

Элементы языка Python для  
обработки и анализа данных

# Содержание

<b>1</b>	<b>Введение</b>	<b>2</b>
1.1	Jupyter Notebook . . . . .	2
<b>2</b>	<b>Основы Python</b>	<b>3</b>
2.1	Значения . . . . .	4
2.2	Переменные . . . . .	4
2.3	Операции . . . . .	6
<b>3</b>	<b>Операторы</b>	<b>11</b>
3.1	Условный оператор . . . . .	11
3.2	Циклы . . . . .	14
3.3	Заключение по разделу . . . . .	17
<b>4</b>	<b>Функции, модули и библиотеки</b>	<b>17</b>
4.1	Функции . . . . .	17
<b>5</b>	<b>Объектно-ориентированный подход</b>	<b>21</b>
<b>6</b>	<b>Структуры данных в Python</b>	<b>24</b>
6.1	Списки . . . . .	24
6.2	Кортежи . . . . .	31
6.3	Словари . . . . .	31
<b>7</b>	<b>Библиотека NumPy</b>	<b>35</b>
7.1	Многомерный массив-объект (ndarray) . . . . .	36
7.2	Операции над массивами . . . . .	41
<b>8</b>	<b>Библиотека pandas</b>	<b>45</b>
8.1	Серии (Series) . . . . .	45
8.2	Фреймы данных (DataFrame) . . . . .	48
<b>9</b>	<b>Визуализация данных</b>	<b>57</b>
9.1	Основы matplotlib. Метод plot() . . . . .	58
9.2	Построение графиков функций . . . . .	62
9.3	Точечные диаграммы. Метод scatter() . . . . .	66

# 1 Введение

В этой лекции мы поговорим о том, как использовать язык программирования **Python** для машинного обучения. **Python** – это популярный язык программирования общего назначения, который стал популярным и предпочтительным среди специалистов по обработке данных. Ведь несмотря на то, что можно реализовать алгоритмы машинного обучения самостоятельно, с помощью любого языка программирования, для **Python** разработано множество библиотек, которые могут значительно облегчить вашу жизнь.

Так как наш курс не предполагает первоначальных знаний языка **Python**, мы начнем с рассмотрения основ самого языка **Python**, а затем перейдем к изучению библиотек, которые будут востребованы непосредственно в машинном обучении.

Один из вопросов, который возникает при знакомстве с новым языком программирования: какую среду использовать, чтобы написать и исполнить программный код? Ведь писать программный код можно хоть в текстовом редакторе, хоть на бумажке. А что дальше?

Так, уже написанный код (допустим, все в том же блокноте), можно исполнить с помощью интерпретатора **Python** через консоль или терминал операционной системы. Однако при таком подходе легко ошибиться в написании самого кода и крайне сложно искать ошибки.

Несколько облегчают процесс написания кода, так называемые редакторы кода, которые имеют подсветку синтаксиса и различные подсказки при вводе конструкций языка программирования, однако исполняется код все равно отдельно с помощью интерпретатора **Python**.

Наконец, интегрированная среда разработки (IDE) – программное обеспечение, устанавливаемое на компьютер, которое позволяет работать в рамках одного продукта, где можно писать и исполнять код, получать подсказки и сообщения об ошибках, выполнять отладку кода и многое другое. Однако, такое программное обеспечение используется, как правило, для больших проектов.

Мы же рассмотрим еще один сервис для написания и исполнения **Python** кода – **Jupyter Notebook** (блокнот).

## 1.1 Jupyter Notebook

Использование **Jupyter** блокнотов является одним из способов оформления и писания программного кода на языке **Python**. **Jupyter** блокноты стали де-факто стандартом для работы с языком **Python** в сфере обработки и анализа данных. Блокнот представляет собой ячейки, в которых можно писать код, комментарии, использовать язык разметки **Markdown** для оформления, вставлять формулы в формате **LaTeX** и многое другое, пример такого блокно-

### Примеры

```
In [2]: # ячейка с кодом, при выполнении которой (нажмите Run или Shift + Enter) появится Out (результат выполнения кода или вычислений)
2 + 2

Out[2]: 4

In [5]: # ячейка в которой создаются переменные
a = 4
b = 6

In [7]: # данные из одной ячейки видны в другой
c = a + b

In [8]: # для вывода можно указать имя переменной
c

Out[8]: 10
```

Рис. 1: Пример Jupyter Notebook.

та представлен на рисунке 1. Отличительной особенностью таких блокнотов является их доступность, предоставляемая большим количеством различных бесплатных сервисов. Причем такие сервисы уже содержат все необходимые библиотеки для обработки и анализа данных, но при желании, **Jupyter** блокноты можно создавать и исполнять на своем личном компьютере.

В дополнительных материалах вы можете найти как инструкции по установке программного обеспечения для работы с блокнотами на своем компьютере, так и инструкции по доступу к облачным сервисам, которыми мы и рекомендуем пользоваться. Кроме того, ко всем упражнениям будут приложены примеры выполненные в **Jupyter** блокнотах.

## 2 Основы Python

Перейдем к рассмотрению основных идей создания простейших компьютерных программ и проверим их на практике. Говоря об основных идеях создания программы, мы имеем в виду то, что написание кода является в некотором смысле диалогом между программистом и компьютером. В этом смысле последний похож на собаку, обученную вполне ограниченному набору трюков. При этом, если собака не понимает, что от нее хотят, то она и не сможет ничего продемонстрировать. При исполнении программного кода, компьютер выполняет ровно то, что ему было велено делать – ни больше, ни меньше. Если он столкнулся с тем, что программа написана некорректно, выдается сообщение об ошибке, часто с указанием конкретного места, где она произошла. Даже самые опытные программисты могут в процессе создания кода совершать ошибки, в основном по невнимательности. Важно понимать, что в этом нет ничего страшного, просто чем больше опыта, тем реже такое случается. Давайте рассмотрим базовые возможности языка **Python** версии 3.x.

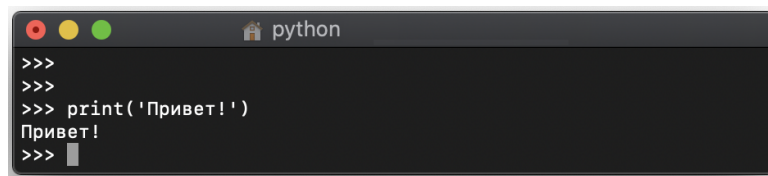


Рис. 2: Пример вывода в консоль.

## 2.1 Значения

Написание программного кода, как правило, связано с некоторыми вычислениями и последующим выводом результатов на экран. Обычно, так говорят, когда подразумевается вывод результатов программы непосредственно в консоль (рисунок 2). За вывод данных на экран отвечает команда `print()`. Простейшими видами данных являются числовые, строковые и булевские значения (`True` – англ. «Верно (Истина)» или `False` – англ. «Ошибочно (Ложь)»). Продемонстрируем, как команда `print()` работает с этими значениями:

```
1 #Выводим на экран числовое значение 5
2 print(5)
3 #Выводим на экран строковое значение 'Привет!'
4 print('Привет!')
5 #Выводим на экран булевские значения True и False
6 print(True)
7 print(False)
```

Отметим, что в результате запуска программы на экран будут выведены следующие строки:

```
1 5
2 Привет!
3 True
4 False
```

Отметим также, что при написании программ текстовые значения помещаются в кавычки (двойные или одинарные). Значения могут использоваться в программе явно или храниться, в так называемых переменных.

## 2.2 Переменные

Переменные используются для хранения данных внутри программы. Можно представить переменную как контейнер, в который можно что-то положить. В нашем случае это будет число, текст или булевское значение. При этом все контейнеры должны быть именованы, т.е. иметь свое уникальное

имя. Кроме того, контейнеры необходимо где-то хранить, происходит это в оперативной памяти компьютера.

**Определение 2.2.1** *Переменная – это именованная область памяти для хранения данных, имя которой можно использовать для доступа и изменения значения в ходе выполнения программы.*

Например, поместим в переменную **a** значение 5, а в переменную **b** значение 8. Далее присвоим переменной **result** сумму **a + b** и выведем на экран значение переменной **result**:

```
1 #Переменной a присваивается значение 5
2 a = 5
3 #Переменной b присваивается значение 8
4 b = 8
5 #Переменной result присваивается значение суммы a + b
6 result = a + b
7 #Выводим на экран значение, содержащееся в переменной result
8 print(result)
```

В результате исполнения кода, мы, конечно, получим число 13. В качестве замечаний к приведенному коду можно отметить следующие:

- строки, начинающиеся с символа **#**, являются комментариями и пропускаются при выполнении кода;
- использование пробелов является «правилом хорошего тона», рекомендуется знаки **=**, **+** и т.п. обрамлять пробелами, однако **a=5** или **result=a+b** будут выполнены точно также, как и с пробелами;
- в данном случае необязательно пользоваться 3-ей переменной, так как **a + b** можно посчитать прямо в команде **print()**, т.е. **print(a + b)**.

Важно понимать, как пользоваться переменными. Например, если в переменной уже находится какая-то информация и туда поместить новую, то старая информация просто перезапишется.

```
1 #Переменной a присваивается значение 5
2 a = 5
3 #Выводим на экран переменную a
4 print(a)
5 #Переменной a присваивается значение hello
6 a = 'hello'
7 #Выводим на экран переменную a
```

```

8 print(a)
9 #Переменной a присваивается значение True
10 a = True
11 #Выводим на экран переменную a
12 print(a)

```

Отдельно стоит упомянуть, что переменные лучше именовать таким образом, чтобы по названию было понятно, для чего эта переменная используется и какая информация там хранится. Например: **speed**, **force**, **normal\_acceleration** и т.п. При этом пробелы в названиях переменных использовать нельзя, а переменные **boss**, **Boss** и **BOSS** будут разными – то есть в названии переменных важен регистр. Кроме рекомендаций, есть и строгие правила именования, так имя переменной может состоять только из цифр, букв и знаков подчеркивания, при этом имя переменной не может начинаться с цифры.

Итак, мы познакомились с переменными и простейшими операциями над ними. Даже для создания новой переменной, мы используем операцию – операцию присваивания. Давайте остановимся на них подробнее.

## 2.3 Операции

**Определение 2.3.1** *Операция – это специальный способ записи некоторых действий, выполняемых над значениями.*

**Определение 2.3.2** *Операнд – объекты, над которыми производится операция.*

**Определение 2.3.3** *Выражение – комбинация операций и операндов по определенным синтаксическим и семантическим правилам.*

Так, в выражении  $5 + 8$ , числа 5 и 8 являются операндами, знак  $+$  операцией.

```

1 #Выводим на экран результат операции сложения
2 print(5 + 8)

```

Рассмотрим основные операции языка программирования **Python**, которые принято делить на несколько групп.

### Арифметические операции

Арифметические операции в **Python** и других языка программирования – наиболее часто использующийся класс операций. Этот класс объединяет результат операции – целый или вещественный, что зависит от выполняемой операции и самих операндов. Операции и их описание указаны в таблице:

Операция	Описание
+	Сложение (суммирует значения операндов)
-	Вычитание (вычитает правый операнд из левого). Этот же знак используется для операции отрицания
*	Умножение (перемножает операнды)
/	Деление (делит левый операнд на правый)
%	Деление по модулю (делит левый операнд на правый и возвращает целочисленный остаток деления)
**	Возведение в степень (возводит левый операнд в степень правого)
//	Целочисленное деление (делит левый операнд на правый и возвращает целочисленный результат деления, отбрасывая дробную часть)

Приведем некоторые примеры кода, где в комментариях указан результат выполнения операции:

```

1 print(5 + 3.6) #8.6
2 print(5 - 3.6) #1.4
3 print(3 * 2) #6
4 print(2.3 * 4) #9.2
5 print(15 / 5) #3
6 print(5 / 2) #2.5
7 print(5 % 2) #1
8 print(5 ** 2) #25
9 print(5 // 2) #2

```

## Операции сравнения

Операции сравнения позволяют выполнить сравнение переменных и пригодятся нам при написании условий. Все операции сравнения возвращают булевские значения, в качестве операндов могут выступать значения любого типа.

Операция	Описание
==	Проверяет равенство операндов
!=	Проверяет неравенство операндов
<	Проверяет, больше ли правый операнд
>	Проверяет, больше ли левый операнд
<=	Проверяет, больше ли или равен правый операнд
>=	Проверяет, больше ли или равен левый операнд



Приведем некоторые примеры кода, где в комментариях указан результат вывода:

```
1 print(5 == 5) #True
2 print(6 == 5) #False
3 print(6 != 5) #True
4 print('hi' == 'hi') #True
5 print(5 > 3) #True
6 print(5 >= 5) #True
```

Особенно следует отметить, что при сравнении строк принимается во внимание символы и их регистр. Так, любой цифровой символ условно меньше, чем любой алфавитный символ:

```
1 print('1' < 'a') #True
2 print('1' < 'AB') #True
```

Алфавитный символ в верхнем регистре условно меньше, чем алфавитный символ в нижнем регистре:

```
1 print('A' < 'b') #True
2 print('A' < 'B') #True
3 print('B' < 'A') #False
4 print('a' < 'B') #False
```

Кроме того, сравнение строк будет давать аналогичный результат, так как их сравнение выполняется посимвольно. В приведенном примере единица условно меньше буквы F и на этом сравнение строк заканчивается. Результат, как видно – «Истина».

```
1 print('1abc' < 'F5bc') #True
```

В следующем примере первые символы равны между собой, и сравнение выполняется между вторыми, где 5 условно меньше буквы a, а значит результат – «Ложь».

```
1 print('1abc' < '15bc') #False
```

## Логические операции

Логические операции позволяют производить операции над логическими операндами и вырабатывают логические значения.

Операция	Описание
<code>and</code>	Логическое И (истина, если все операнды истина)
<code>or</code>	Логическое ИЛИ (истина, если хотя бы один операнд – истина)
<code>not</code>	Логическое отрицание (изменяет логическое значение операнда на противоположное)

Приведем некоторые примеры кода, где в комментариях указан результат вывода:

```

1 print(True and True) #True
2 print(True and True and False) #False
3 print(True or True or False) #True
4 print(not False) #True
5 print((5 > 4) and (6 > 4)) #True
6 print((5 > 4) and (6 > 4) or (3 > 4)) #True

```

## Операция принадлежности

В добавок к перечисленным операциям, в **Python** присутствуют, так называемые, операции принадлежности, предназначенные для проверки на наличие элемента в множестве.

Операция	Описание
<code>in</code>	Проверка принадлежности (истина если элемент присутствует в множестве)
<code>not in</code>	Проверка принадлежности (истина если элемент не присутствует в множестве)

Приведем некоторые примеры кода, где в комментариях указан результат вывода:

```

1 print(1 in [1, 2, 3, 4]) #True
2 print(1 in [2, 3, 4, 5]) #False
3 print('crocodile' not in ['cat', 'dog']) #True
4 print('oco' in 'crocodile') #True

```

## Приоритет операций

Во избежание неоднозначности, любой язык программирования выполняет операции с некоторым заданным приоритетом. Так, операции, имеющие более высокий приоритет, выполняются раньше. Например, как мы знаем из математики, умножение имеет больший приоритет, чем сложение, так и здесь.

```
1 print(5 + 4 * 3) #Вывод: 17
```

Это называется правилом порядка. Второе правило – правило ассоциативности. Оно применяется в том случае, если несколько операций имеют одинаковый приоритет. В таком случае операции выполняются в установленном для них порядке – слева направо. Например деление:

```
1 print(20 / 2 / 5) #Вывод: 2
```

Единственное исключение из правил, операция возведение в степень, которая выполняется справа налево:

```
1 print(2 ** 3 ** 2) #Вывод: 512
```

В следующей таблице приводится сводная информация о приоритетах операций в **Python** от самого высокого до самого низкого. еще раз отметим, что чем больше приоритет, тем раньше выполняется операция. Изменить порядок приоритета можно с помощью скобок.

Операция	Приоритет (порядок)
<b>**</b>	8
- (унарный минус)	7
<b>*, /, //, %</b>	6
<b>+, -</b>	5
<b>in, not in, &lt;, &lt;=, &gt;, &gt;=, !=, ==</b>	4
<b>not</b>	3
<b>and</b>	2
<b>or</b>	1

В целом, такой порядок ожидаем, так как он достаточно общепринят в математике. Вначале что-то вычисляется, затем результаты вычислений сравниваются, а в последнюю очередь результаты сравнений комбинируются при помощи логических операций.

Приведем некоторые примеры:

```
1 print(33 / 7 * 7) #Вывод: 33
2 print(3 > 1 + 2 and 3 < 1 + 2) #Вывод: False
3 print(3 > 1 + 2 and 3 < 1 + 2 or True) #Вывод: True
4 print(-3 ** 2) #Вывод: -9
5 print((-3) ** 2) #Вывод: 9
6 print(5 % 3 in [1, 2, 3]) #Вывод: True
7 print(True and False in [False, False]) #Вывод: True
```

## 3 Операторы

**Определение 3.0.1** *Оператор – это инструкция языка программирования, которая определяет некоторый вполне законченный этап обработки данных.*

Рассмотрим некоторые, наиболее важные операторы языка Python.

### Оператор присваивания

Оператор присваивания позволяет сохранить значение в переменную, или изменить его, используя более короткую форму записи:

Оператор	Описание
=	Простейшее присваивание
+=	Присвоение результата сложения
-=	Присвоение результата вычитания
*=	Присвоение результата умножения
/=	Присвоение результата деления
//=	Присвоение результата целочисленного деления
%=	Присвоение результата остатка от деления
**=	Присвоение результата степени числа

Приведем некоторые примеры:

```
1  #Переменной a присваивается значение 5
2  a = 5
3  #Переменной a присваивается результат сложения a с числом 7
4  a += 7
5  #Выводим на экран значение a, равное 12
6  print(a)
7  #Переменной a присваивается результат умножения a на число a
8  a *= a
9  #Выводим на экран значение a, равное 144
10 print(a)
```

### 3.1 Условный оператор

Возможность ветвления – один из существенных мотивов использования программирования как такового. Связка **Если – То** (**if – then**) является основополагающей в автоматизации различных процессов. Условный оператор состоит из условия и тела. Телом условия называется некоторая последовательность инструкций, выполняемых в случае выполнения условия.

Рассмотрим, как работает оператор **if** на простом примере:

```

1  #Переменной a присваивается значение 10
2  a = 10
3  #Проверка условия (если a меньше 12)
4  if a < 12:
5      #Выполняется, если условие истинно
6      print('Значение a меньше 12')
7  #Выполняется после завершения условного оператора
8  print('Продолжение')

```

Итак, после слова `if` следует логическое выражение, заканчивающееся двоеточием, после которого начинается тело условного оператора. Каждая строка кода из тела условного оператора должна иметь отступ, оформленный, как правило табуляцией или серией из четырех пробелов.

Связка **if – then** может быть расширена до формы условного оператора **Если – То – Иначе (if – then – else)**. Такая связка позволяет разбить программу на две части (ветвления), в каждой из которых могут быть выполнены некие инструкции. Рассмотрим следующий пример:

```

1  #Переменной a присваивается значение 10
2  a = 10
3  #Проверка условия (если a меньше 12)
4  if a < 12:
5      #Выполняется, если условие истинно
6      print('Значение a меньше 12')
7  else:
8      #Выполняется, если условие ложно
9      print('Значение a не меньше 12')
10 #Выполняется после завершения условного оператора
11 print('Продолжение')

```

Таким образом, программа разбилась на две части (ветвления), в каждой из которых выполняется вывод сообщений на экран, который зависит от результата условия. После условного оператора мы возвращаемся к инструкциям основной программы.

Если же необходимо большее количество ветвлений, можно либо использовать вложенные условия, либо воспользоваться формой оператора **if – elif – else**, которая допускает любое число вариантов.

```

1  a = 10
2  if a >= 0 and a < 10:
3      print('Значение a от 0 включительно до 10 не включительно')
4  elif a >= 10 and a < 20:
5      print('Значение a от 10 включительно до 20 не включительно')

```

```

6 elif a >= 20 and a <= 40:
7     print('Значение a от 20 включительно до 40 включительно')
8 else:
9     print('Значение a отрицательно или больше 40')
10 print('Продолжение')

```

Важно отметить, что такая связка проверяет значение переменной `a` последовательно в каждом условии, и как только одно из них истинно, работа условного оператора прекращается.

Рассмотрим еще один пример кода на условный оператор, позволяющий решить простейшую школьную задачу – найти корни квадратного уравнения. Пусть  $a, b, c$  – известные коэффициенты в квадратном уравнении  $ax^2 + bx + c = 0$ , тогда код программы может быть следующим:

```

1 a = 1
2 b = -2
3 c = 2
4 if a != 0: #Проверяем, является ли уравнение квадратным
5     disc = b ** 2 - 4 * a * c #Вычисляем дискриминант
6     if disc < 0: #Что делать, если дискриминант отрицательный
7         print('Вещественных корней нет')
8     elif disc == 0: #Что делать, если дискриминант равен нулю
9         x = -b / (2 * a)
10        print('Двойной корень x =', x)
11    else: #Что делать, если дискриминант положительный
12        print('Два вещественных корня:')
13        x_1 = (-b + disc ** 0.5) / (2 * a)
14        x_2 = (-b - disc ** 0.5) / (2 * a)
15        print('x_1 =', x_1)
16        print('x_2 =', x_2)
17 else: #Относится к самому первому условию if a != 0: и
    ↳ регламентирует, что делать, если условие оказалось ложным
18     #Что делать, если a == 0, т.е. уравнение не квадратное
19     print('Уравнение не является квадратным')
20 print('Конец') #Выводится всегда, так как не относится ни к
    ↳ одному из условий, и находится внутри тела основной
    ↳ программы.

```

Обращаем ваше внимание на то, что все проверки дискриминанта находятся внутри главного условия `a != 0`. При этом в условном операторе еще присутствует ветка `else`, которая будет выполнена, если `a` все-таки равно нулю. Попробуйте поменять коэффициенты и посмотрите, что будет получаться.

## 3.2 Циклы

Одно из основных преимуществ компьютера заключается в том, что ему можно поручить выполнение монотонной рутинной работы, требующей большого количества повторений. Поэтому использование циклов приходится как нельзя кстати. Мы рассмотрим циклы, реализуемые операторами **for** и **while**. Операторы цикла состоят из заголовка и тела цикла. В теле цикла указаны команды, которые выполняются при выполнении условий, указанных в заголовке цикла.

### Оператор **for**

Начнем с оператора **for**. В заголовке такого цикла указывается переменная и множество значений, которые она способна принимать. Способы задания множества значений могут быть весьма разнообразными, но об этом мы еще подробно поговорим далее, а пока рассмотрим следующий простейший пример:

```
1 for i in 4, 20, -3, 'word':  
2     print(i)  
3 print('Конец цикла')
```

В данном случае *i* – переменная, а числа 4, 20, -3 и слово **word** – элементы множества, значения которых последовательно принимает переменная *i*. Таким образом, в приведенном отрывке кода переменная *i* последовательно принимает значения из написанного множества, а команда **print(i)** выводит эти значения на экран. Дословно код из примера можно перевести так: «Для каждого *i* из заданного множества вывести значение *i*».

Приведем еще один пример. Так, следующий код позволяет найти сумму заданных элементов множества, указанных в заголовке цикла:

```
1 sum = 0  
2 for i in 1, 2, 3, 4, 5:  
3     sum += i  
4 #Выполняется после завершения оператора цикла  
5 print('Сумма = ', sum)
```

### Оператор **while**

Цикл на основании оператора **while** реализуется несколько другим образом. Сначала указывается некоторое условие (логическое выражение) и, пока это условие истинно, будет выполняться то, что написано в теле цикла. Если оно ложно – тело цикла пропускается и выполняется переход к следующей команде.

```

1 i = 0 #Задаем начальное значение переменной
2 while i < 5: #Условие выполнения цикла
3     print(i) #Вывод на экран значения переменной i
4     i += 1 #Нарращивая переменную i, обеспечиваем выход из цикла
5 print('Конец цикла')#Выполняется после завершения цикла

```

Приведенный фрагмент кода выводит на экран целые числа от нуля до четырех включительно. Значение переменной `i`, равное пяти, выведено не будет, так как логическое выражение `i < 5` ложно, и тело цикла выполнено не будет.

Заметим, что при использовании цикла `while` нужно быть крайне внимательным, чтобы не получить бесконечный цикл, например такого вида:

```

1 i = 0
2 while True: #Условие всегда истинно, поэтому цикл будет
    ↪   повторяться бесконечно
3     print(i)
4     i += 1 #Нарращиваем переменную i
5 print('Конец цикла')#Не будет выполнено никогда

```

После тела цикла `while` можно использовать оператор `else`, по аналогии с тем, как это было в условных операторах. Тогда код, размещенный внутри блока `else`, будет выполнен один раз, как только логическое условие станет ложно. Например, следующий код позволяет найти сумму квадратов чисел от 0 до 9 включительно:

```

1 i = 0
2 sum_sq = 0
3 while i < 10:
4     sum_sq += i ** 2
5     i += 1
6 else:
7     print('Переменная i =', i, 'Конец цикла.')
8 print('Сумма квадратов: ', sum_sq)

```

Цикл будет работать, пока условие `i < 10` истинно, а затем, при `i = 10` будет выполнен код, размещенный в блоке `else`, то есть на экран будет выведено значение переменной `i`, а затем полученное значение суммы квадратов.

## Операторы `continue` и `break`

Для принудительного перехода к следующей итерации, или для выхода из цикла, используются операторы `continue` и `break`, которые, как правило используются в контексте условного оператора.



Оператор **break** позволяет принудительно закончить цикл ровно на текущей итерации. Например, следующий код выводит нечетные числа только до 5, игнорируя условие цикла `i < 10`, так как ранее сработает оператор **break**:

```
1 i = 1
2 while i < 10:
3     print(i)
4     i += 2
5     if i == 7:
6         break
7 else:
8     print('Переменная i =', i)
9 print('Конец цикла')
```

Кроме того, оператор **else** также будет пропущен, так как **break** прерывает работу цикла полностью.

В случае, если оператор **break** расположен во вложенном цикле, то он прерывает лишь его, а внешний цикл продолжает свою работу. Так, в приведенном коде внутренний цикл будет работать 10 раз, однако число итераций будет соответствовать текущему значению переменной `i`:

```
1 i = 1
2 while i <= 10:
3     j = 1
4     while True:
5         print(i * j)
6         j += 1
7         if j > i:
8             print('Прерывание внутреннего цикла')
9             break
10    i += 1
11    print('Конец итерации внешнего цикла')
12 print('Конец цикла')
```

Оператор **continue** позволяет пропустить итерацию и перейти к следующей итерации цикла. Следующий пример демонстрирует работу оператора. Так, в цикле пробегаются все значения переменной `i` от 1 до 10, и на каждой итерации цикла значение переменной выводится на экран, однако, для `i`, равного трем, срабатывает проверка условия `i == 3` и команда **continue** принудительно переводит цикл на следующую итерацию, минуя вывод данных на экран.

```

1 i = 1
2 while i <= 10:
3     if i == 3:
4         i += 1
5         continue
6     print(i)
7     i += 1

```

Таким образом, для `i`, равного трем, вывод на экран не выполняется.

### 3.3 Заключение по разделу

Мы рассмотрели лишь основы языка **Python**, однако уже с помощью них вы можете попробовать написать первые простенькие программы. И если вы начинаете свой путь, мы настоятельно рекомендуем попробовать написать свои первые программы без использования готовых функций, хотя мы их, конечно, разберем далее. Кроме того, в дополнительных материалах вы можете найти рассмотренные примеры, оформленные в формате **Jupyter** блокнотов.

## 4 Функции, модули и библиотеки

### 4.1 Функции

До сих пор мы рассматривали примеры создания программ, когда операции выполняются одна за другой, однако часто возникает необходимость повторного использования уже написанного кода. В таких случаях имеет смысл выделить фрагменты кода, выполняющие определенную последовательность действий, в отдельные блоки.

**Определение 4.1.1** *Функция – это программный код, который имеет отдельное имя, и может быть вызван из другой части программы.*

#### Работа с функциями

Функции состоят из заголовка и тела функции. В заголовке содержится специальное ключевое слово **def** (от англ. *define*), после которого следует название функции. Заголовок заканчивается двоеточием и переходом на новую строку. Тело функции имеет отступ и содержит инструкции, которые выполняются при вызове функции. Обратимся к примеру.

```

1 def hello(): #Объявление функции
2     print('Привет всем!') #Тело функции
3
4 hello() #Вызов функции

```

Вывод:

```
1 Привет всем!
```

Обращаем внимание, что после названия функции стоят скобки. Зачем они нужны? Дело в том, что если предельно упрощать, то функцию можно сравнить с некоторым черным ящиком. Мы должны что-то туда положить, а потом что-то из него достать (возможно уже что-то другое). Например, духовой шкаф: на вход мы подаем тесто и яблоки, а на выходе получаем шарлотку. В предыдущем примере мы, кстати, ничего в функцию не передавали, поэтому в скобках ничего и не указано. Рассмотрим более содержательный пример:

```
1 def Discriminant(a, b, c): #Объявление функции и параметров
2     if a != 0:
3         discr = b ** 2 - 4 * a * c
4         return discr
5     else:
6         print('a должно быть отлично от нуля!')
7
8 print(Discriminant(1, 3, -2)) #Вызов функции и передача
   ↪ аргументов
```

В результате на экран будет выведено число 17. Значения, которые передаются в функцию при вызове, называются аргументами (в примере это числа 1, 3, -2), а соответствующие этим аргументам переменные, входящие в функцию, называются параметрами.

Из примера понятно, однако обратим на это внимание отдельно, что при таком способе указания аргументов важно соблюдать порядок, то есть в момент выполнения функции параметры будут иметь следующие значения:  $a = 1, b = 3, c = -2$ . В результате своей работы функция должна что-то вернуть. Для того, чтобы указать, что именно, используется команда **return**. Если в этом же примере в качестве аргументов передать, например, значения 0, 3, -2, то на экран будет выведено сообщение **а должно быть отлично от нуля!** и затем еще сообщение **None**. Дело в том, что в этой ветке условия мы ничего не собираемся возвращать, однако функции в **Python** устроены таким образом, что они всегда должны что-то возвращать. Поэтому если **return** не указан, возвращается значение **None** (ничего). Заметим также, что в качестве аргументов могут выступать переменные.

Кроме того, часто требуется, чтобы некоторые параметры функции имели какое-либо значение по умолчанию. Например, если требуется найти дискриминант многочлена  $ax^2 + bx + c$ , но не указано значение числа **a**, то для него будет использоваться значение по умолчанию, т.е. **a = 1**.

```

1 def Discriminant(b, c, a = 1): #Объявление функции. Параметр a =
  ↪ 1 по умолчанию
2     if a != 0:
3         discr = b ** 2 - 4 * a * c
4         return discr
5     else:
6         print('a должно быть отлично от нуля!')
7
8 print(Discriminant(3, -2)) #Вызов функции и передача двух
  ↪ аргументов (b и c)

```

Заметим, что при объявлении функции, значения по умолчанию указываются в последнюю очередь. При вызове же функции `Discriminant` передаются как минимум два аргумента.

**Замечание 4.1.1** *Если функция имеет несколько значений по умолчанию, рекомендуется при вызове функции указывать, какому именно параметру соответствует передаваемый аргумент.*

Таким образом аргументы можно разделить на позиционные (в нашем случае `b` и `c`) и передаваемые по ключевому слову (`a`). Вызов функции с указанием параметра `a` будет выглядеть следующим образом: `Discriminant(3, -2, a = 1)`.

## Импорт модулей

Использование функций позволяет структурировать программу и облегчить ее восприятие. Логичным продолжением является выделение некоторых функций в отдельные файлы, которые затем можно импортировать в основную программу.

**Определение 4.1.2** *Модулем называется файл, содержащий в себе программный код, предназначенный для импорта в основную программу.*

Подключение модуля осуществляется командой `import`. При импорте все функции модуля копируются в основной файл, однако код не отображается в тексте основной программы. Рассмотрим следующий пример. Пусть функция

```

1 def Discriminant(b, c, a = 1): #Объявление функции. Параметр a =
  ↪ 1 по умолчанию
2     if a != 0:
3         discr = b ** 2 - 4 * a * c
4         return discr
5     else:
6         print('a должно быть отлично от нуля!')

```

хранится в отдельном файле `my_function.py`. Тогда подключение этого модуля в основном файле `main.py` будет выглядеть следующим образом.

```
1 import my_function #Подключение модуля
2 print(my_function.Discriminant(3, -2)) #Вызов функции
```

Особо отметим вид вызова нужной функции. Через точку указывается сначала название модуля, а затем имя вызываемой функции. При подключении модуля можно для удобства дальнейшего использования внутри основной программы давать им другие имена. Приведем пример.

```
1 import my_function as mf #Подключение модуля с назначением
  ↪ локального имени mf
2 print(mf.Discriminant(3, -2)) #Вызов функции с учетом нового
  ↪ названия модуля
```

Подход, при котором функции хранятся в отдельных файлах, крайне продуктивен. Таким образом можно делиться этими файлами с другими программами без необходимости делиться всей программой. Умение импортировать функции также позволяет использовать так называемые библиотеки функций, написанные другими программистами.

**Определение 4.1.3** *Библиотека – это набор модулей.*

Некоторые библиотеки имеют достаточно большое количество функций и как следствие импорт будет занимать некоторое время. В таком случае можно импортировать только отдельные функции (команда `from`), тогда при вызове пропадает необходимость указывать название модуля.

```
1 from my_function import Discriminant # Импорт отдельной функции
  ↪ Discriminant из модуля my_function
2 print(Discriminant(3, -2)) #Вызов функции с учетом нового
  ↪ названия модуля
```

**Замечание 4.1.2** *Отметим, что в библиотеках Python содержатся обычно не только функции, но и другие конструкции языка, например, такие как классы, о которых мы поговорим позднее.*

Функциям при импорте также можно давать локальные имена. Приведем пример.

```
1 from my_function import Discriminant as dscr # Импорт функции и
  ↪ назначение локального имени
2 print(dscr(3, -2)) #Вызов функции с учетом нового имени
```

Существует также возможность импортировать несколько функций. В таком случае они перечисляются через запятую, например

```
1 from my_module import my_function_1, my_function_2,  
  ↪ my_function_3
```

Кроме того можно импортировать все функции модуля при помощи команды `from my_module import *` и использовать их без указания названия модуля.

```
1 from my_module import *
```

При использовании больших библиотек такой способ импорта не рекомендуется, так как возможны коллизии между названиями функций основной программы и функций подключаемого модуля. В таком случае логичнее импортировать модуль и обращаться к его функциям через точку.

## 5 Объектно-ориентированный подход

Идея объектно-ориентированного подхода заключается в работе с так называемыми классами.

**Определение 5.0.1** *Класс – это пользовательская структура данных, имеющая отдельное имя и содержащая непосредственно данные и правила работы с ними.*

Классы предназначены для моделирования поведения объектов реального мира. При этом классы описывают некоторую категорию объектов.

**Определение 5.0.2** *Экземпляром класса (сущностью) называется конкретный представитель этого класса.*

Например, собака – это класс. При этом описывая абстрактную собаку, мы можем сказать, что она должна иметь имя, возраст и, допустим, породу. Конкретный пудель с именем Ксюша и возрастом 10 лет будет экземпляром класса.

**Определение 5.0.3** *Атрибутами (полями) класса называются переменные, связанные с классом.*

Кроме атрибутов, класс обычно содержит методы.

**Определение 5.0.4** *Метод – это функция, принадлежащая классу.*

Например, собаку можно похвалить или наказать, то есть выполнить над этим классом некоторое действие из вне. Такие методы называются внешними. Собака может, например, спать – это внутренний метод. Рассмотрим на примере, как создавать описывать классы в Python.

```
1 class Dog(): # Объявление класса
2     def __init__(self, name, age, status='Радость'): #Метод
    ↪ инициализации класса
3         self.name = name #Атрибут имя
4         self.age = age #Атрибут возраст
5         self.status = status #Атрибут статус
6     def pet(self): #Метод похвалить
7         self.status = 'Радость'
8     def punish (self): #Метод наказать
9         self.status = 'Грусть'
```

Обратим внимание, что имя класса обычно начинается с заглавной буквы. Методы начинаются со служебной конструкции `def`, так как по сути являются функциями. Отдельный метод `def __init__` выполняется всегда и служит для создания экземпляра класса, именно поэтому он выделен нижними подчеркиваниями, чтобы не вызывать коллизий в названиях функций. За что отвечает параметр `self` будет понятно из следующего пример. Создадим экземпляр класса `Dog`.

```
1 my_dog = Dog('Ксюша', 10) #Создание экземпляра класса
```

Что мы сделали? Мы сказали, что объект `my_dog` будет экземпляром класса `Dog`, при этом вместо параметра `self` передалось имя экземпляра. То есть при описании класса вместо имени конкретного экземпляра используется обозначение `self`, а при работе с непосредственным экземпляром указывается имя это экземпляра. Также мы передали аргументы `Ксюша` и `10`, а параметр `status` по умолчанию принял значение `Радость`. Рассмотрим следующий пример, демонстрирующий работу с конкретным экземпляром.

```
1 print(my_dog.name, my_dog.age, my_dog.status) #Вывод значений
    ↪ полей класса
2 my_dog.punish() #Метод punish()
3 print(my_dog.name, my_dog.age, my_dog.status) #Вывод значений
    ↪ полей класса
4 my_dog.pet() #Метод pet()
5 print(my_dog.name, my_dog.age, my_dog.status) #Вывод значений
    ↪ полей класса
```

Вывод:

```

1 Ксюша 10 Радость #Сразу после инициализации
2 Ксюша 10 Грусть #После применения метода punish()
3 Ксюша 10 Радость #После применения метода pet()

```

Обратим внимание, что поля класса можно изменять, если обратиться к ним напрямую, например

```

1 my_dog.age = 11
2 print(my_dog.age) #Вывод : 11

```

Однако, это несколько противоречит идее объектно ориентированного подхода. Более логично было бы создать соответствующий метод `happy_birthday`.

```

1 class Dog(): # Объявление класса
2     def __init__(self, name, age, status='Радость'): #Метод
        ↪ инициализации класса
3         self.name = name #Атрибут имя
4         self.age = age #Атрибут возраст
5         self.status = status #Атрибут
6     def pet(self): #Метод похвалить
7         self.status = 'Радость'
8     def punish (self): #Метод наказать
9         self.status = 'Грусть'
10    def happy_birthday(self): #Метод поздравления и изменения
        ↪ возраста
11        self.age = self.age + 1 #Увеличить значение возраста на
        ↪ 1
12        print('С Днем Рождения,', self.name, '. Тебе', self.age,
        ↪ 'лет!' )
13
14 my_dog = Dog('Ксюша', 10) #Создание экземпляра класса. Возраст
    ↪ равен 10
15 my_dog.happy_birthday() #Вызов метода

```

Вывод:

```

1 С Днем Рождения, Ксюша. Тебе 11 лет!

```

В качестве замечания отметим, что классы также часто выделяются в отдельные модули или библиотеки для того, чтобы их можно было повторно использовать большому количеству людей. Кроме того, отметим, что большинство объектов, с которыми вы будете далее иметь дело в **Python**, реализованы в виде классов и соответствующих методов.



## 6 Структуры данных в Python

Перейдем к особенностям организации некоторых структур данных и различным полезным функциям. Структуры данных крайне важны, так как в контексте обработки и анализа данных мы будем иметь дело непосредственно с ними.

### 6.1 Списки

Начнем знакомство с одной из структур данных – списками. До этого момента, при изучении оператора `for`, мы обходили их стороной, оперируя понятием множества, так как это понятие нам хорошо известно из математических наук. Однако в языках программирования есть специальные обозначения и термины.

Итак, начнем с примера. Предположим, что в группе 30 человек, и для каждого нам нужно хранить в программе его возраст. Можно создать 30 уникальных переменных, но это крайне неудобно. Для решения подобной задачи удобно использовать массив. Да, именно массив. Начнем с классического определения и внесем ясность в терминологию.

**Определение 6.1.1** *Массив – это пронумерованная последовательность значений одинакового типа, обозначаемая одним именем и физически хранящаяся последовательно в памяти компьютера.*

Возвращаясь к аналогии с контейнерами, массив – это большой контейнер, имеющий название и содержащий другие контейнеры.

**Определение 6.1.2** *Индекс – это порядковый номер элемента массива.*

При этом индексация элементов массива, как правило, начинается с нуля.

Например, `mas = [5, 8, 10, 32]` – массив целых чисел, элемент с нулевым индексом которого равен 5, с первым — 8 и так далее.

В дальнейшем, для краткости, мы будем позволять себе говорить менее строго, но более коротко: нулевой элемент, первый элемент и так далее.

Но при чем тут списки спросите вы? Дело в том, что в **Python** нет, как таковых, массивов (в классическом их определении), зато есть списки.

**Определение 6.1.3** *Список – это пронумерованная последовательность величин любого типа, обозначаемая одним именем и физически располагающаяся в произвольном порядке в памяти компьютера.*

Таким образом, отличие в следующем: в списках могут храниться любые типы данных, при этом физически расположенные в различных областях памяти компьютера. При этом доступ к элементам списка также осуществляется

с помощью индекса. Это значит, что массив тоже является списком, но не наоборот.

Разобравшись с терминологией, перейдем к практике и тому, как реализованы списки в **Python**. Список задается именем и перечислением своих элементов через запятую, заключенных в квадратные скобки, например это может быть список строковых данных:

```
1 pet = ['cat', 'dog', 'pig']
```

Или список из неких числовых значений, как целых, так и вещественных:

```
1 numbers = [1, 5.2, 6, 9.3]
```

В прочем, эти примеры ничем не отличают список от массива.

Посмотрите еще раз на отличие: список может содержать элементы абсолютно разных типов:

```
1 something = [1, 'cat', 6, 9.3, 'dog']
```

## Операции над списками

Любая работа со списками начинается с его создания. Как правило, список либо создается пустым, а затем в него добавляются элементы, либо элементы добавляются сразу:

```
1 names = [] #Пустой список
2 age = [13, 28, 30, 12, 45] #Список с заданными элементами
```

Конечно, при необходимости, добавить элементы можно в любой из созданных списков. Для **добавления** одного элемента в конец списка следует воспользоваться методом **append**. Метод указывается после имени списка через точку, а аргумент метода – это значение элемента:

```
1 age = []
2 age.append(4) #Добавить значение 4 в список age
```

Для множественного добавления элементов в список, существует метод **extend**. По сути, описанная функция расширяет существующий список элементами другого, добавленными в конец в указанном в списке порядке:

```
1 age = [13, 28]
2 age.extend([30, 12, 45]) #Добавить значения 30, 12, 45 в конец
   ↪ списка age
```

Обращение к конкретному элементу списка происходит по его индексу. Еще раз акцентируем внимание, что нумерация начинается с нуля. Приведем следующие примеры, где в комментариях написан результат вывода:

```

1 pet = ['cat', 'dog', 'pig']
2 print(pet[0]) #cat
3 numbers = [1, 5.2, 6, 9.3]
4 print(numbers[3]) #9.3
5 print(numbers[5]) #IndexError: list index out of range

```

Обратите внимание на последнюю строку кода. Очевидно, что элемента с индексом 5 нет в списке, и мы получим ошибку. Работая с большими объемами данных и циклами, такая ошибка не редкость, запомните ее :)

Вывести элементы списка можно по-разному, мы можем вывести весь список элементов, вывести конкретный элемент по индексу, или сделать так называемый срез.

```

1 age = [13, 28, 30, 12, 45] #Список с заданными элементами
2 print(age) #Вывод всех значений списка
3 print(age[0]) #Вывод элемента с индексом 0

```

Срез позволяет получить новый список из исходного, начиная с индекса *n* включительно до *m* не включительно.

```

1 age = [13, 28, 30, 12, 45] #Список с заданными элементами
2 print(age[1:3]) #Вывод: [28, 30]
3 print(age[:3]) #Вывод: [13, 28, 30]
4 print(age[2:]) #Вывод: [30, 12, 45]

```

Указывать обе границы не обязательно. Так, без указания начала *n*, срез начинается с нулевого элемента, а без указания конца *m*, срез будет до конца списка.

Срезы – полезная операция, так как позволяет получить часть данных из исходного набора. В приведенном коде показано, как можно создать новый список на основании среза.

```

1 age = [13, 28, 30, 12, 45] #Список с заданными элементами
2 age_part = age[1:3] #Новый список из элементов с индексами 1 и 2

```

**Удаление** элементов списка доступно несколькими способами. Для удаления элемента по его значению используют метод `remove`, при этом будет удален только первый такой элемент.

```

1 age = [13, 28, 13, 30, 45] #Список с заданными элементами
2 age.remove(13) #Удаление первого элемента списка, равного 13
3 print(age) #Вывод [28, 13, 30, 45]

```

Если необходимо удалить элемент по индексу, следует воспользоваться оператором `del`.

```

1 age = [13, 28, 13, 30, 45] #Список с заданными элементами
2 del age[3] #Удаление элемента списка с индексом 3
3 print(age) #Вывод [13, 28, 13, 45]

```

**Изменить** элемент списка можно по его индексу, присвоив новое значение.

```

1 age = [13, 28, 13, 30, 45] #Список с заданными элементами
2 age[3] = 99 #Изменение элемента с индексом 3
3 print(age) #Вывод [13, 28, 13, 99, 45]

```

И, наконец, рассмотрим еще **вставку** элемента в список. Вставка позволяет добавить новый элемент в указанную позицию списка, при этом уже имеющиеся элементы сдвигаются вправо, что приводит к изменению длины списка на единицу. За вставку отвечает метод `insert()` с двумя параметрами: позиция вставки и значение элемента.

```

1 age = [13, 28, 13, 30, 45] #Список с заданными элементами
2 age.insert(3, 99) #Вставка элемента 99 в позицию с индексом 3
3 print(age) #Вывод [13, 28, 13, 99, 30, 45]

```

При этом позиция вставки может быть равна длине списка (или даже больше). В таком случае элемент добавится в конец списка.

```

1 age = [13, 28, 13, 30, 45] #Список с заданными элементами
2 age.insert(10, 99) #Вставка элемента 99 в конец списка
3 print(age) #Вывод [13, 28, 13, 30, 45, 99]

```

## Функции `len()` и `range()`

Работа со списками редко обходится без использования функций `len()` и `range()`. Функция `len(s)` для списка `s` позволяет определить количество элементов в списке, при этом количество элементов списка чаще называют длиной списка.

**Определение 6.1.4** *Длина списка – это количество элементов списка.*

Функция `range(n, m, k)` возвращает упорядоченную последовательность целых чисел. Для функции можно указать до тех аргументов: `n` отвечает за левую границу, которая включается в последовательности чисел, `m` отвечает за правую границу, которая не включается в последовательности чисел и аргумент `k` – шаг. Все указанные аргументы не являются обязательными, рассмотрим следующие частные примеры.

Так `range(m)` позволяет получить последовательность целых чисел от нуля включительно до заданного числа `m` не включительно с шагом равным

единице. Таким образом, весьма органичным является совместное использование циклов, списков и рассматриваемых функций. Так, функция `range` от длины списка позволяет получить последовательность всех индексов, а цикл по ним позволяет обратиться к каждому элементу списка на новой итерации.

```
1 numbers = [1, 5, 6, 9]
2 for i in range(len(numbers)): #Равносильно for i in [0, 1, 2,
   ↪ 3]
3     print(numbers[i])
4 print('Конец цикла')
```

Стоит обратить внимание, что длина списка всегда больше на единицу, в сравнении с максимальным значением индекса, однако функция `range()` не включает последнее значение.

Функция `range` от двух параметров – `range(n, m)`, позволяет получить последовательность целых чисел в диапазоне от `n` включительно, до `m` не включительно с шагом равным единице. Рассмотрим следующий пример, допустим, мы хотим на каждой итерации цикла выводить на экран квадрат элемента списка. Тогда в качестве двух параметров функции `range` выступают 0 – индекс начала списка, и значение длины списка.

```
1 numbers = [1, 5, 6, 9]
2 for i in range(0, len(numbers)): #i изменяется от 0 до длины
   ↪ списка, т.е. до 4 не включительно
3     print(numbers[i] ** 2) #возведение в квадрат i-го элемента
```

Рассмотрим еще один пример, пусть имеется список имен, для каждого из которых необходимо сформировать приглашение на вечеринку, итак:

```
1 names = ['Дима', 'Антон', 'Алексей', 'Елена']
2 for i in range(0, len(names)):
3     print(names[i], ', приглашаю Вас на вечеринку!')
```

Такой подход является классическим для большинства языков программирования и использует счетчик для итерации элементов списка, однако **Python** позволяет достичь аналогичного результата в более простом виде, без использования индекса, а именно:

```
1 names = ['Дима', 'Антон', 'Алексей', 'Елена']
2 for name in names:
3     print(name, ', приглашаю Вас на вечеринку!')
```

Итак, цикл `for` вместе с операцией принадлежности `in` позволяет на каждой итерации переменной `name` принимать значения из списка `names`. Такой код является предпочтительным, так как его проще воспринимать.

И наконец, функция `range` может принимать три параметра – `range(n, m, k)`, где третий параметр `k` задает шаг изменения значений от `n` включительно, до `m` не включительно. Например, следующий код позволяет вывести четные числа из отрезка `[0, 50]`:

```
1 for i in range(0, 51, 2):
2     print(i)
```

Кроме того, параметр `k` может быть отрицательным. В таком случае левая граница должна быть больше правой – `range(10, 0, -2)`.

Наконец, функция `range` может быть использована для создания списков целых чисел с помощью функции `list()`, обратимся к примерам:

```
1 even_num = list(range(4, 11, 2)) #Список четных чисел [4, 6, 8,
   ↪ 10]
2 numbers = list(range(-2, 2)) #Список целых чисел [-2, -1, 0, 1]
```

## Метод `count()`

Метод `count(x)` – метод, который позволяет подсчитать, сколько раз конкретный элемент `x` встречается в списке. Например мы можем подсчитать вхождение числовых значений:

```
1 numbers = [1, 5, 6, 9, 1, 5, 1, 13]
2 print(numbers.count(1)) #Вывод 3
```

или строковых:

```
1 numbers = [1, 5, 6, '', 1, 5, '', 13]
2 print(numbers.count('')) #Вывод 2
```

## Функции `sum()`, `min()`, `max()`

Конечно, для таких базовых операций, как сумма всех чисел списка `s` и отыскание минимального, максимального элемента списка `s` существуют встроенные функции – `sum(s)`, `min(s)` и `max(s)` соответственно. Синтаксис крайне прост, в качестве параметра функций указывается имя списка, а на выходе получаем значение.

```
1 numbers = [1, 5, 6, 9, 1, 5, 1, 13]
2 print(sum(numbers)) #Вывод 41
3 print(min(numbers)) #Вывод 1
4 print(max(numbers)) #Вывод 13
```

При этом функции `min(s)` и `max(s)` будут работать и для строковых данных, по правилам отмеченных в операциях сравнения. Напомним, что цифровой символ условно меньше, чем любой алфавитный символ, а алфавитный символ в верхнем регистре условно меньше, чем алфавитный символ в нижнем регистре.

```
1 something = ['1', 'A', 'a', 'b', 'B', '13']
2 print(min(something)) #Вывод 1
3 print(max(something)) #Вывод b
```

## Сортировка списков

Очень часто, при работе со списками возникает необходимость их сортировки. Вспомним хотя бы вариационный ряд. Существует несколько способов сортировки, первый – метод `sort()`, который позволяет отсортировать список по возрастанию элементов. Обратимся к примеру кода:

```
1 numbers = [1, 5, 6, 9, 1, 5, 1, 13]
2 numbers.sort()
3 print(numbers) #Вывод [1, 1, 1, 5, 5, 6, 9, 13]
```

На первый взгляд он может показаться несколько странным, ведь мы не присвоили результат сортировки новому списку, а выводим исходный список `numbers`. Однако так и должно быть, метод `sort()` изменяет исходный список. Что не всегда удобно, так как исходная последовательность данных теряется безвозвратно.

Второй способ сортировки – функция `sorted(s)`. Она не меняет исходный список, так что результат сортировки можно просто вывести на экран, или записать в новый список, как это показано в коде:

```
1 numbers = [1, 5, 6, 9, 1, 5, 1, 13]
2 sorted_numbers = sorted(numbers) #Создание нового списка
  ↪ отсортированных элементов
3 print(sorted_numbers) #Вывод [1, 1, 1, 5, 5, 6, 9, 13]
4 print(numbers) #Вывод [1, 5, 6, 9, 1, 5, 1, 13]
```

Кроме того, оба способа сортировки поддерживают обратную сортировку (по убыванию), для этого необходимо добавить параметр `reverse=True` как это показано в примере кода:

```
1 numbers = [1, 5, 6, 9, 1, 5, 1, 13]
2 print(sorted(numbers, reverse=True)) #Вывод [13, 9, 6, 5, 5, 1,
  ↪ 1, 1]
3 numbers.sort(reverse=True)
4 print(numbers) #Вывод [13, 9, 6, 5, 5, 1, 1, 1]
```

## 6.2 Кортежи

**Определение 6.2.1** *Кортежем (tuple) называется список, который защищен от изменений.*

Элементы кортежа заключаются в круглые скобки и перечисляются через запятую. Рассмотрим пример.

```
1 a = (1, 2, 3, 4, 5, 6) #Создание кортежа
2 print(a[0:3]) #Вывод первых трех элементов
3 a[2] = 4 #Попытка изменить элемент кортежа
```

Вывод:

```
1 (1, 2, 3) #Вывод первых трех элементов кортежа
2 Traceback (most recent call last): #Объявление об ошибке
3   File "main.py", line 4, in <module>
4     a[2] = 4
5   TypeError: 'tuple' object does not support item assignment #
   ↳ Объекты типа кортеж не поддерживают изменение элементов.
```

Отдельно отметим создание кортежа из одного элемента на следующем примере.

```
1 a = ('s') # Не использована запятая
2 print(a) # В переменной содержится текст
3 a = ('s',) # Запятая использована
4 print(a) # Кортеж из одного элемента
5 a = 's', # Запятая есть, скобок нет
6 print(a) # Кортеж из одного элемента
7 a = 's',1 # Запятая есть, скобок нет
8 print(a) #Кортеж из двух элементов
```

Вывод:

```
1 s
2 ('s',)
3 ('s',)
4 ('s', 1)
```

## 6.3 Словари

В этом разделе рассмотрим словари и как с ними работать в Python. Словари позволяют хранить почти безграничное количество информации, что актуально в контексте обработки и анализа данных. Кроме того, словари позволяют более точно моделировать различные объекты реального мира.



**Определение 6.3.1** *Словарь – неупорядоченная структура данных, которая позволяет хранить пары «ключ–значение».*

Например, мы хотим сохранить информацию о мужчине по имени Степан, возраст которого 20 лет. Это можно сделать создав следующий словарь:

```
1 person = {'name': 'Степан', 'gender': 'муж.', 'age': 20}
```

Таким образом, словарь создается именем и перечислением элементов через запятую, заключенных в фигурные скобки, где каждый элемент записан в виде **ключ: значение**.

В указанном примере, ключами являются **name**, **gender**, **age**, которые принимают значения **Степан**, **муж.** и **20** соответственно. Ключи и значения отделяются друг от друга символом двоеточия. Кроме того, ключи могут быть как строковыми, так и числовыми.

```
1 cars = {2: 'BA3', '3': 'HONDA', 3: 'ИЖ'}
```

Тип данных играет важную роль при получении данных из словаря. Так, чтобы получить значение, связанное с ключом, следует указать имя словаря, а затем поместите ключ в квадратные скобки:

```
1 cars = {2: 'BA3', '3': 'HONDA', 3: 'ИЖ'}
2 print(cars[3]) #Вывод ИЖ
3 print(cars['3']) #Вывод HONDA
4 print(cars[0]) #Вывод KeyError: 0
```

Как видим из примеров кода, ключи отличаются по типу данных и соответствуют разным значениям, а в третьей строке кода, мы получаем ошибку, ключа – 0 нет в словаре.

Словари – это динамические структуры, и мы можем добавлять новые пары ключ–значение в словарь в любое время. Например, чтобы **добавить** новую пару ключ-значение, следует указать имя словаря и новый ключ в квадратных скобках вместе с новым значением:

```
1 person = {'name': 'Степан', 'gender': 'муж.', 'age': 20}
2 person['surname'] = 'Степанов' #Добавление пары 'surname':
  ↳ 'Степанов'
```

Окончательная версия словаря содержит четыре пары ключ–значение:

```
1 {'name': 'Степан', 'gender': 'муж.', 'age': 20, 'surname':
  ↳ 'Степанов'}
```

Обратим внимание, что порядок пар ключ–значение не обязательно совпадает с порядком их добавления.

Чтобы **изменить** значение в словаре, следует аналогично добавлению, указать имя словаря с ключом в квадратных скобках, а затем новое значение, при этом такой ключ уже должен быть в словаре. Например, если мы хотим изменить возраст Степана, следует указать имя словаря `person` с имеющимся ключом `age`:

```
1 person = {'name': 'Степан', 'gender': 'муж.', 'age': 20}
2 person['age'] = 21 #Изменение значения для ключа age
```

И, наконец, чтобы **удалить** пару ключ–значения из словаря, следует воспользоваться функцией `del()` и указать имя словаря с ключом в квадратных скобках. Например, если мы хотим удалить возраст Степана, следует указать имя словаря `person` с имеющимся ключом `age`:

```
1 person = {'name': 'Степан', 'gender': 'муж.', 'age': 20}
2 del(person['age']) #Удаление пары ключ-значение
```

## Циклы по словарям

Словари зачастую содержат большое количество данных. Очевидно возникает необходимость вывода этих данных или их преобразования. В случае со словарями, мы можем взаимодействовать как с ключами, так и значениями. Итак, чтобы выполнить цикл по словарю с помощью оператора `for` следует написать следующий код:

```
1 person = {'name': 'Степан', 'gender': 'муж.', 'age': 20}
2 for key, value in person.items():
3     print('Ключ: ', key)
4     print('Значение: ', value)
```

Как мы видим, во второй строке кода уже знакомый оператор `for`, однако вместо привычной одной переменной, указаны две – они будут содержать ключ и значение словаря на каждой итерации. Имена этих переменных могут быть любыми. Но чтобы цикл заработал, необходимо воспользоваться методом `items()`, который возвращает список пар ключ–значение, а оператор `for` сохраняет каждую из этих пар в указанные переменные.

Рассмотрим еще один пример, в нем словарь содержит в качестве ключей – имена людей, а в качестве значений – их любимые фрукты. Тогда цикл по всем парам ключ–значение позволяет нам составить фразы вида **человек любит фрукт**.

```

1 favorite_fruit = {'Дима': 'персик',
2                  'Алексей': 'манго',
3                  'Елена': 'яблоко',
4                  'Антон': 'ананас'}
5 for name, fruit in favorite_fruit.items():
6     print(name, 'любит есть', fruit)

```

Также следует обратить внимание, что элементы словаря при его создании указаны не в строку, а каждый с новой строки. Такое оформление используется исключительно для удобства восприятия кода и не несет смысловой нагрузки.

Существуют ситуации, когда нет необходимости работать со всеми значениями в словаре, а к примеру достаточно знать только ключи всех пар. Метод `keys()` предназначен именно для этого. Данный метод позволяет получить перечисление всех ключей, а цикл может быть написан привычным образом, с указанием одной переменной, которая принимает значения ключей.

```

1 favorite_fruit = {'Дима': 'персик',
2                  'Алексей': 'манго',
3                  'Елена': 'яблоко',
4                  'Антон': 'ананас'}
5 for name in favorite_fruit.keys():
6     print(name)

```

Можно привести еще один интересный пример, комбинированный с условным оператором `if`. В пятой строке приведенного кода выполняется проверка принадлежности. Так, если среди ключей нет имени Ольга, условие истинно и на экран выведется сообщение, что Ольга не приняла участие в опросе.

```

1 favorite_fruit = {'Дима': 'персик',
2                  'Алексей': 'манго',
3                  'Елена': 'яблоко',
4                  'Антон': 'ананас'}
5 if 'Ольга' not in favorite_fruit.keys():
6     print('Ольга, Вы не приняли участие в опросе!')

```

Может возникнуть и обратная ситуация, когда нас в первую очередь интересуют значения пар, содержащиеся в словаре. Метод `values()` предназначен именно для этого. Например, мы просто хотим получить список всех фруктов из указанного в коде словаря:

```

1 favorite_fruit = {'Дима': 'персик',
2                  'Алексей': 'манго',

```

```

3             'Елена': 'яблоко',
4             'Антон': 'ананас'}
5 for fruit in favorite_fruit.values():
6     print(fruit)

```

Этот метод извлекает все значения из словаря без проверки повторений. Чтобы вывести каждое значение словаря без повторений, мы можем использовать функцию `set()`. В приведенном примере, мы применяем функцию `set()` непосредственно для всех извлеченных значений словаря. Выдача будет содержать только фрукты яблоко и манго.

```

1 favorite_fruit = {'Дима': 'яблоко',
2                  'Алексей': 'манго',
3                  'Елена': 'яблоко',
4                  'Антон': 'яблоко'}
5 for fruit in set(favorite_fruit.values()):
6     print(fruit)

```

В заключении знакомства со словарями, отметим еще раз, что словарь всегда поддерживает четкую связь между каждым ключом и связанным с ним значением, однако порядок таких пар может быть любым и вывод данных не всегда предсказуем. Если же принципиален возврат пар в определенном порядке, ключи можно отсортировать в рамках цикла функцией `sorted()`:

```

1 person = {'name': 'Степан', 'gender': 'муж.', 'age': 20}
2 for key, value in sorted(person.items()):
3     print('Ключ: ', key)
4     print('Значение: ', value)

```

## 7 Библиотека NumPy

Библиотека **NumPy** получила свое название как сокращение от «Numerical Python» и используется для высокопроизводительных научных вычислений и анализа данных. Библиотека является фундаментальной, так на ее основе построены почти все библиотеки более высокого уровня. К основным возможностям **NumPy** можно отнести:

- поддержку многомерных массивов и операций над ними;
- наличие стандартных математических функций и их выполнение над массивами данных без необходимости записи циклов;

- поддержку операций линейной алгебры над массивами.

Подключение основных возможностей библиотеки NumPy осуществляется следующей строкой:

```
1 import numpy as np
```

Сокращение `np` считается общепринятым. В последующих примерах будем считать, что библиотека уже импортирована.

## 7.1 Многомерный массив-объект (ndarray)

Одной из ключевых особенностей NumPy является возможность использования так называемых многомерных массивов-объектов (N-dimensional array).

**Определение 7.1.1** *Многомерный массив-объект (ndarray) – это многомерный массив элементов (обычно чисел), одного типа.*

Иными словами, все элементы `ndarray` должны быть одного типа.

### Создание массивов

Для создания массива `ndarray`, например на основе списка, можно использовать метод `array(s)`. Кроме того, в качестве аргумента этого метода могут выступать любые объекты, которые возможно конвертировать в массив (кортеж, список кортежей, кортеж кортежей, кортеж списков и тому подобное).

```
1 list1 = [1, -3, 2, 7] #Список
2 arr1 = np.array(list1) #Преобразование списка в ndarray
3 print(arr1) #Вывод массива
4 print(type(arr1)) #Вывод типа объекта
```

В качестве вывода, получим:

```
1 [ 1 -3  2  7] #Ndarray (элементы без запятых)
2 <class 'numpy.ndarray'> #Тип объекта -- ndarray
```

**Замечание 7.1.1** *В рамках данного раздела, говоря о массивах, будем подразумевать именно ndarray.*

Для двумерного случая, имеем:

```

1 list2 = [[1, 2, 3, 4], [5, 6, 7, 8]] #Список из списков
   ↪ одинаковой длины
2 arr2 = np.array(list2) #Создание двумерного ndarray на основе
   ↪ list2
3 print(arr2) #Вывод массива
4 print(arr2.shape) #Вывод размерности массива

```

В результате получим:

```

1 [[1 2 3 4]
2  [5 6 7 8]]
3 (2, 4)

```

Метод `shape` возвращает кортеж, в котором содержится информация о размерности массива. В нашем случае: количество строк 2, столбцов – 4.

Для одномерного массива длиной, например, пять, метод `shape` вернет кортеж вида (5,). Массивы также можно создавать при помощи метода `zeros()` и `arange()`.

```

1 print(np.zeros(5)) #Массив размера (5, )из пяти нулей
2 print(np.arange(30, 100, 9)) #Массив размера (9, ) из чисел от
   ↪ 30 до 100 с шагом в 9

```

Вывод:

```

1 [0. 0. 0. 0. 0.]
2 [30 39 48 57 66 75 84 93]

```

## Математические операции над массивами

Массивы позволяют выполнять математические операции над целыми блоками данных, используя синтаксис, аналогичный эквивалентным операциям между числовыми элементами. Например код

```

1 arr2 = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
2 print(arr2 * 5)

```

выведет следующий результат:

```

1 [[ 5 10 15 20]
2  [25 30 35 40]]

```

Можно также совмещать операции над скалярными величинами и массивами.

```

1 arr2 = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
2 print(1 / arr2)

```

Вывод:

```

1 [[1.          0.5          0.33333333 0.25          ]
2  [0.2         0.16666667 0.14285714 0.125         ]]

```

## Одномерные массивы

Выборка элементов из массивов в **Numpy** – весьма объемная тема, так как существует достаточно большое количество способов выделения подмножества из исходного набора данных. Мы постараемся привести самые удобные приемы. Операции с одномерными массивами на первый взгляд аналогичны эквивалентным операциям при работе со списками.

```
1 arr = np.array([0, 1, 2, 3, 4, 5, 6, 7]) #Одномерный массив
2 print(arr[5]) #Вывод элемента с индексом 5
3 print(arr[:3]) #Вывод элементов с индексами 0,1, и 2
4 print(arr[5:]) #Вывод всех элементов, индекс которых больше либо
    ↪ равен пяти
5 print(arr[2:4]) #Вывод элементов с индексам 2 и 3
```

Вывод:

```
1 5
2 [0 1 2]
3 [5 6 7]
4 [2 3]
```

Можно заметить, что в последнем случае правая граница не включается. Как мы уже упомянули, на первый взгляд работа с одномерными массивами схожа с работой с одномерными списками. Но это лишь на первый взгляд. Рассмотрим следующий пример:

```
1 arr = np.array([0, 1, 2, 3, 4, 5, 6, 7])
2 arr_slice = arr[3:6] #Выделение подмножества
3 arr_slice[:] = 13 #Присвоение всем элементам arr_slice значения
    ↪ 13
4 print(arr_slice)
5 print(arr)
```

Вывод:

```
1 [13 13 13]
2 [ 0  1  2 13 13 13  6  7]
```

Что же мы видим? В массив **arr\_slice** мы поместили подмножество массива **arr**, поменяли все значения **arr\_slice** на 13, но эти изменения отразились и в оригинальном массиве **arr**. То есть данные в массив **arr\_slice** не скопировались, а просто некоторой части массива **arr** было дано отдельное имя. Такой результат весьма странный с точки зрения большинства языков программирования. Почему это реализовано именно так? Дело в том,

что NumPy разрабатывался непосредственно для обработки больших объемов данных, а операции копирования создавали бы серьезные проблемы с точки зрения производительности и памяти, поэтому использован именно такой подход.

**Замечание 7.1.2** *Для копирования данных используется метод `copy()`.*

Например:

```
1 arr_slice = arr[3:6].copy()
```

## Многомерные массивы

Рассмотрим работу с многомерными массивами на примере двумерного. Отличие от одномерного заключается в том, что теперь элемент массива — это, в свою очередь, тоже массив. Обратимся к примеру.

```
1 arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
2 print(arr2d[0]) # Вывод нулевого элемента (вывод массива)
3 print(arr2d[1][2]) # Вывод отдельного элемента
```

Вывод:

```
1 [1 2 3]
2 6
```

**Замечание 7.1.3** *Для вывода отдельных элементов можно использовать следующую нотацию `print(arr2d[1,2])`.*

Рассмотрим примеры выделения подмножеств из двумерного массива. Для этого через запятую указываются сначала номера строк, затем номера столбцов элементов, которые мы хотим получить.

```
1 arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
2 print(arr2d) #Выведем для удобства весь массив
3 print('Выделение подмножества')
4 print(arr2d[:2, 1:]) #Сначала выбираются строки с индексам
    ↪ меньше двух (нулевая и первая), затем столбцы с индексами
    ↪ больше, либо равными единице
```

Вывод:



```

1  [[1 2 3]
2   [4 5 6]
3   [7 8 9]]
4  Выделение подмножества
5  [[2 3]
6   [5 6]]

```

Обратим внимание, что при таком выделении подмножества количество измерений остается неизменным. Если не использовать двоеточие при указании номеров строк или столбцов, то количество измерений сокращается. Это хорошо видно из следующего примера.

```

1  arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
2  print('Двумерный массив')
3  print(arr2d[:,2:])
4  print('Одномерный массив')
5  print(arr2d[:,2])

```

Вывод:

```

1  Двумерный массив
2  [[3]
3   [6]
4   [9]]
5  Одномерный массив
6  [3 6 9]

```

Примеры выделения подмножеств и получаемые результаты представлены на рисунке 3. Обращаем внимание, что если параметр **shape** представлен только одним числом, то имеется в виду одномерный массив.

Для выделения в подмножество отдельных элементов исходного массива можно использовать индексирование на основе массивов целых чисел. Приведем пример.

```

1  arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
2  print('Исходный массив')
3  print(arr2d)
4  print('Выбор элементов')
5  print(arr2d[[0, 1, 2], [1, 2, 0]])

```

Вывод:

```

1  Исходный массив
2  [[1 2 3]

```

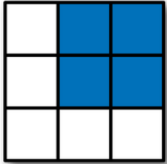
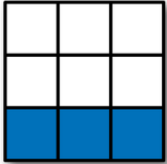
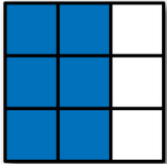
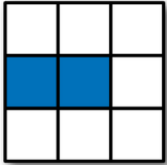
	Expression	Shape
	<code>arr[:2, 1:]</code>	<code>(2, 2)</code>
	<code>arr[2]</code> <code>arr[2, :]</code> <code>arr[2:, :]</code>	<code>(3,)</code> <code>(3,)</code> <code>(1, 3)</code>
	<code>arr[:, :2]</code>	<code>(3, 2)</code>
	<code>arr[1, :2]</code> <code>arr[1:2, :2]</code>	<code>(2,)</code> <code>(1, 2)</code>

Рис. 3: Примеры выделения подмножеств из двумерного массива

```

3  [4 5 6]
4  [7 8 9]]
5  Выбор элементов
6  [2 6 7]
```

Можно заметить, что в качестве результата был получен массив из элементов исходного массива с индексами  $(0, 1)$ ,  $(1, 2)$  и  $(2, 0)$ , то есть с теми индексами, которые были указаны при выборе элементов.

## 7.2 Операции над массивами

### Изменение формы массива

При работе с массивами нередко возникает необходимость каким-то образом изменить форму массива, например, транспонировать, преобразовать одномерный массив в многомерный, выполнить обратное преобразование и так далее. Рассмотрим следующий пример.

```

1  arr = np.arange(15) #Создание одномерного массива из 15
    ↪ элементов
2  print('Исходный массив')
3  print(arr)
```

```

4 print('Преобразование одномерного массива в двумерный размера
   ↪ (3, 5)')
5 arr = arr.reshape(3, 5)
6 print(arr)
7 print('Транспонирование')
8 print(np.transpose(arr))
9 print('Обратно в одномерный массив')
10 print(arr.reshape(15,))

```

Вывод:

```

1 Исходный массив
2 [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14]
3 Преобразование одномерного массива в двумерный размера (3, 5)
4 [[ 0  1  2  3  4]
5  [ 5  6  7  8  9]
6  [10 11 12 13 14]]
7 Транспонирование
8 [[ 0  5 10]
9  [ 1  6 11]
10 [ 2  7 12]
11 [ 3  8 13]
12 [ 4  9 14]]
13 Обратно в одномерный массив
14 [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14]

```

**Замечание 7.2.1** *Операция транспонирования может быть выполнена при помощи специального метода `T`.*

Операция транспонирования достаточно часто используется при выполнении матричных операций. В качестве примера найдем произведение  $XX^T$  при помощи функции `np.dot()`.

```

1 X = np.arange(6).reshape(2, 3) #X - матрица размерности [2 x 3]
2 print(np.dot(X, X.T)) #Матричное умножение

```

Вывод:

```

1 [[ 5 14]
2  [14 50]]

```

## Универсальные функции

В начале этого фрагмента мы уже сталкивались с математическими операциями, которые применялись ко всем элементам массива сразу, например операция `np.array([[1, 2], [3, 4]]) * 5` даст массив вида

```
1  [[ 5 10]
2  [15 20]]
```

**Определение 7.2.1** *Если функция NumPy применяется ко всем элементам массива, то она называется универсальной и обозначается `ufunc`.*

В качестве примеров универсальных функций можно рассматривать `np.sqrt()`, `np.exp()` и так далее. Приведем пример:

```
1  arr = np.arange(8).reshape(2, 4) #Создание массива размера (2,
    ↪ 4)
2  print('Исходный массив')
3  print(arr)
4  print('Извлечение корня')
5  print(np.sqrt(arr))
6  print('Нахождение синуса')
7  print(np.sin(arr))
```

Вывод:

```
1  Исходный массив
2  [[0 1 2 3]
3   [4 5 6 7]]
4  Извлечение корня
5  [[ 0.          1.          1.41421356  1.73205081]
6   [ 2.          2.23606798  2.44948974  2.64575131]]
7  Нахождение синуса
8  [[ 0.          0.84147098  0.90929743  0.14112001]
9   [-0.7568025  -0.95892427 -0.2794155   0.6569866  ]]
```

С полным перечнем универсальных функций можно ознакомиться в документации<sup>1</sup>.

---

<sup>1</sup><https://numpy.org/doc/>

## Выборочные характеристики

В NumPy вычисление различных выборочных характеристик реализовано в виде соответствующих методов, что избавляет от использования циклов. Рассмотрим следующий пример.

```
1 arr = np.arange(8) #Создание массива из 8 элементов
2 print('Исходный массив')
3 print(arr)
4 print('Среднее:', np.mean(arr))
5 print('Сумма:', np.sum(arr))
```

Вывод:

```
1 Исходный массив
2 [0 1 2 3 4 5 6 7]
3 Среднее: 3.5
4 Сумма: 28
```

Для двумерных массивов возможно нахождение выборочных характеристик по строкам или по столбцам.

```
1 arr = np.arange(8).reshape(2,4) #Создание массива
2 print(arr)
3 print(arr.mean(axis = 0)) #Средние значения по столбцами
4 print(arr.mean(axis = 1)) #Средние значения по строкам
```

Вывод:

```
1 [[0 1 2 3]
2  [4 5 6 7]]
3 [ 2.  3.  4.  5.]
4 [ 1.5  5.5]
```

Можно заметить, что при вычислении средних значений по столбцам или строкам, результаты представлены в виде массива.

## Сортировка

Для сортировки массивов в NumPy используется функция `np.sort()`. Приведем пример.

```

1 arr = np.array([[2, 0, 3], [4, 1, 7], [6, 8, 5]]) #Создание
  ↳ массива
2 arr2 = arr.reshape(9,) #Одномерный массив на основе arr
3 print('Отсортированный двумерный массив')
4 print(np.sort(arr))
5 print('Отсортированный одномерный массив')
6 print(np.sort(arr2))

```

Вывод:

```

1 Отсортированный двумерный массив
2 [[0 2 3]
3  [1 4 7]
4  [5 6 8]]
5 Отсортированный одномерный массив
6 [0 1 2 3 4 5 6 7 8]

```

Отметим, что мы осветили далеко не все возможности библиотеки **NumPy**. Дополнительную информацию можно изучить в текстовых материалах и дополнительных источниках.

## 8 Библиотека pandas

Библиотека **pandas** является основной при выполнении анализа данных в **Python**. Библиотека разработана поверх **NumPy**, что позволяет легко использовать их вместе. Подключение основных возможностей библиотеки **pandas** осуществляется двумя строчками кода:

```

1 from pandas import Series, DataFrame
2 import pandas as pd

```

Далее мы не будем их приводить в примерах. Чтобы начать работу с **pandas**, нам следует познакомиться с двумя структурами данных: сериями и фреймами данных, именно их мы и подключаем в первой строке приведенного кода.

### 8.1 Серии (Series)

**Определение 8.1.1** *Серия – это одномерный объект, содержащий массив данных (любого типа данных **NumPy**) и связанный с ним массив меток данных, называемый его индексом.*

Серия создается с помощью функции **Series()**. Рассмотрим самый простой пример, создание серии на основании списка **Python**:

```

1 numbers = Series([1, -3, 2, 7]) #Преобразование списка в серию
2 print(numbers) #Вывод серии

```

Вывод данных будет следующим, слева отображается индекс и значение справа.

```

1  0    1
2  1   -3
3  2    2
4  3    7

```

Поскольку при создании серии были указаны только данные, индекс создается автоматически и состоит из целых чисел, начиная с нуля. Аналогично словарям **Python**, мы можем получить доступ только к индексам или только к значениям серии, с помощью методов `index` и `values` соответственно.

```

1 numbers = Series([1, -3, 2, 7]) #Преобразование списка в серию
2 print(numbers.index) #Вывод индексов в формате индексов pandas
3 print(numbers.values) #Вывод значений серии

```

Вывод:

```

1 RangeIndex(start=0, stop=4, step=1)
2 [ 1 -3  2  7]

```

При создании серии можно указать индексы вручную, добавив параметр `index` в функцию `Series()`:

```

1 numbers = Series([1, -3, 2, 7], index=['d', 'b', 'a', 'c'])
   ↪ #Создание серии на основе списка значений и списка индексов
2 print(numbers) #Вывод серии

```

Вывод:

```

1  d    1
2  b   -3
3  a    2
4  c    7

```

Длина списка значений и индексов должна быть одинаковой, при этом на список индексов не накладывается критерий уникальности значений, как это было с ключами в словарях. Так в следующем примере, создается серия, содержащая повторяющийся индекс `a`.

```

1 numbers = Series([1, -3, 2, 7], index=['a', 'b', 'a', 'c'])
  ↳ #Серия, имеющая пару одинаковых индексов
2 print(numbers['a'].values) #Вывод значений серии, отвечающих
  ↳ индексу 'a'

```

Вывод:

```

1 [1 2]

```

То есть, при обращении к серии по такому индексу, будут выведены все значения, соответствующие этому индексу. В нашем примере – это числа 1 и 2.

Кроме того, можно указать массив индексов. В результате получим массив значений, соответствующих указанным индексам:

```

1 numbers = Series([1, -3, 2, 7], index=['a', 'b', 'a', 'c'])
  ↳ #Серия, имеющая пару одинаковых индексов
2 print(numbers[['a', 'c']].values) #Вывод значений серии,
  ↳ отвечающих индексам 'a' и 'c'

```

Вывод:

```

1 [1 2 7]

```

Еще один способ фильтрации данных, применение логических выражений. Так, операция сравнения серии со значением дает новую серию, где в качестве значений записан результат сравнения значения исходной серии со сравниваемым значением. В примере кода выполняется сравнение серии с числом 0:

```

1 numbers = Series([1, -3, 2, 7], index=['a', 'b', 'a', 'c'])
2 print(numbers > 0)

```

Вывод:

```

1 a      True
2 b      False
3 a      True
4 c      True

```

Полученную серию можно использовать в качестве индексов, по которым осуществляется вывод. Учитываться будут те индексы, которым соответствует значение «Истина»:

```

1 numbers = Series([1, -3, 2, 7], index=['a', 'b', 'a', 'c'])
2 print(numbers[numbers > 0])

```



Вывод:

```
1  a    1
2  a    2
3  c    7
```

Наконец, как и с `ndarray` массивами, можно выполнять математические операции, например прибавить ко всем элементам число или выполнить любые математические операции:

```
1 numbers = Series([1, -3, 2, 7], index=['a', 'b', 'a', 'c'])
2 print(numbers + 2)
```

Вывод:

```
1  a    3
2  b   -1
3  a    4
4  c    9
```

Еще одной распространенной практикой создания серий, является использование словарей **Python**, в качестве источника данных. Так как словари имеют структуру ключ–значение, она соотносится со структурой серий индекс–значение. Покажем это на следующем примере.

```
1 person = {'name': 'Степан', 'gender': 'муж.', 'age': 20}
   ↪ #Создание словаря
2 person_series = Series(person) #Преобразование словаря в серию
3 print(person_series) #Вывод серии
```

Вывод:

```
1 name      Степан
2 gender     муж.
3 age        20
```

## 8.2 Фреймы данных (DataFrame)

**Определение 8.2.1** *Фрейм данных – это табличная структура данных (каждый объект имеет как индекс строки, так и индекс столбца), содержащая упорядоченный набор столбцов, каждый из которых может иметь различный тип данных.*

Фрейм данных, как правило, создается для двумерных данных, либо на основе словарей из списков одинаковой длины, серий, либо на основе CSV набора данных.

Например, создадим фрейм данных на основе словаря, состоящего из списков одинаковой длины, то есть каждому ключу соответствует не одно значение, а список значений. Для создания фрейма данных используется функция `DataFrame()`.

```
1 cars = {'name': ['ford torino', 'peugeot 504', 'fiat 124b'],
    ↪      'hp': [140, 87, 76]} #Создание словаря из списков
2 cars_frame = DataFrame(cars) #Создание фрейма данных на основе
    ↪      словаря cars
3 print(cars_frame) #Вывод фрейма данных
```

Вывод:

```
1           name    hp
2  0  ford torino  140
3  1  peugeot 504   87
4  2   fiat 124b   76
```

Можно заметить, что вывод фрейма данных осуществляется в виде таблицы, где индексы столбцов соответствуют ключам словаря, а каждая строка имеет порядковый номер, соответствующий номерам значений в списках. То есть строка ноль состоит из значений `ford torino` и `140`, которые являются нулевыми элементами в соответствующих списках, строка один – из значений `peugeot 504` и `87`, которые имеют индекс один в соответствующих списках и так далее.

На рисунке 4 представлен пример вывода этого же фрейма данных в Jupyter блокноте. Можно заметить, что таблица имеет некоторое оформление, что возможно способствует лучшему восприятию информации.

Столбец фрейма данных может быть извлечен как серия, либо с помощью обращения по имени столбца, по аналогии со словарями, либо имя может быть использовано как метод:

```
1 cars = {'name': ['ford torino', 'peugeot 504', 'fiat 124b'],
2         'hp': [140, 87, 76]}
3 cars_frame = DataFrame(cars)
4 print(cars_frame['name'])
5 print(cars_frame.name)
```

Вывод:

```
In [9]: from pandas import Series, DataFrame
import pandas as pd

In [10]: cars = {'name': ['ford torino', 'peugeot 504', 'fiat 124b'],
               'hp': [140, 87, 76]}
cars_frame = DataFrame(cars)

In [11]: cars_frame
Out[11]:
```

	name	hp
0	ford torino	140
1	peugeot 504	87
2	fiat 124b	76

Рис. 4: Пример вывода фрейма данных в Jupyter блокноте.

```
1 0    ford torino
2 1    peugeot 504
3 2      fiat 124b
```

При создании серий, за счет табличного формата данных возможно явно указать очередность столбцов, и, конечно, задать индексы строк вручную, по аналогии с сериями:

```
1 cars = {'name': ['ford torino', 'peugeot 504', 'fiat 124b'],
2         'hp': [140, 87, 76],
3         'weight': [3449, 2672, 2065]}
4 cars_frame = DataFrame(cars, columns=['name', 'weight', 'hp'],
5                          index=['one', 'two', 'three'])
```

Вывод данных представлен на рисунке 5.

С другой стороны, указывать индексы вручную, когда мы работаем с реальными данными, очень долго, да и скорее бессмысленно. Так что зачастую, в качестве индекса используется столбец из уже имеющихся данных. Например, мы можем использовать названия машин в качестве индекса:

```
1 cars = {'name': ['ford torino', 'peugeot 504', 'fiat 124b'],
2         'hp': [140, 87, 76],
3         'weight': [3449, 2672, 2065]}
4 cars_frame = DataFrame(cars, columns=['weight', 'hp'],
5                          index=cars['name'])
```

В результате получаем удобный для обработки фрейм данных (рисунок 6). Каждый столбец имеет понятное название, каждая строка соответствует конкретному автомобилю.

```
In [16]: cars = {'name': ['ford torino', 'peugeot 504', 'fiat 124b'],
               'hp': [140, 87, 76],
               'weight': [3449, 2672, 2065]}
cars_frame = DataFrame(cars, columns=['name', 'weight', 'hp'],
                      index=['one', 'two', 'three'])
```

```
In [17]: cars_frame
```

```
Out[17]:
```

	name	weight	hp
one	ford torino	3449	140
two	peugeot 504	2672	87
three	fiat 124b	2065	76

Рис. 5: Пример вывода фрейма данных (задан порядок столбцов и индексы строк).

## Операции с фреймами данных

Метод `loc()` позволяет получить доступ к строке или конкретному значению фрейма данных по индексам. Рассмотрим варианты его применения.

Для получения данных строки, метод `loc()` применяется к фрейму данных с указанием индекса строки.

```
1 cars_frame.loc['ford torino']
```

Вывод:

```
1 weight    3449
2 hp        140
```

Метод возвращает серию, но что если мы хотим продолжить работу с этим объектом, как с фреймом данных? Для этого следует воспользоваться методом `to_frame()`, который преобразует серию в фрейм данных:

```
1 cars_frame.loc['ford torino'].to_frame()
```

Однако, значения будут представлены в виде столбца. По аналогии с операциями над матрицами, фрейм данных можно транспонировать с помощью метода `T`:

```
1 cars_frame.loc['ford torino'].to_frame().T
```

Для вывода конкретного значения фрейма (на пересечении строки и столбца), после метода `loc()` следует указать индекс строки и столбца в квадратных скобках:

```
In [22]: cars = {'name': ['ford torino', 'peugeot 504', 'fiat 124b'],
               'hp': [140, 87, 76],
               'weight': [3449, 2672, 2065]}
cars_frame = DataFrame(cars, columns=['weight', 'hp'],
                      index=cars['name'])
```

```
In [23]: cars_frame
```

Out[23]:

	weight	hp
ford torino	3449	140
peugeot 504	2672	87
fiat 124b	2065	76

Рис. 6: Пример вывода фрейма данных (заданы столбцы и индексы строк из исходных данных).

```
cars_frame.loc['ford torino'].to_frame()
```

ford torino	
weight	3449
hp	140

```
cars_frame.loc['ford torino'].to_frame().T
```

	weight	hp
ford torino	3449	140

Рис. 7: Преобразование в фрейм данных и его транспонирование.

```
1 cars_frame.loc['peugeot 504']['hp'] #Вывод значения 87
```

Кроме того, такая конструкция может быть использована для **изменения** данных. Для этого необходимо присвоить новое значение с помощью операции присваивания:

```
1 cars_frame.loc['peugeot 504']['hp'] = 140
```

Распространенной операцией над фреймами данных является удаление строк и столбцов, так как зачастую мы имеем дело с некоторой выгрузкой всех данных, а уже затем, после анализа, оставляем лишь их часть. За **удаление** строк или столбцов отвечает метод `drop()`, который возвращает новый фрейм данных.

Первый вариант использования метода – удаление строк. В качестве параметра метода `drop()` следует указать массив индексов. Например следующий код удаляет строку для автомобиля фиат и создает новый фрейм данных:

```
1 new_cars_frame = cars_frame.drop(['fiat 124b'])
```

Вывод нового фрейма данных:

```

1           weight    hp
2  ford torino      3449  140
3  peugeot 504      2672   87

```

Второй вариант – удаление столбцов, для этого следует добавить параметр `axis=1` методу `drop()`:

```
1 new_cars_frame = cars_frame.drop(['hp'], axis=1)
```

В результате выполнения указанной команды, получим новый фрейм данных, без столбца `hp`:

```

1           weight
2  ford torino      3449
3  peugeot 504      2672
4  fiat 124b        2065

```

## Сортировка данных

Для сортировки серий и фреймов существует метод `sort_values()`. Метод не изменяет исходные данные, а возвращает объект того же типа, по этой причине результат следует присвоить переменной:

```

1 numbers = Series([1, -3, 2, 7])
2 sorted_numbers = numbers.sort_values()

```

Вывод переменной `sorted_numbers`:

```

1  1   -3
2  0    1
3  2    2
4  3    7

```

По умолчанию сортировка выполняется по возрастанию, для сортировки по убыванию необходимо указать параметр `ascending=False`:

```

1 numbers = Series([1, -3, 2, 7])
2 sorted_numbers = numbers.sort_values(ascending=False)

```

При сортировке фреймов, все аналогично, только следует указать индексы колонок в порядке их сортировки, например для фрейма данных о автомобилях мы можем указать сортировку по возрастанию показателя лошадиных сил:

```

1 cars = {'name': ['ford torino', 'peugeot 504', 'fiat 124b'],
2         'hp': [140, 76, 76],
3         'weight': [3449, 2672, 2065]}
4 cars_frame = DataFrame(cars, columns=['weight', 'hp'],
5                        index=cars['name'])
6 cars_frame.sort_values('hp')

```

Либо по возрастанию лошадиных сил и убыванию массы автомобиля:

```

1 cars_frame.sort_values(['hp', 'weight'], ascending=[True,
↪ False])

```

Для этого названия колонок указываются в виде массива, как и параметр сортировки.

## Математические и статистические функции

Применение фреймов для хранения данных позволяет применять встроенные математические и статистические методы. Приведем лишь некоторые примеры функций, а далее приведем таблицу всех возможных.

Рассмотрим классическую задачу линейной нормировки данных на все тех же данных о автомобилях:

```

1 cars = {'name': ['ford torino', 'peugeot 504', 'fiat 124b'],
2         'hp': [140, 87, 76],
3         'weight': [3449, 2672, 2065]}
4 cars_frame = DataFrame(cars, columns=['weight', 'hp'],
5                        index=cars['name'])

```

Для это нам необходимо найти минимальный и максимальный элемент каждого столбца данных и воспользоваться преобразованием:

$$x = \frac{x - x_{min}}{x_{max} - x_{min}}.$$

Чтобы найти максимальный и минимальный элемент каждого столбца воспользуемся методами `.max()` и `.min()` соответственно. Например, метод `.min()` возвращает серию минимальных значений каждой колонки, с указанием индексов:

```

1 cars_frame.min()

1 weight    2065
2 hp        76

```

Такой формат крайне удобен, так как преобразование всех данных фрейма можно выполнить в одно действие, а именно:

```
1 cars_frame_norm = (cars_frame - cars_frame.min()) /  
    ↪ (cars_frame.max() - cars_frame.min())
```

Пример полного кода из Jupyter блокнота и вывод данных представлены на рисунке 8. Таким образом, нет необходимости выполнять нормировку или любое другое преобразование, над каждым столбцом по отдельности, библиотека **pandas** обеспечивает возможность оперировать всем объектом сразу, что сокращает код и повышает его читаемость.

```
In [6]: from pandas import Series, DataFrame  
import pandas as pd  
  
In [7]: cars = {'name': ['ford torino', 'peugeot 504', 'fiat 124b'],  
               'hp': [140, 87, 76],  
               'weight': [3449, 2672, 2065]}  
cars_frame = DataFrame(cars, columns=['weight', 'hp'],  
                       index=cars['name'])  
  
In [8]: cars_frame_norm = (cars_frame - cars_frame.min()) / (cars_frame.max() - cars_frame.min())  
  
In [9]: cars_frame_norm  
Out[9]:
```

	weight	hp
ford torino	1.000000	1.000000
peugeot 504	0.438584	0.171875
fiat 124b	0.000000	0.000000

Рис. 8: Пример линейной нормировки фрейма данных.

Еще один полезный метод – метод **describe()**, который позволяет узнать некоторые сводные статистики: разброс значений, среднее, медиану и проч.

```
1 cars_frame.describe()
```

Другие методы, представленные в таблице 1, применяются аналогичным образом и могут быть полезны при обработке и анализе данных.

## Импорт данных

Когда речь заходит о наборах данных, не важно, выгружаются ли они из баз данных или из открытых источников, универсальным является **CSV** формат. Библиотека **pandas** содержит ряд полезных функций для работы с данными в таком формате.



```
In [10]: cars_frame.describe()
```

```
Out[10]:
```

	weight	hp
count	3.000000	3.000000
mean	2728.666667	101.000000
std	693.737943	34.219877
min	2065.000000	76.000000
25%	2368.500000	81.500000
50%	2672.000000	87.000000
75%	3060.500000	113.500000
max	3449.000000	140.000000

Рис. 9: Пример вывода сводных статистик.

Базовая функция для чтения данных – `read_csv()` имеет огромное количество параметров, мы рассмотрим наиболее часто применяемые. Функция `read_csv()` на самом деле является методом, реализуемым над классом `pandas`, именно по этой причине принято импортировать библиотеку с указанием короткого обозначения `pd`:

```
1 import pandas as pd
```

Итак, первое, что, конечно, следует указать при импорте данных, это имя **CSV** файла (если он находится в той же директории, что и файл Jupyter блокнота) или полный путь к файлу. Данные файла будут загружены во фрейм данных:

```
1 all_cars = pd.read_csv('auto-mpg.csv')
```

При выводе данных, отображаются лишь первые и последние строки фрейма, названия индексов строк и столбцов, а также его размерность (рисунок 10).

**Замечание 8.2.1** Для быстрого просмотра фрейма используется метод `head()`, который по умолчанию выводит названия индексов и пять строк данных:

```
1 all_cars.head()
```

В таблице 2 указаны дополнительные параметры импорта, которые не являются обязательными, но порой сильно упрощают работу с данными. Так, несмотря на то, что название формата **CSV** расшифровывается как Comma-Separated Values, то есть значения разделенные запятой – это не всегда так.

Таблица 1: Методы описательной и сводных статистик.

Метод	Описание
<code>describe()</code>	Вычисление статистик для серии или фрейма
<code>min(), max()</code>	Определение минимального и максимального значения
<code>argmin(), argmax()</code>	Определение числового идентификатора минимального и максимального значения
<code>idxmin(), idxmax()</code>	Определение идентификатора минимального и максимального значения
<code>quantile()</code>	Вычисление выборочной квантили
<code>sum()</code>	Вычисление суммы
<code>mean()</code>	Вычисление среднего
<code>median()</code>	Вычисление медианы
<code>var()</code>	Вычисление дисперсии (параметр <code>ddof = 0</code> – смещенная $S^2$ , <code>ddof = 1</code> – несмещенная $S_0^2$ )
<code>std()</code>	Вычисление среднеквадратического отклонения (параметр <code>ddof = 0</code> – смещенное $\sigma^*$ , <code>ddof = 1</code> – несмещенное $\sigma_0^*$ )

Аналогично и с разделителем десятичных дробей, в исходных данных он может быть как запятой, так и точкой. Ну и конечно, не всегда нужен весь объем данных, мы можем ограничить число строк параметром `nrows` и выбрать только интересующие нас столбцы с помощью `usecols`. Обратите внимание, что колонка индексов также должна входить в перечисление колонок.

```

1 all_cars = pd.read_csv('auto-mpg.csv',
2                       delimiter=',',
3                       index_col='car name',
4                       nrows=5,
5                       decimal='.',
6                       usecols=['car name', 'horsepower',
                               ↪ 'cylinders', 'weight'])

```

## 9 Визуализация данных

Визуализация данных, пожалуй, одна из наиболее важных задач в анализе данных, так как именно человеческий опыт позволяет зачастую просто посмотрев на визуализацию данных, определить методы их обработки. Тема визуализации данных заслуживает отдельного курса, мы же разберем основы синтаксиса библиотеки `matplotlib`, чтобы научиться отображать как

```
In [16]: all_cars = pd.read_csv('auto-mpg.csv')
```

```
In [17]: all_cars
```

```
Out[17]:
```

	mpg	cylinders	displacement	horsepower	weight	acceleration	model year	origin	car name
0	18.0	8	307.0	130	3504	12.0	70	1	chevrolet chevelle malibu
1	15.0	8	350.0	165	3693	11.5	70	1	buick skylark 320
2	18.0	8	318.0	150	3436	11.0	70	1	plymouth satellite
3	16.0	8	304.0	150	3433	12.0	70	1	amc rebel sst
4	17.0	8	302.0	140	3449	10.5	70	1	ford torino
5	15.0	8	429.0	198	4341	10.0	70	1	ford galaxie 500
...	...	...	...	...	...	...	...	...	...
392	27.0	4	151.0	90	2950	17.3	82	1	chevrolet camaro
393	27.0	4	140.0	86	2790	15.6	82	1	ford mustang gl
394	44.0	4	97.0	52	2130	24.6	82	2	vw pickup
395	32.0	4	135.0	84	2295	11.6	82	1	dodge rampage
396	28.0	4	120.0	79	2625	18.6	82	1	ford ranger
397	31.0	4	119.0	82	2720	19.4	82	1	chevy s-10

398 rows x 9 columns

Рис. 10: Пример импорта данных.

сами данные, так и строить дополнительные объекты, например графики функций.

## 9.1 Основы matplotlib. Метод plot()

Начнем, как обычно, с подключения самой библиотеки и ее общепринятого сокращения `plt`. Но кроме непосредственного подключения библиотеки, воспользуемся «магической» командой `%matplotlib inline`, которая позволяет отображать графики сразу под ячейками в самом Jupyter блокноте. Подключение библиотеки выполняется один раз:

```
1 %matplotlib inline
2 import matplotlib.pyplot as plt
```

### Построение графиков

Начнем с базового метода `plot()`. Данный метод позволяет отображать точки на плоскости и соединить их прямыми. Координаты точек задаются двумя массивами, первый отвечает за координаты по горизонтальной оси ( $X$ ), второй – за координаты по вертикальной ( $Y$ ). Например, следующий код создаст четыре точки с координатами  $(2, 5)$ ,  $(1, 3)$ ,  $(3, 3)$ ,  $(-2, 4)$  и соединить их прямыми линиями, именно в таком порядке, что видно из рисунка 11.

Таблица 2: Параметры функции импорта данных `read_csv()`.

Параметр	Описание
<code>delimiter</code>	Тип разделителя в CSV файле
<code>encoding</code>	Кодировка CSV файла
<code>nrows</code>	Число строк, которые следует считать
<code>index_col</code>	Имя столбца, являющегося индексом для строк
<code>usecols</code>	Массив имен столбцов, которые следует считать
<code>decimal</code>	Тип десятичного разделителя (точка или запятая) в CSV файле

```

1 plt.plot([2, 1, 3, -2], [5, 3, 3, 4])
2 plt.show()

```

```

In [10]: %matplotlib inline
import matplotlib.pyplot as plt

```

```

In [11]: plt.plot([2, 1, 3, -2], [5, 3, 3, 4])
plt.show()

```

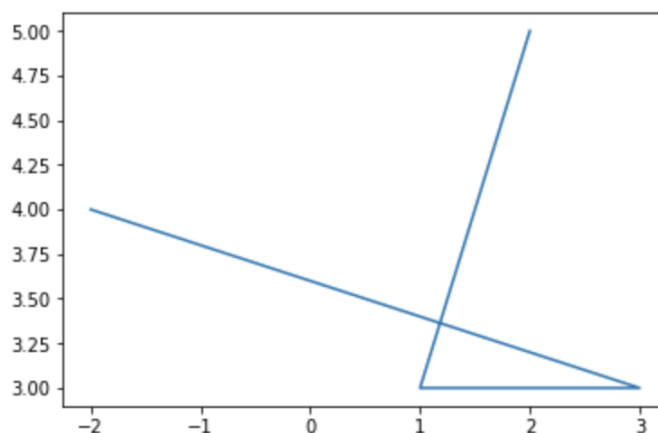


Рис. 11: Пример отображения графика (метод `plot()`).

### Замечание 9.1.1

Зачастую, графики строятся на основании данных из фреймов **pandas**, в таком случае, в качестве параметров указывают названия индексов колонок и параметр **data**, указывающий на фрейм данных (или на самом деле любой объект, например словарь или **ndarray**). В качестве данных возьмем уже знакомые данные по автомобилям, по оси *X* отметим отсортированную массу, а по *Y* мощность:

```

1 all_cars = pd.read_csv('auto-mpg.csv',
2                         delimiter=',',
3                         index_col='car name',
4                         nrows=20,
5                         decimal='.',
6                         usecols=['car name', 'horsepower',
7                               ↪ 'cylinders', 'weight'])
7 all_cars = all_cars.sort_values('weight')
8 plt.plot('weight', 'horsepower', data=all_cars)
9 plt.show()

```

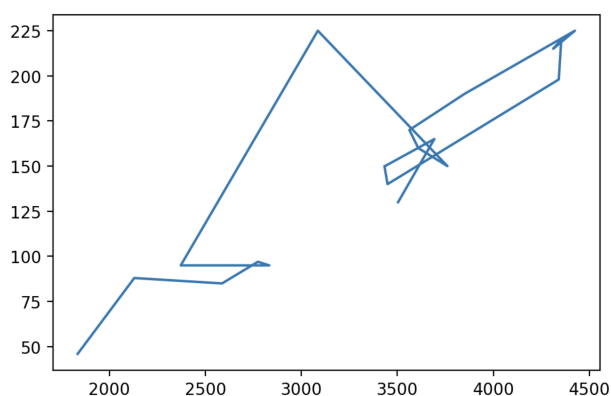
Получим конечно странный результат (рисунок 12а), так как порядок точек произвольный. Так, для визуализации, данные тоже необходимо подготовить, например, отсортировав по возрастанию массы автомобилей:

```

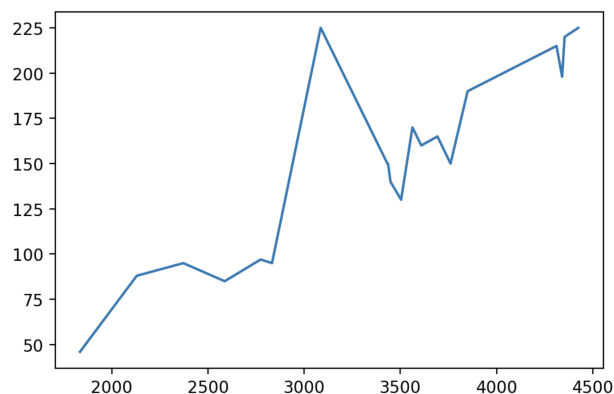
1 all_cars = all_cars.sort_values('weight')

```

В таком случае получим куда более ясный результат, представленный на рисунке 12б.



(a) Фрейм `all_cars` без сортировки.



(b) Фрейм `all_cars`, сортировка по возрастанию массы.

Рис. 12: Зависимость мощности от массы автомобиля.

## Оформление графиков

Важную роль играет оформление графиков. Метод `plot()` может быть дополнен следующими параметрами:

1. `linestyle` или `ls` – стиль линии. Может принимать значения:
2. `linewidth` или `lw` – толщина линии.
3. `marker` – тип маркера. Может принимать значения:

Значение	Тип линии
'_'	Сплошная
'--'	Прерывистая
'-.'	Прерывистая с точкой
':'	Пунктирная

Значение	Тип линии
'.'	Точка
'o'	Круг
'v'	Треугольник
's'	Квадрат
...	И многие другие <sup>2</sup>

4. `color` или `c` – цвет линии. Может принимать строковые значения, названия цвета, например `red`, `black` или шестнадцатеричный код цвета, например `#ff50cd`.

5. `label` – название графика для отображения в легенде.

Указанные параметры объединены в следующем примере кода:

```
1 plt.plot('weight', 'horsepower', data=all_cars, ls='-.', lw='1',
   ↪ c='red', marker='*')
```

График строится на основании данных о массе и мощности автомобилей. Маркеры точек отображаются в виде звездочек, которые соединены прерывистыми с точкой линиями, толщина 1 единица, цвет красный. Результат представлен на рисунке 13а.

## Оформление полотна

Кроме параметров графика, мы можем настроить само полотно. Настройка различных параметров осуществляется с помощью методов:

1. `title()` – название полотна.

```
1 plt.title('Зависимость мощности автомобилей от массы')
```

2. `xticks()` и `yticks()` – значения (названия), отображаемые по оси  $X$  и  $Y$ . Основные параметры метода: значения, отображаемые по оси; текстовые подписи, замещающие указанные числовые значения; настройки отображения текста<sup>3</sup>.

<sup>3</sup>[https://matplotlib.org/api/text\\_api.html#matplotlib.text.Text](https://matplotlib.org/api/text_api.html#matplotlib.text.Text)

```

1 plt.xticks([1500, 3000, 45000], ['Легковесный',
  ↪ 'Средневесный', 'Тяжеловесный'])

```

3. `xlim()` и `ylim()` – диапазон, отображаемый по оси  $X$  и  $Y$ .

```

1 plt.ylim(125, 200)

```

4. `xlabel()` и `ylabel()` – подпись по оси  $X$  и  $Y$ .

5. `legend()` – отображение описания графиков. Параметр `loc` отвечает за расположение на полотне, значения по вертикали – `upper`, `center`, `lower`, по горизонтали `left`, `center`, `right`.

6. `grid()` – отображение сетки полотна. Метод может быть дополнен параметрами, аналогичными для `plot()`, для настройки отображаемых линий сетки (цвет, толщина и т.д.).

Указанные методы и параметры объединены в следующем примере кода:

```

1 plt.plot('weight', 'horsepower', data=all_cars, ls='-.', lw='1',
  ↪ c='red', marker='*', label='Ломаная со звездочками')
2 plt.title('Зависимость мощности автомобилей от массы')
  ↪ #Заголовок полотна
3 plt.xticks(np.arange(1600, 4700, step=300)) #Отображаемые
  ↪ значения по оси $X$
4 plt.xlabel('Масса (кг)') #Подпись по оси X
5 plt.ylabel('Мощность (л/с)') #Подпись по оси Y
6 plt.legend(loc='lower right') #Отображение легенды в нижнем
  ↪ правом углу
7 plt.grid(c='#BFEFEF', ls='-', lw=0.5) #Отображение сетки с
  ↪ заданными параметрами типа линий и их цвета
8 plt.show()

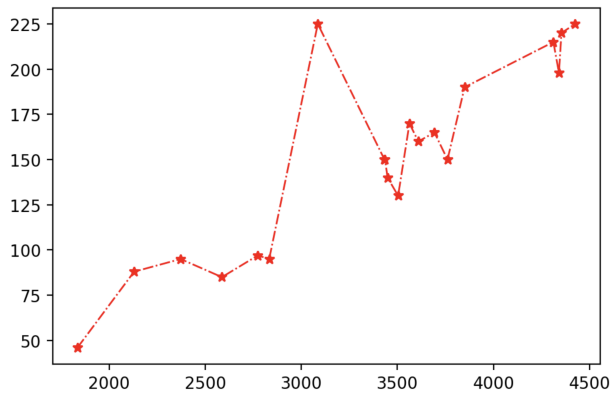
```

В первой строке создается аналогичная прошлому примеру ломаная линия, а далее задаются параметры для полотна. Результат представлен на рисунке 13b.

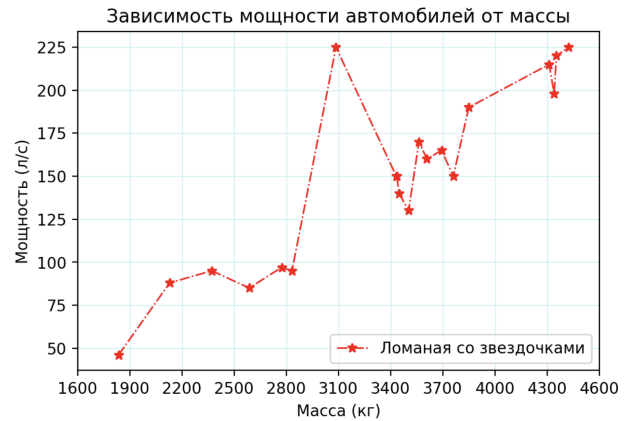
## 9.2 Построение графиков функций

Зачастую нам потребуется изображать на плоскости (или в пространстве) различные множества: как простые, например, заданные явно в виде функциональной зависимости  $y = f(x)$ , так и более сложные.

Начнем с простых функций, заданных явным образом. На плоскости рассмотрим зависимости вида  $y = f(x)$ , заданные на отрезке  $[a, b]$ . Для начала, зададим массив значений аргумента  $x$ , это удобно сделать с помощью



(a) Изменение отображения графика.



(b) Изменение отображения полотна.

Рис. 13: Возможности настроек отображения объектов.

библиотеки **NumPy** и метода `linspace()`, который позволяет создать массив вещественных чисел от некоторой нижней границы  $a$  до верхней  $b$  с заданным количеством элементов  $n$ . Так как график строится соединением соседних точек отрезком, то понятно, что чем больше точек, тем более точный и более гладкий получается график функции.

```
1 x = np.linspace(-5, 5, 100) #Массив из 100 значений на отрезке
   ↪ [-5, 5]
```

Тогда значения функции  $y$  могут быть получены как преобразование массива значений  $x$  посредством функции  $f(x)$ :

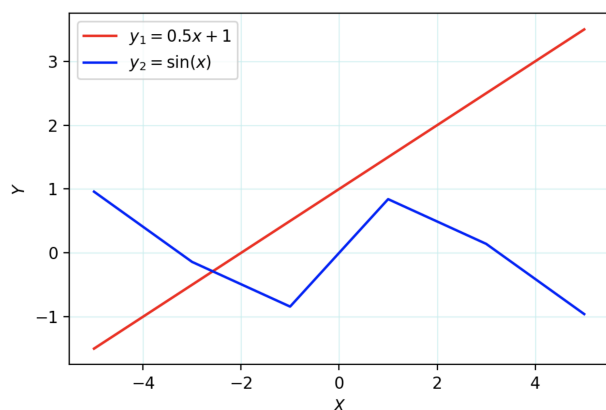
```
1 y_1 = 0.5 * x + 1 #Вычисление функции 0.5x+1 для каждого
   ↪ элемента x
2 y_2 = np.sin(x) #Вычисление функции sin() для каждого элемента x
```

Следующий код позволяет построить графики двух функций на одном полотне:

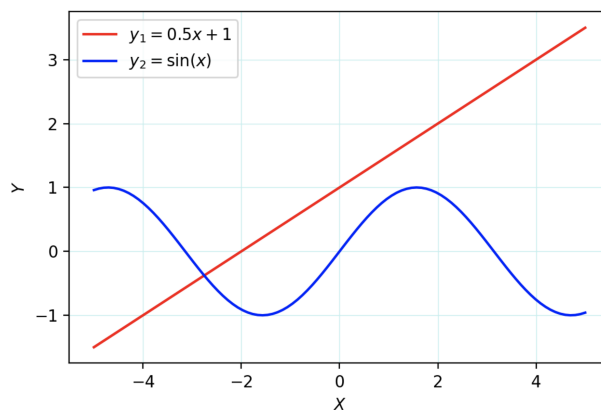
```
1 x = np.linspace(-5, 5, 100)
2 y_1 = 0.5 * x + 1
3 y_2 = np.sin(x)
4 plt.plot(x, y_1, c='red', label='$y_1=0.5x+1$')
5 plt.plot(x, y_2, c='blue', label='$y_2=\sin(x)$')
6 plt.xlabel('$X$')
7 plt.ylabel('$Y$')
8 plt.legend(loc='upper left')
9 plt.grid(c='#BFEFEF', ls='-', lw=0.5)
10 plt.show()
```



На рисунке 14 представлены результаты выполнения кода при разном количестве точек, на рисунке 14а для 6 значений из отрезка  $[-5, 5]$ , на рисунке 14б для 100 значений.



(а) 6 значений.



(б) 100 значений.

Рис. 14: Графики функций при разном количестве точек.

Часто, однако, нам требуется построить не график явной зависимости, а график порции множества, задаваемого уравнением  $F(x, y) = 0$ , при  $x \in [a, b]$ ,  $y \in [c, d]$ . Для построения мы будем использовать метод `contour()`, который предназначен для построения линий уровня, но может быть использован и для рассматриваемой задачи построения графика неявной функции. Термин линия уровня обычно приводится при рассмотрении функций двух переменных.

**Определение 9.2.1** *Линия уровня функции двух переменных  $z = f(x, y)$  с областью определения  $D$  – это множество  $C = f(x, y)$ ,  $(x, y) \in D$ ,  $C \in \mathbb{R}$ .*

Итак, линии уровня – это множества, получающиеся в результате пересечения графика функции  $z = f(x, y)$  с различными плоскостями  $z = C$ , параллельными плоскости  $Oxy$ .

Давайте обратимся к примеру из географии. Пусть  $x$  и  $y$  отвечают за ширину и глубину горы,  $z$  – за высоту. Тогда, фиксируя  $z$ , мы получаем некоторое горизонтальное сечение. Множество таких сечений часто изображают на географических картах (рисунок 15).

Перейдем к построению множества точек, удовлетворяющих уравнению  $\sin(x^2 + y^2) - e^{xy} = 0$ , при  $x \in [-2, 2]$  и  $y \in [-2, 2]$ . Заданное множество отвечает линии уровня функции

$$z = \sin(x^2 + y^2) - e^{xy}$$

при  $z$  (или  $C$ ) равном нулю.

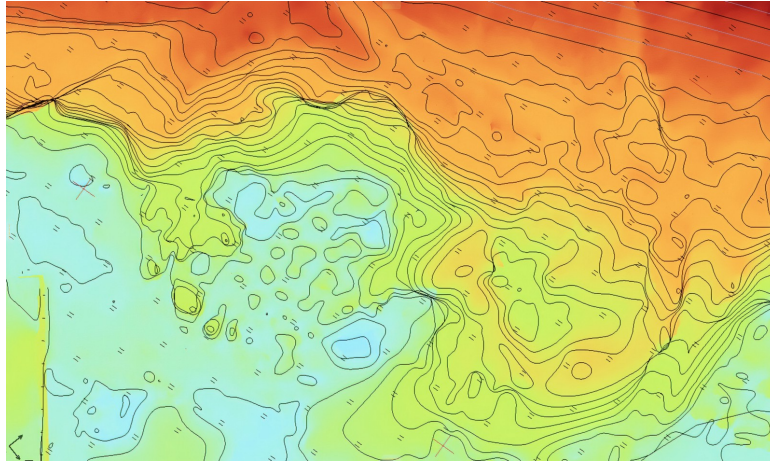


Рис. 15: Линии среза для высот местности.

Для вычисления значения функции  $z$ , в качестве входных значений  $x$  и  $y$  также выступают два массива значений, однако  $z$  необходимо вычислить для всех возможных пар  $x$  и  $y$ , то есть составить декартово произведение  $x \times y$ . Это можно сделать автоматически, воспользовавшись методом `meshgrid()` из библиотеки **NumPy**, который на основании двух одномерных массивов позволит получить два двумерных массива, в которых элементы будут составлять все возможные пары координат. Иначе говоря, если  $x$  – матрица-строка из  $n$  столбцов, а  $y$  – матрица-строка из  $m$  столбцов, метод `meshgrid()` вернет матрицу  $X$  размера  $[m \times n]$ , строки которой совпадают с матрицей-строкой  $x$ , и матрицу  $Y$  размера  $[m \times n]$ , столбцы которой совпадают с матрицей-строкой  $y$ .

```
1 x = np.linspace(1, 5, 5)
2 y = np.linspace(1, 3, 3)
3 X, Y = np.meshgrid(x, y)
```

Визуализация полученных пар координат получена с помощью следующего кода представлена на рисунке 16. Параметр `ls='none'` позволяет отключить отображение линий, соединяющих точки.

```
1 plt.plot(X, Y, color='r', marker='o', ls='none')
```

Теперь, имея возможность получить все пары точек, найдем значение функции  $z = \sin(x^2 + y^2) - e^{xy}$  и построим линию уровня методом `contour()`. Матричные операции доступны с помощью библиотеки **NumPy**:

```
1 Z = np.sin(X ** 2 + Y ** 2) - np.exp(X * Y)
```

Построим график получившейся кривой при  $z = 0$ . Два первых параметра, как обычно, – значения по оси  $X$  и  $Y$ , третий параметр – значение функции и далее массив срезов.

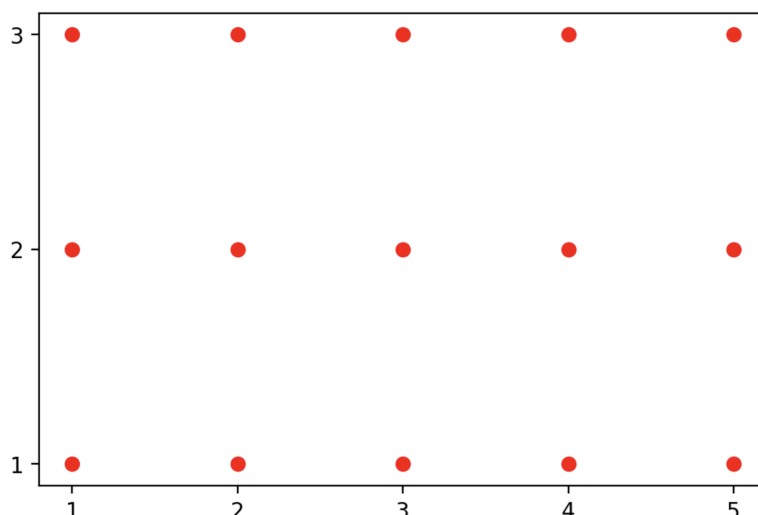
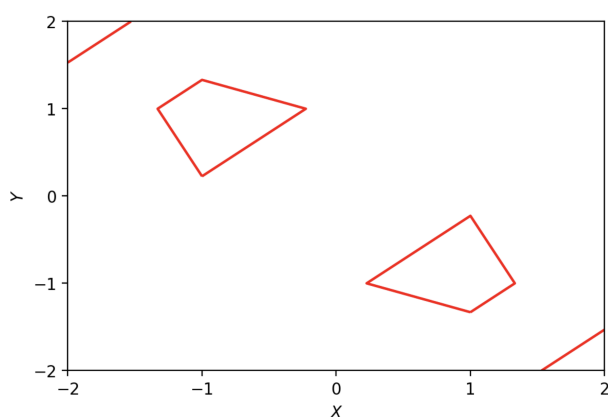


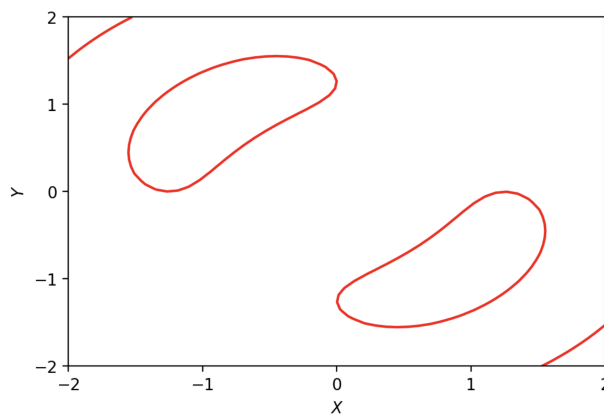
Рис. 16: Визуализация метода `meshgrid()`.

```
1 plt.contour(X, Y, Z, [0], colors=['red'])
```

Напомним, что срез построен при  $z = 0$ , так как изначально неявная функция задавалась следующим уравнением  $\sin(x^2 + y^2) - e^{xy} = 0$ . Итоговый график приведен на рисунке при разном числе точек в заданных диапазонах значений.



(a) 25 пар значений.



(b) 2500 пар значений.

Рис. 17: Графики неявной функций при разном количестве точек.

### 9.3 Точечные диаграммы. Метод `scatter()`

Точечные диаграммы в анализе данных – это полезный способ изучения взаимосвязи между двумя переменными. Например, мы можем на глаз определить вид связи или наличие явных выбросов. В дальнейшем нас будет интересовать вопрос, а не образуют ли данные некие группы?

Построение точечной диаграммы выполняется на основании двух массивов координат по оси  $X$  и  $Y$  методом `scatter()`

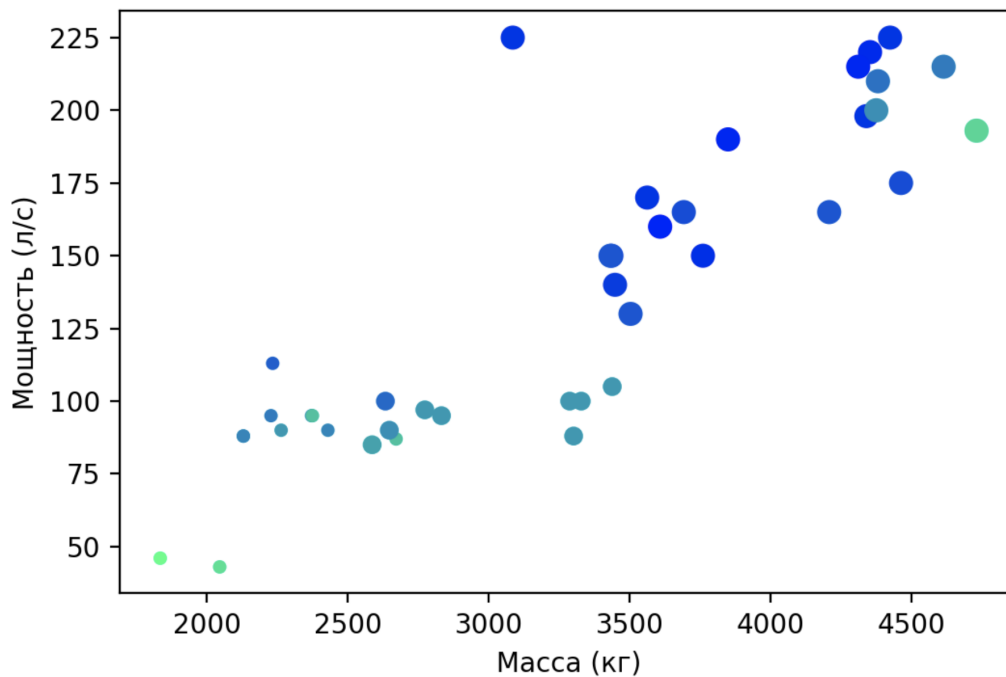


Рис. 18: Пример точечной диаграммы, полученной методом `scatter()`.

```
1 plt.scatter(x, y)
```

Опциональными являются параметры размера и цвета точек. За размер точек отвечает параметр `s`, которому присваивается массив значений размера каждой точки (т.е. длина массива координат и массива размера точек должны совпадать). Второй параметр — `c` отвечает за цвет каждой точки. Ему также присваивается массив значений цветов для каждой точки. Массив может быть из названий цветов (`red`, `blue`, ...), из строк кодов цвета в шестнадцатеричном формате, либо просто из чисел. В последнем случае цвет определяется из цветовой палитры, задаваемой параметром `cmap` и названием палитры, например `summer` или другой<sup>4</sup>.

Следующий пример кода позволяет построить точечную диаграмму из рисунка 18:

```
1 all_cars = pd.read_csv('auto-mpg.csv',
2                         index_col='car name',
3                         nrows=40,
4                         usecols=['car name', 'horsepower',
5                                 ↪ 'cylinders', 'weight',
6                                 ↪ 'acceleration'])
7
8 x = all_cars['weight']
9 y = all_cars['horsepower']
```

<sup>4</sup><https://matplotlib.org/tutorials/colors/colormaps.html>

```

8 colors = all_cars['acceleration']
9 area = all_cars['cylinders'] ** 2
10
11 plt.scatter(x, y, s=area, c=colors, cmap='winter')

```

В качестве исходных данных выступают сорок значений о автомобилях из уже известного по примерам набора данных. Выбраны следующие колонки: название автомобиля – которая используется как индекс строк, мощность, количество цилиндров, масса и ускорение (разгон в секундах до 100 км/час). По оси  $X$  отложена масса, по  $Y$  – мощность. Массив `colors` заполнен данными о ускорении автомобилей, а значение `winter` параметра `cmap` задает градацию цветов согласно палитре (рисунок 19). Более синий цвет соответ-



Рис. 19: Цветовая палитра `winter`.

ствует быстрому разгону автомобилей, зеленый – медленному. Массив `area` заполнен данными о количестве цилиндров и используется для изменения размера каждой точки. Возведение в квадрат выполнено исключительно для наглядности.