можно получить с помощью команды **help(quad)**.

Например, вычислим интеграл

$$\int_0^4 3x^2 dx.$$

Точное значение этого интеграла: 64. Реализация в интерактивной оболочке IPython:

```
In  [1]: from scipy import integrate
In  [2]: def f(x):
…:               return 3.0 * x**2
In  [3]: integrate.quad(f, 0.0, 4.0)
Out [3]: (64.0, 7.105427357601002e-13)
```

Функция **quad()** возвращает кортеж из двух элементов, первый — значение интеграла, второй — точность вычисления. В примере выше ответ совпадает с точным значением и получен с точностью до 13-го знака после запятой.

Рассмотрим пример интеграла с пределом, равным бесконечности:

$$\int_0^\infty e^{-x} dx.$$

Точное значение этого интеграла: 1. Для реализации численного решения интеграла воспользуемся универсальной функцией **exp()** и зарезервированной константой **inf** (бесконечность) из библиотеки **NumPy**.

```
In  [4]: import numpy as np
In  [5]: def f2(x):
…:                 return np.exp(-x)
In  [6]: integrate.quad(f2, 0.0, np.inf)
Out [6]: (1.0000000000000002, 5.842607038578007e-11)
```

Кроме **quad()**, в модуле **scipy.integrate** доступны также другие методы нахождения интегралов: **trapz()** (метод трапеций), **sims()** (метод Симпсона), **romb()** (метод Ромберга). В отличие от **quad()**, в функциях **trapz()**, **simps()** и **romb()** сетка точек интегрирования задаётся пользователем, а не устанавливается автоматически.

4.3. Решение обыкновенных дифференциальных уравнений

Помимо функций для численного интегрирования, в модуле **scipy.integrate** содержатся функции для решения ОДУ первого порядка вида

$$\frac{d\vec{y}}{dx} = \vec{f}(\vec{y}, x),$$

$$\vec{y}(x_0) = \vec{y}_0,$$

где $\vec{y} = [y^0, y^1, ..., y^{n-1}]$ — список неизвестных функций (n штук), $\vec{f} = [f^0, f^1, ..., f^{n-1}]$ — список значений производных функций (правые части уравнений — заданные функции), $\vec{y}_0 = [y_0^0, y_0^1, ..., y_0^{n-1}]$ – список начальных условий.

Рассмотрим функцию **odeint()**. В ней реализован метод LSODA из Fortran-библиотеки ODEPACK (https://computing.llnl.gov/casc/odepack/). Синтаксис функции:

```
odeint(func, y0, x, …)
```

где

- *func* — список функций правых частей уравнений;
- *y0* — список начальных условий;
- *x* — массив точек, в которых необходимо найти значение искомой функции $\vec{y}(x)$. Первый элемент массива должен совпадать с начальной точкой интегрирования x_0;
- многоточием обозначены необязательные аргументы (см. **help(odeint)**).

Функция **odeint()** возвращает двумерный массив типа **ndarray** значений функции *y* во всех заданных точках *x*. Начальное значение *y0* хранится в первой строке. Массив решения имеет следующую форму (**shape**): (**len**(*x*), **len**(*y0*)). То есть число строк равно количеству точек интегрирования, число столбцов — количеству уравнений. Например, в случае интегрирования одного уравнения результатом работы функции **odeint()** будет двумерный массив, содержащий один столбец, в котором хранятся значения искомой функции, начиная с *y0*.

Пример решения ОДУ первого порядка

Для примера решим следующее дифференциальное уравнение:

$$\frac{dy}{dx} = -2y$$

с начальным условием $y(0) = 1$.

```
In  [1]: from scipy import integrate
In  [2]: import numpy as np
In  [3]: def f(y, x):
…:           return -2.0 * y
In  [4]: xi = np.linspace(0, 1, 10)
In  [5]: y0 = 1.0
In  [6]: sol = integrate.odeint(f, y0, xi)
```

```
In  [7]: sol
Out [7]:
array([[ 1.        ],
       [ 0.80073742],
       [ 0.64118042],
       [ 0.51341714],
       [ 0.41111231],
       [ 0.329193  ],
       [ 0.26359714],
       [ 0.21107209],
       [ 0.16901331],
       [ 0.13533527]])
In  [8]: sol[0]
Out [8]: array([ 1.])
In  [9]: sol[9]
Out [9]: array([ 0.13533527])
```

Пример решения системы двух ОДУ

Рассмотрим уравнение гармонических колебаний:

$$\frac{d^2x}{dt^2} + \omega^2 x = 0.$$

с начальными условиями $x(0) = x_0 = -5$, $dx/dt(0) = v_0 = 0$ при $\omega = 1.5$.

Сведём данное уравнение второго порядка к системе двух уравнений первого порядка с помощью замены

$$\frac{dx}{dt} = v, \frac{dv}{dt} = \frac{d^2x}{dt^2}.$$

Тогда получим

$$\frac{dv}{dt} = -\omega^2 x,$$

$$\frac{dx}{dt} = v.$$

С начальными условиями $x(0) = x_0$, $v(0) = v_0$.

```
# подключение модуля integrate из библиотеки scipy
from scipy import integrate
# подключение библиотеки numpy под псевдонимом np
import numpy as np
# подключение модуля pyplot из библиотеки matplotlib
# под псевдонимом plt
from matplotlib import pyplot as plt


# частота гармонического осциллятора
w = 1.5
# вектор-функция правых частей уравнений:
# f = [f0, f1], где f0 = -w^2*x, f1 = v.
# полагается, что f зависит от (y, t), причём y - это
# список из двух чисел:
# y = [v, x]
def f(y, t):
    v = y[0]
    x = y[1]

    f0 = -w**2 * x
    f1 = v
    return [f0, f1]


# массив точек интегрирования
ti = np.linspace(0, 10, 50)
# начальная координата
x0 = -5.0
# начальная скорость
v0 = 0.0
# список начальных условий
y0 = [x0, v0]
```

```
# решение ОДУ
sol = integrate.odeint(f, y0, ti)

# рисунок для построения графиков функций x(t) и v(t)
fig = plt.figure()
# панель для рисования графика координаты
ax1 = fig.add_subplot(121)
# панель для рисования графика скорости
ax2 = fig.add_subplot(122)

ax1.set_xlabel(r'$t$')
ax1.set_ylabel(r'$x$')
ax1.set_title('координата')
# график x(t) - зависимость нулевого столбца вектора
# решения sol от ti
ax1.plot(ti, sol[:, 0])

ax2.set_xlabel(r'$t$')
ax2.set_ylabel(r'$v$')
ax2.set_title('скорость')
# график v(t) - зависимость первого столбца вектора
# решения sol от ti
ax2.plot(ti, sol[:, 1])

# настройка оптимального расположения панелей
plt.tight_layout()
fig.savefig("ode2.png")
```

На рис. 10 графически показано решение рассматриваемого ОДУ.

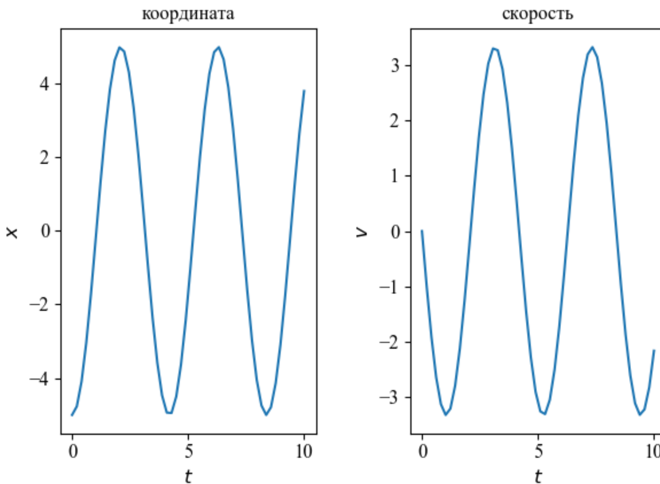

Рис. 10. *Решение ОДУ второго порядка (гармонический осциллятор)*

4.4. Интерполяция

В модуле **scipy.interpolate** содержится набор инструментов для интерполяции одномерных и многомерных данных различными методами. Рассмотрим простейший пример интерполяции одномерного набора данных с помощью функции **interp1d()**. Синтаксис вызова функции:

```
fi = interp1d(x, f0, 'method', …)
```

Функция **interp1d()** принимает в качестве первых двух аргументов одномерные массивы координат *x* и соответствующих значений некоторой величины *f0*. Функция возвращает класс, реализующий функцию, интерполирующую исходные данные заданным методом. В **interp1d()** реализовано несколько способов интерполяции: линейная (*'method'='linear'*), квадратичная (*'method'='qudratic'*), кубическая (*'method'='cubic'*) и др. По умолчанию, когда параметр *'method'* не указан, используется линейная интерполяция.

Ниже приведён пример программы с использованием **interp1d()**

```
# -*- coding: utf-8 -*-
"""
Created on Fri Oct 11 00:00:00 2018
Интерполяция одномерных данных
@author: С.А. Хайбрахманов
"""
#подключение библиотек
import matplotlib.pyplot as plt
import numpy as np
from scipy.interpolate import interp1d

# функция для генерации данных,
# по которым будет строиться интерполяция
def f_exact(x):
    return np.sin(x)**2

# массив координат, по которым будет делаться интерполяция
x = np.linspace(0, 2.0 * np.pi, 10)
# массив значений, по которым будет делаться интерполяция
y = f_exact(x)

# интерполяция данных методом по умолчанию
# (линейная интерполяция)
fi_1 = interp1d(x, y)
# кубическая интерполяция
fi_2 = interp1d(x, y, 'cubic')

# массив координат для построения
# графика интерполированных функций
xi = np.linspace(0, 2.0 * np.pi, 25)
# массивы значений интерполированых функций
# для построения графиков
yi_1 = fi_1(xi)
```

```
yi_2 = fi_2(xi)

# настройки осей
plt.xlabel('x')
plt.ylabel('y')
plt.ylim(0.0, 1.5)

# рисование графиков
plt.plot(x, y, 'o', label='исходные данные')
plt.plot(xi, f_exact(xi), '-', color='grey', linewidth=2.5,
label='точная функция')
plt.plot(xi, yi_1, '--', label='линейная интерполяция')
plt.plot(xi, yi_2, 'k:', label='кубическая интерполяция')

# легенда, располагающаяся сверху по центру панели
plt.legend(loc='upper center')

plt.show()
```

На рис. 11 показан результат работы программы.

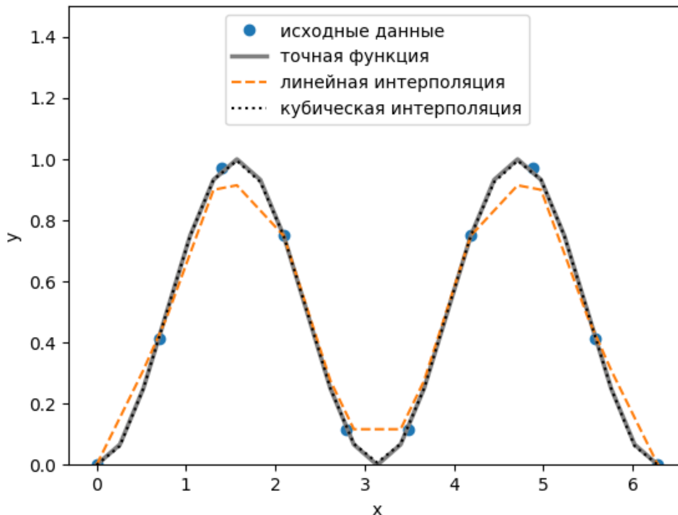

Рис. 11. *Пример интерполяции с помощью* **interp1d()**

Варианты заданий для самостоятельной работы

1. Напишите программу для численного вычисления определённого интеграла с помощью средств **SciPy**. Сравните результаты вычисления различными функциями интегрирования с точным решением.

1) $\int \frac{x^3}{x^8 - 2} dx$, от 0 до 1;
2) $\int \frac{dx}{x\sqrt{x^2 + 1}}$, от 0.1 до 1;
3) $\int \frac{dx}{x\sqrt{1 - x}}$, от 0.01 до 0.99;
4) $\int \frac{dx}{\sin x}$, от 0.01 до 0.5;
5) $\int \frac{dx}{\cos x}$, от 0 до $\frac{\pi}{4}$;
6) $\int \frac{x^2 - 1}{x^4 + 1} dx$, от –0.9 до 0.9;
7) $\int x^3 \sqrt[3]{1 + x^2} dx$, от –5 до 5;
8) $\int \frac{dx}{x^2 + x - 2}$, от –1.9 до 0.9;
9) $\int \frac{xdx}{x^4 + 3x^2 + 2}$, от –5 до 5;
10) $\int \sin(x)\sin(3x)dx$, от 0 до π;
11) $\int \tan(\frac{x}{2})dx$, от 0 до $\frac{\pi}{2}$;
12) $\int \frac{x + 1}{\sqrt{3 - x^2}} dx$, от $-\frac{\pi}{4}$ до $\frac{\pi}{2}$;
13) $\int \frac{\cos(x)dx}{\sin(x) + 2}$, от $-\pi$ до π;
14) $\int x^2 \exp(-x)dx$, от –1 до 1;
15) $\int \frac{x}{\sqrt{3 + x^2}} dx$, от –2 до 5.

2. Напишите программу для численного решения обыкновенного дифференциального уравнения с заданным начальным условием с помощью средств **SciPy**. Сравните результат вычисления с точным решением. Нарисуйте графики точного и численного решений, а также относительной ошибки численного решения.

1) $\ddot{y} + y = \frac{1}{(\sin x)^3}$ при $y\left(\frac{\pi}{4}\right) = 2$, $\dot{y}\left(\frac{\pi}{4}\right) = 1$;
2) $\ddot{y} + y = \cosh x$ при $y(0) = \pi$, $\dot{y}(0) = 1$;
3) $\ddot{y} + y = \sin x - \cos 2x$ при $y(0) = \pi$, $\dot{y}(0) = 2$;

4) $x^2\ddot{y} - 4x\dot{y} + 6y = 2$ при $y(1) = 1,\ \dot{y}(1) = -1;$

5) $\ddot{y} + 2(\dot{y})^2 = 0$ при $y(10) = 1,\ \dot{y}(10) = 1;$

6) $\ddot{y} + 4y = 0$ при $y(0) = 1,\ \dot{y}(0) = 1;$

7) $\ddot{y} + y = 1 - \dfrac{1}{\sin x}$ при $y(\pi/2) = 0,\ \dot{y}(\pi/2) = 1;$

8) $\ddot{y} - 3y = 0$ при $y(0) = 1,\ \dot{y}(0) = 0;$

9) $\ddot{y} + y = x$ при $y(0) = 1,\ \dot{y}(0) = 0;$

10) $\dot{y} = \exp(-y)\sin(x)$ при $y(0) = 0;$

11) $x^2\dot{y} = \dfrac{1}{\cos(y)}$ при $y(1) = 1;$

12) $10\dot{y} + 25y = 0$ при $y(0) = 0;$

13) $2\dot{y} + y = e^{2x}$ при $y(0) = 1;$

14) $3\dot{y} = 2 - x^2$ при $y(0) = 1;$

15) $\dot{y} - 2y = \cos(x) - 3\sin(x)$ при $y(0) = 1.$

Использованные обозначения: $\dot{y} = \dfrac{dy}{dx},\ \ddot{y} = \dfrac{d\dot{y}}{dx}.$

Вопросы для самоконтроля

1. Результатом работы функции **integrate.quad()** является кортеж из двух чисел. Что это за числа?

2. В каком модуле **SciPy** содержатся функции для решения обыкновенных дифференциальных уравнений?

3. Для решения систем ОДУ какого вида предназначена функция **odeint()**?

4. Воспользуйтесь функцией **help()** и выясните, какие функции помимо **odeint()** имеются в модуле **integrate** для решения обыкновенных дифференциальных уравнений.

5. Допустим, что g и f — одномерные массивы одинакового размера. Поясните результат работы команды

```
G = interp1d(g, f, 'cubic')
```

Какой тип имеет величина G?

Рекомендуемая литература

1. Нуньес-Иглесиас, Х. Элегантный SciPy / Х. Нуньес-Иглесиас, Ш. ван дер Уолт, Х. Дэшноу. — М. : ДМК Пресс, 2018. — 266 с.

2. The SciPy community. SciPy Tutorial [Электронный ресурс] / The SciPy community. — 2008–2019. — URL: https://docs.scipy.org/doc/scipy/reference/tutorial/index.html (дата обращения: 16.10.2019)

Заключение

В настоящее время существует большое количество как коммерческих, так и бесплатных программных пакетов для проведения научных расчётов, и анализа данных. Данное пособие посвящено одному из таких инструментов: библиотекам NumPy, Matplotlib и SciPy для языка программирования Python. Данные инструменты являются бесплатными, кроссплатформенными, имеют широкий спектр возможностей, хорошо документированы, что обусловливает удобство их применения. На данный момент Python является распространённым инструментом для решения вычислительных задач и анализа данных в различных областях науки и техники.

Следует отметить, что возможности рассмотренных пакетов ограничены лишь некоторыми стандартными математическими задачами, такими как задачи линейной алгебры, вычисление интегралов, решение алгебраических уравнений и систем ОДУ первого порядка. Решение научных задач зачастую сопряжено с необходимостью рассмотрения систем дифференциальных уравнений в частных производных и интегро-дифференциальных уравнений. Например, моделирование магнитогазодинамических процессов в плазме на Земле и в космосе основано на решении системы нелинейных уравнений магнитной газодинамики в областях со сложной геометрией. Для подобных задач требуется разработка специализированных численных алгоритмов, программных комплексов и кодов.

В пособии изложены лишь базовые возможности Python и библиотек NumPy, Matplotlib и SciPy. Оно может быть использовано для первоначального знакомства с языком Python и данными библиотеками с целью дальнейшего более глубокого самостоятельного изучения их возможностей, таких, например, как написание классов, обработка исключений и др. Помимо продемонстрированных в пособии возможностей вычисления интегралов, решения ОДУ и интерполяции данных, библиотека SciPy обладает ин-

струментами для решения задач линейной алгебры, аппроксимации данных, Фурье-анализа, обработки сигналов, статистических расчётов и т. д.

Удобство применения Python для научных расчётов связано, в частности, с возможностью интерактивного режима работы, когда вводимые команды сразу же исполняются интерпретатором. Интерактивность позволяет быстро производить отдельные виды вычислений и визуализировать результаты расчётов без необходимости тратить время на написание отдельных программ. В пособии рассматривается интерактивный режим работы в оболочке IPython. В настоящее время данный подход развивается в проекте Jupiter (URL: https://jupyter.org/). Jupiter представляет собой интерактивную оболочку с веб-интерфейсом для разработки программ. Веб-приложение Jupiter Notebook позволяет создавать документы, содержащие интерактивный программный код, уравнения, рисунки и обычный текст. Имеется возможность совместной работы над такими документами. Блокноты Jupiter могут использоваться для дальнейшего изучения возможностей Python для научных расчётов.

Список литературы

1. Walt van der, S. The NumPy Array: A Structure for Efficient Numerical Computation / S. van der Walt, S. C. Colbert and Gaël Varo // Computing in Science & Engineering. — 2011. — Vol. 13. — P. 22–30.

2. Millman, K. J. Python for Scientists and Engineers / K. J. Millman, M. Aivazis // Computing in Science & Engineering. — 2011. — Vol. 13. — P. 9–12.

3. Jones, E. SciPy: Open Source Scientific Tools for Python / E. Jones, E. Oliphant, P. Peterson, et al. — 2001. — URL: http://www.scipy.org/ [Online; доступ 11-10-2019].

4. Oliphant, T. E. Python for Scientific Computing / T. E. Oliphant // Computing in Science & Engineering. — 2007. — Vol. 9. — P. 10–20.

5. Hunter, J. D. Matplotlib: A 2D Graphics Environment / J. D. Hunter // Computing in Science & Engineering. — 2007. — Vol. 9. — P. 90–95.

6. Pérez, F. IPython: A System for Interactive Scientific Computing / F. Pérez, B. E. Granger // Computing in Science & Engineering. — 2007. — Vol. 9. — P. 21–29.

Учебное издание

ХАЙБРАХМАНОВ Сергей Александрович

ОСНОВЫ НАУЧНЫХ РАСЧЁТОВ НА ЯЗЫКЕ ПРОГРАММИРОВАНИЯ PYTHON

Учебное пособие

Корректура и вёрстка *М. В. Трифоновой*
Дизайн обложки *Ольги Харитоновой*

Подписано в печать 04.12.19.
Формат 60×84 1/16. Бумага офсетная.
Усл. печ. л. 5,7. Уч.-изд. л. 5,0.
Тираж 100 экз. Заказ 451
Цена договорная

Челябинский государственный университет
454001 Челябинск, ул. Братьев Кашириных, 129

Отпечатано в издательстве
Челябинского государственного университета
454021 Челябинск, ул. Молодогвардейцев, 57б