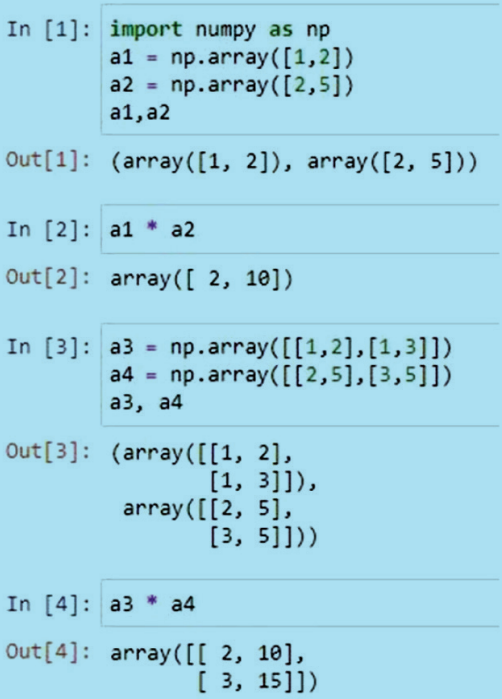
тольяттинский
государственный
университет

В.С. Климов, С.В. Мкртычев

СОВРЕМЕННЫЕ ТЕХНОЛОГИИ АНАЛИЗА ДАННЫХ

Учебно-методическое
пособие



Тольятти
Издательство ТГУ
2025

Министерство науки и высшего образования
Российской Федерации
Тольяттинский государственный университет

В.С. Климов, С.В. Мкртычев

СОВРЕМЕННЫЕ ТЕХНОЛОГИИ АНАЛИЗА ДАННЫХ

Учебно-методическое пособие

Тольятти
Издательство ТГУ
2025

УДК 004.635(075.8)
ББК 16.35я73+32.973.2я73
К492

Рецензенты:
д-р техн. наук, доцент, профессор кафедры «Информатика и программное обеспечение», заместитель первого проректора по учебной работе Брянского государственного технического университета *А.А. Захарова*;
канд. пед. наук, доцент, заведующий кафедрой «Прикладная математика и информатика» Тольяттинского государственного университета *О.М. Гущина*.

К492 Климов, В.С. Современные технологии анализа данных : учебно-методическое пособие / В.С. Климов, С.В. Мкртычев. – Тольятти : Издательство ТГУ, 2025. – 166 с. – ISBN 978-5-8259-1708-5.

В пособии содержатся основные сведения о современных методах работы с данными. Приведена информация об использовании Python для обработки данных, библиотек numpy и pandas для организации хранения и управления данными и библиотеки matplotlib для визуализации данных.

Предназначено для студентов, обучающихся по направлениям подготовки бакалавров 09.03.03 «Прикладная информатика», 01.03.02 «Прикладная математика и информатика», 02.03.03 «Математическое обеспечение и администрирование информационных систем» очной и заочной форм обучения.

УДК 004.635(075.8)
ББК 16.35я73+32.973.2я73

Рекомендовано к изданию научно-методическим советом Тольяттинского государственного университета.

ISBN 978-5-8259-1708-5

ВВЕДЕНИЕ

Материал, представленный в пособии, предназначен для ознакомления с современными технологиями анализа данных: организацией сбора и хранения данных; выбором данных по определенным критериям, содержащим несколько условий; модификацией данных; обменом данными между различными приложениями; интеграцией данных, полученных из различных источников.

Технологии анализа данных применяются при решении широкого круга задач, в том числе и при разработке новых программных продуктов.

Данное пособие поможет вам научиться использовать язык программирования Python 3 для решения практических задач по анализу данных.

В пособии рассмотрены библиотеки numpy и pandas для организации хранения и анализа данных, а также библиотека matplotlib для визуализации данных.

Логически пособие поделено на 3 части. Первая часть посвящена основам программирования на языке Python. Во второй части рассматриваются базовые методы управления данными с использованием возможностей языка Python. В третьей части исследуются инструменты, предоставляемые библиотеками Python для управления, организации хранения и визуализации данных.

1. ОСНОВЫ ЯЗЫКА PYTHON

1.1. Ввод-вывод данных и применение арифметических операций

Ссылки на интерпретатор и среду программирования

Скачайте и установите необходимую версию Miniconda на ПК: https://docs.conda.io/en/latest/miniconda.html.

Для установки интерактивной среды разработки JupyterLab необходимо в командной строке Miniconda выполнить команду «conda install -c conda-forge jupyterlab» (рис. 1).

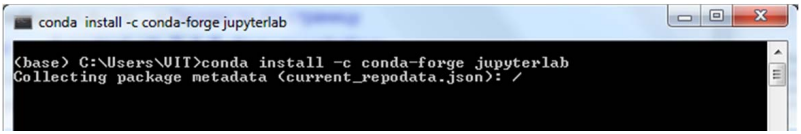


Рис. 1. Установка среды разработки jupyterlab

Если при попытке выполнения программного кода возникла ошибка, а именно отсутствие какой-либо библиотеки, то ее необходимо установить с использованием командной строки Miniconda. Например, библиотека Matplotlib устанавливается командой: «conda install -c conda-forge matplotlib».

Запуск среды разработки jupyterlab осуществляется командой «jupyterlab» (рис. 2).

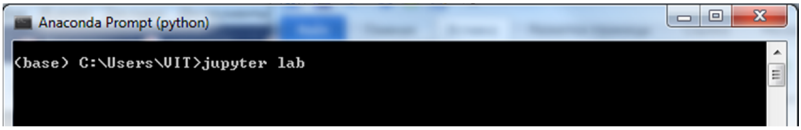


Рис. 2. Запуск jupyterlab

Внешний вид среды разработки jupyterlab представлен ниже (рис. 3).

Программный код в jupyterlab пишется с разделением по блокам. Деление на блоки реализовано для удобства и не влияет на результат выполнения программы (рис. 4).

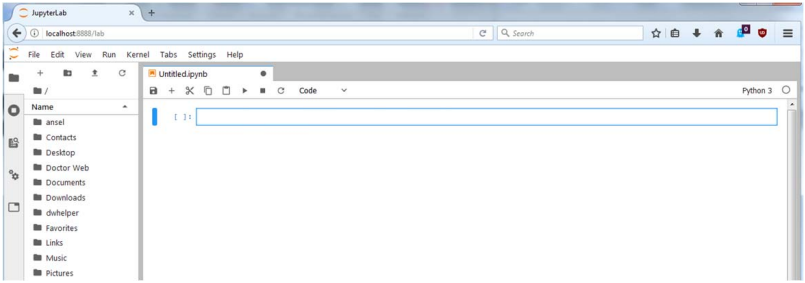


Рис. 3. Внешний вид среды разработки jupyterlab

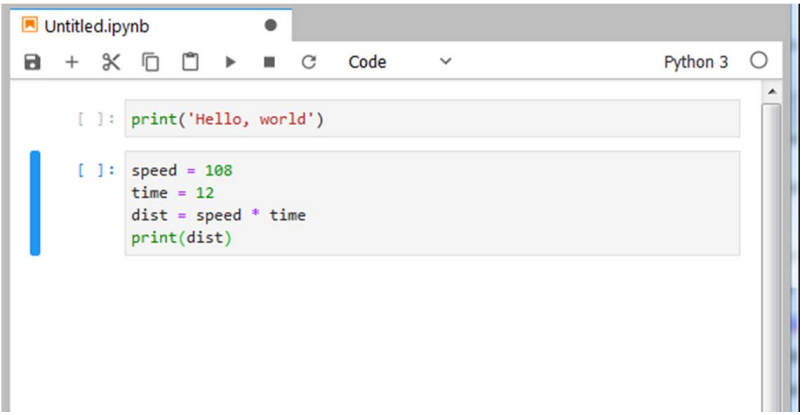


Рис. 4. Программный код в jupyterlab

Запуск на выполнение осуществляется с помощью пункта меню «Run all cells». Результат выполнения данного программного кода представлен ниже (рис. 5).

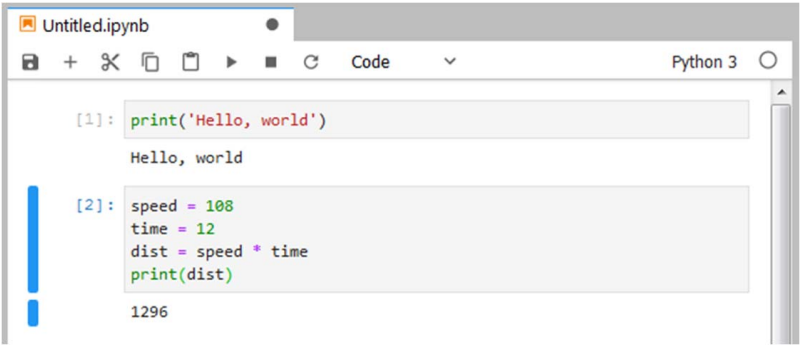


Рис. 5. Результат выполнения программного кода в jupyterlab

Язык Python

Язык Python (по-русски можно произносить Пайтон или Питон) появился в 1991 году и был разработан нидерландским программистом Гвидо ван Россумом. Язык назван в честь шоу «Летающий цирк Монти Пайтона». Одна из главных целей разработчиков – сделать язык забавным для использования.

Сейчас Питон регулярно входит в десятку наиболее популярных языков и хорошо подходит для решения широкого класса задач: обучение программированию, создание скриптов для обработки данных, машинное обучение, серверная веб-разработка и многое другое. Большинству из обучающихся программированию язык Питон потребуется для написания проектов и при изучении ряда предметов по программированию на языке Python, а также в повседневной деятельности для автоматизации задач обработки данных.

Мы будем изучать язык Питон третьей версии.

Типы данных и функции вывода

Программы на языке Питон представляют собой обычные текстовые файлы, в которых записана последовательность команд. Код легко читается и интуитивно понятен.

Например, выводящая «Hello, world!» программа записывается всего в одну строку: вызывается функция печати print, которой в качестве параметра передается строка, содержащая в себе фразу «Hello, world!». Если мы хотим задать какую-то строку, то должны обрамлять её одинарными (‘) или двойными («) кавычками, иначе она будет интерпретироваться как код на языке Питон (рис. 6).

```
1  print('Hello, world!')
```

Рис. 6. Использование функции вывода. Пример 1

Кроме строк, рассмотрим целочисленный тип данных. Например, можно посчитать результат вычисления арифметического выражения 2 + 3 и вывести его с помощью такой однострочной программы на языке Питон (рис. 7).

```
1  print(2 + 3)
```

Рис. 7. Использование функции вывода. Пример 2

Такая программа выведет результат вычисления выражения, равный 5. Если бы числа 2 и 3 были заключены в кавычки, то они интерпретировались бы как строки, а операция «+» проводила бы конкатенацию (склеивание) строк. Например, такой код выведет 23 – строку, состоящую из склеенных символов '2' и '3' (рис. 8).

```
1  print('2' + '3')
```

Рис. 8. Использование функции вывода. Пример 3

Функция print может принимать и несколько параметров, тогда они будут выводиться через пробел, причем параметры могут иметь различные типы. Если мы хотим получить вывод вида 2 + 3 = 5, то можем воспользоваться программой, представленной на рис. 9.

```
1  print('2 + 3 =', 2 + 3)
```

Рис. 9. Использование функции вывода. Пример 4

Обратите внимание, что в строке '2 + 3 =' нет пробела после знака =. Пробел появляется автоматически между параметрами функции print. Что же делать, если хочется вывести строку вида 2+3=5 (без пробелов)? Для этого понадобится именованный параметр sep (separator, разделитель) для функции print. Та строка, которая передается в качестве параметра sep, будет подставляться вместо пробела в качестве разделителя. В этой задаче мы будем использовать пустую строку в качестве разделителя. Пустая строка задается двумя подряд идущими кавычками (рис. 10).

```
1  print('2+3=', 2 + 3, sep='')
```

Рис. 10. Использование функции вывода с использованием параметра sep

В качестве параметра sep можно использовать любую строку, в том числе состоящую из нескольких символов. Если нам нужно сделать несколько разных разделителей для разных частей строк, то не остается другого выбора, кроме как использовать несколько подряд идущих функций print. Например, если мы хотим вывести строку вида 1 + 2 + 3 + 4 = 10, то можем попробовать воспользоваться кодом, представленным на рис. 11.

```
1  print(1, 2, 3, 4, sep=' + ')
2  print(' = ', 1 + 2 + 3 + 4, sep='')
```

Рис. 11. Функция вывода с использованием параметра sep

Однако вывод такого кода представляется некорректным. Он будет выглядеть так:

```
1 + 2 + 3 + 4
= 10
```

Это связано с тем, что после каждой функции print по умолчанию осуществляется перевод строки. Для изменения того, что будет печататься после вывода всего, что есть в функции print, можно использовать именованный параметр end. В нашем случае после первого print мы не хотели бы печатать ничего. Правильный код представлен на рис. 12.

```
1  print(1, 2, 3, 4, sep=' + ', end='')
2  print(' = ', 1 + 2 + 3 + 4, sep='')
```

Рис. 12. Функция вывода с использованием параметров sep и end

В качестве end также можно использовать абсолютно любую строку.

Переменные и арифметические выражения

Переменные

В некоторых задачах удобно проводить вычисления, используя вспомогательные переменные. Например, в школьных формулах по физике было удобно вычислять не гигантское выражение целиком, а запоминая результаты вычисления во вспомогательных переменных. Для примера решим задачу вычисления пройденного расстояния по известному времени и скорости (рис. 13).

```
1  speed = 108
2  time = 12
3  dist = speed * time
4  print(dist)
```

Рис. 13. Использование переменных

В этой программе мы создаем три переменные: speed для скорости, time для времени и dist для вычисленного расстояния. При использовании переменных в арифметическом выражении просто используется значение, которое лежит в переменной.

Для присваивания значения переменной используется знак =. Имя переменной должно быть записано слева от знака присваивания, а арифметическое выражение (в котором могут быть использованы числа и другие уже заданные переменные) – справа. Имя переменной должно начинаться с маленькой латинской буквы, должно быть осмысленным (английские слова или общеупотребимые сокращения) и не должно превышать по длине 10–15 символов. Если логичное имя переменной состоит из нескольких слов, то нужно записывать его с помощью camelTyping (каждое новое слово, кроме первого, должно быть записано с большой буквы).

То, как осуществляется присваивание, подробнее будет описано ниже.

Арифметические выражения

Мы уже использовали в наших программах арифметические выражения, в частности операции + и *. Существует ряд других арифметических операций. Они приведены в табл. 1.

Таблица 1

Арифметические выражения

Знак	Операция	Операнд 1	Операнд 2	Результат
+	Сложение	11	6	17
–	Вычитание	11	6	5
*	Умножение	11	6	66
//	Целочисленное деление	11	6	1
%	Остаток от деления	11	6	5
**	Возведение в степень	2	3	8

Все операции инфиксные (записываются между операндами). Например, для возведения 2 в степень 3 нужно писать 2**3.

Особо остановимся на операциях вычисления целой части и остатка от деления числа.

Пусть заданы два числа A и B, причем $B > 0$. Обозначим как C целую часть от деления A на B: $C = A // B$. Обозначим как D остаток от деления A на B: $D = A \% B$.

Тогда должны выполняться следующие утверждения:

$$A = B \times C + D,$$
$$0 \le D < B.$$

Эти утверждения необходимы для понимания процесса взятия остатка от деления отрицательного числа на положительное. Нетрудно убедиться, что если –5 разделить на 2, то целая часть должна быть равна –3, а остаток равен 1. В некоторых других языках программирования остатки в такой ситуации могут быть отрицательными, что неправильно по математическим определениям.

Если $B < 0$, выполняются следующие утверждения:

$$A = B \times C + D,$$
$$B < D \le 0.$$

Например, при делении 11 на –5 мы получим целую часть, равную –3, а остаток будет равен –4.

Если же разделить –11 на –5, то целая часть будет равна 2, а остаток будет равен –1.

Обратите внимание, что целые числа в Питоне не имеют ограничений на длину (кроме объема доступной памяти).

Операции над строками

Строки также можно сохранять в переменные и использовать в некотором ограниченном количестве выражений. В частности, можно склеивать две строки с помощью операции + (рис. 14).

```
1  goodByePhrase = 'Hasta la vista'
2  person = 'baby'
3  print(goodByePhrase + ', ' + person + '!')
```

Рис. 14. Использование операций над строками. Пример 1

Складывать число со строкой (и наоборот) нельзя. Но можно воспользоваться функцией str, которая по числу генерирует строку. Str – это сокращение от слова «string», которое можно перевести на русский как «строка, которая представляет собой последовательность символов». Например, задачу про вывод 2 + 3 = 5 можно решить и способом, представленным на рис. 15.

```
1  answer = '2 + 3 = ' + str(2 + 3)
2  print(answer)
```

Рис. 15. Использование операций над строками. Пример 2

Чтение данных

Можно умножить строку на целое неотрицательное число, в результате получится исходная строка, повторенная заданное число раз (рис. 16).

```
1  word = 'Bye'
2  phrase = word * 3 + '!'
3  print(phrase)
```

Рис. 16. Использование операций над строками. Пример 3

Программы, которые умеют только писать, но не умеют читать, редко представляют интерес для пользователей. Узнавать что-то из внешнего мира наши программы будут с помощью функции input(). Эта функция считывает строку из консоли. Чтобы закончить ввод строки, нужно нажать Enter. Под строкой в данном случае понимается английское слово «line», что означает «строка, оканчивающаяся переводом строки». Если в такую программу ввести слово Python, то она напечатает «I love Python» (рис. 17).

```
1  name = input()
2  print('I love', name)
```

Рис. 17. Пример 1 использования функции input()

Во многих задачах нам требуется работать со введенными числами, а читать мы умеем только строки. Чтобы преобразовать строку, состоящую из цифр (и, возможно, знака «–» перед ними), в целое число, можно воспользоваться функцией int (сокращение от английского «integer» – «целое число»). Например, решение задачи о сложении двух чисел будет выглядеть, как на рис. 18.

```
1  a = int(input())
2  b = int(input())
3  print(a + b)
```

Рис. 18. Пример 2 использования функции input()

Функция int может быть применена не только к результату, возвращаемому функцией input, но и к произвольной строке.

В строках могут быть не только буквы, цифры и знаки препинания, но и, например, символы табуляции и перевода строки. Чтобы использовать эти символы в константной строке в коде программы, необходимо записывать их как \t и \n соответственно. Использование бэкслеша перед символом называется экранированием. Существуют и другие символы, которые требуют бэкслеша перед собой. Это, например, кавычки \' и \" (использование бэкслеша просто необходимо, если в строке используются оба типа кавычек), а также собственно символ бэкслеша, который надо записывать как \\.

В случае считывания с помощью input символы в консоли экранировать не нужно.

Примеры решения задач

Рассмотрим несколько задач, решаемых с помощью арифметических операций, которые показывают некоторые способы.

Задача 1

Пусть есть два товара, первый из них стоит A рублей B копеек, а второй – C рублей D копеек. Сколько рублей и копеек стоят эти товары вместе?

В задачах с несколькими размерностями величин (например, рубли и копейки, километры и метры, часы и минуты) следует переводить значения в наименьшую единицу измерения, осуществлять необходимые действия, а затем переводить обратно к нужным единицам.

В нашей задаче наименьшей единицей являются копейки, поэтому все цены следует перевести в копейки, сложить их, а затем перевести результат обратно в рубли и копейки. Код решения будет выглядеть, как на рис. 19.

```
1  a = int(input())
2  b = int(input())
3  c = int(input())
4  d = int(input())
5  cost1 = a * 100 + b
6  cost2 = c * 100 + d
7  totalCost = cost1 + cost2
8  print(totalCost // 100, totalCost % 100)
```

Рис. 19. Пример решения задачи 1

Для определения количества рублей нужно разделить цену в копейках на 100 нацело, а для определения оставшегося числа копеек – посчитать остаток от деления на 100.

Задача 2

Вася играет в Super Mario Bros очень хорошо. Он получил N дополнительных жизней. Известно, что переменная, в которой хранится количество жизней, может принимать значения от 0 до 255.

Если было 255 жизней и игрок получил дополнительную жизнь, счетчик обнуляется. Сколько жизней на счетчике?

В этой задаче достаточно посчитать остаток от деления введенного числа на 256. Такие действия часто требуются, например, при работе со временем (при переходе через сутки счетчик времени обнуляется). Решение задачи выглядит, как на рис. 20.

```
1  n = int(input())
2  print(n % 256)
```

Рис. 20. Пример решения задачи 2

Задача 3

Вводится число N, необходимо «отрезать» от него K последних цифр. Например, при N = 123 456 и K = 3 ответ должен быть равен 123.

Для решения этой задачи нужно понять, что происходит при целочисленном делении на 10 (основание системы счисления). Если мы разделим число на 10, то будет отброшена последняя цифра, независимо от того, какой она была. Если разделим число на 100 – будет отброшено две последние цифры. Исходя из этого, получается решение задачи: необходимо просто разделить число N на 10 в степени K (рис. 21).

```
1  n = int(input())
2  k = int(input())
3  print(n // 10**k)
```

Рис. 21. Пример решения задачи 3

1.2. Логические выражения, ветвления и циклы при выполнении операций над данными

Логический тип данных и операции

Логический тип данных

Кроме уже известных нам целочисленных и строковых типов данных, в Питоне существует также логический тип данных, который может принимать значения «истина» (True) или «ложь» (False).

По аналогии с арифметическими выражениями существуют логические выражения, которые могут быть истинными или ложными. Простое логическое выражение имеет вид: <арифметическое выражение> <знак сравнения> <арифметическое выражение>. Например, если у нас есть переменные x и y с какими-то значениями, то логическое выражение x + y < 3 * y в качестве первого арифметического выражения имеет x + y, в качестве знака сравнения < (меньше), а второе арифметическое выражение в нём 3 * y.

В логических выражениях допустимы знаки сравнений, представленные в табл. 2.

Таблица 2

Логические выражения

Знак сравнения	Описание
<	Меньше
>	Больше
<=	Меньше либо равно
>=	Больше либо равно
==	Равно
!=	Не равно

В Питоне допустимы и логические выражения, содержащие несколько знаков сравнения, например x < y < z. При этом все сравнения обладают одинаковым приоритетом, который меньше, чем у любой арифметической операции.

Результат вычисления логического выражения можно сохранять в переменную, которая будет иметь тип bool. Переменные такого типа, как и числа, и строки, являются неизменяемыми объектами.

Строки также могут сравниваться между собой. При этом сравнение происходит в лексикографическом порядке (как упорядочены слова в словаре).

Логические операции

Чтобы записать сложное логическое выражение, часто бывает необходимо воспользоваться логическими связками «и», «или» и «не». В Питоне они обозначаются как and, or и not соответственно. Операции and и or являются бинарными, то есть должны быть записаны между операндами, например x < 3 or y > 2. Операция not – унарная и должна быть записана перед единственным своим операндом.

Все логические операции имеют приоритет ниже, чем операции сравнения (а значит, ниже, чем арифметические операции). Среди логических операций наивысший приоритет имеет операция not, затем идет and, наименьший приоритет имеет операция or. На порядок выполнения операций можно влиять с помощью скобок, как и в арифметических выражениях.

Примеры использования логических выражений

Одним из примеров использования логического выражения является проверка на делимость. Например, чтобы проверить, является ли число четным, необходимо сравнить остаток от деления этого числа на два с нулём (рис. 22).

```
1  isEven = number % 2 == 0
```

Рис. 22. Пример проверки на делимость

Рассмотрим задачу о пересечении двух длительных по времени событий. Оба события характеризуются двумя числами – годами их начала и окончания. Необходимо определить, пересекались ли события во времени. При этом если одно событие началось в тот год, когда закончилось другое, они считаются пересекающимися.

Первая идея заключается в том, чтобы рассмотреть все возможные варианты расположения событий и выделить следующий кри-

терий пересечения: если начало или конец одного из событий лежит между началом и концом другого, то они пересекаются. В виде программы это можно записать так, как представлено на рис. 23.

```
is1startIn2 = start2 <= start1 <= finish2
is1finishIn2 = start2 <= finish1 <= finish2
is1in2 = is1startIn2 or is1finishIn2
is2startIn1 = start1 <= start2 <= finish1
is2finishIn1 = start1 <= finish2 <= finish1
is2in1 = is2startIn1 or is2finishIn1
answer = is1in2 or is2in1
```

Рис. 23. Пример решения задачи о пересечении

Если немного подумать, то можно придумать более короткий критерий для проверки такого пересечения: необходимо, чтобы начало первого события происходило не позже конца второго и начало второго события происходило не позже конца первого (рис. 24).

```
answer = start1 <= finish2 and start2 <= finish1
```

Рис. 24. Пример использования логических выражений

Такой способ значительно проще.

Условный оператор

Наиболее частое применение логические выражения находят в условных операторах.

Условный оператор позволяет выполнять действия в зависимости от того, выполнено условие или нет. Записывается условный оператор как "if <логическое выражение>:". Далее следует блок команд, который будет выполнен, только если логическое выражение приняло значение True. Блок команд, который будет выполняться, выделяется отступами в 4 пробела (в IDE можно нажимать клавишу tab).

Рассмотрим, например, задачу о нахождении модуля числа. Если число отрицательное, то необходимо заменить его на минус это число. Решение выглядит так, как представлено на рис. 25.

Рис. 25. Использование логического оператора

В этой программе с отступом записана только одна строка, x = –x. При необходимости выполнить несколько команд все они должны быть записаны с тем же отступом. Команда print записана без отступа, поэтому она будет выполняться в любом случае, независимо от того, было ли условие в if'е истинным или нет.

В дополнение к if можно использовать оператор else: (иначе). Блок команд, который следует после него, будет выполняться, если условие было ложным. Например, ту же задачу о выводе модуля числа можно было решить, не меняя значения переменной x (рис. 26).

Рис. 26. Использование оператора else:

Все команды, которые выполняются в блоке else, должны быть также записаны с отступом. Else должен следовать сразу за блоком команд if, без промежуточных команд, выполняемых безусловно. Else без соответствующего if'а не имеет смысла.

Если после if записано не логическое выражение, то оно будет приведено к логическому, как если бы от него была вызвана функция bool. Однако злоупотреблять этим не следует, так как это ухудшает читаемость кода.

Для подсчета модуля числа в Питоне существует функция abs, которая избавляет от необходимости каждый раз писать подсчет модуля вручную.

В Питоне, как и во многих других языках программирования, если результат вычисления выражения однозначно понятен по уже вычисленной части, то оставшаяся часть выражения даже не счита-

ется. Например, выражение True or 5 // 0 == 42 не будет вызывать ошибки деления на ноль, так как по левой части выражения (True) уже понятно, что результат его вычисления также будет True, и арифметическое выражение в правой части даже не будет вычисляться.

Вложенный условный оператор и конструкция «иначе – если»

Вложенный условный оператор

Внутри блока команд могут находиться другие условные операторы. Рассмотрим на примере. По заданному количеству глаз и ног нужно научиться отличать кошку, паука, морского гребешка и жучка. У морского гребешка бывает более сотни глаз, а у большинства пауков их восемь. Также у пауков восемь ног, а у морского гребешка их нет совсем. У кошки четыре лапы, а у жучка – шесть ног, но глаз у обоих по два. Решение представлено на рис. 27.

```
1  eyes = int(input())
2  legs = int(input())
3  if eyes >= 8:
4      if legs == 8:
5          print("spider")
6      else:
7          print("scallop")
8  else:
9      if legs == 6:
10         print("bug")
11     else:
12         print("cat")
```

Рис. 27. Использование вложенного условного оператора

Если вложенных условных операторов несколько, то к какому из них относится else, можно понять по отступу. Отступ у else должен быть такой же, как у if, к которому он относится.

Конструкция «иначе – если»

В некоторых ситуациях необходимо осуществить выбор больше чем из двух вариантов, которые могут быть обработаны с помощью if-else. Рассмотрим пример: необходимо вывести словом название числа 1 или 2 или сообщить, что это другое число (рис. 28).

Здесь используется специальная конструкция elif, обозначающая «иначе, если», после которой записывается условие. Такая кон-

струкция введена в язык Питон, потому что запись if-else приведет к увеличению отступа и ухудшению читаемости.

```
1  number = int(input())
2  if number == 1:
3      print('One')
4  elif number == 2:
5      print('Two')
6  else:
7      print('Other')
```

Рис. 28. Использование конструкции if-else

Конструкций elif может быть несколько, условия проверяются последовательно. Как только условие выполнено, запускается соответствующий этому условию блок команд, и дальнейшая проверка не выполняется. Блок else является необязательным, как и в обычном if.

Цикл while

While переводится как «пока» и позволяет выполнять команды до тех пор, пока условие верно. После окончания выполнения блока команд, относящихся к while, управление возвращается на строку с условием, и если оно выполнено, то выполнение блока команд повторяется, а если не выполнено, то продолжается выполнение команд, записанных после while.

С помощью while очень легко организовать вечный цикл, поэтому необходимо следить за тем, чтобы в блоке команд происходили изменения, которые приведут к тому, что в какой-то момент условие перестанет быть истинным.

Рассмотрим несколько примеров.

Есть число N. Необходимо вывести все числа по возрастанию от 1 до N. Для решения этой задачи нужно завести счётчик (переменную i), который будет равен текущему числу. Вначале это единица. Пока значение счетчика не превысит N, необходимо выводить его текущее значение и каждый раз увеличивать его на единицу (рис. 29).

```
n = int(input())
i = 1
while i <= n:
    print(i)
    i = i + 1
```

Рис. 29. Использование цикла while. Пример 1

Еще одна часто встречающаяся задача – поиск минимума (или максимума) в последовательности чисел. Пусть задана последовательность чисел, оканчивающаяся нулём. Необходимо найти минимальное число в этой последовательности. Эта задача может быть решена человеком в уме: каждый раз, когда ему называют очередное число, он сравнивает его с текущим запомненным минимумом и при необходимости запоминает новое минимальное число. В качестве первого запомненного числа нужно взять первый элемент последовательности, который должен быть считан отдельно до цикла (рис. 30).

```
now = int(input())
nowMin = now
while now != 0:
    if now < nowMin:
        nowMin = now
    now = int(input())
print(nowMin)
```

Рис. 30. Использование цикла while. Пример 2

Инструкция для прерывания цикла называется break. После её выполнения работа цикла прекращается (как будто не было выполнено условие цикла). Осмысленное использование конструкции break возможно, только если выполнено какое-то условие, то есть break должен вызываться только внутри if (находящегося внутри цикла). Использование break – плохой тон, по возможности следует обходиться без него. Рассмотрим пример вечного цикла, выход из которого осуществляется с помощью break. Для этого решим задачу о выводе всех целых чисел от 1 до 100. Использовать break таким образом ни в коем случае не нужно, это просто пример (рис. 31).

```
i = 1
while True:
    print(i)
    i = i + 1
    if i > 100:
        break
```

Рис. 31. Использование цикла while. Пример 3

В языке Питон к циклу while можно написать блок else. Команды в этом блоке будут выполняться, если цикл завершил свою работу нормальным образом (то есть условие в какой-то момент перестало быть истинным), и не будут выполняться только в случае, если выход из цикла произошел с помощью команды break.

Подсчет суммы и оператор continue

Часто возникает задача о подсчете суммы последовательности. Для подсчета суммы чисел необходимо завести переменную, которая будет хранить накопленную на данный момент сумму, и при чтении очередного числа прибавлять его к накопленной сумме (рис. 32).

```
now = int(input())
seqSum = 0
while now != 0:
    seqSum = seqSum + now
    now = int(input())
print(seqSum)
```

Рис. 32. Использование цикла while. Пример 4

Команда continue начинает исполнение тела цикла заново, начиная с проверки условия. Её нужно использовать, если начиная с какого-то места в теле цикла и при выполнении каких-то условий дальнейшие действия нежелательны.

Приведём пример использования continue (хотя при решении этой задачи можно и нужно обходиться без него): дана последовательность чисел, оканчивающаяся нулём. Необходимо вывести все положительные числа из этой последовательности. Решение представлено на рис. 33.

```
now = -1
while now != 0:
    now = int(input())
    if now <= 0:
        continue
    print(now)
```

Рис. 33. Пример использования continue

В этом решении есть интересный момент: перед циклом переменная инициализируется заведомо подходящим значением. Команда вывода будет выполняться только в том случае, если не выполнится условие в if.

1.3. Работа с данными, содержащими вещественные числа

Как устроены вещественные числа

В отличие от целых чисел, вещественные числа в языке Питон имеют ограниченную длину.

Подумаем, как хранить десятичную дробь в памяти. Поскольку вещественных чисел бесконечно много (даже больше, чем натуральных), то нам придется ограничить точность. Например, мы можем хранить только несколько первых значащих цифр, не храня незначащие нули. Будем отдельно хранить целое число с первыми значащими цифрами и отдельно хранить степень числа 10, на которую нужно умножить это число.

Например, число 5.972*10**24 (это масса Земли в килограммах) можно сохранить как 5972 (цифры числа, мантисса) и 21 (на какую степень 10 нужно умножить число, экспонента). С помощью такого представления можно хранить вещественные числа любой размерности.

Примерно так и хранятся числа в памяти компьютера, однако вместо десятичной системы используется двоичная. На большинстве аппаратных систем в языке Питон для хранения float используется 64 бита, из которых 1 бит уходит на знак, 52 бита – на мантиссу и 11 бит – на экспоненту. Это упрощенное описание, но достаточно неплохо описывает реальность.

52 бита дают около 15–16 десятичных знаков, которые будут храниться точно. 11 бит на экспоненту также накладывает ограничения на размерность хранимых чисел (примерно от –1000 до 1000 степени числа 10).

Любое вещественное число на языке Питон представимо в виде дроби, где в числителе хранится целое число, а в знаменателе находится какая-либо степень двойки. Например, 0.125 представимо как 1/8, а 0.1 как 3602879701896397/36028797018963968. Несложно заметить, что эта дробь не равна 0.1, то есть хранение числа 0.1 точно в типе float невозможно, как и многих других «красивых» десятичных дробей.

В целом будет полезно представлять себе вещественное число X как отрезок [X – epsilon; X + epsilon]. Как же определить величину epsilon?

Для этого нужно понять, что погрешность не является абсолютной, то есть одинаковой для всех чисел, а является относительной. Упрощенно аппаратную погрешность хранения числа X можно оценить как X*2**(–54).

Чаще всего в задачах входные данные имеют определенную точность. Рассмотрим на примере. Заданы два числа X и Y с точностью 6 знаков после точки (значит, epsilon=5*10**(–7)), по модулю не превосходящие 10**9. Оценить абсолютную погрешность вычисления X * Y. Рассмотрим худший случай, когда X и Y равны 10**9 и отклонились на максимально возможное значение epsilon в одну сторону. Тогда результат вычисления будет выглядеть так:

(X + epsilon) * (Y + epsilon) = XY + (X + Y) * epsilon + epsilon**2.

Величина epsilon**2 пренебрежимо мала, XY – это правильный ответ, а (X + Y) * epsilon – искомое значение абсолютной погрешности. Подставим числа и получим:

2 * 10**9 * 5 * 10**(–7) = 10**3.

Абсолютная погрешность вычисления составила 1000 (одну тысячу). Что довольно неожиданно и грустно.

Таким образом, нужно аккуратно вычислять значение погрешности для сравнения вещественных чисел.

Основы работы с вещественными числами

Для записи констант или при вводе-выводе может использоваться как привычное представление в виде десятичной дроби, например: 123.456, так и «инженерная» запись числа, где мантисса записывается в виде вещественного числа с одной цифрой до точки и некоторым количеством цифр после точки, затем следуют буква «e» (или «E») и экспонента. Число 123.456 в инженерной записи будет выглядеть так: 1.23456e2. Это означает, что 1.23456 нужно умножить на 10**2. И мантисса, и экспонента могут быть отрицательными и записываются в десятичной системе.

Такая запись чисел может применяться при создании вещественных констант, а также при вводе и выводе. Инженерная запись удобна для хранения очень больших или очень маленьких чисел, чтобы не считать количество нулей в начале или конце числа.

Если хочется вывести число не в инженерной записи, а с фиксированным количеством знаков после точки, то следует воспользоваться методом format, который имеет массу возможностей. Нам нужен только вывод фиксированного количества знаков, поэтому воспользуемся готовым рецептом для вывода 25 знаков после десятичной точки у числа 0.1 (рис. 34).

```
1  x = 0.1
2  print('{0:.25f}'.format(x))
```

Рис. 34. Использование метода format

Вывод такой программы будет выглядеть как 0.1000000000000000055511151, что еще раз подтверждает мысль о том, что число 0.1 невозможно сохранить точно.

Проблемы вещественных чисел

Рассмотрим простой пример с вещественными числами (рис. 35).

Если запустить эту программу, можно легко убедиться в том, что 0.1 + 0.2 не равно 0.3. Хотя можно было надеяться, что несмотря на неточное представление, оно окажется одинаково неточным для всех чисел.

```
if 0.1 + 0.2 == 0.3:
    print('Yes')
else:
    print('No')
```

Рис. 35. Пример с вещественными числами

Поэтому при использовании вещественных чисел нужно следовать простым правилам:

1. Если можно обойтись без использования вещественных чисел, нужно это сделать. Вещественные числа проблемные, неточные и медленные.

2. Два вещественных числа равны между собой, если они отличаются не более чем на epsilon. Число X меньше числа Y, если $X < Y - \text{epsilon}$.

Код для сравнения двух чисел, заданных с точностью 6 знаков после точки, выглядит так, как представлено на рис. 36.

```
x = float(input())
y = float(input())
epsilon = 10 ** -6
if abs(x - y) < epsilon:
    print('Equal')
else:
    print('Not equal')
```

Рис. 36. Сравнение двух чисел

Если над числами совершались какие-то действия, то значения epsilon нужно вычислять. В учебных задачах это можно сделать не внутри программы, а один раз вручную для худшего случая, и применять вычисленное значение как константу.

Округление вещественных чисел

При использовании целых и вещественных чисел в одном выражении вычисления производятся в вещественных числах. Тем не менее иногда возникает необходимость преобразовать вещественное число в целое. Для этого можно использовать несколько видов функций округления:

– int – округляет в сторону нуля (отбрасывает дробную часть);
– round – округляет до ближайшего целого. Если ближайших целых несколько (дробная часть равно 0.5), то к чётному;
– floor – округляет в меньшую сторону;
– ceil – округляет в большую сторону.

Примеры для различных чисел приведены в табл. 3.

Таблица 3

Сравнение функций округления

Функция	2.5	3.5	–2.5
int	2	3	–2
round	2	4	–2
floor	2	3	–3
ceil	3	4	–2

Функции floor и ceil находятся в библиотеке math. Есть два способа получить возможность воспользоваться ими в своей программе.

В первом способе импортируется библиотека math, тогда перед каждым вызовом функции оттуда нужно писать слово «math.», а затем – имя функции (рис. 37).

```
1  import math
2
3  print(math.floor(-2.5))
4  print(math.ceil(-2.5))
```

Рис. 37. Использование округления. Пример 1

Во втором способе из библиотеки импортируются некоторые функции, и доступ к ним можно получить без написания «math.» (рис. 38).

```
1  from math import floor, ceil
2
3  print(floor(-2.5))
4  print(ceil(-2.5))
```

Рис. 38. Использование округления. Пример 2

Второй способ предпочтительно применять в случае, если какие-то функции используются часто и нет конфликта имен (функций с одинаковыми именами в нескольких подключенных библиотеках).

В библиотеке math также есть функция округления trunc, которая работает аналогично int.

Срезы строк

Нам известны способы считывать, выводить и задавать константные строки, а также склеивать строки между собой и умножать строку на число.

Чтобы определить длину строки s, можно воспользоваться функцией len(s). Она возвращает целое число, равное длине строки.

Срез – это способ извлечь из строки отдельные символы или подстроки. При применении среза конструируется новая строка. Строка, к которой был применён срез, остается без изменений.

Простейший вид среза – это обращение к конкретному символу строки по номеру. Чтобы получить i-й символ строки, нужно написать s[i]. В результате этого будет сконструирована строка, содержащая только один символ – тот, который стоял на месте i. Нумерация символов идет с нуля. При попытке обратиться к символу с номером, больше либо равным длине строки, возникает ошибка.

В языке Питон присутствует и нумерация символов строки отрицательными числами. Последний символ строки имеет номер –1, предпоследний –2 и так далее. При попытке обратиться к символу с номером, меньшим –len(s), возникает ошибка.

Нумерация символов в строке «String» представлена в табл. 4.

Таблица 4

Нумерация символов в строке «String»

S	t	r	i	n	g
0	1	2	3	4	5
–6	–5	–4	–3	–2	–1

Получить доступ, например, к символу n можно двумя способами: s[4] и s[–2].

Также существуют срезы с двумя параметрами: в результате применения среза s[a:b] будет сконструирована подстрока, начиная с символа на позиции a и заканчивая символом на позиции b–1 (правая граница не включается). Каждый из индексов может быть как положительным, так и отрицательным.

Например, при s = "String", s[1:5] будет равно "trin", это можно было бы записать и как s[1:–1]. Если в качестве второго числа в срезе взять число, больше либо равное длине строки, то ошибки не возникнет, и будут взяты все символы до конца строки.

Если нужно взять все символы строки, начиная с позиции a и до конца, то второй параметр можно опускать. Например, s[2:] будет равно "ring".

Если опустить первый параметр в срезе, то будет взята подстрока с начала, например s[:2] будет равно "St". Если же написать S[:], то будет взята вся строка от начала до конца.

Если первый параметр находится правее второго, то будет сгенерирована пустая строка.

Существует срез с тремя параметрами, где третий параметр задает шаг, с которым нужно брать символы. Например, можно взять все символы с начала до конца с шагом 2. Это будет выглядеть как s[::2]. В результате получится строка "Srn". Естественно, первый и второй параметры можно не опускать. Если третий параметр не указан, то есть в квадратных скобках записано только одно двоеточие, то шаг считается равным 1.

Шаг в срезе может быть и отрицательным. В таком случае первый параметр должен находиться правее второго. Например, s[5:1:–2] даст строку "gi" – 5-й и 3-й символы, а символ с номером 1 уже не входит. Развернутую строку можно получить срезом s[::–1] – все символы от «начала» до «конца» в обратном порядке. Если третий параметр отрицательный, то началом среза считается последний символ, а концом – позиция перед нулевым символом.

Метод find

Методы – это функции, применяемые к объектам. Метод вызывается с помощью записи ИмяОбъекта.НазваниеМетода(Параметры). Методы очень похожи на функции, но позволяют лучшим

образом организовывать хранение и обработку данных. Например, вы написали свою структуру данных и хотели бы, чтобы функция len возвращала длину вашей структуры. Чтобы это заработало, придется лезть в исходный код интерпретатора Питона и вносить изменения в функцию len. Если бы len было методом, то вы могли бы описать этот метод при создании структуры, и никаких изменений в коде интерпретатора или стандартной библиотеки не потребовалось бы. Поэтому методы предпочтительнее для сложных структур, таких, например, как строки.

У строк есть множество различных методов. В этом разделе мы рассмотрим методы поиска подстроки в строке. Метод find возвращает индекс первого вхождения подстроки в строку, а если она не нашлась, то –1. Например, 'String'.find('ing') вернет 3 – индекс, с которого начинается вхождение подстроки ing.

Существуют модификации этих методов с двумя параметрами. s.find(substring, from) будет осуществлять поиск в подстроке s[from:]. Например, 'String'.find('ing', 1) вернет 3 (остается нумерация символов, как в исходной строке). По аналогии со срезами параметры могут быть и отрицательными.

Есть модификации с тремя параметрами, они ищут подстроку в срезе s[a:b].

Часто возникает задача найти и вывести все вхождения подстроки в строку, включая накладывающиеся. Например, для строки 'ABABA' и подстроки 'ABA' ответ должен быть 0, 2. Ее решение выглядит так, как представлено на рис. 39.

```
1  string = input()
2  substring = input()
3  pos = string.find(substring)
4  while pos != -1:
5      print(pos)
6      pos = string.find(substring, pos + 1)
```

Рис. 39. Использование метода find

Методы rfind, replace и count

Метод rfind работает аналогично find, но ищет самое правое вхождение.

Метод replace(old, new) позволяет заменить все вхождения подстроки old на подстроку new. При этом конструируется новая строка, где были произведены замены. Нужно обратить внимание, что метод replace заменяет вхождения подстрок без учета предыдущих совершенных замен. Если далее применить операцию 'AAAAAA'.replace('AA', 'A'), то в результате получится строка 'AAA', а не 'A', как можно было бы ожидать.

Фактически можно считать, что метод replace находит очередное вхождение подстроки old, осуществляет замену и продолжает поиск с позиции после всех замененных символов (без наложения и поиска в замененной части).

Существует модификация replace(old, new, count), которая осуществляет не более count замен самых левых вхождений подстроки old.

Для строк существует метод count, который позволяет подсчитать количество вхождений подстроки. По аналогии с методом find определены методы count с двумя и тремя параметрами.

1.4. Создание функции для работы с данными

Функции

Функции – части программы, которые можно повторно вызывать с разными параметрами, чтобы не писать много раз одно и то же. Функции в программировании немного отличаются от математических функций. В математике функции могут только получить параметры и дать ответ, в программировании же функции умеют делать что-нибудь полезное, например ничего не возвращать, но что-то печатать.

Функции чрезвычайно полезны, если одни и те же действия нужно выполнять несколько раз. Но некоторые логические блоки работы с программой иногда тоже удобно оформлять в виде функции. Это связано с тем, что человек может одновременно держать в голове ограниченное количество вещей. Когда программа разрас-

тается, отследить все очень сложно. В пределах одной небольшой функции запутаться гораздо сложнее: известно, что она получает на вход, что должна выдать, а об остальной программе в это время можно не думать.

В программировании считается хорошим стилем писать функции, умещающиеся на один экран. Тогда можно одновременно окинуть взглядом всю функцию. Поэтому, если получилась очень длинная запись, нужно нарезать ее на кусочки так, чтобы каждый из них был логичным (делал какое-то определенное действие, которое можно назвать) и не превышал 10–15 строк.

Мы уже использовали готовые функции, такие как print, len и некоторые другие. Эти функции описаны в стандартной библиотеке или других подключаемых библиотеках. Сегодня мы научимся создавать свои функции.

Использование функций

Рассмотрим, как создать свою функцию, на примере вычисления факториала. Текст программы без функции выглядит так, как представлено на рис. 40.

```
1  n = int(input())
2  fact = 1
3  i = 2
4  while i <= n:
5      fact *= i
6      i += 1
7  print(fact)
```

Рис. 40. Текст программы без функции

Вычисление факториала можно вынести в функцию, тогда эта же программа будет выглядеть, как на рис. 41.

Описание функции должно идти в начале программы. На самом деле оно может быть в любом месте до первого вызова функции factorial.

Определение функции должно начинаться со слова def (сокращение от «define» – определить). Дальше идет имя функции, после которого в скобках через запятую перечисляются параметры

(у нашей функции всего один параметр). После закрытия скобки должно стоять двоеточие.

```
def factorial(num):
    fact = 1
    i = 2
    while i <= num:
        fact *= i
        i += 1
    return fact

n = int(input())
print(factorial(n))
```

Рис. 41. Текст программы с функцией

Команды, выполняемые в функции, должны записываться с отступом, как в блоках команд if или while.

В нашей функции num – это параметр, на его место подставляется то значение, с которым функция была вызвана. Действия внутри функции точно такие же, как в обычной программе, кроме дополнительной команды return. Команда return возвращает значение функции (оно должно быть записано через пробел после слова return) и прекращает её работу. Возвращенное значение подставляется на то место, где осуществлялся вызов функции.

Команда return может встречаться в любом месте функции. После того как она выполнится, работа функции будет прекращена. Здесь есть некоторая аналогия с командой break, применяемой для выхода из цикла.

Вызовы функций из функции

Функцию подсчета факториала можно использовать для подсчета биномиальных коэффициентов (числа сочетаний). Формула для подсчета числа сочетаний выглядит так: n! / (k! * (n – k)!).

Если бы мы не пользовались функциями, то нам потребовалось бы три раза записать почти одно и то же. С помощью функций вычисление выглядит намного проще (рис. 42).

Подсчет биномиальных коэффициентов можно также оформить в виде функции с двумя параметрами (рис. 43).

```
1  print(factorial(n) // (factorial(k) * factorial(n - k)))
```

Рис. 42. Пример использования функции factorial

```
1  def binomial(n, k):
2      return factorial(n) // (factorial(k) * factorial(n - k))
```

Рис. 43. Пример 1 задания функции

Возврат значений

Как было сказано выше, выполнение функции прерывается по команде return. Для примера рассмотрим функцию поиска максимума из двух чисел, которые передаются ей в качестве параметров (рис. 44).

```
1  def max2(a, b):
2      if a > b:
3          return a
4      else:
5          return b
```

Рис. 44. Пример 2 задания функции

Её можно было бы записать и по-другому (рис. 45).

Если условие в if'е было истинным, то выполнится команда return a, выполнение функции будет прекращено, до команды return b выполнение просто не дойдет.

```
1  def max2(a, b):
2      if a > b:
3          return a
4      return b
```

Рис. 45. Пример 3 задания функции

С помощью функции max2 можно реализовать функцию max3, возвращающую максимум из трех чисел (рис. 46).

```
def max3(a, b, c):
    return max2(max2(a, b), c)
```

Рис. 46. Пример 4 задания функции

Эта функция дважды вызывает max2: сначала для выбора максимума среди чисел a и b, а затем для выбора максимума между найденным значением и оставшимся числом c.

Здесь нужно обратить внимание, что в качестве аргумента функции может передаваться не только переменная или константное значение, но и результат вычисления любого арифметического выражения. Например, результат, возвращенный другой функцией.

Функции max2 и max3 будут работать не только для чисел, но и для любых сравнимых объектов, например для строк.

Возврат нескольких значений функцией

Рассмотрим случай, когда функция должна вернуть несколько значений, на примере функции, упорядочивающей два числа. Чтобы вернуть несколько значений, достаточно записать их в return через запятую. Аналогично через запятую должны быть перечислены переменные, в которые будут попадать вычисленные значения (рис. 47).

```
def sort2(a, b):
    if a < b:
        return a, b
    else:
        return b, a
a = int(input())
b = int(input())
minimum, maximum = sort2(a, b)
print(minimum, maximum)
```

Рис. 47. Пример 5 задания функции

На самом деле при перечислении значений через запятую формируются объекты типа «кортеж» (подробно о них – в разделе 1.5). Имеющихся знаний достаточно для использования функций, возвращающих несколько значений.

Возврат логических значений

Иногда удобно оформлять даже простые вещи в виде функций, чтобы повысить читаемость программы. Например: если нужно проверить число на четность, то гораздо понятнее будет вызывать функцию isEven(n), а не писать каждый раз n % 2 == 0.

Такая функция может выглядеть так, как представлено на рис. 48.

```
1 def isEven(n):
2     return n % 2 == 0
```

Рис. 48. Пример 6 задания функции

Результатом работы этой функции будет истина или ложь. Теперь функцию очень удобно применять в if‘ах (рис. 49).

```
1 if isEven(n):
2     print("EVEN")
3 else:
4     print("ODD")
```

Рис. 49. Пример использования заданной функции

Если есть сложное логическое выражение, то лучше оформить его в виде функции с говорящим названием. Так программу будет легче читать, а вероятность ошибок в ней резко снизится.

Локальные и глобальные переменные

Все переменные, которыми мы пользовались до настоящего момента, были глобальными. Глобальные переменные видны во всех функциях программы.

Например, такой код напечатает 1 и выполнится без ошибок (рис. 50).

```
1 def f():
2     print a
3 a = 1
4 f()
```

Рис. 50. Обращения к глобальной переменной

Переменная a – глобальная, поэтому мы можем смотреть на её значение из любой функции. На момент вызова функции f переменная a уже создана, хотя описание функции и идет раньше присваивания.

Если же инициализировать переменную внутри функции, то использовать её вне функции невозможно. Например, код на рис. 51 завершится с ошибкой «builtins.NameError: name 'a' is not defined» (переменная a не определена).

```
1  def f():
2      a = 1
3  f()
4  print(a)
```

Рис. 51. Объявление переменной внутри функции

Переменные, значения которых изменяются внутри функции по умолчанию, считаются локальными, то есть доступными только внутри функции. Как только функция заканчивает свою работу, переменная уничтожается.

Таким образом, если в функции происходило присваивание какой-то переменной, то эта переменная считается локальной. Если присваиваний не происходило, то переменная считается глобальной.

Локальные переменные можно называть такими же именами, как и глобальные. Например, вывод такого кода будет «1 0» (рис. 52).

```
1  def f():
2      a = 1
3      print(a, end=' ')
4  a = 0
5  f()
6  print(a)
```

Рис. 52. Пример использования локальной и глобальной переменных

Сначала произойдет вызов функции f, в которой будет создана локальная переменная a со значением 1 (получить доступ к глобальной переменной a из функции теперь нельзя). Затем функция

закончит свою работу, и будет выведена глобальная переменная *a*, со значением которой ничего не случилось.

Переменная считается локальной даже в случае, если её присваивание происходило внутри условного оператора (даже если он никогда не выполнится) (рис. 53).

Рис. 53. Пример использования локальной переменной

Эта программа завершится с ошибкой builtins.UnboundLocalError: local variable 'a' referenced before assignment (обращение к переменной до инициализации). Любое присваивание значения переменной внутри тела функции делает переменную локальной.

С помощью специальной команды global можно сделать так, что функция изменит значение глобальной переменной. Для этого нужно записать в начале функции слово global, а затем через запятую перечислить имена глобальных переменных, которые функция сможет менять. Например, такой код выведет «1 1», так как значение глобальной переменной будет изменено внутри функции (рис. 54).

Рис. 54. Пример использования команды global

Все параметры функции являются локальными переменными со значениями, которые были переданы в функцию. Параметры также можно изменять, и это никак не повлияет на значения переменных в том месте, откуда была вызвана функция (если тип объектов-параметров был неизменяемым).

Использование глобальных переменных как на чтение, так и на запись внутри функций некорректно. Это связано с тем, что другие люди могут захотеть использовать некоторые отдельные функции из вашего кода, которые не будут работать вне вашей программы в случае использования глобальных переменных.

Поэтому использование глобальных переменных внутри функций строго не рекомендуется. Все нужное для работы функции должно передаваться в качестве параметров.

Рекурсия

Мы уже пробовали запускать функцию из другой функции. Ничто не мешает запустить из функции саму себя. Тогда просто создастся новый экземпляр функции, который будет выполнять те же команды. Такой процесс называется рекурсией.

Представим себе, что у нас есть миллиард человек (это будущие экземпляры функции), сидящих в ряд, и у каждого из них есть листочек для записи (это его локальная память). Нам нужно произносить числа и написать инструкцию для людей так, чтобы они в итоге назвали все числа из последовательности в обратном порядке. Пусть каждый из них будет записывать на своем листочке только одно число. Тогда инструкция для человека будет выглядеть так:

1. Запиши названное число.

2. Если число не последнее, «потереби» следующего за тобой человека, пришла его очередь работать.

3. Когда следующий за тобой человек сказал, что он закончил, назови записанное число.

4. Скажи тому, кто тебя «теребил» (предыдущий человек), что ты закончил.

Формализуем задачу. Пусть задается последовательность натуральных чисел, заканчивающаяся нулем. Необходимо развернуть ее с помощью рекурсии (рис. 55).

Эта функция осуществляет действие (вывод числа) на рекурсивном спуске, то есть после рекурсивного вызова.

```
def rec():
    n = int(input())
    if n != 0:
        rec()
        print(n)
rec()
```

Рис. 55. Пример использования рекурсии

Использование рекурсии

Рассмотрим задачу 1, где действия выполняются как на рекурсивном подъёме, так и на рекурсивном спуске. Пусть дана последовательность, которая оканчивается нулём. Необходимо вывести все чётные члены последовательности в прямом порядке, а затем все нечётные члены последовательности в обратном порядке. Решение будет выглядеть так, как представлено на рис. 56.

```
def rec():
    n = int(input())
    if n != 0:
        if n % 2 == 0:
            print(n)
        rec()
        if n % 2 != 0:
            print(n)
rec()
```

Рис. 56. Решение задачи 1 с использованием рекурсии

Каждый экземпляр функции считывает в свою локальную переменную n число. Если оно чётное, то сразу выводит его и запускает следующий экземпляр. После того как все последующие экземпляры функции окончили работу (и вывели последующие нечётные числа в обратном порядке), функция выводит число, если оно было нечетным.

Рассмотрим задачу 2: подсчитать факториал числа, не пользуясь циклами (рис. 57).

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)
n = int(input())
print(factorial(n))
```

Рис. 57. Решение задачи 2 с использованием рекурсии

Тем, кто знаком с методом математической индукции, будет довольно просто осознать рекурсию. Как и в математической индукции, в рекурсии должны быть база (момент, когда функция не вызывает другую рекурсивную функцию) и переход (правило, по которому считается результат по известному результату для меньшего параметра). Функция подсчета факториала делает только свою работу, но пользуется результатами чужого труда. Например, если функция получила на вход параметр 4, то должна вернуть 4, умноженное на 3! (факториал будет посчитан другими функциями). В случае факториала аналогом «базы индукции» может выступать 0! По определению он равен единице.

Эти примеры иллюстрируют общую схему написания рекурсивных функций: сначала проверяется условие, когда функция должна закончиться, а дальше делается все остальное. При этом параметр должен сходиться к значению базы. Обычно это означает, что при каждом следующем вызове рекурсии параметр должен уменьшаться.

1.5. Кортежи и списки данных. Цикл for для работы с элементами коллекций

Кортежи

Наряду со строками, которые могут хранить в себе отдельные символы, в языке Питон существует тип кортеж, который позволяет хранить в себе произвольные элементы.

Кортеж может состоять из элементов произвольных типов и является неизменяемым типом, то есть нельзя менять отдельные элементы кортежа, как и символы строки. Константные кортежи можно создавать в программе, записывая элементы через запятую и окружая скобками. Например: testTuple = (1, 2, 3). Если кортеж

является единственным выражением слева или справа от знака присваивания, то скобки могут быть опущены. Во всех остальных случаях скобки опускать не следует, это может привести к ошибкам.

Многие приемы и функции для работы со строками подходят и для кортежей. Например, можно складывать два кортежа (рис. 58).

```
1  a = (1, 2, 3)
2  b = (4, 5, 6)
3  print(a + b)
```

Рис. 58. Пример работы с кортежами

В результате применения этой операции будет выведено (1, 2, 3, 4, 5, 6). В случае сложения создается новый кортеж, который содержит в себе элементы сначала из первого, а затем из второго кортежа (точно так же, как и в случае со строками). Кортеж можно умножить на число, результат этой операции аналогичен умножению строки на число.

Приведем пример, когда опускание скобок приводит к ошибке. (1, 2) + (3, 4) будет давать (1, 2, 3, 4), а 1, 2 + 3, 4 будет давать (1, 5, 4), так как сумма будет понята Питоном как выражение для второго элемента кортежа.

Кортеж можно получить из строки, вызвав функцию tuple от строки. В результате каждая буква станет элементом кортежа. К кортежу можно применять функцию str, которая вернет текстовое представление кортежа (элементы, перечисленные через запятую с пробелом и разделенные пробелами).

К кортежу можно применять функцию len и обращаться к элементам по индексу (в том числе по отрицательному), так же как и к строкам.

В одном кортеже могут храниться элементы различных типов, например строки, числа и другие кортежи вперемешку. Например, в кортеже myTuple = (('a', 1, 3.14), 'abc', ((1), (2,))), myTuple[0] будет кортежем ('a', 1, 3.14), myTuple[1] – строкой 'abc±, а myTuple[2] – кортежем, состоящим из числа 1 и кортежа из одного элемента (2,). Числа, записанные в скобках, интерпретируются как числа. В случае возникновения необходимости создать кортеж из одного

элемента необходимо после значения элемента написать запятую. Если вывести myTuple[2][1], то напечатается (2,), а если вывести myTuple[2][1][0], то будет напечатано число 2.

Кортеж, содержащий в себе один элемент, называется синглтоном. Как и к строкам, к кортежам можно применять операцию среза с тем же смыслом параметров. Если в срезе один параметр, то будет возвращена ссылка на элемент с соответствующим номером. Например, print((1, 2, 3)[2]) напечатает 3. Если же в срезе более одного параметра, будет сконструирован кортеж, даже если он будет синглтоном. Например, в случае вызова print((1, 2, 3)[1:]) будет напечатано (2, 3), а в случае вызова print((1, 2, 3)[2:]) будет напечатан синглтон (3,).

Кортежи обычно предназначаются для хранения разнотиповых значений, доступ к которым может быть получен в результате обращения по индексу или с помощью операции распаковки.

Распаковкой называется процесс присваивания, в котором кортеж, составленный из отдельных переменных, находится в левой части выражения. В таком выражении справа должен находиться кортеж той же длины, например в результате выполнения кода на рис. 59.

```
1  manDesc = ("Ivan", "Ivanov", 28)
2  name, surname, age = manDesc
```

Рис. 59. Пример работы с кортежем

В переменной name хранится «Ivan», в surname – «Ivanov», а в переменной age число 28. На английском распаковка кортежа называется tuple unpacking.

Функция range, цикл for

Процесс создания кортежа называется упаковкой кортежа. Если в одном выражении присваивания происходит и упаковка, и распаковка кортежа, то сначала выполняется упаковка, а затем распаковка кортежа. Так, в результате работы программы будет выведено «3 2 1» (рис. 60).

```
1  a, b, c = 1, 2, 3
2  a, b, c = c, b, a
3  print(a, b, c)
```

Рис. 60. Пример создания кортежа

Обратите внимание, что функции print передается в качестве параметра не кортеж, а три целых числа.

Главное, что нужно понять: запись вида (a, b, c) = (c, b, a) не эквивалентна цепочке присваиваний вида a = c; b = b; c = a. Такая цепочка присваиваний привела бы к тому, что в переменных a, b, c оказались бы значения 3, 2, 3.

Функция range

В языке Питон есть функция range, которая позволяет генерировать объекты типа iterable (к элементам которых можно получать последовательный доступ), состоящие из целых чисел.

Для вывода объектов типа iterable мы будем пользоваться функцией tuple, которая позволяет сделать кортеж, состоящий из всех элементов iterable, записанных последовательно.

Например, если запустить программу, то будет напечатано (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) (рис. 61).

```
1  print(tuple(range(10)))
```

Рис. 61. Пример использования функции range

Функция range с одним параметром n генерирует iterable, содержащий последовательные числа от 0 до n – 1.

Существует вариант range с двумя параметрами, range(from, to) сгенерирует iterable со всеми числами от from до to – 1 включительно.

Существует range с тремя параметрами range(from, to, step), который сгенерирует iterable с числами от from, не превышающие to с шагом изменения step. Если шаг отрицателен, то from должен быть больше to. Например, range(10, 0, –2) сгенерирует последовательность чисел 10, 8, 6, 4, 2.

0 не будет входить в эту последовательность.

Во многом параметры range напоминают значения параметров в срезах строк.

Цикл for

Цикл for позволяет поочередно перебрать элементы из чего-нибудь итерируемого (iterable или tuple). Например, мы можем перебрать названия цветов яблок (рис. 62).

```
1 for color in ('red', 'green', 'yellow'):
2     print(color, 'apple')
```

Рис. 62. Пример использования цикла for

В результате выполнения этой программы будет напечатано:

red apple

green apple

yellow apple

На место переменной color будут поочередно подставляться значения из кортежа. В общем случае цикл for выглядит так: for «имяПеременной» in «нечтоИтерируемое»:

Все действия, которые должны выполняться в for, должны выделяться отступом, как в if или while. Работа цикла for может быть прервана с помощью команды break, или может быть осуществлен переход к следующей итерации с помощью continue. Эти команды имеют тот же эффект, что и при работе с циклом while.

Часто for используется вместе с функцией range. Например, с помощью for можно напечатать нечетные числа от 1 до 100 (рис. 63).

```
1 for i in range(1, 100, 2):
2     print(i)
```

Рис. 63. Пример использования цикла for

Внутри for может быть расположен и другой for. На рис. 64 показано, как выглядит код для вывода таблицы умножения всех чисел от 1 до 10 (не очень красивой).

```
for i in range(1, 11):
    for j in range(1, 11):
        print(i * j, end=' ')
    print()
```

Рис. 64. Пример использования вложенных циклов for

Как можно заметить, при использовании функции range в for мы не преобразовывали iterable в tuple. Это связано с тем, что for как раз хочет получать последовательный доступ, который умеет давать iterable. Tuple умеет намного больше, но здесь его использование приведет к ненужным затратам времени и памяти.

Списки

Список в Питоне является аналогом массивов в других языках программирования. Список – это набор ссылок на объекты (так же, как и кортеж), однако он является изменяемым.

Константные списки записываются в квадратных скобках, все остальное в них аналогично кортежам. Например, можно создать список с числами от 1 до 5: myList = [1, 2, 3, 4, 5].

Списки и кортежи легко преобразуются друг в друга. Для преобразования списка в кортеж надо использовать уже известную нам функцию tuple, а для преобразования кортежа в список нужна функция list. Также функцию list можно применить к строке. В результате этого получится список, каждым элементом которого будет буква из строки. Так list('abc') будет выглядеть как ['a', 'b', 'c'].

К спискам также применимы функция len и срезы, которые работают так же, как в кортежах.

Главным отличием списка от кортежа является изменяемость. То есть можно взять определенный элемент списка и изменить его (он может быть в левой части операции присваивания).

Например, в результате выполнения кода на рис. 65 будет напечатано [1, 4, 3].

Изменение символа (или элемента в кортеже) можно было реализовать, сделав два среза и конкатенацию первой части строки, нового символа и «хвоста» строки. Это очень медленная операция, время ее выполнения пропорционально длине строки. Замена элемента в списке осуществляется за O(1), то есть не зависит от длины списка.

```
1  myList = [1, 2, 3]
2  myList[1] = 4
3  print(myList)
```

Рис. 65. Пример 1 работы со списками

Изменение списков

Список, как и другие типы в языке Питон, является ссылкой на список ссылок. При этом список является изменяемым объектом, то есть содержимое по этой ссылке может поменяться. На рис. 66 приведен пример.

```
1  a = [1, 2]
2  b = a
3  b[0] = 3
4  print(a)
```

Рис. 66. Пример 2 работы со списками

В результате выполнения программы будет напечатано [3, 2]. Это связано с тем, что присваивание в Питоне – это просто «привязывание» нового «ярлычка» к объекту. После присваивания b = a обе ссылки начинают показывать на один и тот же объект, и если он изменен по одной ссылке, то по второй ссылке он также будет доступен в измененном состоянии.

Если же написать другой код (рис. 67), то будет выведено [1, 2].

```
1  a = [1, 2]
2  b = [1, 2]
3  a[0] = 3
4  print(b)
```

Рис. 67. Пример 3 работы со списками

Несмотря на то, что объекты имеют одинаковое значение из-за их мутабельности (изменяемости), для каждого значения будет создан отдельный объект, и ссылки a и b будут показывать на разные объекты. Изменение одного из них не приводит к изменению другого.

В результате выполнения кода, представленного на рис. 68, будет выведено [1, 2].

```
1  a = [1, 2]
2  b = a
3  a = [3, 4]
4  print(b)
```

Рис. 68. Пример 4 работы со списками

Сначала в памяти создается объект [1, 2], к нему привязывается ссылка a. Затем к тому же объекту привязывается ссылка b, и создается новый объект [3, 4], к которому привязывается ссылка a (отвязавшись от своего предыдущего значения). При этом ссылка b не изменилась (она может измениться, только если b будет участвовать в левой части присваивания) и по-прежнему показывает на [1, 2].

Если списки переданы в функцию в качестве параметров, то их содержимое также может быть изменено этой функцией (рис. 69).

```
1  def replaceFirst(myList):
2      myList[0] = 'x'
3
4  nowList = list('abcdef')
5  replaceFirst(nowList)
6  print(nowList)
```

Рис. 69. Пример 5 работы со списками

Вывод этой программы будет ['x', 'b', 'c', 'd', 'e', 'f'].

Однако сама ссылка внутри функции не может быть изменена, если она передана как параметр функции. Рассмотрим пример на рис. 70.

Эта программа не развернет список, то есть вывод будет ['a', 'b', 'c'].

Здесь в основной программе конструируется объект ['a', 'b', 'c'], и к нему привязывается ссылка mainList. При передаче mainList в качестве параметра в функцию создастся еще одна ссылка funcList, показывающая на объект ['a', 'b', 'c']. В результате применения среза создастся новый объект ['c', 'b', 'a'], и ссылка funcList

начнет указывать на него. Однако значение ссылки mainList при этом не изменится, и со значениями по ссылке mainList также ничего не произойдет (напомним, что операция среза создает новый объект, не изменяя старый).

```
1  def reverseList(funcList):
2      funcList = funcList[::-1]
3
4  mainList = list('abc')
5  reverseList(mainList)
6  print(mainList)
```

Рис. 70. Пример 6 работы со списками

Методы split и join

Строки имеют два полезных метода, которые пригодятся при работе со списками.

Метод split позволяет разрезать строку (string) на отдельные слова («токены»). В качестве разделителя может выступать пробел, символ табуляции или перевода строки. Этот метод не изменяет строку и возвращает список строк-токенов.

Например, если запустить такую программу, то будет напечатано ['red', 'green', 'blue'] (рис. 71). Количество разделителей между токенами не играет роли.

```
1  print('red green        blue'.split())
```

Рис. 71. Пример использования split

Чтобы научиться читать числа из одной строки, нужно научиться еще одной функции – map. Функция map принимает два параметра: первый – это функции, а второй – iterable элементов, к которому нужно применить эту функцию. В результате получается iterable с результатом применения функции к каждому элементу списка параметра.

Например, код на рис. 72 напечатает [3, 5, 4] – список с результатом применения функции len к списку ['red', 'green', 'blue'].

```
print(list(map(len, ['red', 'green', 'blue'])))
```

Рис. 72. Пример использования map

Метод split в сочетании с функцией map удобно использовать для считывания списка чисел, записанных в одну строку и разделенных пробелами. Такое считывание будет выглядеть так, как представлено на рис. 73.

```
numList = list(map(int, input().split()))
```

Рис. 73. Пример использования split и map

Сначала осуществляется считывание строки, затем выполняется метод split, который создает список токенов, состоящих из цифр, а затем к каждому токену применяется функция int. В результате этого получается список цифр.

Метод join позволяет объединить iterable строк, используя ту строку, к которой он применен, в качестве разделителя. Например, код на рис. 74 выведет Veni, Vidi, Vici. Строка ', ' выступает в качестве разделителя, который будет вставляться после каждой строки из списка-параметра (кроме последней).

```
print(', '.join(['Veni', 'Vidi', 'Vici']))
```

Рис. 74. Пример 1 использования join

Метод join позволяет быстро и коротко выводить списки чисел. Проблема в том, что он умеет принимать в качестве параметра только iterable строк. Но с помощью функции map мы можем легко получить iterable из списка чисел, применив к каждому элементу функцию str. Вывод списка чисел numList, разделенных пробелами, будет выглядеть так, как представлено на рис. 75.

```
1  numList = [1, 2, 3]
2  print(' '.join(map(str, numList)))
```

Рис. 75. Пример 2 использования join

Полезные методы работы со списками

К переменным типа список можно применять методы, некоторые из которых приведем ниже.

Методы, не изменяющие список и возвращающие значение:

– count(x) – подсчитывает число вхождений значения x в список, работает за время O(N);
– index(x) – находит позицию первого вхождения значения x в список, работает за время O(N);
– index(x, from) – находит позицию первого вхождения значения x в список, начиная с позиции from, работает за время O(N).

Методы, не возвращающие значение, но изменяющие список:

– append(x) – добавляет значение x в конец списка;
– extend(otherList) – добавляет все содержимое списка otherList в конец списка. В отличие от операции + изменяет объект, к которому применен, а не создает новый;
– remove(x) – удаляет первое вхождение числа x в список, работает за время O(N);
– insert(index, x) – вставляет число x в список так, что оно оказывается на позиции index. Число, стоявшее на позиции index, и все числа правее него сдвигаются на один вправо. Работает за время O(N);
– everse() – разворачивает список (меняет значение по ссылке, а не создает новый список как myList[::–1]). Работает за время O(N).

Методы, возвращающие значение и изменяющие список:

– pop() – возвращает последний элемент списка и удаляет его;
– pop(index) – возвращает элемент списка на позиции index и удаляет его, работает за время O(N).

Обработка списков

Рассмотрим такую задачу: необходимо выбрать все нечетные элементы списка myList и удалить их из него.

Попробуем решить задачу «в лоб»: просто будем перебирать все позиции в строке и, если на этой позиции стоит нечетное число, будем удалять его (рис. 76).

```
numbers = list(map(int, input().split()))
for i in range(len(numbers)):
    if numbers[i] % 2 != 0:
        numbers.pop(i)
print(' '.join(map(str, numbers)))
```

Рис. 76. Пример 1 обработки списков

Такое решение будет работать неправильно в ситуации, когда в списке есть хоть одно нечетное число. Это связано с тем, что объект без названия с типом iterable и значением range(len(numbers)) сгенерируется один раз, когда интерпретатор впервые дойдет до этого места, и уже никогда не изменится. Если в процессе мы выкинем из списка numbers хоть одно значение, то в процессе перебора всех индексов выйдем за пределы нашего списка.

range, используемый в for, не будет менять свое значение, если в процессе работы изменились параметры функции range.

Решение можно переписать с помощью while (рис. 77).

```
numbers = list(map(int, input().split()))
i = 0
while i < len(numbers):
    if numbers[i] % 2 != 0:
        numbers.pop(i)
    else:
        i += 1
print(' '.join(map(str, numbers)))
```

Рис. 77. Пример 2 обработки списков

Такое решение будет работать, но оно не очень эффективно. Каждый раз при удалении элемента нам придется совершать количество операций, пропорциональное длине списка. Итоговое

количество операций в худшем случае будет пропорционально квадрату количества элементов в списке.

Если нет очень строгого ограничения в памяти, в задачах, где нужно удалить часть элементов списка, гораздо проще создать новый список, в который нужно добавлять только подходящие элементы (рис. 78).

```
numbers = list(map(int, input().split()))
newList = []
for i in range(len(numbers)):
    if numbers[i] % 2 == 0:
        newList.append(numbers[i])
print(' '.join(map(str, newList)))
```

Рис. 78. Пример 3 обработки списков

Сложность такого решения пропорциональна длине исходного списка, что намного лучше.

Контрольные вопросы (тесты)

1. Что будет выведено в результате выполнения кода?

```
myString = 'Python'
myString[-1]
```

2. Что будет выведено в результате выполнения кода?

```
set('python')
```

3. Что будет выведено в результате выполнения кода?

```
myTuple = (1, 2, 3, 4, 5)
myTuple.__sizeof__()
```

4. Заполните пропуск в вызове метода, результатом выполнения которого является список: ['g', 'a', 'b', 'c', 'd', 'f']

```
myList = ['a','b','c','d','f']
myList._____(0, 'g')
myList
```

5. Заполните пропуск в вызове метода, результатом выполнения которого является: 'Forester'

```
myDict = {
"brand": "Subaru",
"model": "Forester",
"year": 2020
}
x = myDict.___("model")
x
```

6. Введите пропущенную часть кода:

```
a = 200
b = 100
c = 500
if a > b __ c > a:
print(«Оба утверждения верны»)
```

2. УПРАВЛЕНИЕ ДАННЫМИ С ИСПОЛЬЗОВАНИЕМ PYTHON

2.1. Сортировка данных

Сортировка. Сравнение кортежей и списков

Сортировка списков

В программировании очень часто удобнее работать с отсортированными данными. В языке Питон существует возможность отсортировать списки двумя способами. Рассмотрим их на примере решения простой задачи о сортировке последовательности чисел по неубыванию (то есть как по возрастанию, но, возможно, с одинаковыми числами). Первый способ упорядочить его приведен на рис. 79.

```
1  myList = list(map(int, input().split()))
2  myList.sort()
3  print(' '.join(map(str, myList)))
```

Рис. 79. Пример 1 сортировки списков

В этом примере используется метод sort, применяемый к списку. Этот метод изменяет содержимое списка – после применения метода sort элементы в списке становятся упорядоченными. Такой метод определен только для объектов типа список, его нельзя применить к кортежу, iterable или строке.

Второй способ состоит в применении функции sorted, которая возвращает отсортированный список, но не изменяет значение своего параметра (рис. 80).

```
1  myList = list(map(int, input().split()))
2  sortedList = sorted(myList)
3  print(' '.join(map(str, sortedList)))
```

Рис. 80. Пример 2 сортировки списков

Использование функции sorted оправдано в случае, если исходные данные нужно сохранить в неизменном виде с какой-то целью.

Например, sorted можно использовать внутри своей функции для создания отсортированной копии, чтобы не портить переданный нам список.

Чтобы отсортировать список по невозрастанию (убыванию), необходимо передать в метод или функцию именованный параметр reverse. Например, это будет выглядеть как myList.sort(reverse=True) или sorted(myList, reverse=True).

Функция sorted может принимать в качестве параметра не только список, но и что угодно итерируемое: кортежи, iterable или строки (рис. 81).

```
1  print(sorted((1, 3, 2)))
2  print(sorted(range(10, -1, -2)))
3  print(sorted("cba"))
```

Рис. 81. Пример 3 сортировки списков

При этом sorted всегда возвращает список, то есть вывод этой программы будет такой:

[1, 2, 3]

[0, 2, 4, 6, 8, 10]

['a', 'b', 'c']

Сортировку можно применять к спискам, все элементы которых сравнимы между собой. Обычно это однородные списки (состоящие из элементов одного типа) или (в редких случаях) целые и вещественные числа вперемешку.

Сравнение кортежей и списков

Два кортежа или списка можно также сравнивать между собой. Например, выражение (1, 2, 3) < (2, 3, 4) будет истинным, а [1, 2, 3] < [1, 2] – ложным. Сравнение кортежей и списков происходит поэлементно, как и сравнение строк. Как только на каких-то позициях кортежа или списка встретились различные элементы, взаимный порядок кортежей такой же, как у этих элементов. Если же различий найдено не было, то меньше тот кортеж, который короче, как при сравнении строк.

Естественно, сравниваемые кортежи или списки должны содержать на соответствующих позициях сравнимые элементы.

Попытка сравнить кортеж (1, 2) с кортежем («Some text», 42) приведет к ошибке (а сравнение (1, 2) с (42, «Some text») к ошибке не приведет). Обычно сравниваются кортежи, состоящие из элементов одинакового типа.

Это свойство кортежей можно использовать для решения сложных задач на сортировку.

Именованный параметр key

Пусть для каждого человека заданы его рост и имя. Необходимо упорядочить список людей по росту, а список людей одинакового роста – в алфавитном порядке. При решении этой задачи достаточно хранить описание каждого человека в виде кортежа, в котором первым элементом будет рост, а вторым – имя.

Рассмотрим пример: для каждого человека заданы рост и имя. Необходимо упорядочить список людей по возрастанию роста, а список людей одинакового роста – в алфавитном порядке (рис. 82).

```
1  n = int(input())
2  peopleList = []
3  for i in range(n):
4      tempManData = input().split()
5      manData = (int(tempManData[0]), tempManData[1])
6      peopleList.append(manData)
7  peopleList.sort()
8  for manData in peopleList:
9      print(' '.join(map(str, manData)))
```

Рис. 82. Пример 1 работы с данными

В этом примере удалось составить кортеж, который содержит параметры сравниваемых людей ровно в нужном порядке. Часто встречаются более сложные ситуации. Рассмотрим ту же задачу, но теперь список людей нужно упорядочить по убыванию их роста, а список людей одинакового роста по-прежнему должен быть упорядочен по алфавиту. Простое использование reversed=True не приведет к желаемому результату: имена людей одинакового роста будут стоять не в алфавитном порядке.

Здесь можно применить хитрость и превратить рост каждого человека в отрицательное число, модуль которого равен исходному

росту. После этого список можно просто упорядочить по возрастанию – самые высокие люди будут иметь наименьший отрицательный «рост», по которому происходит сравнение в первую очередь. Перед выводом необходимо превратить рост обратно в положительное число (рис. 83).

```
n = int(input())
peopleList = []
for i in range(n):
    tempManData = input().split()
    manData = (-int(tempManData[0]), tempManData[1])
    peopleList.append(manData)
peopleList.sort()
for badManData in peopleList:
    manData = (-badManData[0], badManData[1])
    print(' '.join(map(str, manData)))
```

Рис. 83. Пример 2 работы с данными

Этот код малопонятен и плох.

Параметр key в функции sort

Для реализации нестандартных сортировок лучше не уродовать исходные данные, а использовать параметр key, передающийся в функцию сортировки.

Значением этого параметра должна быть функция, которая применяется к каждому элементу списка, и затем сравнение элементов происходит по значению этой функции (оно называется ключом).

Рассмотрим такой пример: необходимо упорядочить введённые строки по длине, а в случае равной длины оставить их в том порядке, в котором они шли во входном файле. Например, для входных строк "c", "abb", "b" правильным ответом должно быть "c", "b", "abb" ("c" идет раньше "b", так как они имеют равную длину, а "c" стояло во входных данных раньше "b").

К счастью, сортировка, используемая в Питоне, обладает свойством устойчивости (stable), то есть для элементов с равным ключом сохраняется их взаимный порядок.

Решение этой задачи представлено на рис. 84.

```
n = int(input())
strings = []
for i in range(n):
    strings.append(input())
print('\n'.join(sorted(strings, key=len)))
```

Рис. 84. Пример 1 использования параметра key

В качестве еще одного примера рассмотрим задачу о сортировке точек на плоскости, заданных парой целых координат x и y по неубыванию расстояния от начала координат. В данном случае в качестве функции для генерации ключа, по которому будут сравниваться элементы, мы напишем свою функцию, которая будет возвращать квадрат расстояния от точки до начала координат. Квадрат расстояния мы используем для того, чтобы оставаться в целых числах и избавиться от необходимости считать квадратный корень (медленно и неточно) (рис. 85).

```
def dist(point):
    return point[0] ** 2 + point[1] ** 2

n = int(input())
points = []
for i in range(n):
    point = tuple(map(int, input().split()))
    points.append(point)
points.sort(key=dist)
for point in points:
    print(' '.join(map(str, point)))
```

Рис. 85. Пример 2 использования параметра key

Здесь каждый элемент списка – кортеж из двух чисел. Именно такой параметр принимает наша функция. Использование функции hypot из библиотеки math (чтобы не писать свою функцию подсчета ключа) невозможно – она ожидает на вход два числовых параметра, а не кортеж.

Структуры в Питоне

Для хранения сложных записей во многих языках есть специальные типы данных, такие как struct в C++ или record в Паскале.

Переменная типа «структура» содержит в себе несколько именованных полей. Например, возвращаясь к задаче сортировки людей по убыванию роста, нам было бы удобно хранить описание каждого человека в виде структуры с двумя полями: рост и имя.

В чистом виде типа данных «структура» в стандарте языка Питон нет. Есть несколько способов реализации аналога структур: namedtuple из библиотеки collections, использование словарей (будет рассмотрено в следующих разделах) или использование классов в качестве структур. Рассмотрим на примере последний способ.

Условие задачи: список людей нужно упорядочить по убыванию роста, но список людей одинакового роста должен быть представлен в алфавитном порядке. Решение с использованием классов в качестве структур будет выглядеть так, как представлено на рис. 86.

```
class Man:
    height = 0
    name = ''

def manKey(man):
    return (-man.height, man.name)

n = int(input())
peopleList = []
for i in range(n):
    tempManData = input().split()
    man = Man()
    man.height = int(tempManData[0])
    man.name = tempManData[1]
    peopleList.append(man)
peopleList.sort(key=manKey)
for man in peopleList:
    print(man.height, man.name)
```

Рис. 86. Пример использования структур

Для того чтобы пользоваться классами как структурами, мы создаем новый тип данных Man. В описании класса мы перечисляем имена всех полей и их значения по умолчанию.

В дальнейшем мы можем создавать объекты класса Man (это делается строкой man = Man()), которые сначала инициализируют свои поля значениями по умолчанию. Доступ к полям класса осуществляется через точку.

Функция сравнения принимает объект класса и генерирует ключ, по которому эти объекты будут сравниваться при сортировке.

Использование структур для описания сложных объектов намного предпочтительнее, чем использование кортежей. При количестве параметров больше двух использование кортежей запутывает читателя и писателя кода, так как совершенно невозможно понять, что хранится в badNamedTuple[13], и легко понять, что хранится в goodNamedStruct.goodNamedField.

Лямбда-функции

В ряде случаев функции, используемые для получения ключа сортировки, так просты, что хочется не оформлять их стандартным образом, а написать прямо на месте и даже не давать им имени.

Это можно осуществить с помощью лямбда-функций, которые могут заменить собой функции, содержащие в своем теле только оператор return. Запись лямбда-функции, возводящей число в квадрат, может выглядеть так, как представлено на рис. 87.

```
1  lambda x: x**2
```

Рис. 87. Пример 1 использования лямбда-функции

Это эквивалентно привычной записи функции (рис. 88).

```
1  def sqr(x):
2      return x**2
```

Рис. 88. Эквивалентная запись

У лямбда-функции нет имени, и, следовательно, вызов её по имени невозможен. Пока единственным применением лямбда-функций для нас может служить их передача в качестве параметра в такие функции, как sort или map. Например, с помощью лямбда-функции мы можем вывести список квадратов всех чисел от 1 до 100 всего в одну строку (рис. 89).

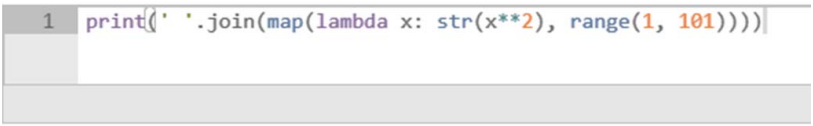


Рис. 89. Пример 2 использования лямбда-функции

В этой программе лямбда-функция принимает в качестве параметра число, а возвращает строковое представление его квадрата.

Вернемся к задаче сортировки точек по удаленности от начала координат. Эта задача также может быть решена с использованием лямбда-функции (рис. 90).

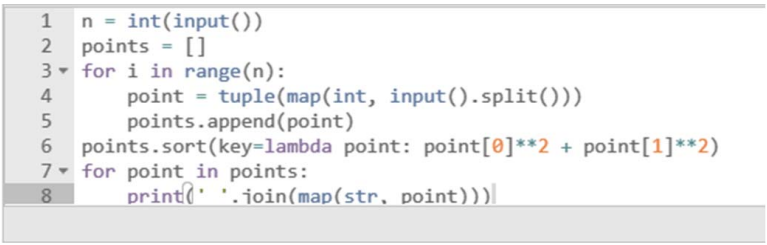


Рис. 90. Пример 3 использования лямбда-функции

Лямбда-функция может принимать несколько параметров (тогда после слова lambda нужно записать их имена через запятую), однако при использовании их в sort или map обычно применяется один параметр.

В языке Питон функция также является объектом, и мы можем создать ссылку на объект типа функция. Например, две записи функции возведения в квадрат эквивалентны (рис. 91).

Рис. 91. Пример 4 использования лямбда-функции

Такой подход позволяет переиспользовать лямбда-функции, но в подавляющем большинстве случаев стоит пользоваться стан-

дартным объявлением функции – это упрощает чтение и отладку программы.

Именованные параметры и неопределенное число параметров

Мы уже много раз пользовались функциями, которые могут принимать (или не принимать) именованные параметры. Например, это необязательные именованные или неименованные параметры sep и end для функции print или параметр key для метода sort и функции sorted.

Сейчас мы научимся создавать функции, которые принимают именованные параметры. Например, напишем функцию, печатающую что угодно итерируемое, состоящее из чего угодно приводимого к строке, с именованным параметром sep, по умолчанию равным пробелу (рис. 92).

```
1  def printList(myList, sep=' '):
2      print(sep.join(map(str, myList)))
3
4  printList([1, 2, 3])
5  printList([3, 2, 1], sep='\n')
```

Рис. 92. Пример 1 использования функций

Именованный параметр в объявлении функции должен идти после основных параметров. В списке параметров записывается его имя, а затем значение по умолчанию (то есть то значение, которое будет подставляться на место соответствующего параметра, если он не был передан при вызове функции).

Мы пользовались функциями, которые умеют принимать произвольное количество параметров. Например, в функцию print можно передать любое количество параметров. Можно написать собственные функции, которые будут принимать произвольное количество параметров. При этом параметры функции будут упакованы в список. Например, функция подсчета суммы всех переданных параметров может выглядеть так, как представлено на рис. 93.

Функция принимает один параметр, перед которым написана звездочка. Это признак того, что аргументы будут упакованы в список.

```
1  def mySum(*args):
2      nowSum = 0
3      for now in args:
4          nowSum += now
5      return nowSum
6
7  print(mySum(1, 2))
8  print(mySum(1, 2, 3, 4))
```

Рис. 93. Пример 2 использования функций

Можно писать функции, которые принимают не менее определенного количества параметров. Например, мы можем написать функцию поиска минимума среди неопределенного числа аргументов, но в нее должно быть передано не менее одного аргумента (рис. 94).

```
1  def myMin(first, *others):
2      nowMin = first
3      for now in others:
4          if now < nowMin:
5              nowMin = now
6      return nowMin
7
8  print(myMin(1))
9  print(myMin(3, 1, 2))
```

Рис. 94. Пример 3 использования функций

Параметр со звездочкой всегда должен быть последним, за исключением ситуации, когда в функции также определены именованные параметры.

Чтение до конца ввода

Во многих задачах заранее неизвестно, сколько данных нам предстоит считать. Особенно яркий пример – это обработка текста, когда мы заранее не знаем, сколько строк нам будет введено.

Наиболее удобно работать с такими данными, не пользуясь функцией input, используя методы чтения файла (или ввода с консоли) целиком или построчно.

Рассмотрим простой пример: считать все строки файла input.txt и вывести каждую строку развернутой в файл output.txt (рис. 95).

```
outFile = open('output.txt', 'w', encoding='utf8')
lines = inFile.readlines()
for line in lines:
    print(line[-2::-1], file=outFile)
inFile.close()
outFile.close()
```

Рис. 95. Пример 1 работы с файлами

Для открытия файла используется функция open, принимающая два параметра: имя файла и режим открытия ("r" для чтения и "w" для записи), а также именованный параметр encoding (значение кодировки "utf8" подходит для большинства современных текстовых файлов). Эта функция возвращает ссылку на объект типа файл.

Для чтения всех строк из файла используется метод readlines, который возвращает список всех строк (в смысле lines) файла. Обратите внимание, что строки попадают в список вместе с символом перевода строки, в нашей программе это учитывается при создании среза (этот символ последний в строке). В тестирующей системе все входные файлы имеют перенос строки после последней строки, в реальной жизни это может оказаться не так, и тогда программа будет работать неверно.

Для печати в файл мы пользуемся стандартной функцией print, которой передается именованный параметр file с указанием, в какой файл печатать.

После окончания работы с файлами нужно вызвать для них методы close.

В этой задаче, на самом деле, можно было обойтись без запоминания всего файла в памяти (это особенно актуально для больших файлов). Решение без запоминания всего файла можно было реализовать так, как представлено на рис. 96.

```
inFile = open('input.txt', 'r', encoding='utf8')
outFile = open('output.txt', 'w', encoding='utf8')
for line in inFile:
    print(line[-2::-1], file=outFile)
inFile.close()
outFile.close()
```

Рис. 96. Пример 2 работы с файлами

Переменные типа файл являются iterable и умеют возвращать очередную строку из файла, не храня его целиком в памяти.

Также существует метод read, который позволяет считать все содержимое файла в одну строковую переменную (при этом содержащую в себе переводы строки \n).

В принципе читать до конца ввода можно и из консоли. Для этого нужно подключить библиотеку sys и использовать определенный в ней файловый дескриптор stdin в качестве файла (например, для перебора строк консоли можно написать for line in sys.stdin). Ввести признак конца файла в консоли можно, нажав Ctrl + Z в Windows или Ctrl + D в Unix-системах.

Еще об одном способе работы с файлами с помощью конструкции with ... as ..., что позволяет автоматически вызывать метод close(), можно почитать здесь: http://book.pythontips.com/en/latest/open_function.html.

Сортировка подсчетом

В ряде задач возможные значения в сортируемом списке сильно ограничены. Например, если мы хотим отсортировать оценки от 0 до 10, то может оказаться эффективнее подсчитать, сколько раз встречалась каждая из оценок, и затем вывести её столько раз.

Реализация такого подхода очень проста (рис. 97).

```
1  marks = map(int, input().split())
2  cntMarks = [0] * 11
3  for mark in marks:
4      cntMarks[mark] += 1
5  for nowMark in range(11):
6      print((str(nowMark) + ' ') * cntMarks[nowMark], end='')
```

Рис. 97. Сортировка подсчетом

В этой программе мы создали список, состоящий из 11 нулей в одну строку. Этот приём часто пригождается и в других задачах.

Связь задач поиска и сортировки

Во многих задачах линейного поиска (поиска минимального элемента, например) возникает соблазн воспользоваться сортировкой.

С этим соблазном следует бороться, так как сложность сортировки в языке Питон составляет O(NlogN). То есть для сортировки списка из N элементов нужно совершить порядка NlogN действий.

При этом алгоритмы линейного поиска работают за O(N), что асимптотически быстрее, чем сортировка. Поэтому в задачах линейного поиска (даже для поиска третьего по величине элемента) следует реализовывать линейный поиск, а не пользоваться сортировкой.

Сортировка в интерпретаторе CPython может оказаться быстрее рукописного линейного поиска (из-за того, что она реализована максимально эффективно и на языке Си). Но это досадное недоразумение не должно побороть в вас желание писать линейный поиск руками.

2.2. Структуры данных – множества и словари

Множества и хеш-функции

В языке Питон множества имеют тот же смысл, что и в математике. Это набор объектов без определенного порядка. В множество можно добавлять и удалять объекты, проверять принадлежность объекта множества и перебирать все объекты множества.

Над множествами можно совершать групповые операции, например, пересекать и объединять два множества.

Проверка принадлежности элемента множеству, а также операции удаления и добавления элементов осуществляются за O(1) (если бы мы хранили элементы в списке и хотели бы проверить принадлежность элемента списку, то нам потребовалось бы O(N) операций, где N – длина списка).

Такая скорость достигается использованием хеш-таблиц. Хеш-таблица – это массив достаточно большого размера (назовем этот размер K). Каждому неизменяемому объекту можно сопоставить по некоторому правилу число M от 0 до K и поместить этот объект в ячейку списка с индексом M. Например, для целых чисел таким правилом сопоставления может быть просто подсчет остатка от деления целого числа на K. Операция взятия остатка будет нашей хеш-функцией.

Теперь, если нам нужно проверить, принадлежит ли некоторое число множеству, мы просто считаем хеш-функцию от него

и проверяем, лежит ли наш объект в ячейке с индексом, равным результату вычисления хеш-функции, или нет. Для других типов данных можно применить такой подход: любой объект так или иначе является последовательностью байт. Будем интерпретировать эту последовательность байт как число и подсчитаем хеш-функцию для этого числа.

Может оказаться, что несколько объектов дают один и тот же хеш (отображение между огромным множеством различных объектов и скромным размером множества допустимых хешей не может быть биективным). Такие проблемы можно разрешить, не ухудшая асимптотическую сложность. Подробнее такие методы изучаются в курсе алгоритмов.

Поскольку числа могут быть достаточно длинными, операция подсчета хеш-функции при каждой операции с этим объектом в множестве может быть очень медленной. Поэтому каждый неизменяемый объект в Питоне имеет заранее насчитанный хеш, который подсчитывается один раз при его создании. Кстати, с помощью этих же хешей можно понимать, есть ли уже объект в памяти, и не создавать новых объектов, а просто подвешивать еще одну ссылку на уже существующий объект.

Изменяемые типы, такие как список, не имеют заранее насчитанных хешей. Изменение всего одного элемента в списке привело бы к полному пересчету хеша для всего списка, что катастрофически замедлило бы работу со списками. Поэтому у изменяемых объектов нет хеша, и они не могут быть добавлены в множество.

Само множество также является изменяемым объектом и не может быть, например, элементом другого множества.

Существуют также неизменяемые множества, которые создаются с помощью функции frozenset.

Создание множеств

Множество в теле программы может быть создано с помощью записи элементов через запятую в фигурных скобках (рис. 98).

Вывод с помощью print осуществляется в том же формате. Порядок элементов в множестве может быть случайным, так как хеш-функция не гарантирует, что если $A > B$, то $h(A) > h(B)$.

```
1  mySet = {3, 1, 2}
2  print(mySet)
```

Рис. 98. Пример 1 работы с множествами

Если при задании множества присутствовало несколько одинаковых элементов, то они попадут в множество в единственном экземпляре (рис. 99).

```
1  firstSet = {1, 2, 1, 3}
2  secondSet = {3, 2, 1}
3  print(firstSet == secondSet)
```

Рис. 99. Пример 2 работы с множествами

Эта программа выведет True (множества можно сравнивать на равенство).

Множества можно создавать также с помощью функции set, которая может принимать в качестве параметра что угодно итерируемое (рис. 100).

```
1  setFromList = set([1, 2, 3])
2  print(setFromList)
3  setFromTuple = set((4, 5, 6))
4  print(setFromTuple)
5  setFromStr = set("lol")
6  print(setFromStr)
7  setFromRange = set(range(2, 22, 3))
8  print(setFromRange)
9  setFromMap = set(map(abs, (1, 2, 3, -2, -4)))
10 print(setFromMap)
11 setFromSet = set({1, 2, 3})
12 print(setFromSet)
```

Рис. 100. Пример 3 работы с множествами

Вывод этой программы такой:

{1, 2, 3}

{4, 5, 6}

{‘l’, ‘o’}

{2, 5, 8, 11, 14, 17, 20}

{1, 2, 3, 4}
{1, 2, 3}

Множество также является итерируемым объектом (еще раз: объекты идут в случайном порядке, не по возрастанию!).

Множество может содержать в себе объекты разных типов (рис. 101).

```
1  mixedSet = {1, 3.14, (1, 2, 3), "i have no idea why i'm here"}
2  print(mixedSet)
```

Рис. 101. Пример 4 работы с множествами

По аналогии со строками, списками и кортежами количество элементов в множестве можно узнать с помощью функции len.

Из множества можно сделать список или кортеж с помощью функций list и tuple соответственно. Применение функции str к множеству даст нам текстовое представление (элементы в фигурных скобках, разделенные запятыми).

Частой операцией является вывод упорядоченных элементов множества. Это можно сделать, применив функцию sorted сразу к множеству (ведь оно итерируемо) (рис. 102).

```
1  mySet = {'abba', 'a', 'long string'}
2  print(', '.join(mySet))
3  print(', '.join(sorted(mySet)))
```

Рис. 102. Пример 5 работы с множествами

Вывод этой программы будет:
long string, a, abba
a, abba, long string
К множеству можно применять функцию map.

Работа с множествами

Работа с элементами множеств

Чтобы создать пустое множество, нужно воспользоваться кодом, показанным на рис. 103.

```
1  emptySet = set()
```

Рис. 103. Пример 6 работы с множествами

Писать пустые фигурные скобки нельзя (позднее узнаем почему).

Добавление элемента в множество осуществляется с помощью метода add. Если элемент уже был в множестве, то оно не изменится.

Перебрать элементы множества можно с помощью for (for умеет ходить по любым итерируемым объектам) (рис. 104).

```
1  mySet = {1, '2', 2, '1'}
2  for elem in mySet:
3      print(elem, end=' ')
```

Рис. 104. Пример 7 работы с множествами

Вывод такой программы будет «1 1 2 2», но упорядоченность является чистой случайностью.

Чтобы проверить, входит ли элемент X в множество A, достаточно написать X in A. Результатом этой операции будет True или False. Чтобы проверить, что элемент не лежит в множестве, можно писать not X in A или, логичнее, X not in A (рис. 105).

```
1  mySet = {1, 2, 3}
2  if 1 in mySet:
3      print('1 in set')
4  else:
5      print('1 not in set')
6  x = 42
7  if x not in mySet:
8      print('x not in set')
9  else:
10     print('x in set')
```

Рис. 105. Пример 8 работы с множествами

Вывод этой программы будет:

1 in set

x not in set

Чтобы удалить элемент из множества, можно воспользоваться одним из двух методов: discard или remove. Если удаляемого элемента в множестве не было, то discard не изменит состояния множества, а remove выпадет с ошибкой.

Групповые операции над множествами

В Питоне можно работать не только с отдельными элементами множеств, но и с множествами в целом. Так, для множеств определены некоторые операции (табл. 5).

Таблица 5

Операции для множеств

Операция	Описание
A | B	Объединение множеств
A & B	Пересечение множеств
A – B	Множество, элементы которого входят в A, но не входят в B
A ^ B	Элементы входят в A | B, но не входят в A & B

В результате этих операций создается новое множество, однако для них определена и сокращенная запись: |=, &=, -= и ^=. Такие операции изменяют множество, находящееся слева от знака операции.

Для множеств также определены операции сравнения (табл. 6).

Таблица 6

Операции сравнения

Операция	Описание
A == B	Все элементы совпадают
A != B	Есть различные элементы
A <= B	Все элементы A входят в B
A < B	A <= B и A != B

Также определены операции > и >=. Все групповые операции и сравнения проводятся над множествами за время, пропорциональное количеству элементов в множествах.

Словари

В жизни нередко возникает необходимость сопоставить ключ и его значение. Например, в англо-русском словаре английскому слову соответствует одно или несколько русских слов. Здесь английское слово является ключом, а русское – значением.

В языке Питон есть структура данных словаря, которая позволяет реализовывать подобные операции. При этом объекты-ключи уникальны, и с каждым из них сопоставлено некоторое объект-значение. Ограничения на ключи такие же, как на элементы множества, а вот значения могут быть и изменяемыми.

По сути, словарь является множеством, где с каждым элементом-ключом сопоставлено еще и объект-значение.

Создать словарь в исходном тексте программы можно, записав в фигурных скобках пары «ключ – значение» через запятую, а внутри пары ключ отделяется от значения двоеточием (рис. 106).

```
1  countries = {'Russia' : 'Europe', 'Germany' : 'Europe', 'Australia' :
     'Australia'}
```

Рис. 106. Пример 1 работы со словарями

Добавлять пары «ключ – значение» в словарь очень просто: это делается по аналогии со списками (рис. 107).

```
1  sqrs = {}
2  sqrs[1] = 1
3  sqrs[2] = 4
4  sqrs[10] = 100
5  print(sqrs)
```

Рис. 107. Пример 2 работы со словарями

Пустой словарь можно создать, написав пустые фигурные скобки (это будет словарь, а не множество).

Словарь также можно конструировать из других объектов с помощью функции dict (рис. 108).

```
1  myDict = dict([['key1', 'value1'], ('key2', 'value2')])
2  print(myDict)
```

Рис. 108. Пример 3 работы со словарями

На вход функции нужно подавать iterable, каждый элемент которого, в свою очередь, является iterable строго с двумя элементами – ключом и значением.

Узнавать значение по ключу можно также с помощью записи ключа после имени словаря в квадратных скобках (рис. 109).

```
1  phones = {'police' : 102, 'ambulance' : 103, 'firefighters' : 101}
2  print(phones['police'])
```

Рис. 109. Пример 4 работы со словарями

Если такого ключа в словаре нет, то возникнет ошибка.

Удаление элемента из словаря делается специальной командой del. Это не функция. После слова del ставится пробел, затем пишется имя словаря, а затем в квадратных скобках – удаляемый ключ (рис. 110).

```
1  phones = {'police' : 102, 'ambulance' : 103, 'firefighters' : 101}
2  del phones['police']
3  print(phones)
```

Рис. 110. Пример 5 работы со словарями

Проверка принадлежности ключа словарю осуществляется с помощью операции key in dictionary (точно так же, как проверка принадлежности элемента множеству).

Словарь является iterable и возвращает ключи в случайном порядке. Например, код, представленный на рис. 111, напечатает содержимое словаря.

```
1  phones = {'police' : 102, 'ambulance' : 103, 'firefighters' : 101}
2  for service in phones:
3      print(service, phones[service])
```

Рис. 111. Пример 6 работы со словарями

Существует метод items, который возвращает iterable, содержащий в себе кортежи «ключ – значение» для всевозможных ключей (рис. 112).

```
phones = {'police' : 102, 'ambulance' : 103, 'firefighters' : 101}
for service, phone in phones.items():
    print(service, phone)
```

Рис. 112. Пример 7 работы со словарями

Когда нужно использовать словари

Словари используют:

– по прямому назначению: сопоставление ключа значению (названия дней недели, переводы слов и т. д.);

– для подсчета числа объектов. При очередной встрече объекта счетчик увеличивается на единицу. Это похоже на сортировку подсчётом;

– для хранения разреженных массивов. Например, если мы хотим хранить цену 92, 95 и 98 бензина, то могли бы создать массив из 99 элементов и хранить в нём. Но большая часть элементов не нужны – массив разреженный. Словарь здесь подходит больше.

Для подсчета числа элементов удобно использовать метод get. Он принимает два параметра: ключ, для которого нужно вернуть значения, и значение, которое будет возвращено, если такого ключа нет. Например, подсчитать, сколько раз входит в последовательность каждое из чисел, можно с помощью кода на рис. 113.

```
seq = map(int, input().split())
countDict = {}
for elem in seq:
    countDict[elem] = countDict.get(elem, 0) + 1
for key in sorted(countDict):
    print(key, countDict[key], sep=' : ')
```

Рис. 113. Пример использования словарей

Полезные методы строк

isalpha – проверяет, что все символы строки являются буквами.

isdigit – проверяет, что все символы строки являются цифрами.

isalnum – проверяет, что все символы строки являются буквами или цифрами.

islower – проверяет, что все символы строки являются маленькими (строчными) буквами.

isupper – проверяет, что все символы строки являются большими (заглавными, прописными) буквами.

lstrip – обрезает все пробельные символы в начале строки.

rstrip – обрезает все пробельные символы в конце строки.

strip – обрезает все пробельные символы в начале и конце строки.

Пример решения сложной задачи на словари

Рассмотрим такую задачу: словарь задан в виде набора строк, в каждой строке записано слово на английском языке, затем следует символ -, затем через запятую перечислены возможные переводы слова на латынь.

Требуется составить латино-английский словарь и вывести его в том же виде. Все слова должны быть упорядочены по алфавиту. Возможные переводы одного слова должны быть также упорядочены по алфавиту.

На рис. 114 представлен набор данных для ввода.

```
3
apple - malum, pomum, popula
fruit - baca, bacca, popum
punishment - malum, multa
```

Рис. 114. Входной набор данных

Вывод должен выглядеть так, как представлено на рис. 115.

```
7
baca - fruit
bacca - fruit
malum - apple, punishment
multa - punishment
pomum - apple
popula - apple
popum - fruit
```

Рис. 115. Требуемый выходной набор данных

Идея решения заключается в следующем: разрежем каждую строку на английское и латинские слова. Каждое из латинских слов возьмем в качестве ключа и добавим к его значениям английское слово (переводов может быть несколько). Затем пройдем по сортированным ключам и для каждого ключа выведем отсортированный список переводов (рис. 116).

```
n = int(input())
latinEnglish = {}
for i in range(n):
    line = input()
    english = line[:line.find('-')].strip()
    latinsStr = line[line.find('-') + 1:].strip()
    latins = map(lambda s : s.strip(), latinsStr.split(','))
    for latin in latins:
        if latin not in latinEnglish:
            latinEnglish[latin] = []
        latinEnglish[latin].append(english)
print(len(latinEnglish))
for latin in sorted(latinEnglish):
    print(latin, '-', ', '.join(sorted(latinEnglish[latin])))
```

Рис. 116. Программный код для решения задачи

2.3. Использование функций языка Python для обработки последовательностей

Парадигмы программирования и функциональное программирование

Языки программирования предлагают различные средства для декомпозиции задачи. Существует несколько парадигм программирования:

– императивное (структурное, процедурное) программирование: программы являются последовательностью инструкций, которые могут читать и записывать данные из памяти. При изучении предыдущих тем мы пользовались в основном императивной парадигмой. Ряд языков, такие как Паскаль (не Object Pascal) или C, являются яркими примерами императивных языков;

– декларативное программирование: описывается задача и ожидаемый результат, но не описываются пути её решения. Ярким при-

мером является язык запросов к базам данных SQL: большая часть внутреннего устройства скрыта в СУБД, программист описывает только структуру базы данных и ожидаемый результат запросов;

– объектно ориентированное программирование: программы манипулируют наборами объектов, при этом объекты обладают сохраняющимся во времени состоянием и методами для изменения этого состояния (или создания новых объектов). Мы ознакомимся с ООП подробнее в дальнейшем. Примером языка с объектно ориентированной парадигмой является Java, ООП также поддерживается в Питоне и C++;

– функциональное программирование: задача разбивается на набор функций. В идеале функции только принимают параметры и возвращают значения, не изменяя состояния объектов или программы. Примером функциональных языков является Haskell.

Иногда выделяют и другие парадигмы программирования.

Некоторые языки предназначены для написания программ в основном в рамках одной парадигмы, другие же поддерживают несколько парадигм. Например, Питон и C++ поддерживают различные парадигмы программирования. Разные части программы можно писать в разных парадигмах. Например, можно использовать функциональный стиль для обработки больших данных, объектно ориентированный подход для реализации интерфейса и императивное программирование для промежуточной логики.

Функциональное программирование

Несмотря на то, что в функциональном стиле писать достаточно сложно и непривычно, функциональное программирование имеет массу плюсов:

– достаточно легко доказать формальную корректность алгоритмов. Хотя доказательство, даже для функциональных программ, часто намного длиннее, чем сама программа, но для фундаментальных алгоритмов, которые широко используются, лучше иметь формальное доказательство. Строить формальные доказательства императивных программ намного сложнее;

– для программ, написанных в функциональном стиле, легко проводить декомпозицию, отладку и тестирование. Когда вся про-

грамма разбита на функции, выполняющие элементарные действия, их разработка и проверка занимают намного меньше времени;

– для функциональных программ легко автоматически проводить распараллеливание, векторизацию и конвейеризацию.

Задача распараллеливания выполнения программ крайне актуальна сейчас, когда скорости ядер процессоров практически перестали расти. Единственным способом ускорить выполнение программ является их параллельное вычисление на нескольких устройствах.

Последние успехи в решении задач машинного обучения связаны с использованием большого количества простых вычислительных устройств, например расчетов на видеокарте.

В функциональных программах не принято вносить изменения в объекты (и, вообще говоря, желательно, чтобы все объекты были неизменяемыми). Поэтому, если нам нужно посчитать результат вычисления двух функций, во многих случаях можно делать это параллельно на разных ядрах процессора. Такое распараллеливание легко сделать автоматически.

Векторизация – выполнение одних и тех же действий над большим набором данных. Тогда данные можно нарезать на куски и раскидать по разным вычислительным устройствам, а затем собрать результат вычислений в один объект.

Конвейеризация – это разбиение вычислений на несколько этапов, причём данные поступают на следующий этап обработки по времени готовности. Например, если нам нужно попарно перемножить значения в векторах A и B, а затем сложить со значениями в векторе C, то мы можем посчитать несколько первых результатов подсчета произведения и выполнять с ними сложение, не дожидаясь подсчета остальных значений. Это позволяет быстрее получить первые результаты и занять еще большее количество вычислительных устройств параллельно.

Встроенные функции для работы с последовательностями

В языке Питон есть много функций, которые принимают в качестве параметра iterable и могут сделать что-то полезное. С некоторыми из них, такими как sorted или map, мы уже немного знакомы. Рассмотрим еще некоторые из них.

sum – находит сумму всех элементов iterable.

min, max – находит минимум и максимум в последовательности iterable.

map – умеет принимать более двух параметров. Например, запись map(f, iterA, iterB) вернет iterable со значениями f(iterA[0], iterB[0]), f(iterA[1], iterB[1]), ...

filter(predicate, iterable) – применяет функцию predicate ко всем элементам iterable и возвращает iterable, который содержит только те элементы, которые удовлетворяли предикаты (то есть функция predicate вернула True). Например, решение задачи о поиске минимального положительного элемента в списке может выглядеть так, как представлено на рис. 117.

```
1  print(min(filter(lambda x: x > 0, map(int, input().split()))))
```

Рис. 117. Использование функции filter

Здесь в качестве предиката использована лямбда-функция, которая возвращает True при значении параметра больше 0, а в качестве входного iterable – результат вызова map для функции int и нарезанного на слова ввода. Функция min применена к тому iterable, который был возвращен функцией filter.

enumerate – возвращает кортежи из номера элемента (при нумерации с нуля) и значения очередного элемента. С помощью enumerate, например, удобно перебирать элементы iterable (доступ по индексу в которых невозможен) и выводить номера элементов, которые обладают некоторым свойством (рис. 118).

```
1  f = open('data.txt', 'r', encoding='utf8')
2  for i, line in enumerate(f):
3      if line.strip() == '':
4          print('Blank line at line', i)
```

Рис. 118. Использование функции enumerate

any, all – возвращают истину, если хотя бы один или все элементы iterable истинны соответственно. Например, так, как показано

на рис. 119, можно проверить, не превышают ли все члены последовательности 100 по модулю.

```
print(all(map(lambda x: abs(int(x)) <= 100, input().split())))
```

Рис. 119. Использование функции all

zip(iterA, iterB, ...) – конструирует кортежи из элементов (iterA[0], iterB[0], ...), (iterA[1], iterB[1], ...), ...

Пример решения сложной задачи в функциональном стиле

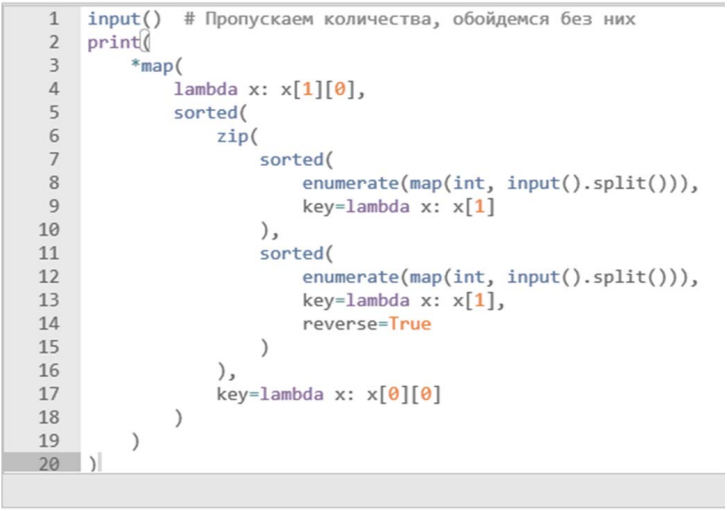
С помощью этих функций можно решить достаточно сложные задачи без использования циклов и условных операторов. Например, задачу по сортировкам про такси: в первой строке задано количество людей и автомобилей такси, в следующих двух строках расстояние в километрах для каждого человека и цена за километр для каждого такси. Необходимо сопоставить каждого человека и номер такси так, чтобы суммарная цена поездок была минимальна. Идея решения заключается в том, что люди, которым ехать дальше, должны ехать на более дешевых такси (рис. 120).

```
input()  # Пропускаем количества, обойдемся без них
people = map(int, input().split())
sortedPeople = sorted(enumerate(people), key=lambda x: x[1])
taxi = map(int, input().split())
sortedTaxi = sorted(enumerate(taxi), key=lambda x: x[1], reverse=True)
ans = zip(sortedPeople, sortedTaxi)
sortedAns = sorted(ans, key=lambda x: x[0][0])
print(*map(lambda x: x[1][0], sortedAns))
```

Рис. 120. Исходная программа

А теперь избавимся от переменных, подставляя на их места исходные значения. Сделаем переносы для удобного восприятия (рис. 121).

Это та же программа!



```
input()  # Пропускаем количества, обойдемся без них
print(
    *map(
        lambda x: x[1][0],
        sorted(
            zip(
                sorted(
                    enumerate(map(int, input().split())),
                    key=lambda x: x[1]
                ),
                sorted(
                    enumerate(map(int, input().split())),
                    key=lambda x: x[1],
                    reverse=True
                )
            ),
            key=lambda x: x[0][0]
        )
    )
)
```

Рис. 121. Функциональный стиль написания программы

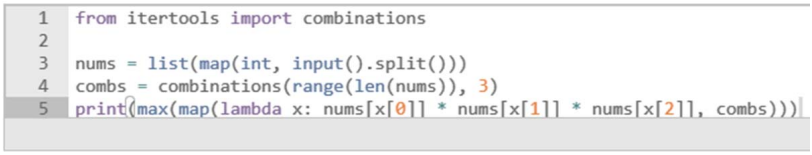
itertools, functools

Генерация комбинаторных объектов itertools

В Питоне есть библиотека itertools, которая содержит много функций для работы с итерируемыми объектами. С этими функциями можно ознакомиться в официальной документации к языку.

Нам наиболее интересны функции, генерирующие комбинаторные объекты.

itertools.combinations(iterable, size) генерирует все подмножества множества iterable размером size в виде кортежей. Это может быть использовано вместо вложенных циклов при организации перебора. Например, мы можем неэффективно решить задачу о поиске трех чисел в последовательности, дающих наибольшее произведение (рис. 122).



```
from itertools import combinations

nums = list(map(int, input().split()))
combs = combinations(range(len(nums)), 3)
print(max(map(lambda x: nums[x[0]] * nums[x[1]] * nums[x[2]], combs)))
```

Рис. 122. Генерация комбинаторных объектов

itertools.permutations(iterable) генерирует все перестановки iterable. Существует вариант функции с двумя параметрами, второй параметр является размером подмножества. Тогда генерируются все перестановки всех подмножеств заданного размера.

itertools.combinations_with_replacement(iterable, size) генерирует все подмножества iterable размером size с повторениями, то есть одно и то же число может входить в подмножество несколько раз.

Модуль functools и функции partial, reduce, accumulate

Модуль functools содержит некоторые функции, которые могут быть полезны для обработки последовательностей и не только.

Функция functools.partial предназначена для оборачивания существующих функций с подстановкой некоторых параметров. Например, мы можем создать функцию для печати в файл, чтобы каждый раз не указывать какие-то параметры. Например, существует вариант функции int с двумя параметрами: первый – это переменная, которую необходимо преобразовать в число, второй – система счисления, в которой записано число. С помощью partial можем создать функцию-обёртку, преобразующую строки из 0 и 1 в числа (рис. 123).

```
1  from functools import partial
2
3  binStrToInt = partial(int, base=2)
4  print(binStrToInt('10010'))
```

Рис. 123. Пример 1 использования функций модуля functools

В модуле functools также содержатся функции для обработки последовательностей.

functools.reduce(func, iterable) позволяет применить функцию ко всем элементам последовательности, используя в качестве первого аргумента накопленный результат. Например, если в последовательности были элементы myList = [A, B, C], то результатом применения reduce(f, myList) будет f(f(A, B), C). С помощью reduce, например, можно найти НОД всех чисел в iterable (рис. 124).

```
from functools import reduce

def gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a

print(reduce(gcd, map(int, input().split())))
```

Рис. 124. Пример 2 использования функций модуля functools

itertools.accumulate(iterable, func) возвращает iterable со всеми промежуточными значениями, то есть для списка [A, B, C] accumulate вернет значения A, f(A, B), f(f(A, B), C). Например, можно узнать максимальный элемент для каждого префикса (некоторого количества первых элементов) заданной последовательности (рис. 125).

```
from itertools import accumulate

print(*accumulate(map(int, input().split()), max))
```

Рис. 125. Пример 3 использования функций модуля functools

Итераторы и генераторы

В Питоне итераторы – это объекты, которые имеют внутреннее состояние и метод __next__ для перехода к следующему состоянию. Например, можно сконструировать итератор от списка и перебрать все значения (рис. 126).

```
myList = [1, 2, 3]
for i in iter(myList):
    print(i)
for i in myList:
    print(i)
```

Рис. 126. Пример использования итератора

В этой программе два цикла эквивалентны.

В Питоне можно создавать и свои итераторы. Их разумно использовать в том случае, если нужно перебрать большое количество значений. Существует правило, по которому можно получить

следующее значение, однако хранение всех этих значений не имеет смысла, так как они пригодятся только один раз.

Для создания итераторов в Питоне используется специальный вид функций, называемых генераторами. В обычной функции return прекращает работу функции. В генераторе вместо return используется оператор yield, который также возвращает значение, но не прекращает выполнение функции, а приостанавливает его до тех пор, пока не потребуется следующее значение итератора. При этом работа функции продолжится с того места и в том состоянии, в котором она находилась на момент вызова yield. Посмотрим, как может быть реализован генератор, аналогичный стандартному range с одним параметром (рис. 127).

```
1  def myRange(n):
2      i = 0
3      while i < n:
4          yield i
5          i += 1
6  for i in myRange(10):
7      print(i)
```

Рис. 127. Пример 1 использования генератора

Генераторы могут иметь и сложную рекурсивную структуру. Например, мы можем написать генератор, который будет выдавать все числа заданной длины, цифры в которых не убывают и старшая цифра не превосходит заданного параметра (рис. 128).

```
1  def genDecDigs(cntDigits, maxDigit):
2      if cntDigits > 0:
3          for nowDigit in range(maxDigit + 1):
4              for tail in genDecDigs(cntDigits - 1, nowDigit):
5                  yield nowDigit * 10**(cntDigits - 1) + tail
6      else:
7          yield 0
8
9  print(*genDecDigs(2, 3))
```

Рис. 128. Пример 2 использования генератора

Вывод этой программы будет выглядеть так: 0 10 11 20 21 22 30 31 32 33.

В этой программе рекурсивный генератор перебирал все допустимые цифры в качестве той, которая должна стоять на заданной позиции, и генерировал все возможные последующие цифры.

В этой программе мы использовали также одну особенность функции print: если перед именем iterable (а результатом, возвращаемым генератором, является iterable) поставить *, то будут напечатаны все значения через пробел.

Результат работы генератора можно сохранить, например, в список с помощью функции list, как мы уже делали это с результатом работы range.

2.4. Объектно ориентированное программирование при анализе данных

Объектно ориентированное программирование

Объектно ориентированное программирование (ООП) является одной из парадигм программирования, созданной на базе императивного программирования для более удобного повторного использования кода и облегчения читаемости программ.

ООП не является «серебряной пулей», которая решает все задачи наиболее удобным образом. Императивное программирование удобно для решения простых задач, использующих стандартные объекты. Повторное использование кода в императивной парадигме обеспечивается с помощью циклов и функций, написанных в императивном стиле.

Функциональное программирование удобно для задач, где существует ярко выраженный поток данных (data flow), который «перетекает» из одной функции в другую, изменяясь по пути.

ООП же позволяет обеспечить повторное использование кода за счет того, что обрабатываемые программой объекты имеют много общего и лишь незначительные отличия в своем поведении.

ООП основано на трёх концепциях: инкапсуляции, наследовании, полиморфизме.

Инкапсуляция – это помещение в «капсулу»: логическое объединение данных и функций для работы с ними, а также сокрытие внутреннего устройства объекта с предоставлением интерфейса взаимодействия с ним (публичные методы).

Наследование – это получение нового типа объектов на основе уже существующего с частично или полностью заимствованной у родительского типа функциональностью.

Полиморфизм – это предоставление одинаковых средств взаимодействия с объектами разной природы. Например, операция + может работать как с числами, так и со строками, несмотря на разную природу этих объектов.

Классом в Питоне называется описание структуры объекта (полей структуры) и методов (функций) для работы с данными в этой структуре.

Объектом называется экземпляр класса, где поля заполнены конкретными значениями. Объекты находятся в памяти программы и могут изменять своё состояние или выполнять какие-то действия с помощью вызова методов класса для этого объекта.

Инкапсуляция и конструкторы

Мы уже использовали ключевое слово class для создания структур – набора именованных полей, совокупность которых описывает объект. Однако мы пользовались для их обработки отдельно лежащими функциями или кусками кода.

Было бы намного удобнее, если бы описание структуры объекта и методов работы с ним лежало рядом для удобства изучения, модификации и использования.

Мы будем рассматривать элементы ООП на примере компле́ксных (ударение на «е») чисел. Это забавный математический объект, который состоит из действительной (real) и мнимой (imaginary) части. Запись этого числа выглядит как re + im * i, где i – это квадратный корень из –1. Глубокое математическое понимание комплексных чисел нам не понадобится. Достаточно понимать, что это структура с двумя полями re и im, где оба эти поля – вещественные числа.

Для создания новых объектов класса используется специальный метод, который называется «конструктор». Методы класса записываются внутри описания класса как функции, конструктор должен называться __init__. В качестве первого параметра они должны принимать переменную self – конкретный объект класса, с которым они работают.

Рассмотрим класс для комплексного числа, вызов конструктора и печать полей объекта (рис. 129).

```
1  class Complex:
2      def __init__(self, re=0, im=0):
3          self.re = re
4          self.im = im
5
6  a = Complex(1, 2)
7  b = Complex(3)
8  c = Complex()
9  print(a.re, a.im)
10 print(b.re, b.im)
11 print(c.re, c.im)
```

Рис. 129. Задание конструктора класса

Здесь конструктор содержит три параметра: self – пустой объект класса Complex, re и im, по умолчанию равные нулю. Вызов конструктора осуществляется с помощью написания названия класса, в скобках указываются параметры конструктора (все, кроме self). Названия классов принято записывать с большой буквы, а объекты – с маленькой.

Вывод этой программы будет:

1 2

3 0

0 0

Обратите внимание, что мы меняем переменные конкретного объекта класса, который передан в качестве параметра self. Если бы мы создали какие-то переменные в описании класса, то их значения были бы доступны во всех объектах этого класса и их можно было бы даже изменить, перечислив их имена в начале метода после выражения nonlocal. Такие переменные называются статическими, обычно они предназначены для хранения каких-то констант (что очень удобно, например, при описании какого-то класса для физических вычислений). Их изменение может понадобиться, например, в экзотических ситуациях при подсчете количества объектов класса.

Определение методов и стандартные функции

Некоторые стандартные функции языка Питон являются всего лишь обертками над вызовом метода для передаваемого параметра. Например, функция str вызывает метод __str__ для своего параметра. Если мы опишем такой метод для нашего класса, то можно будет применять к нему функцию str явно и неявно (например, она автоматически вызовется при вызове print для объекта нашего класса).

Мы бы хотели, чтобы __str__ возвращал текстовое представление нашего комплексного числа. Например, число с действительной частью 1 и мнимой 2 должно быть представлено в виде строки "1+2i", а число с действительной частью 3 и мнимой –4.5 – как "3-4.5i". Полное описание класса с добавленным методом будет выглядеть так, как представлено на рис. 130.

```
1  class Complex:
2      def __init__(self, re=0, im=0):
3          self.re = re
4          self.im = im
5      def __str__(self):
6          strRep = str(self.re)
7          if self.im >= 0:
8              strRep += '+'
9          strRep += str(self.im) + 'i'
10         return strRep
11
12 a = Complex(1, 2)
13 print(a)
14 b = Complex(3, -4.5)
15 print(b)
```

Рис. 130. Определение методов и стандартные функции

Переопределение операторов

В языке Питон можно переопределить и поведение операторов. Например, если у нас есть два числа x и y, то запись x + y реально преобразуется в вызов метода x.__add__(y). Значок операции + является всего лишь удобным для человека переопределением вызова метода add.

Для комплексных чисел логично определена операция сложения: это сложение отдельно действительных и отдельно мнимых

частей. В результате вызова метода для сложения двух чисел должен конструироваться новый объект класса Complex, а переданные в качестве параметров объекты не должны изменяться. Действительно, когда мы выполняем операцию z = x + y для обычных чисел, то ожидаем, что сконструируется новый объект, к которому привяжется ссылка z, а x и y останутся без изменения.

Будем придерживаться этой же логики при реализации метода для сложения двух комплексных чисел. Наш метод __add__ должен принимать два параметра, каждый из которых является комплексным числом (рис. 131).

```
1  class Complex:
2      def __init__(self, re=0, im=0):
3          self.re = re
4          self.im = im
5      def __str__(self):
6          strRep = str(self.re)
7          if self.im >= 0:
8              strRep += '+'
9          strRep += str(self.im) + 'i'
10         return strRep
11     def __add__(self, other):
12         newRe = self.re + other.re
13         newIm = self.im + other.im
14         return Complex(newRe, newIm)
15
16 a = Complex(1, 2)
17 b = Complex(3, -4.5)
18 print(a + b)
```

Рис. 131. Переопределение операторов

Переопределять метод add имеет смысл только в тех ситуациях, когда программисту, использующему ваш класс, будет очевиден смысл операции +. Например, если бы вы создали класс для описания некоторых характеристик человека, то операция + для двух объектов-людей воспринималась бы разными пользователями вашего класса совершенно по-разному, в зависимости от развитости фантазии читателя. Такого неоднозначного понимания лучше избегать и вовсе не переопределять операцию +, если результат её работы не очевиден.

Проверка класса объекта

Переопределим для комплексных чисел ещё одну операцию – умножение. При этом мы хотим уметь умножать комплексные числа как на целые или действительные, так и на другие комплексные числа.

При умножении комплексного числа вида a + b * i$ на целое или вещественное число x результатом будет комплексное число a * x + b * x * b.

Перемножение двух комплексных чисел производится аналогично умножению двух многочленов первой степени:

(a + b * i) * (c + d * i) = a * c + (a * d + b * c) * i + b * d * i**2.

Мы знаем, что i**2 = –1. Значит, окончательно результат умножения будет выглядеть так:

a * c – b * d + (a * d + b * c) * i.

Нам осталось понять, как определить, передано ли в наш метод комплексное или не комплексное число.

В языке Питон существует функция isinstance, которая в качестве первого параметра принимает объект, а в качестве второго – название класса. Она возвращает истину, если объект относится к данному классу, и ложь в противном случае. Эта функция позволит нам добиться нужной функциональности от метода __mul__, умножающего числа (рис. 132).

Кроме добавленного метода __mul__, внимания также заслуживает строка __rmul__ = __mul__. Это присваивание одного метода (функции) другому, то есть при вызове метода __rmul__ будет вызываться тот же самый метод __mul__.

В языке Питон операция a * b заменяется на вызов метода a.__mul__(b). Если a было комплексным числом, а b – вещественным, то вызовется метод __mul__ для объекта a нашего класса Complex.

Однако, если a было вещественным числом, а b – комплексным, то произойдет попытка вызвать метод __mul__ для объекта класса float. Естественно, разработчики стандартной библиотеки языка Питон не могли предположить, что мы когда-нибудь напишем класс Complex и будем пытаться умножить на него веще-

ственное число. Поэтому метод __mul__, где в качестве параметра передается нечто неизвестное, будет заканчивать свою работу с ошибкой. Чтобы избежать таких ситуаций, в языке Питон после неудачной попытки совершить a.__mul__(b) происходит попытка совершить действие b.__rmul__(a), и в нашем случае она заканчивается успехом.

```
class Complex:
    def __init__(self, re=0, im=0):
        self.re = re
        self.im = im
    def __str__(self):
        strRep = str(self.re)
        if self.im >= 0:
            strRep += '+'
        strRep += str(self.im) + 'i'
        return strRep
    def __add__(self, other):
        newRe = self.re + other.re
        newIm = self.im + other.im
        return Complex(newRe, newIm)
    def __mul__(self, other):
        if isinstance(other, Complex):
            newRe = self.re * other.re - self.im * other.im
            newIm = self.re * other.im + self.im * other.re
        elif isinstance(other, int) or isinstance(other, float):
            newRe = self.re * other
            newIm = self.im * other
        return Complex(newRe, newIm)
    __rmul__ = __mul__

a = Complex(1, 2)
b = Complex(3, -4.5)
print(a * b)
print(a * 2)
```

Рис. 132. Проверка класса объектов

Обработка ошибок

Время от времени в программах возникают ошибочные ситуации, которые не могут быть обработаны в том месте, где возникла ошибка, а должны быть обработаны тем или иным образом в иной части программы.

Например, если на этапе выполнения промежуточной логики вы пытаетесь записать строку в численную переменную, вы не

сможете этого сделать. В таком случае нужно выходить из стека вызовов функций или методов промежуточной логики до тех пор, пока не дойдем до фронтенда, который сообщит пользователю о том, что он ввел недопустимое значение, и попросит, например, ввести его заново.

Иногда в ситуации заполнения огромной формы попытка отправить ее приводит к тому, что появляется окошко со словом «ошибка» без каких-либо уточнений. Это неправильный подход, сообщение об ошибке должно быть информативным, чтобы позволить быстро её найти и исправить. Таким образом, при создании ошибки нужно передавать исчерпывающую информацию о ней.

В примере с вещественными числами можно рассмотреть такую ошибку: умножение комплексного числа на что-то, отличное от целого, вещественного или комплексного числа. На этапе умножения уже ничего нельзя предпринять для исправления этой ошибки, кроме как просигнализировать о ней в то место, откуда была вызвана операция умножения.

При этом на этапе вызова операции умножения уже нельзя предпринять какие-то действия. Например: попросить пользователя ввести все-таки комплексное число или при обработке последовательности, из которой нужно вычленить и перемножить комплексные числа, – просто перейти к следующему элементу последовательности. На этапе неудачного выполнения операции умножения мы не знаем и не можем знать, как должна вести себя конкретная программа при возникновении такой ошибки.

При ошибочной операции мы можем сконструировать специальный класс, содержащий подробное описание ошибки, и использовать его в том месте программы, которое способно его обработать.

Выбрасывается ошибка с помощью команды raise, а ловится блоком try-except. Для случая умножения комплексного числа на некорректное значение можно сконструировать класс ошибки, содержащий в себе ссылку как на комплексное число, так и на второй аргумент метода умножения.

Класс для ошибки должен быть наследником стандартного класса BaseException. Пока для нас это значит только то, что при

создании описания класса ошибки нужно написать в скобках после его названия BaseException. Потенциально ошибочные действия должны выполняться в блоке try, а команды для обработки ошибки должны быть в блоке except. Пример с обработкой ошибки будет выглядеть так, как это представлено на рис. 133.

```
class ComplexError(BaseException):
    def __init__(self, Complex, other):
        self.arg1 = Complex
        self.arg2 = other

class Complex:
    def __init__(self, re=0, im=0):
        self.re = re
        self.im = im
    def __str__(self):
        strRep = str(self.re)
        if self.im >= 0:
            strRep += '+'
        strRep += str(self.im) + 'i'
        return strRep
    def __add__(self, other):
        newRe = self.re + other.re
        newIm = self.im + other.im
        return Complex(newRe, newIm)
    def __mul__(self, other):
        if isinstance(other, Complex):
            newRe = self.re * other.re - self.im * other.im
            newIm = self.re * other.im + self.im * other.re
        elif isinstance(other, int) or isinstance(other, float):
            newRe = self.re * other
            newIm = self.im * other
        else:
            raise ComplexError(self, other)
        return Complex(newRe, newIm)
    __rmul__ = __mul__

a = Complex(1, 2)
try:
    res = a * 'abcd'
except ComplexError as ce:
    print('Error in mul with args:', ce.arg1, ce.arg2)
```

Рис. 133. Пример реализации обработки ошибок

Вывод этой программы:

Error in mul with args: 1+2i abcd.

По нему легко понять, что ошибка возникает при операции умножения, и увидеть, что было передано в неё в качестве аргументов.

После команды except мы можем указать имя класса ошибки, который он должен обрабатывать, затем написать «as» и указать имя переменной, в которую попадет объект с описанием конкретной ошибки.

Блоков except может быть несколько для обработки ошибок разных типов. Проверки выполняются последовательно, будет выполнен тот блок команд, у которого имя класса совпадает с именем класса ошибки или является его предком в дереве иерархии наследования.

Наследование и полиморфизм

Операции сложения и умножения на вещественное или целое число для комплексных чисел очень похожи на поведение свободных векторов на плоскости. Они соответствуют сложению векторов или умножению вектора на число, где действительная часть комплексного числа является x-координатой вектора, а мнимая – y-координатой.

Кроме того, свободный вектор обладает некоторыми операциями, которые характерны только для него, но не для вещественного числа. Это может быть метод length, вычисляющий длину вектора. Нам не хотелось бы засорять код для описания комплексного числа методами для работы со свободным вектором, но, с другой стороны, не хотелось бы и заново переписывать методы сложения и умножения для свободных векторов.

В такой ситуации разумно создать новый класс для описания свободного вектора (или, то же самое, точки на плоскости), который унаследовал бы все методы комплексных чисел и добавил бы новый метод length.

Мы уже знаем, что для того, чтобы пронаследовать класс от другого, достаточно в описании класса указать в круглых скобках, от кого он наследуется. Таким образом, мы можем записать наше описание класса Point с определенным методом length так, как показано на рис. 134.

Вывод этой программы:

```
5.0
4+6i
```

```
class ComplexError(BaseException):
    def __init__(self, Complex, other):
        self.arg1 = Complex
        self.arg2 = other

class Complex:
    def __init__(self, re=0, im=0):
        self.re = re
        self.im = im
    def __str__(self):
        strRep = str(self.re)
        if self.im >= 0:
            strRep += '+'
        strRep += str(self.im) + 'i'
        return strRep
    def __add__(self, other):
        newRe = self.re + other.re
        newIm = self.im + other.im
        return Complex(newRe, newIm)
    def __mul__(self, other):
        if isinstance(other, Complex):
            newRe = self.re * other.re - self.im * other.im
            newIm = self.re * other.im + self.re * other.im
        elif isinstance(other, int) or isinstance(other, float):
            newRe = self.re * other
            newIm = self.im * other
        else:
            raise ComplexError(self, other)
        return Complex(newRe, newIm)
    __rmul__ = __mul__

class Point(Complex):
    def length(self):
        return (self.re**2 + self.im**2)**(1/2)

a = Point(3, 4)
b = Complex(1, 2)
print(a.length())
c = a + b
print(c)
```

Рис. 134. Пример использования наследования

Здесь мы не только убедились в том, что свежесозданный метод работает, но и сделали довольно странную вещь: сложили точку на плоскости с комплексным числом. Дело в том, что объект a класса Point также одновременно является и объектом типа Complex, так как Point пронаследован от Complex. Point лишь расширяет и дополняет Complex, а значит, Point может смело быть интерпретирован как Complex, но не наоборот.

В этом примере истинны выражения isinstance(a, Point) и isinstance(a, Complex), но ложно выражение isinstance(b, Point).

Переопределение методов

Объект типа Point напечатается как комплексное число. Нам хотелось бы, чтобы точки на плоскости печатались в виде (x, y), а не x+yi.

```
class ComplexError(BaseException):
    def __init__(self, Complex, other):
        self.arg1 = Complex
        self.arg2 = other

class Complex:
    def __init__(self, re=0, im=0):
        self.re = re
        self.im = im
    def __str__(self):
        strRep = str(self.re)
        if self.im >= 0:
            strRep += '+'
        strRep += str(self.im) + 'i'
        return strRep
    def __add__(self, other):
        newRe = self.re + other.re
        newIm = self.im + other.im
        return Complex(newRe, newIm)
    def __mul__(self, other):
        if isinstance(other, Complex):
            newRe = self.re * other.re - self.im * other.im
            newIm = self.re * other.im + self.re * other.im
        elif isinstance(other, int) or isinstance(other, float):
            newRe = self.re * other
            newIm = self.im * other
        else:
            raise ComplexError(self, other)
        return Complex(newRe, newIm)
    __rmul__ = __mul__

class Point(Complex):
    def length(self):
        return (self.re**2 + self.im**2)**(1/2)
    def __str__(self):
        return str((self.re, self.im))

a = Point(3, 4)
print(a)
```

Рис. 135. Пример переопределения методов

В языке Питон любой метод можно переопределить в наследнике, что мы и сделаем для метода __str__ для класса Point (рис. 135).

Наш метод обязан возвращать строку, но мы можем воспользоваться функцией str от кортежа, которая выдаст нам нужный результат.

Проектирование структуры классов

Структура описания классов представляет собой дерево (на самом деле в Питоне – ациклический граф) с корнем в виде базового пустого класса.

С помощью грамотно спроектированной структуры классов можно добиться легкой читаемости и максимального повторного использования кода, обеспечения единого интерфейса и множества других преимуществ.

Однако внесение фичи или изменение на высоком уровне иерархии классов могут привести к необходимости выполнить огромное количество работы по модификации всех потомков этого класса, а также к полной несовместимости с предыдущей версией. Многочисленные изменения такого рода приводят к уродливым конструкциям, которые невозможно понимать и отлаживать.

В то же время закладывание перспективных фичей в структуру классов ведет к переусложнению и сводит на нет все повторное использование кода за счет громоздкости конструкций. Кроме того, перспективные фичи могут быть недостаточно обдуманы и приведут к еще большим неудобствам, когда дело дойдет до их реальной реализации (если дойдет).

Таким образом, грамотное проектирование системы классов требует не только хорошего знания паттернов проектирования, но и большого практического опыта. На начальном этапе стоит обучаться проектированию систем, в которые не планируется внесение изменений.

Контрольные вопросы (тесты)

1. Введите правильное ключевое слово, чтобы переменная x принадлежала глобальной области видимости.

```
def myfunc():
 ___x
 x = "fantastic"
```

2. Введите пропущенную часть кода, в результате которого будет получен следующий результат: {'apple', 'cherry'}

```
mySet1 = {"apple", "mango", "cherry"}
mySet2 = {"kivi", "mango", "lime"}
mySet1._____(mySet2)
```

3. Что будет выведено в результате выполнения кода?

```
x = {"f", "e", "d", "c", "b", "a"}
y = {"a", "b", "c"}
z = x.issuperset(y)
z
```

4. Что будет выведено в результате выполнения кода?

```
def add_mean(x, *data):
return x + sum(data)*int(len(data))
add_mean(10,0,1,2,-1,0,-1,1,2)
```

5. Что будет выведено в результате выполнения кода?

```
try:
k = 1 / 0
except ArithmeticError:
k = 0
print(k)
```

6. Что будет выведено в результате выполнения кода?

```
x = lambda a, b : a * b
print(x(10, 2))
```

3. БИБЛИОТЕКИ ДЛЯ УПРАВЛЕНИЯ ДАННЫМИ

3.1. Python pandas

Pandas – библиотека языка Python, инструменты которой позволяют работать с наборами разных типов данных, хранить, обрабатывать и анализировать данные в одномерных и двумерных таблицах.

При помощи pandas можно создавать объекты типа Series (одномерная структура данных) и DataFrame (двумерная структура данных), считывать и записывать файлы ряда форматов, таких как .csv, .xlsx, .html и др. (рис. 136).

Series([данные, индекс, тип, имя, копия, ...]) – одномерный массив (ndarray) с метками оси (включая временные ряды).

```
In [3]: import pandas as pd
        s = pd.Series([1, 3, 5, 'пропуск'])
        s

Out[3]: 0         1
        1         3
        2         5
        3    пропуск
        dtype: object
```

Рис. 136. Создание объекта Series pandas

DataFrame([данные]) – двумерные, изменяемые по размеру, потенциально неоднородные табличные данные (рис. 137).

```
In [2]: df = pd.DataFrame({'A':['A','B', 'C', 'D', 'E'],
                           'B':['1', '2', '3', '4', '5']})
        df
```

Out[2]:

	A	B
0	A	1
1	B	2
2	C	3
3	D	4
4	E	5

Рис. 137. Создание объекта DataFrame pandas

В библиотеке pandas существует поддержка специального временного типа данных – дат (рис. 138).

```
In [3]: import pandas as pd

        dates = pd.date_range('20200101', periods=8)
        dates

Out[3]: DatetimeIndex(['2020-01-01', '2020-01-02', '2020-01-03', '2020-01-04',
                       '2020-01-05', '2020-01-06', '2020-01-07', '2020-01-08'],
                      dtype='datetime64[ns]', freq='D')
```

Рис. 138. Работа с типом данных дата в pandas

Официальное руководство (https://pandas.pydata.org) рекомендует начать знакомство с библиотекой pandas с раздела «10 минут до pandas», предназначенного для начинающих пользователей.

При обращении к библиотеке pandas может быть использовано сокращение pd. В строке кода – import pandas as pd – библиотека pandas импортируется в переменную pd.

Чтение файлов в форматах .csv, .xlsx

Инструменты ввода/вывода pandas – это набор функций верхнего уровня – readers и writers. Для чтения файлов разных форматов в библиотеке pandas предусмотрены разные функции (табл. 7).

Таблица 7

Функции для чтения файлов в библиотеке pandas

Тип формата	Описание данных	Функция readers
Текст	CSV	read_csv
Текст	JSON	read_json
Текст	HTML	read_html
Текст	Локальный буфер обмена / Local clipboard	read_clipboard
	MS Excel	read_excel
Двоичный	OpenDocument	read_excel
Двоичный	HDF5 Format	read_hdf
Двоичный	Stata	read_stata
SQL	SQL	read_sql
SQL	Google BigQuery	read_gbq

Функция pandas.read_csv считывает файл значений, разделенных запятыми (csv), в DataFrame. Также поддерживает опциональную итерацию или разбиение файла на куски.

В результате обработки csv-файла с помощью pandas получается объект под названием DataFrame, который состоит из строк и столбцов.

Необходимо не только указать название файла, но также установить некоторые аргументы (такие как разделитель sep=','), указать индекс строки (по умолчанию header=None), которая будет заголовком выведенной на экран таблицы. Слева добавляется колонка индекса (рис. 139).

```
In [2]: file = pd.read_csv('Моря Тихого океана.csv', sep='\t', header=0)
        file.head()
```

Out[2]:

	Название	Площадь в тыс. кв. км	Наибольшая глубина в метрах
0	Бали	40	1589
1	Банда	714	7440
2	Берингово	2315	5500
3	Восточно-Китайское	836	2719
4	Жёлтое	416	106

Рис. 139. Пример 1 кода – чтение файла и вывод первых строк файла в формате .csv

Параметр index_col указывает название столбца, имена строк в котором будут использованы в качестве индекса. Запись index_col=False используется, чтобы не использовать первый столбец в качестве индекса (рис. 140).

```
In [3]: file = pd.read_csv('Моря Тихого океана.csv', sep='\t', header=0, index_col=['Название'])
        file.tail()
```

Out[3]:

Название	Площадь в тыс. кв. км	Наибольшая глубина в метрах
Тасманово	3336	6015
Фиджи	3177	7633
Филиппинское	5726	10265
Южно-Китайское	3537	5560
Японское	1062	3720

Рис. 140. Пример 2 кода – чтение файла и вывод последних строк файла в формате .csv

Основные атрибуты метода:

– filepath_or_buffer – указывает путь к файлу, URL (включая расположения http, ftp и S3) или любой объект с методом read();

– sep – str, разделитель, по умолчанию «,» для read_csv(), \t для read_table();

– delimiter – альтернатива аргументу «sep»;

– delim_whitespace – boolean, определяет, будет ли пробел использоваться в качестве разделителя.

Атрибуты для расположения и названия столбцов и указателей:

– header – int или список;

– names – массивоподобный список имен столбцов для использования. Если файл не содержит строку заголовка, то должно быть передано header=None, дубликаты в этом списке не допускаются.

Функция pandas.read_excel считывает файл Excel в pandas. DataFrame. Поддерживает расширения файлов xls, xlsx, xlsm, xlsb и odf, считываемые из локальной файловой системы или URL-адреса. Поддерживает возможность чтения одного листа или списка листов.

При считывании файлов Excel требуется парсинг листов документа. При этом данные по умолчанию выводятся в таком виде, в каком представлены в таблице. Не требуется дополнительно указывать, на какой строке будет находиться шапка заголовков таблицы, как это происходит при чтении файлов в формате .csv (рис. 141).

```
In [4]: import pandas as pd
        from pandas import read_excel

In [5]: file1 = pd.ExcelFile('Фрукты.xlsx')
        file_1 = file1.parse('Лист1')
        file_1.head()
```

Out[5]:

	Название	Цвет	Вкус	Тип
0	Яблоко	Зеленый, красный	Кислый, сладкий	Фрукт
1	Груша	Зеленый, желтый	Сладкий	Фрукт
2	Абрикос	Абрикосовый	Сладкий	Фрукт
3	Персик	Персиковый	Сладкий	Фрукт
4	Слива	Сливовый, фиолетовый	Кислый, сладкий	Фрукт

Рис. 141. Пример кода – чтение файла и вывод первых строк файла в формате .xlsx

Некоторые параметры метода:

– io – str, bytes, ExcelFile, xlrd.Book, путь к объекту или файловый объект – путь к объекту, может быть URL-адресом;

– sheet_name – str, int, список, или None, по умолчанию – 0 – строки используются для имен листов. Целые числа используются в позициях листа с нулевой индексацией. Списки строк / целых чисел используются для запроса нескольких листов;

– header – int или список, по умолчанию – 0 – строка (с 0 индексом), используемая для меток столбцов анализируемого фрейма данных;

– index_col – int или список, по умолчанию – None – столбец (с 0 индексом) для использования в качестве меток строк в DataFrame.

Чтобы вывести на экран только часть DataFrame, используются методы:

– pandas.DataFrame.head() – по умолчанию возвращает первые n (по умолчанию 5) строк файла, можно самостоятельно задать количество строк в выборке. При отрицательных значениях n эта функция возвращает все строки, кроме последних n строк, что эквивалентно df[:-n];

– pandas.DataFrame.tail() – по умолчанию возвращает последние n (по умолчанию 5) строк файла, также можно указать несколько строк на выбор. Полезно для быстрой проверки данных, например, после сортировки или добавления строк. При отрицательных значениях n эта функция возвращает все строки, кроме первых n строк, что эквивалентно df[n:];

– pandas.DataFrame.sample(n) – возвращает случайные n строк файла.

Работа с данными в pandas

После чтения файла и создания объекта pandas.DataFrame можно приступать к работе с данными. Для начала можно воспользоваться методом pandas.DataFrame.dtypes, чтобы посмотреть, какие данные хранятся в столбцах таблицы (рис. 142).

По полученным результатам можно сделать вывод, что в таблице содержится столбец с категориальным признаком (тип object), и два столбца содержат количественные данные (тип int64). Эти

сведения важны для проведения в дальнейшем анализа имеющихся в DataFrame данных, так как при использовании некоторых алгоритмов (классификации или кластеризации) требуется проводить дополнительные преобразования над категориальными данными.

```
In [3]: file.dtypes

Out[3]: Название                        object
        Площадь в тыс. кв. км            int64
        Наибольшая глубина в метрах      int64
        dtype: object
```

Рис. 142. Типы данных, хранящиеся в столбцах таблицы

Чтобы получить более подробные сведения о наборе данных, можно воспользоваться методами, возвращающими размер DataFrame, статистические сведения о его содержании и др.

Использование метода pandas.DataFrame.info показано на рис. 143.

```
In [7]: file.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 17 entries, 0 to 16
Data columns (total 3 columns):
 #   Column                       Non-Null Count  Dtype
---  ------                       --------------  -----
 0   Название                     17 non-null     object
 1   Площадь в тыс. кв. км        17 non-null     int64
 2   Наибольшая глубина в метрах  17 non-null     int64
dtypes: int64(2), object(1)
memory usage: 404.0+ bytes
```

Рис. 143. Информация по содержимому таблицы

```
In [8]: file.info(memory_usage='deep')

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 17 entries, 0 to 16
Data columns (total 3 columns):
 #   Column                       Non-Null Count  Dtype
---  ------                       --------------  -----
 0   Название                     17 non-null     object
 1   Площадь в тыс. кв. км        17 non-null     int64
 2   Наибольшая глубина в метрах  17 non-null     int64
dtypes: int64(2), object(1)
memory usage: 1.3 KB
```

Рис. 144. Информация по содержимому таблицы с параметром memory_usage

Добавление параметра memory_usage=«deep» позволяет посмотреть, сколько на самом деле памяти занимает набор данных (рис. 144).

Значение параметра verbose=False выводит сводку количества столбцов и их тип данных, но не информацию о столбце (рис. 145).

```
In [9]: file.info(verbose=False)
        <class 'pandas.core.frame.DataFrame'>
        RangeIndex: 17 entries, 0 to 16
        Columns: 3 entries, Название to Наибольшая глубина в метрах
        dtypes: int64(2), object(1)
        memory usage: 404.0+ bytes
```

Рис. 145. Информация по содержимому таблицы с параметром verbose

Основные параметры метода pandas.DataFrame.info:

– verbose – bool, необязательный параметр, по умолчанию используется настройка в pandas.options.display.max_info_columns;

– buf – буферизация, по умолчанию sys.stdout; указывает, куда отправить вывод;

– max_cols – int, необязательный параметр; по умолчанию используется настройка в pandas.options.display.max_info_columns;

– memory_usage – bool, str, необязательный параметр; указывает, следует ли отображать общее использование памяти элементами DataFrame (включая индекс); по умолчанию соответствует pandas.options.display.memory_usage; True – всегда показывает использование памяти, False – никогда не показывает использование памяти; значение «deep» эквивалентно «True с глубоким самоанализом»;

– null_counts – bool, необязательный параметр; указывает, показывать ли ненулевые значения; значение True всегда показывает количество, а False никогда не показывает количество.

Методы pandas.DataFrame.shape и pandas.DataFrame.size позволяют увидеть размерность и количество записей в таблице (рис. 146).

Содержимое столбца можно получить, указав его наименование (рис. 147).

```
In [8]: file.shape
Out[8]: (17, 3)

In [9]: file.size
Out[9]: 51
```

Рис. 146. Размерность таблицы и количество записей DataFrame

```
In [3]: file['Название'][:10]
Out[3]: 0                  Бали
        1                 Банда
        2             Берингово
        3    Восточно-Китайское
        4                Жёлтое
        5            Коралловое
        6              Охотское
        7                  Саву
        8                 Серам
        9            Соломоново
        Name: Название, dtype: object
```

Рис. 147. Вывод указанного количества строк файла из целевого столбца

Pandas позволяет вывести из файла несколько определенных колонок. Это можно сделать несколькими способами. Если необходимо посмотреть на выборку данных, то достаточно указать множество столбцов (рис. 148).

```
In [4]: file[['Название','Наибольшая глубина в метрах']].head()
Out[4]:
```

	Название	Наибольшая глубина в метрах
0	Бали	1589
1	Банда	7440
2	Берингово	5500
3	Восточно-Китайское	2719
4	Жёлтое	106

Рис. 148. Вывод указанных колонок и первых строк файла

В случае когда новая выборка требуется для дальнейшей работы с данными, нужно записать указанные столбцы в другую переменную и, выполняя дальнейшие преобразования, обращаться уже к новому имени объекта (рис. 149).

```
In [4]: file_new = file[['Название','Наибольшая глубина в метрах']]
        file_new.head(10)
```

Out[4]:

	Название	Наибольшая глубина в метрах
0	Бали	1589
1	Банда	7440
2	Берингово	5500
3	Восточно-Китайское	2719
4	Жёлтое	106
5	Коралловое	9174
6	Охотское	3521
7	Саву	3475
8	Серам	5319
9	Соломоново	9103

Рис. 149. Вывод указанных колонок и первых строк файла в новом объекте DataFrame

Получить список заголовков (индексы) колонок можно, воспользовавшись методом pandas.DataFrame.columns. Обращение к колонке идет при помощи среза по индексу, начиная с [0]. Обращение к последней колонке по индексу [–1] (рис. 150, 151).

```
In [10]: file.columns[0:]
Out[10]: Index(['Название', 'Площадь в тыс. кв. км', 'Наибольшая глубина в метрах'], dtype='object')
```

Рис. 150. Вывод списка наименований колонок объекта DataFrame, начиная с первой

```
In [12]: file.columns[-1]
Out[12]: 'Наибольшая глубина в метрах'
```

Рис. 151. Вывод наименования колонки объекта DataFrame по индексу

Иногда данные в таблицах могут быть неполные и содержать пустые ячейки. Для такого случая в арсенале методов pandas есть pandas.DataFrame.dropna, позволяющий удалить столбцы или строки, где встречается значение NaN:

DataFrame.dropna(self, axis=0, how='any', thresh=None, subset=None, inplace=False.

Основные параметры метода:

– axis – {0 или 'index', 1 или 'columns'}, по умолчанию – 0; ось = 0 – удаляет все строки с пустыми значениями, ось = 1 – удаляет колонки с пустыми значениями;

– how – по умолчанию – «any», определяет, удаляется ли строка или столбец из DataFrame, когда у нее есть хотя бы один NaN или все NaN.

```
In [15]: file2 = pd.ExcelFile('Звездное небо России.xlsx')
         file_2 = file2.parse('Лист1')
         file_2.head()
```

Out[15]:

	Название созвездия	Латинское название	Площадь	Время видимости	Прямое восхождение	Склонение	Максимальная зведная величина
0	Жираф	Camelopardalis (Cam)	757 кв. градусов	Круглый год (наилучшее: февраль)	от 03 ч. 06 м. до 14 ч. 30 м.	от + 52°30' до + 86°30'	NaN
1	Кассиопея	Cassiopeia (Cas)	598 кв. градусов	Круглый год (наилучшее: сентябрь-ноябрь)	от 22 ч. 52 м. до 03 ч. 25 м.	от + 46° до + 77°	NaN
2	Малый Конь	Equuleus (Equ)	72 кв. градуса	Лето - очень (наилучшее: август)	от 20 ч. 50 м. до 21 ч. 20 м.	от + 02° до + 12° 30'	NaN
3	Пегас	Pegasus (Peg)	1121 кв. градус	Почти круглый год, кроме весны (наилучшее: сен...	от 21 ч. 03 м. до 00 ч. 08 м.	от + 01° 45' до + 36°	NaN
4	Цефей	Cepheus (Cep)	588 кв. градусов	Круглый год (наилучшее: июль-сентябрь)	от 20 ч.00 м. до 08 ч. 00 м.	от + 52° до + 88°	NaN

Рис. 152. DataFrame с пустой последней колонкой

После считывания файла и передачи его значений в объект DataFrame можно вывести двумерный массив со всем содержимым, при этом пустые ячейки файла заполняет значение NaN, эквивалентное отсутствию данных (рис. 152).

```
In [17]: file_2.dropna(axis=1)
```

Out[17]:

	Название созвездия	Латинское название	Площадь	Время видимости	Прямое восхождение	Склонение
0	Жираф	Camelopardalis (Cam)	757 кв. градусов	Круглый год (наилучшее: февраль)	от 03 ч. 06 м. до 14 ч. 30 м.	от + 52°30' до + 86°30'
1	Кассиопея	Cassiopeia (Cas)	598 кв. градусов	Круглый год (наилучшее: сентябрь-ноябрь)	от 22 ч. 52 м. до 03 ч. 25 м.	от + 46° до + 77°
2	Малый Конь	Equuleus (Equ)	72 кв. градуса	Лето - очень (наилучшее: август)	от 20 ч. 50 м. до 21 ч. 20 м.	от + 02° до + 12° 30'
3	Пегас	Pegasus (Peg)	1121 кв. градус	Почти круглый год, кроме весны (наилучшее: сен...	от 21 ч. 03 м. до 00 ч. 08 м.	от + 01° 45' до + 36°
4	Цефей	Cepheus (Cep)	588 кв. градусов	Круглый год (наилучшее: июль-сентябрь)	от 20 ч.00 м. до 08 ч. 00 м.	от + 52° до + 88°

Рис. 153. Результат метода DataFrame.dropna с параметром axis=1

Применением метода DataFrame.dropna с параметром удаления пустых значений по axis=1 позволяет удалить целиком столбец с отсутствующими значениями. Таким образом, не нужно вручную удалять неполные данные, это легко выполняется библиотекой pandas (рис. 153).

```
In [18]: file_2.dropna(axis=0)
```

Out[18]:

Название созвездия	Латинское название	Площадь	Время видимости	Прямое восхождение	Склонение	Максимальная зведная величина

Рис. 154. Пример 1 вывода метода DataFrame.dropna с параметром axis=0

Использование в данном случае параметра axis=0 приводит к полному удалению данных из таблицы, так как пустое значение есть в каждой строке таблицы и выбор axis=0, соответственно, приводит к потере всех данных (рис. 154).

```
In [19]: file3 = pd.ExcelFile('Звездное небо России - копия.xlsx')
         file_3 = file3.parse('Лист1')
         file_3.head()
```

Out[19]:

	Название созвездия	Латинское название	Площадь	Время видимости	Прямое восхождение	Склонение
0	Жираф	Camelopardalis (Cam)	757 кв. градусов	Круглый год (наилучшее: февраль)	от 03 ч. 06 м. до 14 ч. 30 м.	от + 52°30' до + 86°30'
1	Кассиопея	Cassiopeia (Cas)	NaN	Круглый год (наилучшее: сентябрь-ноябрь)	от 22 ч. 52 м. до 03 ч. 25 м.	от + 46° до + 77°
2	Малый Конь	Equuleus (Equ)	72 кв. градуса	Лето - очень (наилучшее: август)	от 20 ч. 50 м. до 21 ч. 20 м.	от + 02° до + 12° 30'
3	Пегас	Pegasus (Peg)	1121 кв. градус	Почти круглый год, кроме весны (наилучшее: сен...	NaN	от + 01° 45' до + 36°
4	Цефей	Cepheus (Cep)	588 кв. градусов	Круглый год (наилучшее: июль-сентябрь)	от 20 ч.00 м. до 08 ч. 00 м.	от + 52° до + 88°

Рис. 155. DataFrame с несколькими пустыми значениями

В примере – тот же DataFrame, но неполные значения есть в разных колонках, они рассеяны по таблице данных.

```
In [20]: file_3.dropna(axis=0)
```

Out[20]:

	Название созвездия	Латинское название	Площадь	Время видимости	Прямое восхождение	Склонение
0	Жираф	Camelopardalis (Cam)	757 кв. градусов	Круглый год (наилучшее: февраль)	от 03 ч. 06 м. до 14 ч. 30 м.	от + 52°30' до + 86°30'
2	Малый Конь	Equuleus (Equ)	72 кв. градуса	Лето - очень (наилучшее: август)	от 20 ч. 50 м. до 21 ч. 20 м.	от + 02° до + 12° 30'
4	Цефей	Cepheus (Cep)	588 кв. градусов	Круглый год (наилучшее: июль-сентябрь)	от 20 ч.00 м. до 08 ч. 00 м.	от + 52° до + 88°

Рис. 156. Пример 2 вывода метода DataFrame.dropna с параметром axis=0

В этом случае применение параметра axis=0 дает лучший результат, чем axis=1, так как в последнем варианте теряются данные сразу двух колонок, а при использовании axis=0 убираются строки с неполными данными (рис. 156, 157).

```
In [22]: file_3.dropna(axis=1)
```

Out[22]:

	Название созвездия	Латинское название	Время видимости	Склонение
0	Жираф	Camelopardalis (Cam)	Круглый год (наилучшее: февраль)	от + 52°30' до + 86°30'
1	Кассиопея	Cassiopeia (Cas)	Круглый год (наилучшее: сентябрь-ноябрь)	от + 46° до + 77°
2	Малый Конь	Equuleus (Equ)	Лето - очень (наилучшее: август)	от + 02° до + 12° 30'
3	Пегас	Pegasus (Peg)	Почти круглый год, кроме весны (наилучшее: сен...	от + 01° 45' до + 36°
4	Цефей	Cepheus (Cep)	Круглый год (наилучшее: июль-сентябрь)	от + 52° до + 88°

Рис. 157. Результат метода DataFrame.dropna с параметром axis=1

На примере файла можно посмотреть работу еще одного полезного метода сортировки содержимого колонки и сортировки содержимого таблицы – pandas.DataFrame.sort_values (рис. 158).

```
In [16]: file_2['Название созвездия'].sort_values

Out[16]: <bound method Series.sort_values of 0         Жираф
         1     Кассиопея
         2    Малый Конь
         3         Пегас
         4         Цефей
         Name: Название созвездия, dtype: object>
```

Рис. 158. Сортировка значений одной таблицы

Для того чтобы продолжить работу с файлом, из которого были удалены столбцы или строки с неполными данными, необходимо применить метод pandas.DataFrame.copy. Это позволит продолжить дальнейшие преобразования над данными и работу с DataFrame, как с новым объектом (рис. 159).

```
In [24]: newfile = file_3.dropna(axis=0)
         newfile2 = newfile.copy()
         newfile2
```

Out[24]:

	Название созвездия	Латинское название	Площадь	Время видимости	Прямое восхождение	Склонение
0	Жираф	Camelopardalis (Cam)	757 кв. градусов	Круглый год (наилучшее: февраль)	от 03 ч. 06 м. до 14 ч. 30 м.	от + 52°30' до + 86°30'
2	Малый Конь	Equuleus (Equ)	72 кв. градуса	Лето - очень (наилучшее: август)	от 20 ч. 50 м. до 21 ч. 20 м.	от + 02° до + 12° 30'
4	Цефей	Cepheus (Cep)	588 кв. градусов	Круглый год (наилучшее: июль-сентябрь)	от 20 ч.00 м. до 08 ч. 00 м.	от + 52° до + 88°

Рис. 159. Передача копии DataFrame в новый объект

Метод pandas.DataFrame.copy имеет один главный параметр:

«deep» – bool, по умолчанию True; делает глубокую копию, включая копию данных и индексов; при выборе deep=False ни индексы, ни данные не копируются.

После создания копии набора данных может возникнуть необходимость переименовать колонки, сохранив данные неизменными. Для этого в pandas есть метод pandas.DataFrame.rename (рис. 160).

```
In [25]: newfile2 = newfile2.rename(columns={"Название созвездия": "Название", "Площадь": "Площадь созвездия"})
         newfile2
```

Out[25]:

	Название	Латинское название	Площадь созвездия	Время видимости	Прямое восхождение	Склонение
0	Жираф	Camelopardalis (Cam)	757 кв. градусов	Круглый год (наилучшее: февраль)	от 03 ч. 06 м. до 14 ч. 30 м.	от + 52°30' до + 86°30'
2	Малый Конь	Equuleus (Equ)	72 кв. градуса	Лето - очень (наилучшее: август)	от 20 ч. 50 м. до 21 ч. 20 м.	от + 02° до + 12° 30'
4	Цефей	Cepheus (Cep)	588 кв. градусов	Круглый год (наилучшее: июль-сентябрь)	от 20 ч.00 м. до 08 ч. 00 м.	от + 52° до + 88°

Рис. 160. Переименование наименований столбцов

Значения функции/словаря должны быть уникальными (1-к-1). Метки, не содержащиеся в словаре/Series, будут оставлены как есть.

Использован один из основных параметров метода – «columns» – словарь или функция, альтернатива указанию колонки (mapper, axis=1 эквивалентно columns=mapper).

После удаления строк с пустыми значениями нумерация строк стала непоследовательной. Для переименования индексов используется другой параметр – «index» – словарь или функция, альтернатива указанию колонки (mapper, axis=0 эквивалентно index=mapper) (рис. 161).

```
In [26]: newfile2 = newfile2.rename(index={0 : 1, 4 : 3})
         newfile2
```

Out[26]:

	Название	Латинское название	Площадь созвездия	Время видимости	Прямое восхождение	Склонение
1	Жираф	Camelopardalis (Cam)	757 кв. градусов	Круглый год (наилучшее: февраль)	от 03 ч. 06 м. до 14 ч. 30 м.	от + 52°30' до + 86°30'
2	Малый Конь	Equuleus (Equ)	72 кв. градуса	Лето - очень (наилучшее: август)	от 20 ч. 50 м. до 21 ч. 20 м.	от + 02° до + 12° 30'
3	Цефей	Cepheus (Cep)	588 кв. градусов	Круглый год (наилучшее: июль-сентябрь)	от 20 ч.00 м. до 08 ч. 00 м.	от + 52° до + 88°

Рис. 161. Переименование индексов строк

Переименование колонок и строк можно объединить в одну строку кода (рис. 162).

```
In [25]: newfile2 = newfile2.rename(columns={"Название созвездия": "Название", "Площадь": "Площадь созвездия"},
                                    index={0 : 1, 4 : 3})
         newfile2
```

Out[25]:

	Название	Латинское название	Площадь созвездия	Время видимости	Прямое восхождение	Склонение
1	Жираф	Camelopardalis (Cam)	757 кв. градусов	Круглый год (наилучшее: февраль)	от 03 ч. 06 м. до 14 ч. 30 м.	от + 52°30' до + 86°30'
2	Малый Конь	Equuleus (Equ)	72 кв. градуса	Лето - очень (наилучшее: август)	от 20 ч. 50 м. до 21 ч. 20 м.	от + 02° до + 12° 30'
3	Цефей	Cepheus (Cep)	588 кв. градусов	Круглый год (наилучшее: июль-сентябрь)	от 20 ч.00 м. до 08 ч. 00 м.	от + 52° до + 88°

Рис. 162. Переименование индексов столбцов и строк

При помощи метода .rename можно произвести преобразования типа данных в индексах строк и изменить регистр в заголовках колонок (рис. 163).

```
In [34]: newfile2.rename(index=str).index
Out[34]: Index(['1', '2', '3'], dtype='object')

In [35]: newfile2.rename(str.lower, axis='columns')
```

Out[35]:

	название	латинское название	площадь созвездия	время видимости	прямое восхождение	склонение
1	Жираф	Camelopardalis (Cam)	757 кв. градусов	Круглый год (наилучшее: февраль)	от 03 ч. 06 м. до 14 ч. 30 м.	от + 52°30' до + 86°30'
2	Малый Конь	Equuleus (Equ)	72 кв. градуса	Лето - очень (наилучшее: август)	от 20 ч. 50 м. до 21 ч. 20 м.	от + 02° до + 12° 30'
3	Цефей	Cepheus (Cep)	588 кв. градусов	Круглый год (наилучшее: июль-сентябрь)	от 20 ч.00 м. до 08 ч. 00 м.	от + 52° до + 88°

Рис. 163. Изменение регистра и типа данных при помощи метода .rename

Метод pandas.DataFrame.rename_axis устанавливает наименование для индекса или столбцов (рис. 164).

```
In [26]: newfile2.rename_axis("список", axis="columns")
```

Out[26]:

список	Название	Латинское название	Площадь созвездия	Время видимости	Прямое восхождение	Склонение
1	Жираф	Camelopardalis (Cam)	757 кв. градусов	Круглый год (наилучшее: февраль)	от 03 ч. 06 м. до 14 ч. 30 м.	от + 52°30' до + 86°30'
2	Малый Конь	Equuleus (Equ)	72 кв. градуса	Лето - очень (наилучшее: август)	от 20 ч. 50 м. до 21 ч. 20 м.	от + 02° до + 12° 30'
3	Цефей	Cepheus (Cep)	588 кв. градусов	Круглый год (наилучшее: июль-сентябрь)	от 20 ч.00 м. до 08 ч. 00 м.	от + 52° до + 88°

Рис. 164. Добавление названия к колонке нумерации строк

Метод pandas.DataFrame.reset_index сбрасывает индекс DataFrame и использует вместо него индекс по умолчанию (рис. 165).

```
In [40]: newfile2.reset_index()
```

Out[40]:

	index	Название	Латинское название	Площадь созвездия	Время видимости	Прямое восхождение	Склонение
0	1	Жираф	Camelopardalis (Cam)	757 кв. градусов	Круглый год (наилучшее: февраль)	от 03 ч. 06 м. до 14 ч. 30 м.	от + 52°30' до + 86°30'
1	2	Малый Конь	Equuleus (Equ)	72 кв. градуса	Лето - очень (наилучшее: август)	от 20 ч. 50 м. до 21 ч. 20 м.	от + 02° до + 12° 30'
2	3	Цефей	Cepheus (Cep)	588 кв. градусов	Круглый год (наилучшее: июль-сентябрь)	от 20 ч.00 м. до 08 ч. 00 м.	от + 52° до + 88°

Рис. 165. Результат работы метода .reset_index

Замена значений столбцов возможна при помощи метода pandas.DataFrame.replace (рис. 166).

```
In [27]: filenew = newfile2.reset_index()
         filenew
```

Out[27]:

	index	Название	Латинское название	Площадь созвездия	Время видимости	Прямое восхождение	Склонение
0	1	Жираф	Camelopardalis (Cam)	757 кв. градусов	Круглый год (наилучшее: февраль)	от 03 ч. 06 м. до 14 ч. 30 м.	от + 52°30' до + 86°30'
1	2	Малый Конь	Equuleus (Equ)	72 кв. градуса	Лето - очень (наилучшее: август)	от 20 ч. 50 м. до 21 ч. 20 м.	от + 02° до + 12° 30'
2	3	Цефей	Cepheus (Cep)	588 кв. градусов	Круглый год (наилучшее: июль-сентябрь)	от 20 ч.00 м. до 08 ч. 00 м.	от + 52° до + 88°

```
In [28]: filenew.replace([1, 2, 3], 'строка')
```

Out[28]:

	index	Название	Латинское название	Площадь созвездия	Время видимости	Прямое восхождение	Склонение
0	строка	Жираф	Camelopardalis (Cam)	757 кв. градусов	Круглый год (наилучшее: февраль)	от 03 ч. 06 м. до 14 ч. 30 м.	от + 52°30' до + 86°30'
1	строка	Малый Конь	Equuleus (Equ)	72 кв. градуса	Лето - очень (наилучшее: август)	от 20 ч. 50 м. до 21 ч. 20 м.	от + 02° до + 12° 30'
2	строка	Цефей	Cepheus (Cep)	588 кв. градусов	Круглый год (наилучшее: июль-сентябрь)	от 20 ч.00 м. до 08 ч. 00 м.	от + 52° до + 88°

Рис. 166. Результат работы метода .replace

В библиотеке pandas есть собственные методы визуализации хранящихся в таблицах данных. Один из них – pandas.DataFrame.hist.

Для построения диаграмм используются только количественные данные, поэтому попытка передать в метод таблицу, содержащую только категориальные (строковые) значения, вызовет предупреждение о несоответствии полученного типа данных:

«ValueError: hist method requires numerical columns, nothing to plot».

Некоторые параметры:

– data – DataFrame, объект pandas, содержащий данные;
– bins – str или последовательность, количество бинов гистограммы, которые будут использоваться, по умолчанию – 10;
– column – str или последовательность, если используется, то ограничен выбранным подмножеством столбцов;
– grid – bool, по умолчанию True, показывать ли линии сетки оси.

Примеры реализации показаны на рис. 167, 168.

```
In [38]: file.hist()
Out[38]: array([[<matplotlib.axes._subplots.AxesSubplot object at 0x088AA990>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x0497EC90>]],
               dtype=object)
```

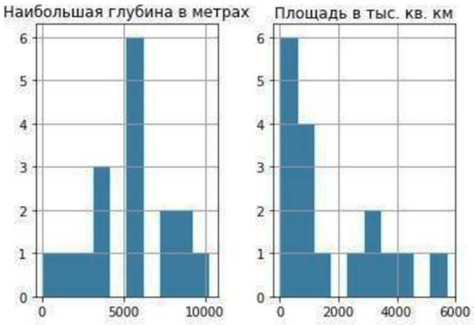


Рис. 167. Результат работы метода .hist
(использованы параметры по умолчанию)

```
In [15]: file.hist(column=['Площадь в тыс. кв. км', 'Наибольшая глубина в метрах'], grid=False, bins=5)
Out[15]: array([[<matplotlib.axes._subplots.AxesSubplot object at 0x08791650>,
                 <matplotlib.axes._subplots.AxesSubplot object at 0x08840ED0>]],
               dtype=object)
```

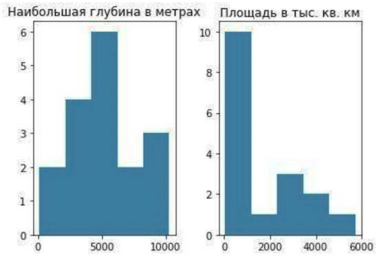


Рис. 168. Результат работы метода .hist (использованы пользовательские параметры columns, grid=False, bins=5)

«hist» является представлением распределения данных. Эта функция вызывает matplotlib.pyplot.hist() для каждого объекта Series в таблице данных, что приводит к одной гистограмме на столбец.

Объединение наборов данных и запись DataFrame в файл

Pandas предоставляет различные средства для простого объединения Series или DataFrame с различными типами данных в индексах, а также функции реляционной алгебры для выполнения операций соединения/слияния. Это функции «pandas.DataFrame.join», «pandas.DataFrame.merge» и «pandas.DataFrame.concat».

Объединение объектов при помощи .concat. Функция выполняет операцию объединения (конкатенации) вдоль оси при выполнении дополнительного набора логики (объединение или пересечение) из индексов (если таковые имеются) на других осях.

В качестве простого примера работы метода можно использовать три набора данных с последовательными индексами (рис. 169, 170).

```
In [1]: import pandas as pd

In [2]: df1 = pd.DataFrame({'A': ['A0', 'A1', 'A2', 'A3'],'B': ['B0', 'B1', 'B2', 'B3'],'C': ['C0', 'C1', 'C2', 'C3'],
                            'D': ['D0', 'D1', 'D2', 'D3']},index=[0, 1, 2, 3])
        df1
```

Out[2]:

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

```
In [3]: df2 = pd.DataFrame({'A': ['A4', 'A5', 'A6', 'A7'],'B': ['B4', 'B5', 'B6', 'B7'],'C': ['C4', 'C5', 'C6', 'C7'],
                            'D': ['D4', 'D5', 'D6', 'D7']},index=[4, 5, 6, 7])
        df2
```

Out[3]:

	A	B	C	D
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7

```
In [4]: df3 = pd.DataFrame({'A': ['A8', 'A9', 'A10', 'A11'],'B': ['B8', 'B9', 'B10', 'B11'],'C': ['C8', 'C9', 'C10', 'C11'],
                            'D': ['D8', 'D9', 'D10', 'D11']},index=[8, 9, 10, 11])
        df3
```

Out[4]:

	A	B	C	D
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

Рис. 169. Создание тестовых наборов данных

```
In [5]: frames = [df1, df2, df3]
        result = pd.concat(frames)
        result
```

Out[5]:

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

Рис. 170. Результат работы метода .concat с использованием параметров по умолчанию

Основные параметры метода:

– objs – последовательность или отображение объектов Series или DataFrame. Если передан словарь, отсортированные ключи будут использоваться в качестве аргумента (keys); любые объекты None будут отброшены без уведомления;

– axis – {0, 1, ...}, по умолчанию 0; ось для конкатенации;

– join – {«inner», «external»}, по умолчанию «external», указывает, как обрабатывать индексы на другой оси; наружный («inner») для объединения и внутренний («external») для пересечения;

– ignore_index – boolean, по умолчанию False. Если True, не используются значения индекса на оси конкатенации. Результирующая ось будет помечена 0, ..., n – 1;

– keys – последовательность, по умолчанию None; строит иерархический индекс, используя переданные ключи в качестве внешнего уровня. Если пройдено несколько уровней, должен содержать кортежи;

– levels – список последовательностей, по умолчанию None; определяет уровни (уникальные значения), чтобы использовать для создания MultiIndex. В противном случае они будут выведены из ключей;

– names – список, по умолчанию None; названия уровней в результирующем иерархическом индексе;

– copy – логическое значение, по умолчанию True; если False, не копирует данные без необходимости (рис. 171, 172, 173).

```
In [1]: import pandas as pd

In [2]: df1 = pd.DataFrame({'A': ['A0', 'A1', 'A2', 'A3'],'B': ['B0', 'B1', 'B2', 'B3'],'C': ['C0', 'C1', 'C2', 'C3'],
                            'D': ['D0', 'D1', 'D2', 'D3']},index=[0, 1, 2, 3])
        df1
```

Out[2]:

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

```
In [5]: df4 = pd.DataFrame({'B': ['B2', 'B3', 'B6', 'B7'],'D': ['D2', 'D3', 'D6', 'D7'],'F': ['F2', 'F3', 'F6', 'F7']},
                           index=[2, 3, 6, 7])
        df4
```

Out[5]:

	B	D	F
2	B2	D2	F2
3	B3	D3	F3
6	B6	D6	F6
7	B7	D7	F7

Рис. 171. Примеры наборов данных с повторяющимся индексом

```
In [6]: result = pd.concat([df1, df4], axis=1, sort=False)
        result
```

Out[6]:

	A	B	C	D	B	D	F
0	A0	B0	C0	D0	NaN	NaN	NaN
1	A1	B1	C1	D1	NaN	NaN	NaN
2	A2	B2	C2	D2	B2	D2	F2
3	A3	B3	C3	D3	B3	D3	F3
6	NaN	NaN	NaN	NaN	B6	D6	F6
7	NaN	NaN	NaN	NaN	B7	D7	F7

Рис. 172. Результат работы метода .concat с пользовательскими параметрами, объединение по axis=1

```
In [7]: result = pd.concat([df1, df4], axis=0, sort=False)
        result
```

Out[7]:

	A	B	C	D	F
0	A0	B0	C0	D0	NaN
1	A1	B1	C1	D1	NaN
2	A2	B2	C2	D2	NaN
3	A3	B3	C3	D3	NaN
2	NaN	B2	NaN	D2	F2
3	NaN	B3	NaN	D3	F3
6	NaN	B6	NaN	D6	F6
7	NaN	B7	NaN	D7	F7

Рис. 173. Результат работы метода .concat с пользовательскими параметрами, объединение по axis=0

При склеивании нескольких DataFrames можно выбрать, как обрабатывать другие оси (кроме сцепляемой). Это можно сделать следующими двумя способами:

1. Опция по умолчанию join=«outer» – объединение всех записей, приводит к нулевой потере информации.

2. Опция join=«inner» – для пересекающихся записей (рис. 174).

```
In [8]: result = pd.concat([df1, df4], axis=1, join='inner')
        result
```

Out[8]:

	A	B	C	D	B	D	F
2	A2	B2	C2	D2	B2	D2	F2
3	A3	B3	C3	D3	B3	D3	F3

```
In [9]: result = pd.concat([df1, df4], axis=0, join='inner')
        result
```

Out[9]:

	B	D
0	B0	D0
1	B1	D1
2	B2	D2
3	B3	D3
2	B2	D2
3	B3	D3
6	B6	D6
7	B7	D7

Рис. 174. Результат объединения по axis=0 и axis= 1 пересекающихся значений

Объединение таблиц может быть сделано при помощи .append, в результате создается копия одного DataFrame, к которому добавляется один или несколько других наборов данных (рис. 175).

```
In [10]: result = df1.append(df4, sort=False)
         result
```

Out[10]:

	A	B	C	D	F
0	A0	B0	C0	D0	NaN
1	A1	B1	C1	D1	NaN
2	A2	B2	C2	D2	NaN
3	A3	B3	C3	D3	NaN
2	NaN	B2	NaN	D2	F2
3	NaN	B3	NaN	D3	F3
6	NaN	B6	NaN	D6	F6
7	NaN	B7	NaN	D7	F7

Рис. 175. Результат работы метода .append

Метод pandas.DataFrame.merge позволяет произвести объединение по одному или нескольким целевым столбцам с указанием индекса (рис. 176, 177).

```
In [11]: first = pd.DataFrame({'key': ['K0', 'K1', 'K2', 'K3'],'A': ['A0', 'A1', 'A2', 'A3'],'B': ['B0', 'B1', 'B2', 'B3']})
         first
```

Out[11]:

	key	A	B
0	K0	A0	B0
1	K1	A1	B1
2	K2	A2	B2
3	K3	A3	B3

```
In [12]: second = pd.DataFrame({'key': ['K0', 'K1', 'K2', 'K3'], 'C': ['C0', 'C1', 'C2', 'C3'], 'D': ['D0', 'D1', 'D2', 'D3']})
         second
```

Out[12]:

	key	C	D
0	K0	C0	D0
1	K1	C1	D1
2	K2	C2	D2
3	K3	C3	D3

```
In [13]: result = pd.merge(first, second, on='key')
         result
```

Out[13]:

	key	A	B	C	D
0	K0	A0	B0	C0	D0
1	K1	A1	B1	C1	D1
2	K2	A2	B2	C2	D2
3	K3	A3	B3	C3	D3

Рис. 176. Результат работы метода .merge, объединение по целевому столбцу «key»

```
In [14]: first = pd.DataFrame({'key1': ['K0', 'K0', 'K1', 'K2'], 'key2': ['K0', 'K1', 'K0', 'K1'],
                               'A': ['A0', 'A1', 'A2', 'A3'],'B': ['B0', 'B1', 'B2', 'B3']})
         first
```

Out[14]:

	key1	key2	A	B
0	K0	K0	A0	B0
1	K0	K1	A1	B1
2	K1	K0	A2	B2
3	K2	K1	A3	B3

```
In [15]: second = pd.DataFrame({'key1': ['K0', 'K1', 'K1', 'K2'],'key2': ['K0', 'K0', 'K0', 'K0'],
                                'C': ['C0', 'C1', 'C2', 'C3'],'D': ['D0', 'D1', 'D2', 'D3']})
         second
```

Out[15]:

	key1	key2	C	D
0	K0	K0	C0	D0
1	K1	K0	C1	D1
2	K1	K0	C2	D2
3	K2	K0	C3	D3

```
In [16]: result = pd.merge(first, second, how='outer', on=['key1', 'key2'])
         result
```

Out[16]:

	key1	key2	A	B	C	D
0	K0	K0	A0	B0	C0	D0
1	K0	K1	A1	B1	NaN	NaN
2	K1	K0	A2	B2	C1	D1
3	K1	K0	A2	B2	C2	D2
4	K2	K1	A3	B3	NaN	NaN
5	K2	K0	NaN	NaN	C3	D3

Рис. 177. Результат работы метода .merge, с двумя целевыми столбцами, параметр «how» – внешнее объединение

Метод pandas.DataFrame.join подходит для объединения DataFrame с разными индексами столбцов (рис. 178).

Методы объединения одинаково работают как с категориальными, так и с количественными типами данных.

После объединения больших DataFrame некоторые значения могут дублироваться, и метод pandas.DataFrame.drop_duplicates позволяет автоматически избавляться от строк с идентичным содержанием (рис. 179, 180).

```
In [22]: AB = pd.DataFrame({'A': ['A0', 'A1', 'A2'],'B': ['B0', 'B1', 'B2']},
                           index=['K0', 'K1', 'K2'])
         AB
```

Out[22]:

	A	B
K0	A0	B0
K1	A1	B1
K2	A2	B2

```
In [24]: CD = pd.DataFrame({'C': ['C0', 'C2', 'C3'],'D': ['D0', 'D2', 'D3']},
                           index=['K0', 'K2', 'K3'])
         CD
```

Out[24]:

	C	D
K0	C0	D0
K2	C2	D2
K3	C3	D3

```
In [25]: result = AB.join(CD)
         result
```

Out[25]:

	A	B	C	D
K0	A0	B0	C0	D0
K1	A1	B1	NaN	NaN
K2	A2	B2	C2	D2

Рис. 178. Результат работы метода .join

```
In [2]: file1
```

Out[2]:

	Название	Площадь в тыс. кв. км	Наибольшая глубина в метрах
0	Бали	40	1589
1	Банда	714	7440
2	Беринговo	2315	5500

```
In [3]: file2
```

Out[3]:

	Название	Площадь в тыс. кв. км	Наибольшая глубина в метрах
0	Бали	40	1589
1	Банда	714	7440
2	Восточно-Китайское	836	2719
3	Жёлтое	416	106
4	Коралловое	4068	9174
5	Охотское	1603	3521

Рис. 179. Пример двух DataFrame с одинаковой строкой (индекс 1)

```
In [4]: result = pd.concat([file1, file2]).drop_duplicates()
        result
```

Out[4]:

	Название	Площадь в тыс. кв. км	Наибольшая глубина в метрах
0	Бали	40	1589
1	Банда	714	7440
2	Берингово	2315	5500
2	Восточно-Китайское	836	2719
3	Жёлтое	416	106
4	Коралловое	4068	9174
5	Охотское	1603	3521

Рис. 180. Объединение файлов с удалением дубликатов

Полученный в результате различных преобразований DataFrame может быть записан во внешний файл.

Некоторые функции pandas для записи разных типов файлов представлены в табл. 8.

Таблица 8

Функции pandas для разных типов файлов

Тип формата	Описание данных	Функция writers
Текст	CSV	to_csv
Текст	JSON	to_json
Текст	HTML	to_html
Текст	Локальный буфер обмена / Local clipboard	to_clipboard
	MS Excel	to_excel
Двоичный	OpenDocument	to_excel
Двоичный	HDF5 Format	to_hdf
Двоичный	Stata	to_stata
SQL	SQL	to_sql
SQL	Google BigQuery	to_gbq

При помощи методов pandas.DataFrame.to_csv и pandas.DataFrame.to_excel можно записать полученный после объединения таблиц результат в файл с расширением .csv и .xlsx. При этом,

если первоначальные значения были получены из файла .xlsx, при помощи pandas они могут быть записаны в файл с другим расширением, например, .csv (рис. 181).

```
In [4]: result.to_csv('result.csv',index=False, encoding='utf-8')

In [5]: result.to_excel('result2.xlsx')
```

Рис. 181. Запись во внешний файл

Данные экспортируются в файл указанного формата. Методы содержат параметры для указания пути к файлу, наличия разделителя данных, перечня записываемых колонок и др.

Для pandas.DataFrame.to_csv путь к файлу – это аргумент «path_or_buf», для «pandas.DataFrame.to_excel» – «excel_writer». Идентичные параметры: «columns», «header», «index», «index_label».

Мы рассмотрели основные, но далеко не все методы библиотеки pandas. Более подробно с ними можно ознакомиться в официальной документации: https://pandas.pydata.org/pandas-docs/stable/index.html.

3.2. Python numpy

Numpy – библиотека Python для хранения и обработки данных, анализа данных, машинного обучения и научных вычислений – облегчает обработку векторов и матриц.

Основной объект numpy – однородный многомерный массив (numpy.ndarray). Это многомерный массив элементов (обычно чисел) одного типа.

Главными атрибутами объектов ndarray являются:

– ndarray.ndim – число измерений (чаще их называют «оси») массива;

– ndarray.shape – размеры массива, его форма. Это кортеж натуральных чисел, показывающий длину массива по каждой оси. Для матрицы из n строк и m столбов, shape будет (n,m). Число элементов кортежа shape равно ndim;

– ndarray.size – количество элементов массива. Очевидно, равно произведению всех элементов атрибута shape;

– ndarray.dtype – объект, описывающий тип элементов массива. Можно определить dtype, используя стандартные типы данных Python. Numpy здесь предоставляет целый ряд типов, как встроенных, например bool_, character, int8, int16, int32, int64, float8, float16, float32, float64, complex64, object_, так и собственных типов данных, в том числе и составных;

– ndarray.itemsize – размер каждого элемента массива в байтах;

– ndarray.data – буфер, содержащий фактические элементы массива. Обычно не нужно использовать этот атрибут, так как обращаться к элементам массива проще всего с помощью индексов.

Numpy используется многими пакетами Python как основной элемент инфраструктуры, среди них Scikit-Learn, SciPy, Pandas и Tenorflow.

При обращении к библиотеке pandas может быть использовано сокращение np: import numpy as np.

Возможности библиотеки numpy могут быть использованы при создании объектов pandas, таких как Series и DataFrame. Посмотрим простой пример, когда при помощи numpy создается массив данных ndarray и записывается в качестве значений индексов в таблице набора данных pandas (рис. 182).

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

        a = np.arange(0,10,2)
        b = np.arange(1,10,2)
        df = pd.DataFrame({'A' : a, 'B' : b})
        df
```

Out[1]:

	A	B
0	0	1
1	2	3
2	4	5
3	6	7
4	8	9

Рис. 182. Создание объекта DataFrame с использованием numpy

Метод np.arange возвращает равномерно распределенные значения в заданном интервале. Значения генерируются в пределах

полуоткрытого интервала. В данном случае стартовое значение 0 или 1, окончание – значение 10, с шагом 2.

Объекты Series и DataFrame могут быть преобразованы в массив numpy. Для этого существует два основных метода библиотеки pandas: Series.to_numpy() и DataFrame.to_numpy() (рис. 183).

```
In [1]: import pandas as pd
        import numpy as np
        series = pd.Series(pd.Categorical(['a', 'b', 'a']))
        series.to_numpy()

Out[1]: array(['a', 'b', 'a'], dtype=object)

In [2]: df = pd.DataFrame({"A": [1, 2], "B": [3, 4]}).to_numpy()
        df

Out[2]: array([[1, 3],
               [2, 4]], dtype=int64)
```

Рис. 183. Преобразование объектов pandas в массив numpy

В результате выполнения кода мы получаем массив и информацию о типе содержащихся в нем данных. Столбцы объекта Data Frame становятся, соответственно, столбцами двумерного массива.

По умолчанию Python имеет следующие типы данных:

– strings – используется для представления текстовых данных, текст указывается в кавычках, например, «ABCD» или ‘абвгд’;

– integer – используется для представления целых чисел, например: –1, 0, 2;

– float – используется для представления реальных чисел, например: 1.5, 10.90;

– boolean – используется для представления True или False;

– complex – используется для представления комплексных чисел, например: 1.0 + 2.0j, 1.5 + 2.5j.

Numpy имеет несколько дополнительных типов данных и ссылается на типы данных одним символом, например: i – для целых чисел, u – для целых чисел без знака и т. д. Некоторые типы данных numpy:

– i – целое число;

– b – логическое значение;

– u – целое число без знака;

– f – число с плавающей точкой;
– c – комплексное число;
– M – дата и время;
– O – объект;
– S – строка;
– U – строка в кодировке Юникод.

Массивы, матрицы и арифметические операции numpy

В numpy существует много способов создать массив. Один из наиболее простых – создать массив из обычных списков или кортежей Python, используя функцию numpy.array() (рис. 184).

```
In [1]: import numpy as np
        np.array([1,2,3])
Out[1]: array([1, 2, 3])

In [2]: np.array([1,2.5,3])
Out[2]: array([1. , 2.5, 3. ])

In [3]: np.array([1,'test',3])
Out[3]: array(['1', 'test', '3'], dtype='<U11')

In [4]: np.array([[1,2,3],['первый','второй','третий']])
Out[4]: array([['1', '2', '3'],
               ['первый', 'второй', 'третий']], dtype='<U11')
```

Рис. 184. Создание простых массивов

Мы видим несколько простых массивов, созданных при помощи библиотеки numpy. В качестве элементов списка могут быть переданы данные любого типа.

Функция array() трансформирует вложенные последовательности в многомерные массивы. Тип элементов массива зависит от типа элементов исходной последовательности, но можно и переопределить его в момент создания (рис. 185).

```
In [1]: import numpy as np
        np.array([[1.5, 2, 3], [4, 5, 6]], dtype=np.complex)
Out[1]: array([[1.5+0.j, 2. +0.j, 3. +0.j],
               [4. +0.j, 5. +0.j, 6. +0.j]])
```

Рис. 185. Переопределение типа данных при создании массива

Кроме функции array(), в numpy существуют другие способы создавать массивы. Если элементы массива вначале неизвестны, а массив, в котором они будут храниться, уже нужен, можно воспользоваться несколькими функциями для того, чтобы создавать массивы с каким-то исходным содержимым (по умолчанию тип создаваемого массива – float64).

Функция zeros() создает массив из нулей, а функция ones() – массив из единиц. Обе функции принимают кортеж с размерами массива и аргумент dtype. По умолчанию тип данных созданного массива – float (рис. 186, 187).

```
In [1]: import numpy as np
        np.zeros((3, 5), dtype=int)

Out[1]: array([[0, 0, 0, 0, 0],
               [0, 0, 0, 0, 0],
               [0, 0, 0, 0, 0]])

In [2]: import numpy as np
        np.zeros((3, 5), dtype=float)

Out[2]: array([[0., 0., 0., 0., 0.],
               [0., 0., 0., 0., 0.],
               [0., 0., 0., 0., 0.]])
```

Рис. 186. Создание массива функцией zeros()

```
In [1]: import numpy as np
        np.ones((3, 5), dtype=int)

Out[1]: array([[1, 1, 1, 1, 1],
               [1, 1, 1, 1, 1],
               [1, 1, 1, 1, 1]])

In [2]: import numpy as np
        np.ones((3, 5))

Out[2]: array([[1., 1., 1., 1., 1.],
               [1., 1., 1., 1., 1.],
               [1., 1., 1., 1., 1.]])

In [3]: import numpy as np
        np.ones((3, 5), dtype=complex)

Out[3]: array([[1.+0.j, 1.+0.j, 1.+0.j, 1.+0.j, 1.+0.j],
               [1.+0.j, 1.+0.j, 1.+0.j, 1.+0.j, 1.+0.j],
               [1.+0.j, 1.+0.j, 1.+0.j, 1.+0.j, 1.+0.j]])
```

Рис. 187. Создание массива функцией ones()

Получить тип данных массива, содержащего строки, можно при помощи атрибута .dtype (рис. 188).

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

        a = np.array([1, 2, 3])

        a.dtype

Out[1]: dtype('int32')

In [2]: import numpy as np

        a = np.array(['apple', 'mango', 'lemon'])

        a.dtype

Out[2]: dtype('<U5')
```

Рис. 188. Получение типа данных массива

Если указан тип, в котором элементы не могут быть преобразованы, то numpy вызовет ValueError. В Python ValueError вызывается, когда тип передаваемого аргумента функции является неожиданным или неправильным.

Например, нецелочисленная строка, такая как «a», не может быть преобразована в целое число (вызовет ошибку) (рис. 189).

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

        a = np.array(['a', '2', '3'], dtype='i')

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-1-3d1c17d7e080> in <module>
      1 import numpy as np
      2
----> 3 a = np.array(['a', '2', '3'], dtype='i')

ValueError: invalid literal for int() with base 10: 'a'
```

Рис. 189. Ошибки при создании массивов

Лучший способ изменить тип данных существующего массива – это сделать копию массива с помощью метода .astype(). Функция создает копию массива и позволяет указать тип данных в качестве параметра (рис. 190).

Тип данных может быть указан с использованием строки («f'» для «float», «i» для целого числа и т. д.), или можно использовать тип данных напрямую («float» для «float» и «int» для целого числа).

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

        a = np.array([1.5, 2.5, 3.5])

        a_new = a.astype('i')

        print(a_new)
        print(a_new.dtype)
```

```
[1 2 3]
int32
```

```
In [2]: import numpy as np

        a = np.array([1.5, 2.5, 3.5])

        a_new = a.astype(int)

        print(a_new)
        print(a_new.dtype)
```

```
[1 2 3]
int32
```

Рис. 190. Изменение типа данных массива

В рассмотренном случае массив был скопирован при помощи метода astype(). В библиотеке numpy есть и другие методы для создания копии данных, содержащихся в массиве – это copy() и view().

Между copy() и view() есть разница: copy() – это новый массив, а view() – это просто представление исходного массива.

Копия (copy()) хранит данные, и любые изменения, внесенные в копию, не влияют на исходный массив, а любые изменения, внесенные в исходный массив, не влияют на копию (рис. 191).

Представление (view()) не хранит данные, и любые изменения, внесенные в представление, влияют на исходный массив, а внесенные в исходный массив, влияют на представление.

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

        a = np.array([1, 2, 3, 4, 5])
        x = a.copy()
        a[0] = 100

        print(a)
        print(x)
```

```
[100   2   3   4   5]
[1 2 3 4 5]
```

Рис. 191. Метод copy()

Рассмотрим пример подробно. При помощи метода array() создается целочисленный одномерный массив a (вектор). Затем хранящиеся в нем данные копируются в переменную x. В следующей строке кода первый элемент массива заменяется новым значением.

Выводим оба массива и смотрим на результат. Первый массив, в котором был заменен элемент с индексом [0], выводится с новыми значениями. Массив x, в который мы передали копию первоначальных данных, содержит неизмененное значение первого элемента.

Посмотрим работу метода view() (рис. 192). Мы передали наш одномерный массив в новую переменную и сделали замену первого элемента. Запускаем код и в результате получаем два одинаковых массива. То же самое происходит, когда заменяется элемент второго массива.

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

        a = np.array([1, 2, 3, 4, 5])
        x = a.view()
        a[0] = 100

        print(a)
        print(x)

        [100   2   3   4   5]
        [100   2   3   4   5]

In [2]: import numpy as np

        a = np.array([1, 2, 3, 4, 5])
        x = a.view()
        x[0] = 100

        print(a)
        print(x)

        [100   2   3   4   5]
        [100   2   3   4   5]
```

Рис. 192. Метод view()

Атрибут .base позволяет посмотреть, хранятся ли в новых массивах данные (рис. 193).

Как видно из результатов выполнения программы, возвращено значение None и исходный массив.

Метод .eye возвращает многомерный массив (матрицу) с единицами по диагонали и нулями в остальных местах (рис. 194).

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

        arr = np.array([1, 2, 3, 4, 5])

        x = arr.copy()
        y = arr.view()

        print(x.base)
        print(y.base)

        None
        [1 2 3 4 5]
```

Рис. 193. Атрибут .base

```
In [1]: import numpy as np
        x = np.eye(5, dtype=int)
        x

Out[1]: array([[1, 0, 0, 0, 0],
               [0, 1, 0, 0, 0],
               [0, 0, 1, 0, 0],
               [0, 0, 0, 1, 0],
               [0, 0, 0, 0, 1]])
```

Рис. 194. Создание массива методом .eye

Функция arange() в numpy предназначена для создания последовательностей чисел. Функция принимает три аргумента – начало интервала, конец и интервал между последовательностями. При использовании arange() с аргументами типа float сложно быть уверенным в том, сколько элементов будет получено (из-за ограничения точности чисел с плавающей запятой). Поэтому в таких случаях обычно лучше использовать функцию linspace(), которая вместо шага в качестве одного из аргументов принимает число, равное количеству нужных элементов (рис. 195).

```
In [1]: import numpy as np
        x = np.arange(1,10,2)
        x

Out[1]: array([1, 3, 5, 7, 9])

In [2]: x = np.linspace(1,20,5)
        x

Out[2]: array([ 1.  ,  5.75, 10.5 , 15.25, 20.  ])
```

Рис. 195. Методы .arange() и .linspace()

После создания над массивами можно производить арифметические операции. Например, операцию сложения одномерных и многомерных массивов (рис. 196).

```
In [1]: import numpy as np
        a1 = np.array([1,2])
        a2 = np.array([2,5])
        a1,a2

Out[1]: (array([1, 2]), array([2, 5]))

In [2]: a1 + a2

Out[2]: array([3, 7])

In [3]: a3 = np.array([[1,2],[1,3]])
        a4 = np.array([[2,5],[3,5]])
        a3, a4

Out[3]: (array([[1, 2],
                [1, 3]]),
         array([[2, 5],
                [3, 5]]))

In [4]: a3 + a4

Out[4]: array([[3, 7],
               [4, 8]])
```

Рис. 196. Сложение массивов

При сложении массивов складываются значения каждого ряда, в матрицах складываются значения столбцов. Матрицы должны быть одного и того же размера. Помимо сложения, можно выполнять и другие простые операции: вычитание, умножение, деление (рис. 197, 198, 199).

```
In [1]: import numpy as np
        a1 = np.array([2,5])
        a2 = np.array([1,2])
        a1,a2

Out[1]: (array([2, 5]), array([1, 2]))

In [2]: a1 - a2

Out[2]: array([1, 3])

In [3]: a3 = np.array([[2,5],[3,5]])
        a4 = np.array([[1,2],[1,3]])
        a3, a4

Out[3]: (array([[2, 5],
                [3, 5]]),
         array([[1, 2],
                [1, 3]]))

In [4]: a3 - a4

Out[4]: array([[1, 3],
               [2, 2]])
```

Рис. 197. Вычитание массивов

```
In [1]: import numpy as np
        a1 = np.array([1,2])
        a2 = np.array([2,5])
        a1,a2
Out[1]: (array([1, 2]), array([2, 5]))

In [2]: a1 * a2
Out[2]: array([ 2, 10])

In [3]: a3 = np.array([[1,2],[1,3]])
        a4 = np.array([[2,5],[3,5]])
        a3, a4
Out[3]: (array([[1, 2],
                [1, 3]]),
         array([[2, 5],
                [3, 5]]))

In [4]: a3 * a4
Out[4]: array([[ 2, 10],
               [ 3, 15]])
```

Рис. 198. Умножение массивов

```
In [1]: import numpy as np
        a1 = np.array([2,5])
        a2 = np.array([1,2])
        a1,a2
Out[1]: (array([2, 5]), array([1, 2]))

In [2]: a1 / a2
Out[2]: array([2. , 2.5])

In [3]: a3 = np.array([[2,5],[3,5]])
        a4 = np.array([[1,2],[1,2]])
        a3, a4
Out[3]: (array([[2, 5],
                [3, 5]]),
         array([[1, 2],
                [1, 2]]))

In [4]: a3 / a4
Out[4]: array([[2. , 2.5],
               [3. , 2.5]])
```

Рис. 199. Деление массивов

При вычитании массивов также вычитаются значения каждого ряда. В матрицах вычитаются значения столбцов соответственно.

Так же, как при операции сложения, здесь матрицы должны быть одного и того же размера.

Массив можно нарезать и проиндексировать. Принцип работы похож на то, как это происходит со списками Python (рис. 200).

```
In [1]: import numpy as np
        a = np.array([1,2,3,4,5])
        a
Out[1]: array([1, 2, 3, 4, 5])

In [2]: a[0]
Out[2]: 1

In [3]: a[1:4]
Out[3]: array([2, 3, 4])

In [4]: a[-1]
Out[4]: 5
```

Рис. 200. Нарезка и индексация массива

Преимуществом numpy является наличие в нем функций агрегирования: min(), max(), sum(), mean(), prod(), std() и др.

```
In [1]: import numpy as np
        a = np.array([1,2,3])
        a
Out[1]: array([1, 2, 3])
```

Рис. 201. Создание одномерного массива (вектора), содержащего последовательность из трех элементов

```
In [2]: a.max()
Out[2]: 3
```

Рис. 202. Максимальное значение массива

```
In [3]: a.min()
Out[3]: 1
```

Рис. 203. Минимальное значение массива

```
In [4]: a.sum()
Out[4]: 6
```

Рис. 204. Сумма значений функции

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

Рис. 205. Среднее арифметическое массива

```
In [6]: a.std()
Out[6]: 0.816496580927726
```

Рис. 206. Среднеквадратическое отклонение массива

```
In [7]: a.prod()
Out[7]: 6
```

Рис. 207. Результат умножения всех элементов массива

В представленных примерах (рис. 201–207) использован одномерный массив (вектор). При работе с многомерным массивом (матрицей) можно указать, по какой оси нужно вычислить функцию агрегирования. Например, найти максимальное значение в каждом столбце, указав ось axis = 0 или axis = 1 (рис. 208).

```
In [1]: import numpy as np
        a = np.array([[1,2,3],[3,4,5],[10,20,30]])
        a
Out[1]: array([[ 1,  2,  3],
               [ 3,  4,  5],
               [10, 20, 30]])

In [2]: a.max()
Out[2]: 30

In [3]: a.max(axis=0)
Out[3]: array([10, 20, 30])

In [4]: a.max(axis=1)
Out[4]: array([ 3,  5, 30])
```

Рис. 208. Максимальное значение массива в каждом столбце

Арифметические операции над матрицами разных размеров возможны в том случае, если размерность одной из матриц равна одному. Это значит, что в матрице только один столбец или один ряд. В таком случае для выполнения операции numpy используют правила трансляции (рис. 209).

```
In [1]: import numpy as np
        a1 = np.array([[1,2,3],[7,8,9]])
        a1

Out[1]: array([[1, 2, 3],
               [7, 8, 9]])

In [2]: a2 = np.array([1,2,5])
        a2

Out[2]: array([1, 2, 5])

In [3]: a1 + a2

Out[3]: array([[ 2,  4,  8],
               [ 8, 10, 14]])
```

Рис. 209. Сложение матриц разных размеров

В некоторых случаях необходимо перевернуть матрицу. Это может потребоваться при вычислении скалярного произведения двух матриц. В numpy каждая матрица может использовать метод dot(), который применяется для проведения скалярных операций (рис. 210).

```
In [1]: import numpy as np
        a = np.array([1,2,3])

In [2]: a.dot(np.array([4,9,7]))

Out[2]: 43
```

Рис. 210. Скалярное произведение матриц

Тогда возникает необходимость наличия совпадающих размерностей. У массивов NumPy есть полезное свойство T – транспонирование матрицы (рис. 211).

```
In [1]: import numpy as np
        a = np.array([[1,2,3],[4,5,6],[7,8,9]])
        a

Out[1]: array([[1, 2, 3],
               [4, 5, 6],
               [7, 8, 9]])

In [2]: a.T

Out[2]: array([[1, 4, 7],
               [2, 5, 8],
               [3, 6, 9]])
```

Рис. 211. Транспонирование матрицы

Некоторые более сложные ситуации требуют возможности переключения между размерностями рассматриваемой матрицы. В этом случае подходит метод reshape(). Здесь требуется только передать новые размерности для матрицы. Для размерности можно передать –1, и numpy выведет ее верное значение, опираясь на данные рассматриваемой матрицы (рис. 212).

```
In [1]: import numpy as np
        a = np.array([0,1,2,3,4,5])
        a

Out[1]: array([0, 1, 2, 3, 4, 5])

In [2]: a.reshape(2,3)

Out[2]: array([[0, 1, 2],
               [3, 4, 5]])

In [3]: a.reshape(-1)

Out[3]: array([0, 1, 2, 3, 4, 5])
```

Рис. 212. Изменение размерности матрицы

Для добавления, удаления и сортировки элементов матрицы в numpy есть методы: .append(), .delete() и .sort() (рис. 213).

Форму и размер массива можно узнать при помощи следующих атрибутов:

– ndarray.ndim() – показывает количество осей или размеров массива (рис. 214, 215);

– ndarray.size() – выводит общее количество элементов массива (рис. 216);

– ndarray.shape() – выводит кортеж целых чисел, которые указывают количество элементов, хранящихся вдоль каждого измерения массива (рис. 217).

```
In [1]: import numpy as np
        a = np.array([3,2,1])
        a

Out[1]: array([3, 2, 1])

In [2]: a = np.append(a,[4,5])
        a

Out[2]: array([3, 2, 1, 4, 5])

In [3]: a = np.delete(a, -1)
        a

Out[3]: array([3, 2, 1, 4])

In [4]: a = np.sort(a)
        a

Out[4]: array([1, 2, 3, 4])
```

Рис. 213. Добавление, удаление и сортировка элементов массива

```
In [1]: import numpy as np
        a1 = np.array([1,2,3])
        a2 = np.array([[1,2,3],[0,3,1],[2,1,0]])
        a1, a2

Out[1]: (array([1, 2, 3]),
         array([[1, 2, 3],
                [0, 3, 1],
                [2, 1, 0]]))

In [2]: np.ndim(a1)

Out[2]: 1

In [3]: np.ndim(a2)

Out[3]: 2
```

Рис. 214. Количество осей и размеров массива

```
In [4]: a3 = np.zeros((2, 3, 4))
        a3

Out[4]: array([[[0., 0., 0., 0.],
                [0., 0., 0., 0.],
                [0., 0., 0., 0.]],

               [[0., 0., 0., 0.],
                [0., 0., 0., 0.],
                [0., 0., 0., 0.]]])

In [5]: a3.ndim

Out[5]: 3
```

Рис. 215. Количество осей и размеров нулевой матрицы

```
In [1]: import numpy as np
        a = np.array([[1,2,3],[0,3,1],[2,1,0]])
        a

Out[1]: array([[1, 2, 3],
               [0, 3, 1],
               [2, 1, 0]])

In [2]: np.size(a)

Out[2]: 9
```

Рис. 216. Общее количество элементов массива

```
In [1]: import numpy as np
        a = np.array([[1,2,3],[0,3,1],[2,1,0]])
        a

Out[1]: array([[1, 2, 3],
               [0, 3, 1],
               [2, 1, 0]])

In [2]: np.shape(a)

Out[2]: (3, 3)
```

Рис. 217. Количество элементов, хранящихся вдоль каждого измерения массива

Метод ndarray.itemsize позволяет узнать размер каждого элемента массива в байтах (рис. 218). Например, для массива из элементов типа float64 значение itemsize равно 8 (=64/8), а для complex32 этот атрибут равен 4 (=32/8).

```
In [1]: import numpy as np
        a1 = np.array([[1,2,3],[0,3,1],[2,1,0]])
        a1

Out[1]: array([[1, 2, 3],
               [0, 3, 1],
               [2, 1, 0]])

In [2]: a1.itemsize

Out[2]: 4

In [3]: a2 = np.array(['adc','dfg','klm'])
        a2

Out[3]: array(['adc', 'dfg', 'klm'], dtype='<U3')

In [4]: a2.itemsize

Out[4]: 12
```

Рис. 218. Размер каждого элемента массива в байтах

При помощи методов numpy .concatenate и .array_split можно объединять и разделять массивы.

Объединение (рис. 219) означает помещение содержимого двух или более массивов в один массив. В SQL объединение таблиц происходит на основе ключа, в numpy массивы объединяются по осям. Мы передаем последовательность массивов, которые хотим присоединить функцией .concatenate(), вместе с осью. Если ось не передана явно, она принимается за 0.

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

        a1 = np.array([1, 2, 3])
        a1

Out[1]: array([1, 2, 3])

In [2]: a2 = np.array([4, 5, 6])
        a2

Out[2]: array([4, 5, 6])

In [3]: a = np.concatenate((a1, a2))
        a

Out[3]: array([1, 2, 3, 4, 5, 6])
```

Рис. 219. Объединение двух массивов

Разделение (рис. 220) является обратной операцией присоединения. Функция .concatenate() объединяет несколько массивов в один, а разделение разбивает один массив на несколько. Используя array_split() для разделения массивов, мы передаем ему массив, который хотим разделить, и количество разделений:

```
In [1]: import numpy as np
        a = np.array([1, 2, 3, 4, 5, 6])
        a

Out[1]: array([1, 2, 3, 4, 5, 6])

In [2]: a_new = np.array_split(a, 3)
        a_new

Out[2]: [array([1, 2]), array([3, 4]), array([5, 6])]
```

Рис. 220. Разделение массивов

В numpy есть возможность извлекать элементы из массива. Извлечение некоторых элементов из существующего массива и создание из них нового массива называется фильтрацией (filtering) (рис. 221).

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

        a = np.array([41, 42, 43, 44])
        a

Out[1]: array([41, 42, 43, 44])

In [2]: x = [True, False, True, False]
        x

Out[2]: [True, False, True, False]

In [3]: a_new = a[x]
        a_new

Out[3]: array([41, 43])
```

Рис. 221. Фильтрация массива

В numpy вы фильтруете массив, используя список логических индексов. Список логических индексов – это список логических индексов, соответствующих индексам в массиве.

Если значением индекса является True, то элемент сохраняется в фильтруемом массиве. Если значением этого индекса является False, этот элемент исключается из фильтруемого массива.

Numpy позволяет искать в массиве определенное значение и возвращать соответствующие индексы. Для поиска в массиве используется метод where() (рис. 222).

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

        a1 = np.array(['apple','mango','lemon','apple'])

        x = np.where(a1 == 'apple')

        x

Out[1]: (array([0, 3], dtype=int32),)
```

Рис. 222. Поиск элементов в массиве

В numpy еще много полезных функций, позволяющих хранить и обрабатывать данные. Полную информацию о них можно найти в официальном руководстве: https://numpy.org/doc/.

3.3. Python matplotlib

Matplotlib – библиотека на языке программирования Python для визуализации данных двумерной и трехмерной графикой. Получаемые изображения могут быть использованы в качестве иллюстраций в публикациях. Далее будут представлены примеры использования различных графиков.

Точечный график

Scatteplot – это классический и фундаментальный вид диаграммы, используемый для изучения взаимосвязи между двумя переменными. Если есть несколько групп в данных, можно визуализировать каждую группу в другом цвете. В matplotlib легко сделать это, используя plt.scatterplot() (рис. 223, 224).

```
# Import dataset
midwest = pd.read_csv("https://raw.githubusercontent.com/selva86/datasets/master/midwest_
filter.csv")

# Prepare Data
# Create as many colors as there are unique midwest['category']
categories = np.unique(midwest['category'])
colors = [plt.cm.tab10(i/float(len(categories)-1)) for i in range(len(categories))]

# Draw Plot for Each Category
plt.figure(figsize=(16, 10), dpi= 80, facecolor='w', edgecolor='k')

for i, category in enumerate(categories):
    plt.scatter('area', 'poptotal',
                data=midwest.loc[midwest.category==category, :],
                s=20, c=colors[i], label=str(category))

# Decorations
plt.gca().set(xlim=(0.0, 0.1), ylim=(0, 90000),
              xlabel='Area', ylabel='Population')

plt.xticks(fontsize=12); plt.yticks(fontsize=12)
plt.title("Scatterplot of Midwest Area vs Population", fontsize=22)
plt.legend(fontsize=12)
plt.show()
```

Рис. 223. Программный код 1 для построения графика

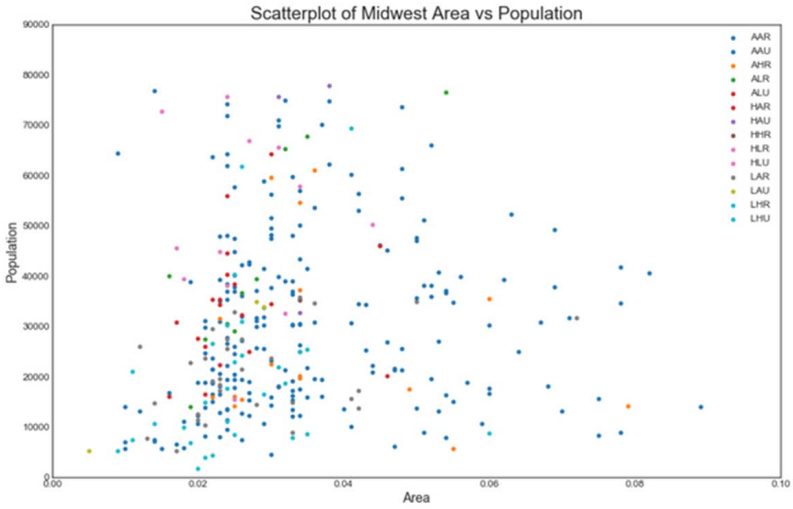


Рис. 224. Результат выполнения программного кода 1

Пузырьковая диаграмма с захватом группы

Иногда хочется показать группу точек внутри границы, чтобы подчеркнуть их важность. В этом примере мы получаем записи из фрейма данных, которые должны быть выделены, и передаем их в encircle(), описанный в приведенном ниже коде (рис. 225, 226).

```
from matplotlib import patches
from scipy.spatial import ConvexHull
import warnings; warnings.simplefilter('ignore')
sns.set_style("white")

# Step 1: Prepare Data
midwest = pd.read_csv("https://raw.githubusercontent.com/selva86/datasets/master/midwest_
filter.csv")

# As many colors as there are unique midwest['category']
categories = np.unique(midwest['category'])
colors = [plt.cm.tab10(i/float(len(categories)-1)) for i in range(len(categories))]

# Step 2: Draw Scatterplot with unique color for each category
fig = plt.figure(figsize=(16, 10), dpi= 80, facecolor='w', edgecolor='k')

for i, category in enumerate(categories):
    plt.scatter('area', 'poptotal', data=midwest.loc[midwest.category==category, :], s='d
ot_size', c=colors[i], label=str(category), edgecolors='black', linewidths=.5)

# Step 3: Encircling
# https://stackoverflow.com/questions/44575681/how-do-i-encircle-different-data-sets-in-s
catter-plot
def encircle(x,y, ax=None, **kw):
    if not ax: ax=plt.gca()
    p = np.c_[x,y]
    hull = ConvexHull(p)
    poly = plt.Polygon(p[hull.vertices,:], **kw)
    ax.add_patch(poly)

# Select data to be encircled
midwest_encircle_data = midwest.loc[midwest.state=='IN', :]

# Draw polygon surrounding vertices
encircle(midwest_encircle_data.area, midwest_encircle_data.poptotal, ec="k", fc="gold", a
lpha=0.1)
encircle(midwest_encircle_data.area, midwest_encircle_data.poptotal, ec="firebrick", fc="
none", linewidth=1.5)

# Step 4: Decorations
plt.gca().set(xlim=(0.0, 0.1), ylim=(0, 90000),
              xlabel='Area', ylabel='Population')

plt.xticks(fontsize=12); plt.yticks(fontsize=12)
plt.title("Bubble Plot with Encircling", fontsize=22)
plt.legend(fontsize=12)
plt.show()
```

Рис. 225. Программный код 2 для построения графика

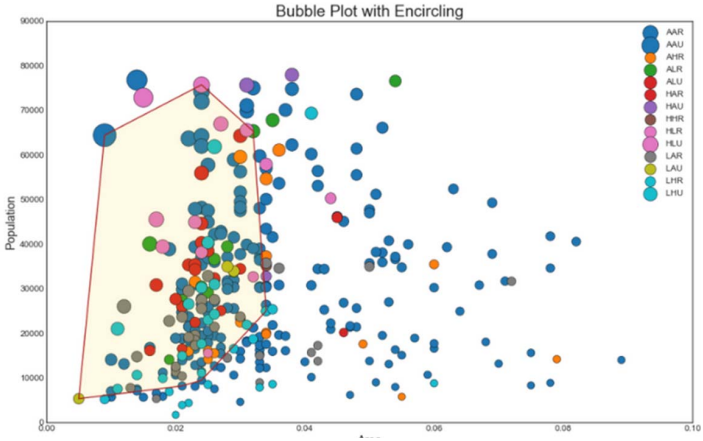


Рис. 226. Результат выполнения программного кода 2

График линейной регрессии best fit

Если вы хотите понять, как две переменные изменяются по отношению друг к другу, лучше всего подойдет линия best fit. На графике ниже показано, как best fit выделяется среди различных групп данных. Чтобы отключить группировки и просто нарисовать одну линию best fit для всего набора данных, нужно удалить параметр hue='cyl' из sns.lmplot() ниже (рис. 227, 228).

Кроме того, можно показать линию best fit для каждой группы в отдельном столбце, установив параметр col=groupingcolumn внутри sns.lmplot() (рис. 229, 230).

```
# Import Data
df = pd.read_csv("https://raw.githubusercontent.com/selva86/datasets/master/mpg_ggplot2.c
sv")
df_select = df.loc[df.cyl.isin([4,8]), :]

# Plot
sns.set_style("white")
gridobj = sns.lmplot(x="displ", y="hwy", hue="cyl", data=df_select,
                     height=7, aspect=1.6, robust=True, palette='tab10',
                     scatter_kws=dict(s=60, linewidths=.7, edgecolors='black'))

# Decorations
gridobj.set(xlim=(0.5, 7.5), ylim=(0, 50))
plt.title("Scatterplot with line of best fit grouped by number of cylinders", fontsize=2
0)
plt.show()
```

Рис. 227. Программный код 3 для построения графика

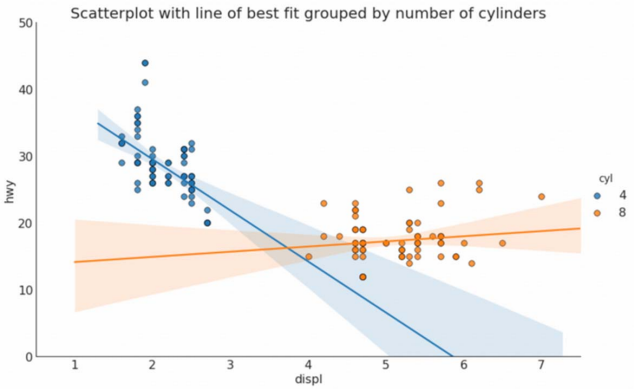


Рис. 228. Результат выполнения программного кода 3

```
# Import Data
df = pd.read_csv("https://raw.githubusercontent.com/selva86/datasets/master/mpg_ggplot2.c
sv")
df_select = df.loc[df.cyl.isin([4,8]), :]

# Each line in its own column
sns.set_style("white")
gridobj = sns.lmplot(x="displ", y="hwy",
                     data=df_select,
                     height=7,
                     robust=True,
                     palette='Set1',
                     col="cyl",
                     scatter_kws=dict(s=60, linewidths=.7, edgecolors='black'))

# Decorations
gridobj.set(xlim=(0.5, 7.5), ylim=(0, 50))
plt.show()
```

Рис. 229. Программный код 4 для построения графика

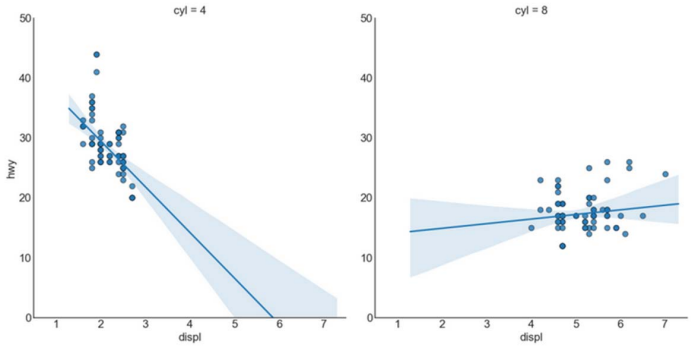


Рис. 230. Результат выполнения программного кода 4

Stripplot

Часто несколько точек данных имеют одинаковые значения X и Y. В результате несколько точек наносятся друг на друга и скрываются. Чтобы избежать этого, нужно слегка раздвинуть точки, чтобы можно было видеть их по отдельности. Это удобно делать с помощью стрипплота (stripplot()) (рис. 231, 232).

```
# Import Data
df = pd.read_csv("https://raw.githubusercontent.com/selva86/datasets/master/mpg_ggplot2.c
sv")

# Draw Stripplot
fig, ax = plt.subplots(figsize=(16,10), dpi= 80)
sns.stripplot(df.cty, df.hwy, jitter=0.25, size=8, ax=ax, linewidth=.5)

# Decorations
plt.title('Use jittered plots to avoid overlapping of points', fontsize=22)
plt.show()
```

Рис. 231. Программный код 5 для построения графика

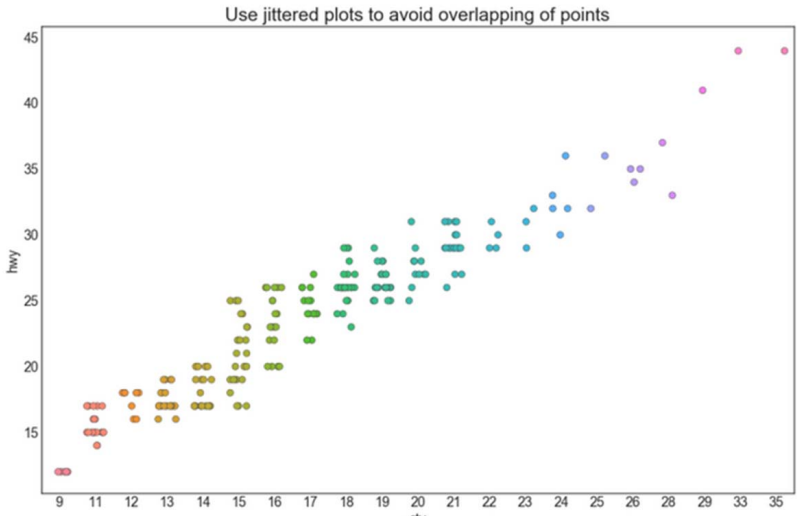


Рис. 232. Результат выполнения программного кода 5

График подсчета (Counts Plot)

Другим вариантом, позволяющим избежать проблемы наложения точек, является увеличение размера точки в зависимости от того, сколько точек лежит в этом месте. Таким образом, чем больше размер точки, тем больше концентрация точек вокруг нее (рис. 233, 234).

```
# Import Data
df = pd.read_csv("https://raw.githubusercontent.com/selva86/datasets/master/mpg_ggplot2.c
sv")
df_counts = df.groupby(['hwy', 'cty']).size().reset_index(name='counts')

# Draw Stripplot
fig, ax = plt.subplots(figsize=(16,10), dpi= 80)
sns.scatterplot(df_counts.cty, df_counts.hwy, size=df_counts.counts*2, ax=ax)

# Decorations
plt.title('Counts Plot - Size of circle is bigger as more points overlap', fontsize=22)
plt.show()
```

Рис. 233. Программный код 6 для построения графика

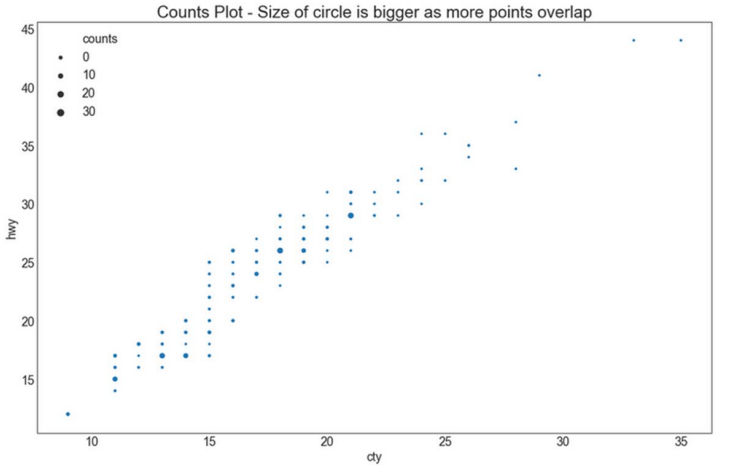


Рис. 234. Результат выполнения программного кода 6

Построчная гистограмма

Построчные гистограммы имеют гистограмму вдоль переменных осей X и Y. Это используется для визуализации отношений между X и Y вместе с одномерным распределением X и Y по отдельности. Этот график часто используется в анализе данных (EDA) (рис. 235, 236).

```
# Import Data
df = pd.read_csv("https://raw.githubusercontent.com/selva86/datasets/master/mpg_ggplot2.csv")

# Create Fig and gridspec
fig = plt.figure(figsize=(16, 10), dpi= 80)
grid = plt.GridSpec(4, 4, hspace=0.5, wspace=0.2)

# Define the axes
ax_main = fig.add_subplot(grid[:-1, :-1])
ax_right = fig.add_subplot(grid[:-1, -1], xticklabels=[], yticklabels=[])
ax_bottom = fig.add_subplot(grid[-1, 0:-1], xticklabels=[], yticklabels=[])

# Scatterplot on main ax
ax_main.scatter('displ', 'hwy', s=df.cty*4, c=df.manufacturer.astype('category').cat.codes, alpha=.9, data=df, cmap="tab10", edgecolors='gray', linewidths=.5)

# histogram on the right
ax_bottom.hist(df.displ, 40, histtype='stepfilled', orientation='vertical', color='deeppink')
ax_bottom.invert_yaxis()

# histogram in the bottom
ax_right.hist(df.hwy, 40, histtype='stepfilled', orientation='horizontal', color='deeppink')

# Decorations
ax_main.set(title='Scatterplot with Histograms \n displ vs hwy', xlabel='displ', ylabel='hwy')
ax_main.title.set_fontsize(20)
for item in ([ax_main.xaxis.label, ax_main.yaxis.label] + ax_main.get_xticklabels() + ax_main.get_yticklabels()):
    item.set_fontsize(14)

xlabels = ax_main.get_xticks().tolist()
ax_main.set_xticklabels(xlabels)
plt.show()
```

Рис. 235. Программный код 7 для построения графика

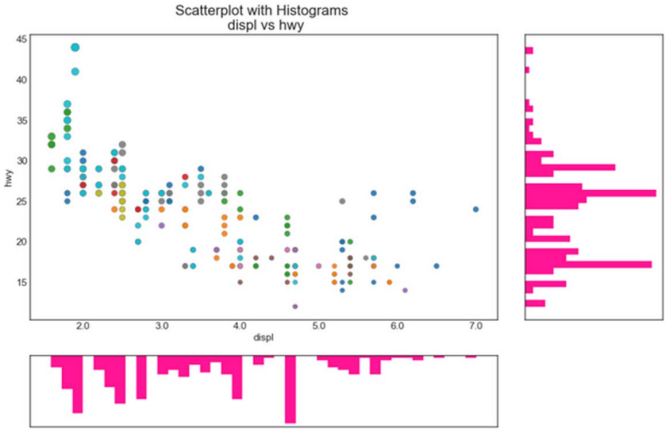


Рис. 236. Результат выполнения программного кода 7

Boxplot

Boxplot служит той же цели, что и построчная гистограмма. Тем не менее этот график помогает точно определить медиану, 25-й и 75-й персентили X и Y (рис. 237, 238).

```
# Import Data
df = pd.read_csv("https://raw.githubusercontent.com/selva86/datasets/master/mpg_ggplot2.c
sv")

# Create Fig and gridspec
fig = plt.figure(figsize=(16, 10), dpi= 80)
grid = plt.GridSpec(4, 4, hspace=0.5, wspace=0.2)

# Define the axes
ax_main = fig.add_subplot(grid[:-1, :-1])
ax_right = fig.add_subplot(grid[:-1, -1], xticklabels=[], yticklabels=[])
ax_bottom = fig.add_subplot(grid[-1, 0:-1], xticklabels=[], yticklabels=[])

# Scatterplot on main ax
ax_main.scatter('displ', 'hwy', s=df.cty*5, c=df.manufacturer.astype('category').cat.code
s, alpha=.9, data=df, cmap="Set1", edgecolors='black', linewidths=.5)

# Add a graph in each part
sns.boxplot(df.hwy, ax=ax_right, orient="v")
sns.boxplot(df.displ, ax=ax_bottom, orient="h")

# Decorations ------------------
# Remove x axis name for the boxplot
ax_bottom.set(xlabel='')
ax_right.set(ylabel='')

# Main Title, Xlabel and YLabel
ax_main.set(title='Scatterplot with Histograms \n displ vs hwy', xlabel='displ', ylabel='
hwy')

# Set font size of different components
ax_main.title.set_fontsize(20)
for item in ([ax_main.xaxis.label, ax_main.yaxis.label] + ax_main.get_xticklabels() + ax_
main.get_yticklabels()):
    item.set_fontsize(14)

plt.show()
```

Рис. 237. Программный код 8 для построения графика

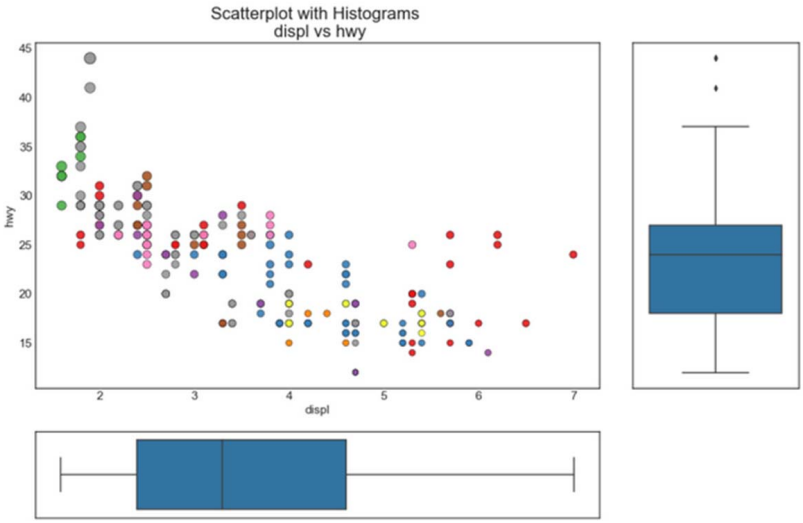


Рис. 238. Результат выполнения программного кода 8

Диаграмма корреляции

Диаграмма корреляции используется для визуального просмотра метрики корреляции между всеми возможными парами числовых переменных в наборе данных (или двумерном массиве) (рис. 239, 240).

```
# Import Dataset
df = pd.read_csv("https://github.com/selva86/datasets/raw/master/mtcars.csv")

# Plot
plt.figure(figsize=(12,10), dpi= 80)
sns.heatmap(df.corr(), xticklabels=df.corr().columns, yticklabels=df.corr().columns, cmap
='RdYlGn', center=0, annot=True)

# Decorations
plt.title('Correlogram of mtcars', fontsize=22)
plt.xticks(fontsize=12)
plt.yticks(fontsize=12)
plt.show()
```

Рис. 239. Программный код 9 для построения графика

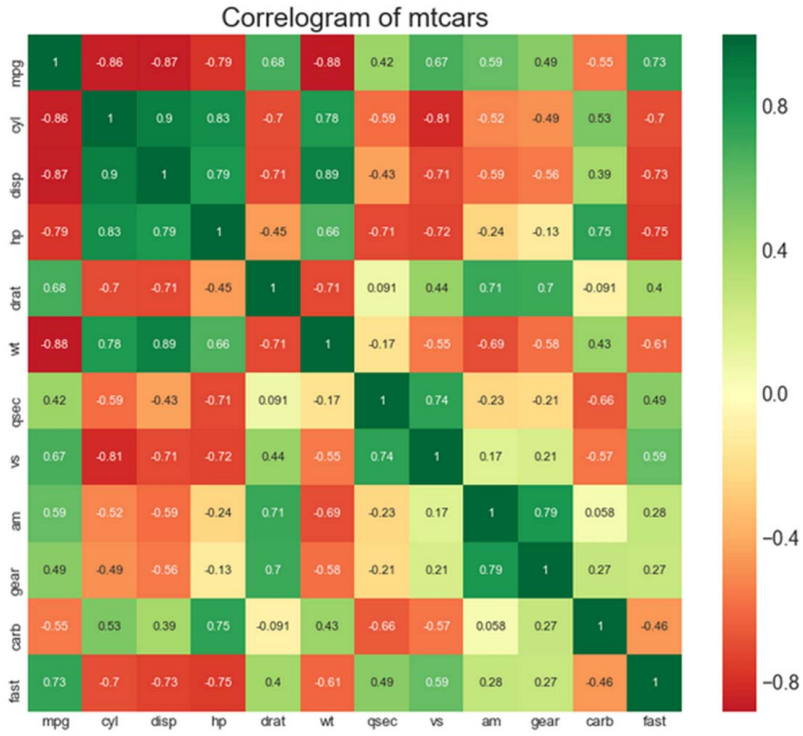


Рис. 240. Результат выполнения программного кода 9

Парный график

Часто используется в исследовательском анализе, чтобы понять взаимосвязь между всеми возможными парами числовых переменных. Это обязательный инструмент для двумерного анализа (рис. 241–244).

```
# Load Dataset
df = sns.load_dataset('iris')

# Plot
plt.figure(figsize=(10,8), dpi= 80)
sns.pairplot(df, kind="scatter", hue="species", plot_kws=dict(s=80, edgecolor="white", li
newidth=2.5))
plt.show()
```

Рис. 241. Программный код 10 для построения графика

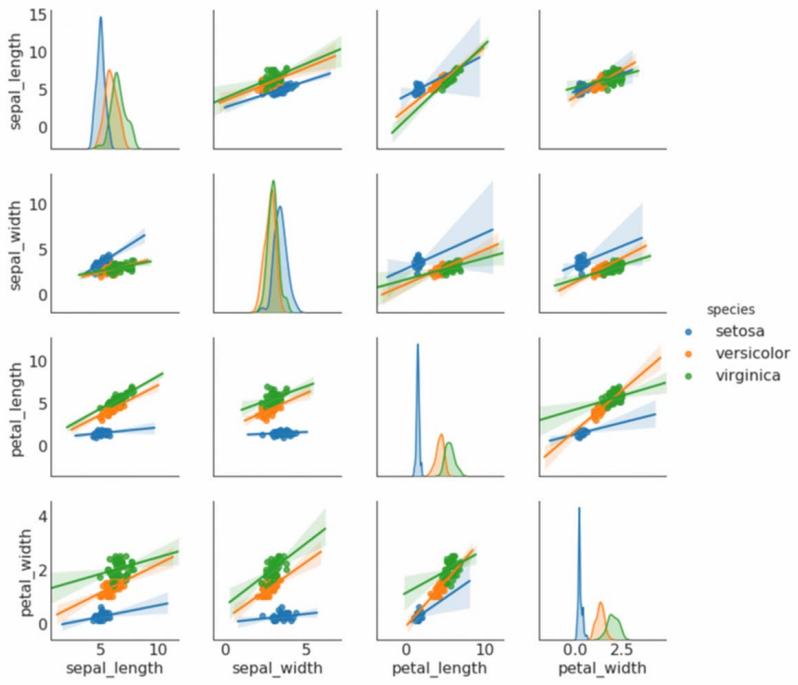


Рис. 242. Результат выполнения программного кода 10

```
# Load Dataset
df = sns.load_dataset('iris')

# Plot
plt.figure(figsize=(10,8), dpi= 80)
sns.pairplot(df, kind="reg", hue="species")
plt.show()
```

Рис. 243. Программный код 11 для построения графика

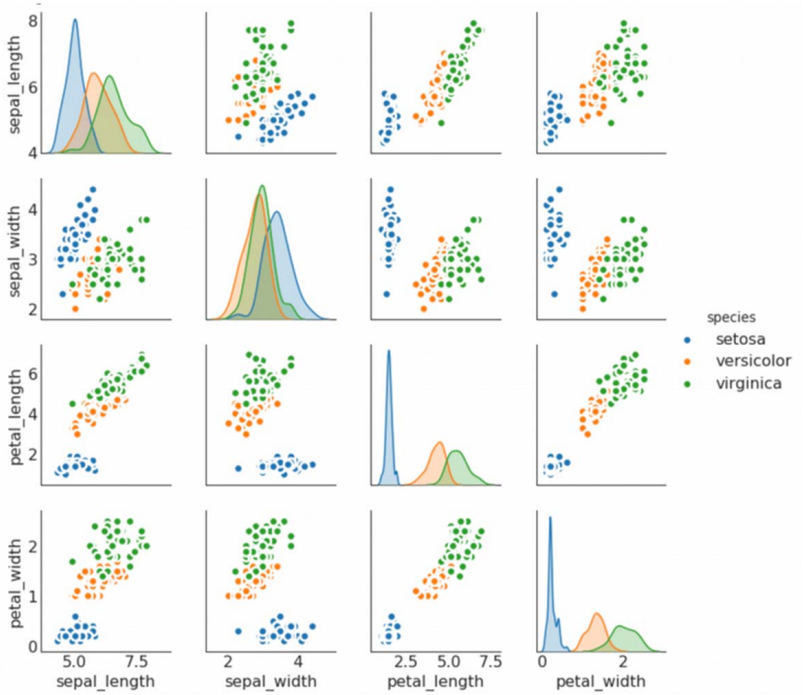


Рис. 244. Результат выполнения программного кода 11

Расходящиеся столбцы

Чтобы увидеть, как элементы меняются в зависимости от одной метрики, и визуализировать порядок и величину этой дисперсии, расходящиеся столбцы – отличный инструмент. Он помогает быстро дифференцировать производительность групп в ваших данных, является достаточно интуитивным и мгновенно передает смысл (рис. 245, 246).

```
# Prepare Data
df = pd.read_csv("https://github.com/selva86/datasets/raw/master/mtcars.csv")
x = df.loc[:, ['mpg']]
df['mpg_z'] = (x - x.mean())/x.std()
df['colors'] = ['red' if x < 0 else 'green' for x in df['mpg_z']]
df.sort_values('mpg_z', inplace=True)
df.reset_index(inplace=True)

# Draw plot
plt.figure(figsize=(14,10), dpi= 80)
plt.hlines(y=df.index, xmin=0, xmax=df.mpg_z, color=df.colors, alpha=0.4, linewidth=5)

# Decorations
plt.gca().set(ylabel='$Model$', xlabel='$Mileage$')
plt.yticks(df.index, df.cars, fontsize=12)
plt.title('Diverging Bars of Car Mileage', fontdict={'size':20})
plt.grid(linestyle='--', alpha=0.5)
plt.show()
```

Рис. 245. Программный код 12 для построения графика

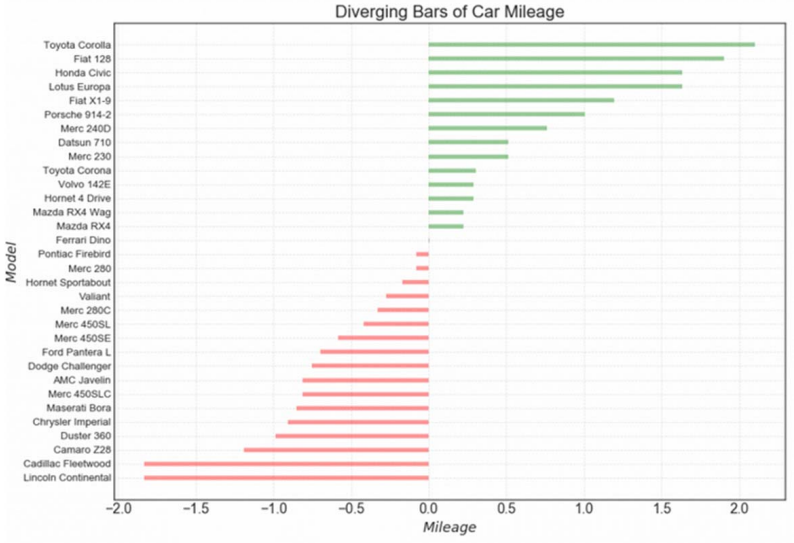


Рис. 246. Результат выполнения программного кода 12

Расходящиеся точки

График расходящихся точек похож на расходящиеся столбцы. Однако отсутствие столбцов уменьшает степень контрастности и несоответствия между группами (рис. 247, 248).

```
# Prepare Data
df = pd.read_csv("https://github.com/selva86/datasets/raw/master/mtcars.csv")
x = df.loc[:, ['mpg']]
df['mpg_z'] = (x - x.mean())/x.std()
df['colors'] = ['red' if x < 0 else 'darkgreen' for x in df['mpg_z']]
df.sort_values('mpg_z', inplace=True)
df.reset_index(inplace=True)

# Draw plot
plt.figure(figsize=(14,16), dpi= 80)
plt.scatter(df.mpg_z, df.index, s=450, alpha=.6, color=df.colors)
for x, y, tex in zip(df.mpg_z, df.index, df.mpg_z):
    t = plt.text(x, y, round(tex, 1), horizontalalignment='center',
                 verticalalignment='center', fontdict={'color':'white'})

# Decorations
# Lighten borders
plt.gca().spines["top"].set_alpha(.3)
plt.gca().spines["bottom"].set_alpha(.3)
plt.gca().spines["right"].set_alpha(.3)
plt.gca().spines["left"].set_alpha(.3)

plt.yticks(df.index, df.cars)
plt.title('Diverging Dotplot of Car Mileage', fontdict={'size':20})
plt.xlabel('$Mileage$')
plt.grid(linestyle='--', alpha=0.5)
plt.xlim(-2.5, 2.5)
plt.show()
```

Рис. 247. Программный код 13 для построения графика

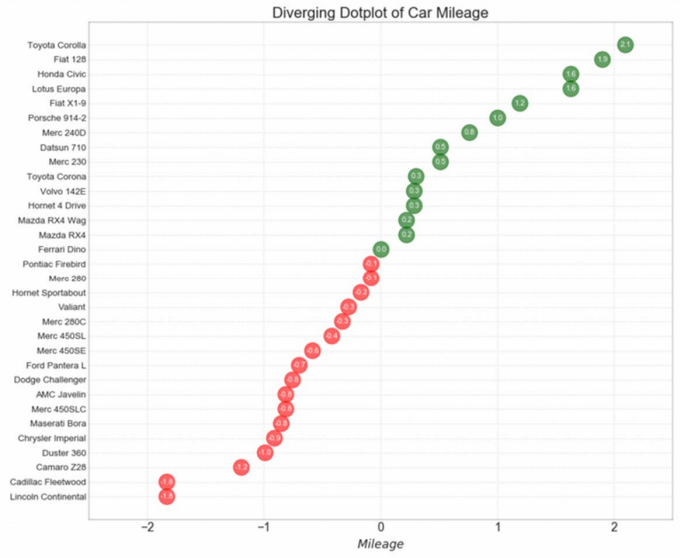


Рис. 248. Результат выполнения программного кода 13

Расходящаяся диаграмма Lollipop с маркерами

Lollipop обеспечивает гибкий способ визуализации расхождения, делая акцент на любых значимых точках данных, на которые вы хотите обратить внимание (рис. 249, 250).

```
# Prepare Data
df = pd.read_csv("https://github.com/selva86/datasets/raw/master/mtcars.csv")
x = df.loc[:, ['mpg']]
df['mpg_z'] = (x - x.mean())/x.std()
df['colors'] = 'black'

# color fiat differently
df.loc[df.cars == 'Fiat X1-9', 'colors'] = 'darkorange'
df.sort_values('mpg_z', inplace=True)
df.reset_index(inplace=True)


# Draw plot
import matplotlib.patches as patches

plt.figure(figsize=(14,16), dpi= 80)
plt.hlines(y=df.index, xmin=0, xmax=df.mpg_z, color=df.colors, alpha=0.4, linewidth=1)
plt.scatter(df.mpg_z, df.index, color=df.colors, s=[600 if x == 'Fiat X1-9' else 300 for
x in df.cars], alpha=0.6)
plt.yticks(df.index, df.cars)
plt.xticks(fontsize=12)

# Annotate
plt.annotate('Mercedes Models', xy=(0.0, 11.0), xytext=(1.0, 11), xycoords='data',
            fontsize=15, ha='center', va='center',
            bbox=dict(boxstyle='square', fc='firebrick'),
            arrowprops=dict(arrowstyle='-[, widthB=2.0, lengthB=1.5', lw=2.0, color='stee
lblue'), color='white')

# Add Patches
p1 = patches.Rectangle((-2.0, -1), width=.3, height=3, alpha=.2, facecolor='red')
p2 = patches.Rectangle((1.5, 27), width=.8, height=5, alpha=.2, facecolor='green')
plt.gca().add_patch(p1)
plt.gca().add_patch(p2)

# Decorate
plt.title('Diverging Bars of Car Mileage', fontdict={'size':20})
plt.grid(linestyle='--', alpha=0.5)
plt.show()
```

Рис. 249. Программный код 14 для построения графика

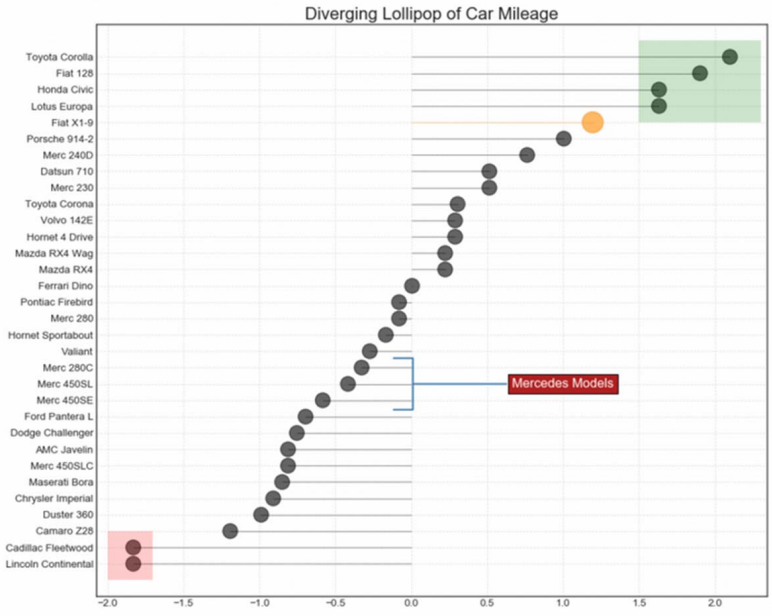


Рис. 250. Результат выполнения программного кода 14

Диаграмма площади

Раскрашивая область между осью и линиями, диаграмма площади подчеркивает пики и впадины. Такая диаграмма позволяет оценить продолжительности максимумов и минимумов. Чем больше продолжительность максимумов, тем больше площадь под линией (рис. 251, 252).

```
import numpy as np
import pandas as pd

# Prepare Data
df = pd.read_csv("https://github.com/selva86/datasets/raw/master/economics.csv", parse_da
tes=['date']).head(100)
x = np.arange(df.shape[0])
y_returns = (df.psavert.diff().fillna(0)/df.psavert.shift(1)).fillna(0) * 100

# Plot
plt.figure(figsize=(16,10), dpi= 80)
plt.fill_between(x[1:], y_returns[1:], 0, where=y_returns[1:] >= 0, facecolor='green', in
terpolate=True, alpha=0.7)
plt.fill_between(x[1:], y_returns[1:], 0, where=y_returns[1:] <= 0, facecolor='red', inte
rpolate=True, alpha=0.7)

# Annotate
plt.annotate('Peak \n1975', xy=(94.0, 21.0), xytext=(88.0, 28),
             bbox=dict(boxstyle='square', fc='firebrick'),
             arrowprops=dict(facecolor='steelblue', shrink=0.05), fontsize=15, color='whi
te')


# Decorations
xtickvals = [str(m)[:3].upper()+"-"+str(y) for y,m in zip(df.date.dt.year, df.date.dt.mon
th_name())]
plt.gca().set_xticks(x[::6])
plt.gca().set_xticklabels(xtickvals[::6], rotation=90, fontdict={'horizontalalignment': '
center', 'verticalalignment': 'center_baseline'})
plt.ylim(-35,35)
plt.xlim(1,100)
plt.title("Month Economics Return %", fontsize=22)
plt.ylabel('Monthly returns %')
plt.grid(alpha=0.5)
plt.show()
```

Рис. 251. Программный код 15 для построения графика

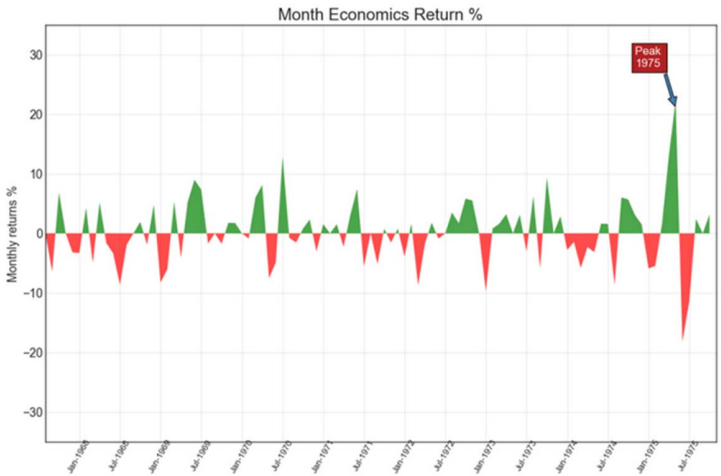


Рис. 252. Результат выполнения программного кода 15

Упорядоченная гистограмма

Упорядоченная гистограмма эффективно передает порядок ранжирования элементов. Добавив значение показателя над диаграммой, пользователь получает точную информацию от самой диаграммы (рис. 253, 254).

```
# Prepare Data
df_raw = pd.read_csv("https://github.com/selva86/datasets/raw/master/mpg_ggplot2.csv")
df = df_raw[['cty', 'manufacturer']].groupby('manufacturer').apply(lambda x: x.mean())
df.sort_values('cty', inplace=True)
df.reset_index(inplace=True)

# Draw plot
import matplotlib.patches as patches

fig, ax = plt.subplots(figsize=(16,10), facecolor='white', dpi= 80)
ax.vlines(x=df.index, ymin=0, ymax=df.cty, color='firebrick', alpha=0.7, linewidth=20)

# Annotate Text
for i, cty in enumerate(df.cty):
    ax.text(i, cty+0.5, round(cty, 1), horizontalalignment='center')


# Title, Label, Ticks and Ylim
ax.set_title('Bar Chart for Highway Mileage', fontdict={'size':22})
ax.set(ylabel='Miles Per Gallon', ylim=(0, 30))
plt.xticks(df.index, df.manufacturer.str.upper(), rotation=60, horizontalalignment='right
', fontsize=12)

# Add patches to color the X axis labels
p1 = patches.Rectangle((.57, -0.005), width=.33, height=.13, alpha=.1, facecolor='green',
transform=fig.transFigure)
p2 = patches.Rectangle((.124, -0.005), width=.446, height=.13, alpha=.1, facecolor='red',
transform=fig.transFigure)
fig.add_artist(p1)
fig.add_artist(p2)
plt.show()
```

Рис. 253. Программный код 16 для построения графика

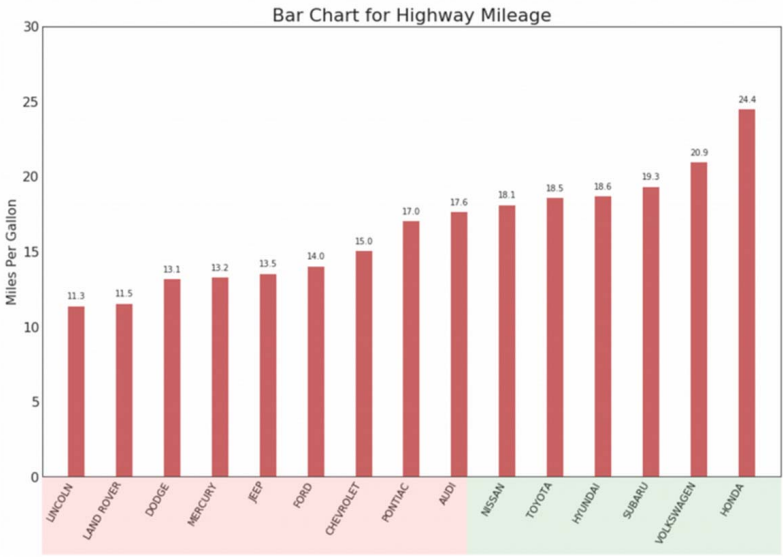


Рис. 254. Результат выполнения программного кода 16

Контрольные вопросы (тесты)

1. Объект какого типа будет создан при выполнении следующего кода?

```
import pandas as pd
s = pd.DataFrame({'A': ['a','b','c'], 'B' : [1,2,3]})
s
```

2. Введите пропущенную часть кода, чтобы прочитать данные в формате MS Excel

```
import pandas as pd
from pandas import read_excel
file1 = pd.ExcelFile('Фрукты.xlsx')
file_1 = _____('Лист1')
file_1.head()
```

3. Введите пропущенную часть кода, чтобы переопределить тип данных при создании массива

```
import numpy as np
np.array([[1.5, 2, 3], [4, 5, 6]], _____np.complex)
```

4. Введите пропущенную часть кода, чтобы создать простой массив NumPy

```
import numpy as np
_____([1,2,3])
```

5. Что будет выведено в результате выполнения кода?

```
import pandas as pd
s = pd.Series([1,2,3])
s.shape
```

6. Что будет выведено в результате выполнения кода?

```
import pandas as pd
import pandas as pd
dates = pd.date_range('20200101', periods=8)
dates.size
```

ЗАКЛЮЧЕНИЕ

В рамках данного пособия были рассмотрены следующие темы:

- ввод-вывод данных и применение арифметических операций;
- логические выражения, ветвления и циклы при выполнении операций над данными;
- работа с данными, содержащими вещественные числа;
- создание функции для работы с данными;
- кортежи и списки данных. Цикл for для работы с элементами коллекций;
- сортировка данных;
- структуры данных – множества и словари;
- использование функций языка Python для обработки последовательностей;
- объектно ориентированное программирование при анализе данных.

Рассмотрены приемы работы со следующими библиотеками:

- Python pandas,
- Python numpy,
- Python matplotlib.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Гуриков, С. Р. Основы алгоритмизации и программирования на Python : учеб. пособие / С. Р. Гуриков. – Москва : ФОРУМ [и др.], 2020. – 341, [1] с. – (Высшее образование – Бакалавриат). – URL: znanium.ru/catalog/document?id=366970 (дата обращения: 05.07.2020). – Режим доступа: по подписке. – ISBN 978-5-16-102278-8.
2. Грекул, В. И. Управление внедрением информационных систем : учеб. пособие / В. И. Грекул, Г. Н. Денищенко, Н. Л. Коровкина. – 3-е изд. (электрон.). – Москва : Интернет-Университет Информационных Технологий [и др.], 2021. – 277 с. – URL: www.iprbookshop.ru/102073.html (дата обращения: 05.01.2021). – Режим доступа: по подписке. – ISBN 978-5-4497-0910-3.
3. Грекул, В. И. Методические основы управления ИТ-проектами : учебник / В. И. Грекул, Н. Л. Коровкина, Ю. В. Куприянов. – 3-е изд. (электрон.). – Москва : Интернет-Университет Информационных Технологий [и др.], 2021. – 467 с. – URL: www.iprbookshop.ru/102019.html (дата обращения: 05.01.2021). – Режим доступа: по подписке. – ISBN 978-5-4497-0894-6.
4. Гринберг, А. С. Информационные технологии управления : учеб. пособие / А. С. Гринберг, Н. Н. Горбачев, А. С. Бондаренко. – Москва : ЮНИТИ-ДАНА, 2017. – 479 с. – URL: www.iprbookshop.ru/71234.html (дата обращения: 05.07.2020). – Режим доступа: по подписке. – ISBN 5-238-00725-6.
5. Цехановский, В. В. Управление данными : учебник / В. В. Цехановский, В. Д. Чертовской. – Санкт-Петербург [и др.] : Лань, 2015. – 432 с. – URL: e.lanbook.com/book/212084 (дата обращения: 05.01.2022). – Режим доступа: по подписке. – ISBN 978-5-8114-1853-4.

Содержание

Учебное издание

Климов Виталий Сергеевич,
Мкртычев Сергей Вазгенович

СОВРЕМЕННЫЕ ТЕХНОЛОГИИ АНАЛИЗА ДАННЫХ

Учебно-методическое пособие

Редактор *О.В. Горбань*
Технический редактор *Н.П. Крюкова*
Компьютерная верстка: *Л.В. Сызганцева*
Дизайн обложки: *И.И. Шишкина*

В оформлении обложки использованы изображения
от rawpixel.com и xb100 на сайте ru.freepik.com

Подписано в печать 07.02.2025. Формат 60×84/16.
Печать оперативная. Усл. п. л. 9,65.
Тираж 100 экз. Заказ № 1-38-22.

Издательство Тольяттинского государственного университета
445020, г. Тольятти, ул. Белорусская, 14,
тел. 8 (8482) 44-91-47, www.tltsu.ru