

Библиотека matplotlib (Часть II)



§ 10 Примеры построения 3D графиков

10.1 Создание трехмерных осей

Для того, чтобы нарисовать трехмерный график, **в первую очередь надо создать трехмерные оси**. Чтобы создать трехмерные оси, сначала мы создаем с помощью функции `figure()` экземпляр класса `matplotlib.figure.Figure`. Затем у экземпляра этого класса вызывается метод `add_subplot()`, который создает оси координат и может принимать различное количество параметров, но в данном случае мы передаем один именованный параметр – `projection`, который описывает тип осей. Для создания трехмерных осей значение параметра `projection` должно быть строкой "3d".

Ввод [5]:

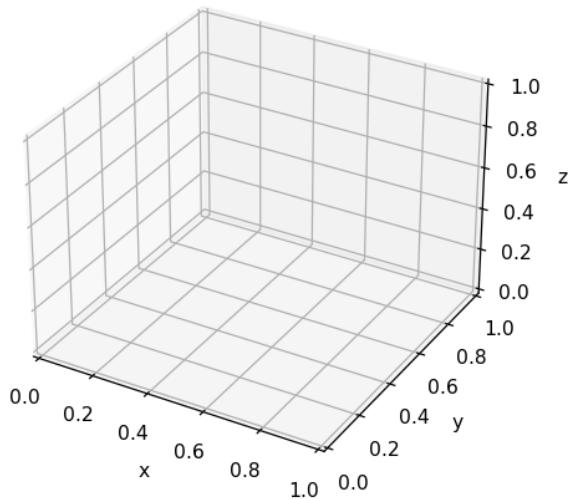
```
# создание трехмерных осей
import matplotlib.pyplot as plt
import numpy as np
%matplotlib notebook

fig = plt.figure()
axes = fig.add_subplot(projection='3d')

axes.set_xlabel('x')
axes.set_ylabel('y')
axes.set_zlabel('z')

plt.show()
```

<IPython.core.display.Javascript object>



При использовании варианта отображения графика `%matplotlib notebook` полученные оси можно вращать мышкой. При использовании `%matplotlib inline` трёхмерную картинку нельзя будет воротить мышкой (можно будет задать угол обзора).

10.2 Использование `numpy.meshgrid()` для формирования регулярной сетки

Теперь надо подготовить данные для рисования. Здесь понадобятся **три** двумерные матрицы: `X`, `Y`, `Z`. Матрицы `X` и `Y` будут хранить координаты сетки точек, в которых будет вычисляться строящаяся функция, а матрица `Z` будет хранить значения этой функции в соответствующей точке.

Для создания на плоскости XY прямоугольной сетки можно воспользоваться функцией `numpy.meshgrid()`. Эта функция, в простейшем случае, принимает несколько одномерных массивов. Нам нужно два массива для осей X и Y , которые содержат значения координат узлов вдоль соответствующей оси, и могут иметь разный размер. Эта функция возвратит в нашем случае две двумерные матрицы, описывающие координаты X и Y на двумерной сетке. Первая матрица будет создана размножением первого переданного параметра в функцию `numpy.meshgrid()` вдоль строк первой возвращаемой матрицы, а вторая матрица создается размножением второго переданного одномерного массива вдоль столбцов второй возвращаемой матрицы.

Такие двумерные матрицы позволяют очень легко рассчитывать значения функций от двух аргументов, используя поэлементные операции (**векторизация**). И кроме того, эти же матрицы требуются для рисования трехмерных поверхностей в *matplotlib*.

Ввод [18]:

```
# Распаковка координатных сеток в отдельные массивы
import numpy as np

x = [1, 2, 3]
y = [2, 5, 6, 8]
X, Y = np.meshgrid(x, y)
print(X)
print(Y)
```

```
[[1 2 3]
 [1 2 3]
 [1 2 3]
 [1 2 3]]
 [[2 2 2]
 [5 5 5]
 [6 6 6]
 [8 8 8]]
```

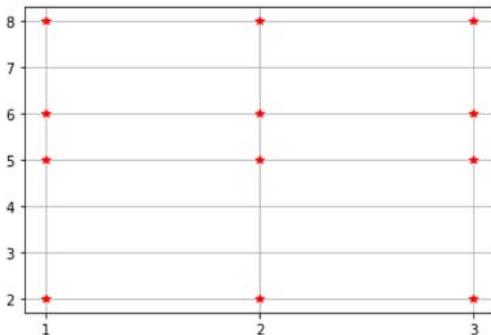
Для большего понимания визуализируем в *matplotlib* результаты на сетке, полученные после запуска функции `np.meshgrid()`

Ввод [27]:

```
import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline

x = [1, 2, 3]
y = [2, 5, 6, 8]
X, Y = np.meshgrid(x, y)

plt.figure()
plt.plot(X, Y, marker='*', color='red', linestyle='none')
plt.xticks(x)
plt.grid()
plt.show()
```



Ввод [17]:

```
# Список координатных сеток
import numpy as np

x = [1, 2, 3]
y = [2, 5, 6, 8]
XY = np.meshgrid(x, y)
print(XY)
```

```
[array([[1, 2, 3],
 [1, 2, 3],
 [1, 2, 3],
 [1, 2, 3]]), array([[2, 2, 2],
 [5, 5, 5],
 [6, 6, 6],
 [8, 8, 8]])]
```

10.3 Построение 3D поверхности через параметр `projection` функции `add_subplot()`

После подготовки прямоугольной сетки на плоскости XY необходимо рассчитать значения отображаемой функции (значение по оси Z) в ее узлах.

$$Z = f(X, Y)$$

Для отображения 3D-поверхности можно воспользоваться функцией `plot_surface()`, в который передадим полученные матрицы X , Y , Z .

Пример. Для примера рассмотрим построение графика **функции Химмельблау**:

$$f(x, y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2$$

определенную на отрезке $X \in [-5; 5]$, $Y \in [-5; 5]$.

Ввод [9]:

```
# создание трехмерных осей
import matplotlib.pyplot as plt
import numpy as np
%matplotlib notebook

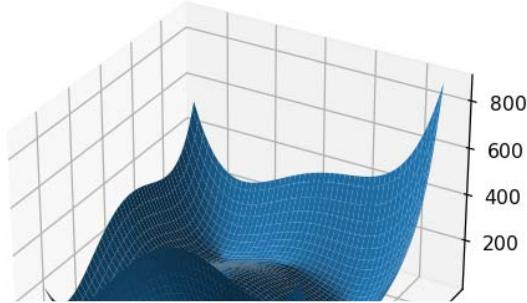
# Строим сетку в интервале от -5 до 5, имеющую 100 отсчетов по обоим координатам
x = np.linspace (-5, 5, 100)
y = np.linspace (-5, 5, 100)

# Создаем двумерную матрицу-сетку
X, Y = np.meshgrid(x, y)

# В узлах рассчитываем значение функции
Z = (X**2 + Y - 11)**2 + (X + Y**2 - 7)**2

fig = plt.figure()
axes = fig.add_subplot(projection='3d')
axes.plot_surface(X, Y, Z)
plt.show()
```

<IPython.core.display.Javascript object>



При использовании магической команды `%matplotlib notebook` полученную поверхность можно вращать с помощью мышки. *Matplotlib* не использует графический ускоритель, поэтому вращение происходит довольно медленно, хотя скорость зависит от количества точек на поверхности.

Следующие параметры функции `plot_surface()` можно использовать для настройки внешнего вида 3D-поверхности:

Параметр	Описание
<code>x</code> , <code>y</code> , <code>z</code>	2D массивы для построения трехмерных графиков
<code>rcount</code> , <code>ccount</code>	Максимальное число элементов каркаса по координатам <code>x</code> и <code>y</code> (по умолчанию 50)
<code>rstride</code> , <code>cstride</code>	Величина шага, с которым будут выбираться элементы из массивов <code>x</code> , <code>y</code> (параметры <code>rstride</code> , <code>cstride</code> и <code>rcount</code> , <code>ccount</code> – взаимоисключающие)
<code>color</code>	Цвет графика. Цвет можно определить английским словом для соответствующего цвета или одной буквой: { b , g , r , c , m , y , k , w } или { blue , green , red , cyan , magenta , yellow , black , white }. Можно задавать цвет так, как это принято в HTML после символа <code>color="#11aa55"</code>
<code>cmap</code>	Цветовая карта графика
<code>edgecolors</code>	Цвет линий
<code>linewidths</code>	Толщина линий

Пояснения.

- Параметры `rcount` и `ccount` задают количество отсчетов по осям, а `rstride` и `cstride` задают степень прореженности (децимации) по осям, то есть сколько отсчетом надо пропустить. Если мы отключим прореживание, установив параметры `rstride = cstride = 1`, то сетка на поверхности станет более мелкой.
- В параметре `cmap` указывается название цветовой карты, готовые варианты которых располагаются в модуле `matplotlib.cm`. Чтобы узнать какие цветовые карты существуют, можно воспользоваться следующим скриптом:

```
from matplotlib import cm
print(dir(cm))
```

Как эти цветовые карты выглядят, можно увидеть на этой странице документации. <https://matplotlib.org/stable/tutorials/colors/colormaps.html#miscellaneous>
<https://matplotlib.org/stable/tutorials/colors/colormaps.html#miscellaneous>

Для улучшения читаемости кода разделим процессы подготовки данных и рисования. Для этого создание сетки и расчет функции выделим в отдельную функцию `make_data()`, а сам график отформатируем с использованием рассмотренных параметров функции `plot_surface()`:

Ввод [1]:

```
# создание трехмерных осей
import matplotlib.pyplot as plt
import numpy as np

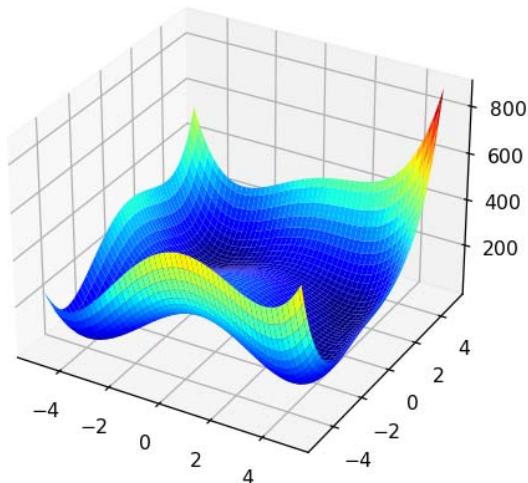
%matplotlib notebook

def make_data():
    # Строим сетку
    x = np.linspace (-5, 5, 100)
    y = np.linspace (-5, 5, 100)

    # Создаем матрицу-сетку и в ее узлах рассчитываем значение функции
    X, Y = np.meshgrid(x, y)
    Z = (X**2 + Y - 11)**2 + (X + Y**2 - 7)**2
    return X, Y, Z

if __name__ == '__main__':
    x, y, z = make_data()
    fig = plt.figure()
    axes = fig.add_subplot(projection='3d')
    axes.plot_surface(x, y, z, cmap='jet')
    plt.show()
```

<IPython.core.display.Javascript object>



Следующий пример строит в одном окне два трехмерных графика, отформатированные различными цветовыми картами.

Ввод [2]:

```

import matplotlib.pyplot as plt
import numpy as np

def make_bill():
    """
    # Функция Била
    """

    # Строим сетку
    x = np.arange(-4.5, 4.5, 0.25)
    y = np.arange(-4.5, 4.5, 0.25)

    # Создаем матрицу-сетку и в ее узлах рассчитываем значение функции
    X, Y = np.meshgrid(x, y)
    Z = (1.5 - X + X*Y)**2 + (2.25 - X + X*Y**2)**2 + (2.625 - X + X*Y**3)**2
    return X, Y, Z

def make_saddle():
    """
    # Функция седла
    """

    # Строим сетку
    x = np.linspace(-10, 10, 100)
    y = np.linspace(-10, 10, 100)

    # Создаем матрицу-сетку и в ее узлах рассчитываем значение функции
    X, Y = np.meshgrid(x, y)
    Z = X ** 2 - Y ** 2
    return X, Y, Z

if __name__ == '__main__':
    fig = plt.figure(figsize=(12, 10))
    axes_1 = fig.add_subplot(1, 2, 1, projection='3d')
    axes_2 = fig.add_subplot(1, 2, 2, projection='3d')

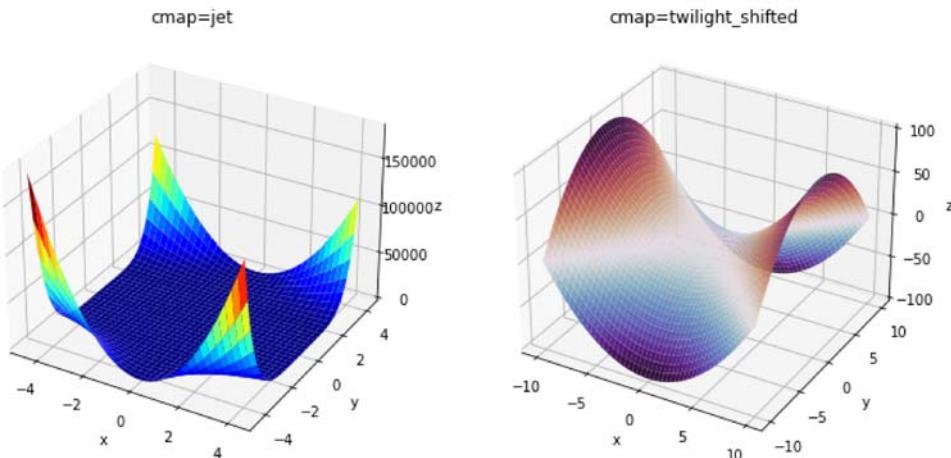
    # Функция Била
    x, y, z = make_bill()
    axes_1.plot_surface(x, y, z, cmap='jet')
    axes_1.set_title('cmap=jet')
    axes_1.set_zticks(range(0, 200_000, 50_000))

    # Функция седла
    x, y, z = make_saddle()
    axes_2.plot_surface(x, y, z, cmap='twilight_shifted')
    axes_2.set_title('cmap=twilight_shifted')
    axes_2.set_xticks(range(-10, 11, 5))
    axes_2.set_yticks(range(-10, 11, 5))
    axes_2.set_zticks(range(-100, 101, 50))

    # Добавим подписи осей
    for ax in [axes_1, axes_2]:
        ax.set_xlabel('x')
        ax.set_ylabel('y')
        ax.set_zlabel('z')

    plt.show()

```



Рассмотрим создание собственной цветовой карты на примере.

Пример использования класса `LinearSegmentedColormap` (производного от `Colormap`) для создания градиента перехода от синего цвета к красному через белый.

Ввод [6]:

```

import matplotlib.pyplot as plt
import numpy as np
from matplotlib.colors import LinearSegmentedColormap

%matplotlib notebook

def make_data():
    # Строим сетку
    x = np.linspace (-10, 10, 100)
    y = np.linspace (-10, 10, 100)

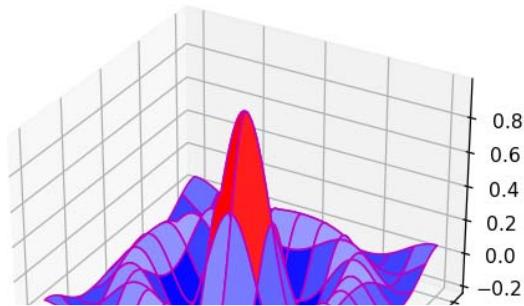
    # Создаем матрицу-сетку и в ее узлах рассчитываем значение функции
    X, Y = np.meshgrid(x, y)
    # Z = np.sin(X) * np.sin(Y) / (X * Y)
    Z = np.sin(np.sqrt(X ** 2 + Y ** 2)) / np.sqrt(X ** 2 + Y ** 2)

    return X, Y, Z

if __name__ == '__main__':
    x, y, z = make_data()
    fig = plt.figure()
    axes = fig.add_subplot(projection='3d')
    cmap = LinearSegmentedColormap.from_list('my_color', ['b', 'w', 'r'], 256)
    axes.plot_surface(x, y, z, rstride=10, cstride=10, cmap=cmap, edgecolors='m')
    plt.show()

<IPython.core.display.Javascript object>

```



Здесь для создания цветовой карты используется статический метод `from_list()`, который принимает три параметра:

- имя создаваемой карты;
- список цветов, начиная с цвета для минимального значения на графике (голубой - `b`), через промежуточные цвета (у нас это белый - `w`) к цвету для максимального значения функции (красный - `r`);
- количество цветовых переходов. Чем это число больше, тем более плавный градиент, но тем больше памяти он занимает.

Для построения каркасной поверхности в 3D используется функция `plot_wireframe()`. Её параметры аналогичны функции `plot_surface()`.

Рассмотрим построение каркасной поверхности на примере.

Ввод [7]:

```
# создание трехмерных осей
import matplotlib.pyplot as plt
import numpy as np

%matplotlib notebook

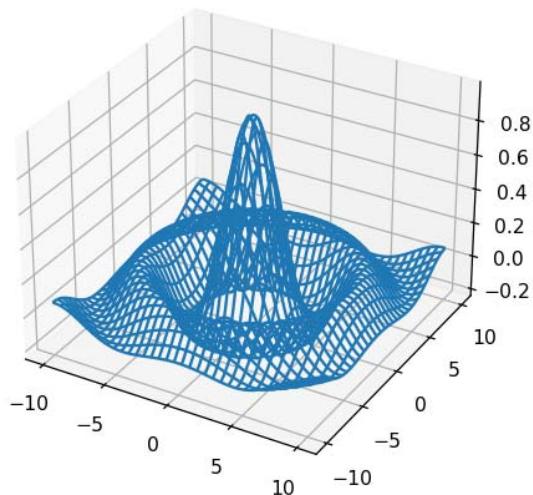
def make_data():
    # Строим сетьку
    x = np.linspace (-10, 10, 100)
    y = np.linspace (-10, 10, 100)

    # Создаем матрицу-сетку и в ее узлах рассчитываем значение функции
    X, Y = np.meshgrid(x, y)
    Z = np.sin(X) * np.sin(Y) / (X * Y)
    Z = np.sin(np.sqrt(X ** 2 + Y ** 2)) / np.sqrt(X ** 2 + Y ** 2)

    return X, Y, Z

if __name__ == '__main__':
    x, y, z = make_data()
    fig = plt.figure()
    axes = fig.add_subplot(projection='3d')
    axes.plot_wireframe(x, y, z, rstride=3, cstride=3)
    plt.show();
```

<IPython.core.display.Javascript object>



Метод `fig.colorbar()` рисует справа от графика цветовую полосу (палитру). Ее аргумент `shrink` задает коэффициент сжатия (по высоте) полосы относительно рисунка, а аргумент `aspect` определяет пропорцию высоты и ширины полосы. В примере ниже опция `cmap=rrp1.cm.hsv` задает `hsv` способ раскраски в зависимости от `z` координаты точек поверхности.

Ввод [1]:

```

import matplotlib.pyplot as plt
import matplotlib as mpl
import numpy as np

%matplotlib notebook

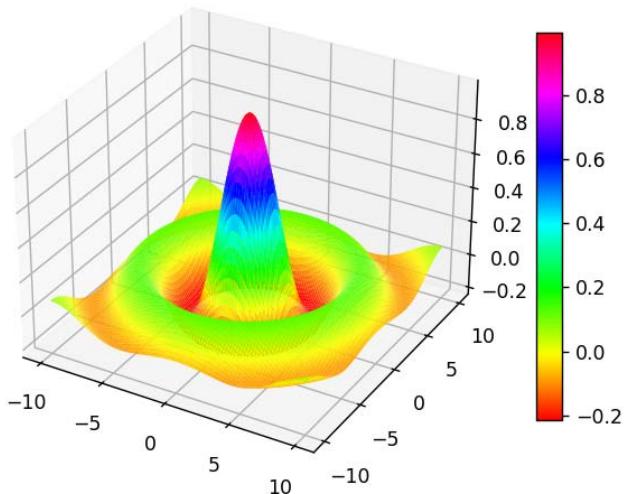
def make_data():
    # Строим сетьку
    x = np.linspace (-10, 10, 100)
    y = np.linspace (-10, 10, 100)

    # Создаем матрицу-сетку и в ее узлах рассчитываем значение функции
    X, Y = np.meshgrid(x, y)
    # Z = np.sin(X) * np.sin(Y) / (X * Y)
    Z = np.sin(np.sqrt(X ** 2 + Y ** 2)) / np.sqrt(X ** 2 + Y ** 2)
    return X, Y, Z

if __name__ == '__main__':
    x, y, z = make_data()
    fig = plt.figure()
    axes = fig.add_subplot(projection='3d')
    surf = axes.plot_surface(x, y, z, rstride=1, cstride=1, linewidth=0, cmap=mpl.cm.hsv)
    fig.colorbar(surf, shrink=0.75, aspect=15)
    plt.show();

```

<IPython.core.display.Javascript object>



Можно строить трехмерные графики **на основе набора отдельных точек** с помощью функции `scatter()`. Рассмотрим пример такого построения.

Ввод [8]:

```
# создание трехмерных осей
import matplotlib.pyplot as plt
import numpy as np

%matplotlib notebook

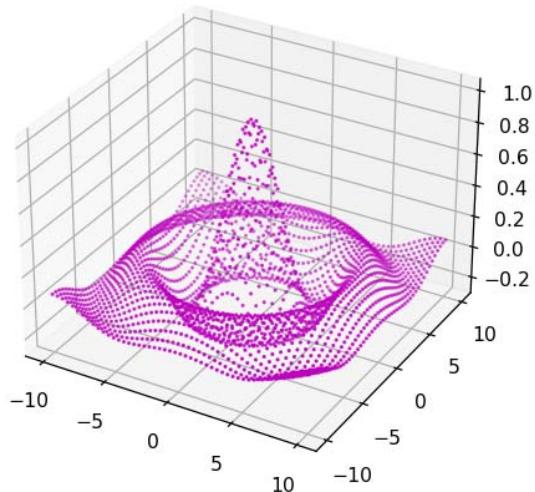
def make_data():
    # Строим сетку
    x = np.linspace (-10, 10, 50)
    y = np.linspace (-10, 10, 50)

    # Создаем матрицу-сетку и в ее узлах рассчитываем значение функции
    X, Y = np.meshgrid(x, y)
    Z = np.sin(X) * np.sin(Y) / (X * Y)
    Z = np.sin(np.sqrt(X ** 2 + Y ** 2)) / np.sqrt(X ** 2 + Y ** 2)

    return X, Y, Z

if __name__ == '__main__':
    x, y, z = make_data()
    fig = plt.figure()
    axes = fig.add_subplot(projection='3d')
    axes.scatter(x, y, z, s=1, color='m')
    plt.show();
```

<IPython.core.display.Javascript object>



10.4 Построение 3D поверхности с помощью класса Axes3D

Рассмотрим еще один способ построения трехмерных графиков. Все дополнительные классы для работы в 3D находятся в модуле `mpl_toolkits.mplot3d`. Его необходимо импортировать вместе с `numpy` и `matplotlib.pyplot`.

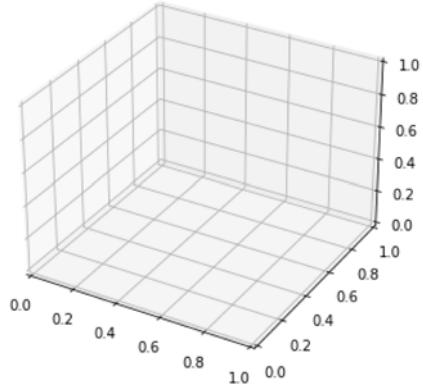
Далее (аналогично предыдущему) создадим фигуру и трехмерную ось.

Ввод [4]:

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure(figsize=(7, 4))
ax_3D = Axes3D(fig)

plt.show()
```



Здесь трехмерную систему координат мы создаём с помощью класса `Axes3D`, методом `plt.show()` её отображаем.

Ранее подобный результат мы получали строкой вида: `axes = fig.add_subplot(projection='3d')`. Как именно строить трехмерные системы координат зависит от выбора разработчика и удобства написания программ.

После создания трехмерной системы координат, мы можем в них строить двумерные и трехмерные графики:

- `plot()` - линейный 2D график в трех измерениях;
- `step()` - ступенчатый 2D график в трех измерениях;
- `scatter` - точечный 3D график;
- `plot_wireframe()` - построение каркасной поверхности в 3D;
- `plot_surface()` - построение непрерывной 3D поверхности.

Ввод [2]:

```

import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

def make_data():
    # Строим сетьку
    x = np.linspace (-10, 10, 100)
    y = np.linspace (-10, 10, 100)

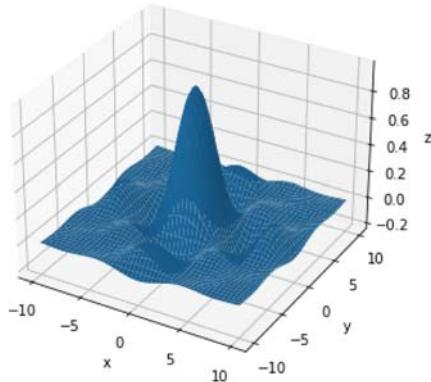
    # Создаем матрицу-сетку и в ее узлах рассчитываем значение функции
    X, Y = np.meshgrid(x, y)
    Z = np.sin(X) * np.sin(Y) / (X * Y)
    return X, Y, Z

fig = plt.figure(figsize=(7, 4))
ax = Axes3D(fig)

x, y, z = make_data()
ax.plot_surface(x, y, z)
ax.set_xticks(range(-10, 11, 5))
ax.set_yticks(range(-10, 11, 5))
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')

plt.show()

```



Функция `plot_wireframe()` из `Axes3D` используется для построения каркасной поверхности:

```
Axes3D.plot_wireframe(self, X, Y, Z, *args, **kwargs)
```

Её параметры аналогичны параметрам функции `plot_surface()`.

Ввод [14]:

```

import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

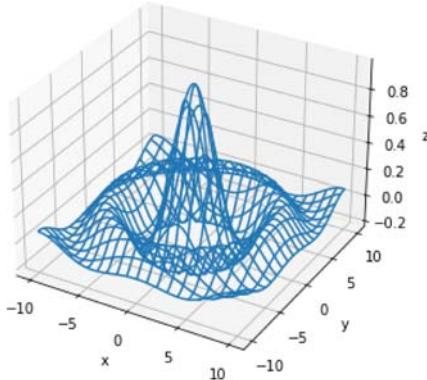
%matplotlib inline

def make_data():
    # Строим сетьку
    x = np.linspace (-10, 10, 100)
    y = np.linspace (-10, 10, 100)

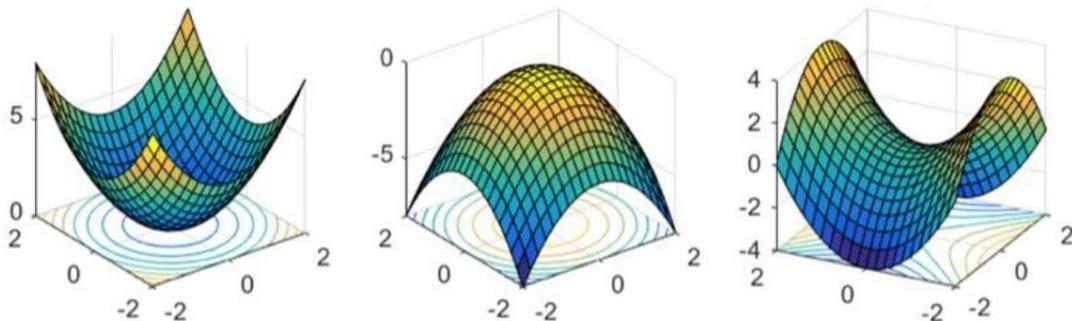
    # Создаем матрицу-сетку и в ее узлах рассчитываем значение функции
    X, Y = np.meshgrid(x, y)
    Z = np.sin(np.sqrt(X ** 2 + Y ** 2)) / np.sqrt(X ** 2 + Y ** 2)
    return X, Y, Z

if __name__ == '__main__':
    x, y, z = make_data()
    fig = plt.figure()
    axes = Axes3D(fig)
    axes.plot_wireframe(x, y, z, rstride=5, cstride=5)
    axes.set_xticks(range(-10, 11, 5))
    axes.set_yticks(range(-10, 11, 5))
    axes.set_xlabel('x')
    axes.set_ylabel('y')
    axes.set_zlabel('z')
    plt.show()

```



10.5 Построение линий уровня



Трехмерные графики часто изображают в виде линий уровня. Для этого используют следующие функции:

- `contour()` и `contourf()` – если данные по осям x, y, z представлены в виде двумерных массивов;
- `tricontour()` и `tricontourf()` – если данные для x, y, z представлены одномерными массивами.

При использовании `contourf()` линии уровня будут отображаться в виде поверхности.

Рассмотрим пример построения линий уровня, когда значения координат представлены двумерными массивами.

Ввод [36]:

```
# создание трехмерных осей
import matplotlib.pyplot as plt
import numpy as np

%matplotlib notebook

def make_data():
    # Строим сетку
    x = np.linspace (-10, 10, 100)
    y = np.linspace (-10, 10, 100)

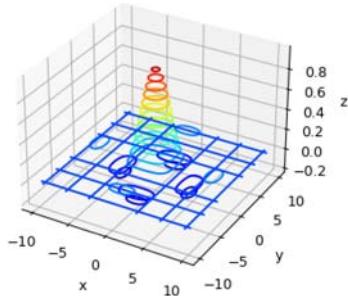
    # Создаем матрицу-сетку и в ее узлах рассчитываем значение функции
    X, Y = np.meshgrid(x, y)
    Z = np.sin(X) * np.sin(Y) / (X * Y)
    # Z = np.sin(np.sqrt(X ** 2 + Y ** 2)) / np.sqrt(X ** 2 + Y ** 2)
    return X, Y, Z

if __name__ == '__main__':
    x, y, z = make_data()
    fig = plt.figure()
    axes = fig.add_subplot(projection='3d')
    axes.set_xlabel('x')
    axes.set_ylabel('y')
    axes.set_zlabel('z')

    axes.contour(x, y, z,
                  levels=15, # число контуров
                  cmap='jet')

    plt.show()
```

<IPython.core.display.Javascript object>



На практике линии уровня рисуются в двумерной плоскости. Для этого вместо создания трехмерных координатных осей создадут обычные двумерные координатные оси. Затем в созданных координатных осях той же функцией `contour()` или `contourf()` формируются нужные графики.

Ввод [8]:

```
# создание трехмерных осей
import matplotlib.pyplot as plt
import numpy as np

%matplotlib inline

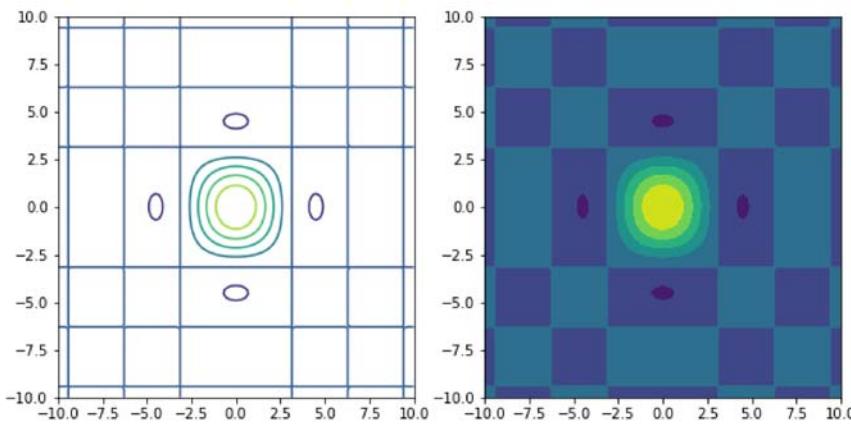
def make_data():
    # Строим сетьку
    x = np.linspace (-10, 10, 100)
    y = np.linspace (-10, 10, 100)

    # Создаем матрицу-сетку и в ее узлах рассчитываем значение функции
    X, Y = np.meshgrid(x, y)
    Z = np.sin(X) * np.sin(Y) / (X * Y)
    # Z = np.sin(np.sqrt(X ** 2 + Y ** 2)) / np.sqrt(X ** 2 + Y ** 2)
    return X, Y, Z

if __name__ == '__main__':
    x, y, z = make_data()
    fig, axes = plt.subplots(1, 2, figsize=(10, 5))

    axes[0].contour(x, y, z)
    axes[1].contourf(x, y, z)

    plt.show()
```



Как правило у каждой линии уровня находятся подписи, показывающие значение линии уровня.

Функции `contour()` и `contourf()` возвращают ссылку на экземпляр класса `QuadContourSet` (который является дочерним от класса от базового `ContourSet`). Это позволяет через объект `QuadContourSet` управлять отображением самого графика. Для этого можно сохранить ссылку на этот объект в виде некоторой переменной, например, `contour_1`, а затем для нужной координатной оси вызовем метод `clabel()`, в который в качестве параметра передадим созданный объект (`contour_1`).

```
contour_1 = axes.contour(x, y, z)
axes.clabel(contour_1)
```

или

```
contour_1 = axes.contour(x, y, z)
contour_1.clabel()
```

Ввод [18]:

```
# создание трехмерных осей
import matplotlib.pyplot as plt
import numpy as np

%matplotlib inline

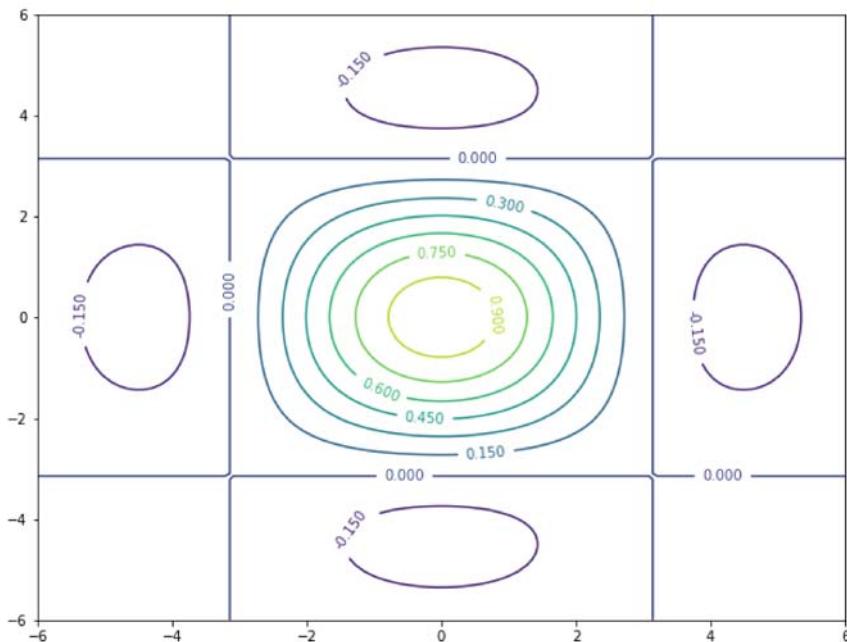
def make_data():
    # Строим сетку
    x = np.linspace (-6, 6, 100)
    y = np.linspace (-6, 6, 100)

    # Создаем матрицу-сетку и в ее узлах рассчитываем значение функции
    X, Y = np.meshgrid(x, y)
    Z = np.sin(X) * np.sin(Y) / (X * Y)
    # Z = np.sin(np.sqrt(X ** 2 + Y ** 2)) / np.sqrt(X ** 2 + Y ** 2)
    return X, Y, Z

if __name__ == '__main__':
    x, y, z = make_data()
    fig = plt.figure(figsize=(8, 6))
    axes = fig.add_axes([0, 0, 1, 1])

    contour_1 = axes.contour(x, y, z, levels=10)
    # axes.clabel(contour_1)
    contour_1.clabel()

    plt.show()
```



У функции `contour()` есть много полезных параметров, наиболее интересные из которых:

- `levels` – который определяет количество линий уровня, которые нужно отобразить на графике, причем здесь вместо конкретного числа можно указать список уровней, которые требуется отобразить;
- `colors` – позволяет управлять цветом самих линий;
- `cmap` – позволяет раскрасить линии в соответствие с выбранной картой цветов.

Кроме того метод `clabel()` позволяет форматировать текст, выводимый на линиях.

Ввод [25]:

```
# создание трехмерных осей
import matplotlib.pyplot as plt
import numpy as np

%matplotlib inline

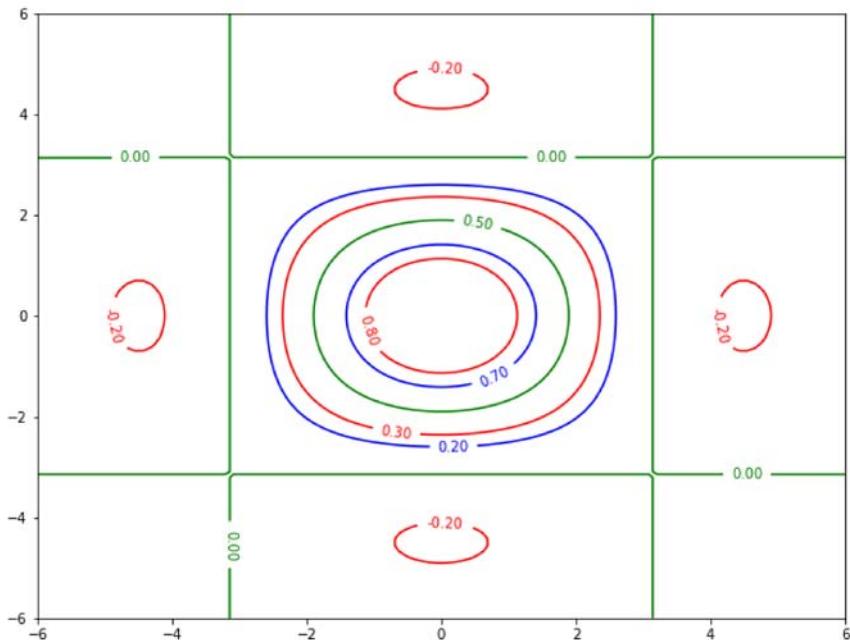
def make_data():
    # Строим сетьку
    x = np.linspace (-6, 6, 100)
    y = np.linspace (-6, 6, 100)

    # Создаем матрицу-сетку и в ее узлах рассчитываем значение функции
    X, Y = np.meshgrid(x, y)
    Z = np.sin(X) * np.sin(Y) / (X * Y)
    # Z = np.sin(np.sqrt(X ** 2 + Y ** 2)) / np.sqrt(X ** 2 + Y ** 2)
    return X, Y, Z

if __name__ == '__main__':
    x, y, z = make_data()
    fig = plt.figure(figsize=(8, 6))
    axes = fig.add_axes([0,0,1,1])

    contour_1 = axes.contour(x, y, z,
                            levels=[-0.2, 0, 0.2, 0.3, 0.5, 0.7, 0.8],
                            colors=['r', 'g', 'b'])
    contour_1.clabel(colors=['r', 'g', 'b'], fmt='%.2f')

    plt.show()
```



Можно на одной картинке совместить вывод поверхности и линий уровня.

Ввод [12]:

```
# создание трехмерных осей
import matplotlib.pyplot as plt
import numpy as np

%matplotlib inline

def make_data():
    # Строим сетку
    x = np.linspace (-6, 6, 100)
    y = np.linspace (-6, 6, 100)

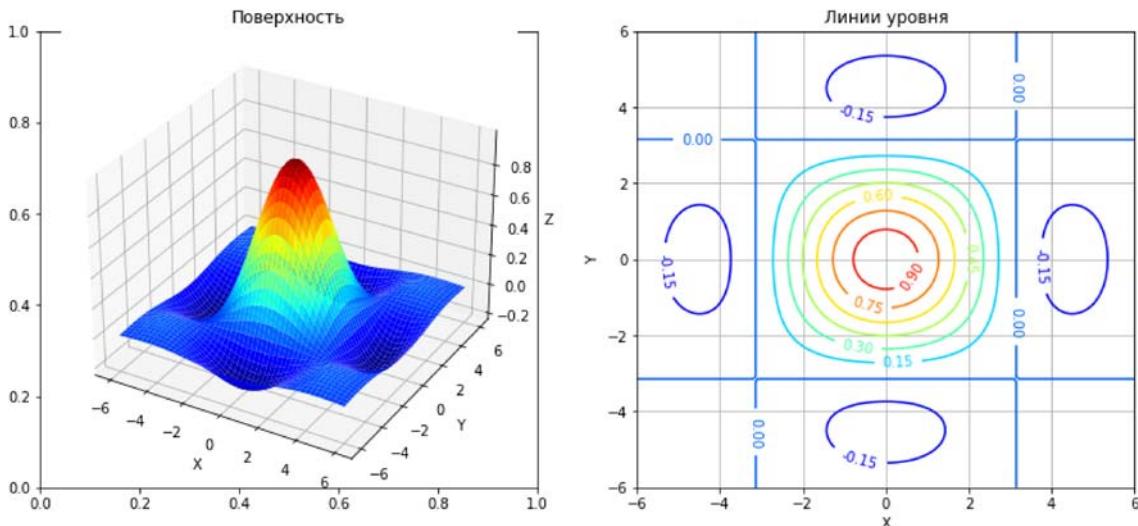
    # Создаем матрицу-сетку и в ее узлах рассчитываем значение функции
    X, Y = np.meshgrid(x, y)
    Z = np.sin(X) * np.sin(Y) / (X * Y)
    # Z = np.sin(np.sqrt(X ** 2 + Y ** 2)) / np.sqrt(X ** 2 + Y ** 2)
    return X, Y, Z

if __name__ == '__main__':
    x, y, z = make_data()
    fig, ax = plt.subplots(1, 2, figsize=(14, 6))

    # Строим поверхность
    ax[0] = fig.add_subplot(1, 2, 1, projection='3d')
    ax[0].plot_surface(x, y, z,          # отмечаем аргументы и уравнение поверхности
                        rstride = 2,   # шаг прорисовки сетки - чем меньше значение,
                        cstride = 2,   # тем плавнее прорисовка
                        cmap = 'jet',  # цветовая схема
                        )
    ax[0].set(xlabel='X', ylabel='Y', zlabel='Z', title='Поверхность')

    # рисуем линии уровня
    contour = ax[1].contour(x, y, z, levels=10, cmap='jet')
    ax[1].clabel(contour, fontsize=10, fmt='%.2f');
    ax[1].set(xlabel='X', ylabel='Y', title='Линии уровня')
    ax[1].grid()

plt.show()
```



§ 11 Создание анимированных графиков

Часто требуется посмотреть в динамике на изменение графика, т.е. необходима анимация для графика.

Конечно, можно добиться эффекта анимации, обновляя в цикле вывод графика с некоторой временной задержкой, как например в примере, где красная точка движется по синусоиде. Этот пример надо запустить в PyCharm, причем если во время движения закрыть окно с выводом графика, то оно будет появляться, пока точка не дойдет до конца отрезка.

Ввод [25]:

```
# Запустить код в PyCharm
import time
import matplotlib.pyplot as plt
import numpy as np

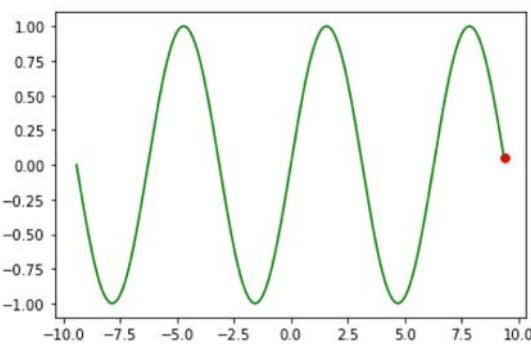
x = np.arange(-3 * np.pi, 3 * np.pi, 0.1)

plt.ion()      # Включение интерактивного режима отображения графика
for t in x:
    plt.clf()  # очистка окна

    plt.plot(x, np.sin(x), c='green')  # перерисовка синусоиды
    plt.scatter(t, np.sin(t), c='red')  # перерисовка точки

    # Функции для обновления окна
    plt.draw() #
    plt.gcf().canvas.flush_events() # метод flush_events() заставляет matplotlib
                                    # обработать свои внутренние события, в т.ч.
                                    # и те, что отвечают за перерисовку окна
    time.sleep(0.02)                # задержка на 0.02 сек = 20 миллисекунд

plt.ioff()      # отключение интерактивного режима отображения графика
plt.show()
```



Базовый класс `matplotlib` под названием `animation` занимается вопросом анимации. Он предоставляет структуру, вокруг которой строится функционал создаваемой анимации. Здесь для упрощения создания анимации предусмотрено два специальных класса:

- `FuncAnimation` — создает анимацию путем повторяющегося вызова функции `func`.
- `ArtistAnimation` — анимация, использующая фиксированный набор объектов `Artist`.

Впрочем, из этих двоих, предпочтение чаще отдают `FuncAnimation` благодаря удобному использованию.

Идея использования класса `FuncAnimation` состоит в том, что конструктору этого класса помимо других параметров, о которых будет сказано позднее, обязательно передаются:

```
FuncAnimation(fig, func, frames=count [,init_func=None, fargs=None, interval=ms, repeat=True,...])
```

- `fig` — экземпляр класса `matplotlib.figure.Figure` (этот класс отвечает за окно с графиком);
- `func` — функция, которая будет вызываться для каждого кадра анимации.

Анимация создается путем многократного вызова функции `func`, которая должна рисовать кадры. В простейшем случае опция `frames` задает количество кадров, а функция `func` (номер_кадра [, `fargs`]) в качестве первого аргумента принимает номер кадра. Функция, переданная в параметр `func` (ее удобно называть `update()`), принимает как минимум один параметр, который будет меняться от кадра к кадру, производит какие-то вычисления и возвращает список объектов, которые изменились в новом кадре. Строго говоря, эта функция должна возвращать список объектов, производных от базового класса `matplotlib.artist.Artist` (все объекты, которые вы видите на графике являются производными от этого класса).

Другие параметры `FuncAnimation`:

- `frames` — задает некую изменяемую последовательность, каждый элемент которой для каждого кадра передается в функцию создания кадра. Этот параметр может быть: списком любых объектов; итератором; целым числом (это равносильно значению `range(N)`); функцией-генератором; `None`;
- `fargs` — кортеж с дополнительными параметрами, которые передаются в функцию создания кадра, которые ей понадобятся для расчета и обновления кадра (тогда `fargs= [arg1, ...]` является списком). Если не требуется передавать дополнительные параметры, этот параметр можно опустить, что будет равносильно передаче значения `None`;
- `init_func` — функция, заданная необязательной опцией `init_func= funcname`, используется для рисования стартового кадра и инициализации графических объектов. Она вызывается один раз перед построением первого кадра;
- `interval` — задержка между кадрами в миллисекундах (по умолчанию 200);
- `blit` — использовать ли буферизацию для уменьшения моргания графика при обновлении, установка опции `blit=True` гарантирует, что только изменяющаяся часть битового массива, представляющего изображение, будет перерисовываться. Это ускоряет и улучшает качество анимации;
- `repeat` — логический параметр, если этот параметр равен `True`, то анимация начнется заново после достижения конца последовательности `frames` (функция `funcname` будет вызываться в начале каждой «анимационной сессии»).

Кроме перечисленных параметров есть еще и другие необязательные параметры, которые позволяют изменять параметры кэширования данных и задержку между перезапуском анимации (параметр `repeat_delay`), если значение `repeat` равно `True`.

Для демонстрации работы анимации в *Jupyter Notebook* необходимо использовать режим отображения графиков `%matplotlib notebook`.

Ввод [5]:

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation

%matplotlib notebook

xmin, xmax, hx = -3 * np.pi, 3 * np.pi, 0.1           # диапазон точек для расчета графика функции и шаг

fig = plt.figure(figsize=(6, 4))                         # создание окна для графика
ax = plt.axes(xlim=(xmin, xmax), ylim=(-1.1, 1.1))      # установка отображаемых интервалов по осям

graph, = plt.plot([], [], 'ro') # создание линии, которую будем анимировать

x = np.arange(xmin, xmax, hx)
plt.plot(x, np.sin(x), 'b-')

# Функция, вызываемая для каждого кадра
def update(x):
    graph.set_data(x, np.sin(x))
    return graph,

anim = FuncAnimation(fig,
                     update,
                     frames=np.arange(xmin, xmax, hx), # параметр, который будет меняться от кадра к кадру
                     interval=50,                   # задержка между кадрами в мс
                     blit=True,                     # использовать ли буферизацию для устранения мерцания
                     repeat=False                  # будет ли анимация циклической
)
plt.show();
```

<IPython.core.display.Javascript object>



Пояснение. Вначале загружаются необходимые модули и создается графическое окно `fig` с графической областью `ax`. В строке `graph, = plt.plot([], [], 'ro')` создаем пустой объект (ломаная, которая не имеет данных, но для нее сразу устанавливается цвет и тип линии), который по сути будет модифицирован в анимации. Объект `graph` будет заполнен данными в дальнейшем.

В строках

```
def update(x):
    graph.set_data(x, np.sin(x))
    return graph,
```

мы определяем функцию анимации, которая принимает `x` в качестве параметра и создает точку (или любую другую анимацию), которая сдвигается в зависимости от значения `x`. Функция здесь возвращает кортеж объектов графика, который был модифицирован, и говорит фреймворку анимации, какие части графика должны быть анимированы.

В строке `anim = FuncAnimation(...)` создаем объект анимации. Функция `FuncAnimation()` первым аргументом принимает графическое окно `fig`, в котором создается анимация. Второй аргумент является именем функции `update`, которая рисует кадры. Затем мы указываем количество кадров и интервал времени между ними. Параметр `blit` гарантирует, что будут перерисованы только те части графика, которые были изменены.

Ввод [8]:

```

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation

%matplotlib notebook

fig = plt.figure()
ax = plt.axes(xlim=(0, 8), ylim=(-1.1, 1.1))
text = fig.text(0, 1, "Frame: ", va='top') # for debugging

ax.grid(True)

line, = ax.plot([], [], lw=3) # объект - пока пустая кривая

def init():
    x = np.linspace(0, 8, 200)
    y = np.sin(100 * x/10)/(1 + x**2)
    plt.plot(x, y, 'r')

def update(i):
    x = np.linspace(0, 8, 200)
    y = np.sin(i * x/10)/(1 + x**2)
    line.set_data(x, y)
    text.set_text("Frame: %d" % i)
    return line,

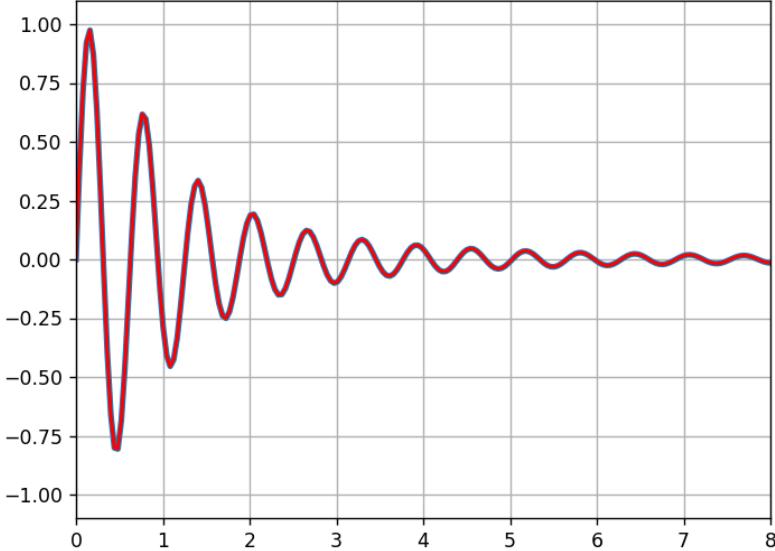
# результатам обязательно присваивать переменной
anim = FuncAnimation(fig,
                      update,
                      frames=101,
                      interval=50,
                      init_func=init
                     )

plt.show()

```

<IPython.core.display.Javascript object>

Frame: 100



Функция `plot` возвращает объект линии `line` класса `Line2D`. Напомним, что если кривых строится несколько, то функция `plot` возвращает кортеж (или список) таких объектов, например, `line1, line2 = plot(x1, y1, x2, y2)`.

Если возвращаемый объект один, то в инструкции

```
line, = ax.plot(...)
```

после имени `line` нужно ставить запятую (признак кортежа). Используя методы `set_linestyle()`, `set_marker()`, `set_drawstyle()` объекта `line`, можно менять стили оформления кривой и маркеров.

Функция `update(i)` рисует i -й кадр. Для рисования кривой используется объект `line`, созданный ранее. У него есть метод `line.set_data(x,y)`, который меняет его данные (массивы `x` и `y` координат вершин ломаной).

Для добавления фона создана функция инициализации анимации `init()`.

Рассмотрим **пример обновления графика**, который строится по данным, хранящимся в файле, которые часто обновляются. После сохранения файла график должен обновляться.

Ввод [9]:

```
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation

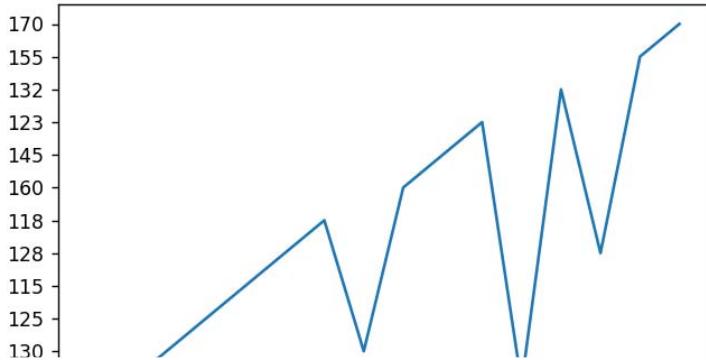
%matplotlib notebook

fig = plt.figure(figsize=(6, 4)) # создание окна для графика
ax = plt.axes()

# Функция, вызываемая для каждого кадра
def update(x):
    graph_data = open('./Data/Price.txt').read()
    lines = graph_data.split('\n')
    x = []
    y = []
    for line in lines:
        if len(line) > 2:
            day, cost = line.split(' ')
            x.append(day)
            y.append(cost)
    ax.clear()
    ax.plot(x, y)

ani = FuncAnimation(fig,
                    update,
                    interval=1000, # задержка 1000 миллисекунд (одна секунда)
                    )
plt.show();
```

<IPython.core.display.Javascript object>



§ 12 Столбчатые и круговые диаграммы

Теперь, когда вы знаете, как настраивать параметры графиков, мы рассмотрим построение различных диаграмм, которые могут пригодиться для визуализации статистической информации.

Сформируем набор данных (метки и значения) на основании которых будем строить диаграммы

Ввод [4]:

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

labels = np.array(['ФРТ', 'ФЭ', 'ФВТ', 'ФАИТУ', 'ИЭФ'])
values = np.array([235, 308, 424, 375, 256])
men = np.array([195, 286, 297, 164, 208])
women = values - men
```

12.1 Столбчатые диаграммы

12.1.2 Классические столбчатые диаграммы

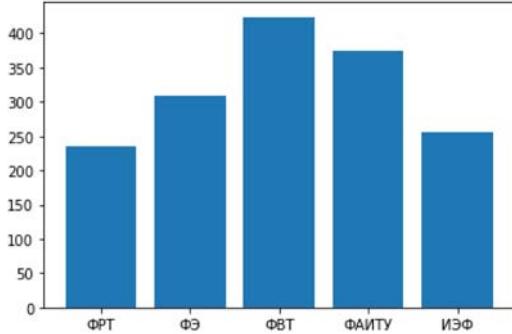
Для визуализации категориальных данных хорошо подходят столбчатые диаграммы. Для их построения используются функции:

- bar() - вертикальная столбчатая диаграмма;
- barh() - горизонтальная столбчатая диаграмма.

На вход, в самом простом варианте, этим функциям нужно передать список отметок для столбцов `x` и значения высот каждого столбца `y`.

Ввод [2]:

```
fig = plt.figure(figsize=(6, 4)) # создание окна для графика
plt.bar(labels, values);
```



Если заменим `bar()` на `barh()`, то получим горизонтальную диаграмму

Ввод [3]:

```
fig = plt.figure(figsize=(6, 4))
plt.barh(labels, values);
```

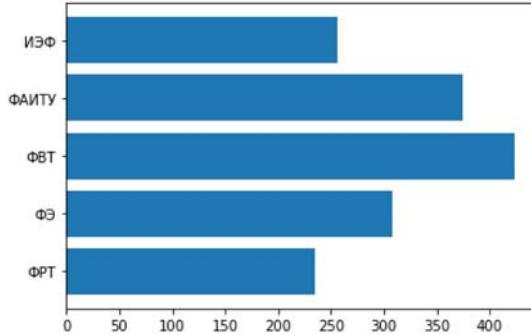


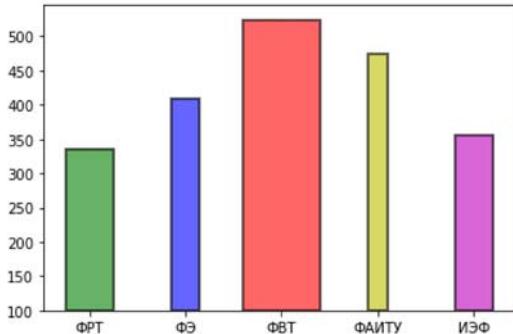
Таблица. Параметры функций `bar()` и `barh()`

Параметр	Описание параметра
x	Массив координат столбцов
height	Высоты столбцов (скалярная величина или массив)
width	Ширина столбцов (скалярная величина или массив)
bottom	у -координата базы (начальное значение столбцов, по умолчанию 0)
align	Выравнивание столбцов относительно риски { 'center', 'edge' } ; значение по умолчанию: center
color	Цвет столбцов диаграммы или набор цветовых элементов
alpha	Степень прозрачности (число от 0 до 1)
edgecolor	Цвет границы столбцов
linewidth	Ширина границы (скалярная величина или массив)
tick_label	Метки для столбца (строка или массив)
orientation	Ориентация: вертикальная или горизонтальная (vertical , horizontal)
xerr , yerr	Отображение величины погрешности (ошибки) для столбцов по горизонтали и по вертикали (число или список)
ecolor	Цвет рисок линий погрешностей

Ввод [4]:

```
fig = plt.figure(figsize=(6, 4))

plt.bar(labels,
        values,
        width=[0.5, 0.3, 0.8, 0.2, 0.4],
        bottom=100,
        color=['g', 'b', 'r', 'y', 'm'],
        alpha=0.6,
        edgecolor='k',
        linewidth=2
);
```



На гистограммах можно указывать погрешность измерения величины, как по горизонтали `xerr` так и вертикали `yerr`.

Ввод [33]:

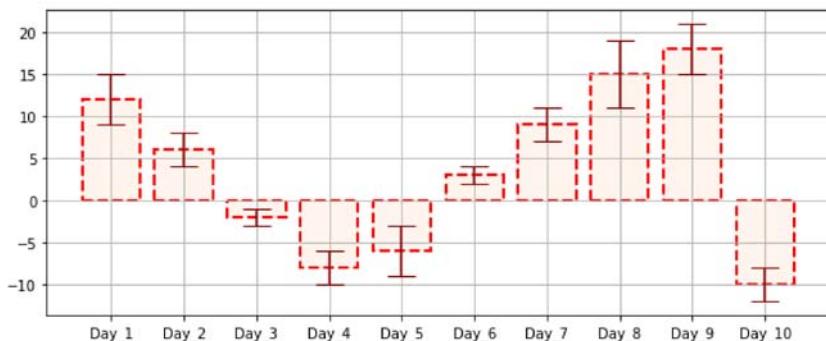
```
import numpy as np
import matplotlib.pyplot as plt

fig = plt.figure(figsize=(10, 4))
ax = fig.add_subplot()

x = [f'Day_{i+1}' for i in range(10)]
y = [12, 6, -2, -8, -6, 3, 9, 15, 18, -10]
y_error = [3, 2, 1, 2, 3, 1, 2, 4, 3, 2]      # задаем разную, но симметричную погрешность
                                                # для каждого прямоугольника

ax.bar(x, y,
       width=0.8,
       color = 'seashell',      # цвет прямоугольника
       linestyle = '--',        # начертание линии
       linewidth=2,             # ширина крайней линии
       edgecolor='r',           # цвет края прямоугольника
       yerr=y_error,            # границы погрешностей
       ecolor='darkred',         # цвет линии погрешности
       capsize = 10,             # горизонтальная черточка
       )

ax.grid()
```



Если значение погрешности не симметрично, то минимальное и максимальное значение погрешности для каждого прямоугольника задается массивов чисел с формой $(2, n)$.

Ввод [31]:

```

import numpy as np
import matplotlib.pyplot as plt

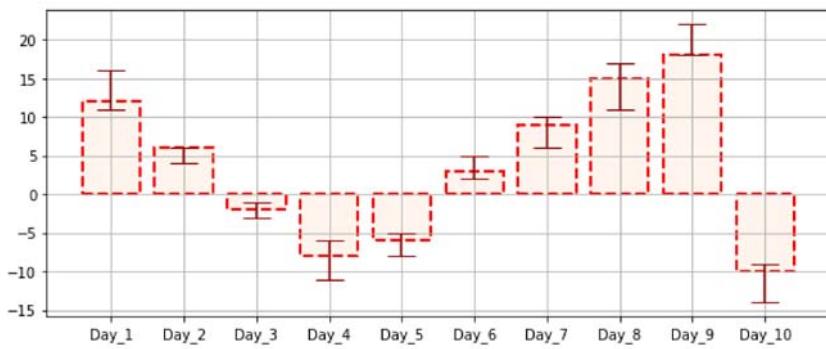
fig = plt.figure(figsize=(10, 4))
ax = fig.add_subplot()

x = [f'Day_{i+1}' for i in range(10)]
y = [12, 6, -2, -8, -6, 3, 9, 15, 18, -10]
y_error = [[1, 2, 1, 3, 2, 1, 3, 4, 0, 4],           # задаем минимальное и максимальное значение
            [4, 0, 1, 2, 1, 2, 1, 2, 4, 1]]          # погрешности для каждого прямоугольника

ax.bar(x, y,
       width=0.8,
       color = 'seashell',      # цвет прямоугольника
       linestyle = '--',        # начертание линии
       linewidth=2,             # ширина крайней линии
       edgecolor='r',           # цвет края прямоугольника
       yerr=y_error,            # границы погрешностей
       ecolor='darkred',         # цвет линии погрешности
       capsized = 10,            # горизонтальная черточка
       )

ax.grid()

```



12.1.2 Групповые столбчатые диаграммы

Ширина прямоугольников при построении столбчатой диаграммы подбирается автоматически (что нас устраивает). Но если мы имеем дело с несколькими наборами данных, которые нужно отобразить в пределах одной области `Axes`, то прямоугольники начинают перекрывать друг друга:

Ввод [6]:

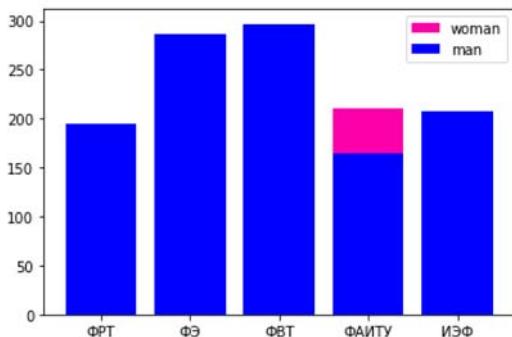
```

fig = plt.figure(figsize=(6, 4))

x = np.arange(len(labels))

plt.bar(labels, women, label='woman', color="#fc00aa")
plt.bar(labels, men, label='man', color='b')
plt.xticks(range(len(labels)), labels)
plt.legend();

```



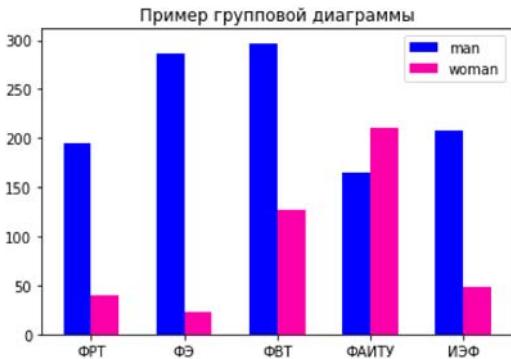
Используя определенным образом подготовленные данные, можно строить групповые диаграммы: необходимо вручную задавать ширину с помощью параметра `width` и смещение прямоугольников в массиве `x`:

Ввод [13]:

```
fig = plt.figure(figsize=(6, 4))

x = np.arange(len(labels))
width = 0.3

plt.bar(x - width/2, men, width, label='man', color='b')
plt.bar(x + width/2, women, width, label='woman', color="#fc00aa")
plt.title('Пример групповой диаграммы')
plt.xticks(range(len(labels)), labels)
plt.legend();
```



Для того чтобы состыковать прямоугольники нескольких наборов данных, достаточно выровнять одни прямоугольники по значениям других:

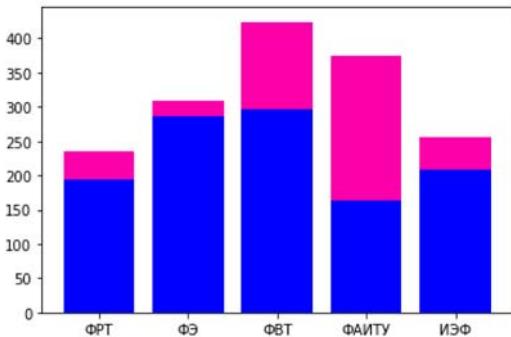
Ввод [10]:

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

fig = plt.figure(figsize=(6, 4))

plt.bar(labels, men, color='b')
plt.bar(labels, women, color="#fc00aa", bottom=men)

plt.show()
```



Задавая цвет в модели RGBA можно контролировать прозрачность прямоугольников, что так же позволяет размещать несколько наборов данных, но без смещения:

Ввод [15]:

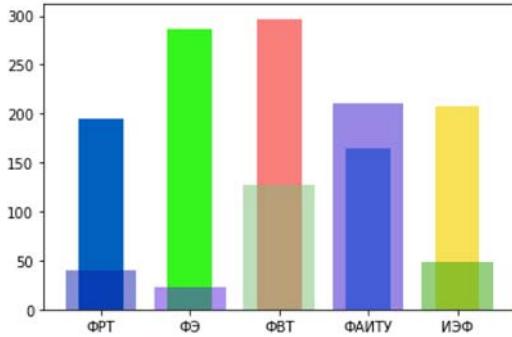
```
import numpy as np
import matplotlib.pyplot as plt

fig = plt.figure(figsize=(6, 4))

color_men = np.random.rand(7, 3)      # RGB
plt.bar(labels, men, color=color_men, width = 0.5)

color_women = np.random.rand(7, 4)      # RGBA
color_women[:,3] = 0.5
plt.bar(labels, women, color=color_women)

plt.show()
```



12.2 Круговые диаграммы

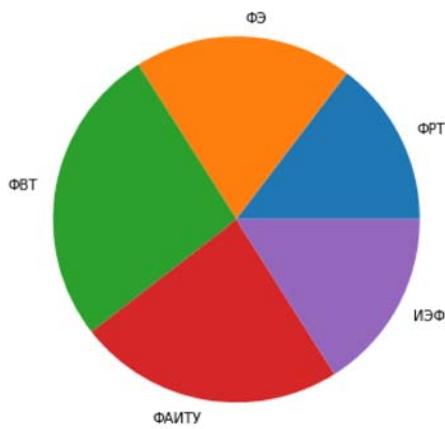
Круговые диаграммы — это наглядный способ показать доли компонентов в наборе. Они идеально подходят для отчётов, презентаций и т.п. Для построения круговых диаграмм в Matplotlib используется функция `pie()`.

Таблица. Параметры функций `pie()`

Параметр	Описание
labels	Список подписей для долей
explode	Список долей, выносимых из диаграммы
colors	Цвета долей
autopct	Формат числа величины доли внутри сегмента
pctdistance	Расстояние от центра доли до текстовой метки
shadow	Отображение тени у диаграммы
labeldistance	Расстояние для текстовых метод долей (по умолчанию 1.1)
startangle	Начальный угол поворота диаграммы (против часовой стрелки)
radius	Радиус диаграммы
counterclock	Порядок размещения долей на диаграмме (по часовой стрелке или против часовой) True/False
center	Координата центра диаграммы (по умолчанию (0, 0))
frame	Отображение рамки вокруг диаграммы (True/False)
wedgeprops	Словарь дополнительных параметров (класс <code>matplotlib.patches.Wedge</code>)

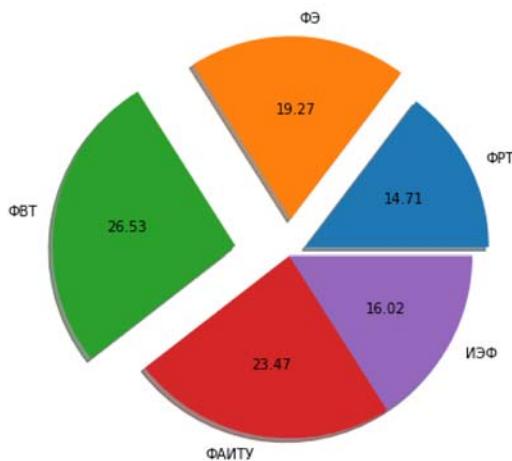
Ввод [22]:

```
fig, ax = plt.subplots(figsize=(6, 6))
ax.pie(values, labels=labels);
```



Ввод [26]:

```
fig, ax = plt.subplots(figsize=(6, 6))
exp = (0.1, 0.2, 0.3, 0, 0)
ax.pie(values, labels=labels, autopct='%.2f', explode=exp, shadow=True);
```



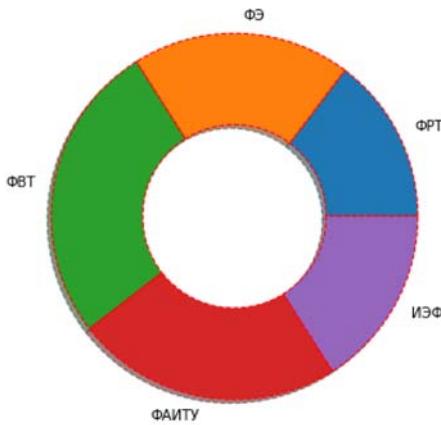
Используя словарь `wedgeprops`, можно сформировать диаграмму с пустотой внутри.

Построим круговую диаграмму с пустым круговым пространством внутри, где ширина каждого сегмента составляет половину от начальной величины. .

Ввод [38]:

```
fig, ax = plt.subplots(figsize=(6, 6))

ax.pie(values,
        labels=labels,
        shadow=True,
        wedgeprops={'lw':1, 'ls':'--', 'edgecolor':'r', 'width': 0.5}
);
```



12.3 Гистограммы hist()

Иногда данные требуется сгруппировать по определенным диапазонам и подсчитать сколько значений попадает в тот или иной интервал. Для выполнения такой задачи хорошо подходят столбчатые диаграммы и довольно известный их вид – это **гистограмма распределения случайной величины**.

Давайте сгенерируем вектор из 500 случайных величин и выведем их в виде гистограммы, используя функцию `hist()`:

Ввод [1]:

```
import numpy as np

data = np.random.normal(0, 2, 500)
```

Ввод [2]:

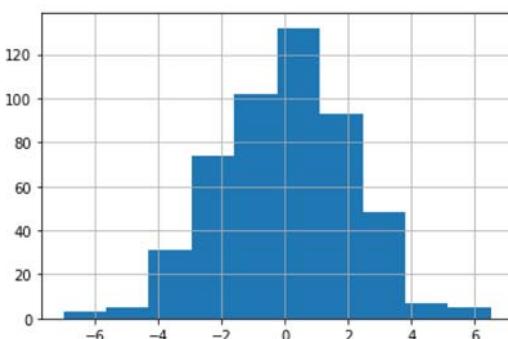
```
import matplotlib.pyplot as plt

%matplotlib inline

fig = plt.figure(figsize=(6, 4))

plt.hist(data)
plt.grid()

plt.show()
```



Фактически, мы здесь имеем набор столбиков, высота которых определяется числом случайных величин, попавших в тот или иной диапазон. Причем, по умолчанию функция `hist()` разбивает весь интервал на равных 10 диапазонов. Если требуется изменить это число, то мы можем его указать вторым параметром:

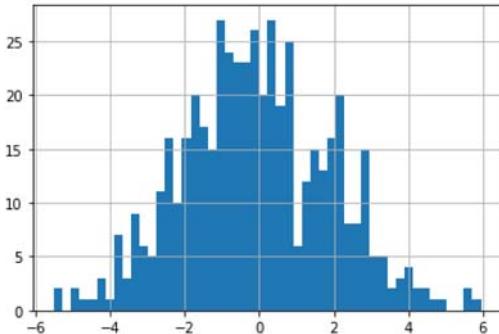
Ввод [15]:

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

fig = plt.figure(figsize=(6, 4))
ax = fig.add_subplot()

y = np.random.normal(0, 2, 500)
ax.hist(y, 50)
ax.grid()

plt.show()
```



Аргумент `bins` задает количество бинов гистограммы. По умолчанию используется значение 10. Если вместо целого числа в аргумент `bins` передать кортеж значений, то они будут использованы для задания границ бинов. Таким образом можно построить гистограмму с произвольным разбиением.

Таблица. Некоторые другие аргументы функции `pyplot.hist`

Параметр	Описание
<code>range</code>	Диапазон значений (<code>float, float</code>), в котором строится гистограмма. Значения за пределами заданного диапазона игнорируются
<code>density</code>	Значение типа <code>bool</code> . При значении <code>True</code> будет построена гистограмма, соответствующая плотности вероятности, так что площадь гистограммы будет равна единице
<code>weights</code>	Список <code>float</code> значений того же размера, что и набор данных. Определяет вес каждого значения при построении гистограммы
<code>histtype</code>	Определяет тип отрисовки гистограммы. Стока, принимавшая значения <code>{ bar, barstacked, step, stepfilled }</code>

В качестве первого аргумента можно передать кортеж наборов значений. Для каждого из них будет построена гистограмма. Аргумент `stacked` со значением `True` позволяет строить сумму гистограмм для кортежа наборов.

Ввод [65]:

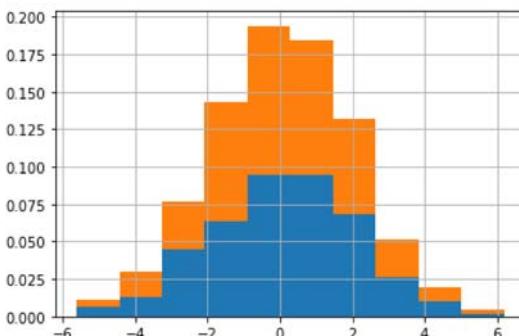
```
import numpy as np
import matplotlib.pyplot as plt

fig = plt.figure(figsize=(6, 4))
ax = fig.add_subplot()

data1 = np.random.normal(0, 2, 500)
data2 = np.random.normal(0, 2, 500)

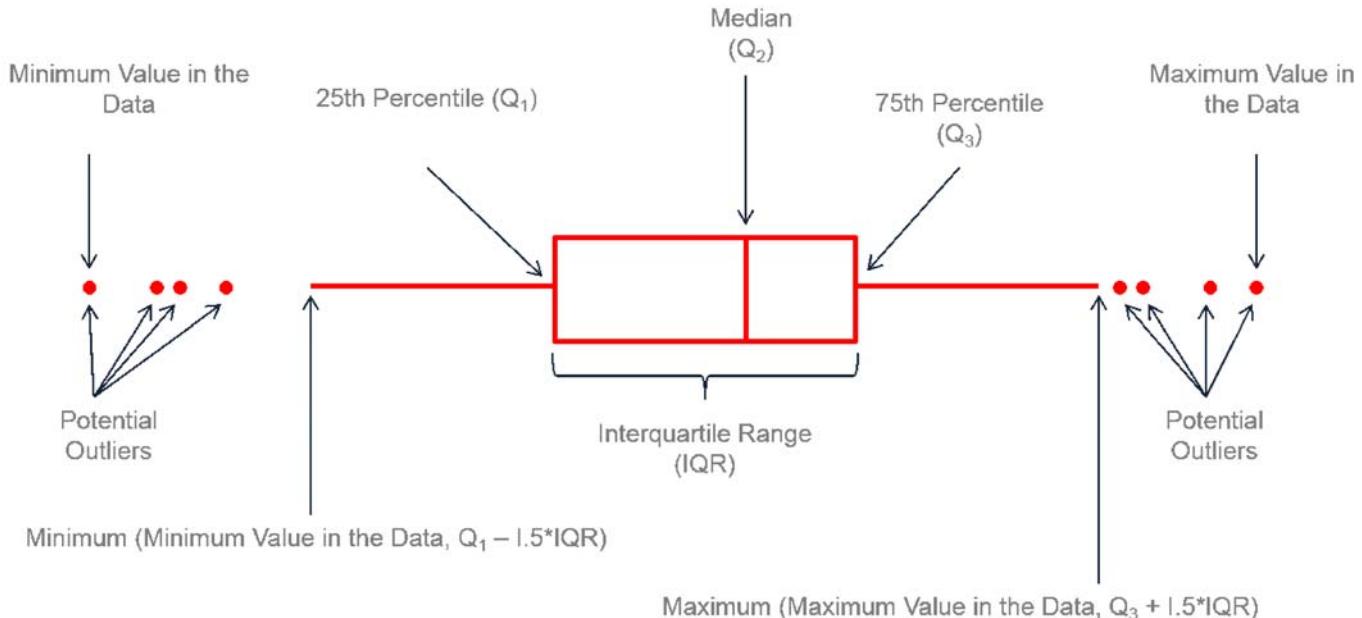
ax.hist([data1, data2], bins=10,
        density=True,
        stacked=True
       )
ax.grid()

plt.show()
```



12.4 Ящик с усами boxplot()

Boxplot, часто называемый **ящик с усами**, - это график, который используется в описательной статистике, компактно изображающий одномерные статистики распределения переменной. Такой вид диаграммы в удобном формате показывает медиану, 25-процентный квантиль, 75-процентный квантиль (оба этих квантиля называют квартилями), минимальное значение и максимальное значение, а также выбросы. Расстояние между различными частями ящика позволяет определить степень разброса, асимметрию данных и выявить выбросы.



Усы у ящика (как правило) показывают максимальное и минимальное значение в наборе данных. Иногда на графике, рядом с усами появляются одна или две точки. Такие точки обозначают выбросы - значения которые находятся очень далеко от статистически значимой части данных:

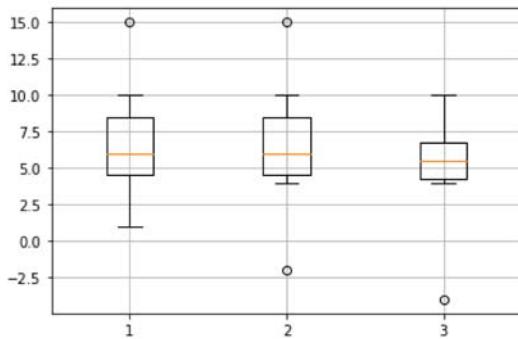
Ввод [23]:

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

data = [[1, 5, 7, 4, 6, 10, 15],
        [-2, 5, 7, 4, 6, 10, 15],
        [-4, 5, 7, 4, 6, 10]]

plt.boxplot(data)
plt.grid()

plt.show()
```



§ 13 Images (пиксельные картинки)

Matplotlib позволяет не только отображать двумерные массивы, но и загружать их в виде массивов numpy .

В самом простом случае, изображение можно представить в виде двумерного массива чисел. Количество строк и столбцов в массиве - это высота и ширина изображения, а каждое число в массиве соответствует определенному цвету из встроенной палитры (изменяется параметром `cmap`). Как правило, числа, должны находиться в интервале [0, 1]. Если указать целые числа, то все они будут автоматически нормализованы и приведены к данному интервалу.

Ввод [82]:

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

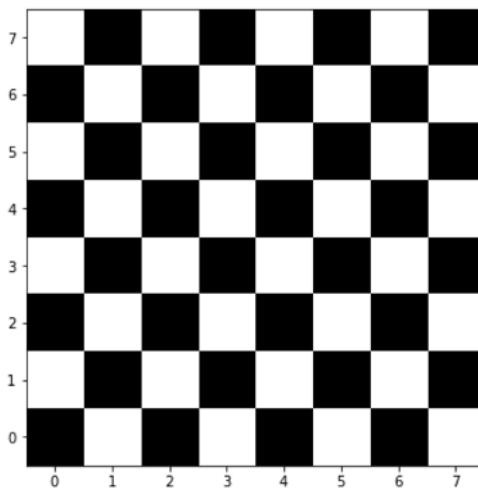
%matplotlib inline

# Двумерный массив пикселей:
data = [[1, 0, 1, 0, 1, 0, 1, 0],
        [0, 1, 0, 1, 0, 1, 0, 1],
        [1, 0, 1, 0, 1, 0, 1, 0],
        [0, 1, 0, 1, 0, 1, 0, 1],
        [1, 0, 1, 0, 1, 0, 1, 0],
        [0, 1, 0, 1, 0, 1, 0, 1],
        [1, 0, 1, 0, 1, 0, 1, 0],
        [0, 1, 0, 1, 0, 1, 0, 1]]

fig, ax = plt.subplots(figsize=(6, 6))

ax.imshow(data, cmap='binary', origin='lower')

plt.show();
```



С помощью параметра `origin` начало координат переместилось в нижний левый угол.

Ввод [1]:

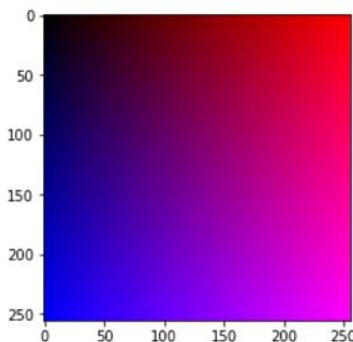
```
import matplotlib.pyplot as plt
import numpy as np

n = 256
u = np.linspace(0, 1, n)

x, y = np.meshgrid(u, u)

z = np.zeros((n, n, 3))
z[:, :, 0] = x
z[:, :, 2] = y

plt.figure()
plt.imshow(z)
plt.show()
```



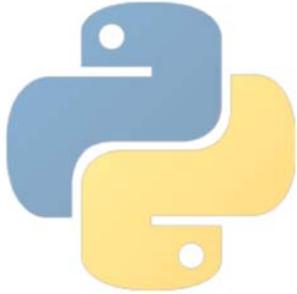
Можно загрузить картинку из файла. Для отображения изображения сначала нужно прочитать это изображение с помощью функции `imread` библиотеки `matplotlib.pyplot`. Это будет обычный `numpy.array`. Как мы знаем, цветные изображения хранятся в трехмерном массиве: первые два - размерность картинки $n \times m$. (третье измерение представляет собой RGB(красный, Зеленый, Синий) цвета). Можно сделать это изображение более или менее прозрачным, используя параметр `alpha`.

Ввод [86]:

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

picture = plt.imread('Image/python.png')
plt.axis('off')
plt.imshow(picture, alpha=0.6);
print(type(picture), picture.shape)

<class 'numpy.ndarray'> (231, 231, 4)
```



Синтаксис срезов позволяет вывести часть картинки.

Ввод [7]:

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

picture = plt.imread('Image/python.png')
n, m = picture.shape[0], picture.shape[1]
plt.axis('off')
plt.imshow(picture[n // 3: 2 * n // 3, m // 3: 2 * n // 3]);
```

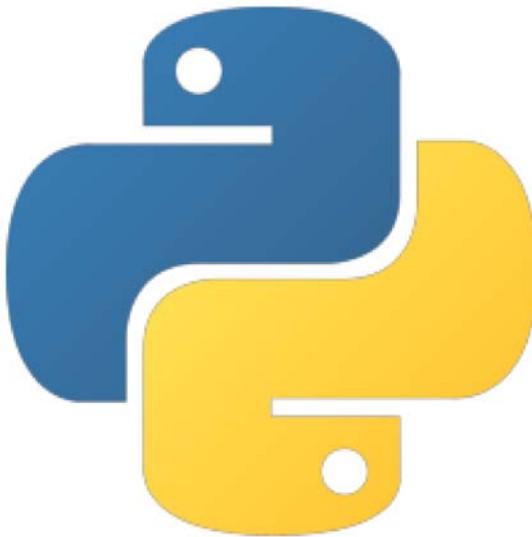


Альтернативный способ отображения картинок из файла:

Ввод [1]:

```
from IPython.display import Image
Image(filename='Image/python.png',width=400,height=300)
```

Out[1]:



§ 14 Эффект рисования от руки

В *Matplotlib 1.3* появилась возможность применения **эффекта рисования от руки**, чтобы графики выглядели в стиле комиксов с сайта *xkcd.com*. Такая возможность может быть полезна, если вы хотите придать вашим графикам некоторую несерьезность, например, на презентациях.

Чтобы включить этот эффект достаточно вызвать функцию `xkcd()` библиотеки `pyplot`:

```
import matplotlib.pyplot as plt

plt.xkcd() # Включить оформление в стиле xkcd.com
```

Однажды вызваный функцией `pylab.xkcd()` **эффект рисования от руки начинает применяться ко всем последующим графикам**, поэтому если требуется применить этот эффект только к одному графику, то **отключить действие этого эффекта** можно одним из следующих способов:

- воспользоваться `plt.rcdefaults()` для возврата к параметрам оформления, заданным по-умолчанию;
- воспользоваться оператором `with`.

Ввод []:

```
from matplotlib.animation import style
style.use('fivethirtyeight')
```

Ввод [3]:

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

x = np.linspace(-np.pi, np.pi, 100)

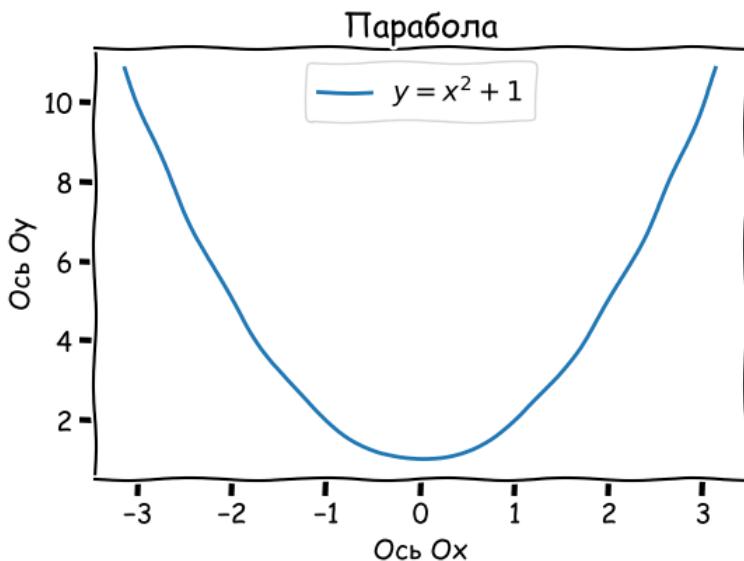
plt.xkcd() # Включить оформление в стиле xkcd.com

plt.figure(figsize=(6, 4))

plt.plot(x, x * x + 1, label='$y = x^2 + 1$')
plt.xlabel('Ось Ox')
plt.ylabel('Ось Oy')
plt.title('Парабола')
plt.legend()

plt.rcdefaults() # Вернуться к параметрам оформления, заданным по-умолчанию

plt.show()
```



Такой эффект применим не только к обычным графика вида $y = f(x)$, но и к остальным типам графиков, в том числе и к трехмерным.

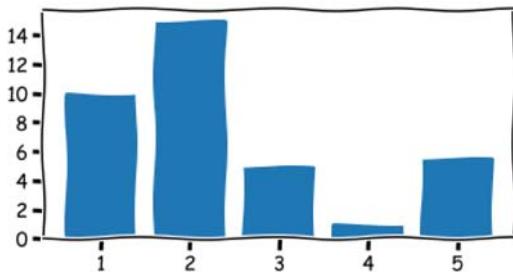
Ввод [1]:

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

plt.figure(figsize=(6, 3))

# Нарисовать гистограмму
with plt.xkcd():
    xbar = [1, 2, 3, 4, 5]
    hbar = [10.0, 15.0, 5.0, 1.0, 5.5]
    plt.bar(xbar, hbar)

plt.show()
```



У функции `pylab.xkcd()` есть три необязательных параметра, которые позволяют настроить искажения, имитирующие дрожание руки. Все три параметра являются дробными числами (`float`). Объявление функции выглядит следующим образом:

```
matplotlib.pyplot.xkcd(scale=1, length=100, randomness=2)
```

Здесь:

- `scale` задает размах дрожания "руки" при рисовании.
- `length` задает период дрожания руки. Чем меньше это число, тем больше периодов колебания будет на графике.

- `randomness` задает величину случайного шума, количество резких дрожаний.

Ввод [2]:

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

x = np.arange (-5, 5, 0.05)
y = np.sin(x)

plt.subplots(2, 2, figsize=(10, 6))

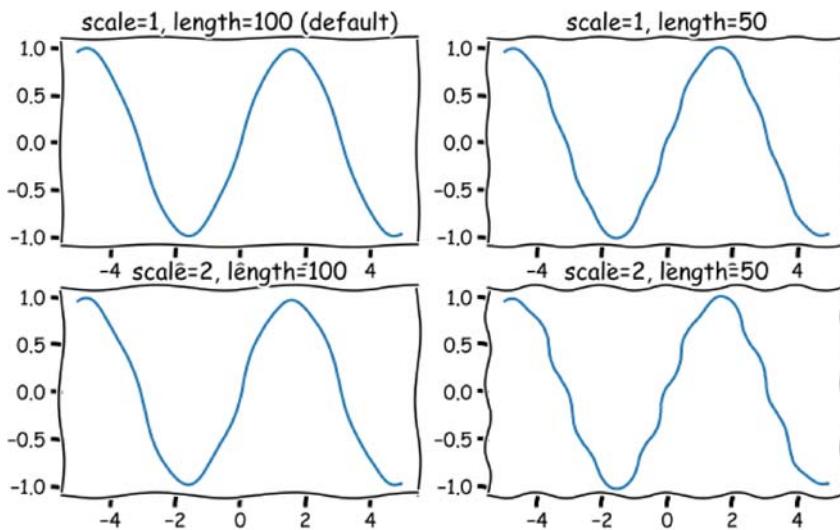
# Первый график
with plt.xkcd(scale=1, length=100):
    plt.subplot(2, 2, 1)
    plt.plot(x, y)
    plt.title('scale=1, length=100 (default)')

# Второй график
with plt.xkcd(scale=1, length=50):
    plt.subplot(2, 2, 2)
    plt.plot(x, y)
    plt.title('scale=1, length=50')

# Третий график
with plt.xkcd(scale=2, length=100):
    plt.subplot(2, 2, 3)
    plt.plot(x, y)
    plt.title('scale=2, length=100')

# Четвертый график
with plt.xkcd(scale=2, length=50):
    plt.subplot(2, 2, 4)
    plt.plot(x, y)
    plt.title('scale=2, length=50')

plt.show()
```



§ 15 Глобальное изменение настроек графика

В настройках `matplotlib` у области рисования `Axes` имеется несколько атрибутов, которые заданы по умолчанию. Именно эти параметры определяют свойства графических элементов (размеры окна с фигурой, толщина линий, цвет линий, наличие вспомогательной сетки `grid`, цвет и размер шрифта подписей координатных осей) для каждого вновь созданного тем или иным способом экземпляра `Axes`.

Часто требуется менять одни и те же параметры для многих графиков (например, у всех включать сетку), в этом случае можно для каждого графика устанавливать нужные параметры явно или один раз в начале скрипта изменить значения параметров по умолчанию, чтобы больше о них не задумываться. Кроме того, можно изменять эти параметры глобально для всех пользователей или для конкретного пользователя с помощью специального файла настроек с именем `matplotlibrc`. Располагаться он может в разных местах в зависимости от его области действия:

1. В директории со скриптом. В этом случае изменение глобальных параметров с помощью `matplotlibrc` влияет только на скрипты, расположенные в этой директории.
2. В поддиректории `.matplotlib` внутри директории пользователя. В этом случае параметры, измененные в этом файле будут касаться всех скриптов, запущенных из-под данного пользователя.
3. В директории, где установлена библиотека `matplotlib`. Изменения, внесенные в этот файле, касаются всех пользователей.

Особенностью работы `Matplotlib` с файлами `matplotlibrc` заключается в том, что при выполнении скрипта учитывается только один такой файл. То есть, если в папке со скриптом есть этот файл, то в директории пользователя `matplotlibrc` уже не учитывается. Аналогично, если есть файл в директории пользователя, то не важно, что содержит файл в директории с установленной библиотекой.

[matplotlibrc \(\[https://jenyay.net/uploads/Matplotlib/RcParams/matplotlibrc_default\]\(https://jenyay.net/uploads/Matplotlib/RcParams/matplotlibrc_default\)\)](https://jenyay.net/uploads/Matplotlib/RcParams/matplotlibrc_default) представляет собой текстовый файл, содержащий строки вида:

```
### FIGURE
# See http://matplotlib.sourceforge.
#figure.figsize   : 8, 6    # figure
#figure.dpi       : 80     # figure
#figure.facecolor : 0.75   # figure
#figure.edgecolor: white  # figure
# The figure subplot parameters. All
# figure width or height
#figure.subplot.left  : 0.125
#figure.subplot.right : 0.9
#figure.subplot.bottom : 0.1
#figure.subplot.top   : 0.9
#figure.subplot.wspace : 0.2
### GRIDS
#grid.color      : black  # grid color
#grid.linestyle   : :     # dotted
#grid.linewidth   : 0.5    # in points
### CONFIGURATION BEGINS HERE
# the default backend; one of GTK GTKAgg GTKCairo Co
# Mac OSX QtAgg Qt4Agg TkAgg WX WXAgg Agg Cairo GDK P
# You can also deploy your own backend outside of ma
# referring to the module name (which must be in the
# 'module.:./my_backend')
backend          : GTKAgg
### IMAGES
#image.aspect : equal
#image.interpolation : bilinear
#image.cmap   : jet
#image.lut    : 256
#image.origin : upper
#image.resample : False
### Legend
#legend.fancybox : False # if True, u
#legend.el        : None  # legend, el
#legend.isaxes   : True
#legend.numpoints: 2     # the number
#legend.fontsize : large
#legend.pad      : 0.0    # deprecated
#legend.borderpad: 0.5   # border whi
#legend.markerscale: 1.0  # the relati
#legend.labelssep: 0.010 # deprecated
#legend.labelsspacing: 0.5 # the vertic
### GRIDS
#grid.color      : black  # grid color
#grid.linestyle   : :     # dotted
#grid.linewidth   : 0.5    # in points
### Legend
#legend.fancybox : False # if True, use a rc
#legend.el        : None  # legend, else a re
#legend.isaxes   : True
#legend.numpoints: 2     # the number of poi
#legend.fontsize : large
#legend.pad      : 0.0    # deprecated; the f
#legend.borderpad: 0.5   # border whitespace

```

В этом файле описаны возможные параметры с подробными комментариями, на английском языке. Для комментариев используется символ `#`, работающий так же, как и в Python. Каждая строка файла `matplotlibrc` описывает параметры в виде:

параметр : значение

При этом некоторые параметры имеют вложенную структуру:

параметр.подпараметр1 : значение

параметр.подпараметр2 : значение

Параметр `backend`, обязательно должен присутствовать в этом файле (то есть файл `matplotlibrc` не может быть пустым). Этот параметр задает, с помощью какой библиотеки будут создаваться окна с фигурами. Параметр `backend` может принимать следующие значения (значения этих параметров регистронезависимы): `GTK3Agg` , `GTK3Cairo` , `nbAgg` , `MacOSX` , `Qt4Agg` , `Qt4Cairo` , `Qt5Agg` , `Qt5Cairo` , `TkAgg` , `TkCairo` , `WebAgg` , `WX` , `WXAgg` , `WCairo` , `agg` , `cairo` , `ps` , `pdf` , `svg` , `pgf` , `template` .

Под Windows в качестве значения по умолчанию для `backend` используется `TkAgg` , то есть для создания окон используется `Tkinter` . Если у вас установлены другие перечисленные библиотеки, то их также можно использовать. Значения параметра `backend` `Agg` , `Cairo` , `GDK` , `PS` , `PDF` , `SVG` предназначены для вывода результатов в файлы, а не на экран.

Изменять глобальные параметры сразу для многих скриптов можно, изменяя значения нужного параметра в файле `matplotlibrc`: можно добиться пользовательской настройки шрифтов, цвета и других параметров области рисования `axes` .

Однако есть способ изменения этих параметров в пределах скрипта. Для этого используется переменная `rcParams` из модуля `matplotlib` , которая представляет собой словарь, где в качестве ключа используются все те же параметры, что используются в файле `matplotlibrc` , а в качестве значений - соответственно значения этих параметров.

```
matplotlib.rcParams['figure.facecolor'] = 'yellow'
matplotlib.rcParams['axes.facecolor'] = 'lightblue'
matplotlib.rcParams['font.family'] = 'serif'
matplotlib.rcParams['font.size'] = 8
matplotlib.rcParams['font.weight'] = 'bold'
matplotlib.rcParams['text.color'] = 'red'
matplotlib.rcParams["lines.linewidth"] = 3
matplotlib.rcParams["lines.linestyle"] = "--"
matplotlib.rcParams["axes.grid"] = True
matplotlib.rcParams['scatter.marker'] = '*'
```

Вы можете легко найти все параметры `rcParams` , просто распечатав их.

```
print(matplotlib.rcParams)
```

Кроме того, в модуле `matplotlib` есть еще переменная `rcParamsDefault` , аналогичная `rcParams` (большой словарь переменных), но содержащая значения по умолчанию. Так что при желании можно восстановить значения по умолчанию из `rcParamsDefault` . Но только **НЕ ДЕЛАЙТЕ** так:

`matplotlib.rcParams = matplotlib.rcParamsDefault` . А делается это так:

```
import matplotlib as mpl
mpl.rcParams.update(mpl.rcParamsDefault)
```

Ввод [1]:

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

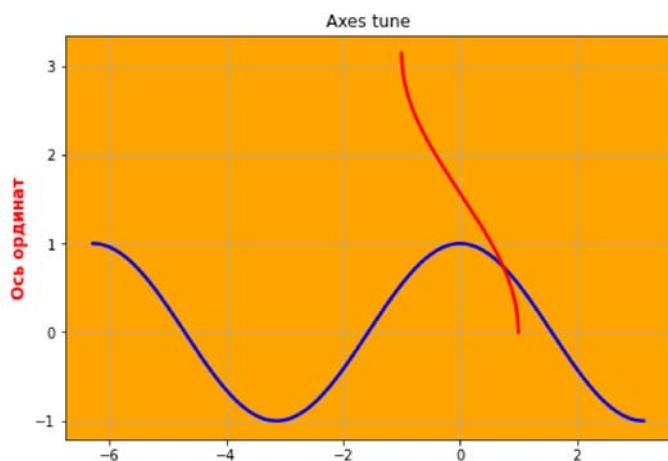
rcParams['axes.grid'] = True
rcParams['axes.facecolor'] = 'orange'
rcParams['axes.labelcolor'] = 'red'
rcParams['axes.labelsize'] = 'large'
rcParams['axes.labelweight'] = 'bold'
rcParams['axes.titlesize'] = 12

x1 = np.linspace(-2*np.pi, np.pi, 200, endpoint=True)
x2 = np.linspace(-1, 1, 200, endpoint=True)
cos, arccos = np.cos(x1), np.arccos(x2)

fig = plt.figure()
ax = fig.add_axes([0, 0, 1, 1])
ax.set_title("Axes tune")
ax.set_xlabel(u'Ось абсцисс')
ax.set_ylabel(u'Ось ординат')

ax.plot(x1, cos, 'b-', lw = 2.5, label = "Косинус")
ax.plot(x2, arccos, 'r-', lw = 2.5, label = "Арккосинус")

plt.show()
```



Ввод [6]:

```

import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl

mpl.rcParams.update(mpl.rcParamsDefault)

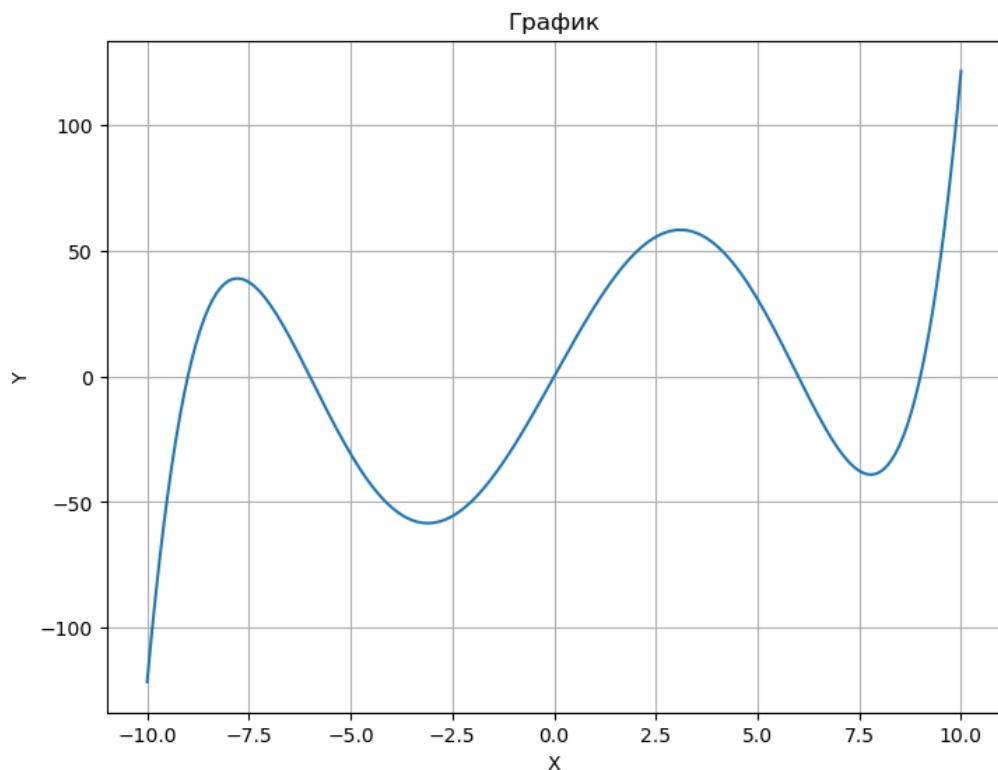
x = np.linspace(-10, 10, 200)
y = 0.01*(x + 9)*(x + 6)*(x - 6)*(x - 9) * x

fig = plt.figure()

# Добавим на рисунок область рисования
ax = fig.add_axes([0, 0, 1, 1])
ax.plot(x, y)
ax.grid(True)
ax.set_title("График")
ax.set_xlabel('X')
ax.set_ylabel('Y')

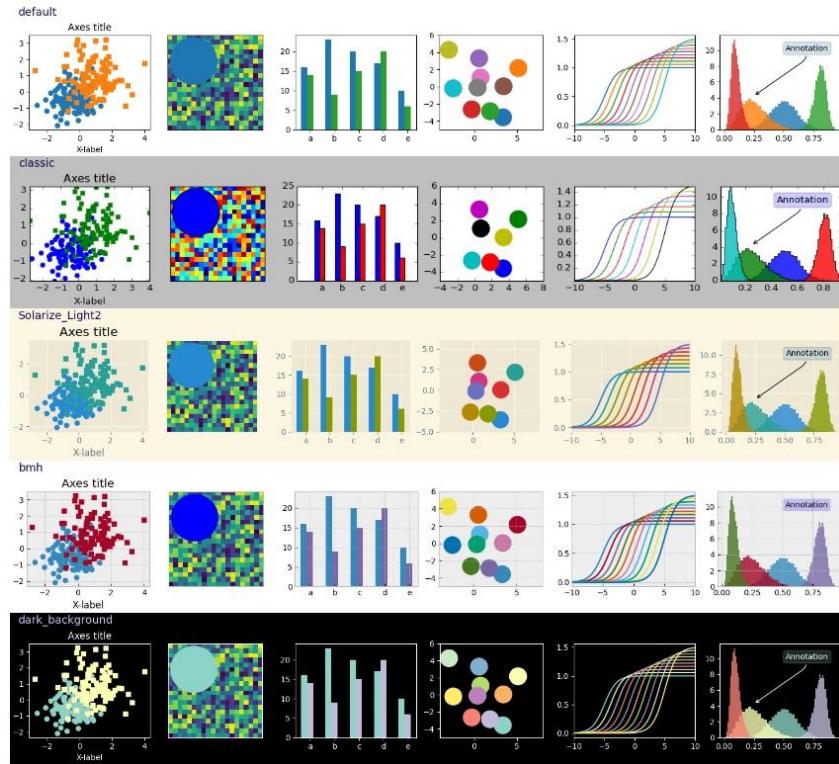
plt.show()

```



Итак, как видите, *Matplotlib rcParams* позволяет вам создавать свой собственный уникальный и многоразовый стиль для ваших графиков.

В *Matplotlib* существуют **стили** - предопределенные наборы параметров `rcParams`, которые определяют внешний вид графика, и применяются для оформления точечной диаграммы, изображения, гистограммы, линейного графика и гистограммы.



Справочник по таблицам [стилей](https://matplotlib.org/3.5.1/gallery/style_sheets/style_sheets_reference.html) (https://matplotlib.org/3.5.1/gallery/style_sheets/style_sheets_reference.html) дает обзор встроенных стилей.

Список встроенных стилей можно посмотреть, используя скрипт (список доступных имен стилей посмотрите `style.available`):

```
import matplotlib.pyplot as plt

style_list = sorted(style for style in plt.style.available)
print(style_list)
```

Для применения параметров нужного стиля к оформлениям графика используется метод `matplotlib.style.use('style_name')`, в который передается название соответствующего стиля.

Ввод [11]:

```
import matplotlib.pyplot as plt

style_list = sorted(style for style in plt.style.available)
print(style_list)
```

[Solarize_Light2, '_classic_test_patch', 'bmh', 'classic', 'dark_background', 'fast', 'fivethirtyeight', 'ggplot', 'grayscale', 'seaborn', 'seaborn-bright', 'seaborn-colorblind', 'seaborn-dark', 'seaborn-dark-palette', 'seaborn-darkgrid', 'seaborn-deep', 'seaborn-muted', 'seaborn-notebook', 'seaborn-paper', 'seaborn-pastel', 'seaborn-poster', 'seaborn-talk', 'seaborn-ticks', 'seaborn-white', 'seaborn-whitegrid', 'tableau-colorblind10']

Ввод [20]:

```

import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl

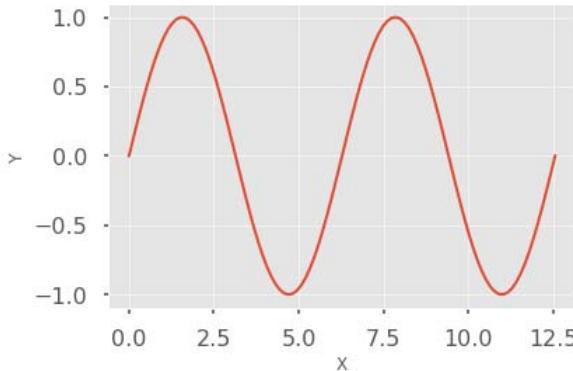
%matplotlib inline

mpl.style.use('ggplot')

x = np.linspace(0, 4 * np.pi, 100)
y = np.sin(x)

plt.figure()
plt.plot(x, y)
plt.xlabel('X')
plt.ylabel('Y')
plt.show()

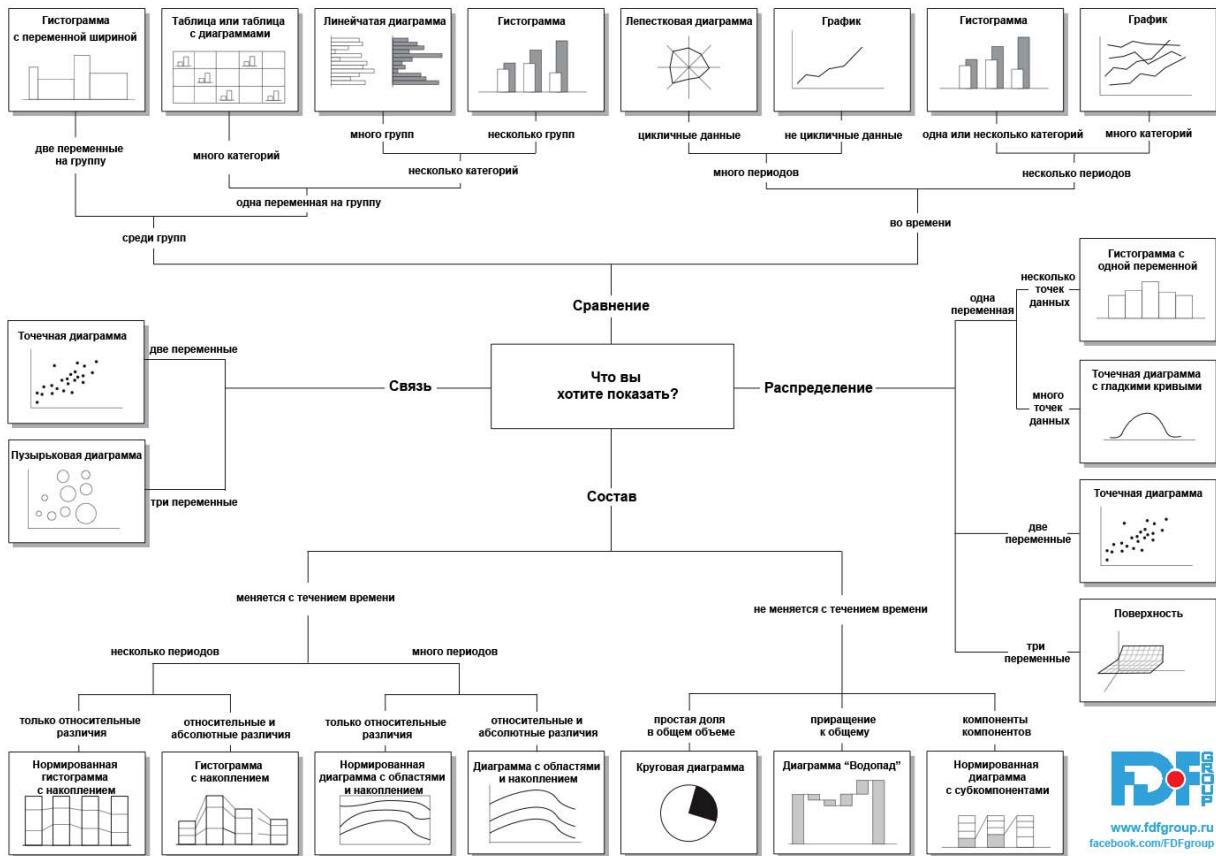
```



§ 16 Рекомендации по выбору вида графика/диаграммы

На рисунке показано, какой вид графика/диаграммы выбрать для визуализации данных в зависимости от того, что требуется показать.

Как выбрать диаграмму?



Подготовлено: <http://www.extremepresentation.com/design/charts/> (c)
Перевод: маркетинговое агентство FDFgroup

Источник: <http://www.fdfgroup.ru/?id=279> (<http://www.fdfgroup.ru/?id=279>)

§ 17 Дополнительные материалы и ссылки

Визуализация данных — это очень обширная тема. Поэтому, если в своей необходимо создавать отчёты или готовить графики для публикации, можно сохранить список ссылок, приведённый ниже.

Matplotlib

При построении графиков можно использовать официальную документацию по наиболее мощной графической библиотеке [Matplotlib](https://matplotlib.org/) (<https://matplotlib.org/>). С методами создания простейших графиков можно познакомиться в разделе [Tutorials](https://matplotlib.org/tutorials/index.html) (<https://matplotlib.org/tutorials/index.html>). Более разнообразные графики и полный код для их создания можно найти в разделе [Галерея](https://matplotlib.org/gallery/index.html) (<https://matplotlib.org/gallery/index.html>).

Seaborn

Для создания визуализаций со сложными настройками параметров отображения данных обычно используется библиотека [Seaborn](https://seaborn.pydata.org/) (<https://seaborn.pydata.org/>). На официальном сайте можно найти разделы с [пошаговыми инструкциями](https://seaborn.pydata.org/tutorial.html) (<https://seaborn.pydata.org/tutorial.html>) и [галерею](https://seaborn.pydata.org/examples/index.html) (<https://seaborn.pydata.org/examples/index.html>) со сложными визуализациями и готовыми кодами.

Plotly

Интерактивные графики и диаграммы, можно создаваться с использованием библиотеки [Plotly](https://plot.ly/python/) (<https://plot.ly/python/>). Это коммерческий продукт, но многие функции доступны для бесплатного использования.

Источники на русском языке

На русском языке можно посмотреть [здесь](https://pythonworld.ru/novosti-mira-python/scientific-graphics-in-python.html) (<https://pythonworld.ru/novosti-mira-python/scientific-graphics-in-python.html>) и [здесь](https://pyprog.pro/mpl/mpl_short_guide.html) (https://pyprog.pro/mpl/mpl_short_guide.html). Интересные примеры работы с с библиотеками Seaborn и Plotly можно найти в разделе [Открытого курса машинного обучения от Хабра](#). Ещё одно небольшое руководство по использованию Plotly можно найти по этой [ссылке](https://proglab.io/p/plotly) (<https://proglab.io/p/plotly>).

1. <http://matplotlib.org/gallery.html> (<http://matplotlib.org/gallery.html>) — тысячи примеров
2. https://pyprog.pro/mpl/mpl_main_components.html (https://pyprog.pro/mpl/mpl_main_components.html)
3. https://matplotlib.org/api/axes_api.html?highlight=axes#module-matplotlib.axes (https://matplotlib.org/api/axes_api.html?highlight=axes#module-matplotlib.axes) — полный список методов к объекту axes в официальной документации.
4. <http://matplotlib.org/gallery.html> (<http://matplotlib.org/gallery.html>) — тысячи примеров
5. http://matplotlib.org/users/text_intro.html (http://matplotlib.org/users/text_intro.html) — для просмотра более полной информации по работе с текстом
6. <https://habr.com/ru/post/468295/> (<https://habr.com/ru/post/468295/>)
7. https://s3.amazonaws.com/assets.datacamp.com/blog_assets/Python_Matplotlib_Cheat_Sheet.pdf (https://s3.amazonaws.com/assets.datacamp.com/blog_assets/Python_Matplotlib_Cheat_Sheet.pdf) — шпаргалка DataCamp в matplotlib
8. https://nbviewer.org/github/whitehorn/Scientific_graphics_in_python/blob/master/P1%20Chapter%201%20Pyplot.ipynb (https://nbviewer.org/github/whitehorn/Scientific_graphics_in_python/blob/master/P1%20Chapter%201%20Pyplot.ipynb)
9. <https://jenyay.net/Matplotlib/Matplotlib> (<https://jenyay.net/Matplotlib/Matplotlib>).

У функций в `matplotlib` много параметров. Для того, чтобы **посмотреть все параметры**, можно воспользоваться справкой, например,

```
plt.plot?
```

Ввод []:

```
plt.plot?
```