

Python предлагает множество библиотек, которые применяются на всех этапах анализа данных.

Поиск данных

С помощью библиотеки **Scrapy** можно создавать программы, которые собирают структурированные данные в сети. Также его можно использовать для сбора данных из API.

BeautifulSoup применяется там, где получить данные из API не выходит; он собирает данные и расставляет их в определенном формате.

Обработка и моделирование данных

NumPy (Numerical Python) используется для сортировки больших наборов данных. Он упрощает математические операции и их векторизацию на массивах. **Pandas** упрощает работу с табличными данными.

Визуализация данных

Matplotlib и **Seaborn** визуализация списки чисел (графики, гистограммы, диаграммы, тепловые карты и так далее).

Менеджер пакетов pip

Pip – это консольная утилита (без графического интерфейса).

При развертывании современной версии Python (начиная с Python 3.4), pip устанавливается автоматически.

Установка последней версии пакета

```
> pip install ProjectName
```

Просмотр установленных пакетов

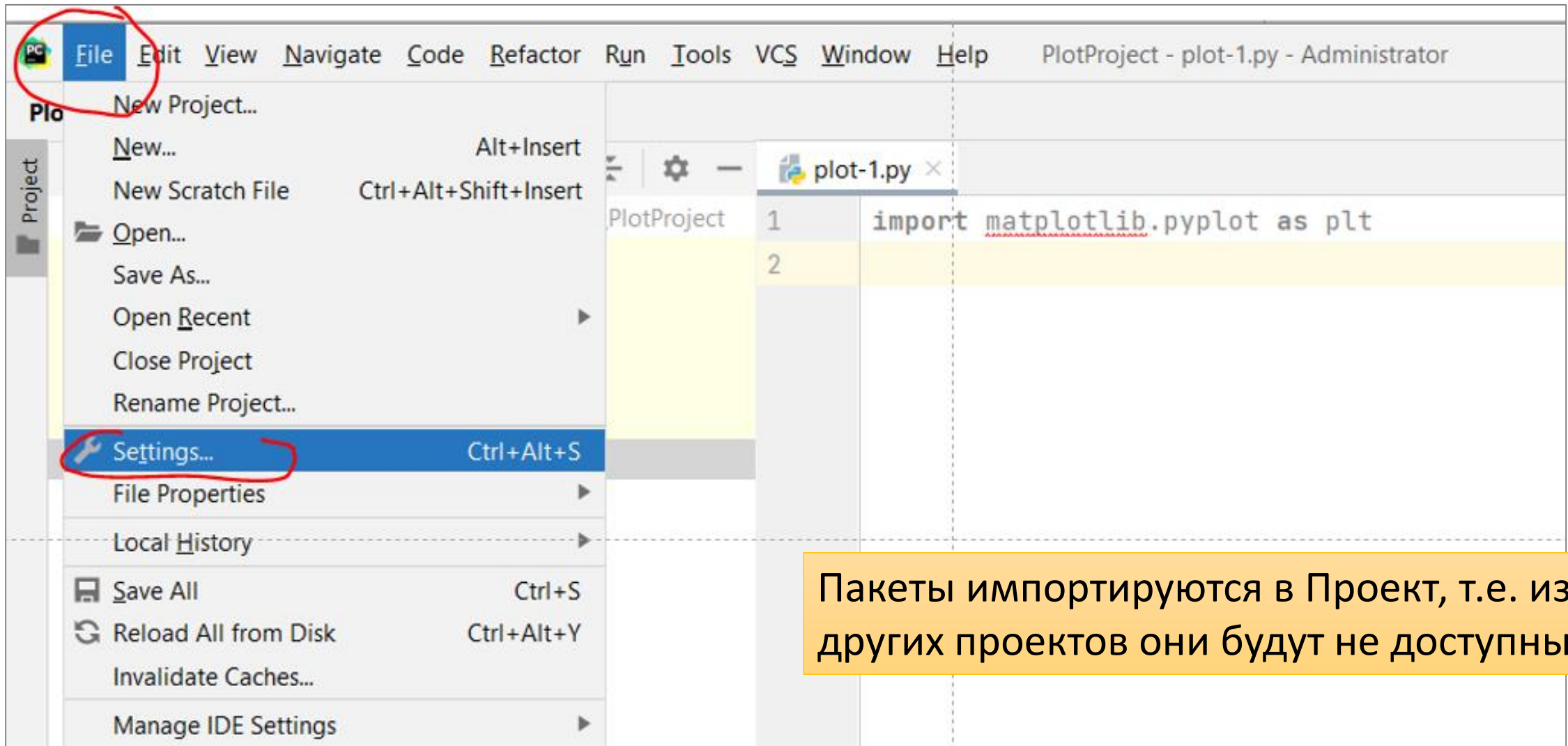
```
> pip list
```

Более подробная информацию о конкретном пакете

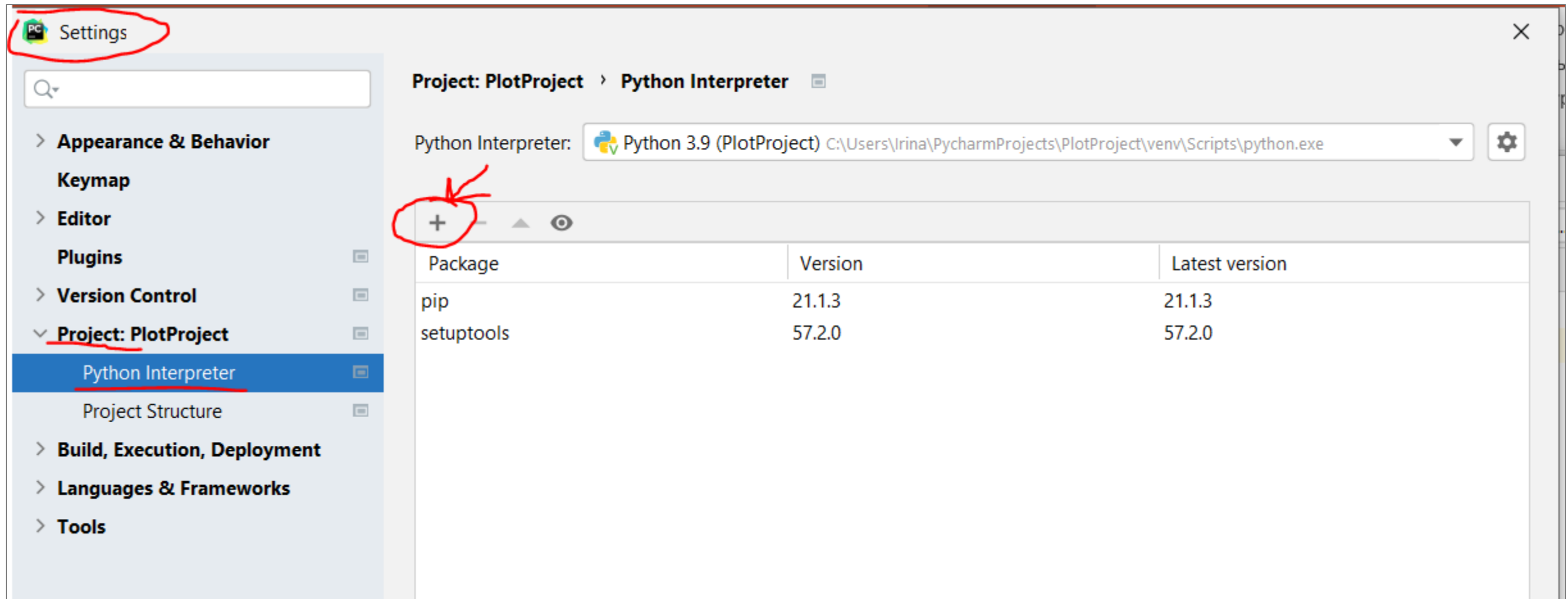
```
> pip show ProjectName
```

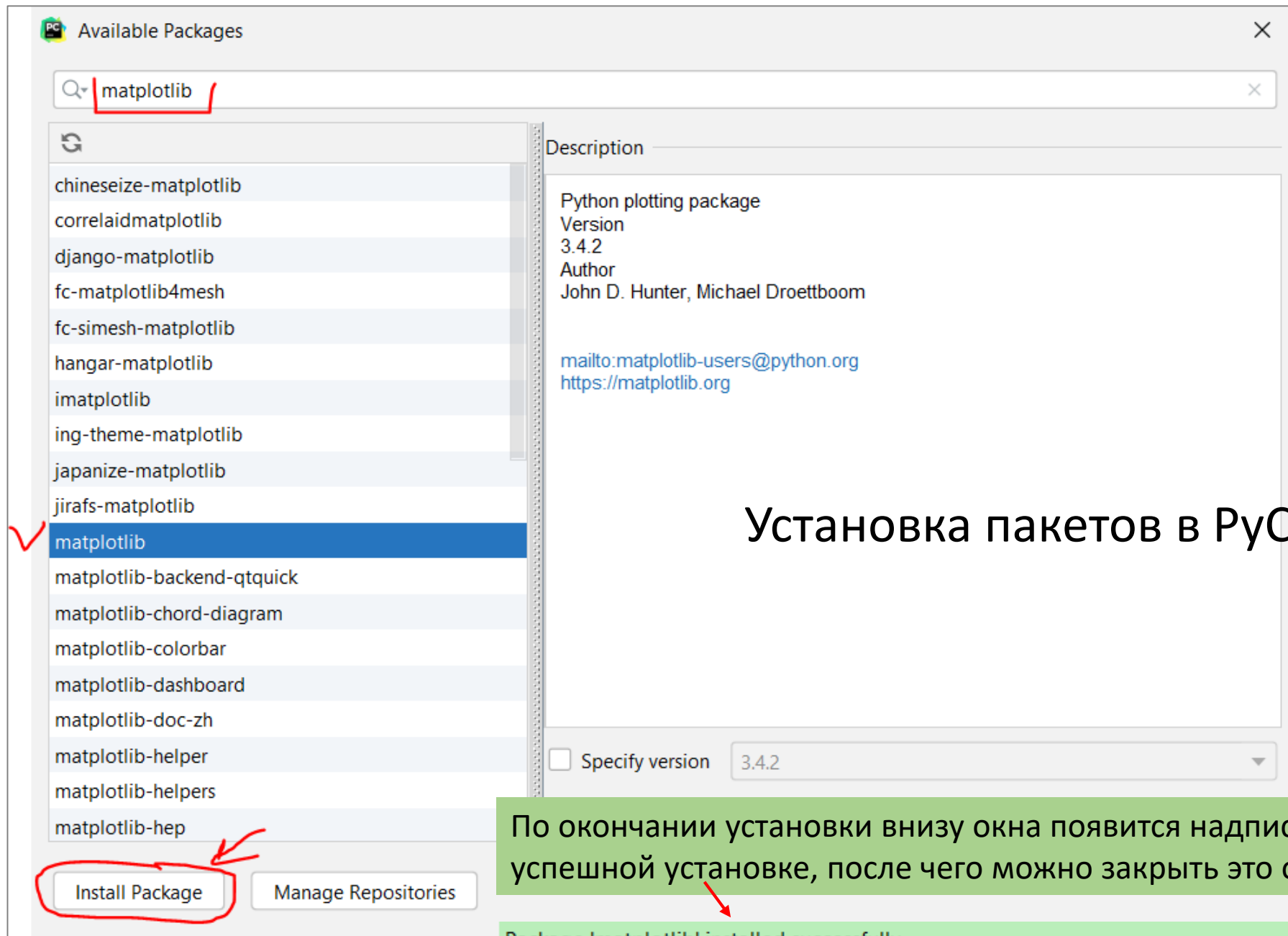
```
import pandas as pd  
import numpy as np
```

Установка пакетов в PyCharm. Шаг 1



Установка пакетов в PyCharm. Шаг 2





Установка пакетов в PyCharm. Шаг 3

По окончании установки внизу окна появится надпись об успешной установке, после чего можно закрыть это окно.

Package 'matplotlib' installed successfully

Settings

> Appearance & Behavior

Keymap

> Editor

Plugins

> Version Control

> Project: PlotProject

Python Interpreter

Project Structure

> Build, Execution, Deployment

> Languages & Frameworks

> Tools

Project: PlotProject > Python Interpreter

Python Interpreter: Python 3.9 (PlotProject) C:\Users\Irina\PycharmProjects\PlotProject\venv\Scripts\python.exe

+ - ▲ 🔍

Package	Version	Latest version
Pillow	8.3.1	8.3.1
cycler	0.10.0	0.10.0
kiwisolver	1.3.1	1.3.1
<u>matplotlib</u>	3.4.2	3.4.2
numpy	1.21.0	1.21.0
pip	21.1.3	21.1.3
pyparsing	2.4.7	2.4.7
python-dateutil	2.8.2	2.8.2
setuptools	57.2.0	57.2.0
six	1.16.0	1.16.0

6

Что такое NumPy? [demo](#)



NumPy — библиотека с открытым исходным кодом для Python, реализующая множество математических операций для работы с векторами, матрицами и массивами. Важное преимущество NumPy перед собственной реализацией массивов (например на списках) - **это векторные операции, которые происходят гораздо быстрее, последовательных.**

Фактически, NumPy - это основная математическая библиотека для работы с данными (если вы решаете задачи машинного обучения или анализа данных). Именно NumPy, а не встроенный Math.

NumPy лежит в основе других важных библиотек: Pandas (работа с табличными данными), SciPy (работы с методами оптимизации и научными расчётами), Matplotlib (построение графиков) и т.д.

Pandas — это библиотека Python, предоставляющая широкие возможности для анализа данных.

Pandas спроектирована на основе библиотеки NumPy. Такой выбор делает pandas совместимой с большинством других модулей.

Еще одно важное решение — разработка специальных структур для анализа данных. Вместо того, чтобы использовать встроенные в Python или предоставляемые другими библиотеками структуры, были разработаны две новых **Series** и **DataFrame**.

Данные, часто хранятся в форме табличек — например, в форматах .csv, .tsv или .xlsx. С помощью библиотеки Pandas такие табличные данные очень удобно загружать, обрабатывать и анализировать.

В связке с библиотеками Matplotlib и Seaborn Pandas предоставляет широкие возможности визуального анализа табличных данных.

Структуры данных в pandas

- **Series**
- **Dataframe**

Series — это объект библиотеки pandas для представления одномерных структур данных.

Series состоит из двух связанных между собой массивов. Основной содержит данные (данные любого типа NumPy), а в дополнительном, index, хранятся метки.

Series	
index	value
0	12
1	-4
2	7
3	9

Создать структуру **Series** можно на базе различных типов данных:

- словарей Python;
- списков Python;
- массивов из numpy: ndarray;
- скалярных величин.

Конструктор класса Series выглядит следующим образом:

`pandas.Series(data=None, index=None, dtype=None, name=None, copy=False, fastpath=False)`

data – массив, словарь или скалярное значение, на базе которого будет построен Series;

index – список меток, который будет использоваться для доступа к элементам Series;

dtype – объект `numpy.dtype`, определяющий тип данных;

copy – создает копию массива данных, если параметр равен `True`.

В большинстве случаев, при создании Series, используют только первые два параметра.

```
>>> s = pd.Series([12, -4, 7, 9])
```

```
>>> s
```

```
0    12
```

```
1    -4
```

```
2     7
```

```
3     9
```

```
dtype: int64
```

Series	
index	value
0	12
1	-4
2	7
3	9

```
>>> s = pd.Series([12, -4, 7, 9], index=['a', 'b', 'c', 'd'])
```

```
>>> s
```

```
a    12
```

```
b    -4
```

```
c     7
```

```
d     9
```

```
dtype: int64
```

Создание Series из массивов NumPy

```
>>> arr = np.array([1,2,3,4])
>>> s3 = pd.Series(arr)
>>> s3
0    1
1    2
2    3
3    4
```

Создание Series из уже существующих Series

```
>>> s4 = pd.Series(s)
>>> s4
a    12
b     1
c     7
d     9
```

Важно: значения в массиве NumPy или оригинальном объекте Series не копируются, а **передаются по ссылке**. Это значит, что элементы объекта вставляются динамически в новый Series. Если меняется оригинальный объект, то меняются и его значения в новом.

```
>>> arr[2] = -2
>>> s3
0    1
1    2
2   -2
3    4
```

Например, при изменении третьего элемента массива `arr`, меняется соответствующий элемент и в `s3`

Операции и математические функции

для Series можно использовать операторы (+, -, * и /):

```
>>> s / 2  
a    6.0  
b    0.5  
c    3.5  
d    4.5  
dtype: float64
```

Но в случае с математическими функциями NumPy необходимо указать функцию через np, а Series передать в качестве аргумента.

```
>>> np.log(s)  
a    2.484907  
b    0.000000  
c    1.945910  
d    2.197225  
dtype: float64
```

NaN - это значение (Not a Number) используется в структурах данных pandas для обозначения наличия пустого поля или чего-то, что невозможно обозначить в числовой форме.

Как правило, NaN — это проблема, для которой нужно найти определенное решение, особенно при работе с анализом данных. Эти данные часто появляются при извлечении информации из непроверенных источников или когда в самом источнике недостает данных.

Также значения NaN могут генерироваться в случае исключений при вычислениях или при использовании функций.

pandas позволяет явно определять NaN

```
>>> s2 = pd.Series([5, -3, np.NaN, 14])
>>> s2
0      5.0
1     -3.0
2      NaN
3     14.0
dtype: float64
```

```
>>> np.log(s)

Warning (from warnings module):
  File "C:\Users\Irina\AppData\Local\Programs\Python\Python38-32\lib\site-packages\pandas\core\arraylike.py", line 364
    result = getattr(ufunc, method)(*inputs, **kwargs)
RuntimeWarning: invalid value encountered in log
a      2.484907
b      NaN
c      1.945910
d      2.197225
dtype: float64
>>> |
```

Функции **isnull()** и **notnull()** позволяют найти значения NaN. Эти функции используют в фильтрах для создания условий.

```
>>> s2 = pd.Series([5,-3,np.NaN,14])
>>> s2
0      5.0
1     -3.0
2      NaN
3     14.0
dtype: float64
```

```
>>> s2.isnull()
0      False
1      False
2       True
3      False
dtype: bool
```

```
>>> s2.notnull()
0      True
1      True
2     False
3      True
dtype: bool
```

Фильтр,
исключающий
значения NaN

```
>>> s2[s2.notnull()]
0      5.0
1     -3.0
3     14.0
dtype: float64
```

Создание Series из словарей

```
>>> mydict = {'red': 2000, 'blue': 1000, 'yellow': 500, 'orange': 1000}
>>> myseries = pd.Series(mydict)
>>> myseries
red          2000
blue         1000
yellow        500
orange        1000
dtype: int64
>>> |
```

На этом примере можно увидеть, что массив индексов заполнен ключами, а данные — соответствующими значениями. В таком случае соотношение будет установлено между ключами dict и метками массива индексов. Если есть несоответствие, pandas заменит его на NaN.

```
>>> colors = ['red', 'yellow', 'orange', 'blue', 'green']
>>> myseries = pd.Series(mydict, index=colors)
>>> myseries
red          2000.0
yellow        500.0
orange        1000.0
blue          1000.0
green           NaN
dtype: float64
```

DataFrame – это двумерная структура данных, представляющая собой таблицу, каждый столбец которой содержит данные одного типа. Можно представлять её как словарь объектов типа Series. Структура DataFrame отлично подходит для представления реальных данных.

В отличие от Series у которого есть массив индексов с метками, ассоциированных с каждым из элементов, Dataframe имеет сразу два таких. Первый ассоциирован со строками (рядами) и напоминает таковой из Series. Каждая метка ассоциирована со всеми значениями в ряду. Второй содержит метки для каждой из колонок.

DataFrame			
index	columns		
	color	object	price
0	blue	ball	1.2
1	green	pen	1.0
2	yellow	pencil	0.6
3	red	paper	0.9
4	white	mug	1.7

DataFrame												
	date	number_of_game	day_of_week	v_name	v_league	v_game_number	h_name	h_league	h_game_number	v_score	h_score	length_outs
0	01871054	0	Thu	CL1	na	1	FW1	na	1	0	2	54.0
1	18710505	0	Fri	BS1	na	1	WS3	na	1	20	18	54.0
2	18710506	0	Sat	CL1	na	2	RC1	na	1	12	4	54.0

IntBlock

	0	1	2	3	4	5
0	01871054	0	1	1	0	2
1	18710505	0	1	1	20	18
2	18710506	0	2	1	12	4

ObjectBlock

	0	1	2	3	4
0	Thu	CL1	na	FW1	na
1	Fri	BS1	na	WS3	na
2	Sat	CL1	na	RC1	na

FloatBlock

	0
0	54.0
1	54.0
2	54.0

Внутреннее представление данных разных типов в pandas
Внутри pandas столбцы данных группируются в блоки со значениями одинакового типа. На рисунке пример того, как в pandas хранятся первые 12 столбцов объекта DataFrame.

Создание DataFrame

Структуру DataFrame можно создать на базе:

- словаря (dict) в качестве элементов которого должны выступить: одномерные ndarray, списки, другие словари, структуры Series;
- двумерных ndarray;
- структуры Series;
- структурированных ndarray;
- других DataFrame.

Конструктор класса DataFrame:

```
class pandas.DataFrame(data=None, index=None, columns=None, dtype=None, copy=False)
```

data – массив ndarray, словарь (dict) или другой DataFrame;

index – список меток для записей (имена строк таблицы);

columns – список меток для полей (имена столбцов таблицы);

dtype – объект numpy.dtype, определяющий тип данных;

copy – создает копию массива данных, если параметр равен True.

Создание Dataframe из словаря

Простейший способ создания Dataframe — передать объект dict в конструктор DataFrame().
Объект dict содержит ключ для каждой колонки, которую требуется определить, а также массив значений для них.

```
>>> import pandas as pd
>>> import numpy as np
>>> data = {'color' : ['blue', 'green', 'yellow', 'red', 'white'],
            'object' : ['ball', 'pen', 'pencil', 'paper', 'mug'],
            'price' : [1.2, 1.0, 0.6, 0.9, 1.7]}
>>> frame = pd.DataFrame(data)
>>> frame
   color object  price
0   blue   ball    1.2
1  green   pen    1.0
2 yellow pencil    0.6
3   red   paper    0.9
4  white    mug    1.7
>>> |
```

	color	object	price
0	blue	ball	1.2
1	green	pen	1.0
2	yellow	pencil	0.6
3	red	paper	0.9
4	white	mug	1.7

Если объект dict содержит больше данных, чем требуется, можно сделать выборку. Для этого в конструкторе DataFrame нужно определить последовательность колонок с помощью параметра **column**. Колонки будут созданы в заданном порядке вне зависимости от того, как они расположены в объекте dict.

```
>> data = {'color': ['blue', 'green', 'yellow', 'red', 'white'], 'object': ['ball', 'pen', 'pencil', 'paper', 'mug'],  
'price': [1.2, 1.0, 0.6, 0.9, 1.7]}  
>>> frame2 = pd.DataFrame(data, columns=['object', 'price'])  
>>> frame2
```

```
...  
      object  price  
0       ball    1.2  
1        pen    1.0  
2    pencil    0.6  
3    paper    0.9  
4       mug    1.7
```

Создание DataFrame из двумерного массива

```
>>> nda1 = np.array([[1, 2, 3], [10, 20, 30]])
>>> frame4 = pd.DataFrame(nda1)
>>> frame4
```

	0	1	2
0	1	2	3
1	10	20	30

Если метки явно не заданы в массиве **index**, pandas автоматически присваивает числовую последовательность, начиная с нуля. Если же индексам Dataframe нужно присвоить метки, необходимо использовать параметр `index` и присвоить ему массив с метками.

```
>>> nda1 = np.array([[1, 2, 3], [10, 20, 30]])
>>> frame4 = pd.DataFrame(nda1, columns=['col1', 'col2', 'col3'])
>>> frame4
```

	col1	col2	col3
0	1	2	3
1	10	20	30

```
>>>
```

В большинстве случаев
простейший способ создать
матрицу значений —
использовать запись вида:
`np.arange(16).reshape((4,4)).`

```
>>> frame3 = pd.DataFrame(np.arange(16).reshape((4,4)),
index=['red', 'blue', 'yellow', 'white'],
columns=['ball', 'pen', 'pencil', 'paper'])
>>> frame3
```

	ball	pen	pencil	paper
red	0	1	2	3
blue	4	5	6	7
yellow	8	9	10	11
white	12	13	14	15

```
>>>
```

Выбор элементов DataFrame (по колонкам)

Если нужно узнать названия всех колонок Dataframe, можно вызвать атрибут `columns` для экземпляра объекта.

```
>>> frame.columns  
Index(['color', 'object', 'price'], dtype='object')
```

Указав в квадратных скобках название колонки, можно получить значений в ней. Возвращаемое значение — объект `Series`.

```
>>> frame['price']  
0    1.2  
1    1.0  
2    0.6  
3    0.9  
4    1.7  
Name: price, dtype: float64
```

Название колонки можно использовать и в качестве атрибута.

```
>>> frame.price  
0    1.2  
1    1.0  
2    0.6  
3    0.9  
4    1.7  
Name: price, dtype: float64
```

Выбор элементов DataFrame (по строкам)

Для строк внутри Dataframe используется атрибут `loc` со значением индекса нужной строки.

Возвращаемый объект — Series, где названия колонок — это уже метки массива индексов, а значения — данные Series.

```
>>> frame.loc[2]
color      yellow
object     pencil
price              0.6
Name: 2, dtype: object
```

	color	object	price
0	blue	ball	1.2
1	green	pen	1.0
2	yellow	pencil	0.6
3	red	paper	0.9
4	white	mug	1.7

Выбор отдельных строк и диапазонов.

```
>>> frame.loc[[2,4]]
```

```
>>> frame[0:1]
```

```
>>> frame[1:3]
```

Если необходимо получить одно значение из объекта, сперва нужно указать название колонки, а потом — индекс или метку строки.

```
>>> frame['object'][3]
```

```
>>> frame.loc[[2,4]]
   color object  price
2  yellow pencil   0.6
4   white   mug   1.7
>>> frame[0:1]
   color object  price
0   blue   ball   1.2
>>> frame[1:4]
   color object  price
1  green   pen    1.0
2  yellow pencil   0.6
3    red  paper   0.9
>>> frame['object'][3]
'paper'
>>>
```

Добавление новой колонки

Одна из главных особенностей структур данных pandas — их гибкость. Можно вмешаться на любом уровне для изменения внутренней структуры данных. Например, добавление новой колонки — крайне распространенная операция.

Ее можно выполнить, присвоив значение экземпляру Dataframe и определив новое имя колонки.

```
>>> frame['new'] = 24
>>> frame
```

```
>>> frame['new']=24
>>> frame
   color object  price  new
0  blue   ball    1.2    24
1  green   pen     1.0    24
2  yellow pencil    0.6    24
3   red   paper    0.9    24
4  white    mug     1.7    24
>>>
```

```
frame['new'] = [3.0, 1.3, 2.2, 0.8, 1.1]
```

```
>>> frame['new'] = [3.0, 1.3, 2.2, 0.8, 1.1]
>>> frame
   color object  price  new
0  blue   ball    1.2  3.0
1  green   pen     1.0  1.3
2  yellow pencil    0.6  2.2
3   red   paper    0.9  0.8
4  white    mug     1.7  1.1
```

Колонки Dataframe также могут быть созданы с помощью присваивания объекта Series одной из них, например:

```
>>> ser = pd.Series(np.arange(5))
>>> frame['new'] = ser
```

```
>>> ser=pd.Series(np.arange(5))
>>> frame['new']=ser
>>> frame
   color object  price  new
0  blue   ball    1.2    0
1  green   pen     1.0    1
2  yellow pencil    0.6    2
3   red   paper    0.9    3
4  white    mug     1.7    4
>>> |
```

Фильтрация

Даже для `Dataframe` можно применять фильтры, используя определенные условия. Например, вам нужно получить все значения меньше определенного числа

```
>>> df = pd.DataFrame(np.random.randn(6, 4), columns=list("ABCD"))
>>> df
```

	A	B	C	D
0	0.950291	-0.092007	0.520807	-0.069023
1	-0.566679	2.853575	1.510428	0.341496
2	0.675643	-0.842572	-0.310943	0.503564
3	-0.229467	1.829476	-0.387503	0.733274
4	-0.291800	-0.505279	1.044840	-0.291007
5	1.254795	-1.557840	-0.401389	0.845697

```
>>> df[df<0.3]
```

	A	B	C	D
0	NaN	-0.092007	NaN	-0.069023
1	-0.566679	NaN	NaN	NaN
2	NaN	-0.842572	-0.310943	NaN
3	-0.229467	NaN	-0.387503	NaN
4	-0.291800	-0.505279	NaN	-0.291007
5	NaN	-1.557840	-0.401389	NaN

```
>>> |
```

Результатом будет `Dataframe` со значениями меньше 0.3 на своих местах. На месте остальных будет `NaN`.

```
>>> frame
   color object  price  new
0  blue   ball    1.2    0
1  green   pen    1.0    1
2 yellow  pencil    0.6    2
3   red   paper    0.9    3
4  white   mug    1.7    4
```

```
>>> frame[frame==1]
   color object  price  new
0  NaN   NaN   NaN   NaN
1  NaN   NaN    1.0    1.0
2  NaN   NaN   NaN   NaN
3  NaN   NaN   NaN   NaN
4  NaN   NaN   NaN   NaN
```


Вхождение значений

Функция `isin()` используется с объектами `Series` для определения вхождения значений в колонку. Она же подходит и для объектов `Dataframe`.

```
>>> frame.isin([1.0, 'pen'])
```

item	color	object	price	new
id				
0	blue	ball	1.2	0
1	green	pen	1.0	1
2	yellow	pencil	0.6	2
3	red	paper	0.9	3
4	white	mug	1.7	4

item	color	object	price	new
id				
0	False	False	False	False
1	False	True	True	True
2	False	False	False	False
3	False	False	False	False
4	False	False	False	False

Возвращается `Dataframe` с булевыми значениями, где `True` указывает на те значения, где членство подтверждено. Если передать это значение в виде условия, тогда вернется `Dataframe`, где будут только значения, удовлетворяющие условию.

```
>>> frame[frame.isin([1.0, 'pen'])]
```

item	color	object	price	new
id				
0	NaN	NaN	NaN	NaN
1	NaN	pen	1.0	1.0
2	NaN	NaN	NaN	NaN
3	NaN	NaN	NaN	NaN
4	NaN	NaN	NaN	NaN

Dataframe из вложенного словаря

В Python часто используется вложенный dict:

```
>>> nestdict = {'red': { 2012: 22, 2013: 33},  
                'white': { 2011: 13, 2012: 22, 2013: 16},  
                'blue': { 2011: 17, 2012: 27, 2013: 18}}  
>>> frame2 = pd.DataFrame(nestdict)  
>>> frame2
```

Эта структура данных, будучи переданной в качестве аргумента в DataFrame(), интерпретируется pandas так, что внешние ключи становятся названиями колонок, а внутренние — метками индексов.

При интерпретации вложенной структуры возможно такое, что не все поля будут совпадать. pandas компенсирует это несоответствие, добавляя NaN на место недостающих значений.

	blue	red	white
2011	17	NaN	13
2012	27	22.0	22
2013	18	33.0	16

```
>>> frame2
```

	blue	red	white
2011	17	NaN	13
2012	27	22.0	22
2013	18	33.0	16

Транспонирование Dataframe

При работе с табличными структурами данных иногда появляется необходимость выполнить операцию перестановки (сделать так, чтобы колонки стали рядами и наоборот). pandas позволяет добиться этого очень просто. Достаточно добавить атрибут `T`.

```
>>> frame2.T
```

	2011	2012	2013
blue	17.0	27.0	18.0
red	NaN	22.0	33.0
white	13.0	22.0	16.0

Функция **data_range** библиотеки pandas, служит для генерации диапазонов дат и возвращает массив дат, увеличенных на дни, месяцы, года, и т.д.

- Справка <https://pandas.pydata.org/pandas-docs/stable/reference/frame.html>
- Справка в Jupyter Notebook:

Ввод [3]: `pd.date_range?`

Examples

****Specifying the values****

The next four examples generate the same `DatetimeIndex`, but vary the combination of `start`, `end` and `periods`.

Specify `start` and `end`, with the default daily frequency.

```
>>> pd.date_range(start='1/1/2018', end='1/08/2018')
DatetimeIndex(['2018-01-01', '2018-01-02', '2018-01-03', '2018-01-04',
               '2018-01-05', '2018-01-06', '2018-01-07', '2018-01-08'],
              dtype='datetime64[ns]', freq='D')
```

Specify `start` and `periods`, the number of periods (days).

```
>>> pd.date_range(start='1/1/2018', periods=8)
DatetimeIndex(['2018-01-01', '2018-01-02', '2018-01-03', '2018-01-04',
```

```
In [ ]: import pandas as pd
import numpy as np
```

```
In [6]: ave_data = np.array(np.arange(24)).reshape((6,4))
dates = pd.date_range('20210101', periods=6)
ave_df = pd.DataFrame(ave_data, index=dates, columns=list('1234'))
ave_df
```

Out[6]:

	1	2	3	4
2021-01-01	0	1	2	3
2021-01-02	4	5	6	7
2021-01-03	8	9	10	11
2021-01-04	12	13	14	15
2021-01-05	16	17	18	19
2021-01-06	20	21	22	23

```
In [7]: ave_df['4']
```

Out[7]:

2021-01-01	3
2021-01-02	7
2021-01-03	11
2021-01-04	15
2021-01-05	19
2021-01-06	23

Freq: D, Name: 4, dtype: int32

```
In [9]: ave_df[1:4]
```

Out[9]:

	1	2	3	4
2021-01-02	4	5	6	7
2021-01-03	8	9	10	11
2021-01-04	12	13	14	15

10 minutes to pandas

[Intro to data structures](#)[Essential basic functionality](#)[IO tools \(text, CSV, HDF5, ...\)](#)[Indexing and selecting data](#)[MultiIndex / advanced indexing](#)[Merge, join, concatenate and compare](#)[Reshaping and pivot tables](#)[Working with text data](#)[Working with missing data](#)[Duplicate Labels](#)[Categorical data](#)[MultiIndex data structures](#)

10 minutes to pandas

This is a short introduction to pandas, geared mainly for new users. You can see more complex recipes in the [Cookbook](#).

Customarily, we import as follows:

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

```
In [2]: import pandas as pd
```

Object creation

See the [Data Structure Intro](#) section.

Creating a **Series** by passing a list of values, letting pandas create a default integer index:

Выбрать по критерию с несколькими столбцами

```
In [19]: df = pd.DataFrame(
.....:     {"AAA": [4, 5, 6, 7], "BBB": [10, 20, 30, 40], "CCC": [100, 50, -30, -50]}
.....: )
.....:
```

```
In [20]: df
```

```
Out[20]:
```

	AAA	BBB	CCC
0	4	10	100
1	5	20	50
2	6	30	-30
3	7	40	-50

... И (без присваивания возвращает серию)

```
In [21]: df.loc[(df["BBB"] < 25) & (df["CCC"] >= -40), "AAA"]
```

```
Out[21]:
```

0	4
1	5

Name: AAA, dtype: int64

... Или (без присваивания возвращает серию)

```
In [22]: df.loc[(df["BBB"] > 25) | (df["CCC"] >= -40), "AAA"]
```

```
Out[22]:
```

0	4
1	5
2	6
3	7

Name: AAA, dtype: int64

Загрузка данных из Excel

Для загрузки данных из Excel служит функция

`pandas.read_excel`

`pandas.read_excel(io, sheet_name=0, header=0, names=None, index_col=None, usecols=None, squeeze=False, dtype=None, engine=None, converters=None, true_values=None, false_values=None, skiprows=None, nrows=None, na_values=None, keep_default_na=True, na_filter=True, verbose=False, parse_dates=False, date_parser=None, thousands=None, comment=None, skipfooter=0, convert_float=None, mangle_dupe_cols=True, storage_options=None)[source]`

```
df_salesG = pd.read_excel('https://github.com/datagy/mediumdata/raw/master/pythonexcel.xlsx', sheet_name = 'sales')
df_statesG = pd.read_excel('https://github.com/datagy/mediumdata/raw/master/pythonexcel.xlsx', sheet_name = 'states')
```

```
df_sales = pd.read_excel('D:\IBA\pandas\pythonexcel.xlsx', sheet_name = 'sales')
df_states = pd.read_excel('D:\IBA\pandas\pythonexcel.xlsx', sheet_name = 'states')
```

Для просмотра начала большой таблицы можно использовать функцию `head()`

```
df_sales.head()
```

	Product	Sales	Date	City
0	Bananas	121	2019-06-13	Atlanta
1	Bananas	236	2019-10-20	Atlanta
2	Apples	981	2019-03-12	Atlanta
3	Bread	996	2019-07-28	New York City
4	Broccoli	790	2019-10-22	New York City

```
df_salesG.head(3)
```

	Product	Sales	Date	City
0	Bananas	121	2019-06-13	Atlanta
1	Bananas	236	2019-10-20	Atlanta
2	Apples	981	2019-03-12	Atlanta

Для работы функции `pandas.read_excel` может понадобиться установка модуля **openpyxl**

```
pip install openpyxl
```


Чтение определенных колонок из файла Excel

```
cols = [0, 2, 3]

df_2 = pd.read_excel('pythonexcel.xlsx', sheet_name = 'sales', usecols=cols)
df_2.head()
```

	Product	Date	City
0	Bananas	2019-06-13	Atlanta
1	Bananas	2019-10-20	Atlanta
2	Apples	2019-03-12	Atlanta
3	Bread	2019-07-28	New York City
4	Broccoli	2019-10-22	New York City

	A	B	C	D
1	Product	Sales	Date	City
2	Bananas	\$ 121,00	13.06.2019	Atlanta
3	Bananas	\$ 236,00	20.10.2019	Atlanta
4	Apples	\$ 981,00	12.03.2019	Atlanta
5	Bread	\$ 996,00	28.07.2019	New York City
6	Broccoli	\$ 790,00	22.10.2019	New York City
7	Apples	\$ 762,00	03.11.2019	Toronto
8	Bananas	\$ 870,00	18.11.2019	New York City
9	Bananas	\$ 852,00	21.03.2019	Atlanta
10	Apples	\$ 427,00	11.05.2019	Toronto
11	Bread	\$ 576,00	14.09.2019	Atlanta

Запись данных в Excel

pandas.DataFrame.to_excel

```
DataFrame.to_excel(excel_writer, sheet_name='Sheet1', na_rep='', float_format=None, columns=None, header=True, index=True, index_label=None, startrow=0, startcol=0, engine=None, merge_cells=True, encoding=None, inf_rep='inf', verbose=True, freeze_panes=None, storage_options=None) \[source\]
```

Write object to an Excel sheet.

df

	A	B	C	D	E
2021-01-01	0.373750	0.195065	0.723205	0.645976	6.459764
2021-01-02	0.505477	0.897053	0.260444	0.727749	7.277486
2021-01-03	0.435105	0.568952	0.680831	0.143844	1.438444
2021-01-04	0.575463	0.490253	0.864146	0.623469	6.234691
2021-01-05	0.076086	0.813468	0.458565	0.959124	9.591240
2021-01-06	0.597003	0.220661	0.323245	0.603117	6.031175

```
df.to_excel('df.xlsx', sheet_name='date')
```

	A	B	C	D	E	F	G
1		A	B	C	D	E	
2	2021-01-01 00:00:00	0,37375	0,195065	0,723205	0,645976	6,459764	
3	2021-01-02 00:00:00	0,505477	0,897053	0,260444	0,727749	7,277486	
4	2021-01-03 00:00:00	0,435105	0,568952	0,680831	0,143844	1,438444	
5	2021-01-04 00:00:00	0,575463	0,490253	0,864146	0,623469	6,234691	
6	2021-01-05 00:00:00	0,076086	0,813468	0,458565	0,959124	9,59124	
7	2021-01-06 00:00:00	0,597003	0,220661	0,323245	0,603117	6,031175	
8							
9							

Запись нескольких DataFrame в файл Excel

Для этого нужно использовать класс **ExcelWriter()**:

```
df1 = pd.DataFrame([["AAA", "BBB"]], columns=["Spam", "Egg"])
df2 = pd.DataFrame([["ABC", "XYZ"]], columns=["Foo", "Bar"])

writer = pd.ExcelWriter('pandas_simple.xlsx')

df1.to_excel(writer, sheet_name="Sheet1")
df2.to_excel(writer, sheet_name="Sheet2")
```

Вы можете упаковать файл Excel в zip-архив:

```
>>> import zipfile
>>> df = pd.DataFrame([["ABC", "XYZ"]], columns=["Foo", "Bar"])
>>> with zipfile.ZipFile("path_to_file.zip", "w") as zf:
...     with zf.open("filename.xlsx", "w") as buffer:
...         with pd.ExcelWriter(buffer) as writer:
...             df.to_excel(writer)
```

Пример: новый столбец E равен значениям столбца D умноженным на 10

	A	B	C	D
2021-01-01	0.272266	0.772988	0.469972	0.626029
2021-01-02	0.865388	0.873191	0.672783	0.148778
2021-01-03	0.417877	0.898086	0.933327	0.096084
2021-01-04	0.103965	0.111313	0.170599	0.610982
2021-01-05	0.216331	0.342820	0.078500	0.724536
2021-01-06	0.864635	0.081325	0.648201	0.522972

```
df['E']=[x*10 for x in df['D']]
df
```

	A	B	C	D	E
2021-01-01	0.272266	0.772988	0.469972	0.626029	6.260294
2021-01-02	0.865388	0.873191	0.672783	0.148778	1.487778
2021-01-03	0.417877	0.898086	0.933327	0.096084	0.960840
2021-01-04	0.103965	0.111313	0.170599	0.610982	6.109824
2021-01-05	0.216331	0.342820	0.078500	0.724536	7.245362
2021-01-06	0.864635	0.081325	0.648201	0.522972	5.229719

Пример: новый столбец **MoreThan500** равен «Да», если соответствующее значение в столбце **Sales** больше 500 или «Нет», если меньше либо равно 500.

```
df_sales['MoreThan500'] = ['Yes' if x > 500 else 'No' for x in df_sales['Sales']]
df_sales.head(10)
```

	Product	Sales	Date	City	MoreThan500
0	Bananas	121	2019-06-13	Atlanta	No
1	Bananas	236	2019-10-20	Atlanta	No
2	Apples	981	2019-03-12	Atlanta	Yes
3	Bread	996	2019-07-28	New York City	Yes
4	Brocolli	790	2019-10-22	New York City	Yes
5	Apples	762	2019-11-03	Toronto	Yes
6	Bananas	870	2019-11-18	New York City	Yes
7	Bananas	852	2019-03-21	Atlanta	Yes
8	Apples	427	2019-05-11	Toronto	No
9	Bread	576	2019-09-14	Atlanta	Yes

Объединение двух таблиц

pandas.merge [справка](#)

`pandas.merge(left, right, how='inner', on=None, left_on=None, right_on=None, left_index=False, right_index=False, sort=False, suffixes=('_x', '_y'), copy=True, indicator=False, validate=None)`

```
df_sales = pd.merge(df_sales, df_states, how='left', on='City')
df_sales
```

	Product	Sales	Date	City	MoreThan500	State
0	Bananas	121	2019-06-13	Atlanta	No	Georgia
1	Bananas	236	2019-10-20	Atlanta	No	Georgia
2	Apples	981	2019-03-12	Atlanta	Yes	Georgia
3	Bread	996	2019-07-28	New York City	Yes	New York
4	Broccoli	790	2019-10-22	New York City	Yes	New York

	Product	Sales	Date	City	MoreThan500
0	Bananas	121	2019-06-13	Atlanta	No
1	Bananas	236	2019-10-20	Atlanta	No
2	Apples	981	2019-03-12	Atlanta	Yes
3	Bread	996	2019-07-28	New York City	Yes

	City	State
0	Atlanta	Georgia
1	New York City	New York
2	Toronto	Ontario
3	Portland	Oregon

1. Первый аргумент метода **merge** — это исходный датафрейм.
2. Второй аргумент — это датафрейм, в котором мы ищем значения.
3. Аргумент **how** указывает на то, как именно мы хотим соединить данные.
4. Аргумент **on** указывает на переменную, по которой нужно выполнить соединение.

Чтение данных из CSV-файла (txt-файлов)

pandas.read_csv

```
pandas.read_csv(filepath_or_buffer, sep=<no_default>, delimiter=None, header='infer', names=<no_default>, index_col=None, usecols=None, squeeze=False, prefix=<no_default>, mangle_dupe_cols=True, dtype=None, engine=None, converters=None, true_values=None, false_values=None, skipinitialspace=False, skiprows=None, skipfooter=0, nrows=None, na_values=None, keep_default_na=True, na_filter=True, verbose=False, skip_blank_lines=True, parse_dates=False, infer_datetime_format=False, keep_date_col=False, date_parser=None, dayfirst=False, cache_dates=True, iterator=False, chunksize=None, compression='infer', thousands=None, decimal='.', lineterminator=None, quotechar='"', quoting=0, doublequote=True, escapechar=None, comment=None, encoding=None, encoding_errors='strict', dialect=None, error_bad_lines=None, warn_bad_lines=None, on_bad_lines=None, delim_whitespace=False, low_memory=True, memory_map=False, float_precision=None, storage_options=None)
```

[\[source\]](#)

Read a comma-separated values (csv) file into DataFrame.

Чтение данных из CSV-файла (txt-файлов)

```
data = pd.read_csv("hubble_data.csv")  
data.head()
```

	distance	recession_velocity
0	0.032	170
1	0.034	290
2	0.214	-130
3	0.263	-70
4	0.275	-185

```
hubble_data – Блокнот  
Файл  Правка  Формат  Вид  Справка  
distance,recession_velocity  
.032,170  
.034,290  
.214,-130  
.263,-70  
.275,-185  
.275,-220  
.45,200  
.5,290  
.5,270  
.63,200  
.8,300
```

Если данные в файле без заголовков

параметр `names` добавляет список заголовков

```
headers = ["column_1", "column_2"]  
data_2 = pd.read_csv("hubble_data_no_titles.csv", names=headers)  
data_2.head()
```

	column_1	column_2
0	0.032	170
1	0.034	290
2	0.214	-130
3	0.263	-70
4	0.275	-185

```
hubble_data_no_titles – Блокнот  
Файл  Правка  Формат  Вид  Справка  
.032,170  
.034,290  
.214,-130  
.263,-70  
.275,-185  
.275,-220  
.45,200  
.5,290  
.5,270
```


Если разделители не запятая, то можно получить следующий результат:

```
data = pd.read_csv("wages_hours.csv")
data.head()
```

Populating the interactive namespace from numpy and matplotlib

HRS\tRATE\tERSP\tERNO\tNEIN\tASSET\tAGE\tDEP\tRACE\tSCHOOL

0	2157	2.905	1121	291	380	7250	38.5	2.340	32.1	10.5
1	2174	2.970	1128	301	398	7744	39.3	2.335	31.2	10.5
2	2062	2.350	1214	326	185	3068	40.1	2.851	31.2	10.5
3	2111	2.511	1203	49	117	1632	22.4	1.159	31.2	10.5
4	2134	2.791	1013	594	730	12710	57.7	1.22	31.2	10.5

В этом случае используем параметр **sep**

```
data = pd.read_csv("wages_hours.csv", sep="\t")
data.head()
```

	HRS	RATE	ERSP	ERNO	NEIN	ASSET	AGE	DEP	RACE	SCHOOL
0	2157	2.905	1121	291	380	7250	38.5	2.340	32.1	10.5
1	2174	2.970	1128	301	398	7744	39.3	2.335	31.2	10.5

Запись данных в CSV-файл

pandas.DataFrame.to_csv

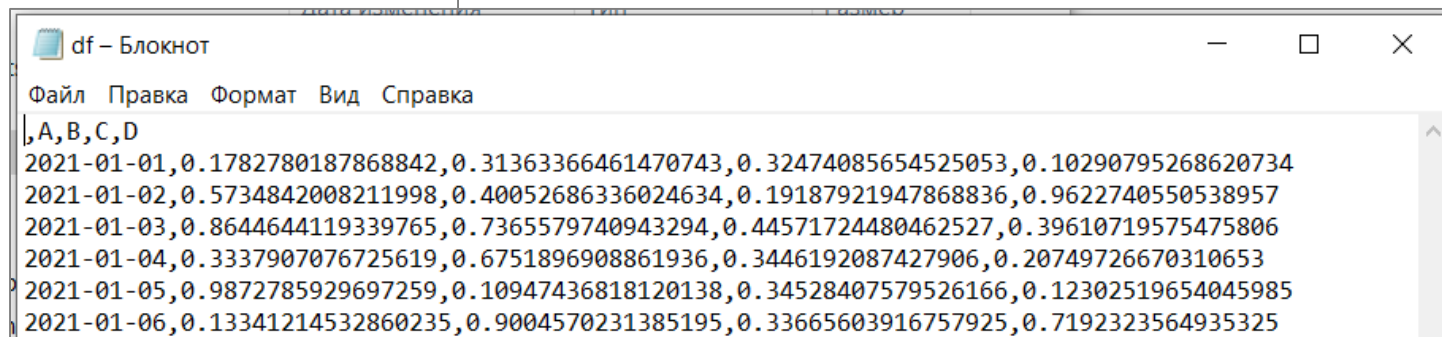
```
DataFrame.to_csv(path_or_buf=None, sep=',', na_rep='', float_format=None, columns=None, header=True, index=True, index_label=None, mode='w', encoding=None, compression='infer', quoting=None, quotechar='"', line_terminator=None, chunksize=None, date_format=None, doublequote=True, escapechar=None, decimal='.', errors='strict', storage_options=None) \[source\]
```

Write object to a comma-separated values (csv) file.

```
df = pd.DataFrame(np.random.rand(6, 4), index=dates, columns=list("ABCD"))
df
```

	A	B	C	D
2021-01-01	0.178278	0.313634	0.324741	0.102908
2021-01-02	0.573484	0.400527	0.191879	0.962274
2021-01-03	0.864464	0.736558	0.445717	0.396107
2021-01-04	0.333791	0.675190	0.344619	0.207497
2021-01-05	0.987279	0.109474	0.345284	0.123025
2021-01-06	0.133412	0.900457	0.336656	0.719232

```
df.to_csv("df.csv")
```



```
df - Блокнот
Файл  Правка  Формат  Вид  Справка
A,B,C,D
2021-01-01,0.1782780187868842,0.31363366461470743,0.32474085654525053,0.10290795268620734
2021-01-02,0.5734842008211998,0.40052686336024634,0.19187921947868836,0.9622740550538957
2021-01-03,0.8644644119339765,0.7365579740943294,0.44571724480462527,0.39610719575475806
2021-01-04,0.3337907076725619,0.6751896908861936,0.3446192087427906,0.20749726670310653
2021-01-05,0.9872785929697259,0.10947436818120138,0.34528407579526166,0.12302519654045985
2021-01-06,0.13341214532860235,0.9004570231385195,0.33665603916757925,0.7192323564935325
```

Параметр index указывает записывать ли индексы

```
df = pd.DataFrame({'name': ['Raphael', 'Donatello', 'Mark'],  
                  'mask': ['red', 'purple', 'green'],  
                  'weapon': ['sai', 'bo staff', 'sai']})
```

df

	name	mask	weapon
0	Raphael	red	sai
1	Donatello	purple	bo staff
2	Mark	green	sai

```
df.to_csv('weapon.csv', index=False)
```

```
df.to_csv('weapon.csv', index=True)
```



weapon – Блокнот

Файл Правка Формат Вид Справка

name,mask,weapon

Raphael,red,sai

Donatello,purple,bo staff

Mark,green,sai



weapon – Блокнот

Файл Правка Формат Вид Справка

,name,mask,weapon

0,Raphael,red,sai

1,Donatello,purple,bo staff

2,Mark,green,sai

Pandas Cookbook [>>>](#)

Это хранилище коротких и простых примеров и ссылок на полезные рецепты pandas. Пользователи могут сами добавлять примеры.

```
In [1]: df = pd.DataFrame(  
...:     {"AAA": [4, 5, 6, 7], "BBB": [10, 20, 30, 40], "CCC": [100, 50, -30, -50]}  
...: )  
...:
```

```
In [2]: df
```

```
Out[2]:
```

	AAA	BBB	CCC
0	4	10	100
1	5	20	50
2	6	30	-30
3	7	40	-50

если-то...

Если-то в одном столбце

```
In [3]: df.loc[df.AAA >= 5, "BBB"] = -1
```

```
In [4]: df
```

```
Out[4]:
```

	AAA	BBB	CCC
0	4	10	100
1	5	-1	50
2	6	-1	-30
3	7	-1	-50

Графики

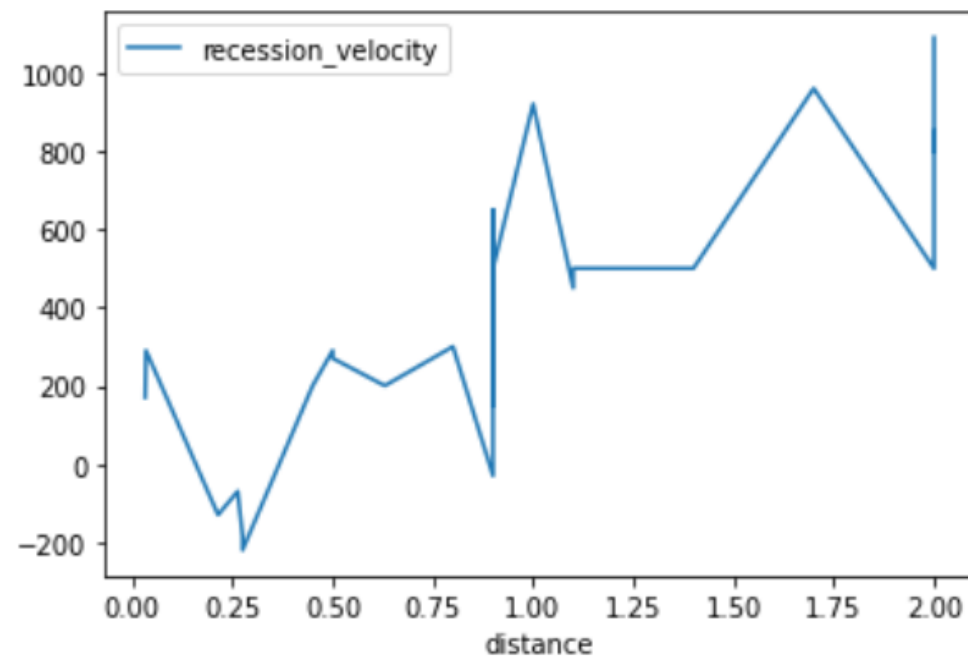
```
import pandas as pd  
  
import matplotlib.pyplot as plt
```

Out[16]:

recession_velocity	
distance	
0.032	170
0.034	290
0.214	-130
0.263	-70
0.275	-185

In [19]:

```
data.plot()  
plt.show()
```



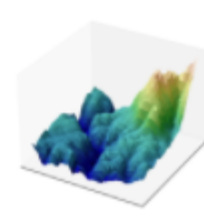
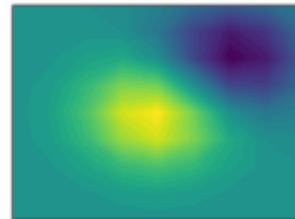
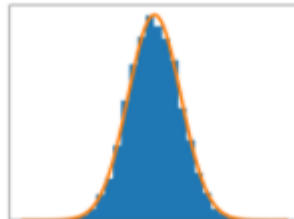


[Монтаж](#) [Документация](#) [Примеры](#) [Учебники](#) [Содействие](#)

[домой](#) | [содержание](#) » [Matplotlib: построение графиков Python](#)

Matplotlib: Визуализация с помощью Python

Matplotlib - это комплексная библиотека для создания статических, анимированных и интерактивных визуализаций в Python.



Matplotlib делает легкие вещи легкими, а сложные - возможными.

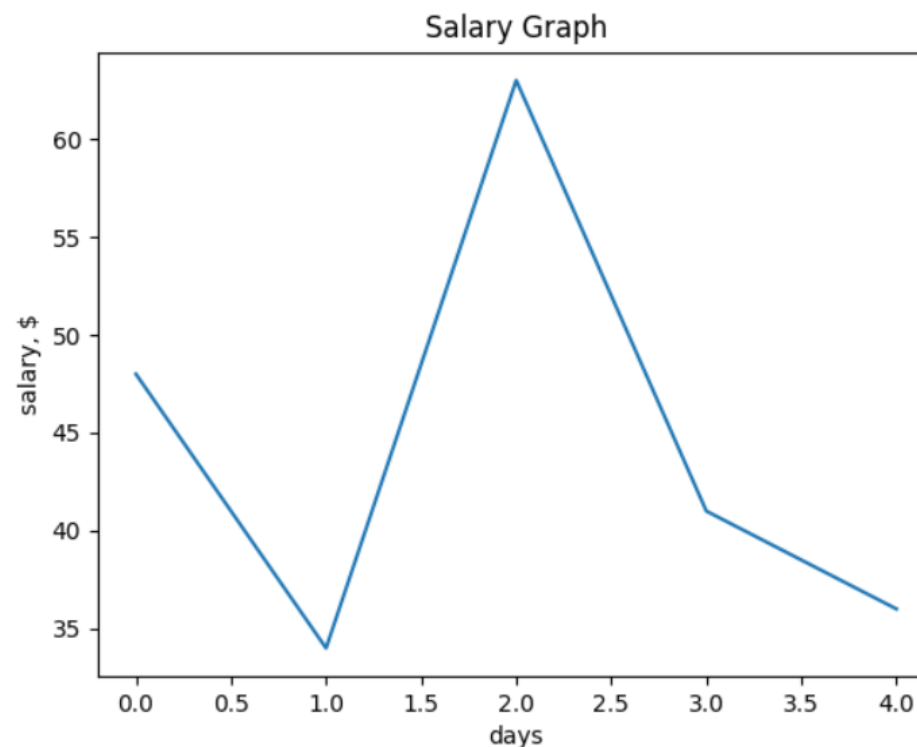
Функция **plot()** является универсальной функцией и принимает произвольное количество аргументов.

```
import matplotlib.pyplot as plt
plt.plot([1, 2, 3, 4]) ← y
plt.ylabel('some numbers')
plt.show()
```

x y

```
plt.plot([1, 2, 3, 4], [1, 4, 9, 16])
```

```
1  import matplotlib.pyplot as plt
2
3  x_list=list(range(0,5))
4  y_list= [48,34,63,41,36]
5
6  plt.title('Salary Graph')
7  plt.xlabel('days')
8  plt.ylabel('salary, $')
9  plt.plot(x_list,y_list)
10
11 plt.show()
```



Маркеры

```
import matplotlib.pyplot as plt

x_list=list(range(0,5))
y_list= [48,34,63,41,36]

plt.title('Salary Graph')
plt.xlabel('days')
plt.ylabel('salary, $')
plt.plot(x_list,y_list,marker='o')

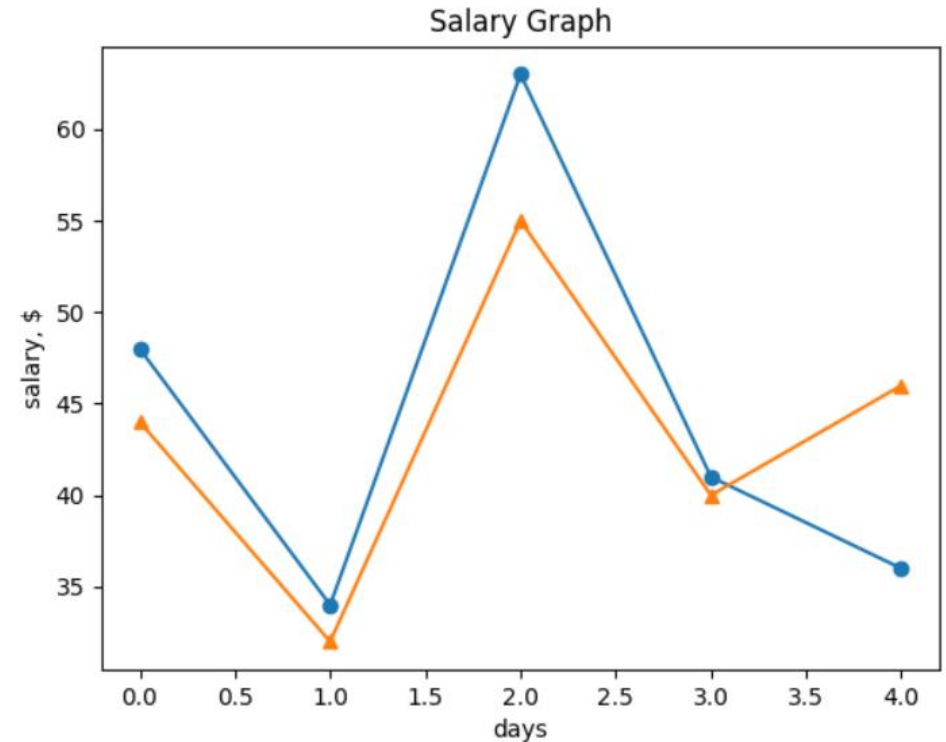
plt.show()
```

marker	symbol	description
"."	•	point
","	.	pixel
"o"	●	circle
"v"	▼	triangle_down
"^"	▲	triangle_up
"<"	◀	triangle_left
">"	▶	triangle_right
"1"	⋈	tri_down
"2"	⋊	tri_up
"3"	↰	tri_left
"4"	↱	tri_right
"8"	●	octagon
"s"	■	square
"p"	⬠	pentagon
"p"	⊕	plus (filled)
"*"	★	star

https://matplotlib.org/stable/api/markers_api.html

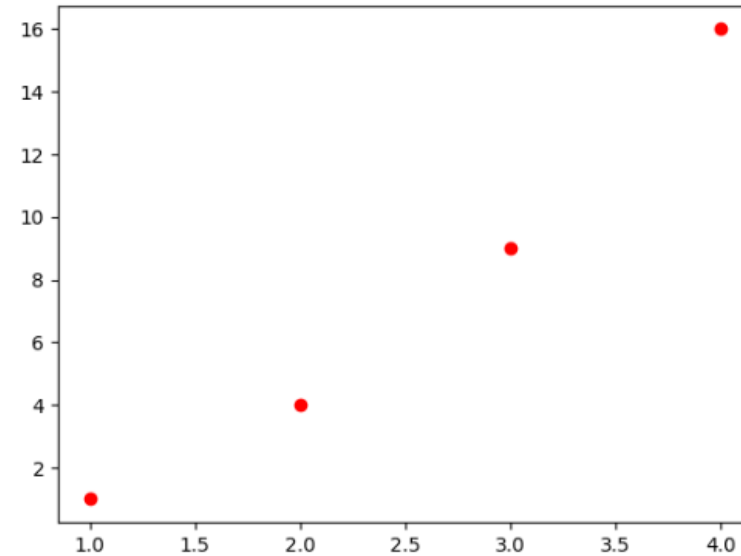
Два графика в одних осях

```
1  import matplotlib.pyplot as plt
2
3  x_list=list(range(0,5))
4  y1_list= [48,34,63,41,36]
5  y2_list= [44,32,55,40,46]
6
7  plt.title('Salary Graph')
8  plt.xlabel('days')
9  plt.ylabel('salary, $')
10 plt.plot(x_list,y1_list,marker='o')
11 plt.plot(x_list,y2_list,marker='^')
12 plt.show()
```



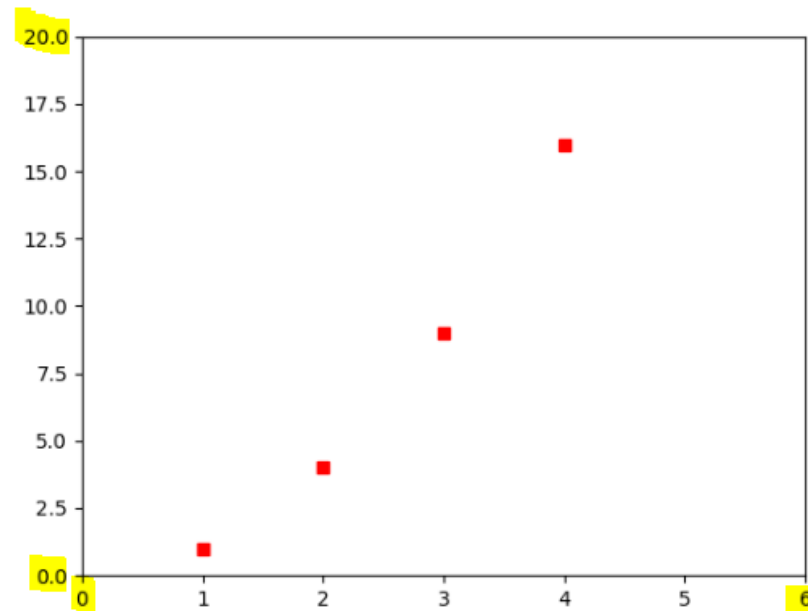
```
>>> plot(x, y)           # plot x and y using default line style and color
>>> plot(x, y, 'bo')     # plot x and y using blue circle markers
>>> plot(y)              # plot y using x as index array 0..N-1
>>> plot(y, 'r+')        # ditto, but with red plusses
```

```
1 import matplotlib.pyplot as plt
2
3 plt.plot([1, 2, 3, 4], [1, 4, 9, 16], 'ro')
4
5 plt.show()
```



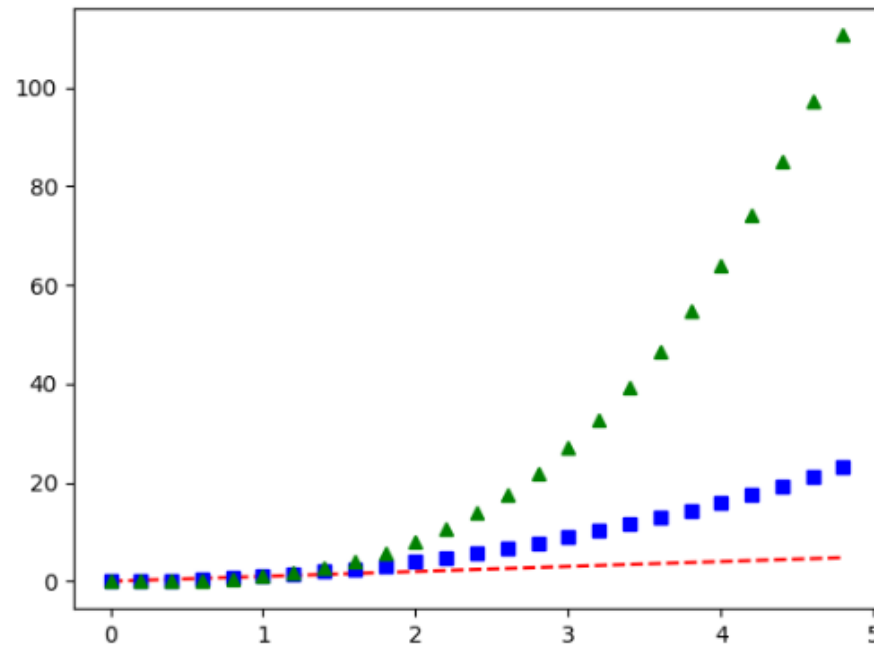
Форматирование диапазона осей

```
1 import matplotlib.pyplot as plt
2
3 plt.plot([1, 2, 3, 4], [1, 4, 9, 16], 'rs')
4 plt.axis([0, 6, 0, 20])
5
6 plt.show()
```



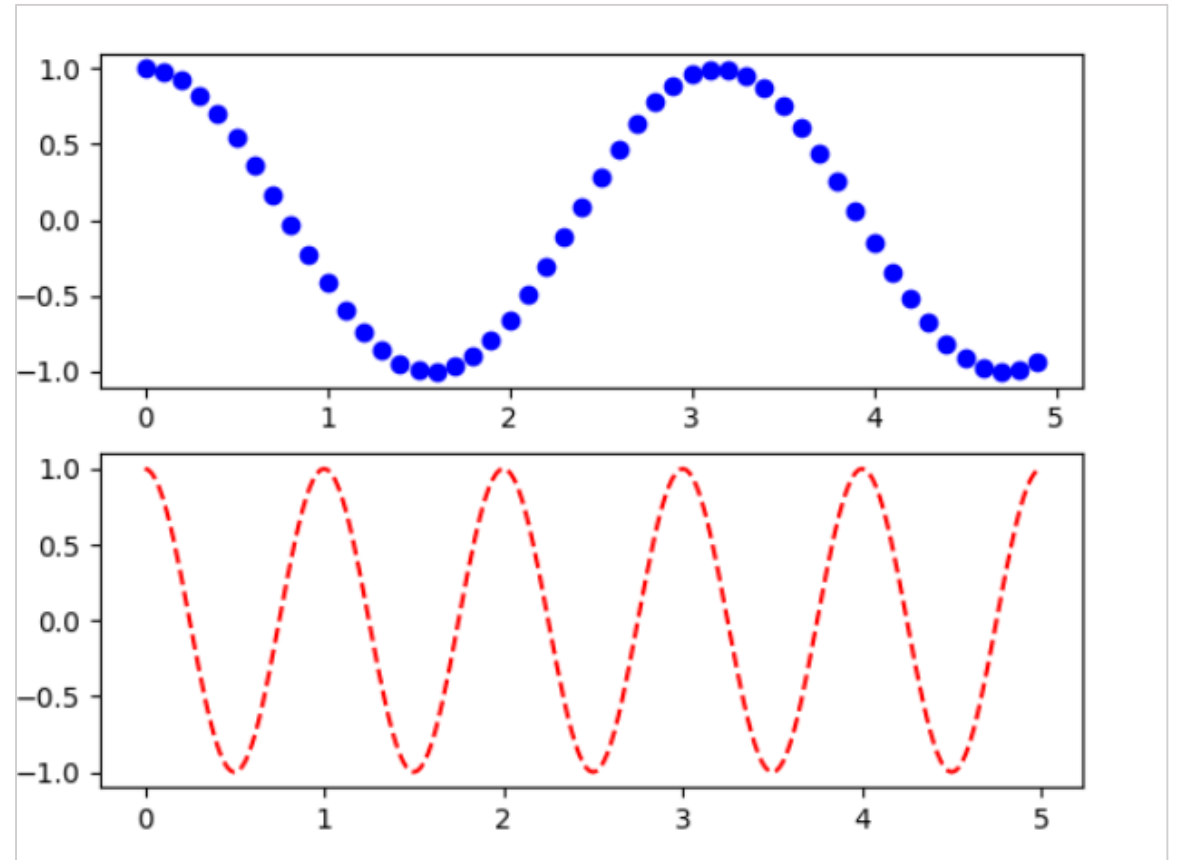
Используем массив numpy

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 t = np.arange(0., 5., 0.2)
5 # red dashes, blue squares and green triangles
6 plt.plot(t, t, 'r--', t, t**2, 'bs', t, t**3, 'g^')
7 plt.show()
```



Размещение графиков

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 def f(t):
5     return np.cos(2*np.pi*t2)
6
7 t1 = np.arange(0.0, 5.0, 0.1)
8 t2 = np.arange(0.0, 5.0, 0.02)
9
10 plt.figure()
11 plt.subplot(211)
12 plt.plot(t1, np.cos(2*t1), 'bo')
13
14 plt.subplot(212)
15 plt.plot(t2, f(t2), 'r--')
16 plt.show()
```



Параметры **subplot()**:

1. количество строк;
2. количество столбцов
3. индекс ячейки.

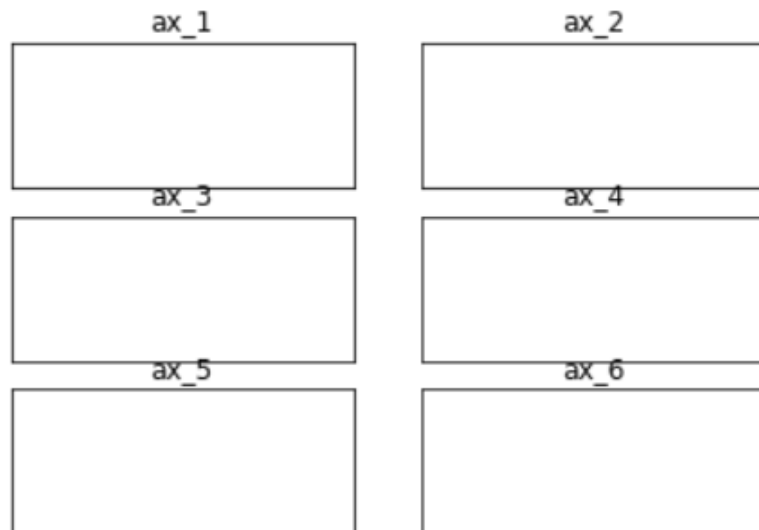
```
import matplotlib.pyplot as plt

fig = plt.figure()

ax_1 = fig.add_subplot(3, 2, 1)
ax_2 = fig.add_subplot(3, 2, 2)
ax_3 = fig.add_subplot(3, 2, 3)
ax_4 = fig.add_subplot(3, 2, 4)
ax_5 = fig.add_subplot(3, 2, 5)
ax_6 = fig.add_subplot(3, 2, 6)

ax_1.set(title = 'ax_1', xticks=[], yticks=[])
ax_2.set(title = 'ax_2', xticks=[], yticks=[])
ax_3.set(title = 'ax_3', xticks=[], yticks=[])
ax_4.set(title = 'ax_4', xticks=[], yticks=[])
ax_5.set(title = 'ax_5', xticks=[], yticks=[])
ax_6.set(title = 'ax_6', xticks=[], yticks=[])

plt.show()
```



Каждый отдельный вызов `add_subplot()` выполняет разбивку Figure, так как как указано в его параметрах и не зависит от предыдущих разбиений, это позволяет располагать графики как вам необходимо:

```
import matplotlib.pyplot as plt

fig = plt.figure()

ax_1 = fig.add_subplot(3, 1, 1)
ax_2 = fig.add_subplot(3, 2, 4)
ax_3 = fig.add_subplot(3, 3, 9)

ax_1.set(title = 'ax_1', xticks=[], yticks=[])
ax_2.set(title = 'ax_2', xticks=[], yticks=[])
ax_3.set(title = 'ax_3', xticks=[], yticks=[])

plt.show()
```

