



# Библиотека matplotlib

Devpractice Team



**УДК 004.4+004.6**

**ББК: 32.973**

Devpractice Team. Библиотека Matplotlib. - devpractice.ru. 2019. - 100 с.: ил.

В книге "Библиотека *Matplotlib*" в форме уроков дана информация, которая поможет решить большую часть задач, возникающих при работе с графиками. Рассмотрены такие темы как визуализация данных: построение линейных, точечных, ступенчатых, *stem*-графиков, столбчатых и круговых диаграмм, а также 3D графиков; настройка внешнего вида графиков: работа с легендой, текстовыми и *colorbar* элементами, компоновка графиков.

УДК 004.4+004.6

ББК: 32.973

*Материал составил и подготовил:*

Абдрахманов М.И.

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Материал, представленный в книге, многократно проверен. Но поскольку человеческие и технические ошибки все же возможны, авторы и коллектив проекта *devpractice.ru* не несут ответственности за возможные ошибки и последствия, связанные с использованием материалов из данной книги.

© devpractice.ru, 2019

© Абдрахманов М.И., 2019

Предлагаем познакомиться с другими нашими проектами.

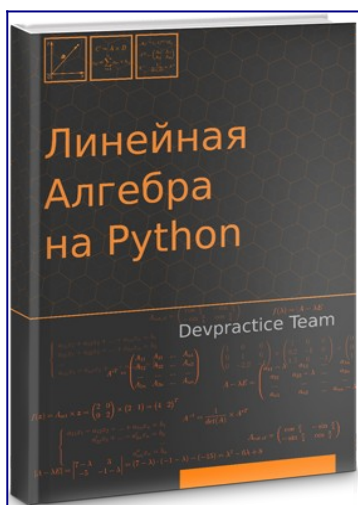
### "Pandas. Работа с данными"

Книга посвящена библиотеке для работы с данным *pandas*. Помимо базовых знаний о структурах *pandas*, вы получите информацию о том как работать с временными рядами, считать статистики, визуализировать данные и т.д. Большое внимание уделено практике, все рассматриваемые возможности библиотеки сопровождаются подробными примерами.



### "Линейная алгебра на Python"

Книга "Линейная алгебра на *Python*" - это попытка соединить две области: математику и программирование. В ней вы познакомитесь с базовыми разделами линейной алгебры и прекрасным инструментом для решения задач - языком программирования *Python*. Основные разделы книги посвящены матрицам и их свойствам, решению систем линейных уравнений, векторам, разложению матриц и комплексным числам.



## Оглавление

Урок 1. Быстрый старт.....	6
1.1 Установка.....	6
1.1.2 Варианты установки <i>Matplotlib</i> .....	6
1.1.3 Установка <i>Matplotlib</i> через менеджер <i>pip</i> .....	6
1.1.4 Проверка установки.....	6
1.2 Быстрый старт.....	7
1.3 Построение графика.....	9
1.4 Несколько графиков на одном поле.....	11
1.5 Представление графиков на разных полях.....	12
1.6 Построение диаграммы для категориальных данных.....	14
1.7 Основные элементы графика.....	15
Урок 2. Основы работы модулем <i>pyplot</i> .....	19
2.1 Построение графиков.....	19
2.2 Текстовые надписи на графике.....	21
2.2.1 Наименование осей.....	21
2.2.2 Заголовок графика.....	22
2.2.3 Текстовое примечание.....	23
2.2.4 Легенда.....	23
2.3 Работа с линейным графиком.....	25
2.3.1 Стилль линии графика.....	25
2.3.2 Цвет линии.....	29
2.3.3 Тип графика.....	30
2.4 Размещение графиков отдельно друг от друга.....	32
2.4.1 Работа с функцией <i>subplot()</i> .....	32
2.4.2 Работа с функцией <i>subplots()</i> .....	35
Урок 3. Настройка элементов графика.....	37
3.1 Работа с легендой.....	37

3.1.1	Отображение легенды.....	37
3.1.2	Расположение легенды на графике.....	39
3.1.3	Дополнительные параметры настройки легенды.....	42
3.2	Компоновка графиков.....	44
3.2.1	Инструмент <i>GridSpec</i> .....	44
3.3	Текстовые элементы графика.....	50
3.3.1	Заголовок фигуры и поля графика.....	52
3.3.2	Подписи осей графика.....	53
3.3.3	Текстовый блок.....	55
3.3.4	Аннотация.....	57
3.4	Свойства класса <i>Text</i> .....	64
3.4.1	Параметры, отвечающие за отображения текста.....	64
3.4.2	Параметры, отвечающие за расположение надписи.....	67
3.4.3	Параметры, отвечающие за настройку заднего фона надписи.....	68
3.5	Цветовая полоса — <i>Colorbar</i> .....	71
3.5.1	Общая настройка с использованием <i>inset_locator()</i> .....	73
3.5.2	Задание шкалы и установка надписи.....	75
3.5.4	Дополнительные параметры настройки <i>colorbar</i> .....	76
Урок 4	Визуализация данных.....	78
4.1	Линейный график.....	78
4.1.1	Построение графика.....	78
4.1.1.1	Параметры аргумента <i>fmt</i> .....	79
4.1.2	Заливка области между графиком и осью.....	82
4.1.3	Настройка маркировки графиков.....	86
4.1.4	Обрезка графика.....	89
4.2	Ступенчатый, стековый, точечный график и другие.....	90
4.2.1	Ступенчатый график.....	90
4.2.2	Стековый график.....	91
4.2.3	<i>Stem</i> -график.....	92

4.2.4 Точечный график.....	95
4.2.5 Дополнительные варианты работы с точечными графиками...	98
4.3 Столбчатые и круговые диаграммы.....	99
4.3.1 Столбчатые диаграммы.....	99
4.3.1.1 Групповые столбчатые диаграммы.....	102
4.3.1.2 Диаграмма с <i>errorbar</i> элементом.....	103
4.3.2 Круговые диаграммы.....	104
4.3.2.1 Классическая круговая диаграмма.....	104
4.3.2.2 Вложенные круговые диаграммы.....	108
4.3.2.3 Круговая диаграмма с отверстием.....	109
4.4 Цветовая сетка.....	110
4.4.1 Цветовые карты ( <i>colormaps</i> ).....	110
4.4.2 Построение цветовой сетки.....	111
<i>imshow()</i> .....	111
<i>pcolormesh()</i> .....	114
Урок 5. Построение 3D графиков. Работа с <i>mplot3d Toolkit</i> .....	117
5.1 Линейный график.....	117
5.2 Точечный график.....	119
5.3 Каркасная поверхность.....	121
5.4 Поверхность.....	123

# Урок 1. Быстрый старт

## 1.1 Установка

### 1.1.2 Варианты установки *Matplotlib*

Существует два основных варианта установки *Matplotlib* на ваш компьютер: в первом случае вы устанавливаете пакет *Anaconda*, в состав которого входит интересующая нас библиотека, во втором - устанавливаете *Matplotlib* самостоятельно, используя менеджер пакетов. Про установку *Anaconda* вы можете прочитать в статье “*Python. Урок 1. Установка*” (<https://devpractice.ru/python-lesson-1-install/>).

### 1.1.3 Установка *Matplotlib* через менеджер *pip*

Для установки *Matplotlib* через менеджер пакетов *pip* введите в командной строке вашей операционной системы следующие команды:

```
python -m pip install -U pip
python -m pip install -U matplotlib
```

Первая из них обновит ваш *pip*, вторая установит *Matplotlib* со всеми необходимыми зависимостями.

### 1.1.4 Проверка установки

Для проверки того, что все установилось правильно, запустите интерпретатор *Python* и введите в нем следующее:

```
>>> import matplotlib
```

Если сообщений об ошибке не было, то значит библиотека *Matplotlib* установлена и ее можно использовать. После этого можете проверить

версию библиотеки (она скорее всего будет отличаться от приведенной ниже):

```
>>> matplotlib.__version__  
'3.0.3'
```

## 1.2 Быстрый старт

Перед тем как углубиться в детали библиотеки *Matplotlib*, рассмотрим несколько примеров, изучив которые вы сможете использовать библиотеку для решения своих задач и получите интуитивное понимание принципов работы с этим инструментом.

Если вы работаете в *Jupyter Notebook*, то для того, чтобы получать графики рядом с ячейками с кодом необходимо выполнить специальную *magic* команду после импорта *Matplotlib*:

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

Результат работы будет выглядеть так, как показано на рисунке ниже.

```
In [1]: import matplotlib.pyplot as plt  
%matplotlib inline  
  
In [2]: plt.plot([1, 2, 3, 4, 5], [1, 2, 3, 4, 5])  
Out[2]: [matplotlib.lines.Line2D at 0x1a4f1dadd30]
```

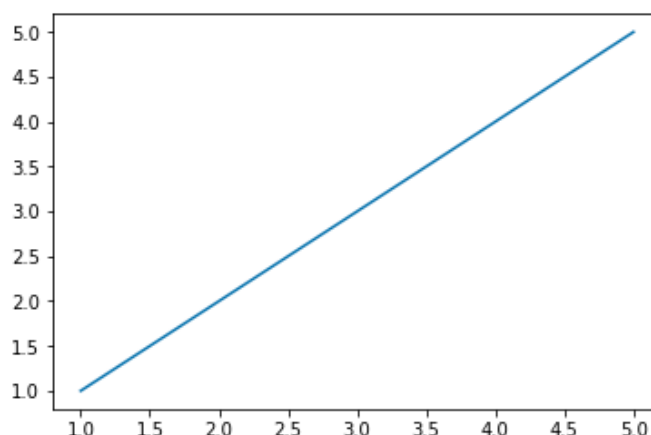


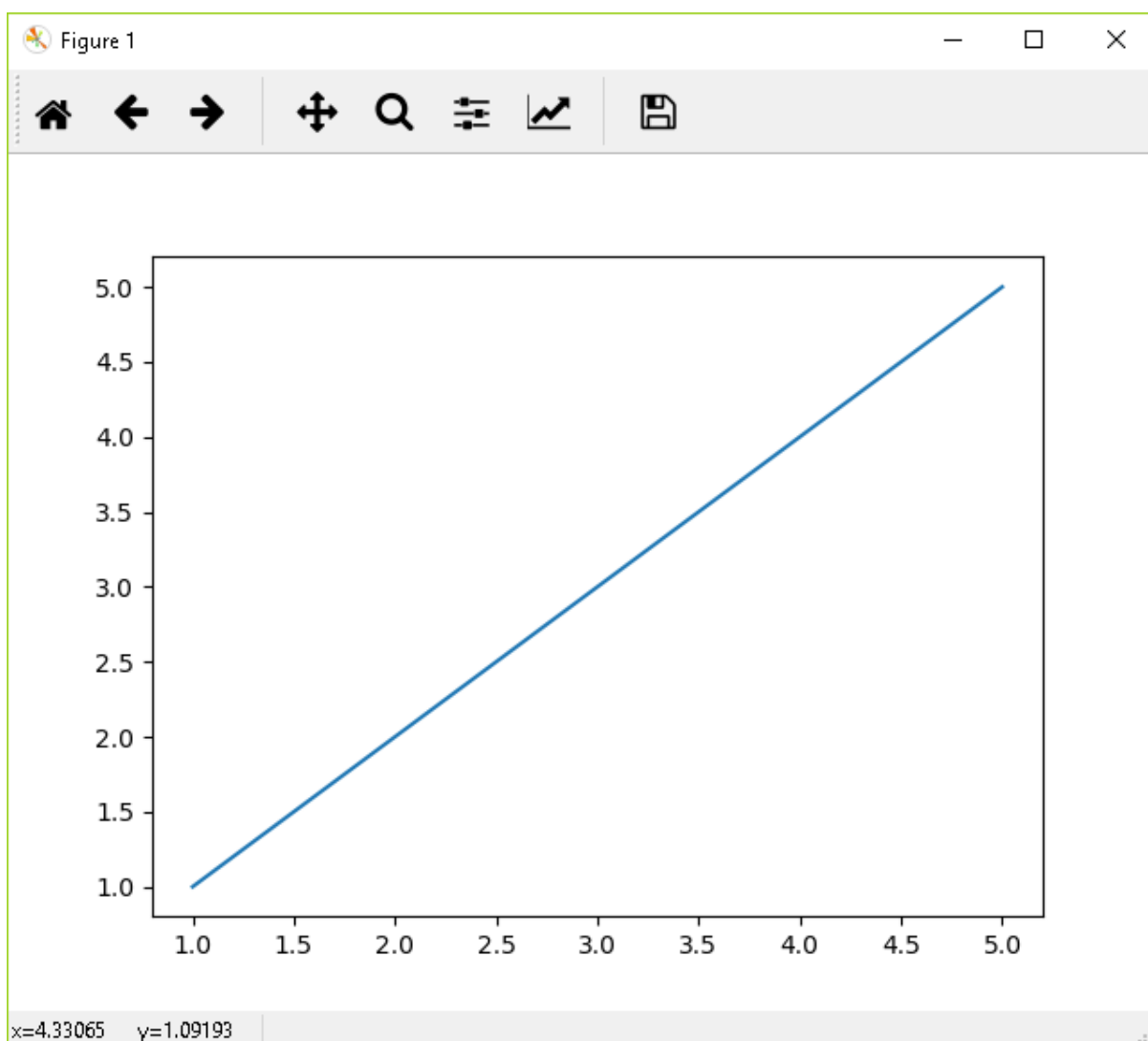
Рисунок 1.1 — Пример работы с *Matplotlib* в *Anaconda*



Если вы пишете код в `.py` файле, а потом запускаете его через вызов интерпретатора *Python*, то строка `%matplotlib inline` вам не нужна, используйте только импорт библиотеки. Пример, аналогичный тому, что представлен на рисунке выше, для отдельного *Python*-файла будет выглядеть так:

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

В результате получите график в отдельном окне.



**Рисунок 1.2 — Изображение графика в отдельном окне**

Далее мы не будем останавливаться на особенностях использования *magic* команды, просто запомните, что если вы используете *Jupyter notebook* при работе с *Matplotlib* вам обязательно нужно выполнить команду `%matplotlib inline`.

Теперь перейдем непосредственно к *Matplotlib*. Задача урока “*Быстрый старт*” - это построить разные типы графиков, настроить их внешний вид и освоиться в работе с этим инструментом.

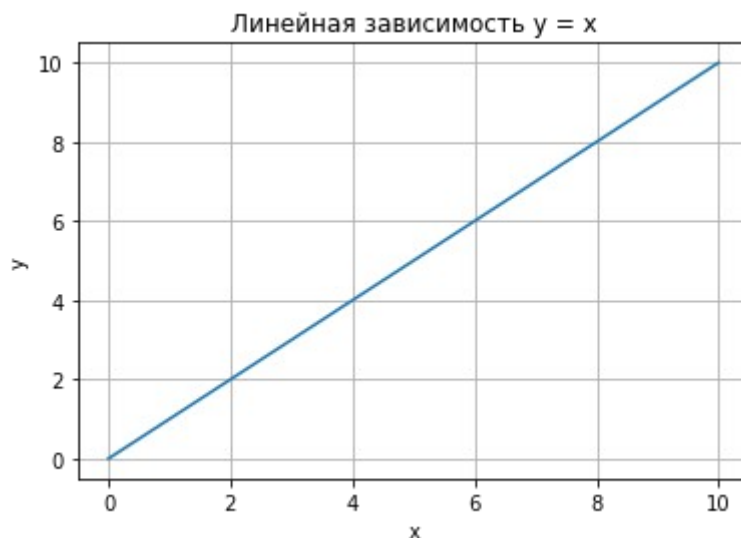
## 1.3 Построение графика

Для начал построим простую линейную зависимость, дадим нашему графику название, подпишем оси и отобразим сетку. Код программы:

```
# Независимая (x) и зависимая (y) переменные
x = np.linspace(0, 10, 50)
y = x

# Построение графика
plt.title('Линейная зависимость y = x') # заголовок
plt.xlabel('x') # ось абсцисс
plt.ylabel('y') # ось ординат
plt.grid()      # включение отображение сетки
plt.plot(x, y)  # построение графика
```

В результате получим следующий график:



**Рисунок 1.3 — Линейный график**

Изменим тип линии и ее цвет, для этого в функцию `plot()`, в качестве третьего параметра, передадим строку, сформированную определенным образом, в нашем случае это `'r--'`, где `'r'` означает красный цвет, а `'--'` - тип линии - пунктирная линия. Более подробно о том, как задавать цвет и тип линии будет рассказано с одним из следующих уроков.

*# Построение графика*

```
plt.title('Линейная зависимость y = x') # заголовок
```

```
plt.xlabel('x') # ось абсцисс
```

```
plt.ylabel('y') # ось ординат
```

```
plt.grid() # включение отображение сетки
```

```
plt.plot(x, y, 'r--') # построение графика
```

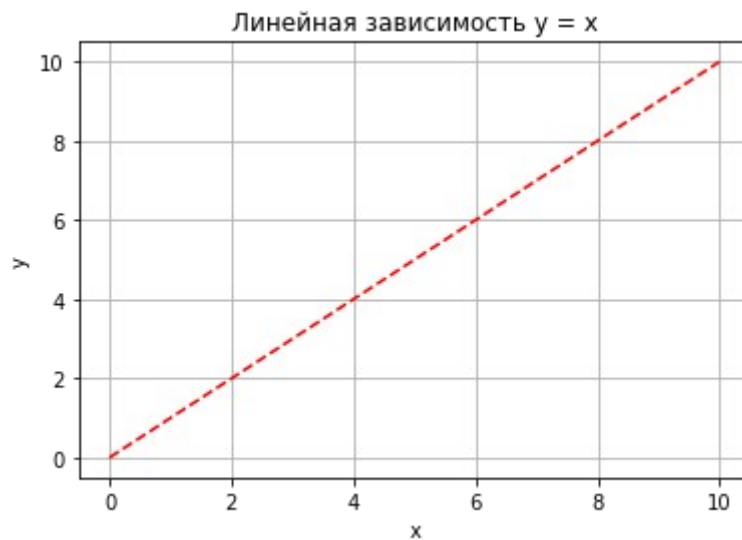


Рисунок 1.4 — Измененный линейный график

## 1.4 Несколько графиков на одном поле

Построим несколько графиков на одном поле, для этого добавим квадратичную зависимость:

*# Линейная зависимость*

```
x = np.linspace(0, 10, 50)
```

```
y1 = x
```

*# Квадратичная зависимость*

```
y2 = [i**2 for i in x]
```

*# Построение графика*

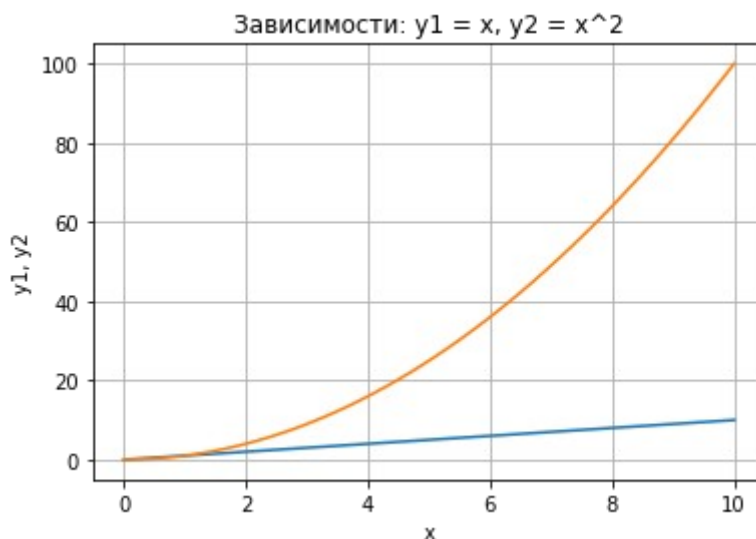
```
plt.title('Зависимости:  $y_1 = x$ ,  $y_2 = x^2$ ') # заголовок
```

```
plt.xlabel('x') # ось абсцисс
```

```
plt.ylabel('y1, y2') # ось ординат
```

```
plt.grid() # включение отображение сетки
```

```
plt.plot(x, y1, x, y2) # построение графика
```



**Рисунок 1.5 — Несколько графиков на одном поле**

В приведенном примере в функцию `plot()` последовательно передаются два массива для построения первого графика и два для построения второго, при этом, как вы можете заметить, для обоих графиков массив значений независимой переменной `x` один и то же.

## 1.5 Представление графиков на разных полях

Третья, довольно часто встречающаяся задача - это отобразить два или более различных поля, на которых будет отображено по одному или более графику. Построим уже известные нам две зависимости на разных полях:

```
# Линейная зависимость
x = np.linspace(0, 10, 50)
y1 = x

# Квадратичная зависимость
y2 = [i**2 for i in x]
```

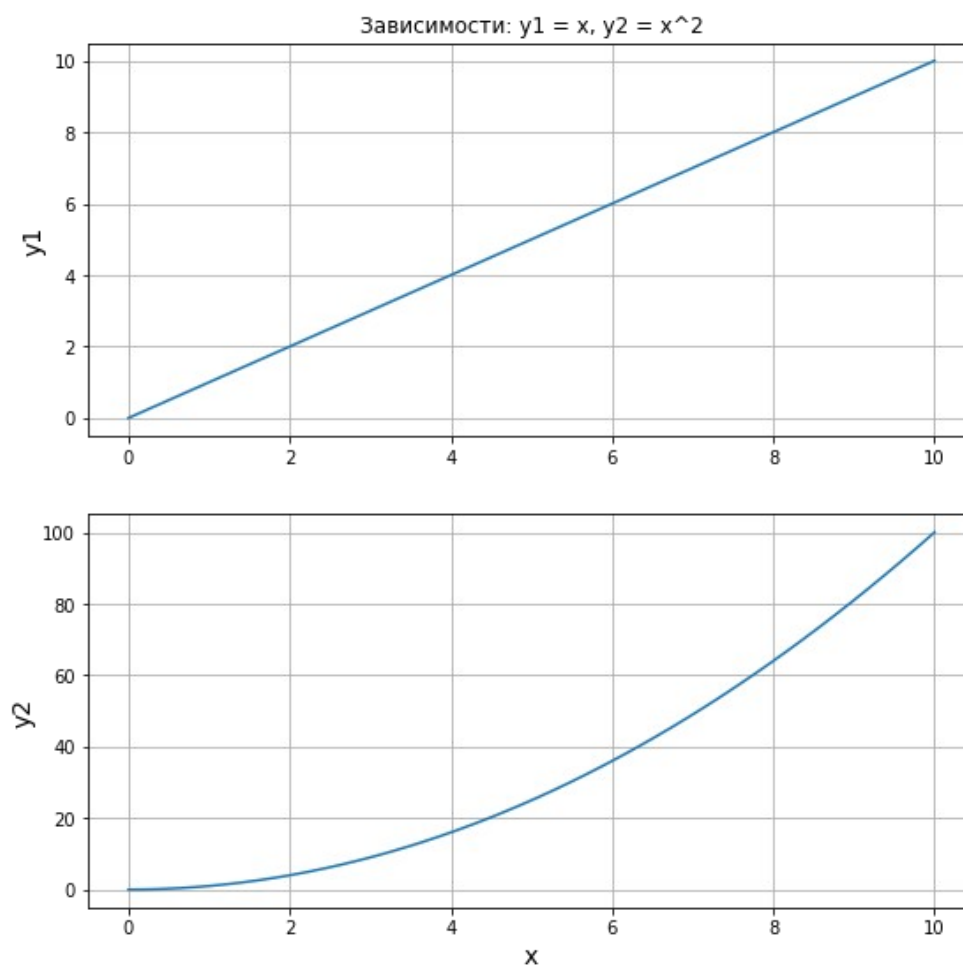
```

# Построение графиков
plt.figure(figsize=(9, 9))

plt.subplot(2, 1, 1)
plt.plot(x, y1)                                # построение графика
plt.title('Зависимости:  $y_1 = x$ ,  $y_2 = x^2$ ') # заголовок
plt.ylabel('y1', fontsize=14)                  # ось ординат
plt.grid(True)                                 # включение отображение сетки

plt.subplot(2, 1, 2)
plt.plot(x, y2)                                # построение графика
plt.xlabel('x', fontsize=14)                   # ось абсцисс
plt.ylabel('y2', fontsize=14)                 # ось ординат
plt.grid(True)                                 # включение отображение сетки

```



**Рисунок 1.6 — Разделенные поля с графиками**

Здесь мы воспользовались новыми функциями:

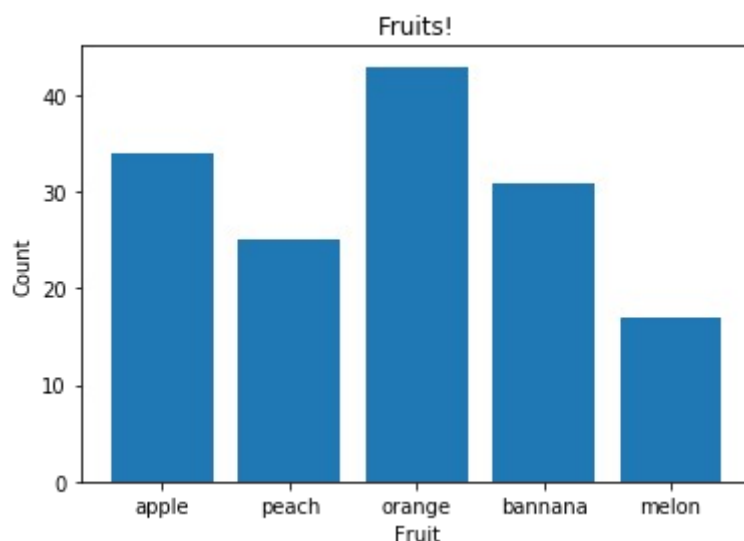
- `figure()` - функция для задания глобальных параметров отображения графиков. В нее, в качестве аргумента, мы передаем кортеж, определяющий размер общего поля.
- `subplot()` - функция для задания местоположения поля с графиком. Существует несколько способов задания областей для вывода графиков. В примере мы воспользовались вариантом, который предполагает передачу трех аргументов: первый аргумент - количество строк, второй - столбцов в формируемом поле, третий - индекс (номер поля, считаем сверху вниз, слева направо).

Остальные функции вам знакомы, дополнительно мы использовали параметр `fontsize` функций `xlabel()` и `ylabel()` для задания размера шрифта.

## 1.6 Построение диаграммы для категориальных данных

До этого мы строили графики для численных данных, то есть зависимая и независимая переменные имели числовой тип. На практике довольно часто приходится работать с данными не числовой природы - имена людей, название городов и т. п. Построим диаграмму, на которой будет отображаться количество фруктов в магазине:

```
fruits = ['apple', 'peach', 'orange', 'bannana', 'melon']
counts = [34, 25, 43, 31, 17]
plt.bar(fruits, counts)
plt.title('Fruits!')
plt.xlabel('Fruit')
plt.ylabel('Count')
```



**Рисунок 1.7 — Столбчатая диаграмма**

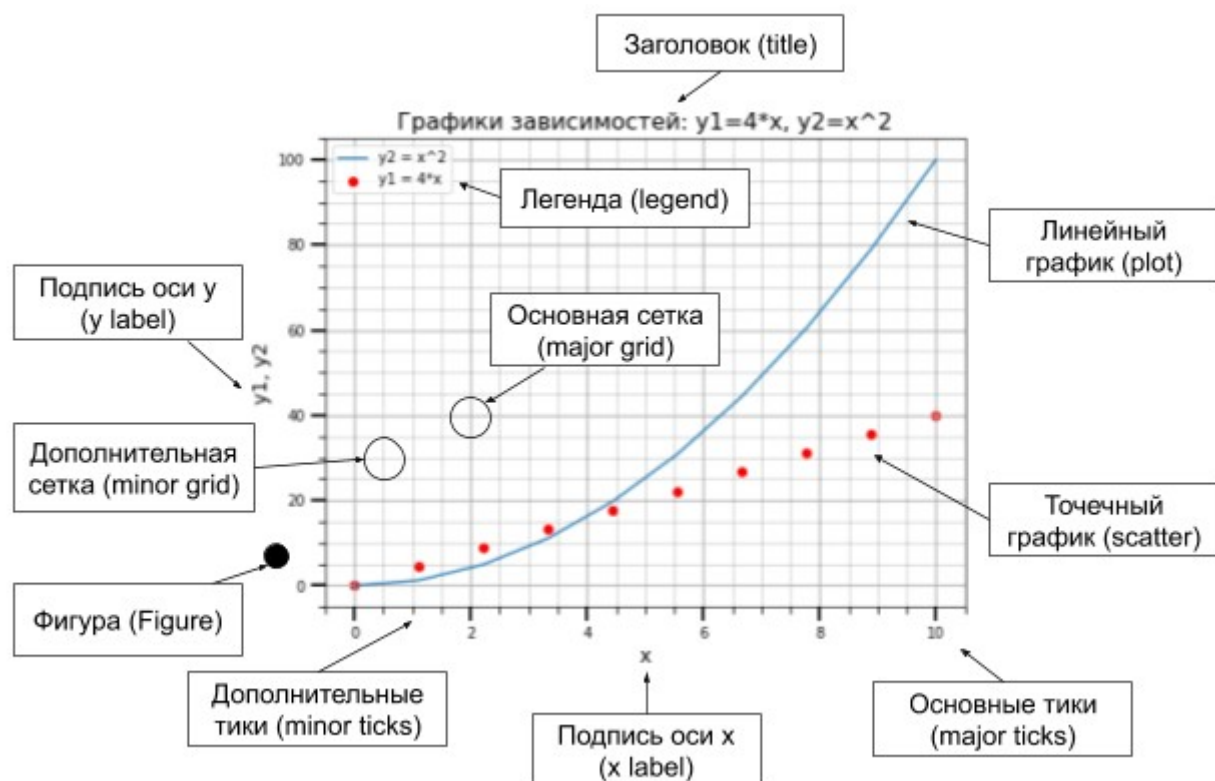
Для вывода диаграммы мы использовали функцию `bar()`.

К этому моменту, если вы самостоятельно попробовали запустить приведенные выше примеры, у вас уже должно сформировать некоторое понимание того, как осуществляется работа с библиотекой *Matplotlib*.

## 1.7 Основные элементы графика

Рассмотрим основные термины и понятия, касающиеся изображения графика, с которыми вам необходимо будет познакомиться, для того, чтобы в дальнейшем у вас не было трудностей при изучении представленных здесь уроков и документации по библиотеке *Matplotlib*.





**Рисунок 1.8 — Основные элементы графика**

Корневым элементом при построении графиков в системе *Matplotlib* является Фигура (*Figure*). Все, что нарисовано на рисунке выше является элементами фигуры. Рассмотрим ее составляющие более подробно.

## График

На рисунке представлены два графика - линейный и точечный. *Matplotlib* предоставляет огромное количество различных настроек, которые можно использовать для того, чтобы придать графику требуемый вид: задать цвет, толщину, тип, стиль линии и многое другое, все это мы рассмотрим в ближайших уроках.

## Оси

Вторым, после непосредственно самого графика, по важности элементом фигуры являются оси. Для каждой оси можно задать метку (подпись), основные (*major*) и дополнительные (*minor*) элементы шкалы, их подписи, размер, толщину и диапазоны.

## Сетка и легенда

Сетка и легенда являются элементами фигуры, которые значительно повышают информативность графика. Сетка может быть основной (*major*) и дополнительной (*minor*). Каждому типу сетки можно задавать цвет, толщину линии и тип. Для отображения сетки и легенды используются соответствующие команды.

Ниже представлен код, с помощью которого была построена фигура, изображенная на рисунке 1.8:

```
import matplotlib.pyplot as plt
from matplotlib.ticker import (MultipleLocator, FormatStrFormatter,
AutoMinorLocator)
import numpy as np
x = np.linspace(0, 10, 10)
y1 = 4*x
y2 = [i**2 for i in x]

fig, ax = plt.subplots(figsize=(8, 6))

ax.set_title('Графики зависимостей: y1=4*x, y2=x^2', fontsize=16)
ax.set_xlabel('x', fontsize=14)
ax.set_ylabel('y1, y2', fontsize=14)
ax.grid(which='major', linewidth=1.2)
ax.grid(which='minor', linestyle='--', color='gray', linewidth=0.5)
```

```
ax.scatter(x, y1, c='red', label='y1 = 4*x')
ax.plot(x, y2, label='y2 = x^2')
ax.legend()

ax.xaxis.set_minor_locator(AutoMinorLocator())
ax.yaxis.set_minor_locator(AutoMinorLocator())
ax.tick_params(which='major', length=10, width=2)
ax.tick_params(which='minor', length=5, width=1)

plt.show()
```

Если в данный момент вам что-то кажется непонятным - не переживайте, далее мы разберем подробно особенности настройки и использования всех элементов, представленных на поле с графиками.

## Урок 2. Основы работы модулем pyplot

Практически все задачи, связанные с построением графиков, можно решить, используя возможности, которые предоставляет модуль pyplot. Для того, чтобы запустить любой из примеров, продемонстрированных в первом уроке, вам предварительно нужно было импортировать pyplot из библиотеки *Matplotlib*. В настоящее время, среди пользователей, принято импорт модуля pyplot производить следующим образом:

```
import matplotlib.pyplot as plt
```

Создатели *Matplotlib* постарались сделать его похожим в использовании на *MATLAB*, так что если вы знакомы с последним, то разобраться с *Matplotlib* будет проще.

### 2.1 Построение графиков

Основным элементом изображения, которое строит pyplot является Фигура (*Figure*), на нее накладываются один или более графиков, осей, надписей и т.д. Для построения графика используется команда plot(). В самом минимальном варианте можно ее использовать без параметров:

```
import matplotlib.pyplot as plt
%matplotlib inline
plt.plot()
```

В результате будет выведено пустое поле (см. рисунок 2.1).

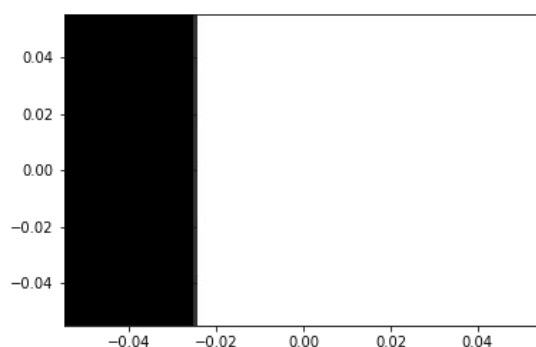
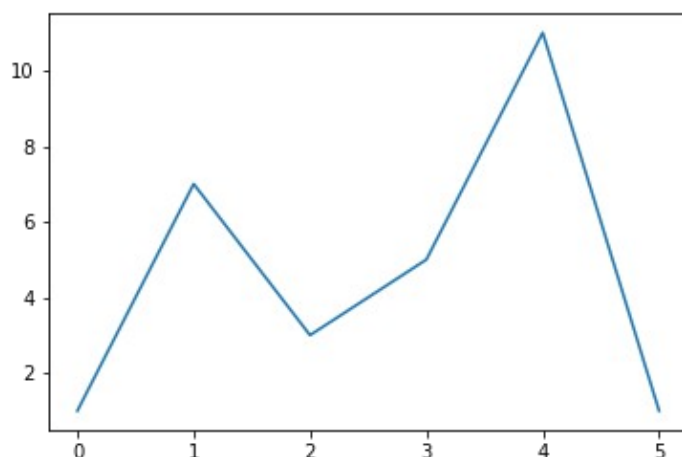


Рисунок 2.1 — Пустое поле

Далее команду импорта и *magic*-команду для *Jupyter* (первая и вторая строки приведенной выше программы) мы не будем указывать.

Если в качестве параметра функции `plot()` передать список, то значения из этого списка будут отложены по оси ординат (ось *y*), а по оси абсцисс (ось *x*) будут отложены индексы элементов массива:

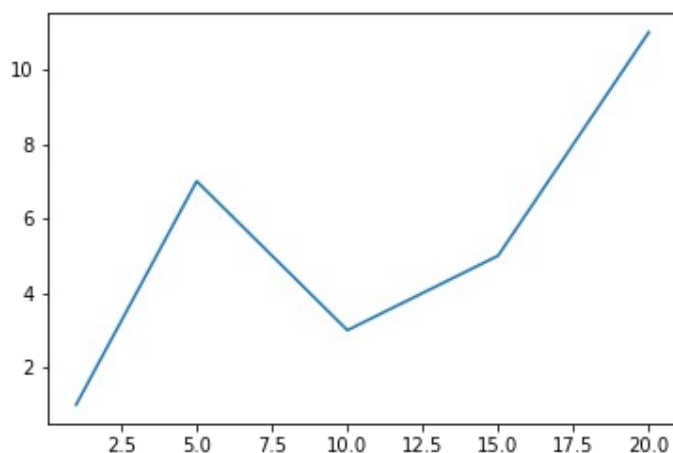
```
plt.plot([1, 7, 3, 5, 11, 1])
```



**Рисунок 2.2 — Линейный график, построенный по значениям *y***

Для того, чтобы задать значения по осям *x* и *y* необходимо в `plot()` передать два списка:

```
plt.plot([1, 5, 10, 15, 20], [1, 7, 3, 5, 11])
```



**Рисунок 2.3 — Линейный график, построенный по значениям *y* и *x***

## 2.2 Текстовые надписи на графике

Наиболее часто используемые текстовые надписи на графике это:

- наименования осей;
- наименование самого графика;
- текстовые примечания на поле с графиком;
- легенда.

Рассмотрим кратко данные элементы, более подробный рассказ о них будет в “3.3 Текстовые элементы графика”.

### 2.2.1 Наименование осей

Для задания подписи оси *x* используется функция `xlabel()`, оси *y* - `ylabel()`. Разберемся с аргументами данных функций. Здесь и далее аргументы будем описывать следующим образом:

- **имя\_аргумента: тип(ы)**
  - **описание**

Для функций `xlabel()/ylabel()` основными являются следующие аргументы:

- `xlabel` (или `ylabel`): `str`
  - Текст подписи.
- `labelpad`: численное значение либо `None`; значение по умолчанию: `None`
  - Расстояние между областью графика, включающую оси, и меткой.

Функции `xlabel()/ylabel()` принимают в качестве аргументов параметры конструктора класса `matplotlib.text.Text`, некоторые из них нам могут пригодиться:

- `fontsize` или `size`: число либо значение из списка: `{'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large'}`
  - Размер шрифта.
- `fontstyle`: значение из списка: `{'normal', 'italic', 'oblique'}`
  - Стилль шрифта.
- `fontweight`: число в диапазоне от 0 до 1000 либо значение из списка: `{'ultralight', 'light', 'normal', 'regular', 'book', 'medium', 'roman', 'semibold', 'demibold', 'demi', 'bold', 'heavy', 'extra bold', 'black'}`
  - Толщина шрифта.
- `color`: одни из доступных способов определения цвета см. “2.3.2 *Цвет линии*”.
  - Цвет шрифта.

Пример использования:

```
plt.xlabel('Day', fontsize=15, color='blue')
```

Нами была рассмотрена только часть аргументов функций `xlabel()/ylabel()`. Так как мы только начинаем знакомиться с инструментом `matplotlib`, приведенного списка будет достаточно.

## 2.2.2 Заголовок графика

Для задания заголовка графика используется функция `title()`:

```
plt.title('Chart price', fontsize=17)
```

Из ее параметров отметим следующие:

- `label: str`
  - Текст заголовка.
- `loc: значение из набора: {'center', 'left', 'right'}`
  - Выравнивание заголовка.

Для функции `title()` также доступны параметры конструктора класса `matplotlib.text.Text`, часть из них представлена в описании аргументов функций `xlabel()/ylabel()`.

### 2.2.3 Текстовое примечание

За размещение текста на поле графика отвечает функция `text()`, которой первым и вторым аргументами передаются координаты позиции надписи, после - текст самой надписи:

```
plt.text(1, 1, 'type: Steel')
```

Для более тонкой настройки внешнего вида текстового примечания используйте параметры конструктора класса `Text`.

### 2.2.4 Легенда

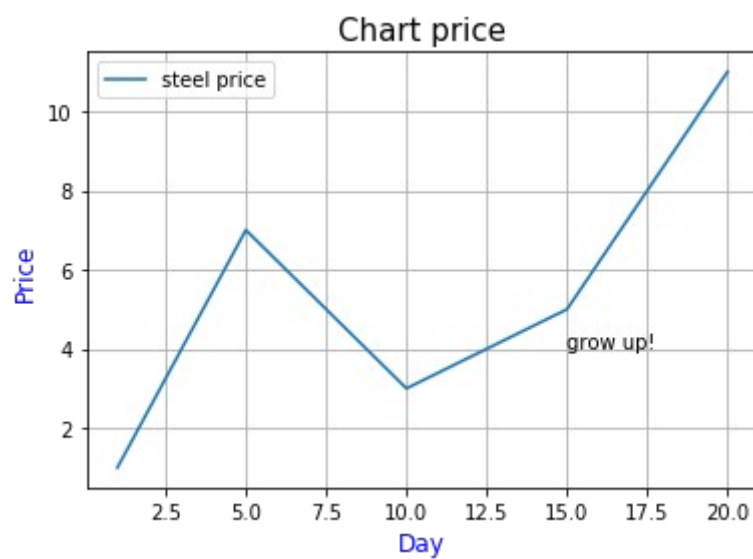
Легенда будет размещена на графике, если вызвать функцию `legend()`, в рамках данного урока мы не будем рассматривать аргументы этой функции.

Разместим на уже знакомом нам графике необходимый набор подписей:

```
x = [1, 5, 10, 15, 20]
y = [1, 7, 3, 5, 11]
plt.plot(x, y, label='steel price')
plt.title('Chart price', fontsize=15)
plt.xlabel('Day', fontsize=12, color='blue')
plt.ylabel('Price', fontsize=12, color='blue')
```



```
plt.legend()
plt.grid(True)
plt.text(15, 4, 'grow up!')
```



**Рисунок 2.4 — Текстовые надписи на графике**

К перечисленным опциям мы добавили сетку, которая включается с помощью функции `grid(True)`.

## 2.3 Работа с линейным графиком

В этом параграфе будут рассмотрены основные параметры для изменения внешнего вида линейного графика. *Matplotlib* предоставляет огромное количество инструментов для построения различных видов графиков. Так как наиболее часто встречающийся вид графика - это линейный, ему и уделим внимание.

Параметры, которые отвечают за отображение графика можно задать непосредственно в самой функции `plot()`:

```
plt.plot(x, y, color='red')
```

либо воспользоваться функцией `setp()`:

```
plt.setp(color='red', linewidth=1)
```

### 2.3.1 Стиль линии графика

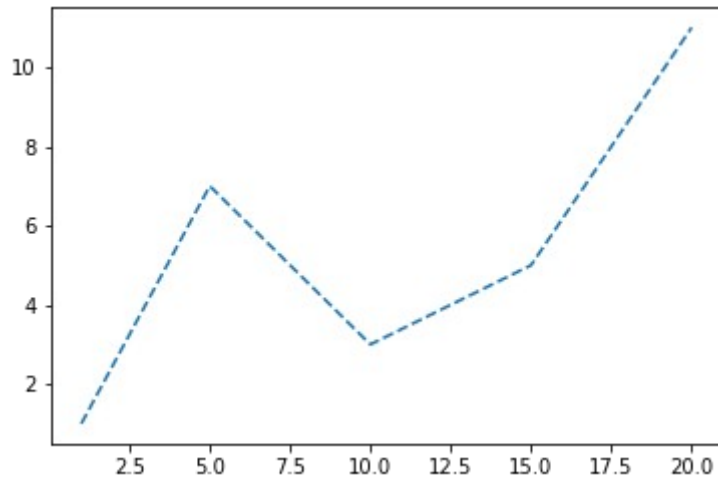
Стиль линии графика задается через параметр `linestyle`, который может принимать значения из приведенной ниже таблицы.

**Таблица 2.1 — Стили линии линейного графика**

Значение параметра	Описание
'-' или 'solid'	Непрерывная линия
'--' или 'dashed'	Штриховая линия
'-.' или 'dashdot'	Штрихпунктирная линия
':' или 'dotted'	Пунктирная линия
'None' или ' ' или ''	Не отображать линию

Стиль линии можно передать сразу после списков с координатами без указания, что это параметр `linestyle`:

```
x = [1, 5, 10, 15, 20]
y = [1, 7, 3, 5, 11]
plt.plot(x, y, '--')
```



**Рисунок 2.5 — Линия с примененным стилем**

Другой вариант - это воспользоваться функцией `setp()`:

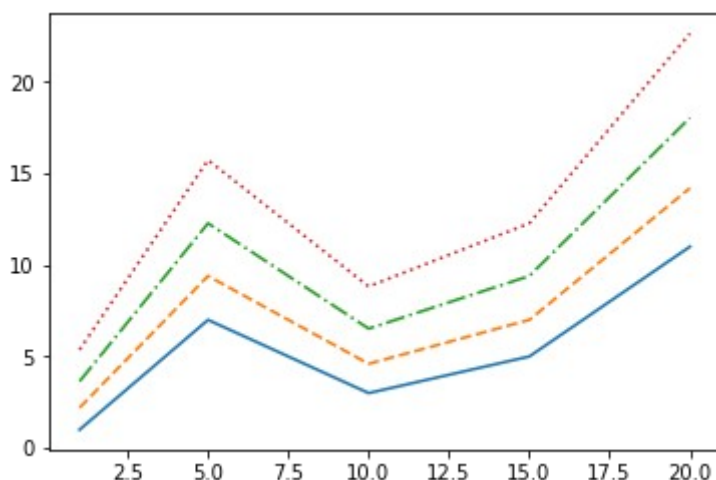
```
x = [1, 5, 10, 15, 20]
y = [1, 7, 3, 5, 11]
line = plt.plot(x, y)
plt.setp(line, linestyle='--')
```

Результат будет тот же, что на рисунке выше.

Для того, чтобы вывести несколько графиков на одном поле необходимо передать соответствующие наборы значений в функцию `plot()`.

Построим несколько наборов данных и выведем их с использованием различных стилей линии:

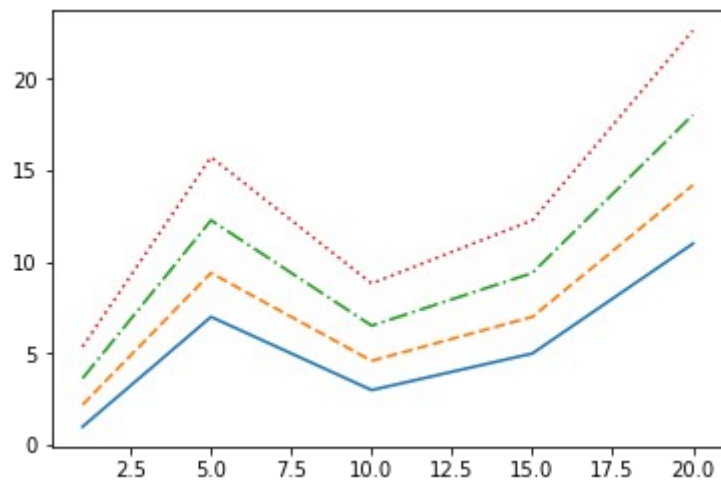
```
x = [1, 5, 10, 15, 20]
y1 = [1, 7, 3, 5, 11]
y2 = [i*1.2 + 1 for i in y1]
y3 = [i*1.2 + 1 for i in y2]
y4 = [i*1.2 + 1 for i in y3]
plt.plot(x, y1, '-', x, y2, '--', x, y3, '-.', x, y4, ':')
```



**Рисунок 2.6 — Несколько графиков, построенных одной функцией `plot()`**

Тот же результат можно получить вызвав `plot()` для построения каждого графика по отдельности. Если вы хотите представить каждый график отдельно на своем поле, то используйте для этого `subplot()` (см. “2.4.1 Работа с функцией `subplot()`”):

```
plt.plot(x, y1, '-')
plt.plot(x, y2, '--')
plt.plot(x, y3, '-.')
plt.plot(x, y4, ':')
```



**Рисунок 2.7 — Несколько графиков, построенных разными функциями plot()**

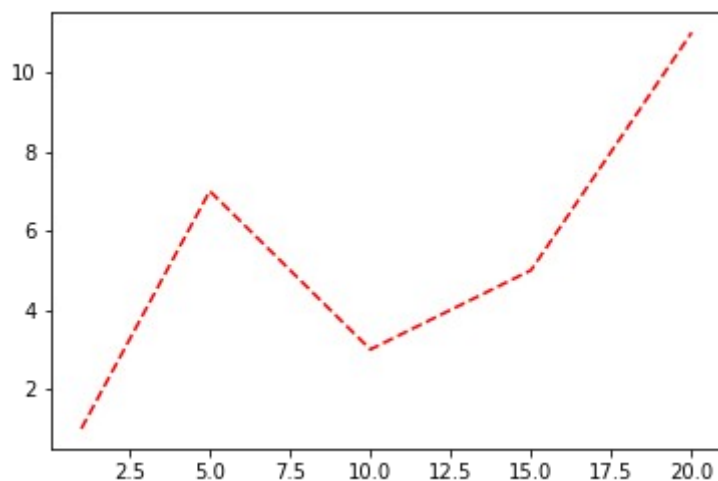
### 2.3.2 Цвет линии

Цвет линии графика задается через параметр `color` (или `c`, если использовать сокращенный вариант). Значение может быть представлено в одном из следующих форматов:

- *RGB* или *RGBA* кортеж значений с плавающей точкой в диапазоне  $[0, 1]$  (пример: `(0.1, 0.2, 0.3)`);
- *RGB* или *RGBA* значение в *hex* формате (пример: `'#0a0a0a'`);
- строковое представление числа с плавающей точкой в диапазоне  $[0, 1]$  (определяет цвет в шкале серого) (пример: `'0.7'`);
- символ из набора: `{'b', 'g', 'r', 'c', 'm', 'y', 'k', 'w'}`;
- имя цвета из палитры *X11/CSS4*;
- цвет из палитры *xkcd* (<https://xkcd.com/color/rgb/>), должен начинаться с префикса `'xkcd:'`;
- цвет из набора *Tableau Color* (палитра *T10*), должен начинаться с префикса `'tab:'`.

Если цвет задается с помощью символа из набора `{'b', 'g', 'r', 'c', 'm', 'y', 'k', 'w'}`, то он может быть совмещен со стилем линии в рамках параметра `fmt` функции `plot()`. Например: штриховая красная линия будет задаваться так: `'--r'`, а штрих пунктирная зеленая так `'-.g'`:

```
x = [1, 5, 10, 15, 20]
y = [1, 7, 3, 5, 11]
plt.plot(x, y, '--r')
```



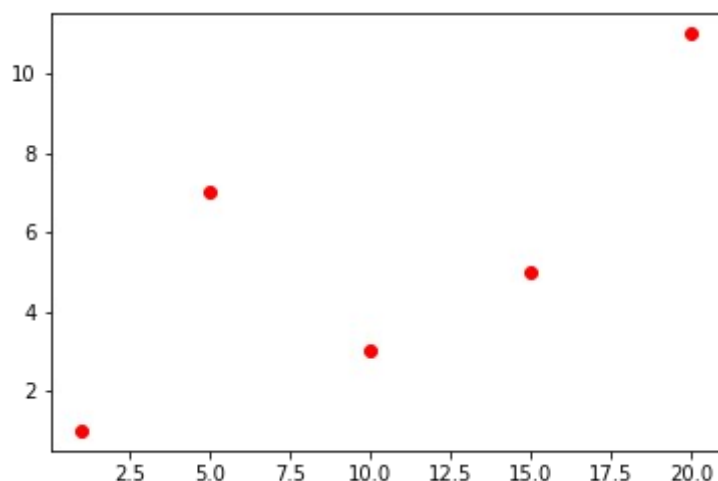
**Рисунок 2.8 — Изменение цвета линии**

### 2.3.3 Тип графика

До этого момента мы работали только с линейными графиками, функция `plot()` позволяет задать тип графика: линейный либо точечный. Для точечного графика нужно указать маркер, который будет использоваться для его вывода.

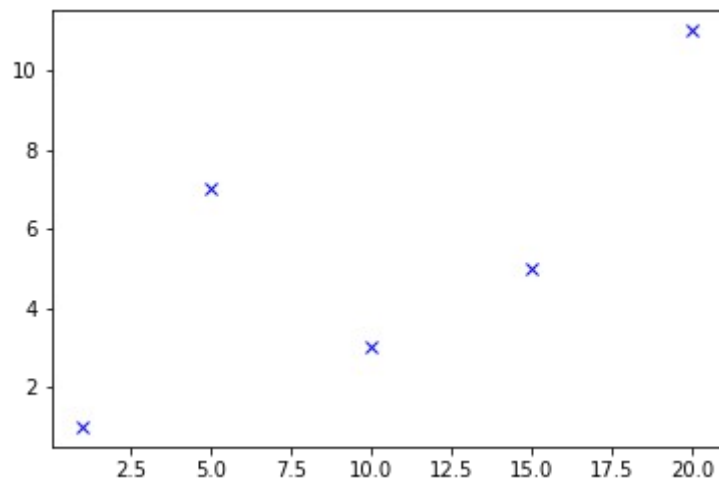
Приведем пару примеров:

```
plt.plot(x, y, 'ro')
```



**Рисунок 2.9 — График состоящий из круглых красных точек**

```
plt.plot(x, y, 'bx')
```



**Рисунок 2.10 — График состоящий из синих x-символов**

Размер маркера можно менять, об этом более подробно будет рассмотрено в уроке, посвященном точечным графикам.



## 2.4 Размещение графиков отдельно друг от друга

Существуют три основных подхода к размещению нескольких графиков на разных полях:

- использование функции `subplot()` для указания места размещения поля с графиком;
- использование функции `subplots()` для предварительного задания сетки, в которую будут укладываться поля;
- использование `GridSpec`, для более гибкого задания геометрии размещения полей с графиками в сетке.

В этом уроке будут рассмотрены первые два подхода.

### 2.4.1 Работа с функцией `subplot()`

Самый простой способ представить графики на отдельных полях - это использовать функцию `subplot()` для задания их мест размещения. До этого момента мы не работали с Фигурой (*Figure*) напрямую, значения ее параметров, задаваемые по умолчанию нас устраивали. Для решения текущей задачи придется один из параметров - размер подложки, задать вручную. За это отвечает аргумент `figsize` функции `figure()`, которому присваивается кортеж из двух `float` элементов, определяющих высоту и ширину подложки.

После задания размера, указывается местоположение: куда будет установлено поле с графиком с помощью функции `subplot()`.

Доступны следующие варианты вызова `subplot()`:

`subplot(nrows, ncols, index)`

- `nrows: int`
  - Количество строк.
- `ncols: int`
  - Количество столбцов.
- `index: int`
  - Местоположение элемента.

`subplot(pos)`

- `pos: int`
  - Позиция. Задается в виде трехзначного числа, содержащего информацию о количестве строк, столбцов и индексе, например: число 212 означает: подготовить разметку с двумя строками и одним столбцом, элемент вывести в первую позицию второй строки.

Второй вариант можно использовать, если количество строк и столбцов сетки не более 10, в ином случае, лучше обратиться к первому варианту.

Рассмотрим на примере работу с данными функциями:

*# Исходный набор данных*

```
x = [1, 5, 10, 15, 20]
```

```
y1 = [1, 7, 3, 5, 11]
```

```
y2 = [i*1.2 + 1 for i in y1]
```

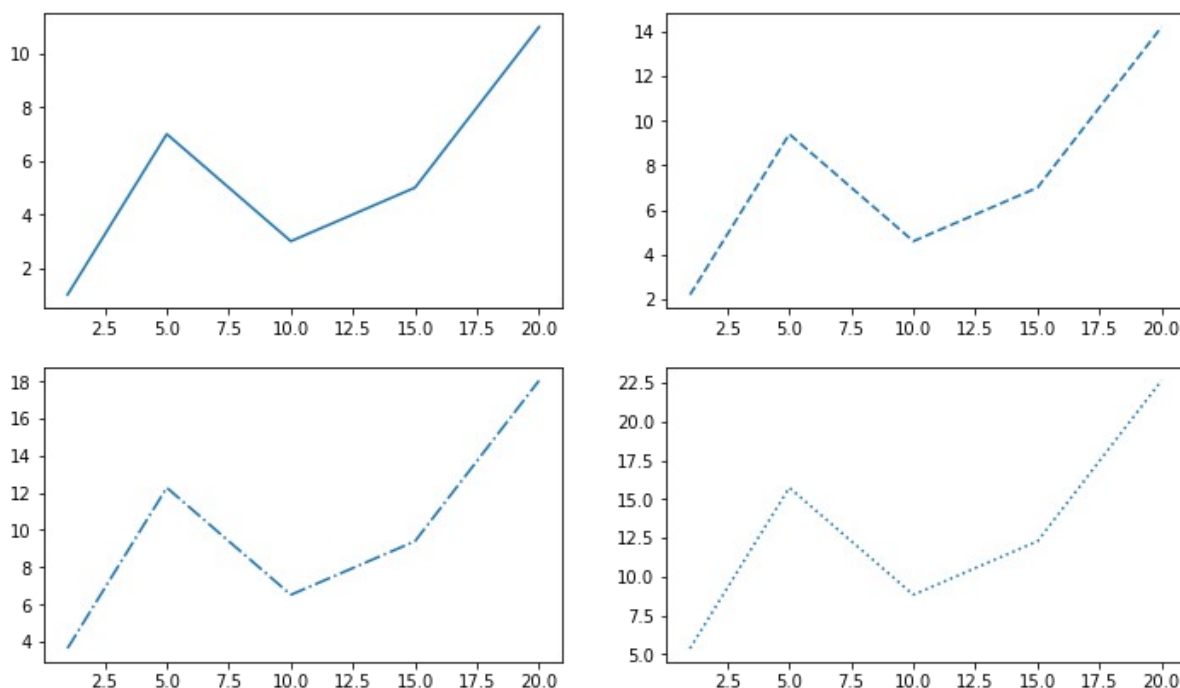
```
y3 = [i*1.2 + 1 for i in y2]
```

```
y4 = [i*1.2 + 1 for i in y3]
```

*# Настройка размеров подложки*

```
plt.figure(figsize=(12, 7))
```

```
# Вывод графиков
plt.subplot(2, 2, 1)
plt.plot(x, y1, '-')
plt.subplot(2, 2, 2)
plt.plot(x, y2, '--')
plt.subplot(2, 2, 3)
plt.plot(x, y3, '-.')
plt.subplot(2, 2, 4)
plt.plot(x, y4, ':')
```



**Рисунок 2.10 — Размещение графиков отдельно друг от друга (пример 1)**

Второй вариант использования `subplot()` будет выглядеть так:

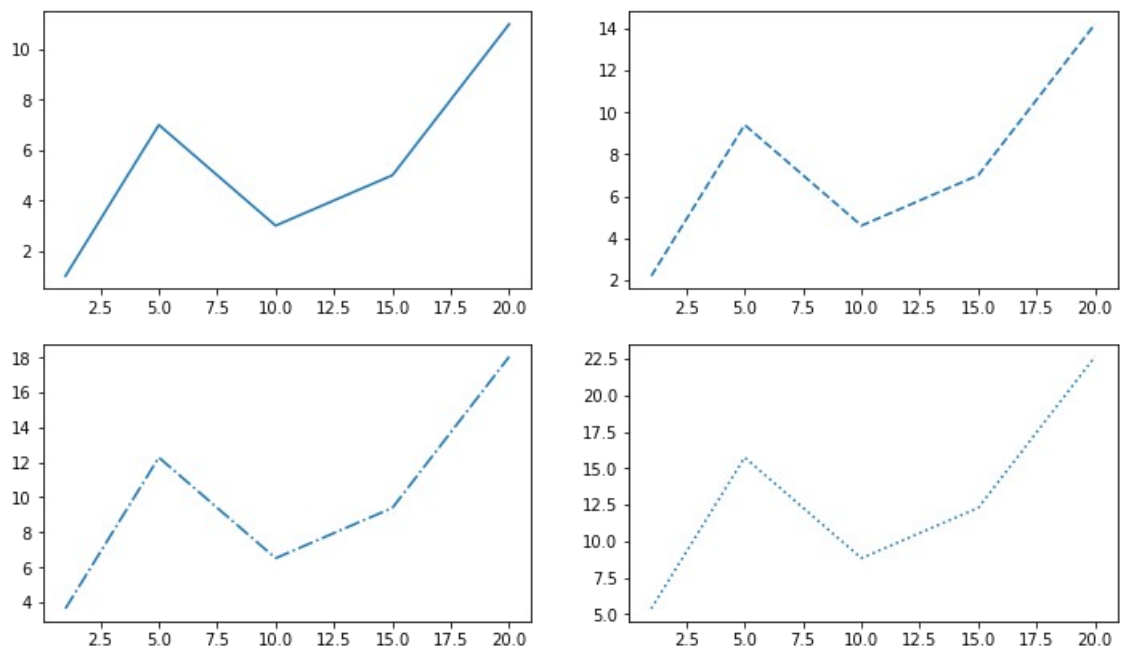
```
# Вывод графиков
plt.subplot(221)
plt.plot(x, y1, '-')
plt.subplot(222)
plt.plot(x, y2, '--')
plt.subplot(223)
plt.plot(x, y3, '-.')
plt.subplot(224)
plt.plot(x, y4, ':')
```

## 2.4.2 Работа с функцией `subplots()`

Неудобство использования последовательного вызова функций `subplot()` заключается в том, что каждый раз приходится указывать количество строк и столбцов сетки. Для того, чтобы этого избежать, можно воспользоваться функцией `subplots()`, из всех ее параметров нас интересуют только первые два, через них передается количество строк и столбцов сетки. Функция `subplots()` возвращает два объекта, первый - это *Figure*, подложка, на которой будут размещены поля с графиками, второй - объект (или массив объектов) *Axes*, через который можно получить полный доступ к настройке внешнего вида отображаемых элементов.

Решим задачу вывода четырех графиков с помощью функции `subplots()`:

```
fig, axs = plt.subplots(2, 2, figsize=(12, 7))
axs[0, 0].plot(x, y1, '-')
axs[0, 1].plot(x, y2, '--')
axs[1, 0].plot(x, y3, '-.')
axs[1, 1].plot(x, y4, ':')
```



**Рисунок 2.11 — Размещение графиков отдельно друг от друга (пример 2)**

# Урок 3. Настройка элементов графика

## 3.1 Работа с легендой

В данном параграфе будут рассмотрены следующие темы: отображение легенды, настройка ее расположения на графике, дополнительные параметры для более тонкой настройки ее внешнего вида.

### 3.1.1 Отображение легенды

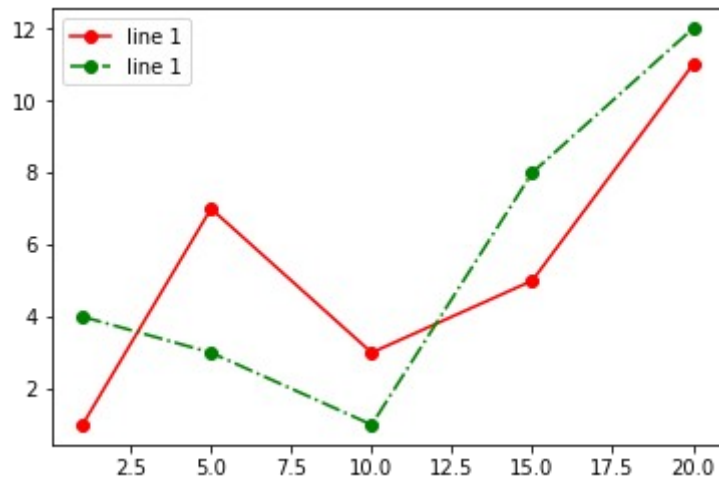
Для отображения легенды на графике используется функция `legend()`.

Возможны следующие варианты ее вызова:

```
legend()  
legend(labels)  
legend(handles, labels)
```

В первом варианте, в качестве меток для легенды, будут использоваться метки, указанные в функциях построения графиков (параметр `label`):

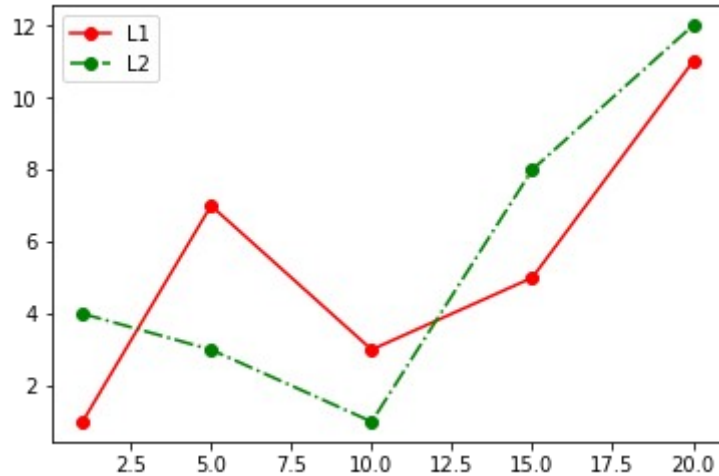
```
x = [1, 5, 10, 15, 20]  
y1 = [1, 7, 3, 5, 11]  
y2 = [4, 3, 1, 8, 12]  
plt.plot(x, y1, 'o-r', label='line 1')  
plt.plot(x, y2, 'o-.g', label='line 1')  
plt.legend()
```



**Рисунок 3.1 — Легенда на графике (пример 1)**

Второй вариант позволяет самостоятельно указать текстовую метку для отображаемых данных:

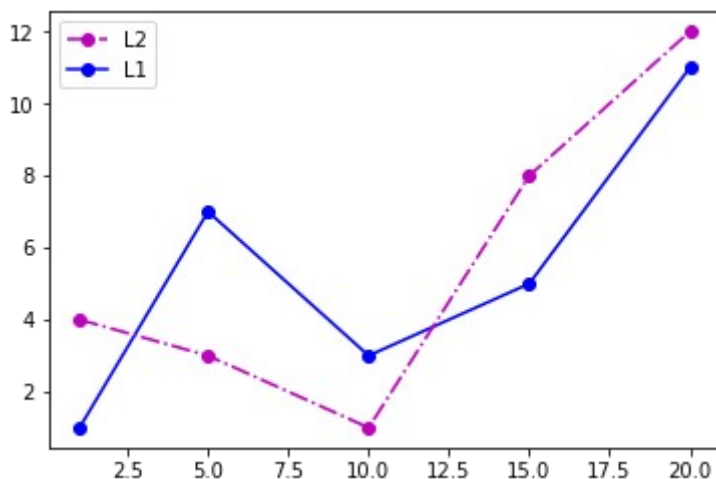
```
plt.plot(x, y1, 'o-r')
plt.plot(x, y2, 'o-.g')
plt.legend(['L1', 'L2'])
```



**Рисунок 3.2 — Легенда на графике (пример 2)**

В третьем варианте можно вручную указать соответствие линий и меток:

```
line1, = plt.plot(x, y1, 'o-b')
line2, = plt.plot(x, y2, 'o-.m')
plt.legend((line2, line1), ['L2', 'L1'])
```



**Рисунок 3.3 — Легенда на графике (пример 3)**

### 3.1.2 Расположение легенды на графике

Место расположения легенды определяется параметром `loc`, который может принимать одно из значений, указанных в таблице 3.1.

**Таблица 3.1 — Параметры расположения легенды на графике**

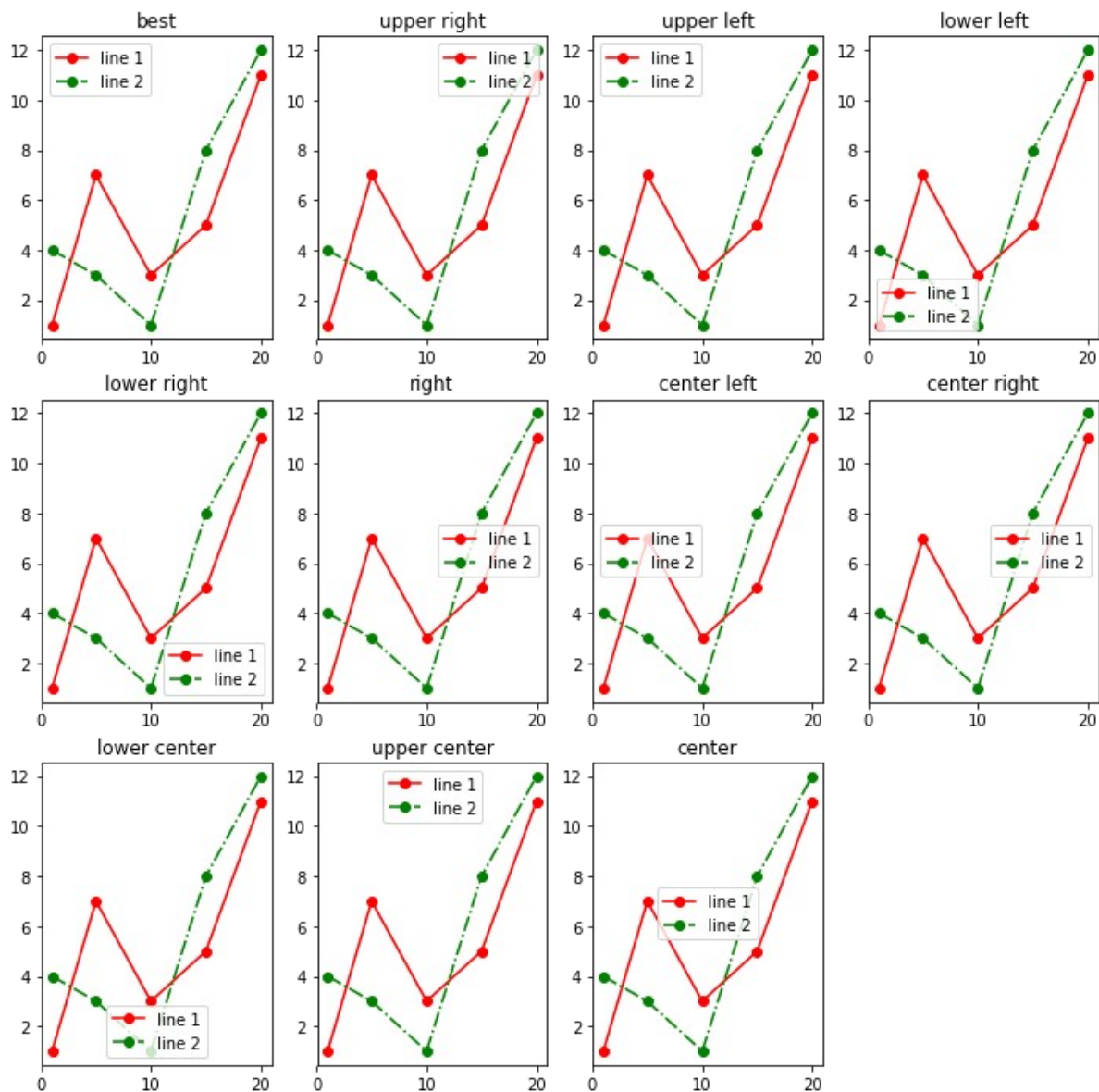
Строковое описание	Код
'best'	0
'upper right'	1
'upper left'	2
'lower left'	3
'lower right'	4
'right'	5
'center left'	6
'center right'	7
'lower center'	8



'upper center'	9
'center'	10

Ниже представлен пример, демонстрирующий различные варианты расстановки легенды через параметр `loc`:

```
locs = ['best', 'upper right', 'upper left', 'lower left',
        'lower right', 'right', 'center left', 'center right',
        'lower center', 'upper center', 'center']
plt.figure(figsize=(12, 12))
for i in range(3):
    for j in range(4):
        if i*4+j < 11:
            plt.subplot(3, 4, i*4+j+1)
            plt.title(locs[i*4+j])
            plt.plot(x, y1, 'o-r', label='line 1')
            plt.plot(x, y2, 'o-.g', label='line 2')
            plt.legend(loc=locs[i*4+j])
        else:
            break
```



**Рисунок 3.4 — Различные варианты расположения легенды на графике**

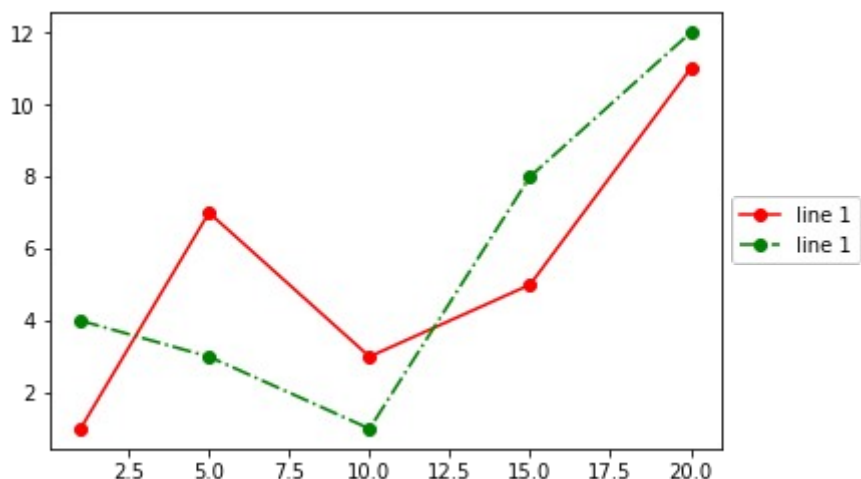
Для более гибкого управления расположением объекта можно воспользоваться параметром `bbox_to_anchor` функции `legend()`. Этому параметру присваивается кортеж, состоящий из четырех или двух элементов:

`bbox_to_anchor = (x, y, width, height)`

`bbox_to_anchor = (x, y)`

Пример использования параметра `bbox_to_anchor`:

```
plt.plot(x, y1, 'o-r', label='line 1')
plt.plot(x, y2, 'o-.g', label='line 1')
plt.legend(bbox_to_anchor=(1, 0.6))
```



**Рисунок 3.5 — Расположение легенды вне поля графика**

### 3.1.3 Дополнительные параметры настройки легенды

В таблице 3.2 представлены дополнительные параметры, которые можно использовать для более тонкой настройки легенды.

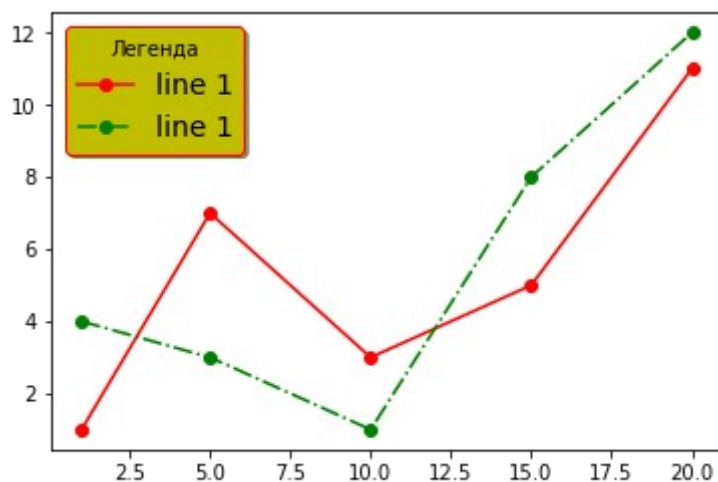
**Таблица 3.2 — Параметры настройки отображения легенды**

Параметр	Тип	Описание
fontsize	int или float или {'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large'}	Размера шрифта надписи легенды
frameon	bool	Отображение рамки легенды
framealpha	None или float	Прозрачность легенды

facecolor	None или str	Цвет заливки
edgecolor	None или str	Цвет рамки
title	None или str	Текст заголовка
title_fontsize	None или str	Размер шрифта

Пример работы с параметрами:

```
plt.plot(x, y1, 'o-r', label='line 1')
plt.plot(x, y2, 'o-.g', label='line 1')
plt.legend(fontsize=14, shadow=True, framealpha=1, facecolor='y',
edgecolor='r', title='Легенда')
```



**Рисунок 3.6 — Пример настройки внешнего вида легенды**

## 3.2 Компоновка графиков

Самые простые и наиболее часто используемые варианты компоновки графиков были рассмотрены в “Уроке 2. Основы работы с модулем `pyplot`”. В этом разделе мы изучим инструмент `GridSpec`, позволяющий более тонко настроить компоновку.

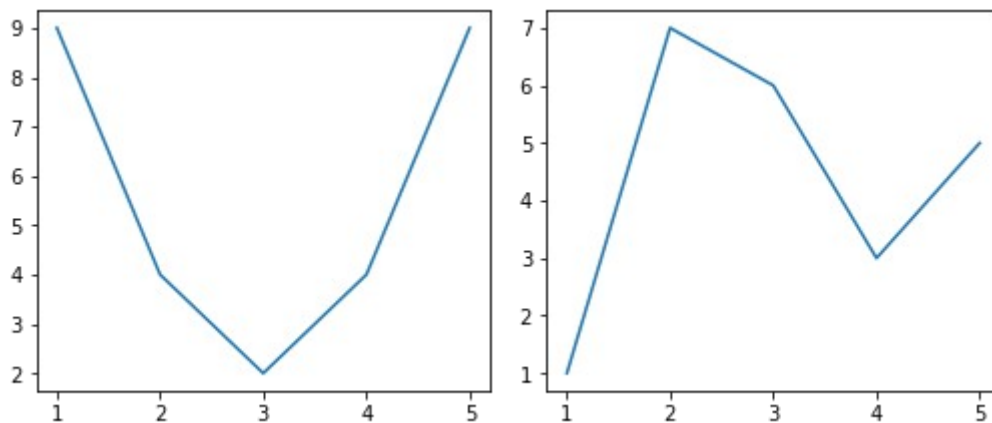
### 3.2.1 Инструмент `GridSpec`

Класс `GridSpec`, позволяет задавать геометрию сетки и расположение на ней полей с графиками. На первый взгляд может показаться, что работа с `GridSpec` довольно неудобна и требует написания лишнего кода, но, если требуется расположить поля с графиками нетривиальным образом, то этот инструмент становится незаменимым. Перед тем как работать с `GridSpec` импортируйте его:

```
import matplotlib.gridspec as gridspec
```

Для начала решим простую задачу отображения двух полей с графиками с использованием `GridSpec`:

```
x = [1, 2, 3, 4, 5]
y1 = [9, 4, 2, 4, 9]
y2 = [1, 7, 6, 3, 5]
fg = plt.figure(figsize=(7, 3), constrained_layout=True)
gs = gridspec.GridSpec(ncols=2, nrows=1, figure=fg)
fig_ax_1 = fg.add_subplot(gs[0, 0])
plt.plot(x, y1)
fig_ax_2 = fg.add_subplot(gs[0, 1])
plt.plot(x, y2)
```



**Рисунок 3.7 — Два поля с графиками**

Объект класса `GridSpec`, создается в строке:

```
gridspec.GridSpec(ncols=2, nrows=1, figure=fg)
```

В конструктор класса передается количество столбцов, строк и Фигура, на которой все будет отображено.

Альтернативный вариант создания объекта `GridSpec` выглядит так:

```
gs = fg.add_gridspec(1, 2)
```

Здесь `fg` - это объект `Figure`, у которого есть метод `add_gridspec()`, позволяющий добавить на него сетку с заданными параметрами (в нашем случае одна строка и два столбца).

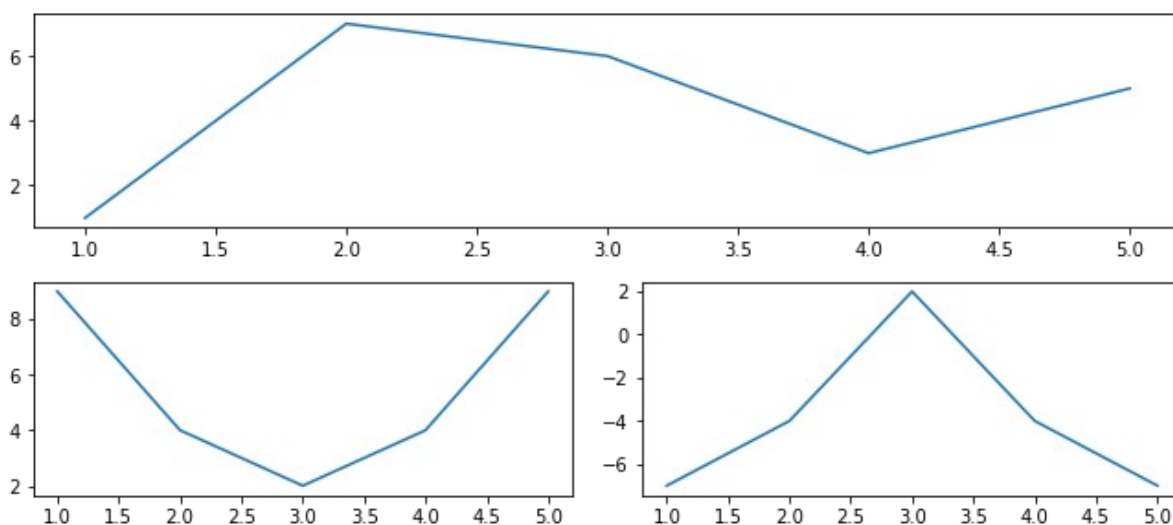
При задании элементов сетки, на которых будет расположено поле с графиком, `GridSpec` позволяет использовать синтаксис подобный тому, что применяется для построения слайсов в *NumPy*.

Добавим еще один набор данных к уже существующему:

```
x = [1, 2, 3, 4, 5]
y1 = [9, 4, 2, 4, 9]
y2 = [1, 7, 6, 3, 5]
y3 = [-7, -4, 2, -4, -7]
```

Построим графики в новой компоновке:

```
fg = plt.figure(figsize=(9, 4), constrained_layout=True)
gs = fg.add_gridspec(2, 2)
fig_ax_1 = fg.add_subplot(gs[0, :])
plt.plot(x, y2)
fig_ax_2 = fg.add_subplot(gs[1, 0])
plt.plot(x, y1)
fig_ax_3 = fg.add_subplot(gs[1, 1])
plt.plot(x, y3)
```



**Рисунок 3.8 — Свободная компоновка (пример 1)**

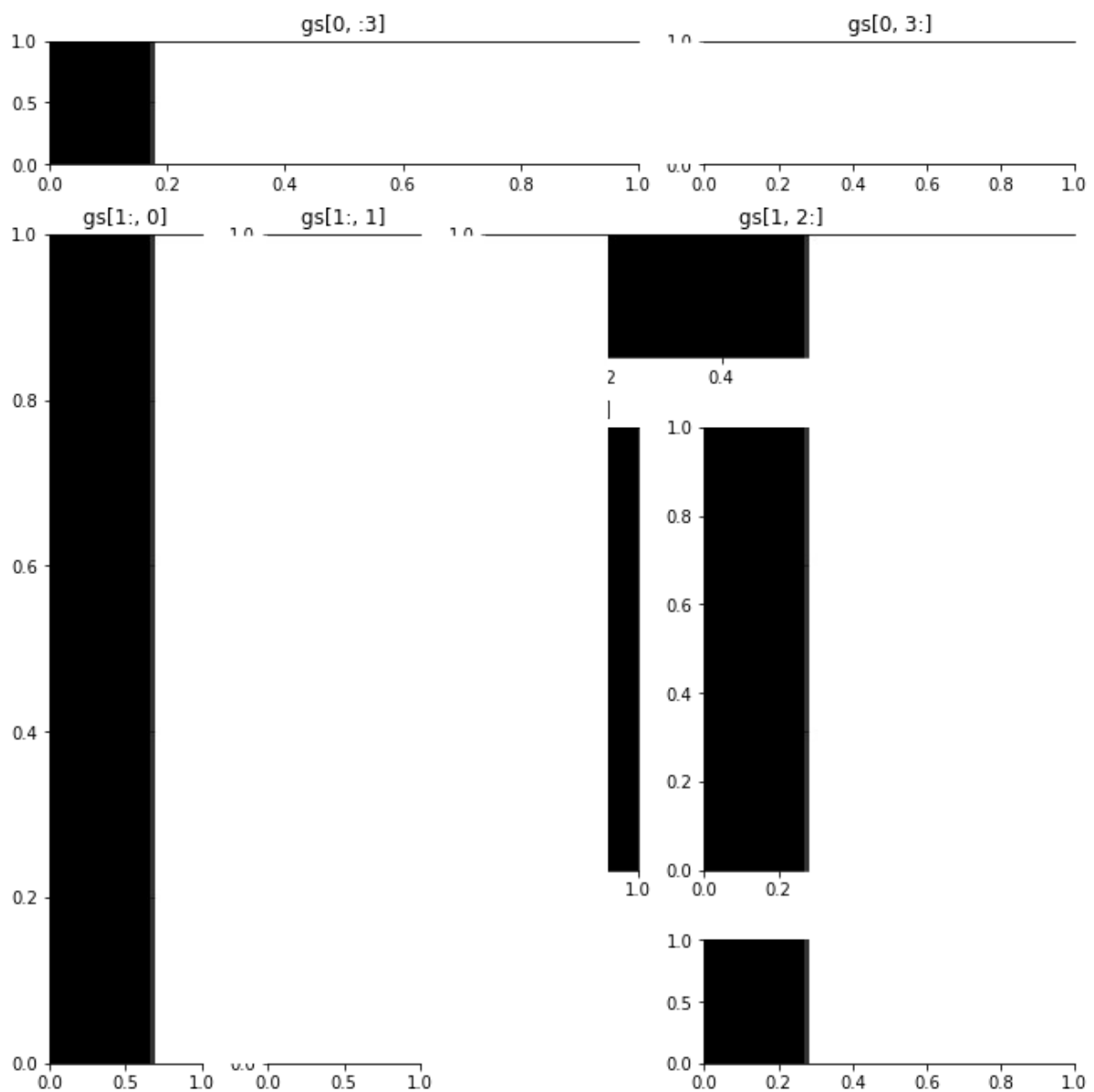
Ниже представлен еще один пример, без данных (с пустыми полями), который иллюстрирует возможности GridSpec:

```
fg = plt.figure(figsize=(9, 9), constrained_layout=True)
gs = fg.add_gridspec(5, 5)
fig_ax_1 = fg.add_subplot(gs[0, :3])
fig_ax_1.set_title('gs[0, :3]')
fig_ax_2 = fg.add_subplot(gs[0, 3:])
fig_ax_2.set_title('gs[0, 3:]')
fig_ax_3 = fg.add_subplot(gs[1:, 0])
fig_ax_3.set_title('gs[1:, 0]')
```

```

fig_ax_4 = fg.add_subplot(gs[1:, 1])
fig_ax_4.set_title('gs[1:, 1]')
fig_ax_5 = fg.add_subplot(gs[1, 2:])
fig_ax_5.set_title('gs[1, 2:]')
fig_ax_6 = fg.add_subplot(gs[2:4, 2])
fig_ax_6.set_title('gs[2:4, 2]')
fig_ax_7 = fg.add_subplot(gs[2:4, 3:])
fig_ax_7.set_title('gs[2:4, 3:]')
fig_ax_8 = fg.add_subplot(gs[4, 3:])
fig_ax_8.set_title('gs[4, 3:]')

```

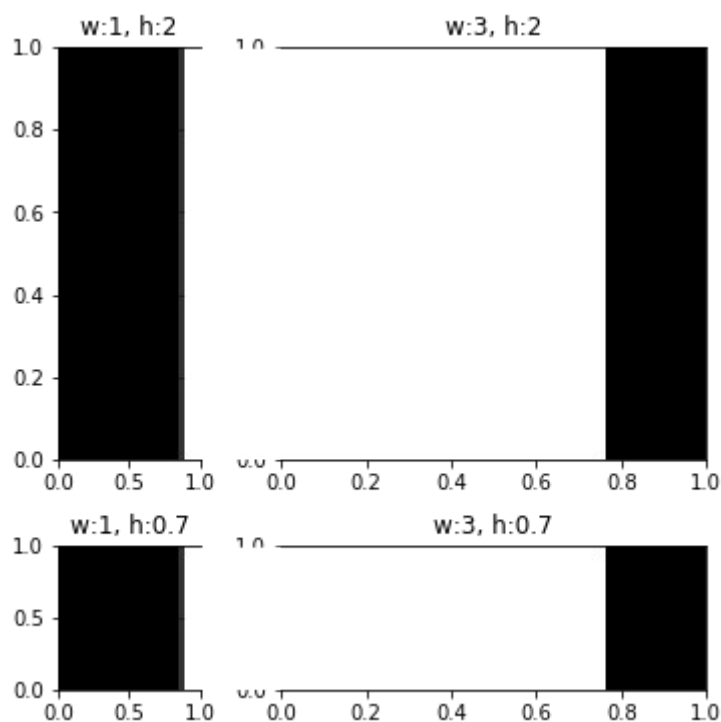


**Рисунок 3.9 — Свободная компоновка (пример 2)**



Можно заранее задать размеры областей и передать их в качестве параметров в виде массивов:

```
fig = plt.figure(figsize=(5, 5), constrained_layout=True)
widths = [1, 3]
heights = [2, 0.7]
gs = fg.add_gridspec(ncols=2, nrows=2, width_ratios=widths,
height_ratios=heights)
fig_ax_1 = fg.add_subplot(gs[0, 0])
fig_ax_1.set_title('w:1, h:2')
fig_ax_2 = fg.add_subplot(gs[0, 1])
fig_ax_2.set_title('w:3, h:2')
fig_ax_3 = fg.add_subplot(gs[1, 0])
fig_ax_3.set_title('w:1, h:0.7')
fig_ax_4 = fg.add_subplot(gs[1, 1])
fig_ax_4.set_title('w:3, h:0.7')
```



**Рисунок 3.10 — Свободная компоновка (пример 3)**

Информации из данного урока и из “Урока 2. Основы работы с модулем `ruplot`” должно быть достаточно, для того чтобы создавать компоновки практически любой сложности.

### 3.3 Текстовые элементы графика

В части текстового наполнения при построении графика выделяют следующие составляющие:

- заголовок поля (title);
- заголовок фигуры (suptitle);
- подписи осей (xlabel, ylabel);
- тестовый блок на поле графика (text), либо на фигуре (figtext);
- аннотация (annotate) - текст и указатель.

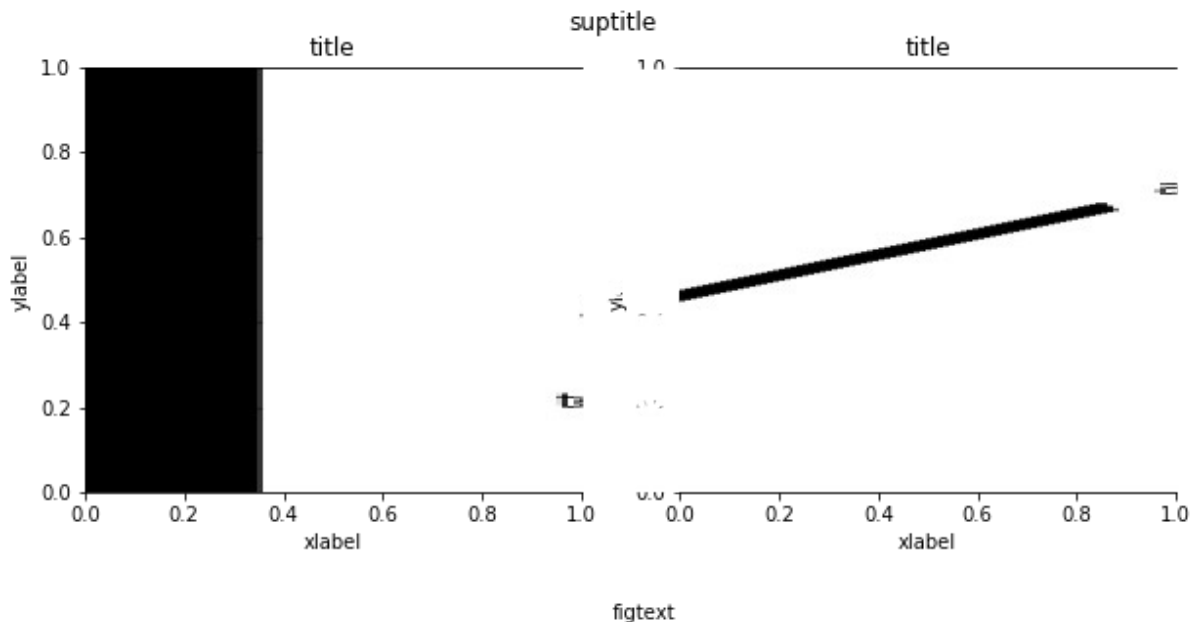
У каждого элемента, который содержит текст, помимо специфических параметров, отвечающих за его настройку, есть параметры класса Text, которые открывают доступ к большому числу настроек внешнего вида и расположения текстового элемента. Более подробно описание параметров, доступных из класса Text, будет дано в “3.4 Свойства класса Text”. Ниже представлен код, отображающий все указанные выше текстовые элементы:

```
plt.figure(figsize=(10,4))

plt.figtext(0.5, -0.1, 'figtext')
plt.suptitle('suptitle')

plt.subplot(121)
plt.title('title')
plt.xlabel('xlabel')
plt.ylabel('ylabel')
plt.text(0.2, 0.2, 'text')
plt.annotate('annotate', xy=(0.2, 0.4), xytext=(0.6, 0.7),
arrowprops=dict(facecolor='black', shrink=0.05))
```

```
plt.subplot(122)
plt.title('title')
plt.xlabel('xlabel')
plt.ylabel('ylabel')
plt.text(0.5, 0.5, 'text')
```



**Рисунок 3.11 — Элементы графика**

Некоторые из представленных текстовых элементов мы уже рассмотрели в “Уроке 2. Основы работы с модулем `pyplot`”, в этом уроке изучим их более подробно. Элементы графика, которые содержат текст, имеют ряд настроечных параметров, которые в официальной документации определяются как `**kwargs`. Это свойства класса `matplotlib.text.Text`, используемые для управления представлением текста.

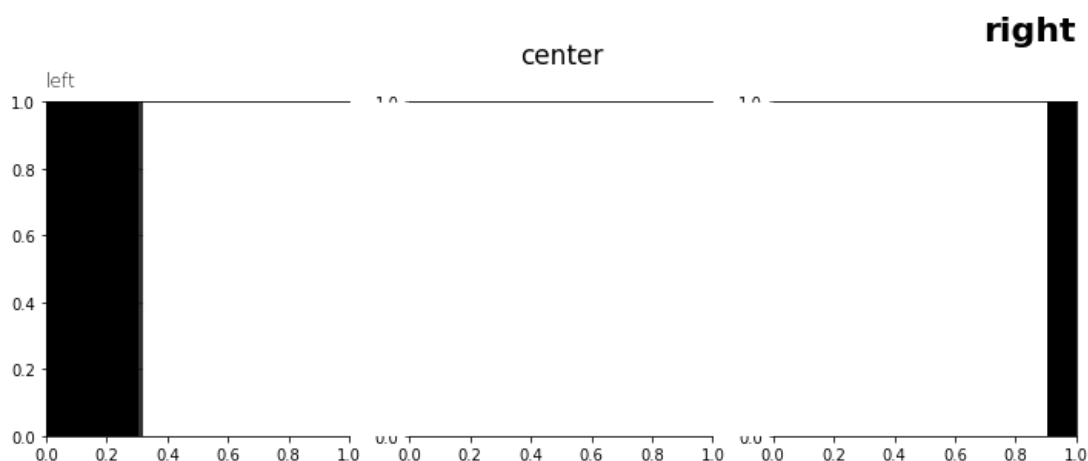
### 3.1.1 Заголовок фигуры и поля графика

Начнем с заголовка поля графика. Текст заголовка поля, устанавливается с помощью функции `title()`, которая имеет следующие основные аргументы:

- `label: str`
  - Текст заголовка.
- `fontdict: dict`
  - Словарь для управления отображением надписи, содержит следующие ключи:
    - `'fontsize'`: размер шрифта;
    - `'fontweight'`: начертание;
    - `'verticalalignment'`: вертикальное выравнивание;
    - `'horizontalalignment'`: горизонтальное выравнивание.
- `loc: {'center', 'left', 'right'}, str, optional`
  - Горизонтальное выравнивание.
- `pad: float`
  - Зазор между заголовком и верхней частью поля графика.

Функция `title()` также поддерживает в качестве аргументов свойства класса `Text`:

```
weight=['light', 'regular', 'bold']
plt.figure(figsize=(12, 4))
for i, lc in enumerate(['left', 'center', 'right']):
    plt.subplot(1, 3, i+1)
    plt.title(label=lc, loc=lc, fontsize=12+i*5, fontweight=weight[i],
pad=10+i*15)
```



**Рисунок 3.12 — Заголовок графика**

Заголовок фигуры, задается с помощью функции `suptitle()`, аргументы этой функции аналогичны тем, что были рассмотрены для `title()`. Более тонкую настройку можно также сделать через свойства класса `Text`.

### 3.1.2 Подписи осей графика

При работе с `pyplot`, для установки подписей осей графика используются функции `labelx()` и `labeley()`, при работе с объектом `Axes` - функции `set_xlabel()` и `set_ylabel()`.

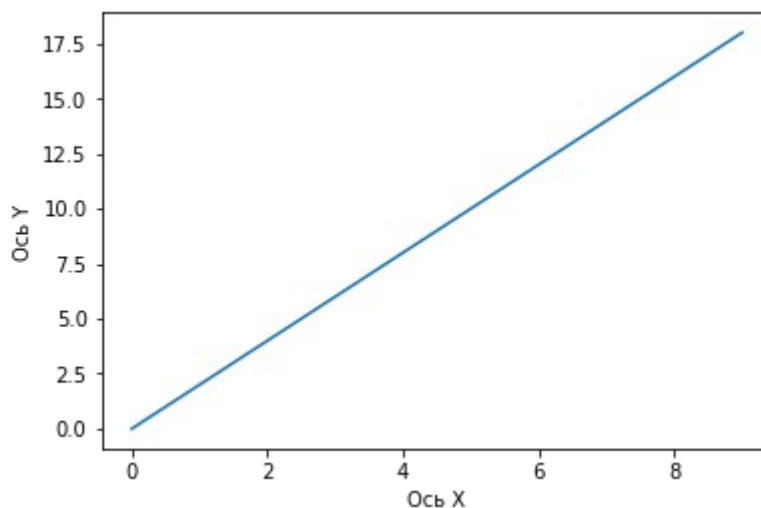
Основные аргументы функций почти полностью совпадают с теми, что были описаны для функции `title()`:

- `label: str`
  - Текст подписи.
- `fontdict: dict`
  - Словарь для управления отображением надписи, содержит следующие ключи:
    - `'fontsize'`: размер шрифта;
    - `'fontweight'`: начертание;

- 'verticalalignment': вертикальное выравнивание;
- 'horizontalalignment': горизонтальное выравнивание.
- labelpad: float
  - Зазор между подписью и осью.

В самом простом случае достаточно передать только подпись в виде строки:

```
x = [i for i in range(10)]  
y = [i*2 for i in range(10)]  
plt.plot(x, y)  
plt.xlabel('Ось X')  
plt.ylabel('Ось Y')
```



**Рисунок 3.13 — Подписи осей графика (пример 1)**

Используем некоторые из дополнительных свойств для настройки внешнего вида подписей осей:

```
plt.plot(x, y)
plt.xlabel('Ось X\nНезависимая величина', fontsize=14, fontweight='bold')
plt.ylabel('Ось Y\nЗависимая величина', fontsize=14, fontweight='bold')
```

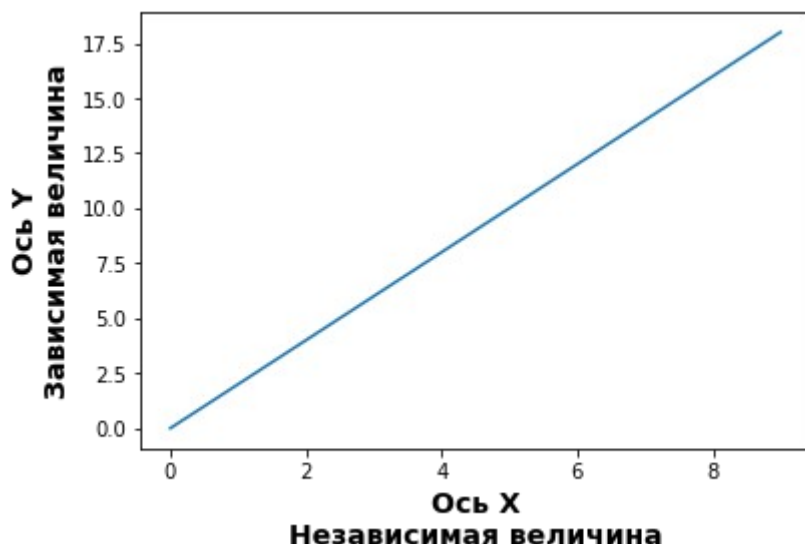


Рисунок 3.14 — Подписи осей графика (пример 2)

### 3.1.3 Текстовый блок

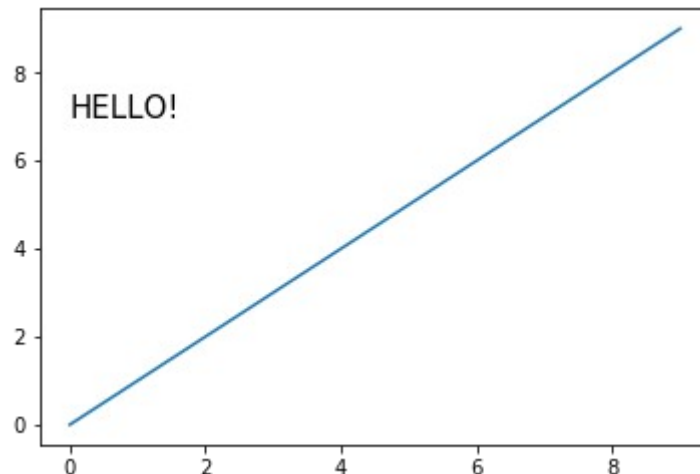
За установку текстовых блоков на поле графика отвечает функция `text()`. Через основные параметры этой функции можно задать расположение, содержание и настройки шрифта:

- `x: float`
  - Значение координаты `x` надписи.
- `y: float`
  - Значение координаты `y` надписи.
- `s: str`
  - Текст надписи.



В простейшем варианте использование `text()` будет выглядеть так:

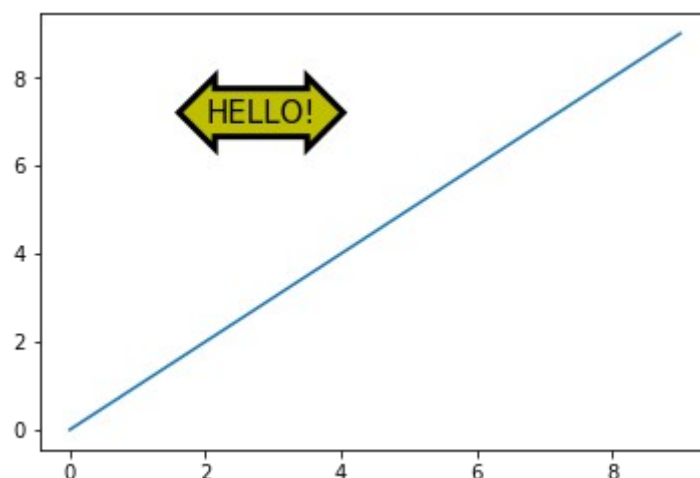
```
plt.text(0, 7, 'HELLO!', fontsize=15)
plt.plot(range(0,10), range(0,10))
```



**Рисунок 3.15 — Текстовый блок (пример 1)**

Используем свойства класса `Text` для модификации этого представления:

```
bbox_properties=dict(boxstyle='darrow, pad=0.3', ec='k', fc='y', ls='--',
                    lw=3)
plt.text(2, 7, 'HELLO!', fontsize=15, bbox=bbox_properties)
plt.plot(range(0,10), range(0,10))
```



**Рисунок 3.16 — Текстовый блок (пример 2)**

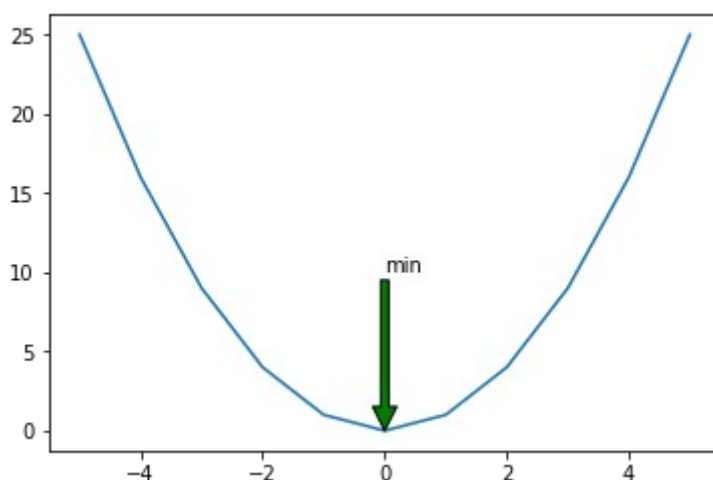
### 3.1.4 Аннотация

Инструмент Аннотация позволяет установить текстовый блок с заданным содержанием и стрелкой для указания на конкретное место на графике. Для создания аннотации используется функция `annotate()`, основными ее аргументами являются:

- `text: str`
  - Текст аннотации.
- `xy: (float, float)`
  - Координаты места, на которое будет указывать стрелка.
- `xytext: (float, float), optional`
  - Координаты расположения текстовой надписи.
- `xycoords: str`
  - Система координат, в которой определяется расположение указателя.
- `textcoords: str`
  - Система координат, в которой определяется расположение текстового блока.
- `arrowprops: dict, optional`
  - Параметры отображения стрелки. Имена этих параметров (ключи словаря) являются параметрами конструктора объекта класса `FancyArrowPatch`.

Ниже представлен пример кода, который демонстрирует простой вариант использования `annotation()`:

```
import math
x = list(range(-5, 6))
y = [i**2 for i in x]
plt.annotate('min', xy=(0, 0), xycoords='data', xytext=(0, 10),
textcoords='data', arrowprops=dict(facecolor='g'))
plt.plot(x, y)
```



**Рисунок 3.17 — Аннотация**

Параметрам `xycoords` и `textcoords` может быть присвоено значение из таблицы 3.3.

**Таблица 3.3 — Значения параметров `xycoords` и `textcoords`**

Значение	Описание
'figure points'	Начало координат - нижний левый угол фигуры (0, 0). Координаты задаются в точках.
'figure pixels'	Начало координат - нижний левый угол фигуры (0, 0). Координаты задаются в пикселях.

'figure fraction'	Начало координат - нижний левый угол фигуры (0, 0) при этом верхний правый угол - это точка (1, 1). Координаты задаются в долях от единицы.
'axes points'	Начало координат - нижний левый угол поля графика (0, 0). Координаты задаются в точках.
'axes pixels'	Начало координат - нижний левый угол поля графика (0, 0). Координаты задаются в пикселях.
'axes fraction'	Начало координат - нижний левый угол поля графика (0, 0) при этом верхний правый угол поля - это точка (1, 1). Координаты задаются в долях от единицы.
'data'	Тип координатной системы: декартова. Работа ведется в пространстве поля графика.
'polar'	Тип координатной системы: полярная. Работа ведется в пространстве поля графика.

Для модификации внешнего вида надписи воспользуйтесь свойствами класса Text.

Рассмотрим настройку внешнего вида стрелки аннотации. За конфигурирование отображения стрелки отвечает параметр `arrowprops`, который принимает в качестве значения словарь, ключами которого являются параметры конструктора класса `FancyArrowPatch`, из них выделим два: `arrowstyle` отвечает за стиль стрелки и `connectionstyle` - отвечает за стиль соединительной линии.

## Стиль стрелки

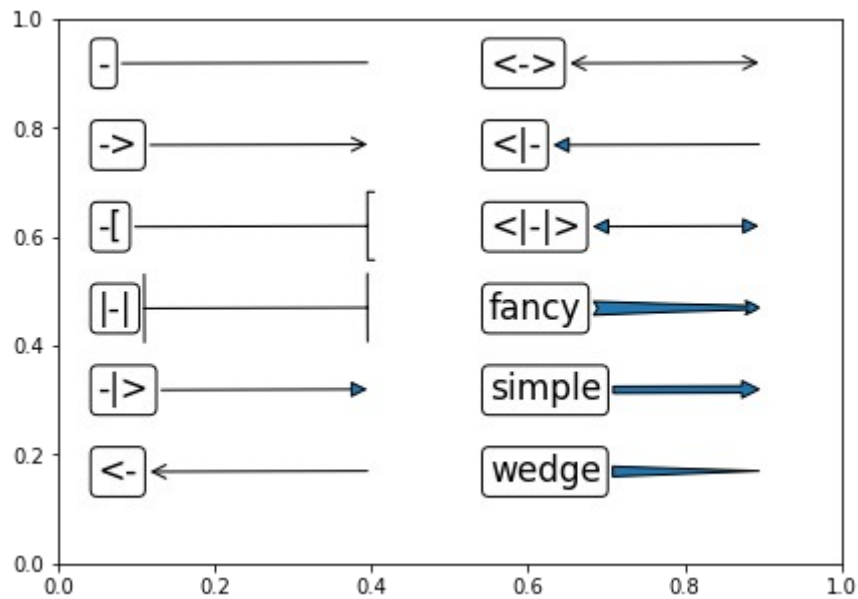
Параметр: **arrowstyle**

Тип: str, ArrowStyle, optional

Доступные стили стрелок представлены в таблице 3.4 и на рисунке 3.18.

**Таблица 3.4 — Стили стрелок аннотации**

Класс	Имя	Атрибуты
Curve	-	None
CurveB	->	head_length=0.4, head_width=0.2
BracketB	-[	widthB=1.0, lengthB=0.2, angleB=None
CurveFilledB	- >	head_length=0.4, head_width=0.2
CurveA	<-	head_length=0.4, head_width=0.2
CurveAB	<->	head_length=0.4, head_width=0.2
CurveFilledA	< -	head_length=0.4, head_width=0.2
CurveFilledAB	< - >	head_length=0.4, head_width=0.2
BracketA	]-	widthA=1.0, lengthA=0.2, angleA=None
BracketAB	]-[	widthA=1.0, lengthA=0.2, angleA=None, widthB=1.0, lengthB=0.2, angleB=None
Fancy	fancy	head_length=0.4, head_width=0.4, tail_width=0.4
Simple	simple	head_length=0.5, head_width=0.5, tail_width=0.2
Wedge	wedge	tail_width=0.3, shrink_factor=0.5



**Рисунок 3.18 — Стили стрелок аннотации**

Программный код для построения изображения, представленного на рисунке 3.18:

```
plt.figure(figsize=(7,5))
arrows = ['-', '->', '-[', '|-|', '-|>', '<-', '<->', '<|-', '<|-|>',
'fancy', 'simple', 'wedge']

bbox_properties=dict(
    boxstyle='round,pad=0.2',
    ec='k',
    fc='w',
    ls='--',
    lw=1
)
ofs_x = 0
ofs_y = 0
```

```

for i, ar in enumerate(arrows):
    if i == 6: ofs_x = 0.5

    plt.annotate(ar, xy=(0.4+ofs_x, 0.92-ofs_y), xycoords='data',
                  xytext=(0.05+ofs_x, 0.9-ofs_y), textcoords='data', fontsize=17,
                  bbox=bbox_properties,
                  arrowprops=dict(arrowstyle=ar)
                  )
    if ofs_y == 0.75: ofs_y = 0
    else: ofs_y += 0.15

```

### **Стиль соединительной линии**

Параметр: **connectionstyle**

Тип: str, ConnectionStyle, None, optional

Через данный параметр можно задать описание стиля линии, которая соединяет точки (xy, xycoords). В качестве значения данный параметр может принимать объект класса ConnectionStyle, или строку, в которой указывается стиль линии соединения с параметрами, перечисленными через запятую.

**Таблица 3.4 — Стили соединительной линии аннотации**

Класс	Имя	Атрибуты
Angle	angle	angleA=90, angleB=0, rad=0.0
Angle3	angle3	angleA=90, angleB=0
Arc	arc	angleA=0, angleB=0, armA=None, armB=None, rad=0.0
Arc3	arc3	rad=0.0
Bar	bar	armA=0.0, armB=0.0, fraction=0.3, angle=None

Ниже представлен пример, который демонстрирует возможности работы с параметром `connectionstyle`:

```
import math
fig, axs = plt.subplots(2, 3, figsize=(12, 7))
conn_style=[
    'angle,angleA=90,angleB=0,rad=0.0',
    'angle3,angleA=90,angleB=0',
    'arc,angleA=0,angleB=0,armA=0,armB=40,rad=0.0',
    'arc3,rad=-1.0',
    'bar,armA=0.0,armB=0.0,fraction=0.1,angle=70',
    'bar,fraction=-0.5,angle=180',
]
for i in range(2):
    for j in range(3):
        axs[i, j].text(0.1, 0.5, '\n'.join(conn_style[i*3+j].split(',')))
        axs[i, j].annotate('text', xy=(0.2, 0.2), xycoords='data',
                           xytext=(0.7, 0.8), textcoords='data',
                           arrowprops=dict(arrowstyle='->',
connectionstyle=conn_style[i*3+j]))
```

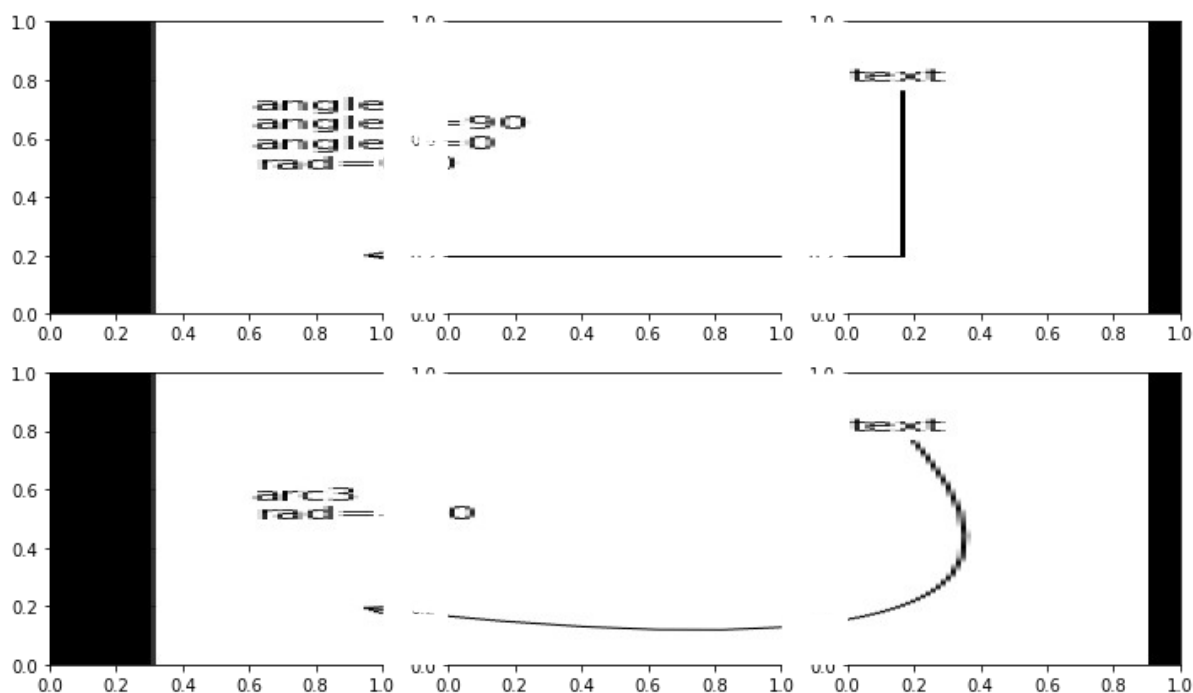


Рисунок 3.19 — Стили соединительной линии аннотации



## 3.4 Свойства класса `Text`

Рассмотрим свойства класса `matplotlib.text.Text`, которые предоставляют доступ к тонким настройкам текстовых элементов. Мы не будем рассматривать все свойства класса `Text`, сделаем обзор наиболее часто используемых.

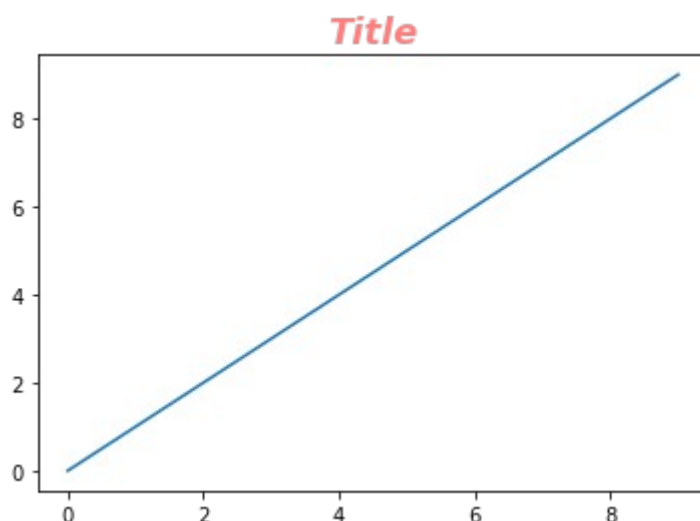
### 3.4.1 Параметры, отвечающие за отображения текста

- `alpha: float`
  - Уровень прозрачности надписи. Параметр задается числом в диапазоне от 0 до 1. 0 - полная прозрачность, 1 - полная непрозрачность.
- `color: color`
  - Цвет текста. Значение параметра имеет тоже тип, что и параметр функции `plot`, отвечающий за цвет графика.
- `fontfamily (или family): str`
  - Шрифт текста, задается в виде строки из набора: `{'serif', 'sans-serif', 'cursive', 'fantasy', 'monospace'}`. Можно использовать свой шрифт.
- `fontsize (или size): str, int`
  - Размер шрифта, можно выбрать из ряда: `{'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large'}`, либо задать в виде численного значения.
- `fontstyle (или style): str`
  - Стилль шрифта, задается из набора: `{'normal', 'italic', 'oblique'}`.
- `fontvariant (или variant): str`
  - Начертание шрифта, задается из набора: `{'normal', 'small-caps'}`.

- `fontweight` (или `weight`): str
  - Насыщенность шрифта, задается из набора: {'ultralight', 'light', 'normal', 'regular', 'book', 'medium', 'roman', 'semibold', 'demibold', 'demi', 'bold', 'heavy', 'extra bold', 'black'} либо численным значением в диапазоне 0-1000.

Рассмотрим пример, демонстрирующий использование перечисленных выше параметров:

```
plt.title('Title', alpha=0.5, color='r', fontsize=18, fontstyle='italic',
fontweight='bold', linespacing=10)
plt.plot(range(0,10), range(0,10))
```



**Рисунок 3.20 — Пример использования свойств класса Text**

Для группового задания свойств можно использовать параметр `fontproperties` или `font_properties`, которому в качестве значения передается объект класса `font_manager.FontProperties`. Конструктор класса `FontProperties` выглядит так:

```
FontProperties(family=None, style=None, variant=None, weight=None,
stretch=None, size=None, fname=None)
```

- `family`
  - Имя шрифта.
- `style`
  - Стилль шрифта.
- `variant`
  - Начертание.
- `stretch`
  - Ширина шрифта.
- `weight`
  - Насыщенность шрифта.
- `size`
  - Размер шрифта.

Типы параметров конструктора такие же как у параметров, отвечающих за отображение шрифта. Не забудьте предварительно импортировать `FontProperties` прежде, чем его использовать:

```
from matplotlib.font_manager import FontProperties
plt.title('Title', fontproperties=FontProperties(family='monospace',
style='italic', weight='heavy', size=15))
plt.plot(range(0,10), range(0,10))
```

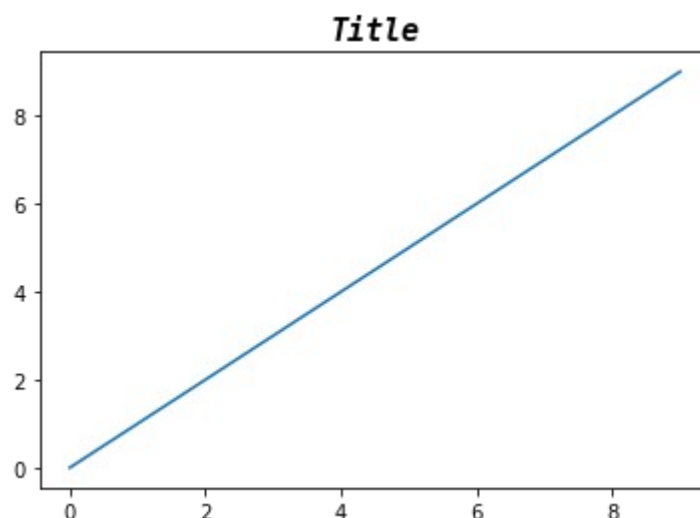


Рисунок 3.21 — Пример использования `fontproperties`

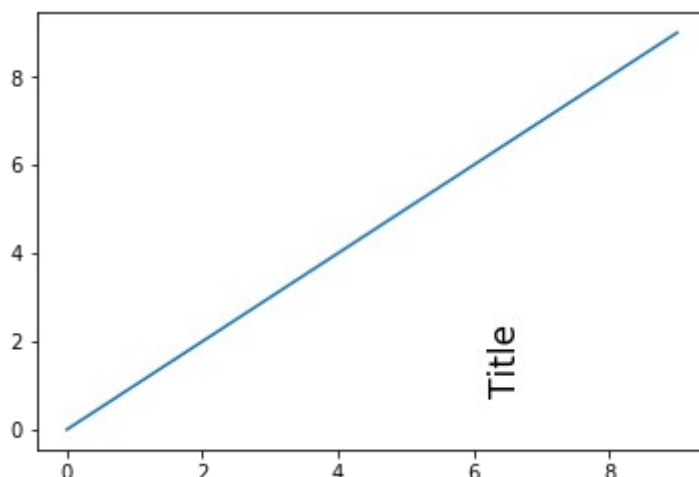
### 3.4.2 Параметры, отвечающие за расположение надписи

Для надписи можно задать выравнивание, позицию, вращение и z-порядок:

- `horizontalalignment` (или `ha`): `str`
  - Горизонтальное выравнивание. Задается из набора: `{'center', 'right', 'left'}`.
- `verticalalignment` (или `va`): `str`
  - Вертикальное выравнивание. Задается из набора `{'center', 'top', 'bottom', 'baseline', 'center_baseline'}`.
- `position`: `(float, float)`
  - Позиция надписи. Определяется двумя координатами `x` и `y`, которые передаются в параметр `position` в виде кортежа из двух элементов.
- `rotation`: `float` или `str`
  - Вращение. Ориентацию надписи можно задать в виде текста `{'vertical', 'horizontal'}` либо численно - значением в градусах.
- `rotation_mode`: `str`
  - Режим вращения. Данный параметр определяет очередность вращения и выравнивания. Если он равен `'default'`, то вначале производится вращение, а потом выравнивание. Если равен `'anchor'`, то наоборот.
- `zorder`: `float`
  - Порядок расположения. Значение параметра определяет очередность вывода элементов. Элемент с минимальным значением `zorder` выводится первым.

Рассмотрим на примере заголовка использование параметров задания расположения:

```
plt.title('Title', fontsize=17, position=(0.7, 0.2), rotation='vertical')
plt.plot(range(0,10), range(0,10))
```



**Рисунок 3.22 — Пример использования параметров, задающих расположение**

### 3.4.3 Параметры, отвечающие за настройку заднего фона надписи

- `backgroundcolor: color`
  - Цвет заднего фона.


Если требуется более тонкая настройка с указанием цвета, толщины, типа рамки, цвета основной заливки и т.п., то используйте параметр `bbox`, его значение - это словарь, ключами которого являются свойства класса `patches.FancyBboxPatch` (см. таблицу 3.5).



**Таблица 3.5 — Свойства класса `patches.FancyBboxPatch`**

Свойство	Тип значения	Описание
<code>boxstyle</code>	<code>str</code> или <code>matplotlib.patches.BoxStyle</code>	Стиль рамки. См. Таблицу 3.6

alpha	float или None	Прозрачность
color	color	Цвет
edgecolor или ec	Color, None или 'auto'	Цвет границы рамки
facecolor или fc	color или None	Цвет заливки
fill	bool	Заливка (использовать или нет)
hatch	{ '/', '\', ' ', '-', '+', 'x', 'o', 'O', '.', '*' }	Штриховка
linestyle или ls	{ '-', '--', '-.', ':', '', (offset, on-off-seq), ... }	Стиль линии рамки
linewidth или lw	float или None	Толщина линии

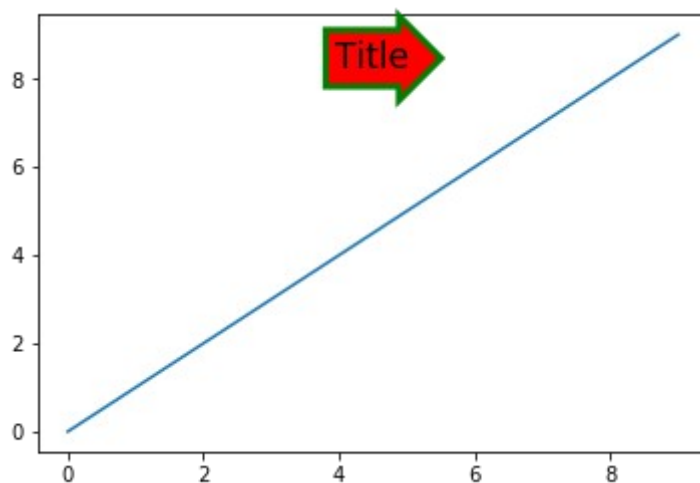
**Таблица 3.6 — Параметры boxstyle**

Класс	Имя	Атрибуты	Внешний вид
Circle	circle	pad=0.3	
DArrow	darrow	pad=0.3	
LArrow	larrow	pad=0.3	
RArrow	rarrow	pad=0.3	
Round	round	pad=0.3, rounding_size=None	
Round4	round4	pad=0.3, rounding_size=None	
Roundtooth	roundtooth	pad=0.3, tooth_size=None	

Sawtooth	sawtooth	pad=0.3, tooth_size=None	
Square	square	pad=0.3	

Пример оформления заднего фона надписи:

```
from matplotlib.patches import FancyBboxPatch
bbox_properties=dict(
    boxstyle='rarrow, pad=0.3',
    ec='g',
    fc='r',
    ls='-',
    lw=3
)
plt.title('Title', fontsize=17, bbox=bbox_properties, position=(0.5,
0.85))
plt.plot(range(0,10), range(0,10))
```



**Рисунок 3.23— Пример настройки заднего фона надписи**

### 3.5 Цветовая полоса — *Colorbar*

Если вы строите цветовое распределение с использованием `colormesh()`, `pcolor()`, `imshow()` и т.п., то для отображения соответствия цвета и численного значения вам может понадобиться аналог легенды, который в *Matplotlib* называется *colorbar*. Создадим случайное распределение с помощью `np.random.rand()`:

```
import numpy as np
np.random.seed(123)
vals = np.random.randint(10, size=(7, 7))
plt.pcolor(vals)
```

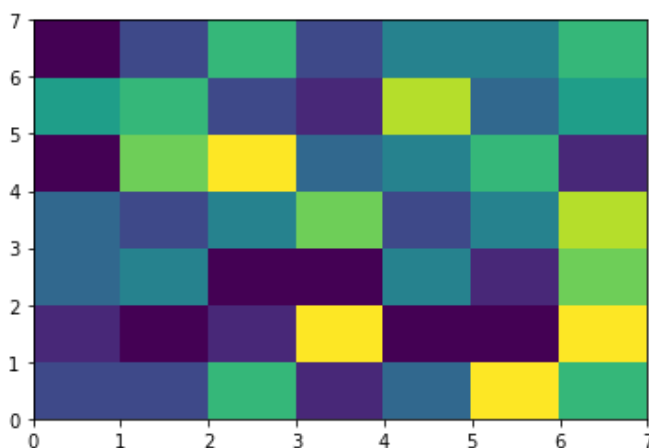
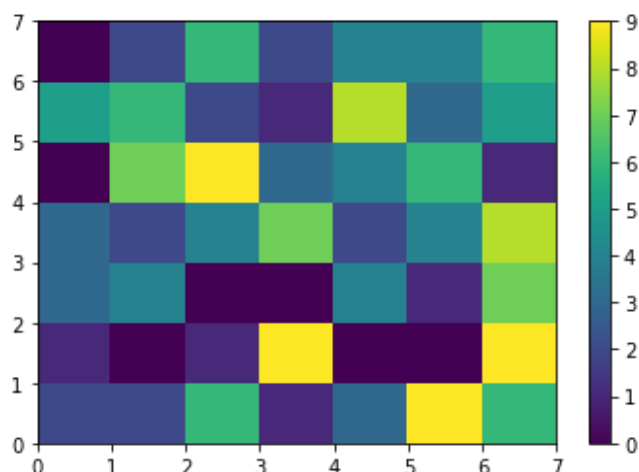


Рисунок 3.23 — Цветовое распределение

Для данного набора построим *colorbar* с помощью соответствующей функции:

```
np.random.seed(123)
vals = np.random.randint(10, size=(7, 7))
plt.pcolor(vals)
plt.colorbar()
```

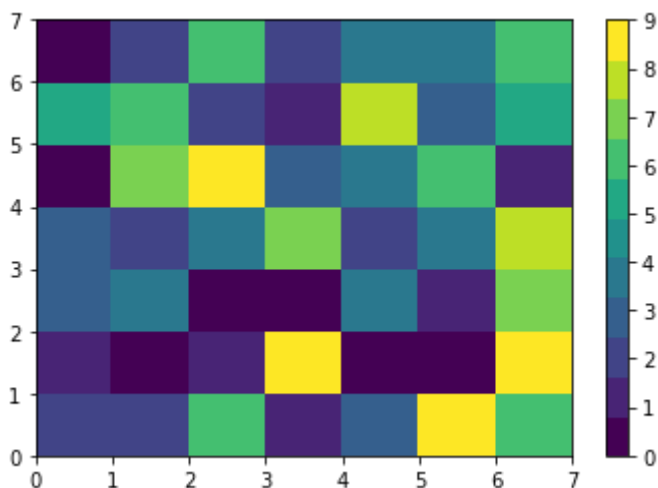




**Рисунок 3.24 — Colorbar для заданного цветового распределения**

Для дискретного разделения цветов на цветовой полосе, нужно при построении изображения в соответствующую функцию (в нашем случае `pcolor()`) через параметр `cmap` передать требуемую цветовую схему:

```
np.random.seed(123)
vals = np.random.randint(10, size=(7, 7))
plt.pcolor(vals, cmap=plt.get_cmap('viridis', 11) )
plt.colorbar()
```



**Рисунок 3.25 — Colorbar с дискретным разделением цветов**

Рассмотрим различные варианты настройки цветовой полосы.

### 3.5.1 Общая настройка с использованием `inset_locator()`

Одни из вариантов более тонкой настройки цветовой полосы - это создать на базе родительского `Axes` элемента свой и модифицировать часть его параметров. Удобно сделать это с помощью функции `inset_axes()` из `mpl_toolkits.axes_grid1.inset_locator`. Основные ее аргументы перечислены в таблице 3.7.

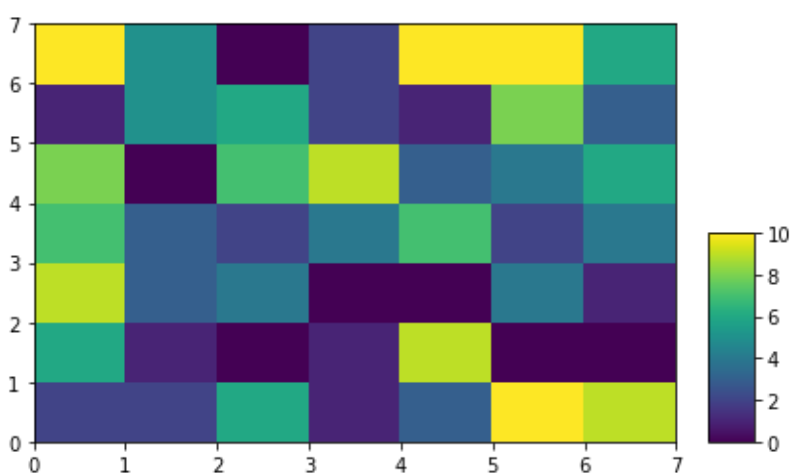
**Таблица 3.7 — Параметры функции `inset_axes()`**

Параметр	Тип	Описание
<code>parent_axes</code>	<code>Axes</code>	Родительский <code>Axes</code> объект
<code>width</code>	<code>float</code> или <code>str</code>	Ширина объекта. Задается в процентах от родительского объекта, либо абсолютным значением в виде числа
<code>height</code>	<code>float</code> или <code>str</code>	Высота объекта. Задается в процентах от родительского объекта, либо абсолютным значением в виде числа
<code>loc</code>	<code>int</code> или <code>string</code> , optional, значение по умолчанию: 1	Расположение объекта. Принимает значение из набора: 'upper right': 1, 'upper left' : 2, 'lower left': 3, 'lower right': 4, 'right': 5, 'center left': 6, 'center right': 7, 'lower center': 8, 'upper center': 9, 'center' : 10

bbox_to_anchor	tuple или matplotlib.transforms.BboxBase или optional	Расположение и соотношение сторон объекта. Задается в формате (левый угол, нижний угол, ширина, высота), либо (левый угол, нижний угол).
bbox_transform	matplotlib.transforms.Transform или optional	Трансформация объекта
borderpad	float или optional	Зазор между bbox_to_anchor и объектом

Продemonстрируем работу с `inset_axes()` на примере:

```
from mpl_toolkits.axes_grid1.inset_locator import inset_axes
np.random.seed(123)
vals = np.random.randint(11, size=(7, 7))
fig, ax = plt.subplots()
gr = ax.pcolor(vals)
axins = inset_axes(ax, width="7%", height="50%", loc='lower left',
bbox_to_anchor=(1.05, 0., 1, 1), bbox_transform=ax.transAxes,
borderpad=0)
plt.colorbar(gr, cax=axins)
```



**Рисунок 3.26 — Colorbar, построенный с использованием `inset_axes()`**

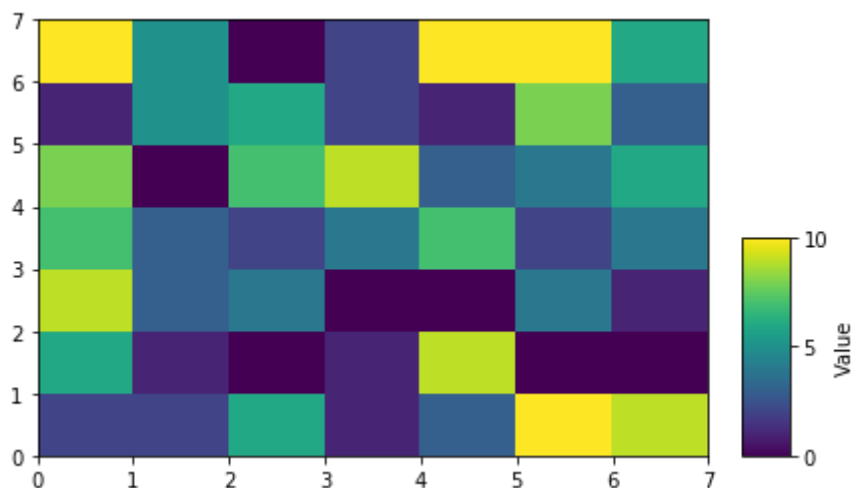
При необходимости можно модифицировать шкалу цветовой полосы с помощью объекта класса `Tick`.

### 3.5.2 Задание шкалы и установка надписи

Для задания собственной шкалы необходимо передать соответствующий список в функцию `colorbar()` через параметр `ticks`. Надпись на шкале, устанавливается с помощью параметра `label` функции `colorbar()`.

Модифицируйте последнюю строку из предыдущего примера следующим образом:

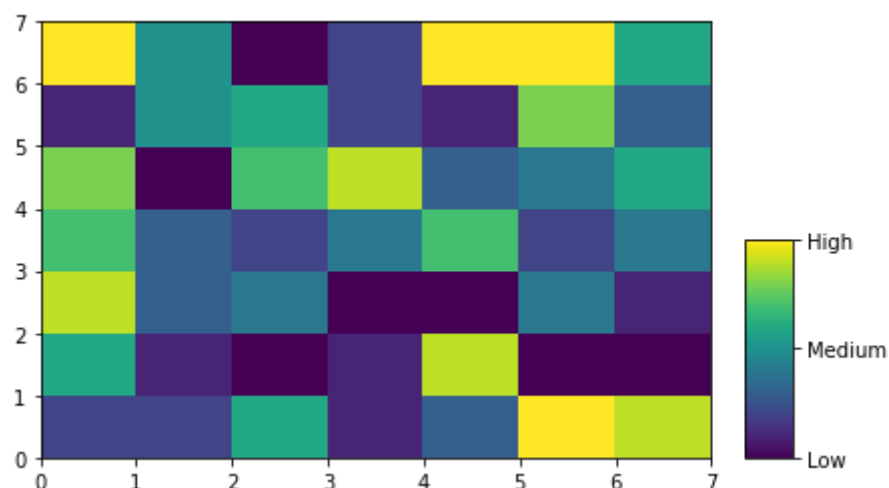
```
plt.colorbar(gr, cax=axins, ticks=[0, 5, 10], label='Value')
```



**Рисунок 3.27 — Colorbar с собственной шкалой**

Если есть необходимость в установке текстовых надписей, то воспользуйтесь функцией `set_yticklabels()`:

```
cbar = plt.colorbar(gr, cax=axins, ticks=[0, 5, 10], label='Value')
cbar.ax.set_yticklabels(['Low', 'Medium', 'High'])
```



**Рисунок 3.28 — Colorbar с текстовыми надписями**

### 3.5.4 Дополнительные параметры настройки *colorbar*

Рассмотрим ряд параметров для настройки внешнего вида *colorbar*, которые доступны как аргументы соответствующей функции `colorbar()`.

**Таблица 3.8 — Параметры функции `colorbar()`**

Свойство	Тип	Описание
<code>orientation</code>	<code>vertical</code> или <code>horizontal</code>	Ориентация
<code>shrink</code>	<code>float</code>	Масштабирование цветовой полосы
<code>extend</code>	[ <code>'neither'</code>   <code>'both'</code>   <code>'min'</code>   <code>'max'</code> ]	Положение указателя продления шкалы
<code>extendfrac</code>	[ <code>None</code>   <code>'auto'</code>   <code>length</code>   <code>lengths</code> ]	Размер указателя продления шкалы
<code>drawedges</code>	<code>bool</code>	Отображение разделительной сетки на цветовой полосе

```
import numpy as np
np.random.seed(123)
vals = np.random.randint(10, size=(7, 7))
plt.pcolor(vals, cmap=plt.get_cmap('viridis', 11))
plt.colorbar(orientation='horizontal',
             shrink=0.9, extend='max', extendfrac=0.2,
             extendrect=False, drawedges=False)
```

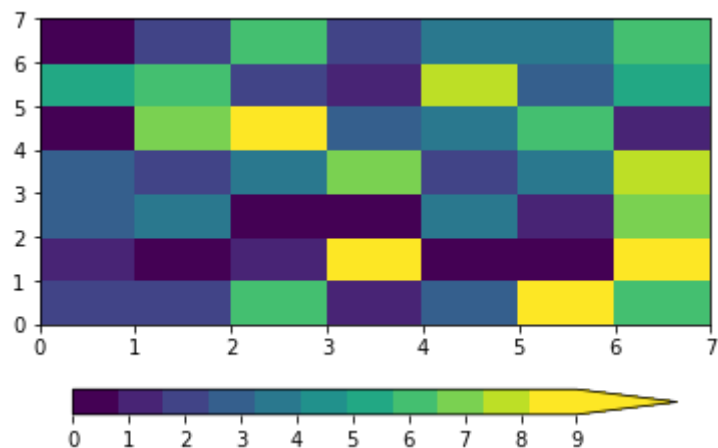


Рисунок 3.29 — *Colorbar* с дополнительными параметрами

# Урок 4 Визуализация данных

## 4.1 Линейный график

Линейный график - это один из наиболее часто используемых видов графиков для визуализации данных. Этот вид графика использовался нами для демонстрации возможностей *Matplotlib* в предыдущих уроках, в этом уроке мы более подробно рассмотрим возможности настройки его внешнего вида.

### 4.1.1 Построение графика

Для построения линейного графика используется функция `plot()`, со следующей сигнатурой:

```
plot([x], y, [fmt], *, data=None, **kwargs)
plot([x], y, [fmt], [x2], y2, [fmt2], ..., **kwargs)
```

Если вызвать функцию `plot()` с одним аргументом - вот так: `plot(y)`, то мы получим график, у которого по оси ординат (ось *y*) будут отложены значения из переданного списка, по по оси абсцисс (ось *x*) - индексы элементов массива.

Рассмотрим аргументы функции `plot()`:

- `x, x2, ...`: массив
  - Набор данных для оси абсцисс первого, второго и т.д. графика.
- `y, y2, ...`: массив
  - Набор данных для оси ординат первого, второго и т.д. графика.
- `fmt: str`
  - Формат графика. Задается в виде строки: `'[marker][line][color]'`.

- **\*\*kwargs** - свойства класса `Line2D`, которые предоставляют доступ к большому количеству настроек внешнего вида графика, наиболее полезные из них представлены в таблице 4.1.

**Таблица 4.1 — Свойства класса `Line2D`**

Свойство	Тип	Описание
<i>alpha</i>	<i>float</i>	Прозрачность
<i>color</i> или <i>c</i>	<i>color</i>	Цвет
<i>fillstyle</i>	<code>{'full', 'left', 'right', 'bottom', 'top', 'none'}</code>	Стиль заливки
<i>label</i>	<i>object</i>	Текстовая метка
<i>linestyle</i> или <i>ls</i>	<code>{'--', '---', '-.', ':', '', (offset, on-off-seq), ...}</code>	Стиль линии
<i>linewidth</i> или <i>lw</i>	<i>float</i>	Толщина линии
<i>marker</i>	<i>matplotlib.markers</i>	Стиль маркера
<i>markeredgecolor</i> или <i>mec</i>	<i>color</i>	Цвет границы маркера
<i>markeredgewidth</i> или <i>mew</i>	<i>float</i>	Толщина границы маркера
<i>markerfacecolor</i> или <i>mfc</i>	<i>color</i>	Цвет заливки маркера
<i>markersize</i> или <i>ms</i>	<i>float</i>	Размер маркера

#### 4.1.1.1 Параметры аргумента `fmt`

Аргумент `fmt` имеет следующий формат: `'[marker][line][color]'`

- `marker: str`
  - Определяет тип маркера, может принимать одно из значений, представленных в таблице 4.2.



**Таблица 4.2 — Тип маркера**

Символ	Описание
'.'	Точка ( <i>point marker</i> )
','	Пиксель ( <i>pixel marker</i> )
'o'	Окружность ( <i>circle marker</i> )
'v'	Треугольник, направленный вниз ( <i>triangle_down marker</i> )
'^'	Треугольник, направленный вверх( <i>triangle_up marker</i> )
'<'	Треугольник, направленный влево ( <i>triangle_left marker</i> )
'>'	Треугольник, направленный вправо ( <i>triangle_right marker</i> )
'1'	Треугольник, направленный вниз ( <i>tri_down marker</i> )
'2'	Треугольник, направленный вверх( <i>tri_up marker</i> )
'3'	Треугольник, направленный влево ( <i>tri_left marker</i> )
'4'	Треугольник, направленный вправо ( <i>tri_right marker</i> )
's'	Квадрат ( <i>square marker</i> )
'p'	Пятиугольник ( <i>pentagon marker</i> )
'*'	Звезда ( <i>star marker</i> )
'h'	Шестиугольник ( <i>hexagon1 marker</i> )
'H'	Шестиугольник ( <i>hexagon2 marker</i> )
'+'	Плюс ( <i>plus marker</i> )
'x'	X-образный маркер ( <i>x marker</i> )
'D'	Ромб ( <i>diamond marker</i> )
'd'	Ромб ( <i>thin_diamond marker</i> )
' '	Вертикальная линия ( <i>vline marker</i> )
'_'	Горизонтальная линия ( <i>hline marker</i> )

- `line: str`
  - Стиль линии.

**Таблица 4.3 — Стиль линии**

Символ	Описание
' - '	Сплошная линия ( <i>solid line style</i> )
' - - '	Штриховая линия ( <i>dashed line style</i> )
' - . '	Штрих-пунктирная линия ( <i>dash-dot line style</i> )
' : '	Штриховая линия ( <i>dotted line style</i> )

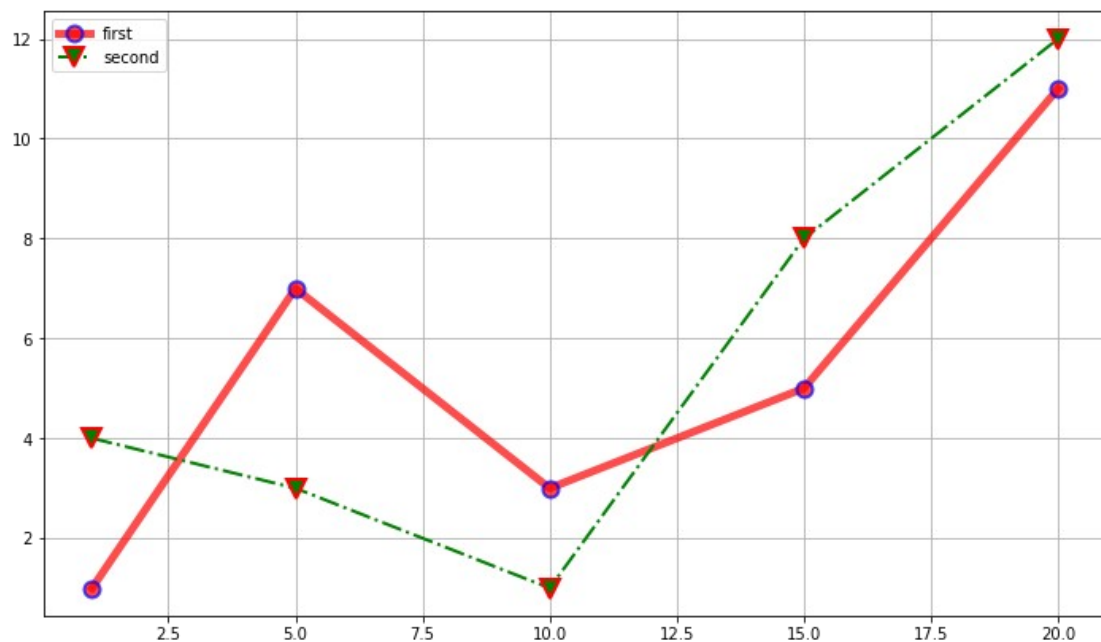
- `color`
  - Цвет графика.

**Таблица 4.4 — Цвет графика**

Символ	Описание
'b'	Синий
'g'	Зеленый
'r'	Красный
'c'	Бирюзовый
'm'	Фиолетовый (пурпурный)
'y'	Желтый
'k'	Черный
'w'	Белый

Реализуем возможности `plot()` на примере:

```
x = [1, 5, 10, 15, 20]
y1 = [1, 7, 3, 5, 11]
y2 = [4, 3, 1, 8, 12]
plt.figure(figsize=(12, 7))
plt.plot(x, y1, 'o-r', alpha=0.7, label='first', lw=5, mec='b', mew=2,
ms=10)
plt.plot(x, y2, 'v-.g', label='second', mec='r', lw=2, mew=2, ms=12)
plt.legend()
plt.grid(True)
```



**Рисунок 4.1 — Графики, построенные с помощью *plot()***

Рассмотрим различные варианты использования линейного графика.

#### 4.1.2 Заливка области между графиком и осью

Для заливки областей используется функция `fill_between()`. Сигнатура функции:

```
fill_between(x, y1, y2=0, where=None, interpolate=False, step=None, *,
data=None, **kwargs)
```

Основные параметры функции:

- `x` : массив длины  $N$ 
  - Набор данных для оси абсцисс.
- `y1` : массив длины  $N$  или скалярное значение
  - Набор данных для оси ординат - первая кривая.
- `y2` : массив длины  $N$  или скалярное значение
  - Набор данных для оси ординат - вторая кривая.

- `where`: массив `bool` элементов (длины  $N$ ), `optional`, значение по умолчанию: `None`
  - Задаёт заливаемый цветом регион, который определяется координатами `x[where]`: интервал будет залит между `x[i]` и `x[i+1]`, если `where[i]` и `where[i+1]` равны `True`.
- `step`: {'pre', 'post', 'mid'}, `optional`
  - Определяет шаг, если используется *step*-функция для отображения графика (будет рассмотрена в одном из следующих уроков).
- `**kwargs`
  - Свойства класса `Polygon`.

Создадим набор данных для эксперимента:

```
import numpy as np
x = np.arange(0.0, 5, 0.01)
y = np.cos(x*np.pi)
```

Отобразим график с заливкой:

```
plt.plot(x, y, c = 'r')
plt.fill_between(x, y)
```

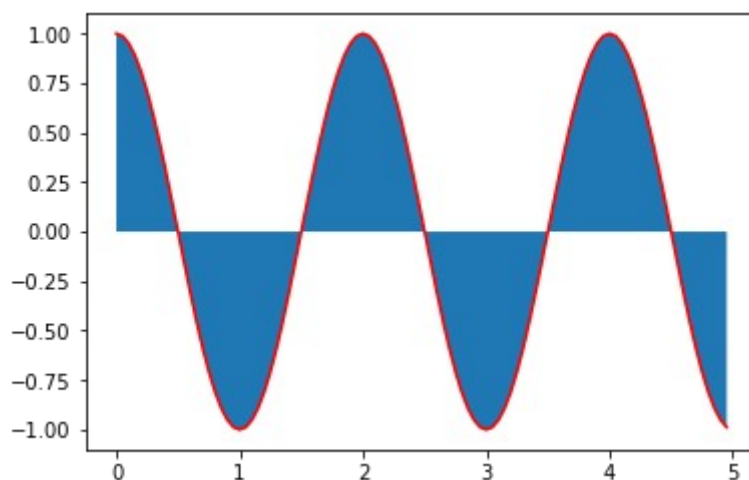
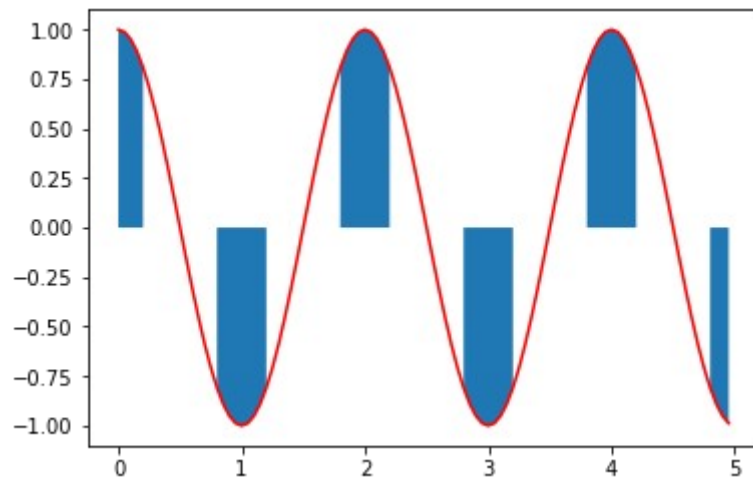


Рисунок 4.2 — График с заливкой (пример 1)

Изменим правила заливки:

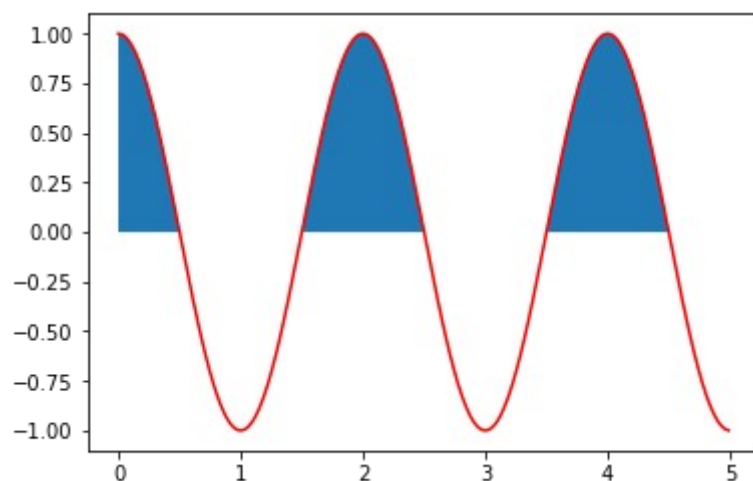
```
plt.plot(x, y, c = 'r')  
plt.fill_between(x, y, where = (y > 0.75) | (y < -0.75))
```



**Рисунок 4.3 — График с заливкой (пример 2)**

Используя параметры  $y1$  и  $y2$  можно формировать более сложные решения. Заливка области между 0 и  $y$ , при условии, что  $y \geq 0$ :

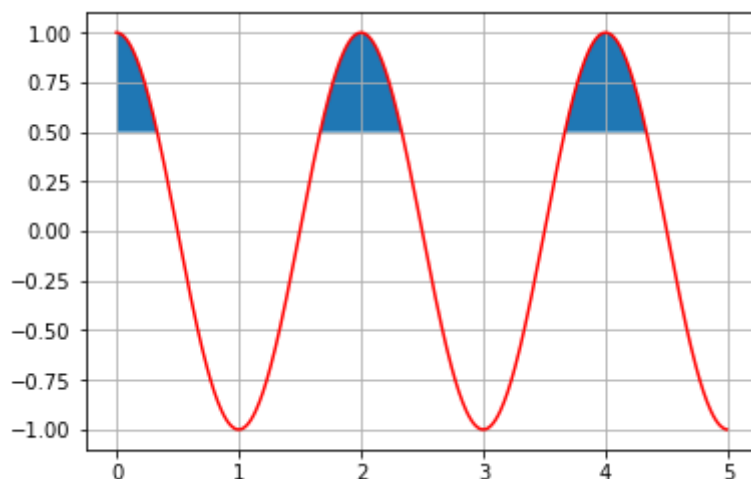
```
plt.plot(x, y, c = 'r')  
plt.fill_between(x, y, where = (y > 0))
```



**Рисунок 4.4 — График с заливкой (пример 3)**

Заливка области между 0.5 и  $y$ , при условии, что  $y \geq 0.5$ :

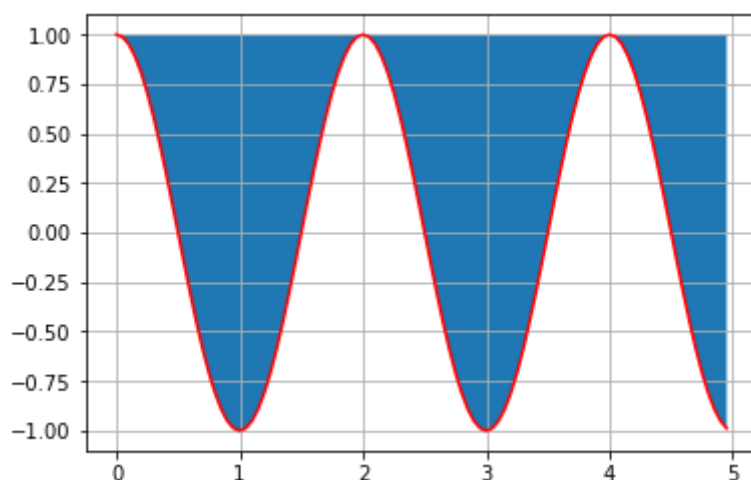
```
plt.plot(x, y, c = 'r')  
plt.grid()  
plt.fill_between(x, 0.5, y, where=y>=0.5)
```



**Рисунок 4.5 — График с заливкой (пример 4)**

Заливка области между  $y$  и 1:

```
plt.plot(x, y, c = 'r')  
plt.grid()  
plt.fill_between(x, y, 1)
```



**Рисунок 4.6 — График с заливкой (пример 5)**

Вариант двухцветной заливки:

```
plt.plot(x, y, c = 'r')  
plt.grid()  
plt.fill_between(x, y, where=y>=0, color='g', alpha=0.3)  
plt.fill_between(x, y, where=y<=0, color='r', alpha=0.3)
```

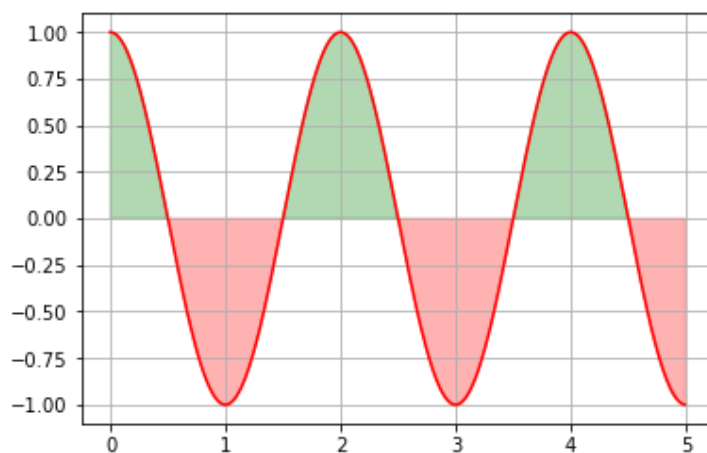


Рисунок 4.7 — График с заливкой (пример 6)

### 4.1.3 Настройка маркировки графиков

В начале этого раздела мы приводили пример использования маркеров при отображении графиков. Сделаем это ещё раз, но уже в упрощенном виде:

```
x = [1, 2, 3, 4, 5, 6, 7]  
y = [7, 6, 5, 4, 5, 6, 7]  
plt.plot(x, y, marker='o', c='g')
```

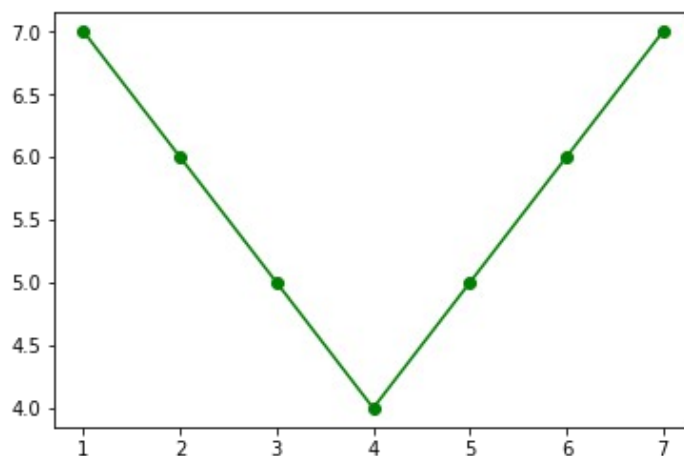


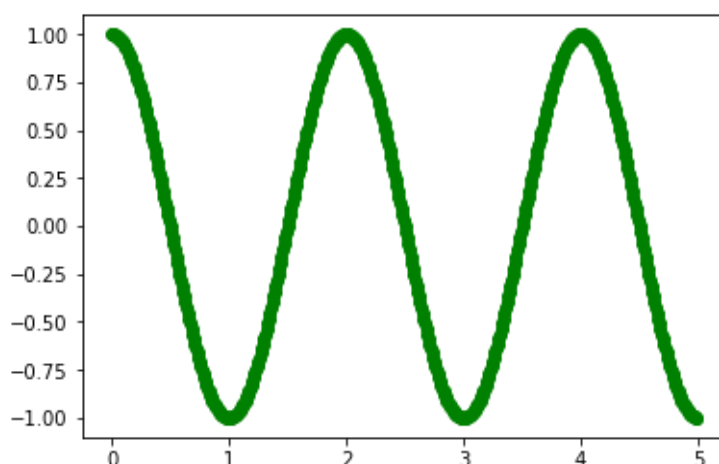
Рисунок 4.8 — График с маркировкой

Создадим набор данных:

```
import numpy as np
x = np.arange(0.0, 5, 0.01)
y = np.cos(x*np.pi)
```

Количество точек в нем равно 500, поэтому представленный выше подход не применим:

```
plt.plot(x, y, marker='o', c='g')
```



**Рисунок 4.8 — График с большим количеством маркеров**

В этом случае нужно задать интервал отображения маркеров, для этого используется параметр `markevery`, который может принимать одно из следующих значений:

- `None` - отображаться будет каждая точка;
- `N` - отображаться будет каждая  $N$ -я точка;
- `(start, N)` - отображается каждая  $N$ -я точка начиная с точки *start*;
- `slice(start, end, N)` - отображается каждая  $N$ -я точка в интервале от *start* до *end*;
- `[i, j, m, n]` - будут отображены только точки  $i, j, m, n$ .



Ниже представлен пример, демонстрирующий работу с `markevery`:

```
x = np.arange(0.0, 5, 0.01)
y = np.cos(x*np.pi)
m_ev_case = [None, 10, (100, 30), slice(100,400,15), [0, 100, 200, 300],
[10, 50, 100]]
```

```
fig, ax = plt.subplots(2, 3, figsize=(10, 7))
axs = [ax[i, j] for i in range(2) for j in range(3)]
for i, case in enumerate(m_ev_case):
    axs[i].set_title(str(case))
    axs[i].plot(x, y, 'o', ls='--', ms=7, markevery=case)
```

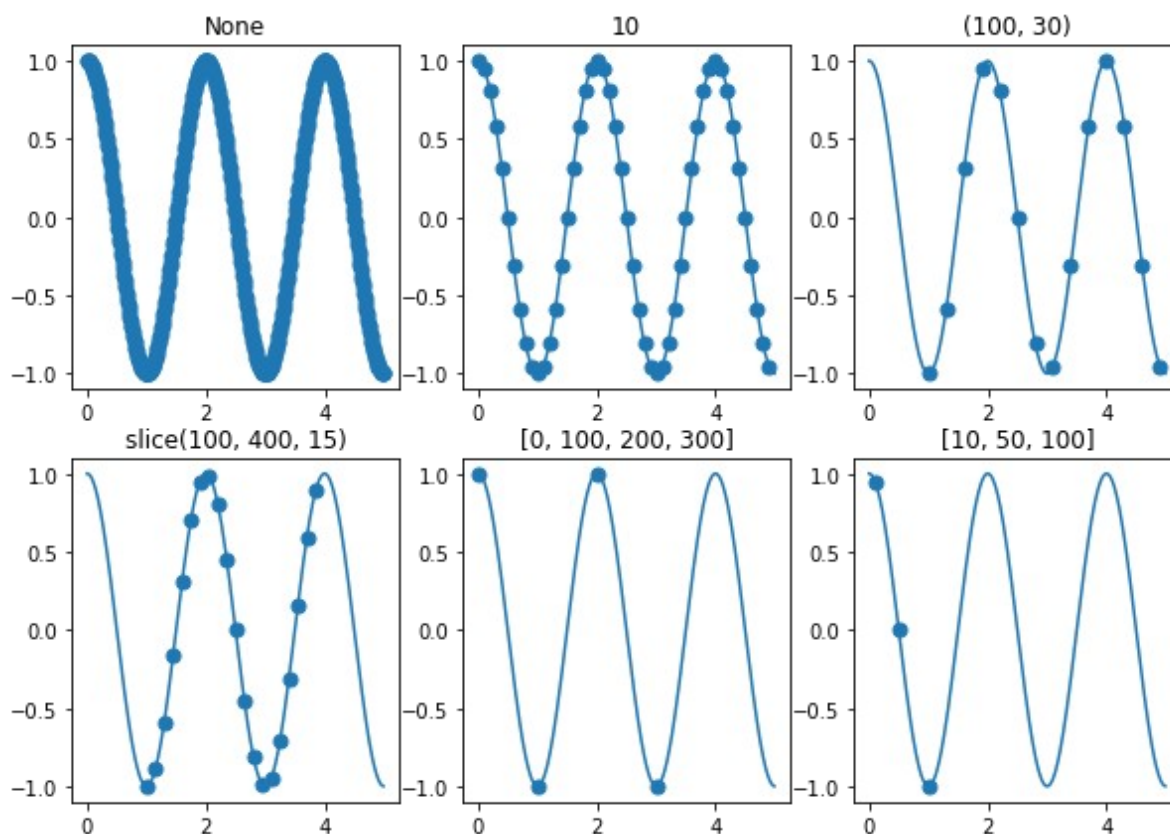


Рисунок 4.9 — Различные варианты маркировки

### 4.1.4 Обрезка графика

Для того, чтобы отобразить только часть графика, которая отвечает определенному условию, используйте предварительное маскирование данных с помощью функции `masked_where()` из пакета `numpy`:

```
x = np.arange(0.0, 5, 0.01)
y = np.cos(x*np.pi)
y_masked = np.ma.masked_where(y < -0.5, y)
plt.ylim(-1, 1)
plt.plot(x, y_masked, linewidth=3)
```

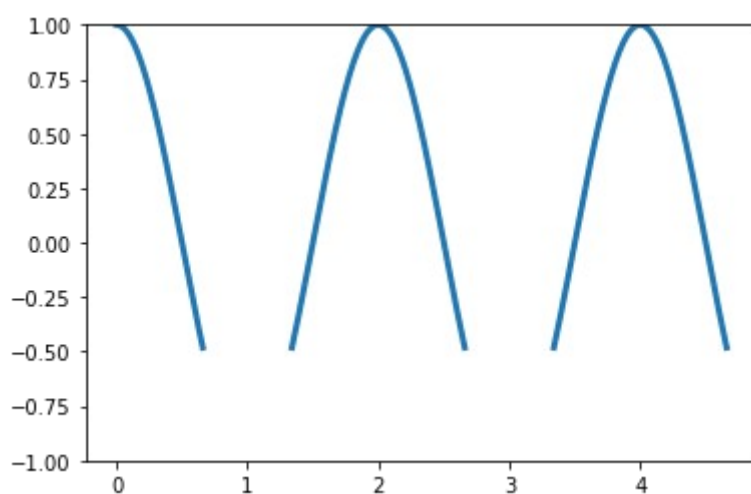


Рисунок 4.10 — Пример обрезки графика

## 4.2 Ступенчатый, стековый, точечный график и другие

### 4.2.1 Ступенчатый график

Рассмотрим еще один график - ступенчатый. Такой график строится с помощью функции `step()`:

```
step(x, y, [fmt], *, data=None, where='pre', **kwargs)
```

которая принимает следующий набор параметров:

- `x`: массив
  - Набор данных для оси абсцисс.
- `y`: массив
  - Набор данных для оси ординат.
- `fmt`: str, optional
  - Формат линии (см. функцию `plot()`).
- `data`: индексруемый объект, optional
  - Метки.
- `where`: {'pre', 'post', 'mid'}, optional; значение по умолчанию: 'pre'
  - Определяет место, где будет установлен шаг:
    - 'pre': значение `y` ставится слева от значения `x`, т.е. значение `y[i]` определяется для интервала  $(x[i-1]; x[i])$ ;
    - 'post': значение `y` ставится справа от значения `x`, т.е. значение `y[i]` определяется для интервала  $(x[i]; x[i+1])$ ;
    - 'mid': значение `y` ставится в середине интервала.

```

x = np.arange(0, 7)
y = x

where_set = ['pre', 'post', 'mid']
fig, axs = plt.subplots(1, 3, figsize=(15, 4))

for i, ax in enumerate(axs):
    ax.step(x, y, 'g-o', where=where_set[i])
    ax.grid()

```

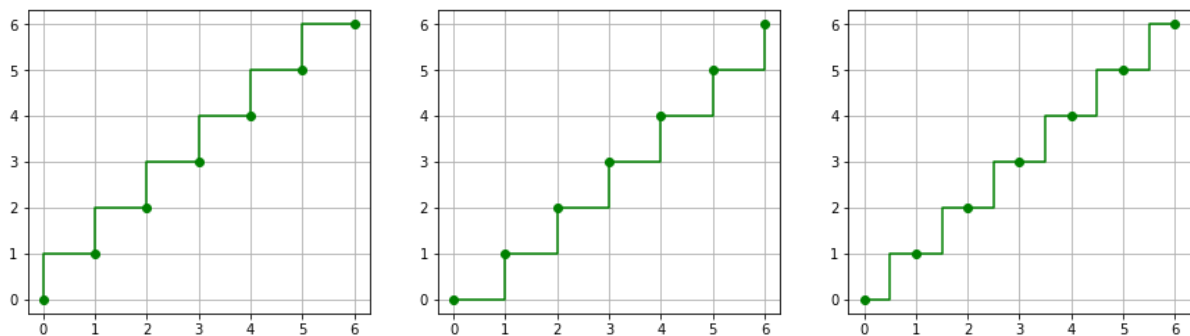


Рисунок 4.11 — Ступенчатый график

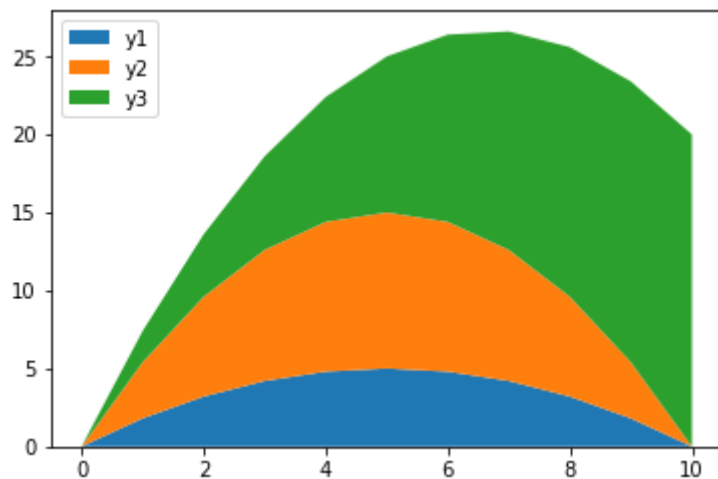
## 4.2.2 Стековый график

Для построения стекового графика используется функция `stackplot()`. Суть его в том, что графики отображаются друг над другом, и каждый следующий является суммой предыдущего и заданного:

```

x = np.arange(0, 11, 1)
y1 = np.array([(-0.2)*i**2+2*i for i in x])
y2 = np.array([(-0.4)*i**2+4*i for i in x])
y3 = np.array([2*i for i in x])
labels = ['y1', 'y2', 'y3']
fig, ax = plt.subplots()
ax.stackplot(x, y1, y2, y3, labels=labels)
ax.legend(loc='upper left')

```



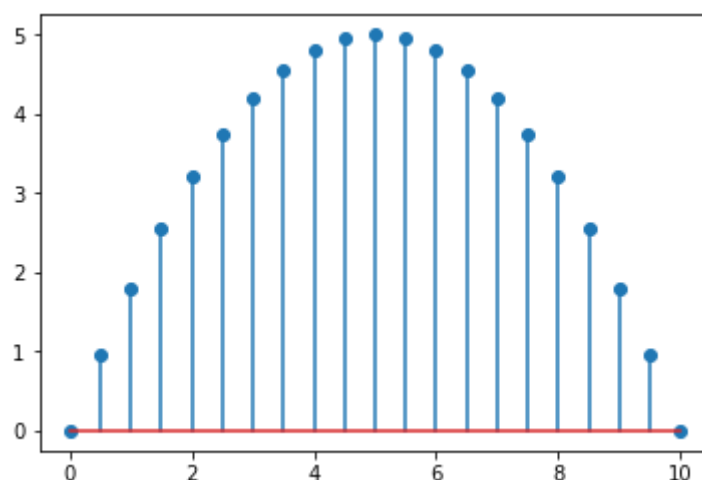
**Рисунок 4.12 — Стековый график**

Верхний край области  $y_2$  определяется как сумма значений из наборов  $y_1$  и  $y_2$ ,  $y_3$  - соответственно сумма  $y_1$ ,  $y_2$  и  $y_3$ .

### 4.2.3 Stem-график

Визуально этот график выглядит как набор линий от точки с координатами  $(x, y)$  до базовой линии, в верхней точке которой ставится маркер:

```
x = np.arange(0, 10.5, 0.5)
y = np.array([(-0.2)*i**2+2*i for i in x])
plt.stem(x, y)
```



**Рисунок 4.13 — Stem-график**

Дополнительные параметры функции `stem()`:

- `linefmt: str, optional`
  - Стиль вертикальной линии.

**Таблица 4.5 — Стиль вертикальной линии**

Символ	Стиль линии
' - '	Сплошная линия ( <i>solid line style</i> )
' -- '	Штриховая линия ( <i>dashed line style</i> )
' - . '	Штрих-пунктирная линия ( <i>dash-dot line style</i> )
' : '	Штриховая линия ( <i>dotted line style</i> )

- `markerfmt: str, optional`
  - Формат маркера.

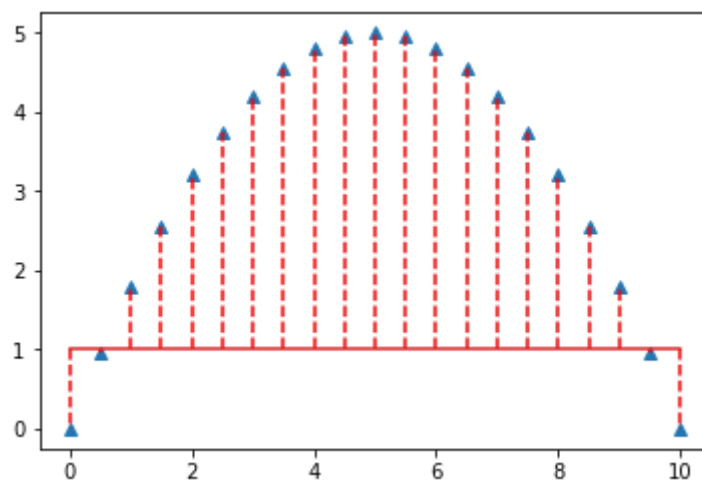
**Таблица 4.6 — Формат маркера**

Значение	Описание
' o '	Круг ( <i>Circle</i> )
' + '	Знак плюс ( <i>Plus sign</i> )
' * '	Звездочка ( <i>Asterisk</i> )
' . '	Точка ( <i>Point</i> )
' x '	Крест ( <i>Cross</i> )
' square ' или ' s '	Квадрат ( <i>Square</i> )
' diamond ' или ' d '	Ромб ( <i>Diamond</i> )
' ^ '	Треугольник, направленный вниз ( <i>triangle_down</i> )
' v '	Треугольник, направленный вверх ( <i>triangle_up</i> )
' < '	Треугольник, направленный влево ( <i>triangle_left</i> )
' > '	Треугольник, направленный вправо ( <i>triangle_right</i> )
' pentagram ' или ' p '	Пятиугольник ( <i>Five-pointed star (pentagram)</i> )
' hexagram ' или ' h '	Шестиугольник ( <i>Six-pointed star (hexagram)</i> )
' none '	Нет маркера ( <i>No markers</i> )

- `basefmt`: str, optional
  - Формат базовой линии.
- `bottom`: float, optional; значение по умолчанию: 0
  - $y$ -координата базовой линии.

Пример, демонстрирующий работу с дополнительными параметрами:

```
plt.stem(x, y, linefmt='r--', markerfmt='^', bottom=1)
```



**Рисунок 4.14 — Модифицированный *Stem*-график**

## 4.2.4 Точечный график

Для отображения точечного графика предназначена функция `scatter()`. В простейшем виде точечный график можно получить, передав функции `scatter()` наборы точек для  $x$ ,  $y$  координат:

```
x = np.arange(0, 10.5, 0.5)
y = np.cos(x)
plt.scatter(x, y)
```

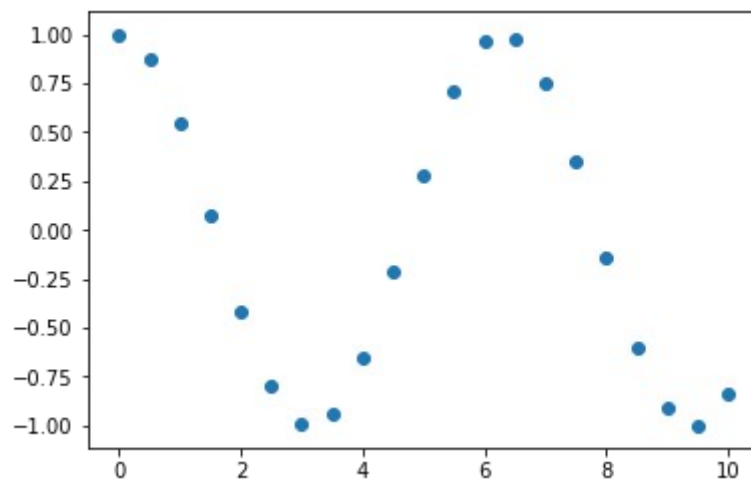


Рисунок 4.15 — Точечный график

Для более детальной настройки отображения необходимо воспользоваться дополнительными параметрами функции `scatter()`.

Сигнатура ее вызова имеет следующий вид:

```
scatter(x, y, s=None, c=None, marker=None, cmap=None, norm=None,
vmin=None, vmax=None, alpha=None, linewidths=None, verts=None,
edgecolors=None, *, plotnonfinite=False, data=None, **kwargs)
```

Рассмотрим некоторые из ее параметров:

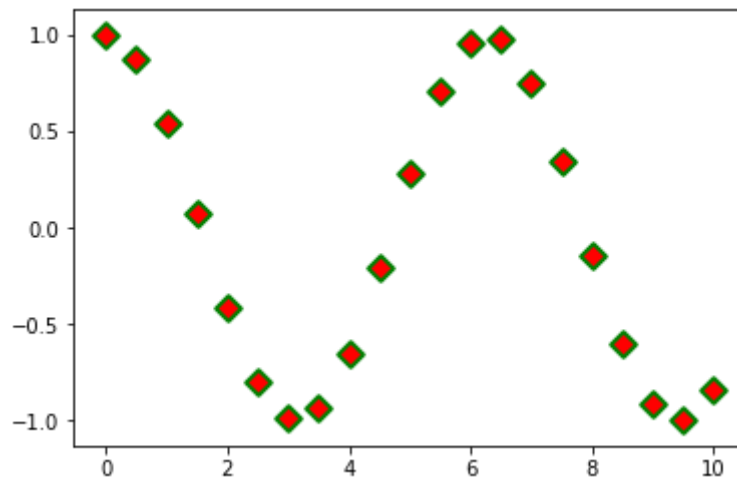
- $x$ : массив, `shape(n, )`
  - Набор данных для оси абсцисс.
- $y$ : массив, `shape(n, )`
  - Набор данных для оси ординат.



- s: скалярная величина или массив, shape(n, ), optional
  - Масштаб точек.
- c: color, набор color элементов, optional
  - Цвет.
- marker: [MarkerStyle](#), optional
  - Стилль точки объекта.
- cmap: [Colormap](#), optional, значение по умолчанию: None
  - Цветовая схема.
- norm: [Normalize](#), optional, значение по умолчанию: None
  - Нормализация данных.
- alpha: скалярная величина, optional, значение по умолчанию: None
  - Прозрачность.
- linewidths: скалярная величина или массив, optional, значение по умолчанию: None
  - Ширина границы маркера.
- edgecolors: {'face', 'none', None}, color или набор color элементов, optional.
  - Цвет границы.

Пример использования параметров функции `scatter()`:

```
x = np.arange(0, 10.5, 0.5)
y = np.cos(x)
plt.scatter(x, y, s=80, c='r', marker='D', linewidths=2, edgecolors='g')
```



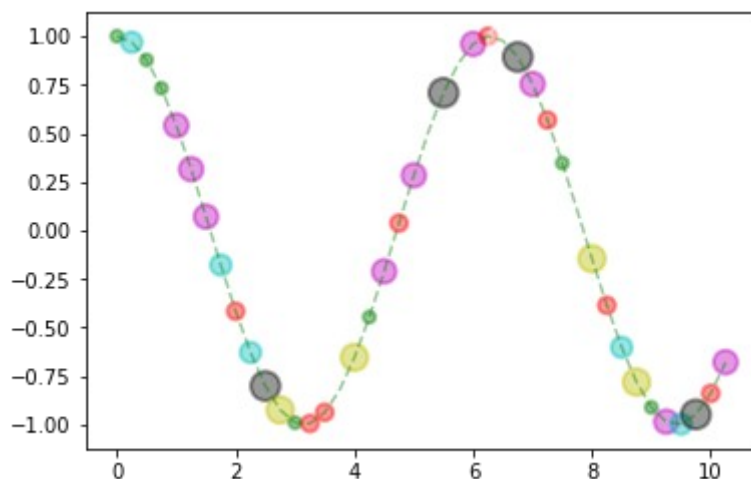
**Рисунок 4.16 — Модифицированный точечный график**

Пример, демонстрирующий работу с цветом и размером:

```
import matplotlib.colors as mcolors
bc = mcolors.BASE_COLORS

x = np.arange(0, 10.5, 0.25)
y = np.cos(x)
num_set = np.random.randint(1, len(mcolors.BASE_COLORS), len(x))
sizes = num_set * 35
colors = [list(bc.keys())[i] for i in num_set]

plt.scatter(x, y, s=sizes, alpha=0.4, c=colors, linewidths=2,
            edgecolors='face')
plt.plot(x, y, 'g--', alpha=0.4)
```



**Рисунок 4.17 — Модифицированный точечный график**

## 4.2.5 Дополнительные варианты работы с точечными графиками

Создание собственных маркеров:

[https://matplotlib.org/gallery/lines\\_bars\\_and\\_markers/scatter\\_custom\\_symbol.html#sphx-glr-gallery-lines-bars-and-markers-scatter-custom-symbol-py](https://matplotlib.org/gallery/lines_bars_and_markers/scatter_custom_symbol.html#sphx-glr-gallery-lines-bars-and-markers-scatter-custom-symbol-py)

[https://matplotlib.org/gallery/lines\\_bars\\_and\\_markers/scatter\\_piecharts.html#sphx-glr-gallery-lines-bars-and-markers-scatter-piecharts-py](https://matplotlib.org/gallery/lines_bars_and_markers/scatter_piecharts.html#sphx-glr-gallery-lines-bars-and-markers-scatter-piecharts-py)

Точный график с гистограммой распределения:

[https://matplotlib.org/gallery/lines\\_bars\\_and\\_markers/scatter\\_hist.html#sphx-glr-gallery-lines-bars-and-markers-scatter-hist-py](https://matplotlib.org/gallery/lines_bars_and_markers/scatter_hist.html#sphx-glr-gallery-lines-bars-and-markers-scatter-hist-py)

Задание отображения точек в зависимости от региона:

[https://matplotlib.org/gallery/lines\\_bars\\_and\\_markers/scatter\\_masked.html#sphx-glr-gallery-lines-bars-and-markers-scatter-masked-py](https://matplotlib.org/gallery/lines_bars_and_markers/scatter_masked.html#sphx-glr-gallery-lines-bars-and-markers-scatter-masked-py)

Работа с легендой:

[https://matplotlib.org/gallery/lines\\_bars\\_and\\_markers/scatter\\_with\\_legend.html#sphx-glr-gallery-lines-bars-and-markers-scatter-with-legend-py](https://matplotlib.org/gallery/lines_bars_and_markers/scatter_with_legend.html#sphx-glr-gallery-lines-bars-and-markers-scatter-with-legend-py)

## 4.3 Столбчатые и круговые диаграммы

### 4.3.1 Столбчатые диаграммы

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

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

Построим простую диаграмму:

```
np.random.seed(123)
groups = [f'P{i}' for i in range(7)]
counts = np.random.randint(3, 10, len(groups))
plt.bar(groups, counts)
```

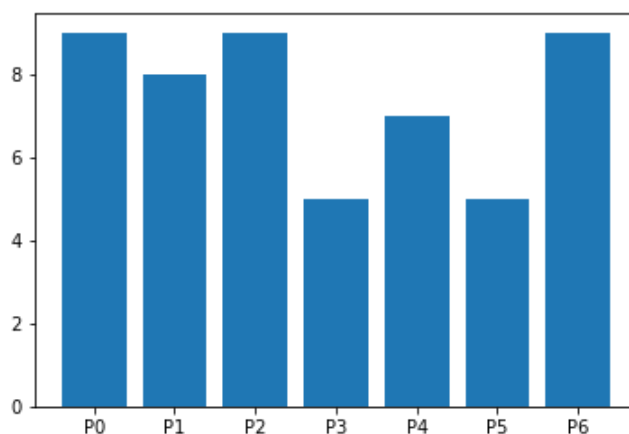
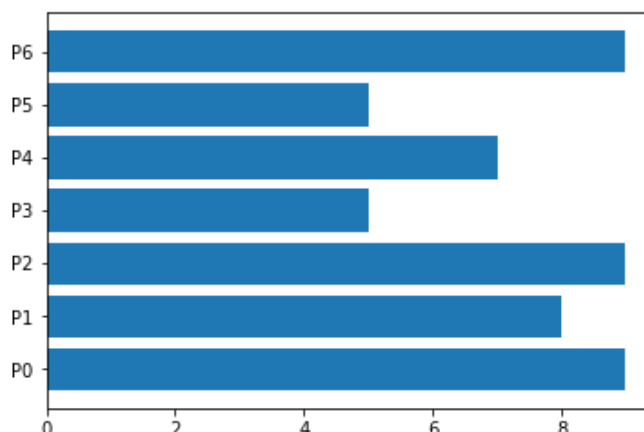


Рисунок 4.18 — Столбчатая диаграмма

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

```
plt.barh(groups, counts)
```



**Рисунок 4.19 — Горизонтальная столбчатая диаграмма**

Рассмотрим более подробно параметры функции `bar()`:

Основные параметры:

- `x`: массив
  - `x`-координаты столбцов.
- `height`: скалярная величина или массив
  - Высоты столбцов.
- `width`: скалярная величина, массив или `optional`
  - Ширина столбцов.
- `bottom`: скалярная величина, массив или `optional`
  - `y`-координата базы.
- `align`: {'center', 'edge'}, `optional`; значение по умолчанию: 'center'
  - Выравнивание по координате `x`.

Дополнительные параметры:

- `color`: скалярная величина, массив или `optional`
  - Цвет столбцов диаграммы.

- `edgecolor`: скалярная величина, массив или `optional`
  - Цвет границы столбцов.
- `linewidth`: скалярная величина, массив или `optional`
  - Ширина границы.
- `tick_label`: `str`, массив или `optional`
  - Метки для столбца.
- `xerr`, `yerr`: скалярная величина, массив размера `shape(N,)`, `shape(2, N)` или `optional`
  - Величина ошибки для графика. Выставленное значение прибавляется/удаляется к верхней (правой - для горизонтального графика) границе. Может принимать следующие значения:
    - скаляр: симметрично +/- для всех баров;
    - `shape(N,)`: симметрично +/- для каждого бара;
    - `shape(2,N)`: выборочного - и + для каждого бара. Первая строка содержит нижние значения ошибок, вторая строка — верхние;
    - `None`: не отображать значения ошибок. Это значение используется по умолчанию.
- `ecolor`: скалярная величина, массив или `optional`; значение по умолчанию: `'black'`
  - Цвет линии ошибки.
- `log`: `bool`, `optional`; значение по умолчанию: `False`
  - Включение логарифмического масштаба для оси `y`.
- `orientation`: `{'vertical', 'horizontal'}`, `optional`
  - Ориентация: вертикальная или горизонтальная.

Пример, демонстрирующий работу с параметрами `bar()`:

```
import matplotlib.colors as mcolors
bc = mcolors.BASE_COLORS

np.random.seed(123)
groups = [f'P{i}' for i in range(7)]
counts = np.random.randint(0, len(bc), len(groups))
width = counts*0.1
colors = [[ 'r', 'b', 'g' ][int(np.random.randint(0, 3, 1))]] for _ in
counts]

plt.bar(groups, counts, width=width, alpha=0.6, bottom=2, color=colors,
edgecolor='k', linewidth=2)
```

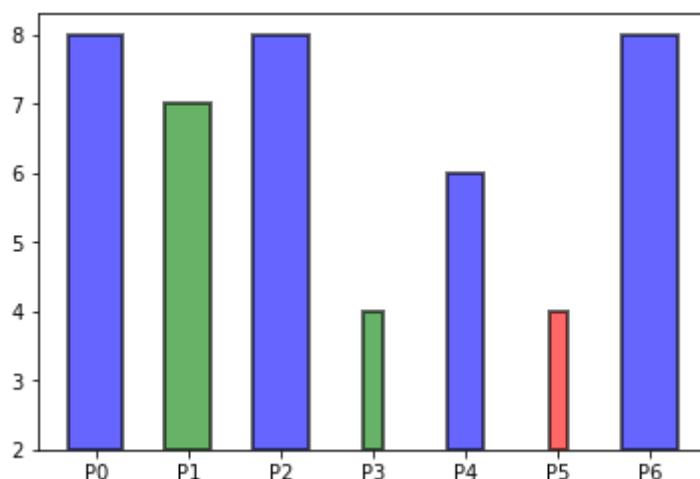


Рисунок 4.20 — Модифицированная столбчатая диаграмма

#### 4.3.1.1 Групповые столбчатые диаграммы

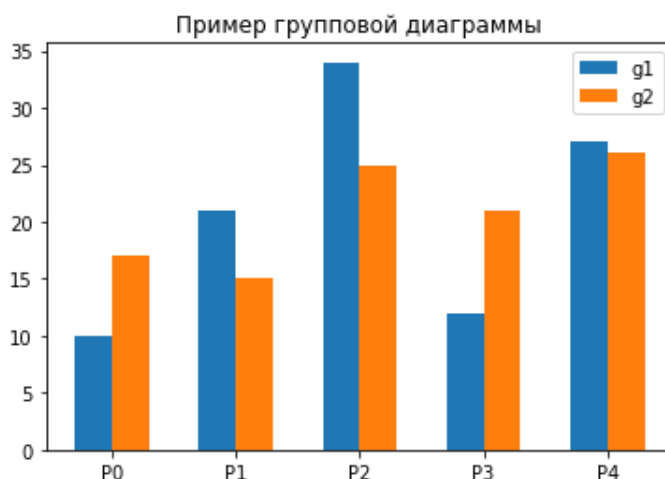
Используя определенным образом подготовленные данные можно строить групповые диаграммы:

```
cat_par = [f'P{i}' for i in range(5)]
g1 = [10, 21, 34, 12, 27]
g2 = [17, 15, 25, 21, 26]
width = 0.3
x = np.arange(len(cat_par))
```

```

fig, ax = plt.subplots()
rects1 = ax.bar(x - width/2, g1, width, label='g1')
rects2 = ax.bar(x + width/2, g2, width, label='g2')
ax.set_title('Пример групповой диаграммы')
ax.set_xticks(x)
ax.set_xticklabels(cat_par)
ax.legend()

```



**Рисунок 4.21 — Групповая столбчатая диаграмма**

#### **4.3.1.2 Диаграмма с *errorbar* элементом**

*Errorbar* элемент позволяет задать величину ошибки для каждого элемента графика. Для этого используются параметры `xerr`, `yerr` и `ecolor`, первые два определяют величину ошибки, последний - цвет:

```

np.random.seed(123)
rnd = np.random.randint
cat_par = [f'P{i}' for i in range(5)]
g1 = [10, 21, 34, 12, 27]
error = np.array([[rnd(2,7),rnd(2,7)] for _ in range(len(cat_par))]).T

```



```
fig, axs = plt.subplots(1, 2, figsize=(10, 5))
axs[0].bar(cat_par, g1, yerr=5, ecolor='r', alpha=0.5, edgecolor='b',
linewidth=2)
axs[1].bar(cat_par, g1, yerr=error, ecolor='r', alpha=0.5, edgecolor='b',
linewidth=2)
```

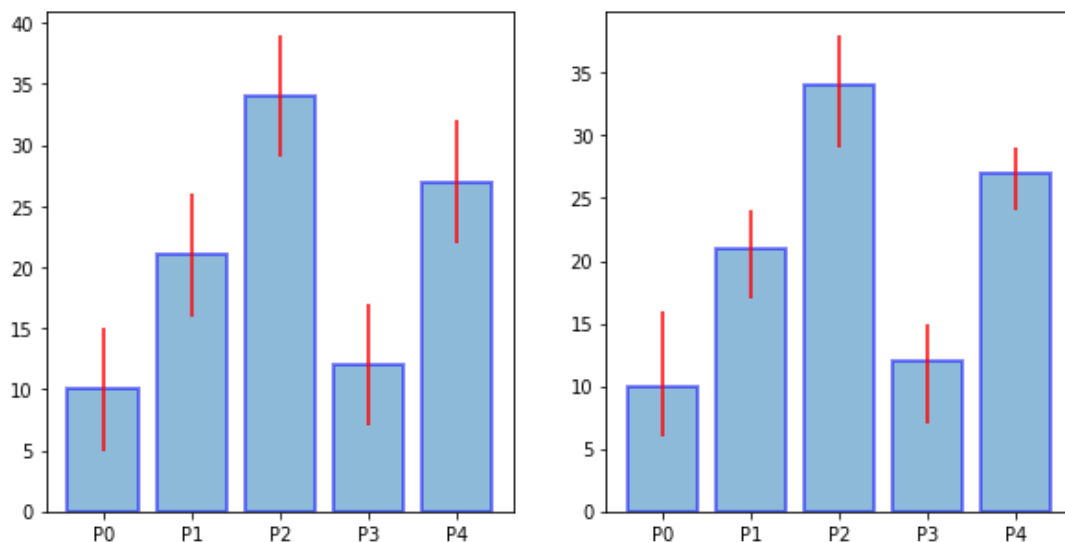


Рисунок 4.22 — Столбчатая диаграмма с *errorbar* элементом

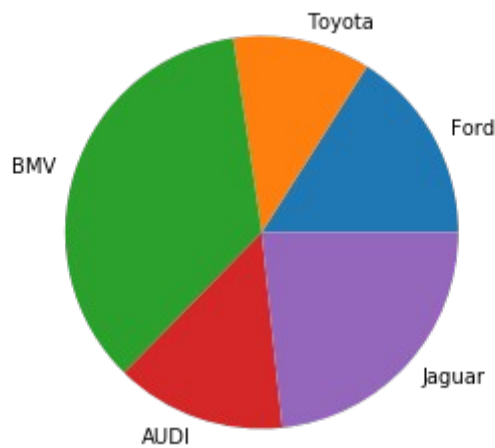
## 4.3.2 Круговые диаграммы

### 4.3.2.1 Классическая круговая диаграмма

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

Пример построения диаграммы:

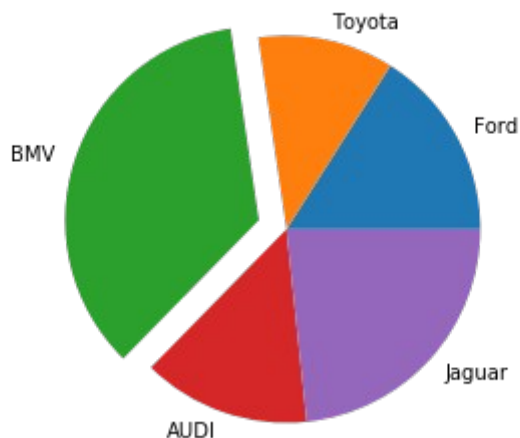
```
vals = [24, 17, 53, 21, 35]
labels = ['Ford', 'Toyota', 'BMW', 'AUDI', 'Jaguar']
fig, ax = plt.subplots()
ax.pie(vals, labels=labels)
ax.axis('equal')
```



**Рисунок 4.23 — Круговая диаграмма**

Рассмотрим параметры функции `pie()`:

- `x`: массив
  - Массив с размерами долей.
- `explode`: массив, *optional*; значение по умолчанию: `None`
  - Если параметр не равен `None`, то часть долей, которые перечислены в передаваемом значении, будут вынесены из диаграммы на заданное расстояние, пример диаграммы:



**Рисунок 4.24 — Круговая диаграмма с отделенным сектором**

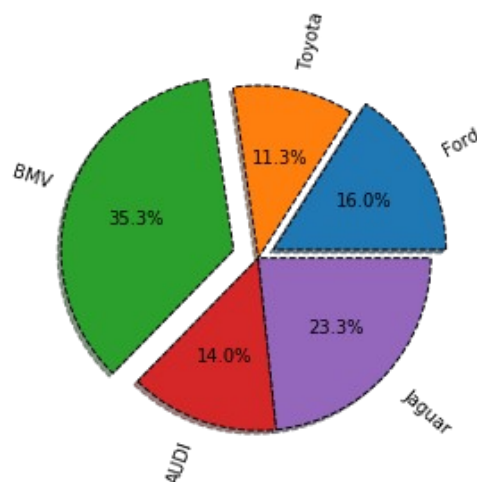
- `labels`: *list*, *optional*; значение по умолчанию: `None`
  - Текстовые метки долей.

- `colors`: массив, `optional`; значение по умолчанию: `None`
  - Цвета долей.
- `autopct`: `str`, функция, `optional`; значение по умолчанию: `None`
  - Формат текстовой метки внутри доли, текст - это численное значение показателя, связанного с конкретной долей.
- `pctdistance`: `float`, `optional`; значение по умолчанию: `0.6`
  - Расстояние между центром каждой доли и началом текстовой метки, которая определяется параметром `autopct`.
- `shadow`: `bool`, `optional`, значение по умолчанию: `False`
  - Отображение тени для диаграммы.
- `labeldistance`: `float`, `None`, `optional`; значение по умолчанию: `1.1`
  - Расстояние, на котором будут отображены текстовые метки долей. Если параметр равен `None`, то метки не будут отображены.
- `startangle`: `float`, `optional`; значение по умолчанию: `None`
  - Задаёт угол, на который нужно повернуть диаграмму против часовой стрелки относительно оси `x`.
- `radius`: `float`, `optional`; значение по умолчанию: `None`
  - Величина радиуса диаграммы.
- `counterclock`: `bool`, `optional`; значение по умолчанию: `True`
  - Определяет направление вращения: по часовой или против часовой стрелки.
- `wedgeprops`: `dict`, `optional`; значение по умолчанию: `None`
  - Словарь параметров, определяющих внешний вид долей (см. класс [matplotlib.patches.Wedge](#)).
- `textprops`: `dict`, `optional`; значение по умолчанию: `None`
  - Словарь параметров, определяющих внешний вид текстовых меток (см. класс [matplotlib.text.Text](#)).

- `center`: list значений float, optional; значение по умолчанию: (0, 0)
  - Центр диаграммы.
- `frame`: bool, optional; значение по умолчанию: False
  - Если параметр равен True, то вокруг диаграммы будет отображена рамка.
- `rotatelabels`: bool, optional; значение по умолчанию: False
  - Если параметр равен True, то текстовые метки будут повернуты на заданный угол.

Пример, демонстрирующий работу с параметрами функции `pie()`:

```
vals = [24, 17, 53, 21, 35]
labels = ['Ford', 'Toyota', 'BMW', 'AUDI', 'Jaguar']
explode = (0.1, 0, 0.15, 0, 0)
fig, ax = plt.subplots()
ax.pie(vals, labels=labels, autopct='%1.1f%%', shadow=True,
explode=explode, wedgeprops={'lw':1, 'ls':'--', 'edgecolor':'k'},
rotatelabels=True)
ax.axis('equal')
```



**Рисунок 4.24 — Модифицированная круговая диаграмма**

### 4.3.2.2 Вложенные круговые диаграммы

Вложенная круговая диаграмма состоит из двух компонент: внутренняя ее часть является детальным представлением информации, а внешняя - суммарной по заданным областям. Каждая область представляет собой список численных значений, вместе они образуют общий набор данных.

Пример:

```
fig, ax = plt.subplots()
offset=0.4
data = np.array([[5, 10, 7], [8, 15, 5], [11, 9, 7]])
cmap = plt.get_cmap('tab20b')
b_colors = cmap(np.array([0, 8, 12]))
sm_colors = cmap(np.array([1, 2, 3, 9, 10, 11, 13, 14, 15]))
ax.pie(data.sum(axis=1), radius=1, colors=b_colors,
wedgeprops=dict(width=offset, edgecolor='w'))
ax.pie(data.flatten(), radius=1-offset, colors=sm_colors,
wedgeprops=dict(width=offset, edgecolor='w'))
```



Рисунок 4.25 — Вложенная круговая диаграмма

### 4.3.2.3 Круговая диаграмма с отверстием

Построим круговую диаграмму в виде бублика (с отверстием посередине). Это можно сделать через параметр `wedgeprops`, который отвечает за внешний вид долей:

```
vals = [24, 17, 53, 21, 35]
labels = ['Ford', 'Toyota', 'BMV', 'AUDI', 'Jaguar']
fig, ax = plt.subplots()
ax.pie(vals, labels=labels, wedgeprops=dict(width=0.5))
```

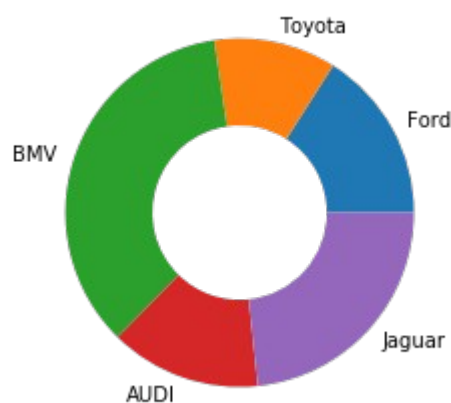


Рисунок 4.26 — Круговая диаграмма с отверстием

## 4.4 Цветовая сетка

Цветовая сетка представляет собой поле, заполненное цветом, который определяется цветовой картой и численными значениями элементов переданного двумерного массива.

### 4.4.1 Цветовые карты (*colormaps*)

Цветовая карта - это подготовленный набор цветов, который хорошо подходит для визуализации того или иного набора данных. Подробное руководство по цветовым картам вы можете найти на официальном сайте *Matplotlib*. Такую карту можно создать самостоятельно, если среди существующих нет подходящей. Ниже представлены примеры некоторых цветовых карт из библиотеки *Matplotlib*.

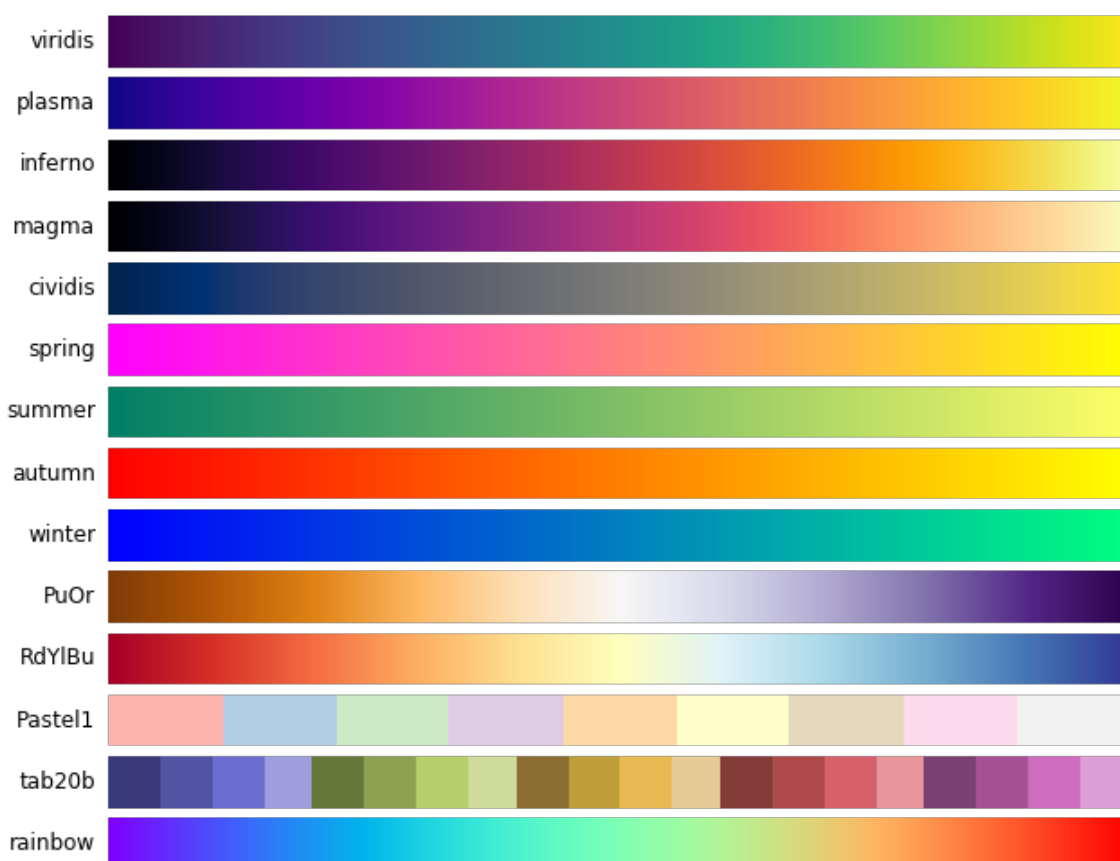


Рисунок 4.27 — Цветовые карты

### 4.4.2 Построение цветовой сетки

Рассмотрим две функции для построения цветовой сетки: `imshow()` и `pcolormesh()`.

#### `imshow()`

Основное назначение функции `imshow()` состоит в представлении *2D* растров. Это могут быть картинки, двумерные массивы данных, матрицы и т. п.

Напишем простую программу, которая загружает картинку из интернета по заданному *URL* и отображает ее с использованием библиотеки *Matplotlib*:

```
from PIL import Image
import requests
from io import BytesIO
response = requests.get('https://matplotlib.org/_static/logo2.png')
img = Image.open(BytesIO(response.content))
plt.imshow(img)
```

В результате получим изображение логотипа *Matplotlib*.

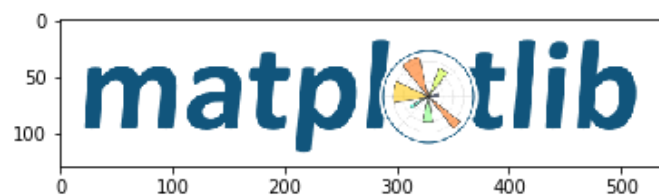
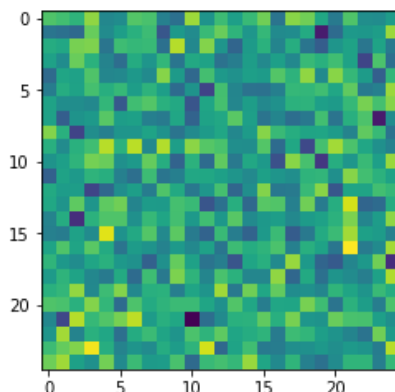


Рисунок 4.28 — Изображение логотипа *Matplotlib*



Создадим двумерный набор данных и отобразим его с помощью `imshow()`:

```
np.random.seed(19680801)
data = np.random.randn(25, 25)
plt.imshow(data)
```



**Рисунок 4.29 — Визуализация двумерного набора данных с использованием `imshow()`**

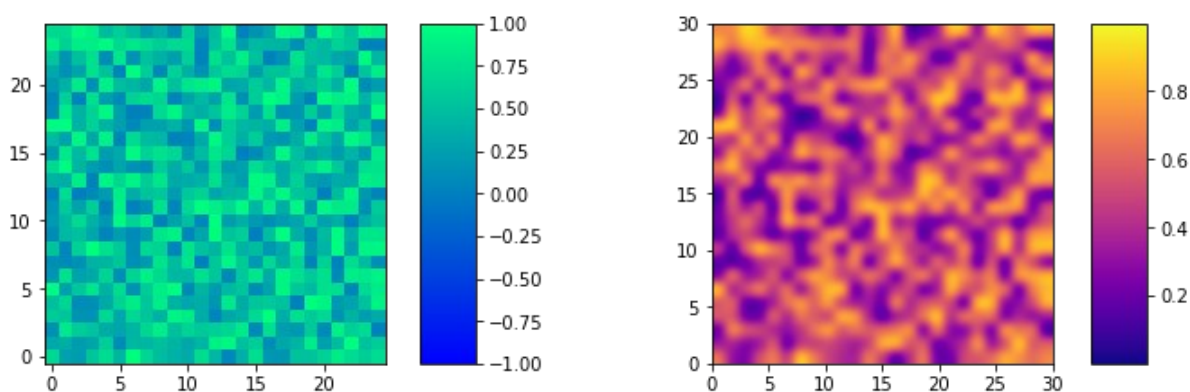
Рассмотрим некоторые из параметров функции `imshow()`:

- `X`: массив или PIL изображение
  - Поддерживаются следующие размерности массивов:
    - $(M, N)$ : двумерный массив со скалярными данными.
    - $(M, N, 3)$ : массив с *RGB* значениями (0-1 float или 0-255 int).
    - $(M, N, 4)$ : массив с *RGBA* значениями (0-1 float или 0-255 int).
- `cmap`: str или [Colormap](#), optional
  - Цветовая карта изображения (см. "4.4.1 Цветовые карты (*colormaps*)")
- `norm`: [Normalize](#), optional
  - Нормализация - приведение скалярных данных к диапазону  $[0,1]$  перед использованием `cmap`. Этот параметр игнорируется для *RGB(A)* данных.

- aspect: {'equal', 'auto'} или float, optional
  - 'equal': обеспечивает соотношение сторон равное 1;
  - 'auto': соотношение не изменяется.
- interpolation: str, optional
  - Алгоритм интерполяции. Доступны следующие значения: 'none', 'nearest', 'bilinear', 'bicubic', 'spline16', 'spline36', 'hanning', 'hamming', 'hermite', 'kaiser', 'quadric', 'catrom', 'gaussian', 'bessel', 'mitchell', 'sinc', 'lanczos'.
- alpha: скалярное значение, optional
  - Прозрачность. Определяется в диапазоне от 0 до 1. Параметр игнорируется для *RGBA* значения.
- vmin, vmax: скалярное значение, optional
  - Численные значения vmin и vmax (если параметр norm не задан явно) определяют диапазон данных, который будет покрыт цветовой картой. По умолчанию цветовая карта охватывает весь диапазон значений отображаемых данных. Если используется параметр norm, то vmin и vmax игнорируются.
- origin: {'upper', 'lower'}, optional
  - Расположение начал координат (точки [0,0]): 'upper' - верхний левый, 'lower' - нижний левый угол координатной плоскости.
- extent: (left, right, bottom, top), optional
  - Изменение размеров изображения вдоль осей x, y.
- filterrad: float > 0, optional; значение по умолчанию: 4.0
  - Параметр filter radius для фильтров, которые его используют, например: 'sinc', 'lanczos' или 'blackman'.

Пример, использующий параметры из приведенного выше списка:

```
fig, axs = plt.subplots(1, 2, figsize=(10,3), constrained_layout=True)
p1 = axs[0].imshow(data, cmap='winter', aspect='equal', vmin=-1, vmax=1,
origin='lower')
fig.colorbar(p1, ax=axs[0])
p2 = axs[1].imshow(data, cmap='plasma', aspect='equal',
interpolation='gaussian', origin='lower', extent=(0, 30, 0, 30))
fig.colorbar(p2, ax=axs[1])
```



**Рисунок 4.29 — Варианты визуализации двумерного набора данных**

### **pcolormesh()**

Ещё одной функцией для визуализации *2D* наборов данных является `pcolormesh()`. В библиотеке *Matplotlib* есть ещё одна функция с аналогичным функционалом - `pcolor()`, в отличие от нее, рассматриваемая нами `pcolormesh()`, более быстрая и является лучшим вариантом в большинстве случаев. Функция `pcolormesh()` похожа по своим возможностям на `imshow()`, но есть и отличия.

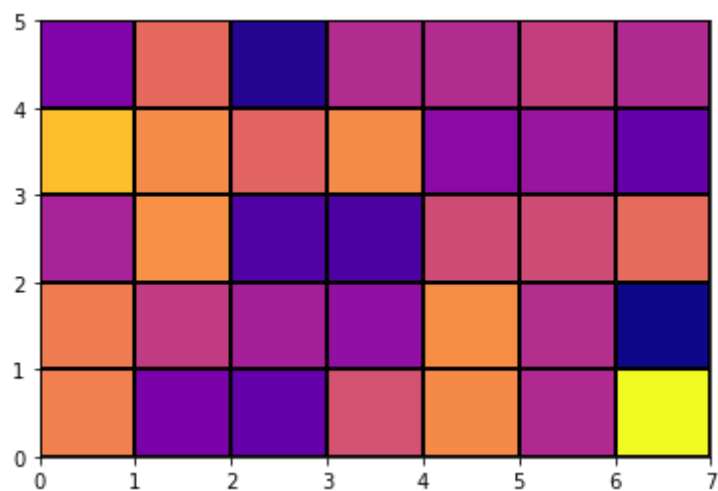
Рассмотрим параметры функции `pcolormesh()`:

- **C:** массив
  - *2D* массив скалярных значений.

- `cmap`: str или [Colormap](#), optional
  - См. `cmap` в `imshow()`.
- `norm`: [Normalize](#), optional
  - См. `norm` в `imshow()`.
- `vmin, vmax`: scalar, optional; значение по умолчанию: `None`
  - См. `vmin, vmax` в `imshow()`.
- `edgecolors`: {'none', `None`, 'face', color, color sequence}, optional; значение по умолчанию: 'none'
  - Цвет границы. Возможны следующие варианты:
    - 'none' или '': без отображения границы;
    - `None`: черный цвет;
    - 'face': используется цвет ячейки;
    - Можно выбрать цвет из доступных наборов.
- `alpha`: scalar, optional; значение по умолчанию: `None`
  - См. `alpha` в `imshow()`.
- `shading`: {'flat', 'gouraud'}, optional
  - Стилль заливки. Доступные значения:
    - 'flat': сплошной цвет заливки для каждого квадрата;
    - 'gouraud': для каждого квадрата будет использован метод затенения *Gouraud*.
- `snap`: bool, optional; значение по умолчанию: `False`
  - Привязка сетки к границам пикселей.

Пример использования функции `pcolormesh()`:

```
np.random.seed(123)
data = np.random.rand(5, 7)
plt.pcolormesh(data, cmap='plasma', edgecolors='k', shading='flat')
```



**Рисунок 4.30 — Визуализация двумерного набора данных с использованием `pcolormesh()`**

## Урок 5. Построение 3D графиков. Работа с *mplot3d Toolkit*

До этого момента все графики, которые мы строили были двумерные, *Matplotlib* позволяет строить 3D графики. Импортируем необходимые модули для работы с 3D:

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

Рассмотрим некоторые из инструментов для построения 3D графиков.

### 5.1 Линейный график

Для построения линейного графика используется функция `plot()`:

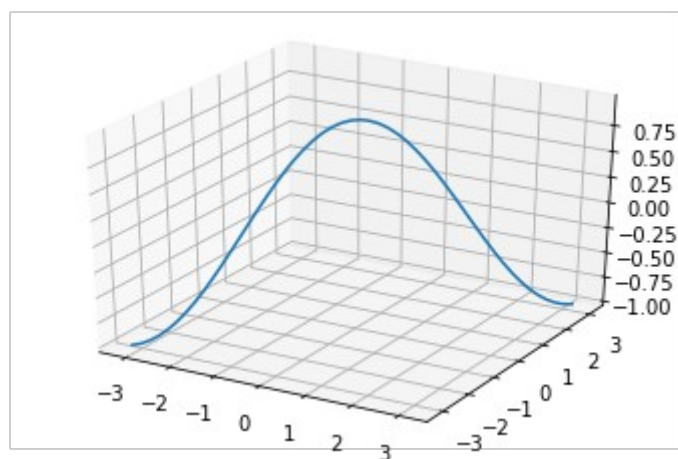
```
Axes3D.plot(self, xs, ys, *args, zdir='z', **kwargs)
```

Параметры функции *Axes3D.plot*:

- `xs`: 1D массив
  - `x` координаты.
- `ys`: 1D массив
  - `y` координаты.
- `zs`: скалярное значение или 1D массив
  - `z` координаты. Если передан скаляр, то он будет присвоен всем точкам графика.
- `zdir`: {'x', 'y', 'z'}; значение по умолчанию: 'z'
  - Определяет ось, которая будет принята за `z` направление.
- `**kwargs`
  - Дополнительные аргументы, аналогичные тем, что используются в функции `plot()` для построения двумерных графиков.

```
x = np.linspace(-np.pi, np.pi, 50)
y = x
z = np.cos(x)

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot(x, y, z, label='parametric curve')
```



**Рисунок 5.1 — Линейный 3D-график**

## 5.2 Точечный график

Для построения точечного графика используется функция `scatter()`:

```
Axes3D.scatter(self, xs, ys, zs=0, zdir='z', s=20, c=None,  
depthshade=True, *args, **kwargs)
```

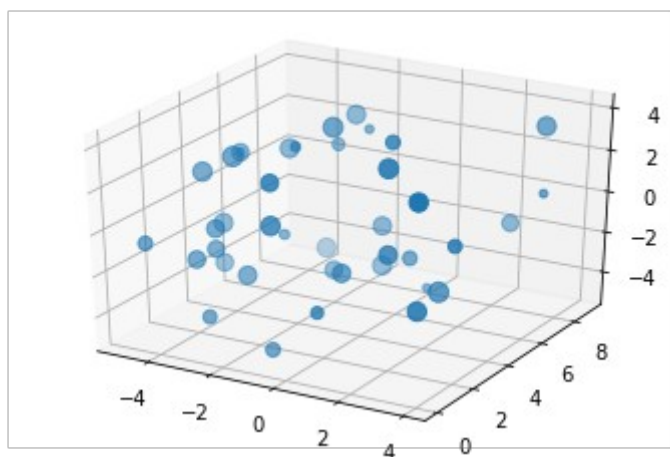
Параметры функции `Axes3D.scatter()`:

- `xs, ys`: массив
  - Координаты точек по осям  $x$  и  $y$ .
- `zs`: `float` или массив, `optional`; значение по умолчанию: 0
  - Координаты точек по оси  $z$ . Если передан скаляр, то он будет присвоен всем точкам графика.
- `zdir`: `{'x', 'y', 'z', '-x', '-y', '-z'}`, `optional`; значение по умолчанию: `'z'`
  - Определяет ось, которая будет принята за  $z$  направление.
- `s`: скаляр или массив, `optional`; значение по умолчанию: 20
  - Размер маркера.
- `c`: `color`, массив, массив значений цвета, `optional`
  - Цвет маркера. Возможные значения:
    - строковое значение цвета для всех маркеров;
    - массив строковых значений цвета;
    - массив чисел, которые могут быть отображены в цвета через функции `cm` и `norm`;
    - 2D массив, элементами которого являются *RGB* или *RGBA*;
- `depthshade`: `bool`, `optional`
  - Затенение маркеров для придания эффекта глубины.
- `**kwargs`
  - Дополнительные аргументы, аналогичные тем, что используются в функции `scatter()` для построения двумерных графиков.



```
np.random.seed(123)
x = np.random.randint(-5, 5, 40)
y = np.random.randint(0, 10, 40)
z = np.random.randint(-5, 5, 40)
s = np.random.randint(10, 100, 20)

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(x, y, z, s=s)
```



**Рисунок 5.2— Точечный 3D-график**

## 5.3 Каркасная поверхность

Для построения каркасной поверхности используется функция `plot_wireframe()`:

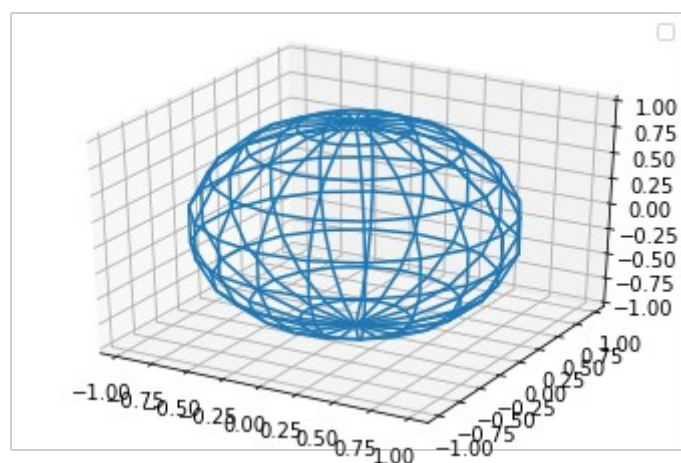
```
plot_wireframe(self, X, Y, Z, *args, **kwargs)
```

Параметры функции `Axes3D.wireframe()`:

- `X, Y, Z`: 2D массивы
  - Данные для построения поверхности.
- `rcount, ccount`: int, значение по умолчанию: 50
  - Максимальное количество элементов каркаса, которое будет использовано в каждом из направлений.
- `rstride, cstride`: int
  - Параметры, определяющие величину шага, с которым будут браться элементы строки / столбца из переданных массивов. Параметры `rstride`, `cstride` и `rcount`, `ccount` являются взаимоисключающими.
- `**kwargs`
  - Дополнительные аргументы, определяемые [Line3DCollection](#).

```
u, v = np.mgrid[0:2*np.pi:20j, 0:np.pi:10j]
x = np.cos(u)*np.sin(v)
y = np.sin(u)*np.sin(v)
z = np.cos(v)
```

```
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot_wireframe(x, y, z)
ax.legend()
```



**Рисунок 5.3— Каркасная поверхность**

## 5.4 Поверхность

Для построения поверхности используйте функцию `plot_surface()`:

```
plot_surface(self, X, Y, Z, *args, norm=None, vmin=None, vmax=None,
lightsource=None, **kwargs)
```

Параметры функции `Axes3D.plot_surface()`:

- `X, Y, Z`: 2D массивы
  - Данные для построения поверхности.
- `rcount, ccount`: int
  - см. `rcount, ccount` в “5.3 Каркасная поверхность”.
- `rstride, cstride`: int
  - см. `rstride, cstride` в “5.3 Каркасная поверхность”.
- `color`: color
  - Цвет для элементов поверхности.
- `cmap`: Colormap
  - Colormap для элементов поверхности.
- `facecolors`: массив элементов color
  - Индивидуальный цвет для каждого элемента поверхности.
- `norm`: Normalize
  - Нормализация для colormap.
- `vmin, vmax`: float
  - Границы нормализации.
- `shade`: bool; значение по умолчанию: True
  - Использование тени для facecolors.
- `lightsource`: [LightSource](#)
  - Объект класса `LightSource` - определяет источник света, используется, только если `shade = True`.
- `**kwargs`
  - Дополнительные аргументы, определяемые [Poly3DCollection](#).

```

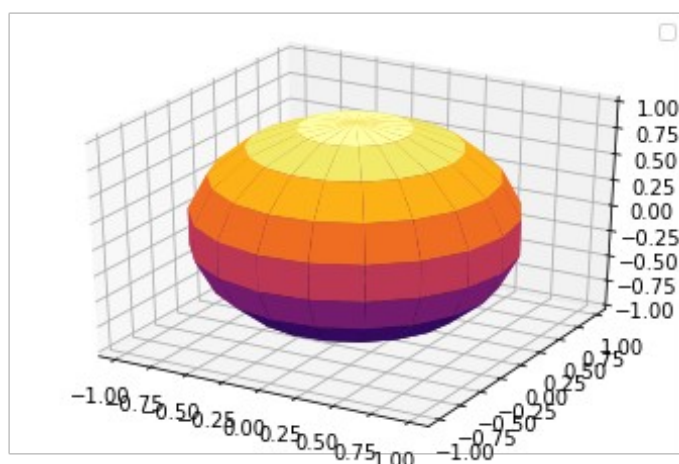
u, v = np.mgrid[0:2*np.pi:20j, 0:np.pi:10j]
x = np.cos(u)*np.sin(v)
y = np.sin(u)*np.sin(v)
z = np.cos(v)

```

```

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(x, y, z, cmap='inferno')
ax.legend()

```



**Рисунок 5.4— Каркасная поверхность**