

ВИЗУАЛИЗАЦИЯ ДАННЫХ С ИСПОЛЬЗОВАНИЕМ БИБЛИОТЕКИ MATPLOTLIB

Построение :

- ❖ графиков
- ❖ диаграмм
- ❖ гистограмм

ПОЛЕЗНЫЕ ССЫЛКИ И ИСТОЧНИКИ

- Наглядно о том, как работает NumPy: <https://habr.com/ru/companies/skillfactory/articles/564240/>
- Работа с NumPy на примерах: <https://numpy.org/devdocs/user/quickstart.html>
- 50 оттенков matplotlib — The Master Plots (с полным кодом на Python): <https://habr.com/ru/articles/468295/>
- Уэс Маккинни Python и анализ данных: Первичная обработка данных с применением pandas, NumPy и Jupiter / пер. с англ. А. А. Слинкина. 3-е изд. — М.: МК Пресс, 2023. — 536 с.: ил.

NUMPY

Прежде, чем строить графики, диаграммы, гистограммы, необходимо получить требуемые значения. Для этого следует выполнить анализ и обработку данных. Т.к. основные структуры представления и хранения данных в Python это матрицы и их модификации, то важно уметь использовать библиотеку NumPy.

- ❖ **NumPy** — это библиотека для работы с массивами и матрицами. Она предоставляет функции для выполнения математических операций над массивами, что делает её незаменимой для научных вычислений.
- ❖ NumPy позволяет эффективно работать с большими объёмами данных, выполняя операции над целыми массивами без необходимости написания циклов. Например, можно легко выполнять операции сложения, умножения или нахождения среднего значения для всего массива.

MATPLOTLIB

На этапе анализа данных важно иметь их представление в виде графических форм, полученные в результате выполнения функций обработки этих данных. Для этого будем использовать ресурсы библиотеки Matplotlib.

Matplotlib — это библиотека для построения графиков и визуализации данных. Она предоставляет гибкие инструменты для создания различных типов графиков, таких как гистограммы, точечные графики, линейные графики и многое другое. Matplotlib позволяет настраивать практически все аспекты графиков, включая заголовки, подписи осей, легенды и цвета.

ПОЗНАКОМИМСЯ С БИБЛИОТЕКОЙ NUMPY

Прежде, чем строить графики, диаграммы, гистограммы, необходимо получить требуемые значения. Для этого следует выполнить анализ и обработку данных. Т.к. основные структуры представления и хранения данных в Python это матрицы и их модификации, то важно уметь использовать библиотеку NumPy.

- ❖ **NumPy** — это библиотека для работы с массивами и матрицами. Она предоставляет функции для выполнения математических операций над массивами, что делает её незаменимой для научных вычислений.
- ❖ NumPy позволяет эффективно работать с большими объёмами данных, выполняя операции над целыми массивами без необходимости написания циклов. Например, можно легко выполнять операции сложения, умножения или нахождения среднего значения для всего массива.

NUMPY NDARRAY: ОБЪЕКТ МНОГОМЕРНОГО МАССИВА

Одна из ключевых особенностей NumPy – объект **ndarray** для представления N-мерного массива;

объект **ndarray** - это быстрый и гибкий контейнер для хранения больших наборов данных в Python.

✓ Массивы позволяют выполнять математические операции над целыми блоками данных, применяя такой же синтаксис, как для соответствующих операций над скалярами.

ndarray – это обобщенный многомерный контейнер для однородных данных, т. е. в нем могут храниться только элементы одного типа.

У любого массива есть :

атрибут **shape** – кортеж, описывающий размер по каждому измерению

```
In [17]: data.shape  
Out[17]: (2, 3)
```

атрибут **dtype** – объект, описывающий тип данных в массиве

```
In [18]: data.dtype  
Out[18]: dtype('float64')
```

NUMPY NDARRAY: ОБЪЕКТ МНОГОМЕРНОГО МАССИВА

Создание массива с помощью функции **array**.

- ✓ Функции **array** принимает любой объект, похожий на последовательность (в том числе другой массив), и порождает новый массив NumPy, содержащий переданные

данные.
Например, такое преобразование можно сделать со списком:

```
In [19]: data1 = [6, 7.5, 8, 0, 1]
In [20]: arr1 = np.array(data1)
In [21]: arr1
Out[21]: array([6. , 7.5, 8. , 0. , 1. ])
```

атрибуты **ndim** и **shape** – возвращают параметры массива

```
In [25]: arr2.ndim
Out[25]: 2

In [26]: arr2.shape
Out[26]: (2, 4)
```

Вложенные последовательности, например, список списков одинаковой длины, можно преобразовать в многомерный

```
In [22]: data2 = [[1, 2, 3, 4], [5, 6, 7, 8]]
In [23]: arr2 = np.array(data2)
In [24]: arr2
Out[24]:
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])
```

Функция **np.array** пытается самостоятельно определить подходящий тип данных для создаваемого массива. Этот тип данных хранится в специальном объекте **dtype**

```
In [27]: arr1.dtype
Out[27]: dtype('float64')

In [28]: arr2.dtype
Out[28]: dtype('int64')
```

NUMPY NDARRAY: ОБЪЕКТ МНОГОМЕРНОГО МАССИВА

Существует еще ряд функций для создания массивов.

Для создания многомерных массивов нужно передать кортеж, описывающий форму:

Например, **numpy.zeros**

```
In [29]: np.zeros(10)
Out[29]: array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

```
In [30]: np.zeros((3, 6))
Out[30]:
array([[0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.]])
```

numpy.empty
создает массив,
не инициализируя его
элементы.

```
In [31]: np.empty((2, 3, 2))
Out[31]:
array([[[0., 0.],
        [0., 0.],
        [0., 0.]],
       [[0., 0.],
        [0., 0.],
        [0., 0.]])
```

Функции создания массива

Функция	Описание
<code>array</code>	Преобразует входные данные (список, кортеж, массив или любую другую последовательность) в <code>ndarray</code> . Тип <code>dtype</code> задается явно или выводится неявно. Входные данные по умолчанию копируются
<code>asarray</code>	Преобразует входные данные в <code>ndarray</code> , но не копирует, если на вход уже подан <code>ndarray</code>
<code>arange</code>	Аналогична встроенной функции <code>range</code> , но возвращает массив, а не список
<code>ones</code> , <code>ones_like</code>	Порождает массив, состоящий из одних единиц, с заданными атрибутами <code>shape</code> и <code>dtype</code> . Функция <code>ones_like</code> принимает другой массив и порождает массив из одних единиц с такими же значениями <code>shape</code> и <code>dtype</code>
<code>zeros</code> , <code>zeros_like</code>	Аналогичны <code>ones</code> и <code>ones_like</code> , только порождаемый массив состоит из одних нулей
<code>empty</code> , <code>empty_like</code>	Создают новые массивы, выделяя под них память, но, в отличие от <code>ones</code> и <code>zeros</code> , не инициализируют ее
<code>full</code> , <code>full_like</code>	Создают массивы с заданными атрибутами <code>shape</code> и <code>dtype</code> , в которых все элементы равны заданному символу-заполнителю. <code>full_like</code> принимает массив и порождает заполненный массив с такими же значениями атрибутов <code>shape</code> и <code>dtype</code>
<code>eye</code> , <code>identity</code>	Создают единичную квадратную матрицу $N \times N$ (элементы на главной диагонали равны 1, все остальные – 0)

NUMPY NDARRAY: ОБЪЕКТ МНОГОМЕРНОГО МАССИВА

Тип данных, или **dtype**, – это специальный объект, который содержит информацию (метаданные), необходимую **ndarray** для

```
In [33]: arr1 = np.array([1, 2, 3], dtype=np.float64)
```

```
In [34]: arr2 = np.array([1, 2, 3], dtype=np.int32)
```

```
In [35]: arr1.dtype  
Out[35]: dtype('float64')
```

```
In [36]: arr2.dtype  
Out[36]: dtype('int32')
```

Типы данных NumPy

Функция	Код типа	Описание
int8, uint8	i1, u1	8-разрядное (1 байт) целое со знаком и без знака
int16, uint16	i2, u2	16-разрядное (2 байта) целое со знаком и без знака
int32, uint32	i4, u4	32-разрядное (4 байта) целое со знаком и без знака
int64, uint64	i8, u8	64-разрядное (8 байт) целое со знаком и без знака
float16	f2	С плавающей точкой половинной точности
float32	f4	Стандартный тип с плавающей точкой одинарной точности. Совместим с типом C <code>float</code>
float64	f8 или d	Стандартный тип с плавающей точкой двойной точности. Совместим с типом C <code>double</code> и с типом Python <code>float</code>
float128	f16	С плавающей точкой расширенной точности
complex64, complex128, complex256	c8, c16, c32	Комплексные числа, вещественная и мнимая части которых представлены соответственно типами <code>float32</code> , <code>float64</code> и <code>float128</code>
bool	?	Булев тип, способный хранить значения <code>True</code> и <code>False</code>
object	0	Тип объекта Python
string_	S	Тип ASCII-строки фиксированной длины (1 байт на символ). Например, строка длиной 10 имеет тип <code>'S10'</code>
unicode_	U	Тип Unicode-строки фиксированной длины (количество байтов на символ зависит от платформы). Семантика такая же, как у типа <code>string_</code> (например, <code>'U10'</code>)

NUMPY NDARRAY: ОБЪЕКТ МНОГОМЕРНОГО МАССИВА

Можно явно преобразовать, или привести, массив одного типа к другому, воспользовавшись методом

astype
Пример: целые
приведены к типу
с плавающей
точкой

```
In [37]: arr = np.array([1, 2, 3, 4, 5])  
  
In [38]: arr.dtype  
Out[38]: dtype('int64')  
  
In [39]: float_arr = arr.astype(np.float64)  
  
In [40]: float_arr  
Out[40]: array([1., 2., 3., 4., 5.])  
  
In [41]: float_arr.dtype  
Out[41]: dtype('float64')
```

Можно также
использовать атрибут
dtype другого массива:

Если исходный массив с числами с плавающей точкой, то к целому типу преобразуется, а дробная часть отбрасывается:

Если имеется массив строк, представляющих целые числа, то **astype** позволит преобразовать их в числовую форму

```
In [42]: arr = np.array([3.7, -1.2, -2.6, 0.5, 12.9, 10.1])  
  
In [43]: arr  
Out[43]: array([ 3.7, -1.2, -2.6, 0.5, 12.9, 10.1])  
  
In [44]: arr.astype(np.int32)  
Out[44]: array([ 3, -1, -2, 0, 12, 10], dtype=int32)
```

```
In [45]: numeric_strings = np.array(["1.25", "-9.6", "42"], dtype=np.string_)  
  
In [46]: numeric_strings.astype(float)  
Out[46]: array([ 1.25, -9.6 , 42.  ])
```

```
In [47]: int_array = np.arange(10)  
  
In [48]: calibers = np.array([.22, .270, .357, .380, .44, .50], dtype=np.float64)  
  
In [49]: int_array.astype(calibers.dtype)  
Out[49]: array([0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])
```

АРИФМЕТИЧЕСКИЕ ОПЕРАЦИИ С МАССИВАМИ

NUMPY

- ✓ Массивы важны, потому что позволяют выразить операции над совокупностями данных без выписывания циклов for. Обычно это называется векторизацией.

Любая арифметическая операция над массивами одинакового размера применяется к соответственным элементам:

```
In [52]: arr = np.array([[1., 2., 3.], [4., 5., 6.]])
```

```
In [53]: arr
Out[53]:
array([[1., 2., 3.],
       [4., 5., 6.]])
```

```
In [54]: arr * arr
Out[54]:
array([[ 1.,  4.,  9.],
       [16., 25., 36.]])
```

```
In [55]: arr - arr
Out[55]:
array([[0., 0., 0.],
       [0., 0., 0.]])
```

Арифметические операции, в которых участвует скаляр, применяются к каждому элементу массива:

```
In [56]: 1 / arr
Out[56]:
array([[1. , 0.5 , 0.3333],
       [0.25 , 0.2 , 0.1667]])
```

```
In [57]: arr ** 2
Out[57]:
array([[ 1.,  4.,  9.],
       [16., 25., 36.]])
```

Сравнение массивов одинакового размера дает булев массив:

```
In [58]: arr2 = np.array([[0., 4., 1.], [7., 2., 12.]])
```

```
In [59]: arr2
Out[59]:
array([[ 0.,  4.,  1.],
       [ 7.,  2., 12.]])
```

```
In [60]: arr2 > arr
Out[60]:
array([[False,  True, False],
       [ True, False,  True]])
```

Операции между массивами разного размера называются укладыванием.

ИНДЕКСИРОВАНИЕ МАССИВОВ NUMPY

- ✓ Индексирование массивов NumPy – подмножество массива или его отдельные элементы можно выбрать различными способами.

Исходный массив :

```
In [61]: arr = np.arange(10)

In [62]: arr
Out[62]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Его элемент [5]

```
In [63]: arr[5]
Out[63]: 5
```

Берём срез от элемента [5] (включается в срез) до элемента [8] (не включается в

```
In [64]: arr[5:8]
Out[64]: array([5, 6, 7])
```

Если присвоить скалярное значение срезу, как в `arr[5:8] = 12`, то оно распространяется (или укладывается) на весь

```
In [65]: arr[5:8] = 12

In [66]: arr
Out[66]: array([ 0, 1, 2, 3, 4, 12, 12, 12, 8, 9])
```

Создам еще один срез массива `arr`:

```
In [67]: arr_slice = arr[5:8]

In [68]: arr_slice
Out[68]: array([12, 12, 12])
```

Если теперь изменить значения в `arr_slice`, то изменения отразятся и на исходном массиве `arr`:

```
In [69]: arr_slice[1] = 12345

In [70]: arr
Out[70]:
array([ 0, 1, 2, 3, 4, 12, 12345, 12, 8, 9])
```

ИНДЕКСИРОВАНИЕ МАССИВОВ NUMPY

- ✓ Индексирование массивов NumPy – в случае двумерного массива элемент с заданным индексом является не скаляром, а одно мерным массивом

Для выбора одного элемента можно указать список индексов через запятую

```
In [76]: arr2d[0, 2]  
Out[76]: 3
```

Если при работе с многомерным массивом опустить несколько последних индексов, то будет возвращен объект **ndarray** меньшей размерности, содержащий данные по указанным при индексировании осям. Так, пусть имеется массив `arr3d` размерности $2 \times 2 \times 3$:

Исходный массив :

```
In [73]: arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
```

```
In [74]: arr2d[2]  
Out[74]: array([7, 8, 9])
```

```
In [77]: arr3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
```

```
In [78]: arr3d  
Out[78]:  
array([[[ 1,  2,  3],  
         [ 4,  5,  6]],  
       [[ 7,  8,  9],  
         [10, 11, 12]]])
```

Тогда `arr3d[0]` – массив размерности 2×3 :

```
In [79]: arr3d[0]  
Out[79]:  
array([[1, 2, 3],  
       [4, 5, 6]])
```


ИНДЕКСИРОВАНИЕ МАССИВОВ NUMPY

Исходный массив

```
In [73]: arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
```

```
In [74]: arr2d[2]
```

```
Out[74]: array([7, 8, 9])
```

Выражению

`arr3d[0]` можно присвоить как скалярное значение, так и массив:

```
In [80]: old_values = arr3d[0].copy()
```

```
In [81]: arr3d[0] = 42
```

```
In [82]: arr3d
```

```
Out[82]:
```

```
array([[[42, 42, 42],
         [42, 42, 42]],
       [[ 7,  8,  9],
        [10, 11, 12]]])
```

```
In [83]: arr3d[0] = old_values
```

```
In [84]: arr3d
```

```
Out[84]:
```

```
array([[[ 1,  2,  3],
         [ 4,  5,  6]],
       [[ 7,  8,  9],
        [10, 11, 12]]])
```

Аналогично `arr3d[1, 0]` дает все значения, список индексов которых начинается с (1, 0), т. е. одномерный массив

```
In [85]: arr3d[1, 0]
```

```
Out[85]: array([7, 8, 9])
```

ИНДЕКСИРОВАНИЕ МАССИВОВ **NUMPY**

Возможно выполнять обращение к элементам массивов еще несколькими способами:

- ✓ Индексирование срезами
- ✓ Булево индексирование
- ✓ Прихотливое индексирование
- ✓ Транспонирование массивов и перестановка осей

Их вы можете при необходимости самостоятельно изучить :
Уэс Маккинни Python и анализ данных: Первичная обработка данных с применением pandas, NumPy и Jupiter / пер. с англ.

А. А. Слинкина. 3-е изд. – М.: МК Пресс, 2023. – 536 с.: ил.

УНИВЕРСАЛЬНЫЕ ФУНКЦИИ: БЫСТРЫЕ ПОЭЛЕМЕНТНЫЕ ОПЕРАЦИИ НАД МАССИВАМИ

Универсальной функцией, или *u*-функцией, называется функция, которая выполняет поэлементные операции над данными, хранящимися в объектах **ndarray**. Можно считать, что это векторные обертки вокруг простых функций, которые принимают одно или несколько скалярных значений и порождают один или несколько скалярных результатов.

Многие *u*-функции – простые поэлементные преобразования, например `numpy.sqrt` или `numpy.exp`: Такие *u*-функции называются унарными.

```
In [150]: arr = np.arange(10)

In [151]: arr
Out[151]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [152]: np.sqrt(arr)
Out[152]:
array([0.    , 1.    , 1.4142, 1.7321, 2.    , 2.2361, 2.4495, 2.6458, 2.8284, 3.    ])

In [153]: np.exp(arr)
Out[153]:
array([ 1.    ,  2.7183,  7.3891, 20.0855, 54.5982, 148.4132,
 403.4288, 1096.6332, 2980.958 , 8103.0839])
```

УНИВЕРСАЛЬНЫЕ ФУНКЦИИ: БЫСТРЫЕ ПОЭЛЕМЕНТНЫЕ ОПЕРАЦИИ НАД МАССИВАМИ

Другие, например `numpy.add` или `numpy.maximum`, принимают 2 массива (и потому называются бинарными) и возвращают один результирующий массив:

```
In [154]: x = rng.standard_normal(8)

In [155]: y = rng.standard_normal(8)

In [156]: x
Out[156]:
array([-1.3678,  0.6489,  0.3611, -1.9529,  2.3474,  0.9685, -0.7594,
        0.9022])

In [157]: y
Out[157]:
array([-0.467 , -0.0607,  0.7888, -1.2567,  0.5759,  1.399 ,  1.3223,
       -0.2997])

In [158]: np.maximum(x, y)
Out[158]:
array([-0.467 ,  0.6489,  0.7888, -1.2567,  2.3474,  1.399 ,  1.3223,
        0.9022])
```

Здесь `numpy.maximum` вычисляет поэлементные максимумы массивов `x` и `y`.

УНИВЕРСАЛЬНЫЕ ФУНКЦИИ: БЫСТРЫЕ ПОЭЛЕМЕНТНЫЕ ОПЕРАЦИИ НАД МАССИВАМИ

Некоторые унарные универсальные функции Некоторые бинарные универсальные функции

Функция	Описание
<code>abs, fabs</code>	Вычислить абсолютную величину целых, вещественных или комплексных элементов массива
<code>sqrt</code>	Вычислить квадратный корень из каждого элемента. Эквивалентно <code>arg ** 0.5</code>
<code>square</code>	Вычислить квадрат каждого элемента. Эквивалентно <code>arg ** 2</code>
<code>exp</code>	Вычислить экспоненту e^x каждого элемента
<code>log, log10, log2, log1p</code>	Натуральный (по основанию e), десятичный, двоичный логарифм и функция <code>log(1 + x)</code> соответственно
<code>sign</code>	Вычислить знак каждого элемента: 1 (для положительных чисел), 0 (для нуля) или -1 (для отрицательных чисел)
<code>ceil</code>	Вычислить для каждого элемента наименьшее целое число, не меньшее его
<code>floor</code>	Вычислить для каждого элемента наибольшее целое число, не большее его
<code>rint</code>	Округлить элементы до ближайшего целого с сохранением <code>dtype</code>
<code>modf</code>	Вернуть дробные и целые части массива в виде отдельных массивов
<code>isnan</code>	Вернуть булев массив, показывающий, какие значения являются NaN (не числами)
<code>isfinite, isinf</code>	Вернуть булев массив, показывающий, какие элементы являются конечными (не inf и не NaN) или бесконечными соответственно
<code>cos, cosh, sin, sinh, tan, tanh</code>	Обычные и гиперболические тригонометрические функции
<code>arccos, arccosh, arcsin, arcsinh, arctan, arctanh</code>	Обратные тригонометрические функции
<code>logical_not</code>	Вычислить значение истинности <code>not x</code> для каждого элемента. Эквивалентно <code>~arg</code>

Функция	Описание
<code>add</code>	Сложить соответственные элементы массивов
<code>subtract</code>	Вычесть элементы второго массива из соответственных элементов первого
<code>multiply</code>	Перемножить соответственные элементы массивов
<code>divide, floor_divide</code>	Деление и деление с отбрасыванием остатка
<code>power</code>	Возвести элементы первого массива в степени, указанные во втором массиве
<code>maximum, fmax</code>	Поэлементный максимум. Функция <code>fmax</code> игнорирует значения NaN
<code>minimum, fmin</code>	Поэлементный минимум. Функция <code>fmin</code> игнорирует значения NaN
<code>mod</code>	Поэлементный модуль (остаток от деления)
<code>copysign</code>	Копировать знаки значений второго массива в соответственные элементы первого массива
<code>greater, greater_equal, less, less_equal, equal, not_equal</code>	Поэлементное сравнение, возвращается булев массив. Эквивалентны инфиксным операторам <code>></code> , <code>>=</code> , <code><</code> , <code><=</code> , <code>==</code> , <code>!=</code>
<code>logical_and, logical_or, logical_xor</code>	Вычислить логическое значение истинности логических операций. Эквивалентны инфиксным операторам <code>&</code> , <code> </code> , <code>^</code>

ПРОГРАММИРОВАНИЕ НА ОСНОВЕ МАССИВОВ

С помощью массивов NumPy многие виды обработки данных можно записать очень кратко, не прибегая к циклам. Такой способ замены явных циклов выражениями-массивами обычно называется векторизацией. Вообще говоря, векторные операции с массивами выполняются на один-два (а то и больше) порядка быстрее, чем эквивалентные операции на чистом Python.

Простой пример: Требуется вычислить функцию $\sqrt{x^2 + y^2}$ на регулярной сетке.

1) Функция `np.meshgrid` принимает два одномерных массива и порождает две двумерные матрицы, соответствующие всем парам (x, y) элементов, взятых из обоих

```
In [169]: points = np.arange(-5, 5, 0.01) # 100 равноотстоящих точек
```

```
In [170]: xs, ys = np.meshgrid(points, points)
```

```
In [171]: ys
```

```
Out[171]:
```

```
array([[ -5.   , -5.   , -5.   , ..., -5.   , -5.   , -5.   ],
       [ -4.99, -4.99, -4.99, ..., -4.99, -4.99, -4.99],
       [ -4.98, -4.98, -4.98, ..., -4.98, -4.98, -4.98],
       ...,
       [  4.97,  4.97,  4.97, ...,  4.97,  4.97,  4.97],
       [  4.98,  4.98,  4.98, ...,  4.98,  4.98,  4.98],
       [  4.99,  4.99,  4.99, ...,  4.99,  4.99,  4.99]])
```

2) Теперь для вычисления функции достаточно написать такое же выражение, как для двух точек:

```
In [172]: z = np.sqrt(xs ** 2 + ys ** 2)
```

```
In [173]: z
```

```
Out[173]:
```

```
array([[ 7.0711,  7.064 ,  7.0569, ...,  7.0499,  7.0569,  7.064 ],
       [ 7.064 ,  7.0569,  7.0499, ...,  7.0428,  7.0499,  7.0569],
       [ 7.0569,  7.0499,  7.0428, ...,  7.0357,  7.0428,  7.0499],
       ...,
       [ 7.0499,  7.0428,  7.0357, ...,  7.0286,  7.0357,  7.0428],
       [ 7.0569,  7.0499,  7.0428, ...,  7.0357,  7.0428,  7.0499],
       [ 7.064 ,  7.0569,  7.0499, ...,  7.0428,  7.0499,  7.0569]])
```

ПРОГРАММИРОВАНИЕ НА ОСНОВЕ МАССИВОВ

Продолжение примера: Требуется вычислить функцию $\sqrt{x^2 + y^2}$ на регулярной сетке.

3) Воспользуемся библиотекой `matplotlib` для визуализации двумерного массива:

```
In [174]: import matplotlib.pyplot as plt
```

```
In [175]: plt.imshow(z, cmap=plt.cm.gray, extent=[-5, 5, -5, 5])
```

```
Out[175]: <matplotlib.image.AxesImage at 0x7f624ae73b20>
```

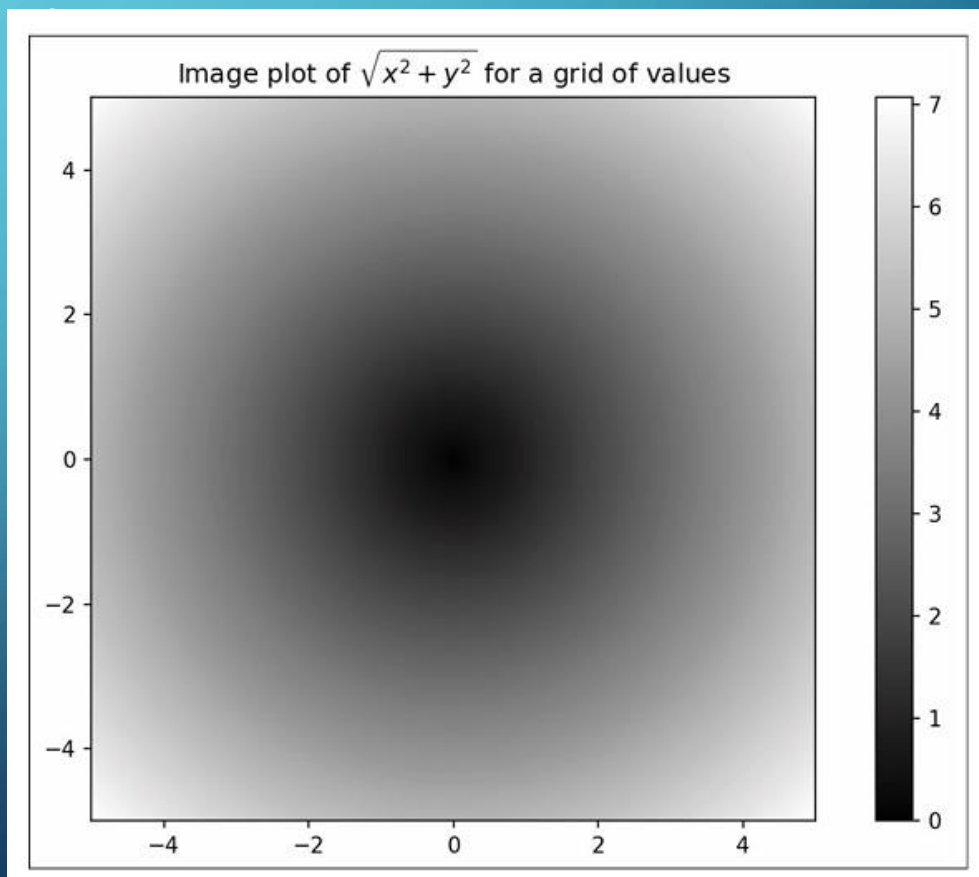
```
In [176]: plt.colorbar()
```

```
Out[176]: <matplotlib.colorbar.Colorbar at 0x7f6253e43ee0>
```

```
In [177]: plt.title("Image plot of  $\sqrt{x^2 + y^2}$  for a grid of values")
```

```
Out[177]: Text(0.5, 1.0, 'Image plot of  $\sqrt{x^2 + y^2}$  for a grid of values')
```

Результат применения функции `imshow` из библиотеки `matplotlib` для создания изображения по двумерному массиву значений



ПРОГРАММИРОВАНИЕ НА ОСНОВЕ МАССИВОВ

- ✓ Запись логических условий в виде операций с массивами

Функция **numpy.where** – векторный вариант тернарного выражения `x if condition else y`.

Пример: Надо брать значение из массива `xarr`, если соответствующее значение в массиве `cond` равно `True`, а в противном случае – значение из `yarr`.

Эту задачу решает функция `numpy.where` написать очень лаконичный код:

```
In [185]: result = np.where(cond, xarr, yarr)
```

```
In [186]: result
```

```
Out[186]: array([1.1, 2.2, 1.3, 1.4, 2.5])
```

Исходные данные:

Пусть имеется булев массив и два массива значений

```
In [180]: xarr = np.array([1.1, 1.2, 1.3, 1.4, 1.5])
```

```
In [181]: yarr = np.array([2.1, 2.2, 2.3, 2.4, 2.5])
```

```
In [182]: cond = np.array([True, False, True, True, False])
```

ПРОГРАММИРОВАНИЕ НА ОСНОВЕ МАССИВОВ

Функция **numpy.where** – векторный вариант тернарного выражения $x \text{ if condition else } y$.

Пример: имеется матрица со случайными данными, и мы хотим заменить все положительные значения на 2, а все отрицательные – на -2.

С помощью **numpy.where** сделать это очень просто:

Исходные данные

1) Создаем массив :

```
In [187]: arr = rng.standard_normal((4, 4))

In [188]: arr
Out[188]:
array([[ 2.6182,  0.7774,  0.8286, -0.959 ],
       [-1.2094, -1.4123,  0.5415,  0.7519],
       [-0.6588, -1.2287,  0.2576,  0.3129],
       [-0.1308,  1.27   , -0.093 , -0.0662]])
```

С помощью **numpy.where** меняем
элементы на 2 и -2

```
In [190]: np.where(arr > 0, 2, -2)
Out[190]:
array([[ 2,  2,  2, -2],
       [-2, -2,  2,  2],
       [-2, -2,  2,  2],
       [-2,  2, -2, -2]])
```

С помощью **numpy.where** можно комбинировать скаляры и массивы.

Например, заменить в этом примере все
положительные элементы **arr** константой 2:

```
In [191]: np.where(arr > 0, 2, arr) # заменить положительные элементы на 2
Out[191]:
array([[ 2.    ,  2.    ,  2.    , -0.959 ],
       [-1.2094, -1.4123,  2.    ,  2.    ],
       [-0.6588, -1.2287,  2.    ,  2.    ],
       [-0.1308,  2.    , -0.093 , -0.0662]])
```

МАТЕМАТИЧЕСКИЕ И СТАТИСТИЧЕСКИЕ ОПЕРАЦИИ

Среди методов класса массива имеются математические функции, которые вычисляют статистики массива в целом или данных вдоль одной оси. Выполнить агрегирование (часто его называют редукцией) типа **sum**, **mean** или стандартного отклонения **std** можно как с помощью метода экземпляра массива, так и функции на верхнем уровне **NumPy**. При использовании таких функций **NumPy**, как **numpy.sum**, агрегируемый массив следует передавать в первом аргументе.

Пример: сгенерируем случайные данные с нормальным распределением и вычислим некоторые агрегаты.

```
In [192]: arr = rng.standard_normal((5, 4))

In [193]: arr
Out[193]:
array([[ -1.1082,  0.136 ,  1.3471,  0.0611],
       [  0.0709,  0.4337,  0.2775,  0.5303],
       [  0.5367,  0.6184, -0.795 ,  0.3    ],
       [-1.6027,  0.2668, -1.2616, -0.0713],
       [  0.474 , -0.4149,  0.0977, -1.6404]])

In [194]: arr.mean()
Out[194]: -0.08719744457434529

In [195]: np.mean(arr)
Out[195]: -0.08719744457434529

In [196]: arr.sum()
Out[196]: -1.743948891486906
```

Функции типа **mean** и **sum** принимают необязательный аргумент **axis**, при на личии которого вычисляется статистика по заданной оси, и в результате рождается массив на единицу меньшей размерности:

```
In [197]: arr.mean(axis=1)
Out[197]: array([ 0.109 ,  0.3281,  0.165 , -0.6672, -0.3709])

In [198]: arr.sum(axis=0)
Out[198]: array([-1.6292,  1.0399, -0.3344, -0.8203])
```

Здесь **arr.mean(axis=1)** означает «вычислить среднее по столбцам», а **arr.sum(axis=0)** – «вычислить сумму по строкам»

МАТЕМАТИЧЕСКИЕ И СТАТИСТИЧЕСКИЕ ОПЕРАЦИИ

Другие методы, например **cumsum** и **cumprod**, ничего не агрегируют, а порождают массив промежуточных результатов:

Для многомерных массивов функция **cumsum** и другие функции с нарастающим итогом возвращают массив того же размера, элементами которого являются частичные агрегаты по указанной оси, вычисленные для каждого среза меньшей размерности.

```
In [201]: arr = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])

In [202]: arr
Out[202]:
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

```
In [199]: arr = np.array([0, 1, 2, 3, 4, 5, 6, 7])
```

```
In [200]: arr.cumsum()
Out[200]: array([ 0, 1, 3, 6, 10, 15, 21, 28])
```

Выражение `arr.cumsum(axis=0)` вычисляет сумму с накопительным итогом по строкам, а `arr.cumsum(axis=1)` – сумму с накопительным итогом по столбцам:

```
In [203]: arr.cumsum(axis=0)
Out[203]:
array([[0, 1, 2],
       [3, 5, 7],
       [9, 12, 15]])
```

```
In [204]: arr.cumsum(axis=1)
Out[204]:
array([[0, 1, 3],
       [3, 7, 12],
       [6, 13, 21]])
```

МАТЕМАТИЧЕСКИЕ И СТАТИСТИЧЕСКИЕ ОПЕРАЦИИ

Статистические методы массива

Метод	Описание
<code>sum</code>	Сумма элементов всего массива или вдоль одной оси. Для массивов нулевой длины функция <code>sum</code> возвращает 0
<code>mean</code>	Среднее арифметическое. Для массивов нулевой длины равно NaN
<code>std, var</code>	Стандартное отклонение и дисперсия соответственно
<code>min, max</code>	Минимум и максимум
<code>argmin, argmax</code>	Индексы минимального и максимального элементов
<code>cumsum</code>	Сумма с нарастающим итогом с начальным значением 0
<code>cumprod</code>	Произведение с нарастающим итогом с начальным значением 1

МЕТОДЫ БУЛЕВЫХ МАССИВОВ

В методах булевы значения приводятся к 1 (True) и 0 (False). Поэтому функция `sum` часто используется для подсчета значений True в булевом массиве:

Здесь скобки в выражении `(arr > 0).sum()` необходимы, чтобы функция `sum()` применялась к временному результату вычисления `arr > 0`.

```
In [205]: arr = rng.standard_normal(100)

In [206]: (arr > 0).sum() # количество положительных значений
Out[206]: 48

In [207]: (arr <= 0).sum() # количество неположительных значений
Out[207]: 52
```

Эти методы работают и для небулевых массивов, и тогда все отличные от нуля элементы считаются равными True.

Существует еще два метода, `any` и `all`, особенно полезных в случае булевых массивов. Метод `any` проверяет, есть ли в массиве хотя бы одно значение, равное True, а `all` – что все значения в массиве равны True:

```
In [208]: bools = np.array([False, False, True, False])

In [209]: bools.any()
Out[209]: True

In [210]: bools.all()
Out[210]: False
```

СОРТИРОВКА

- Как и встроенные в Python списки, массивы NumPy можно сортировать на месте методом `sort`:

```
In [211]: arr = rng.standard_normal(6)
```

```
In [212]: arr  
Out[212]: array([ 0.0773, -0.6839, -0.7208, 1.1206, -0.0548, -0.0824])
```

```
In [213]: arr.sort()
```

```
In [214]: arr  
Out[214]: array([-0.7208, -0.6839, -0.0824, -0.0548, 0.0773, 1.1206])
```

Любой одномерный участок многомерного массива можно отсортировать на месте, передав методу `sort` номер оси. В этом

```
In [215]: arr = rng.standard_normal((5, 3))
```

```
In [216]: arr  
Out[216]:  
array([[ 0.936 ,  1.2385,  1.2728],  
       [ 0.4059, -0.0503,  0.2893],  
       [ 0.1793,  1.3975,  0.292 ],  
       [ 0.6384, -0.0279,  1.3711],  
       [-2.0528,  0.3805,  0.7554]])
```

Вызов `arr.sort(axis=0)` сортирует значения в каждом столбце, В `arr.sort(axis=1)` — в каждой строке:

```
In [217]: arr.sort(axis=0)
```

```
In [218]: arr  
Out[218]:  
array([[ -2.0528, -0.0503,  0.2893],  
       [ 0.1793, -0.0279,  0.292 ],  
       [ 0.4059,  0.3805,  0.7554],  
       [ 0.6384,  1.2385,  1.2728],  
       [ 0.936 ,  1.3975,  1.3711]])
```

```
In [219]: arr.sort(axis=1)
```

```
In [220]: arr  
Out[220]:  
array([[ -2.0528, -0.0503,  0.2893],  
       [-0.0279,  0.1793,  0.292 ],  
       [ 0.3805,  0.4059,  0.7554],  
       [ 0.6384,  1.2385,  1.2728],  
       [ 0.936 ,  1.3711,  1.3975]])
```

ТЕОРЕТИКО-МНОЖЕСТВЕННЫЕ ОПЕРАЦИИ

Самой употребительной является `numpy.unique`, она

- возвращает отсортированное множество уникальных значений в массиве:

```
In [224]: names = np.array(["Bob", "Will", "Joe", "Bob", "Will", "Joe", "Joe"])
```

```
In [225]: np.unique(names)
```

```
Out[225]: array(['Bob', 'Joe', 'Will'], dtype='<U4')
```

```
In [226]: ints = np.array([3, 3, 3, 2, 2, 1, 1, 4, 4])
```

```
In [227]: np.unique(ints)
```

```
Out[227]: array([1, 2, 3, 4])
```

Функция `numpy.in1d` проверяет, присутствуют ли значения из одного массива в другом, и возвращает булев массив.

```
In [229]: values = np.array([6, 0, 0, 3, 2, 5, 6])
```

```
In [230]: np.in1d(values, [2, 3, 6])
```

```
Out[230]: array([ True, False, False, True, True, False, True])
```

Метод	Описание
<code>unique(x)</code>	Вычисляет отсортированное множество уникальных элементов
<code>intersect1d(x, y)</code>	Вычисляет отсортированное множество элементов, общих для <code>x</code> и <code>y</code>
<code>union1d(x, y)</code>	Вычисляет отсортированное объединение элементов
<code>in1d(x, y)</code>	Вычисляет булев массив, показывающий, какие элементы <code>x</code> встречаются в <code>y</code>
<code>setdiff1d(x, y)</code>	Вычисляет разность множеств, т. е. элементы, принадлежащие <code>x</code> , но не принадлежащие <code>y</code>
<code>setxor1d(x, y)</code>	Симметрическая разность множеств; элементы, принадлежащие одному массиву, но не обоим сразу

ФАЙЛОВЫЙ ВВОД-ВЫВОД МАССИВОВ

numpy.save и **numpy.load** – основные функции для эффективного сохранения и загрузки данных с диска. По умолчанию массивы хранятся в несжатом двоичном формате в файле с расширением **.npy**.

```
In [231]: arr = np.arange(10)
```

```
In [232]: np.save("some_array", arr)
```

Если путь к файлу не заканчивается суффиксом **.npy**, то он будет добавлен. Хранящийся на диске массив можно загрузить в память функцией **numpy.load**:

```
In [233]: np.load("some_array.npy")
Out[233]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

При считывании **npz**-файла мы получаем похожий на словарь объект, который лениво загружает отдельные массивы:

```
In [235]: arch = np.load("array_archive.npz")

In [236]: arch["b"]
Out[236]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Можно сохранить несколько массивов в **zip**-архиве с помощью функции **numpy.savez**, которой массивы передаются в виде именованных аргументов:

```
In [237]: np.savez_compressed("arrays_compressed.npz", a=arr, b=arr)
```

Если данные хорошо сжимаются, то можно вместо этого использовать метод **numpy.savez_compressed**:

```
In [237]: np.savez_compressed("arrays_compressed.npz", a=arr, b=arr)
```

NUMPY НАГЛЯДНО

САМОСТОЯТЕЛЬНО АНАЛИЗИРУЙТЕ в источниках:

Наглядно о том, как работает NumPy:

<https://habr.com/ru/companies/skillfactory/articles/564240/>

Работа с NumPy на примерах:

<https://numpy.org/devdocs/user/quickstart.html>

ПОСТРОЕНИЕ ГРАФИКОВ И ВИЗУАЛИЗАЦИЯ С БИБЛИОТЕКОЙ *MATPLOTLIB*

- ✓ Информативная визуализация (называемая также построением графиков) – одна из важнейших задач анализа данных.
- ✓ Она может быть частью процесса исследования, например применяться для выявления выбросов, определения не обходимых преобразований данных или поиска идей для построения моделей.
- ✓ В других случаях построение интерактивной визуализации для веб-сайта может быть конечной целью.
- ✓ Для Python имеется много дополнительных библиотек статической и динамической визуализации, но ограничимся matplotlib (<http://matplotlib.sourceforge.net>) и надстроенные над ней библиотеки.

ПОСТРОЕНИЕ ГРАФИКОВ И ВИЗУАЛИЗАЦИЯ

С БИБЛИОТЕКОЙ **MATPLOTLIB**

- Matplotlib – это пакет для построения графиков (главным образом двумерных) полиграфического качества. Проект был основан Джоном Хантером в 2002 году с целью реализовать на Python интерфейс, аналогичный MATLAB. Впоследствии сообщества matplotlib и IPython совместно работали над тем, чтобы упростить интерактивное построение графиков из оболочки IPython (а теперь и Jupyter-блокнотов). Matplotlib поддерживает разнообразные графические интерфейсы пользователя во всех операционных системах, а также умеет экспортировать графические данные во всех векторных и растровых форматах: PDF, SVG, IPG, PNG, BMP, GIF и т.д.
- Со временем над matplotlib было создано много дополнительных библиотек визуализации. Одна из них это seaborn (<http://seaborn.pydata.org/>)

НАЧАЛО РАБОТЫ С *MATPLOTLIB*

- Выполнить команды `%matplotlib notebook` в Jupyter (или просто `%matplotlib` в IPython)
- После этого можно создавать фигуры
- Соглашение об им порте:
Если все настроено правильно, то

```
In [13]: import matplotlib.pyplot as plt
```

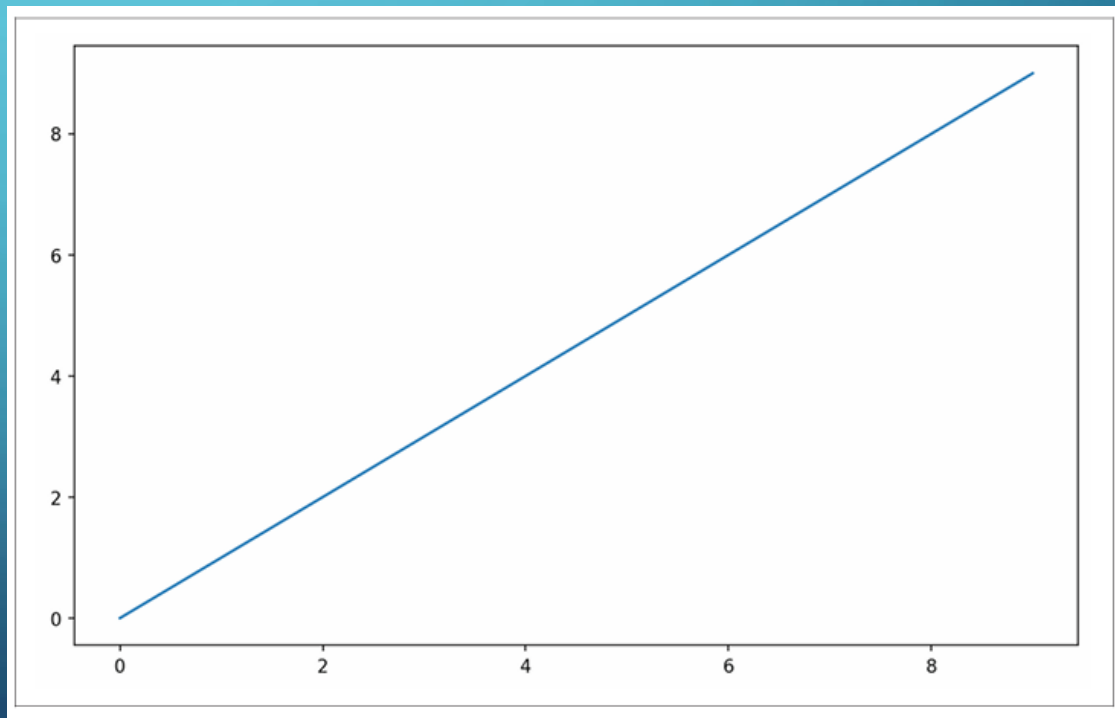
Результат:

```
In [14]: data = np.arange(10)
```

```
In [15]: data
```

```
Out[15]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
In [16]: plt.plot(data)
```



MATPLOTLIB - РИСУНКИ И ПОДГРАФИКИ

Графики в matplotlib «живут» внутри объекта рисунка **Figure**.

Создать новый рисунок методом **plt.figure()**:

```
In [17]: fig = plt.figure()
```

Создать один или несколько подграфиков с помощью метода **add_subplot**:

```
In [18]: ax1 = fig.add_subplot(2, 2, 1)
```

```
In [18]: ax2 = fig.add_subplot(2, 2, 2)
```

```
In [19]: ax3 = fig.add_subplot(2, 2, 3)
```

Это означает, что рисунок будет расчерчен сеткой 2×2, и мы выбираем пер вый из четырех подграфиков (нумерация начинается с 1).

У команды **plt.figure()** имеется ряд параметров:

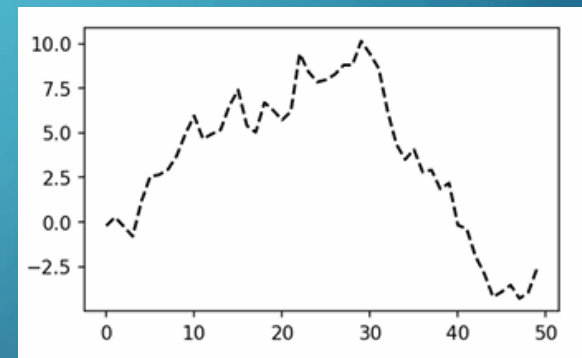
- ✓ **figsize** гарантирует, что при сохранении на диске рисунок будет иметь определенные размер и отношение сторон

Строим график

```
fig = plt.figure()  
ax1 = fig.add_subplot(2, 2, 1)  
ax2 = fig.add_subplot(2, 2, 2)  
ax3 = fig.add_subplot(2, 2, 3)
```

Можно было построить линейный график методом **plot**

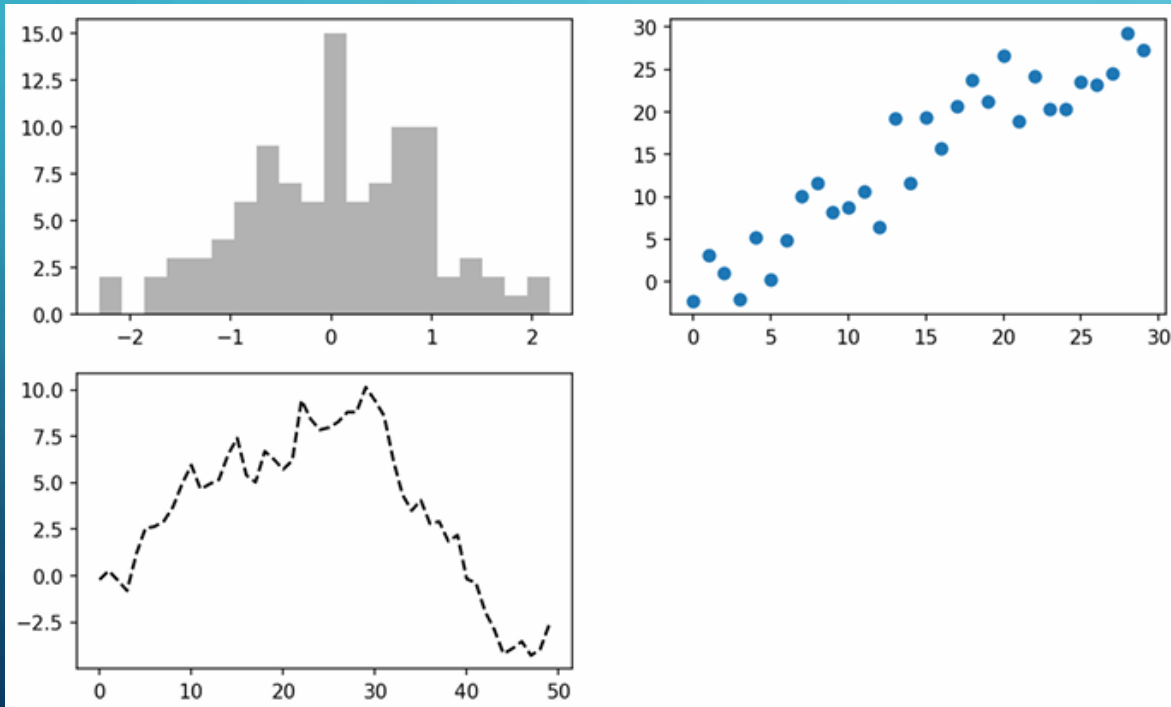
```
In [21]: ax3.plot(np.random.standard_normal(50).cumsum(), color="black",  
.....:          linestyle="dashed")
```



Дополнительные параметры говорят matplotlib, что график должен быть нарисован черной пунктирной линией. Вызов `fig.add_subplot` возвращает объект `AxesSubplot`, который позволяет рисовать в другом пустом подграфике, вызывая его методы экземпляра):

```
In [22]: ax1.hist(np.random.standard_normal(100), bins=20, color="black", alpha=0.3);
```

```
In [23]: ax2.scatter(np.arange(30), np.arange(30) + 3 * np.random.standard_normal(30));
```



Параметр стиля `alpha=0.3` задает степень прозрачности наложенного графика.

**Полный перечень
типов графиков
имеется
в документации по
matplotlib**

ЦВЕТА, МАРКЕРЫ И СТИЛИ ЛИНИЙ

- ✓ Функция рисования линии – `plot` – принимает массивы координат x и y , а также необязательные параметры стили

Нарисовать график зависимости y от x зеленой
штриховой линией, нужно

```
ax.plot(x, y, linestyle="--", color="green")
```

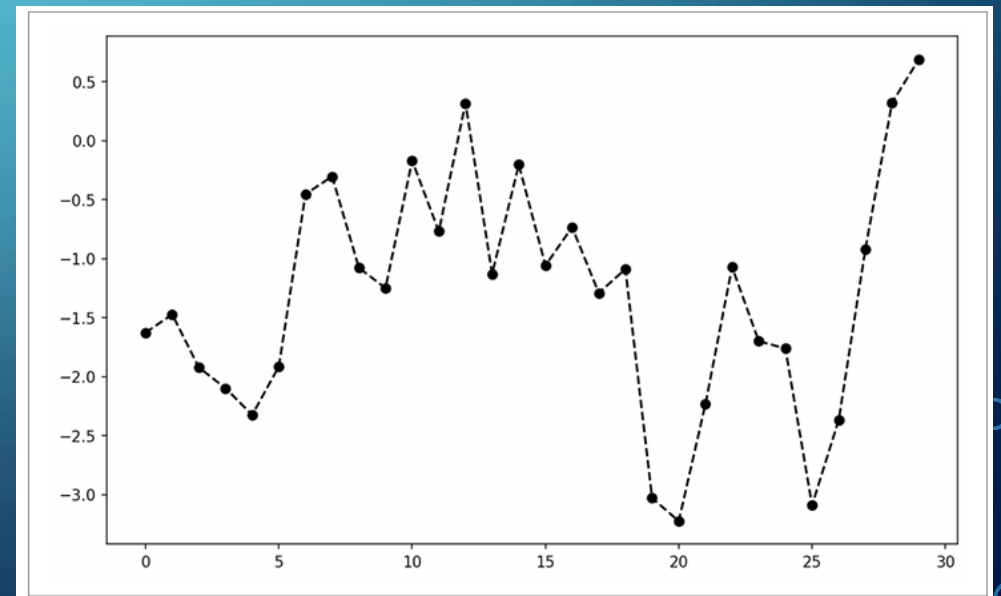
- ✓ Линейные графики могут быть также снабжены маркерами, обозначающими точки, по которым построен график. Поскольку `matplotlib` создает непрерывный линейный график, производя интерполяцию между точками, иногда не ясно, где же находятся исходные точки.

Маркер можно задать в виде дополнительного параметра стиля

```
In [31]: ax = fig.add_subplot()
```

```
In [32]: ax.plot(np.random.standard_normal(30).cumsum(), color="black",  
.....:         linestyle="dashed", marker="o");
```

Для наиболее употребительных цветов предоставляются названия, но вообще-то любой цвет можно представить своим шестнадцатеричным кодом (например, `'#CECECE'`). Некоторые поддерживаемые стили линий перечислены в строке документации для функции `plot` (в IPython или Jupyter введите `plot?`). А полный перечень имеется в онлайн-официальной документации.

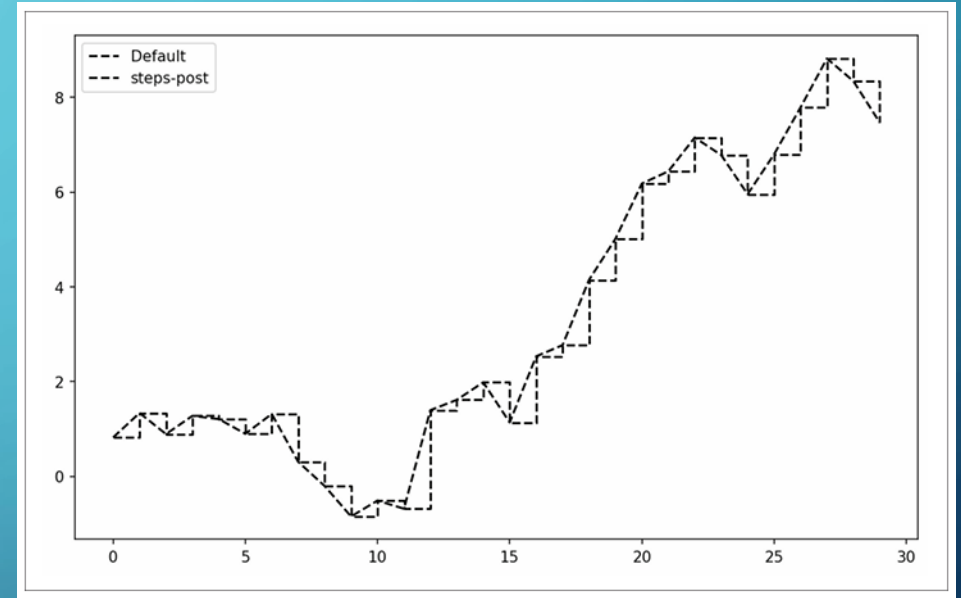


ЦВЕТА, МАРКЕРЫ И СТИЛИ ЛИНИЙ

- ✓ По умолчанию на линейных графиках соседние точки соединяются отрезками прямой, т. е. производится линейная интерполяция.

Параметр **drawstyle** позволяет изменить этот режим:

```
In [34]: fig = plt.figure()
In [35]: ax = fig.add_subplot()
In [36]: data = np.random.standard_normal(30).cumsum()
In [37]: ax.plot(data, color="black", linestyle="dashed", label="Default");
In [38]: ax.plot(data, color="black", linestyle="dashed",
.....:         drawstyle="steps-post", label="steps-post");
In [39]: ax.legend()
```



В данном случае мы передали функции **plot** аргумент **label**, поэтому можем с помощью метода **plt.legend** нанести на график надпись, описывающую каждую линию.

РИСКИ, МЕТКИ И НАДПИСИ

- ✓ Для доступа к большинству средств оформления графиков у объектов осей имеются специальные методы, в т. ч. **xlim**, **xticks** и **xticklabels**. Они управляют размером области, занятой графиком, положением и метками рисок соответственно. Использовать их можно двумя способами:
 - ❖ при вызове без аргументов возвращается текущее значение параметра. Например, метод `plt.xlim()` возвращает текущий диапазон значений по оси X;
 - ❖ при вызове с аргументами устанавливается новое значение параметра. Например, в результате вызова `plt.xlim([0, 10])` диапазон значений по оси X устанавливается от 0 до 10.
- ✓ Все подобные методы действуют на активный или созданный последним объект `AxesSubplot`.

Каждому из них соответствует два метода самого объекта подграфика.

ЗАДАНИЕ НАЗВАНИЯ ГРАФИКА, НАЗВАНИЙ ОСЕЙ, РИСОК И ИХ МЕТОК

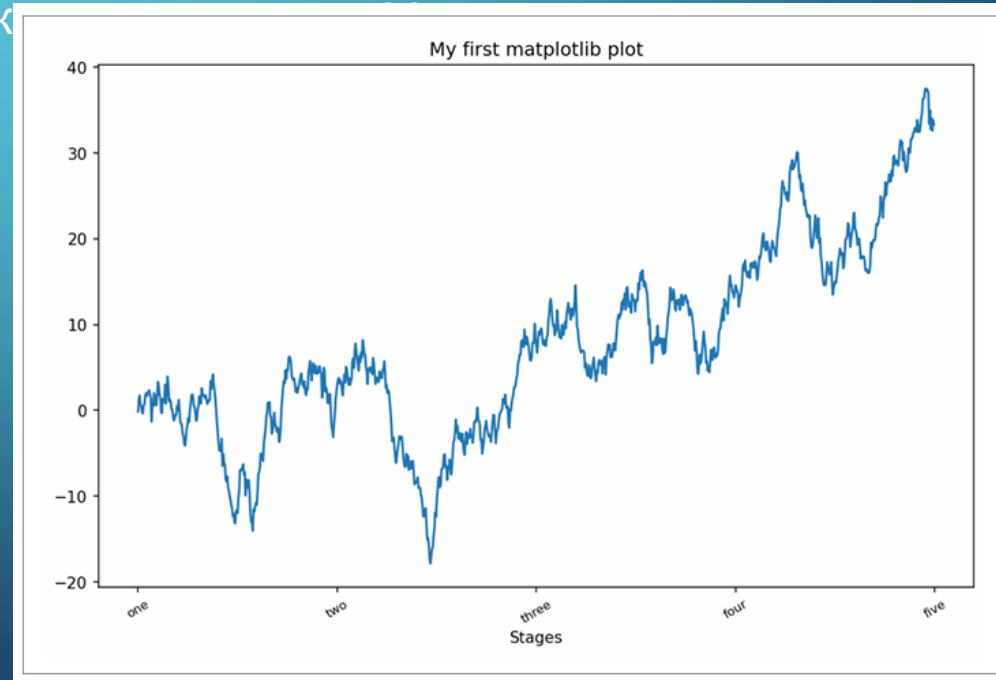
- Для изменения рисок на оси X проще всего воспользоваться методами `set_xticks` и `set_xticklabels`.
- ❖ `set_xticks` ГОВОРИТ `matplotlib`, где в пределах диапазона значений данных ставить риски; по умолчанию их числовые значения изображаются также и в виде меток.
- ❖ `set_xticklabels` - можно задать и другие метки с помощью метода

```
In [42]: ticks = ax.set_xticks([0, 250, 500, 750, 1000])  
  
In [43]: labels = ax.set_xticklabels(["one", "two", "three", "four", "five"],  
.....:                               rotation=30, fontsize=8)
```

- ❖ Аргумент `rotation` устанавливает угол наклона меток рисок к оси x равным 30 градусам.

```
In [44]: ax.set_xlabel("Stages")  
  
Out[44]: Text(0.5, 6.6666666666666652, 'Stages')  
In [45]: ax.set_title("My first matplotlib plot")
```

- ❖ Модификация оси y производится точно так же с заменой x на y. В классе оси имеется метод `set`, позволяющий задавать сразу несколько свойств графика.



```
ax.set(title="My first matplotlib plot", xlabel="Stages")
```

ДОБАВЛЕНИЕ ПОЯСНИТЕЛЬНЫХ НАДПИСЕЙ

- ✓ Пояснительная надпись – еще один важный элемент оформления графика. Добавить ее можно двумя способами. Проще всего передать аргумент `label` при добавлении каждого нового графика:

```
In [46]: fig, ax = plt.subplots()
```

```
In [47]: ax.plot(np.random.randn(1000).cumsum(), color="black", label="one");
```

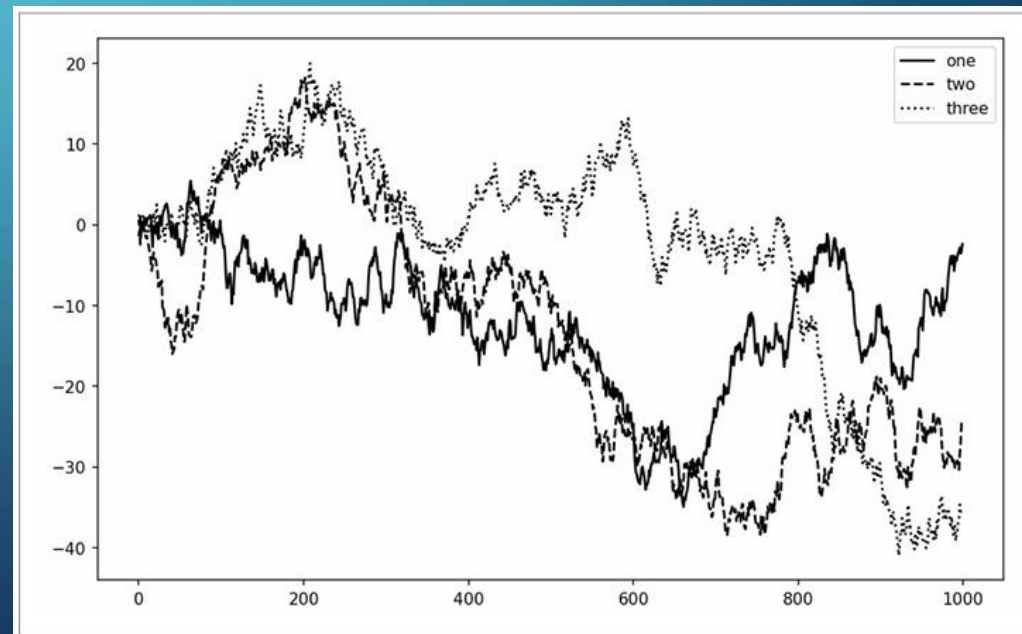
```
In [48]: ax.plot(np.random.randn(1000).cumsum(), color="black", linestyle="dashed",  
.....:         label="two");
```

```
In [49]: ax.plot(np.random.randn(1000).cumsum(), color="black", linestyle="dotted",  
.....:         label="three");
```

- ✓ После этого можно вызвать метод `ax.legend()`, и он автоматически создаст пояснительную надпись.

```
In [50]: ax.legend()
```

Аргумент `loc` говорит, где поместить надпись. По умолчанию подразумевается значение `'best'`, в этом случае место выбирается так, чтобы по возможности не загромождать сам график. Чтобы исключить из надписи один или несколько элементов, не задавайте параметр `label` вовсе или задайте `label='_nolegend_'`.



СОХРАНЕНИЕ ГРАФИКОВ В ФАЙЛЕ

- ✓ Активный рисунок можно сохранить в файле методом экземпляра рисунка `savefig`. Например, чтобы сохранить рисунок в формате SVG, достаточно указать только имя файла:

```
fig.savefig("figpath.svg")
```

Формат выводится из расширения имени файла. Если бы мы задали файл с расширением `.pdf`, то рисунок был бы сохранен в формате PDF. При публикации графики я часто использую параметр `dpi` – разрешение в точках на дюйм. Чтобы получить тот же самый график в формате PNG с разрешением 400 DPI, нужно было бы написать:

```
fig.savefig("figpath.png", dpi=400)
```

ПОСТРОЕНИЕ ГРАФИКОВ

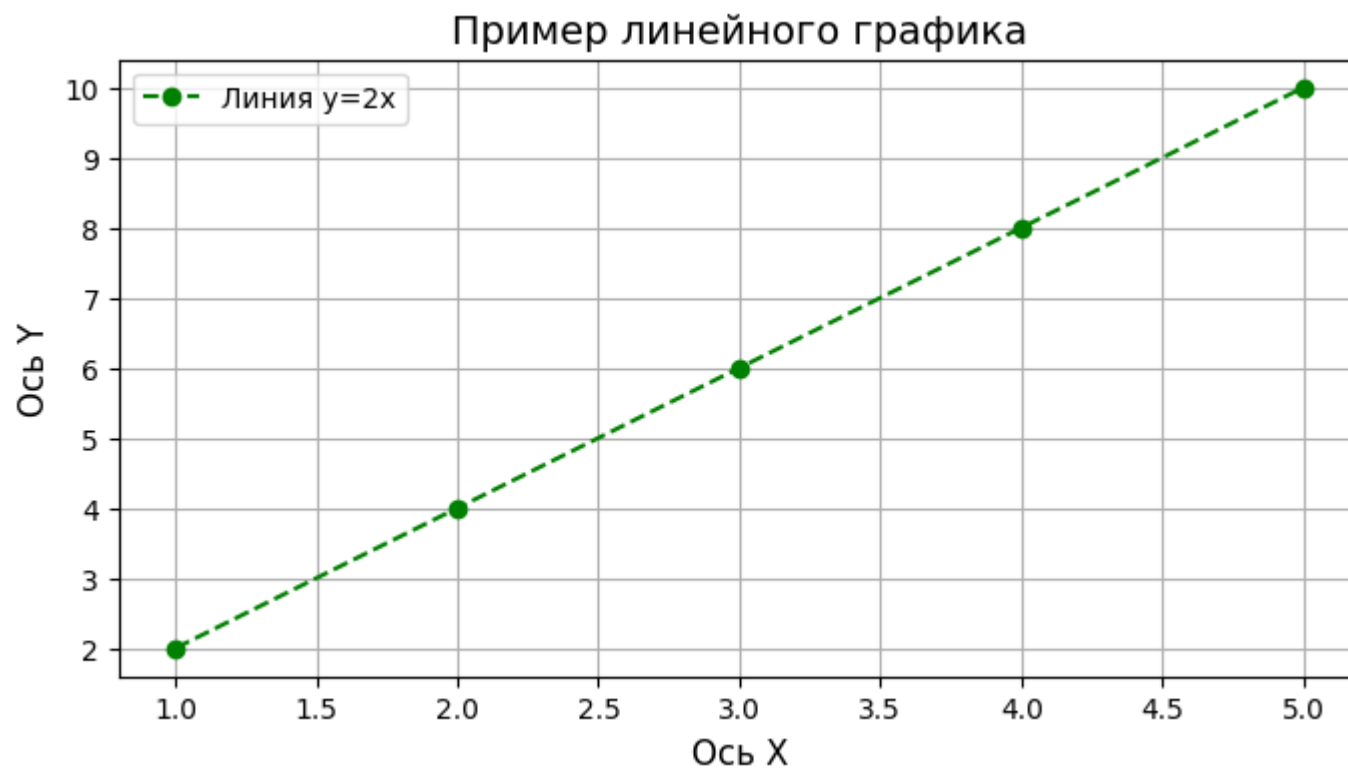
Простой линейный график

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

# Данные
x = [1, 2, 3, 4, 5]
y = [2, 4, 6, 8, 10]

# Создание графика
plt.figure(figsize=(8, 4)) # Размер графика
plt.plot(x, y,
         marker='o',      # Маркеры для точек
         linestyle='--',  # Стиль линии
         color='green',    # Цвет
         label='Линия y=2x') # Легенда

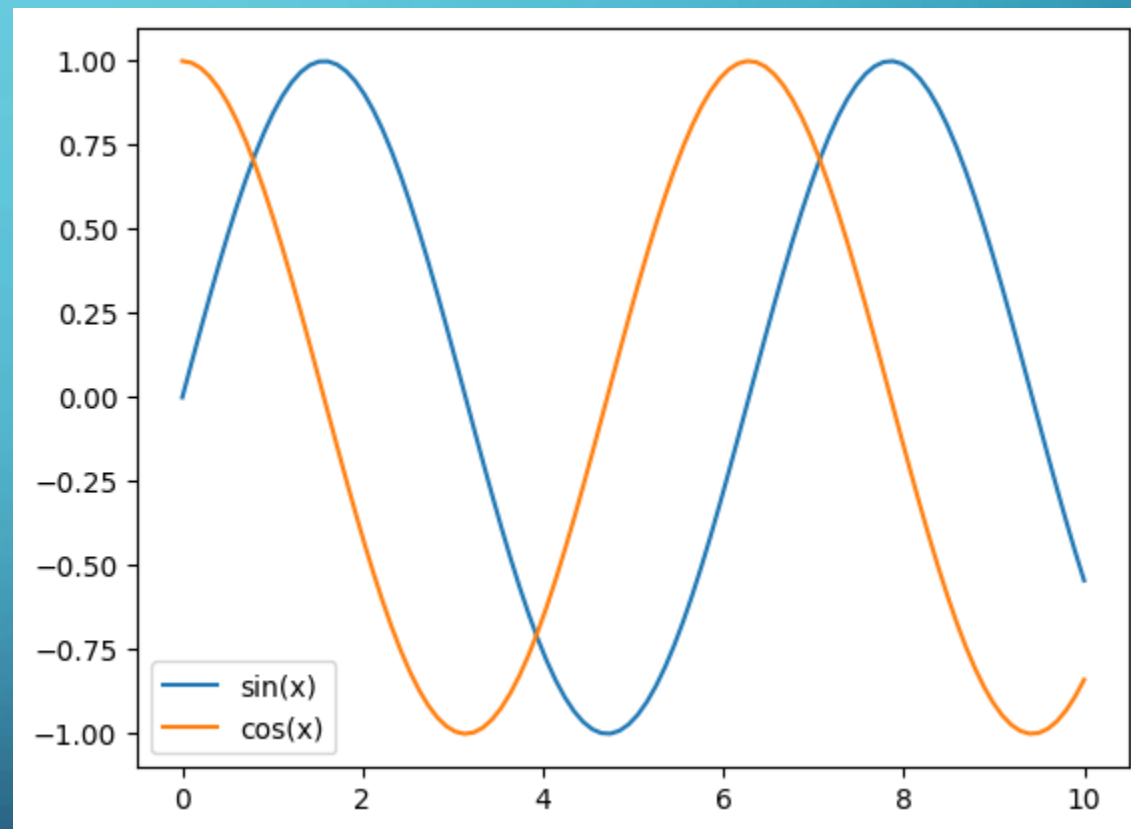
# Настройки
plt.title('Пример линейного графика', fontsize=14)
plt.xlabel('Ось X', fontsize=12)
plt.ylabel('Ось Y', fontsize=12)
plt.grid(True)           # Сетка
plt.legend()              # Отображение легенды
plt.show()
```



ПОСТРОЕНИЕ ГРАФИКОВ

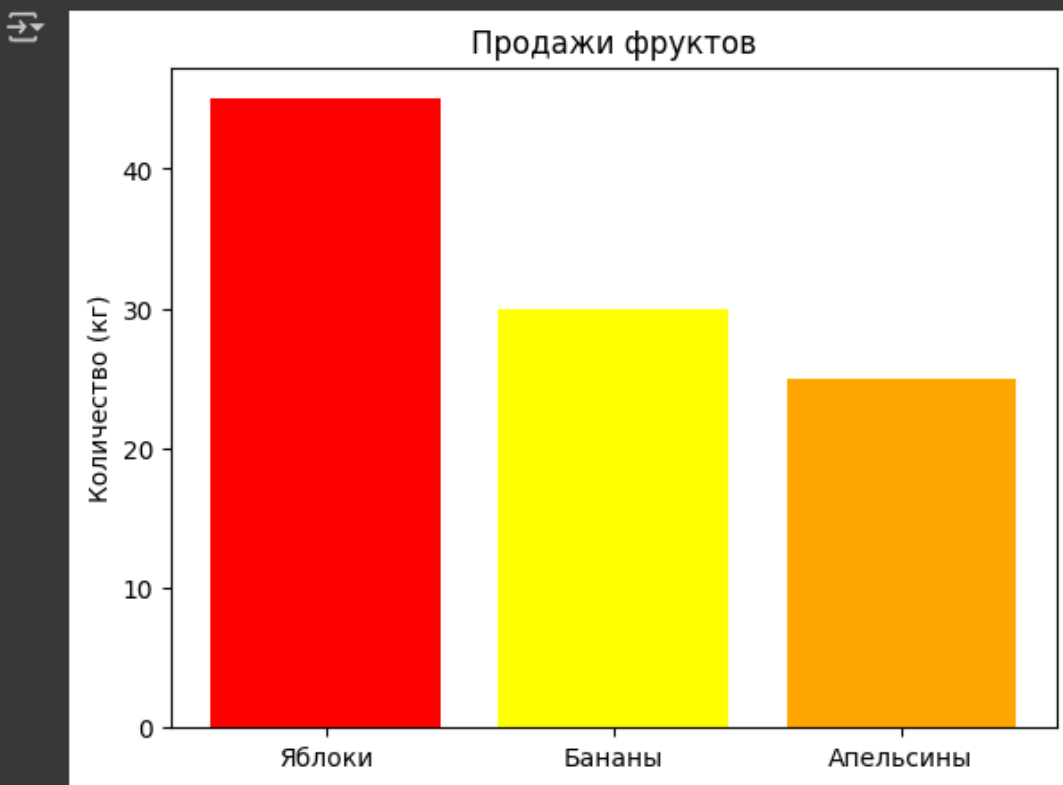
Синусоида и косинусоида

```
x = np.linspace(0, 10, 100) # 100 точек от 0 до 10  
y1 = np.sin(x)  
y2 = np.cos(x)  
  
plt.plot(x, y1, label='sin(x)')  
plt.plot(x, y2, label='cos(x)')  
plt.legend()  
plt.show()
```

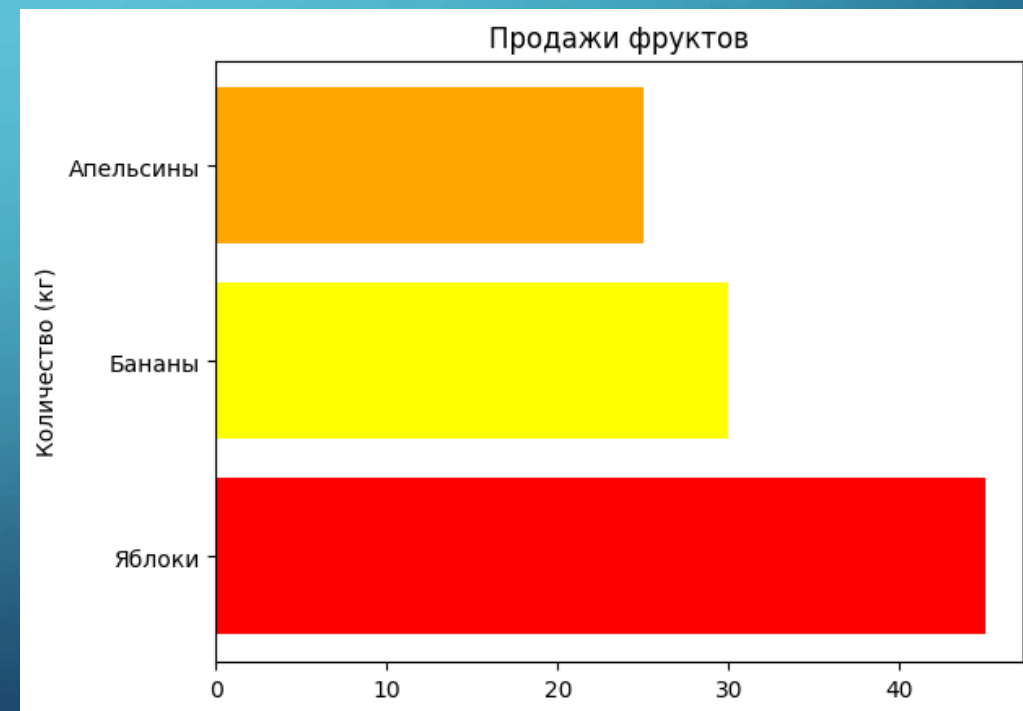


СТОЛБЧАТЫЕ ДИАГРАММЫ

```
categories = ['Яблоки', 'Бананы', 'Апельсины']  
values = [45, 30, 25]  
  
plt.bar(categories, values, color=['red', 'yellow', 'orange'])  
plt.title('Продажи фруктов')  
plt.ylabel('Количество (кг)')  
plt.show()
```



Изменение `plt.bar` на `plt.barh` меняет отображение диаграмм с вертикального на горизонтальное (h – horizontal).



КРУГОВЫЕ ДИАГРАММЫ

```
▶ sizes = [45, 30, 25]
labels = ['Яблоки', 'Бананы', 'Апельсины']
colors = ['#ff9999', '#66b3ff', '#99ff99']

plt.pie(sizes,
        labels=labels,
        colors=colors,
        autopct='%1.1f%%', # Проценты с 1 знаком после запятой
        startangle=90)    # Поворот на 90 градусов

plt.axis('equal') # Чтобы диаграмма была круглой
plt.title('Доля фруктов в продажах')
plt.show()
```

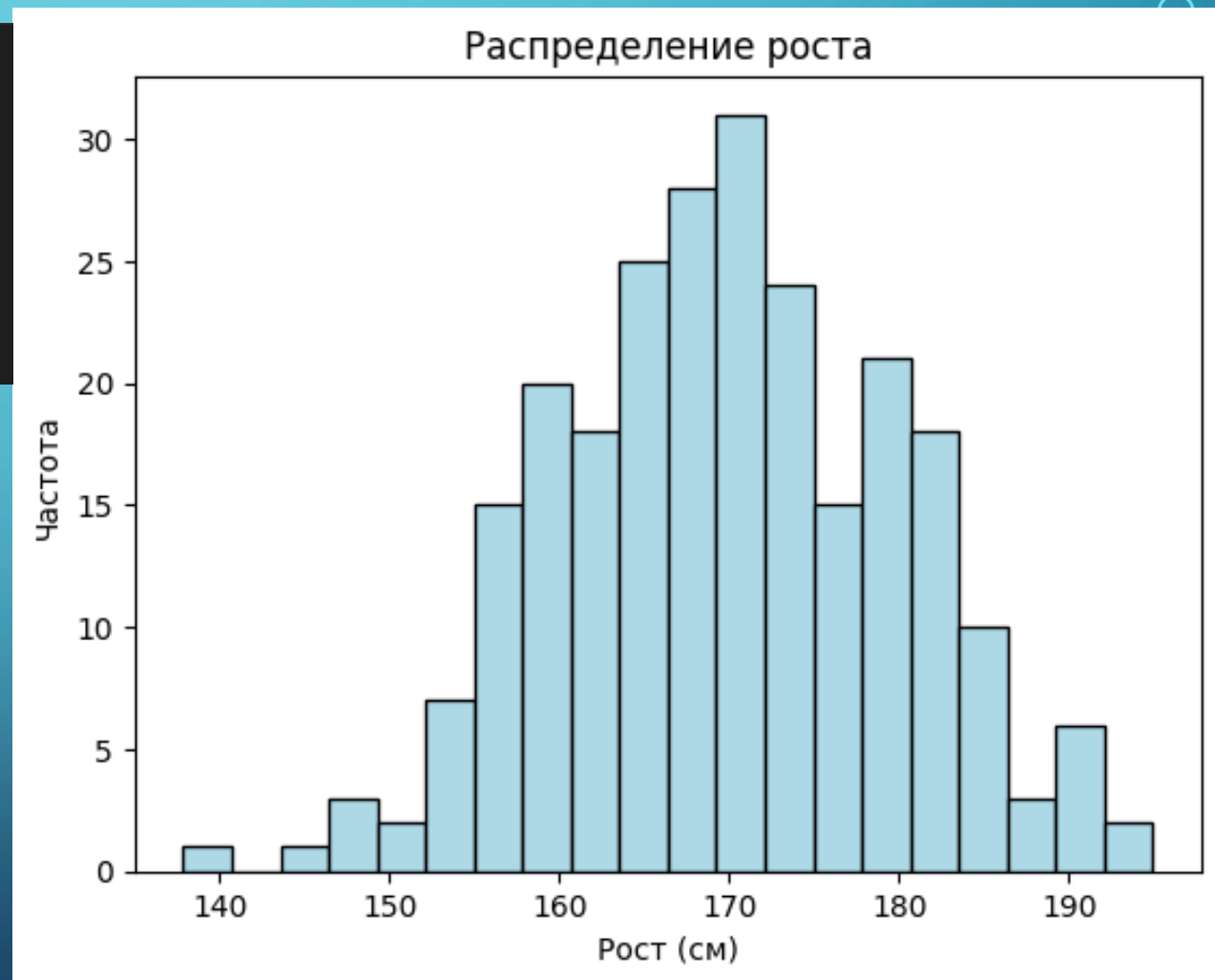


ГИСТОГРАММЫ

```
data = np.random.normal(170, 10, 250) # 250 значений с средним=170, std=10

plt.hist(data,
          bins=20,          # Количество столбцов
          edgecolor='black', # Границы столбцов
          color='lightblue')

plt.title('Распределение роста')
plt.xlabel('Рост (см)')
plt.ylabel('Частота')
plt.show()
```



КОМБИНИРОВАННЫЕ ГРАФИКИ

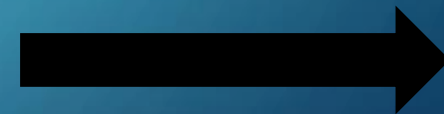
```
# Данные
x = np.linspace(0, 10, 100)
y1 = x
y2 = x**2

# Создание subplot (1 строка, 2 столбца)
plt.figure(figsize=(12, 4))

# Первый график
plt.subplot(1, 2, 1)
plt.plot(x, y1, color='red')
plt.title('Линейный график')

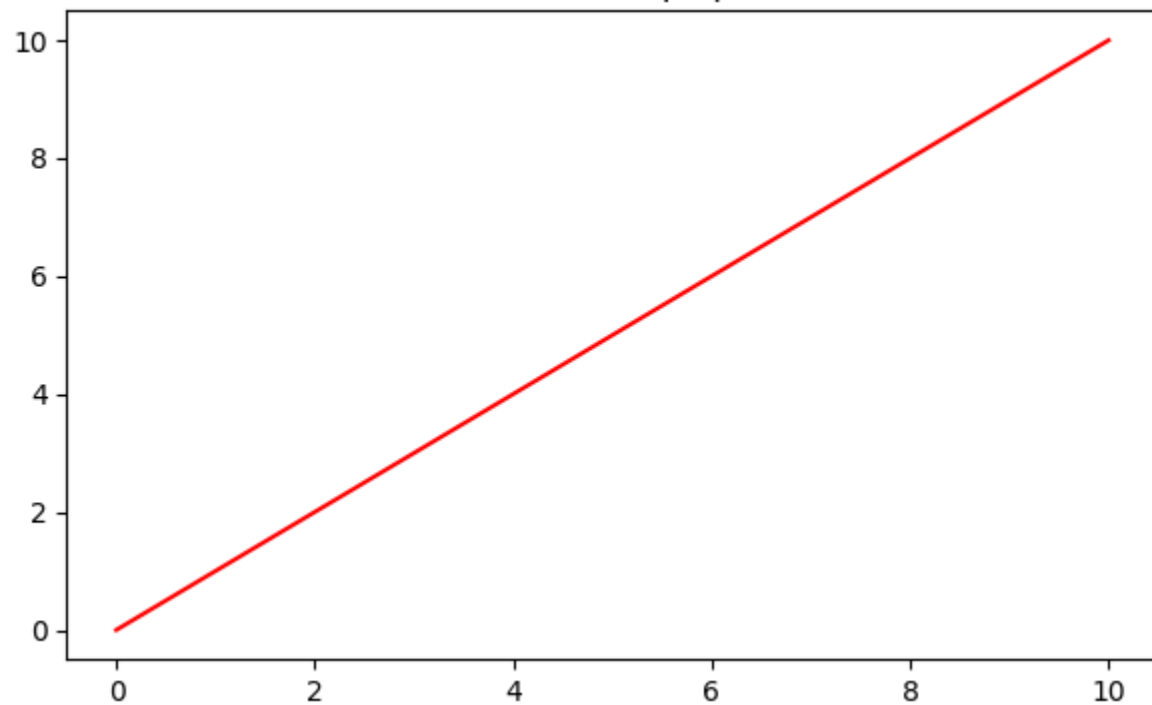
# Второй график
plt.subplot(1, 2, 2)
plt.plot(x, y2, color='blue')
plt.title('Квадратичная функция')

plt.tight_layout() # Автоматическое выравнивание
plt.show()
```



КОМБИНИРОВАННЫЕ ГРАФИКИ

Линейный график



Квадратичная функция

