matplotlib Seaborn Mayavi

Python
Визуализация данных

Devpractice Team

УДК 004.6

ББК: 32.973

Devpractice Team. Python. Визуализация данных. Matplotlib. Seaborn. Mayavi. - devpractice.ru. 2020. - 412 с.: ил.

Данная книга посвящена библиотеками для визуализации данных на языке программирования *Python*: *Matplotlib*, *Seaborn*, *Mayavi*. По каждой библиотеке приведено подробное описание инструментов для визуализации данных, средств настройки внешнего вида и компоновки графиков.

УДК 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, 2020

© Абдрахманов М.И., 2020

Оглавление

Часть I. Библиотека *Matplotlib*......6
Введение......6
Глава 1. Быстрый старт......7
1.1 Установка......7
1.1.1 Варианты установки *Matplotlib*......7
1.1.2 Установка *Matplotlib* с помощью менеджера *pip*......7
1.1.3 Проверка установки......7
1.2 Быстрый старт......8
1.3 Построение графика......10
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
Глава 2. Основы работы с модулем *pyplot*......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......24
2.3.1 Стиль линии графика......24
2.3.2 Цвет линии......27
2.3.3 Тип графика......28
2.4 Размещение графиков отдельно друг от друга......30
2.4.1 Работа с функцией subplot()......30
2.4.2 Работа с функцией subplots()......33
Глава 3. Настройка элементов графика......34
3.1 Работа с легендой......34
3.1.1 Отображение легенды......34
3.1.2 Расположение легенды на графике......36
3.1.3 Дополнительные параметры настройки легенды......38
3.2 Компоновка графиков......40
3.2.1 Инструмент GridSpec......40
3.3 Текстовые элементы графика......45
3.3.1 Заголовок фигуры и поля графика......47
3.3.2 Подписи осей графика......48
3.3.3 Текстовый блок......50
3.3.4 Аннотация......52
3.4 Свойства класса Text......59
3.4.1 Параметры, отвечающие за отображение текста......59
3.4.2 Параметры, отвечающие за расположение надписи......62
3.4.3 Параметры, отвечающие за настройку заднего фона надписи......64
3.5 Цветовая полоса — *colorbar*......66
3.5.1 Общая настройка с использованием inset_locator()......68
3.5.2 Задание шкалы и установка надписи......70
3.5.3 Дополнительные параметры настройки цветовой полосы......71

Глава 4. Визуализация данных..73
4.1 Линейный график..73
4.1.1 Построение графика...73
4.1.1.1 Параметры аргумента fmt..75
4.1.2 Заливка области между графиком и осью..77
4.1.3 Настройка маркировки графиков..82
4.1.4 Обрезка графика..85
4.2 Ступенчатый, стековый, точечный и другие графики................................86
4.2.1 Ступенчатый график..86
4.2.2 Стековый график..87
4.2.3 *Stem*-график...88
4.2.4 Точечный график (Диаграмма рассеяния)...91
4.3 Столбчатые и круговые диаграммы..95
4.3.1 Столбчатые диаграммы...95
4.3.1.1 Групповые столбчатые диаграммы..98
4.3.1.2 Диаграмма с *errorbar*-элементом..99
4.3.2 Круговые диаграммы..100
4.3.2.1 Классическая круговая диаграмма...100
4.3.2.2 Вложенные круговые диаграммы...104
4.3.2.3 Круговая диаграмма с отверстием...105
4.4 Цветовая сетка...106
4.4.1 Цветовые карты (*colormaps*)...106
4.4.2 Построение цветовой сетки...107
Глава 5. Построение *3D*-графиков. Работа с *mplot3d Toolkit*...........................113
5.1 Линейный график..113
5.2 Точечный график (диаграмма рассеяния)..114
5.3 Каркасная поверхность..116
5.4 Поверхность...117
Часть II. Библиотека Seaborn...120
Введение...120
Глава 6. Быстрый старт..122
6.1 Установка...122
6.1.1 Варианты установки *seaborn*...122
6.1.2 Установка *seaborn* через менеджеры *pip* и *conda*..................................122
6.1.3 Проверка корректности установки...123
6.2 Быстрый старт..123
6.2.1 Построение точечного графика...124
6.2.2 Построение линейного графика...125
6.2.3 Работа с категориальными данными...126
Глава 7. Настройка внешнего вида графиков...129
7.1 Стили *seaborn*...129
7.2 Контексты *seaborn*..134
7.3 Настройка сетки и осей...139
7.3.1 Сетка...139
7.3.2 Поле и оси графика...141
7.4 Легенда..145
7.5 Шрифт..147
7.6 Работа с цветом...148
Глава 8. Визуализация отношений в данных...153
8.1 Общие параметры функций...153

8.1.1 Базовые аргументы....154
8.1.2 Параметры для повышения информативности графиков....154
8.2 Линейный график. Функция lineplot()....156
8.2.1 Знакомство с функцией lineplot()....157
8.2.2 Отображение математического ожидания и доверительных интервалов....160
8.2.3 Повышение информативности графика....164
8.2.3.1 Настройка цветовой схемы....165
8.2.3.2 Настройка стиля....166
8.2.3.3 Настройка толщины линии....168
8.2.4 Визуализация временных рядов....170
8.3 Диаграмма рассеяния. Функция scatterplot()....171
8.3.1 Знакомство с функцией scatterplot()....171
8.3.2 Повышение информативности графика *scatterplot*....172
8.3.2.1 Настройка цветовой схемы....173
8.3.2.2 Настройка стиля маркеров....175
8.3.2.3 Настройка размера маркера....177
8.4 Настройка внешнего вида элементов поля графика....179
8.4.1 Легенда....179
8.4.2 Подписи осей....182
8.4.3 Сортировка набора данных....183
8.5 Визуализация отношений с настройкой подложки. Функция relplot()....185
Глава 9. Визуализация категориальных данных....189
9.1 Общие параметры функций....190
9.1.1 Базовые параметры....190
9.1.2 Параметры для повышения информативности графиков....191
9.2 Визуализация категориальных данных в виде точечных диаграмм....192
9.2.1 Функция stripplot()....192
9.2.2 Функция swarmplot()....202
9.3 Визуализации распределений категориальных данных....205
9.3.1 Функция boxplot()....206
9.3.2 Функция violin()....214
9.4 Визуализация оценок категориальных данных....221
9.4.1 Функция pointplot()....222
9.4.2 Функция barplot()....228
9.4.3 Функция countplot()....233
9.5 Работа на уровне фигуры. Функция catplot()....236
Глава 10. Визуализация распределений в данных....246
10.1 Функция distplot()....246
10.2 Функция kdeplot()....253
10.3 Функция rugplot()....262
Глава 11. Визуализация модели линейной регрессии....265
11.1 Общие параметры функций....265
11.2 Функция regplot()....266
11.3 Функция residplot()....278
11.4 Функция lmplot()....281
Глава 12. Управление компоновкой диаграмм....288
12.1 *Facet*-сетка....288
12.2 *Pair*-сетка....296
12.2.1 Функция pairplot()....297

12.2.2 Класс PairPlot....302
12.3 *Joint*-сетка....306
12.3.1 Функция jointplot()....307
12.3.2 Класс JointPlot....313
Часть III. Библиотека *Mayavi*....315
Введение....315
Глава 13. Быстрый старт....317
13.1 Установка....317
13.2 Быстрый старт....319
13.2.1 Работа с *GUI* приложением *Mayavi2*....319
13.2.2 Разработка *Python*-модулей, использующих *Mayavi*....326
13.2.3 Работа с *Mayavi* в *Jupyter notebook*....328
Глава 14. Настройка представления....330
14.1 Управление Фигурой/Сценой....330
14.2 Настройка элементов оформления....336
14.2.1 Заголовок сцены....337
14.2.2 Внешний контур модели....339
14.2.3 Настройка осей координат....340
14.2.4 Настройка цветовой полосы (*colorbar*)....344
14.3 Управление камерой....346
Глава 15. Визуализация данных....349
15.1 Функции для работы с одномерными наборами данных....350
15.1.1 Функция points3d()....351
15.1.2 Функция plot3d()....355
15.2 Функции для работы с двумерными наборами данных....359
15.2.1 Функция imshow()....360
15.2.2 Функция surf()....362
15.2.3 Функция contour_surf()....366
15.2.4 Функция mesh()....368
15.3 Функции для работы с трехмерными наборами данных....370
15.3.1 Функция contour3d()....371
15.3.2 Функция quiver3d()....375
15.3.3 Функция volume_slice()....376
Глава 16. Работа с *pipeline*....379
16.1 Структура *pipline*....382
16.2 Работа с источниками данных....384
16.3 Работа с фильтрами....389
16.4 Работа с модулями....402
Заключение....412

Часть I. Библиотека *Matplotlib*

Введение

Библиотека *Matplotlib* является одним из самых популярных средств визуализации данных на *Python*. Она отлично подходит как для создания статичных изображений, так и анимированных, и интерактивных решений.

Matplotlib является частью *Scientific Python* — набора библиотек для научных вычислений и визуализации данных, куда также входят *NumPy*[1], *SciPy*[2], *Pandas*[3], *SymPy*[4] и ещё ряд других инструментов.

В этой книге будут рассмотрены вопросы визуализации данных, а именно построение линейных и ступенчатых графиков, диаграмм рассеяния, столбчатых и круговых диаграмм, гистограмм и *3D* графиков. Большое внимание уделено настройке внешнего вида графиков, их элементам и компоновке.

При описании параметров функций будет использоваться следующий формат:

- **`имя_аргумента: тип(ы)`**
  - **`описание`**

Если в описании типа данных есть слово `optional`, это значит, что данный параметр имеет значение по умолчанию, и его не обязательно явно указывать.

1 https://numpy.org/
2 https://scipy.org/
3 https://pandas.pydata.org/
4 https://www.sympy.org/en/index.html

Глава 1. Быстрый старт

1.1 Установка

1.1.1 Варианты установки *Matplotlib*

Существует два основных варианта установки *Matplotlib*: первый — это развёртывание пакета *Anaconda,* в состав которого входит интересующая нас библиотека, второй — установка *Matplotlib* с помощью менеджера пакетов. Про инсталляцию *Anaconda* можете прочитать на [devpractice.ru](https://devpractice.ru/python-lesson-1-install/)[5].

1.1.2 Установка *Matplotlib* с помощью менеджера *pip*

Для установки *Matplotlib* с помощью менеджера 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.

1.1.3 Проверка установки

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

```
import matplotlib
```

Если сообщений об ошибке не было, то значит библиотека *Matplotlib* установлена и её можно использовать.

5 https://devpractice.ru/python-lesson-1-install/

Проверим версию библиотеки, она, скорее всего, будет отличаться от приведённой ниже:

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

1.2 Быстрый старт

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

Если вы используете *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 1.1.

```
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 *Jupyter Notebook*

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 1.1, 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 изолированном окне (см. рисунок 1.2).

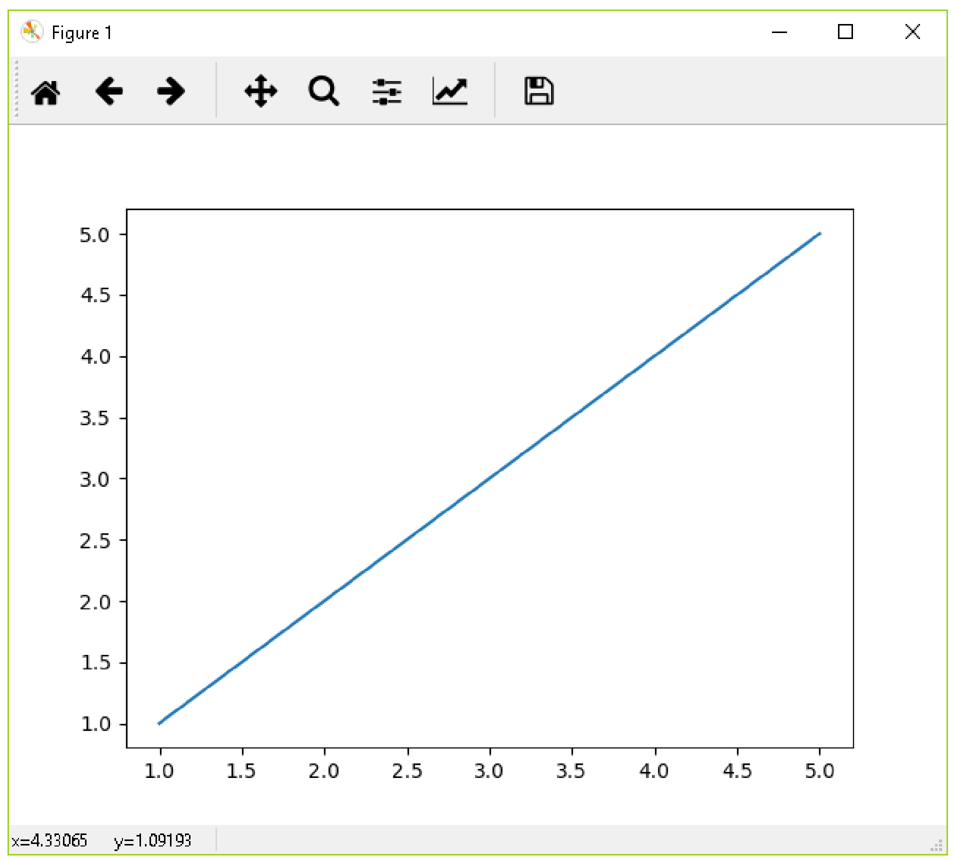


Рисунок 1.2 — Изображение графика в изолированном окне

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

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

1.3 Построение графика

Для начала построим простую линейную зависимость, зададим название графику, подпишем оси и отобразим сетку:

```
# 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 отображения сетки
plt.plot(x, y)  # построение графика
```

В результате получим график, представленный на рисунке 1.3.

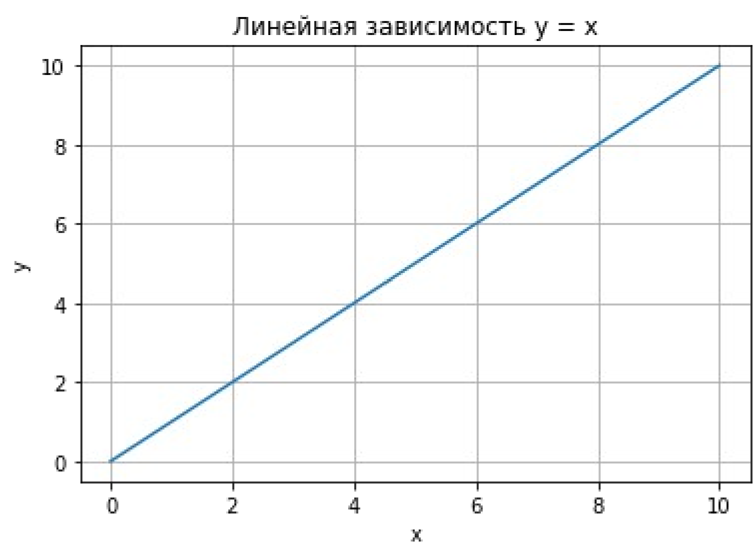


Рисунок 1.3 — Линейный график

Изменим тип линии и её 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:

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

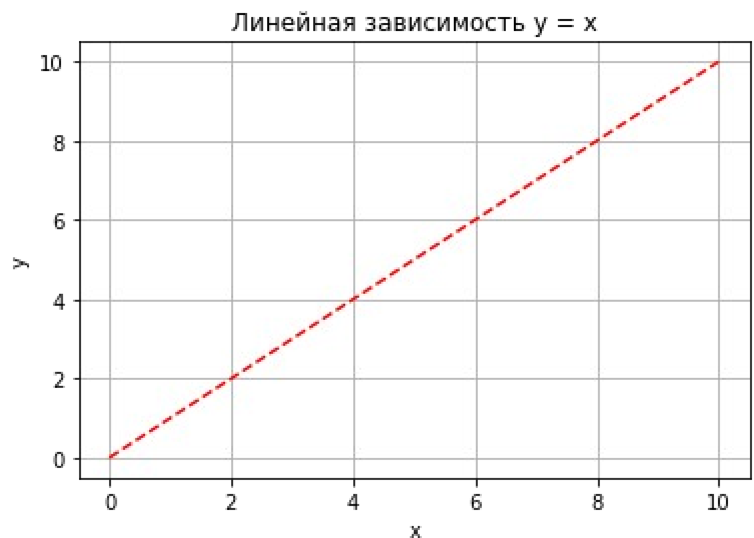


Рисунок 1.4 — Изменённый линейный график

Более подробно о том, как задавать цвет и тип линии будет рассказано в разделе "*2.3 Работа с линейным графиком*".

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: y1 = x, y2 = x^2') # заголовок
plt.xlabel('x')          # ось абсцисс
plt.ylabel('y1, y2')     # ось ординат
plt.grid()               # включение отображения сетки
plt.plot(x, y1, x, y2)  # построение графика
```

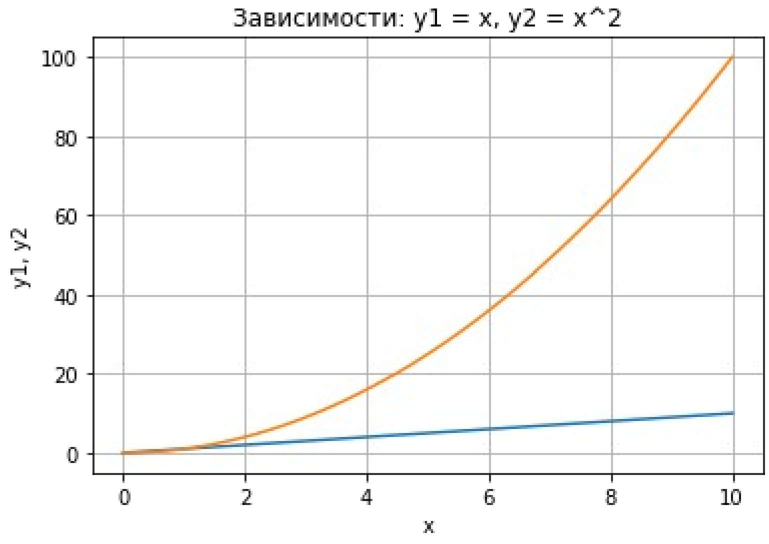


Рисунок 1.5 — Несколько графиков на одном поле

В приведённом 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 отображение двух или более различных полей, на которых может быть представлено несколько графиков.

Построим уже известные нам зависимости на разных полях:

```
x = np.linspace(0, 10, 50)
y1 = x                          # Линейная зависимость
y2 = [i**2 for i in x]          # Квадратичная зависимость
# Построение графиков
plt.figure(figsize=(9, 9))
plt.subplot(2, 1, 1)
plt.plot(x, y1)                # построение графика
plt.title('Зависимости: y1 = x, y2 = 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
```

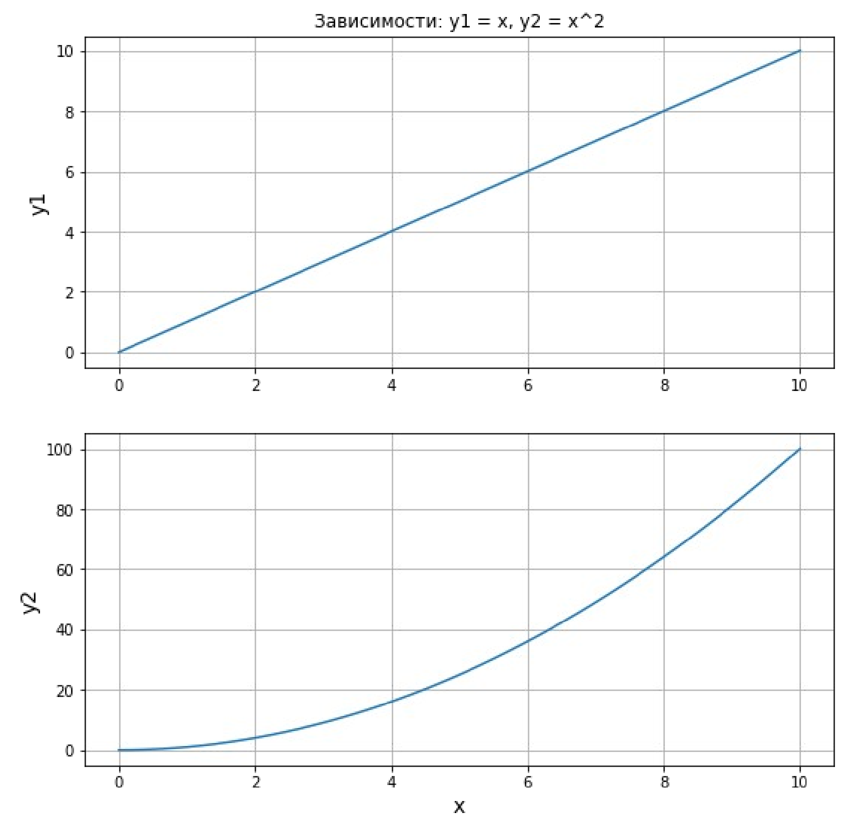


Рисунок 1.6 — Разделённые поля с графиками

Здесь мы воспользовались двумя новыми функциями:

- `figure()` — функция для задания глобальных параметров отображения графиков. В неё через параметр `figsize` передаётся кортеж из двух элементов, определяющий общий размер фигуры.
- `subplot()` — функция для задания местоположения поля с графиком. Существует несколько способов задания областей для вывода графиков. В примере мы воспользовались вариантом, который предполагает передачу трёх 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 по численным данным, то есть зависимая и независимая переменные имели числовой тип. На практике довольно часто приходится работать с данными не числовой природы — имена людей, названия 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')
```

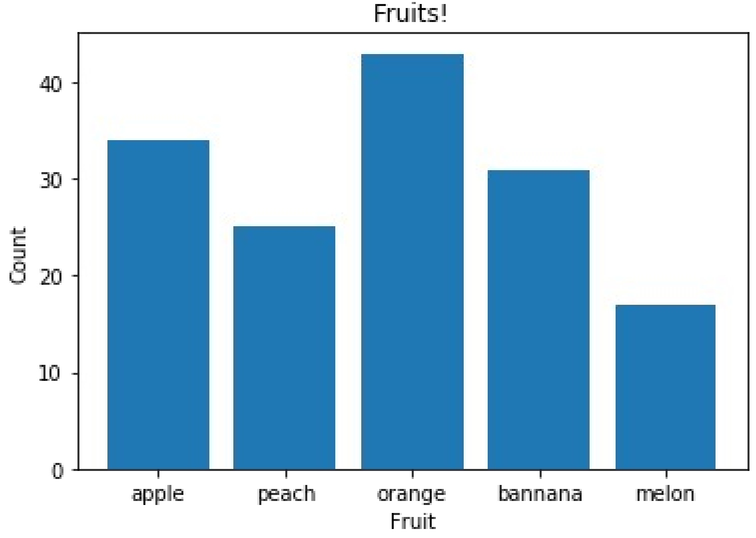


Рисунок 1.7 — Столбчатая диаграмма

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

К этому моменту, если вы самостоятельно попробовали запустить приведённые выше примеры, у вас уже должно сформироваться 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 не было трудностей при изучении глав этой книги и документации по библиотеке *Matplotlib*.

Далее мы будем использовать термин "график" для обозначения всего изображения, которое формирует *Matplotlib* (см. рисунок 1.8), и линии, построенной по заданному набору данных.

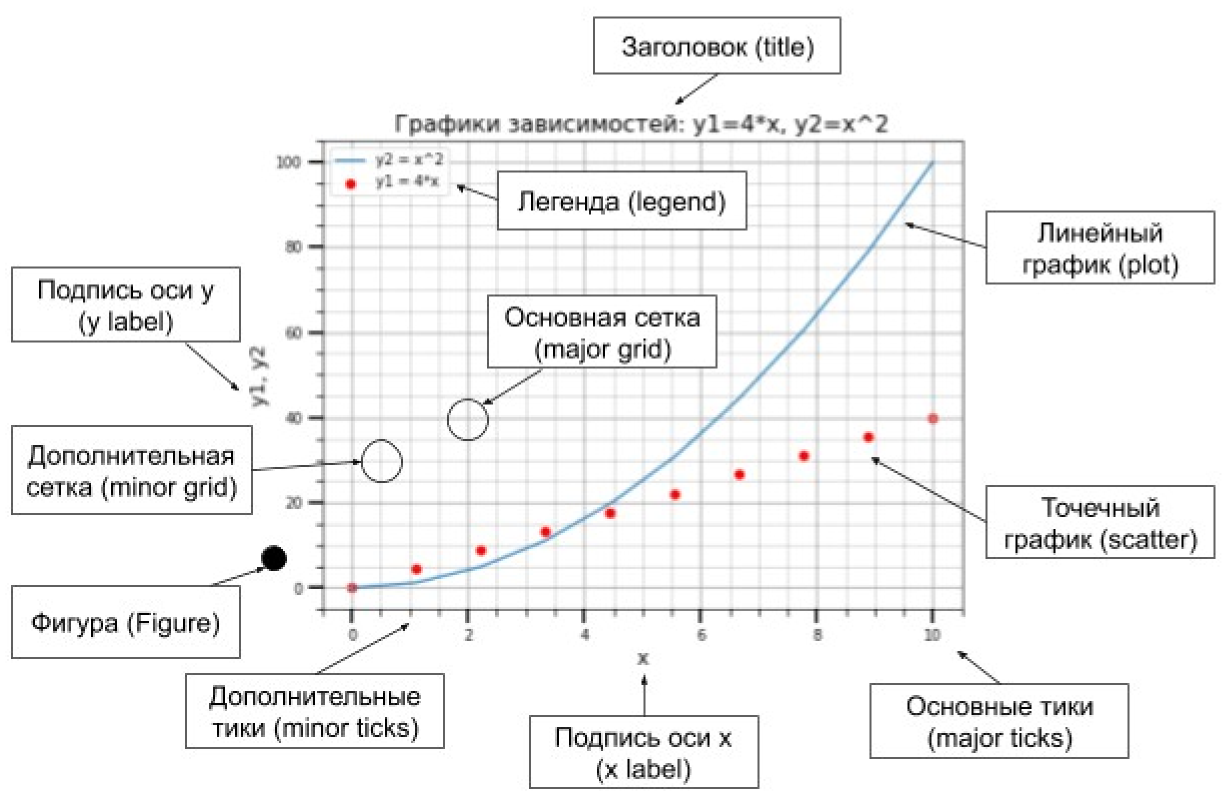


Рисунок 1.8 — Основные элементы графика

Корневым элементом, на котором *Matplotlib* строит изображение, является фигура (*Figure*). Всё, что перечислено на рисунке 1.8 — это элементы фигуры. Рассмотрим её составляющие более подробно.

График

На рисунке 1.8 представлены два графика — линейный и точечный. *Matplotlib* предоставляет огромное количество различных настроек, которые можно использовать для того, чтобы придать графику требуемый вид: цвет, толщина, тип, стиль линии и многое другое.

Оси

Вторым по важности элементом фигуры являются оси. Для каждой оси можно задать метку (подпись), основные (*major*) и дополнительные (*minor*) тики, 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 был построен график, изображённый 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()
```

Далее мы разберём подробно особенности настройки и использования всех элементов, представленных на рисунке 1.8.

Глава 2. Основы работы с 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 первой главе, вам предварительно нужно было импортировать `pyplot` из библиотеки *Matplolib*. В настоящее время среди пользователей принято импорт модуля `pyplot` производить следующим образом:

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

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

2.1 Построение графиков

Основным элементом изображения, которое строит `pyplot`, является фигура (*Figure*), на неё накладывается одно или более поле с графиками, оси координат, текстовые надписи и т.д. Для построения графика используется функция `plot()`. В самом минимальном варианте её можно использовать без параметров:

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

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

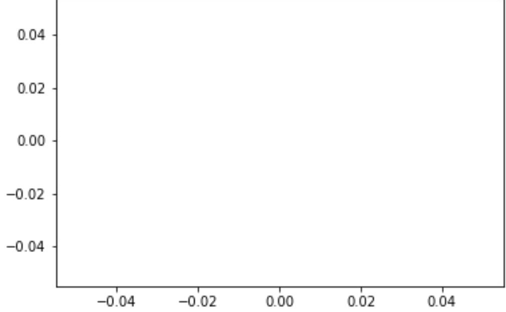


Рисунок 2.1 — Пустое поле

Далее команда импорта и *magic*-команда для *Jupyter* (первая и вторая строки, приведённой выше программы) приводиться не будут.

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])
```

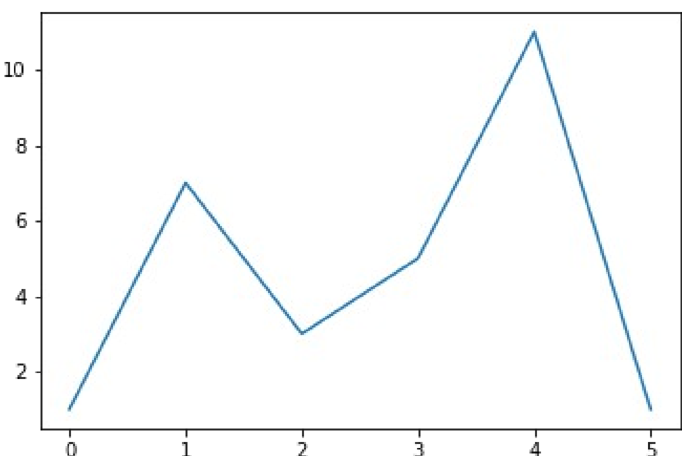


Рисунок 2.2 — Линейный график, построенный по значениям для оси *Y*

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])
```

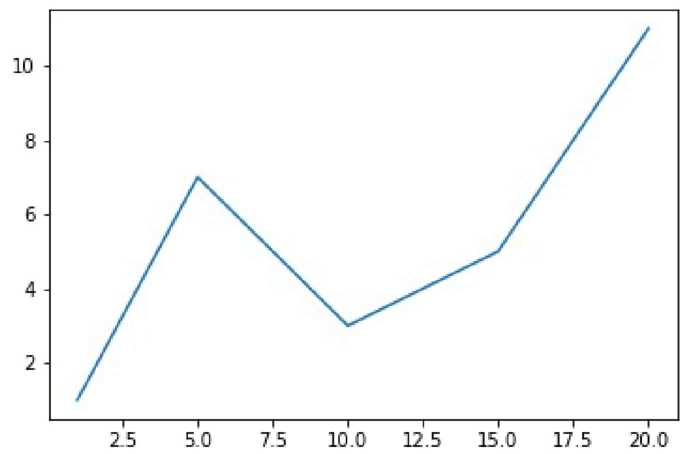


Рисунок 2.3 — Линейный график, построенный по значениям для осей *Y* и *X*

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.

Далее представлен обзор, перечисленных выше элементов. Более подробный рассказ о них будет в разделе “*3.3 Текстовые элементы графика*”.

2.2.1 Наименование осей

Для задания подписи оси *x* используется функция `xlabel()`, оси *y* — `ylabel()`. Разберёмся с аргументами данных функций. Основными параметрами функций `xlabel()` и `ylabel()` являются:

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

Функции `xlabel()` и `ylabel()` дополнительно принимают в качестве аргументов параметры конструктора класса `matplotlib.text.Text` (далее `Text`), вот некоторые из них:

- `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'}`
    - Стиль 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`: один из доступных способов задания цвета (см. раздел “*2.3.2 Цвет линии*”).
    - Цвет текста подписи.

Пример использования:

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

Нами были рассмотрены только некоторые из аргументов функций `xlabel()` и `ylabel()`, более подробная информация о них будет представлена в разделе “*3.1.2 Подписи осей графиков*”.

2.2.2 Заголовок графика

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

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

Из её параметров отметим следующие:

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

Для функции `title()` также доступны параметры конструктора класса `Text`, некоторые из них приведены в описании аргументов функций `xlabel()` и `ylabel()`.

2.2.3 Текстовое примечание

За размещение текста на поле графика отвечает функция `text()`. Первый и второй её аргументы — это координаты позиции, третий — текст надписи, пример использования:

```
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 раздела мы не будем рассматривать аргументы этой функции. Подробная информация о её настройке представлена в разделе “*3.1 Работа с легендой*”.

Разместим на уже знакомом нам графике рассмотренные выше текстовые элементы:

```
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!')
```

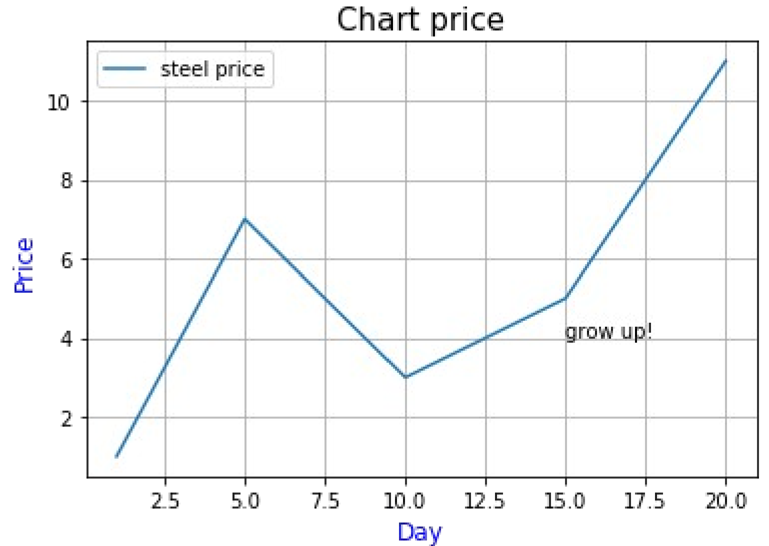


Рисунок 2.4 — Текстовые надписи на графике

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, которые можно использовать для изменения внешнего вида линейного графика.

Линейный график строится с помощью функции `plot()`, его внешний вид можно настроить непосредственно через аргументы указанной функции, например:

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

Либо можно воспользоваться функцией `setp()`:

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

2.3.1 Стиль линии графика

Стиль линии графика задаётся через параметр `linestyle`, который может принимать значения из таблицы 2.1.

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, '--')
```

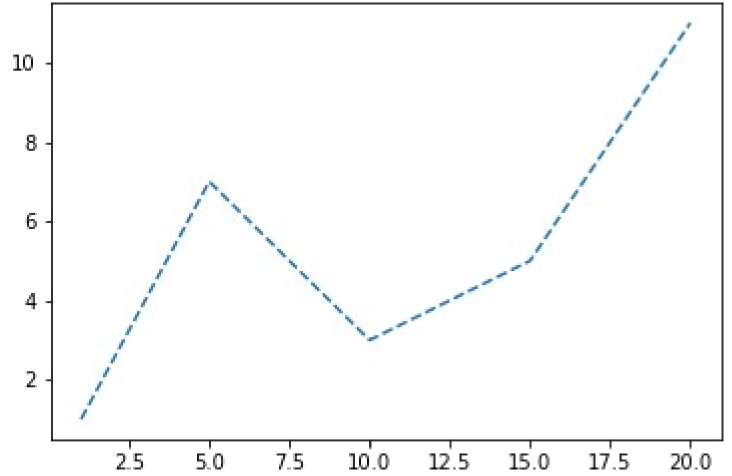


Рисунок 2.5 — Линия с примененным стилем

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 2.5.

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

Построим несколько наборов данных и выведем их, задав различные стили линиям:

```
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, ':')
```

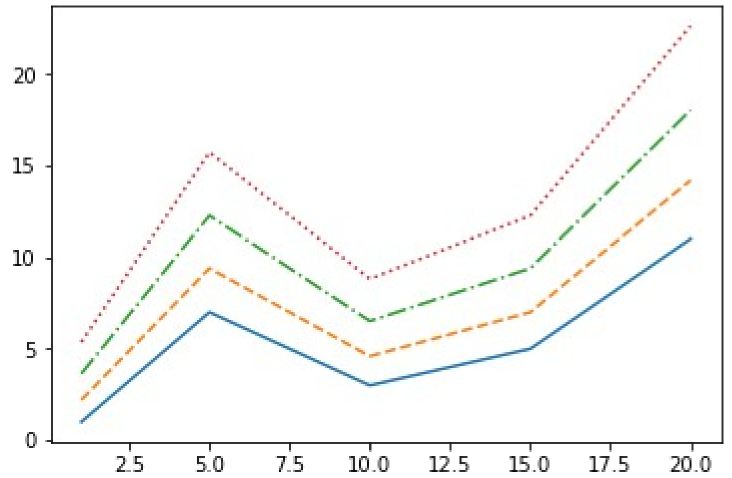


Рисунок 2.6 — Несколько графиков, построенных одной функцией `plot()`

Тот же результат можно получить, вызвав `plot()` отдельно для каждого набора данных. Если вы хотите представить графики изолированно друг от друга (каждый на своём поле), то используйте для этого функцию `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, ':')
```

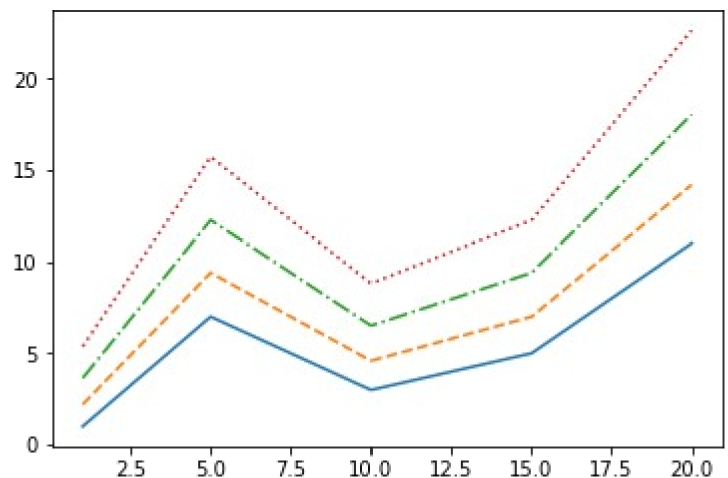


Рисунок 2.7 — Несколько графиков, построенных разными функциями `plot()`

2.3.2 Цвет линии

Цвет линии графика задаётся через параметр `color` (или `c`, если использовать сокращённый 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 задаётся с помощью символа из набора {'b', 'g', 'r', 'c', 'm', 'y', 'k', 'w'}, то он может быть совмещён со стилем линии в рамках параметра fmt функции plot(). Например: штриховая красная линия будет задаваться так: '--r', а штрихпунктирная зелёная так '-.g':

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

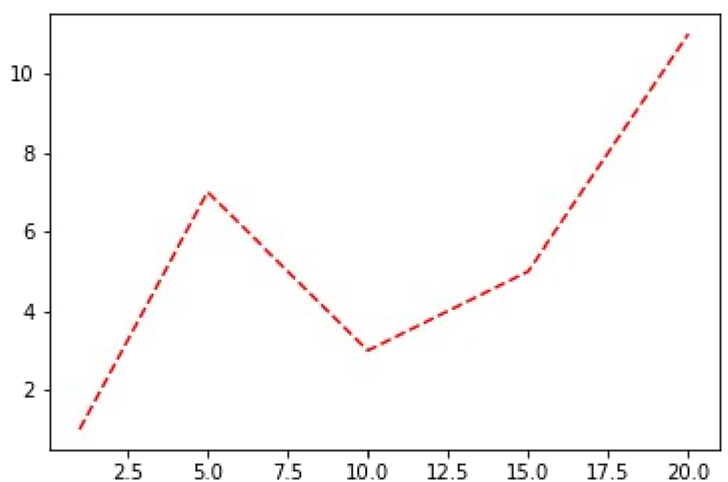


Рисунок 2.8 — График, представленный в виде штриховой красной линии

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 дополнительно можно задать маркер, обозначающий положение точек.

Приведём пару примеров:

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

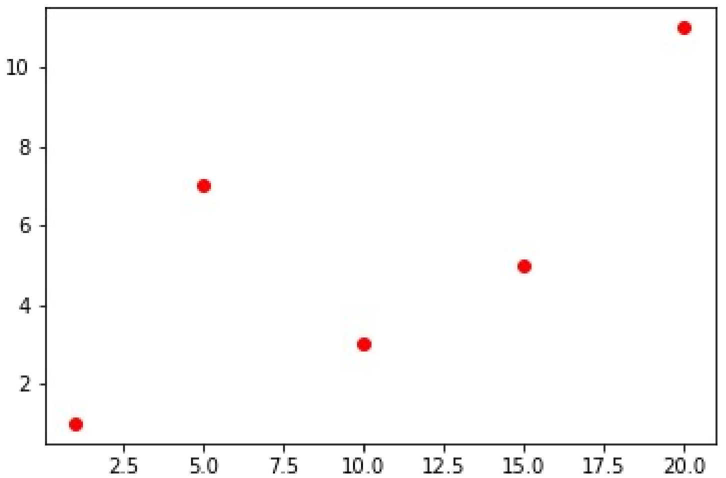


Рисунок 2.9 — График, состоящий из круглых красных точек

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

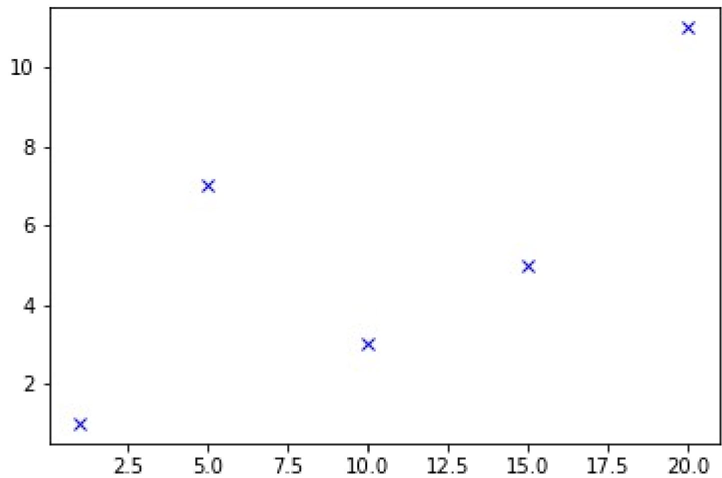


Рисунок 2.10 — График, состоящий из синих *x*-символов

Более подробную информацию по настройке внешнего вида точечного графика можете найти в разделе "*4.2.4 Точечный график (Диаграмма рассеяния)*".

2.4 Размещение графиков отдельно друг от друга

Существуют три основных подхода к размещению 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 на сетке.

В этой главе будут рассмотрены первые два подхода.

2.4.1 Работа с функцией subplot()

Самый простой способ вывести графики на отдельных полях — это использовать функцию `subplot()` для задания мест их размещения. До этого момента мы не работали с Фигурой (*Figure*) напрямую, значения ее параметров, задаваемые по умолчанию, нас устраивали. Для решения текущей задачи придётся 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. Задаётся 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, ':')
```

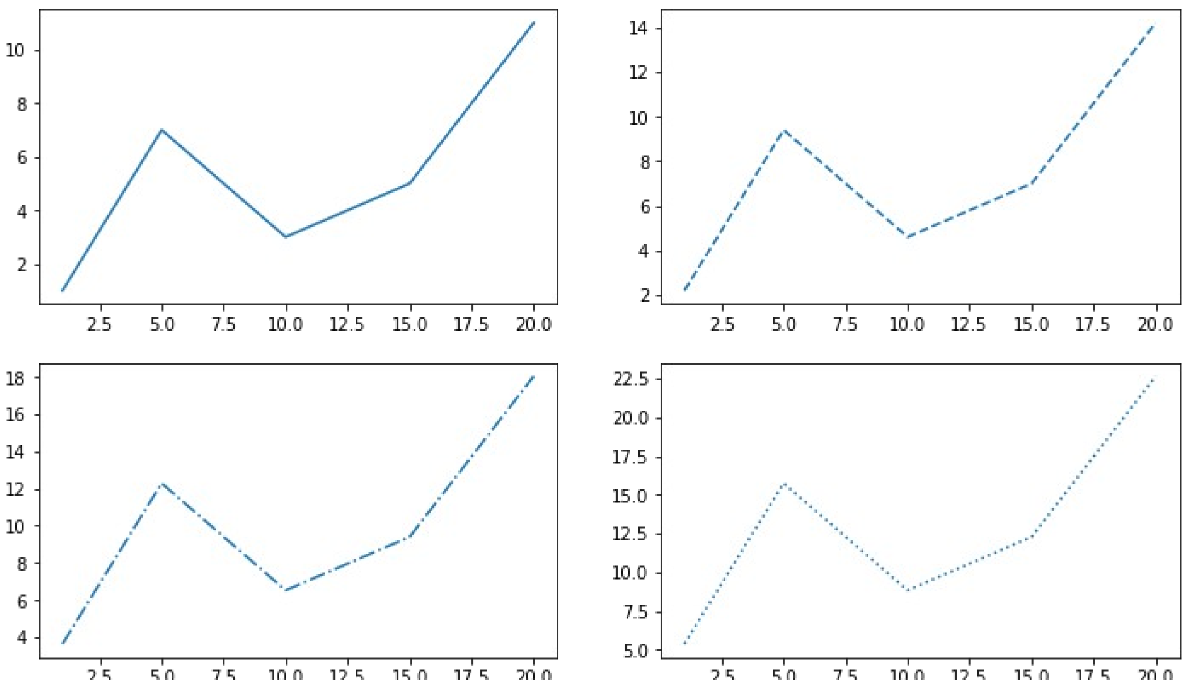


Рисунок 2.11 — Размещение графиков на отдельных полях (пример 1)

Решим эту же задачу, используя второй вариант вызова `subplot()`:

```
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, ':')
```

В результате будет получен график, аналогичный приведённому на рисунке 2.11.

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 передаётся 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 полных доступ к настройке внешнего вида отображаемых элементов.

Решим задачу вывода четырёх графиков с помощью `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, ':')
```

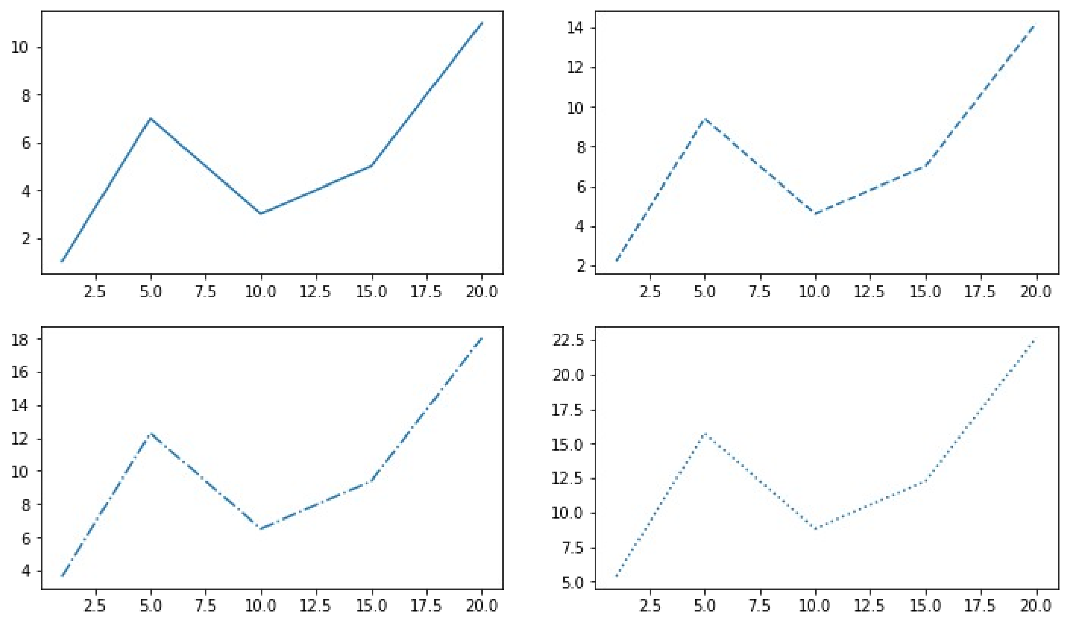


Рисунок 2.12 — Размещение графиков на отдельных полях (пример 2)

Глава 3. Настройка элементов графика

3.1 Работа с легендой

В данном параграфе будут рассмотрены следующие темы: отображение легенды, настройка её расположения на графике, настройка внешнего вида легенды.

3.1.1 Отображение легенды

Для отображения легенды на графике используется функция `legend()`. Возможны следующие варианты её 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()
```

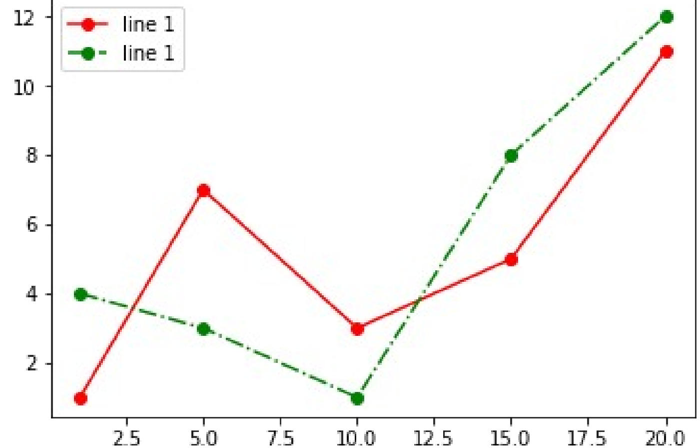


Рисунок 3.1 — Легенда на графике (пример 1)

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'])
```

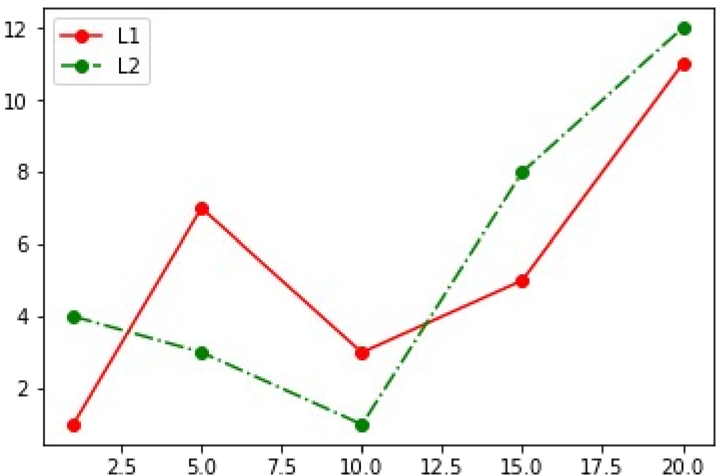


Рисунок 3.2 — Легенда на графике (пример 2)

В третьем варианте можно вручную указать соответствие линий и текстовых меток:

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

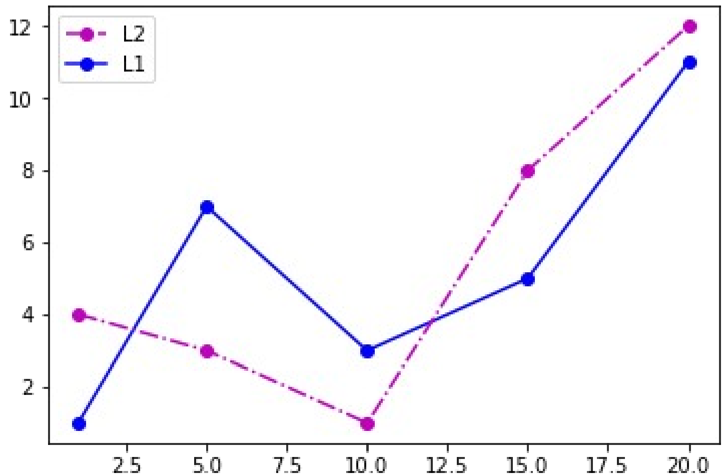


Рисунок 3.3 — Легенда на графике (пример 3)

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 расположения 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
```

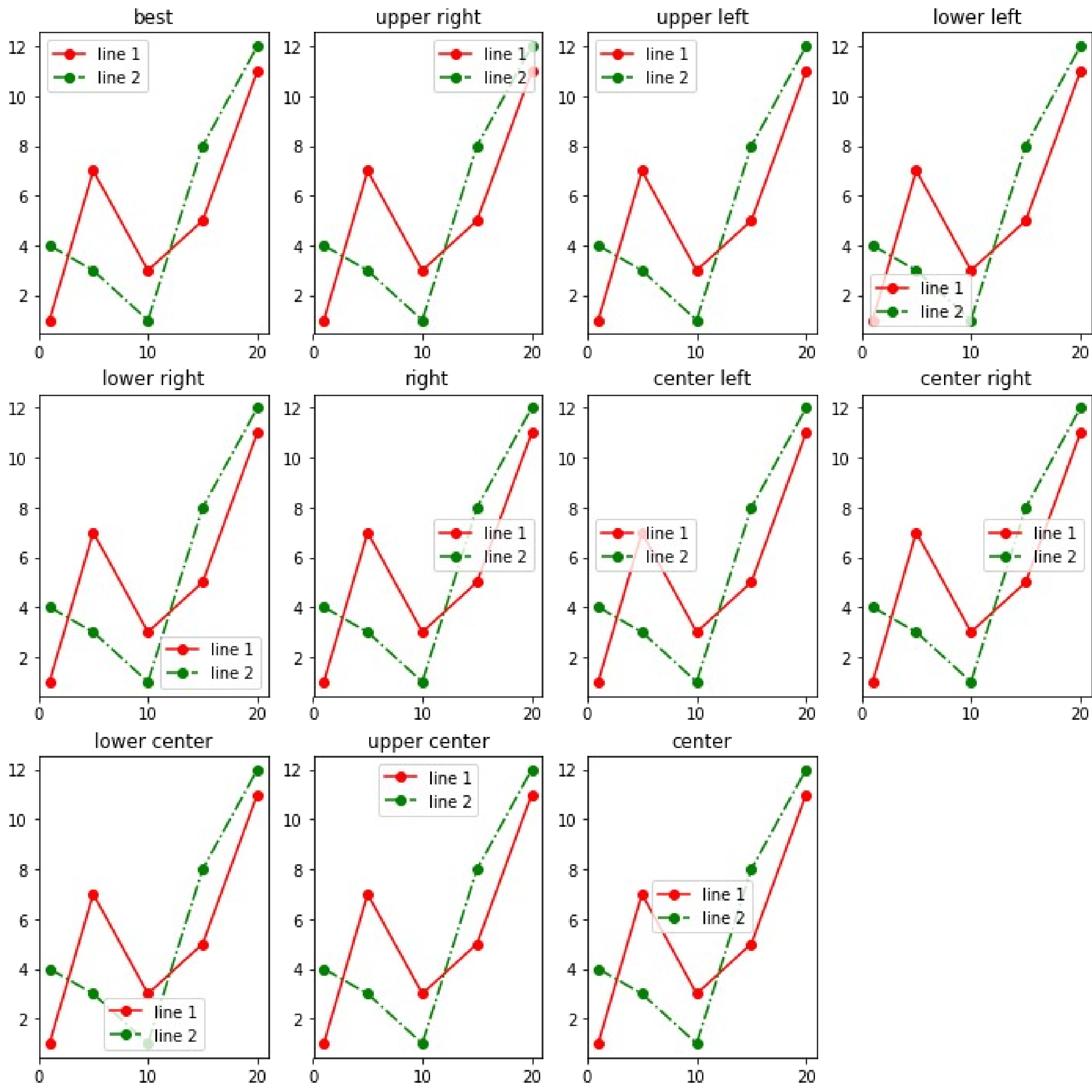


Рисунок 3.4 — Различные варианты расположения легенды на графике

Для более гибкого управления расположением легенды можно воспользоваться параметром `bbox_to_anchor` функции `legend()`.

Этому параметру присваивается кортеж, состоящий из четырёх или двух элементов:

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

где x, y — это координаты расположения легенды;

`width` — ширина;

`height` — высота.

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))
```

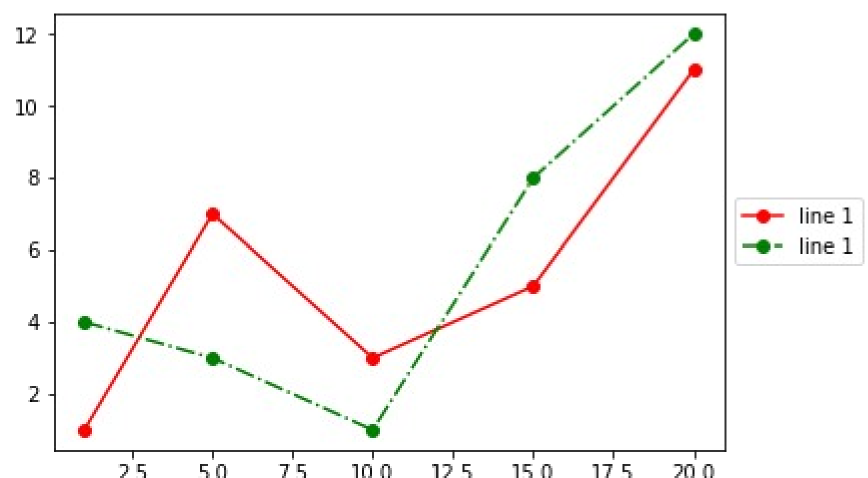


Рисунок 3.5 — Расположение легенды вне поля графика

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, float или {'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large'}	Размер шрифта надписи легенды.
frameon	bool	Отображение рамки.
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 легенды:

```
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')
```

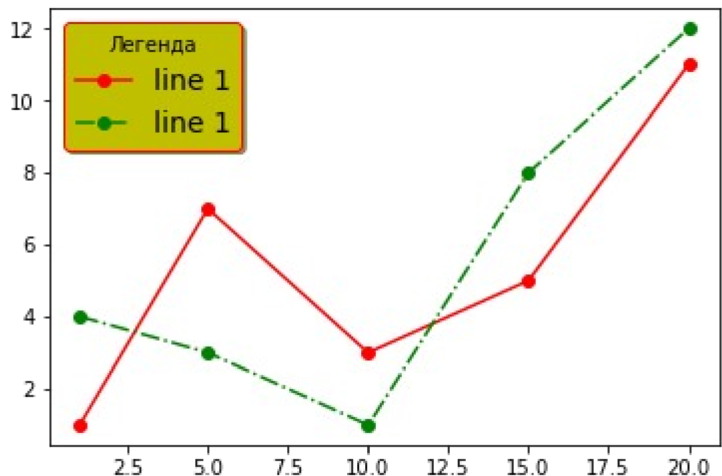


Рисунок 3.6 — Пример настройки внешнего вида легенды

3.2 Компоновка графиков

Самые простые и наиболее часто используемые варианты компоновки графиков были рассмотрены в “*Главе 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. Может показаться, что работа с GridSpec довольно неудобна и требует написания лишнего кода, но, если требуется расположить поля с графиками нетривиальным способом, 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)
```

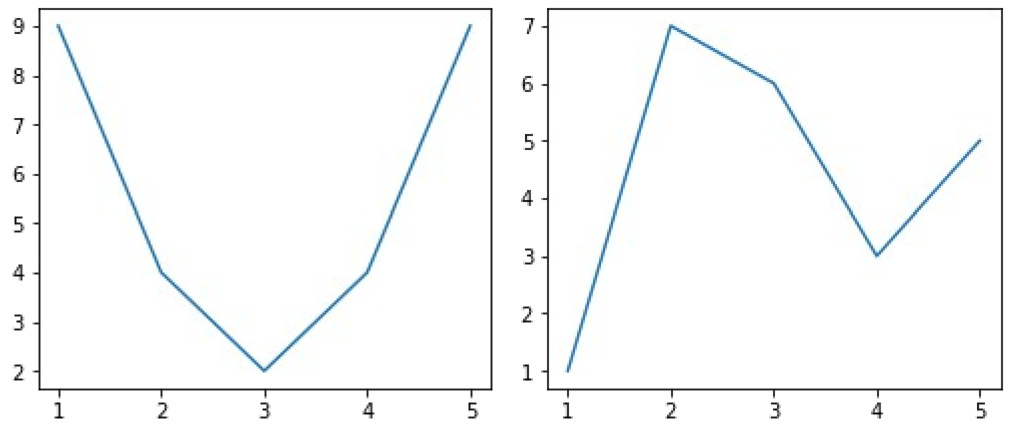


Рисунок 3.7 — Два поля с графиками

Объект класса `GridSpec` создаётся в строке:

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

В конструктор класса передаётся 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).

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

Добавим ещё 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)
```

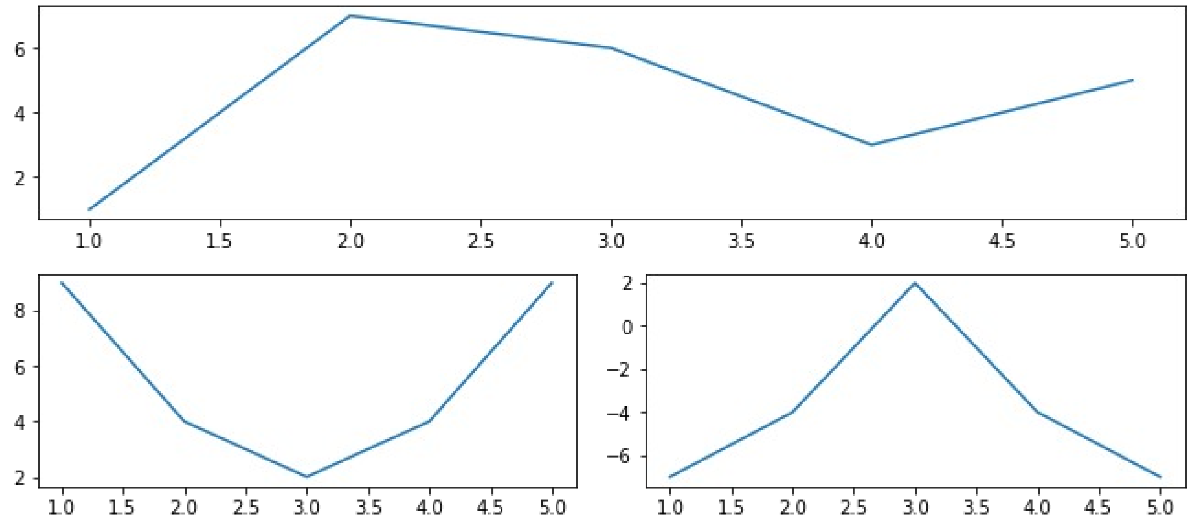


Рисунок 3.8 — Свободная компоновка (пример 1)

Ниже представлен ещё 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:]')
```

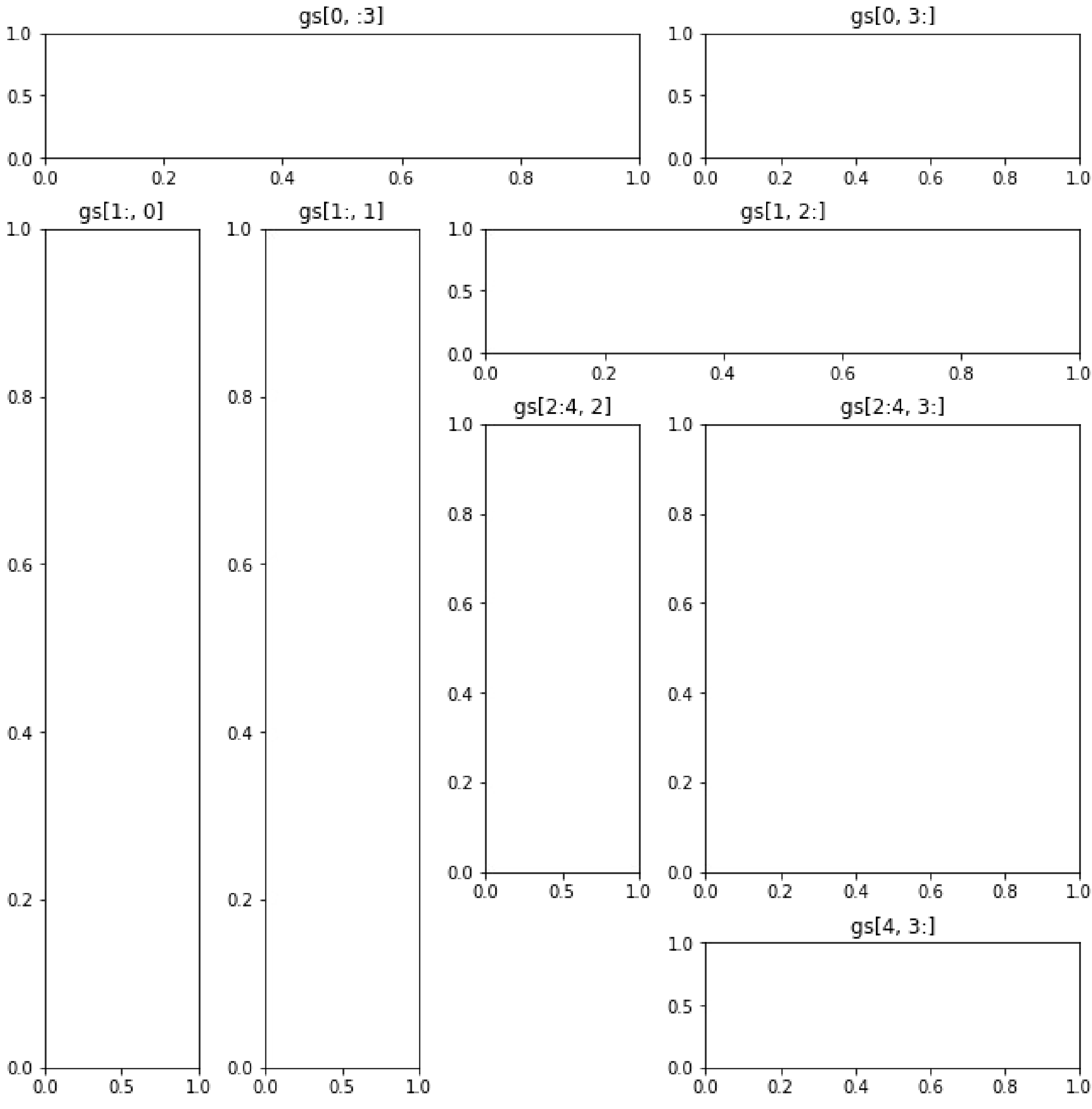


Рисунок 3.9 — Свободная компоновка (пример 2)

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

```
fg = 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')
```

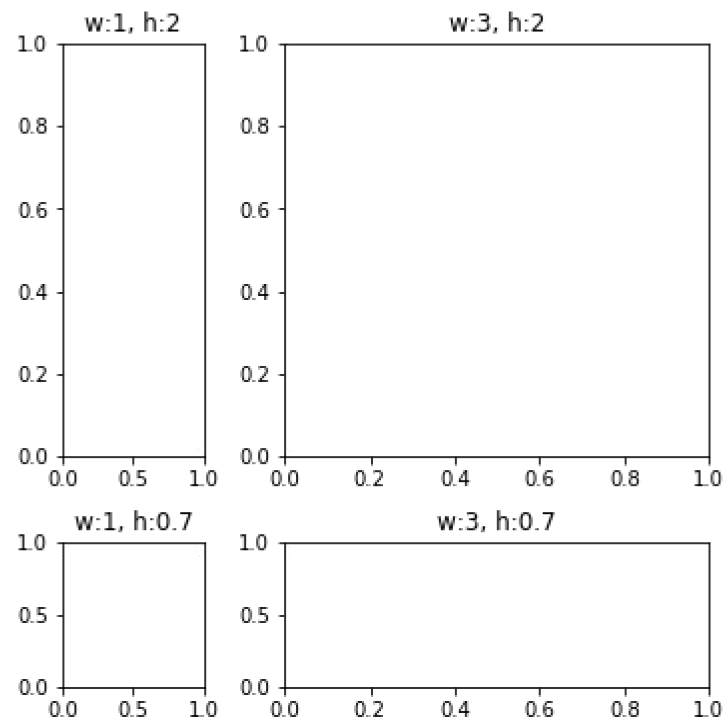


Рисунок 3.10 — Свободная компоновка (пример 3)

Информации из данного раздела и из “*Главы 2. Основы работы с модулем* pyplot” 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);
- текстовый блок на поле графика (text), либо на фигуре (figtext);
- аннотация (annotate) — текст с указателем.

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 разделе “*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')
```

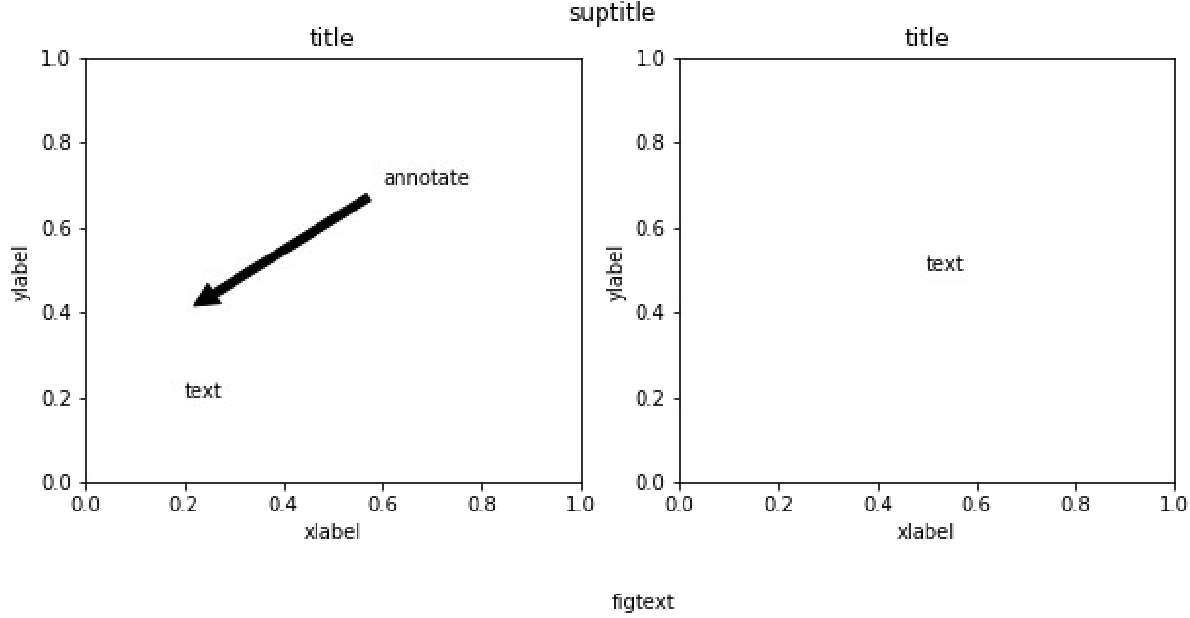


Рисунок 3.11 — Текстовые элементы графика

Некоторые из представленных текстовых элементов мы уже рассмотрели в “*Главе 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 внешним видом текста.

3.3.1 Заголовок фигуры и поля графика

Начнём с заголовка поля графика. Текст заголовка 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`
  - 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)
```

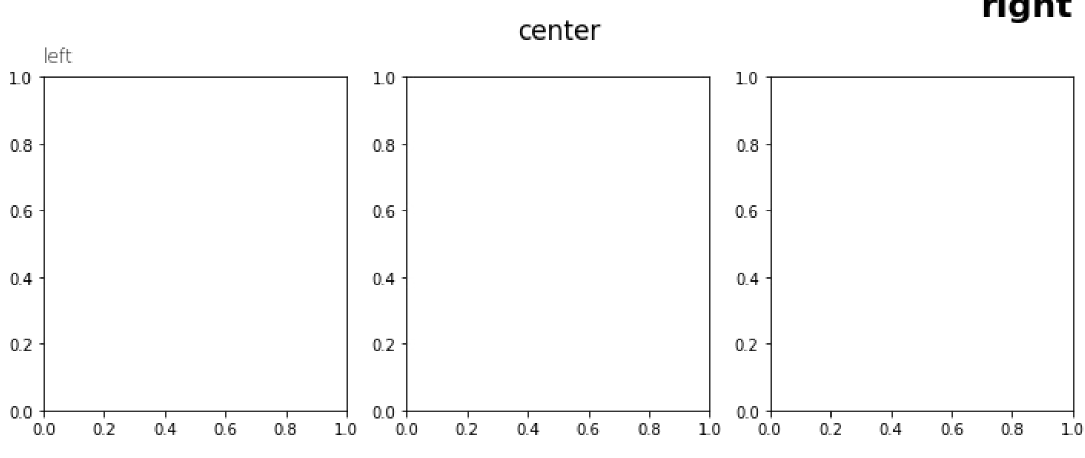


Рисунок 3.12 — Заголовок графика

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

3.3.2 Подписи осей графика

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

Основные аргументы функций почти полностью совпадают с теми, что были даны 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 текст подписи 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')
```

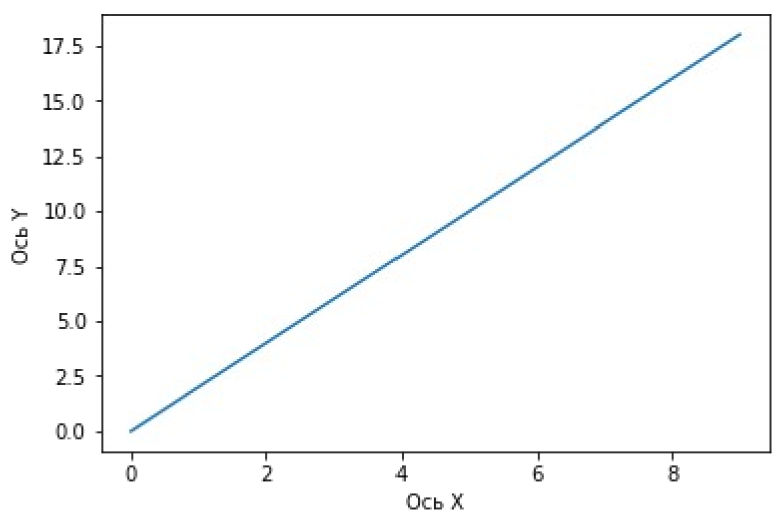


Рисунок 3.13 — Подписи осей графика (пример 1)

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')
```

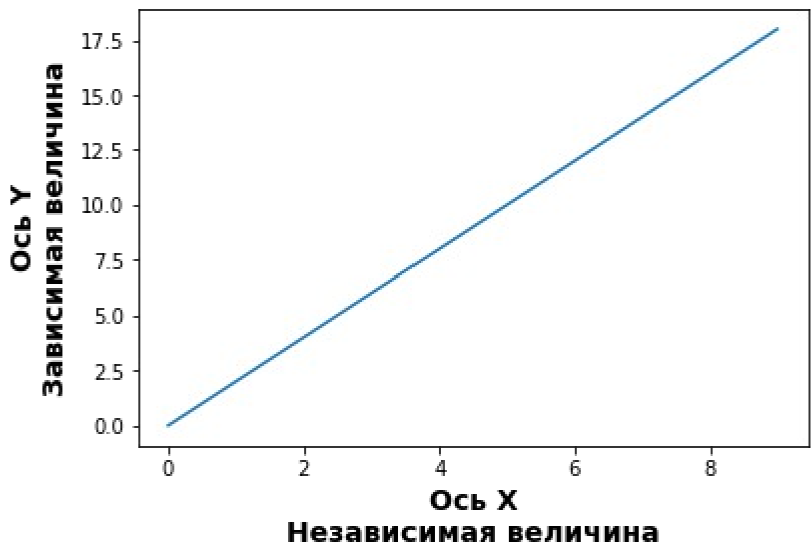


Рисунок 3.14 — Подписи осей графика (пример 2)

3.3.3 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))
```

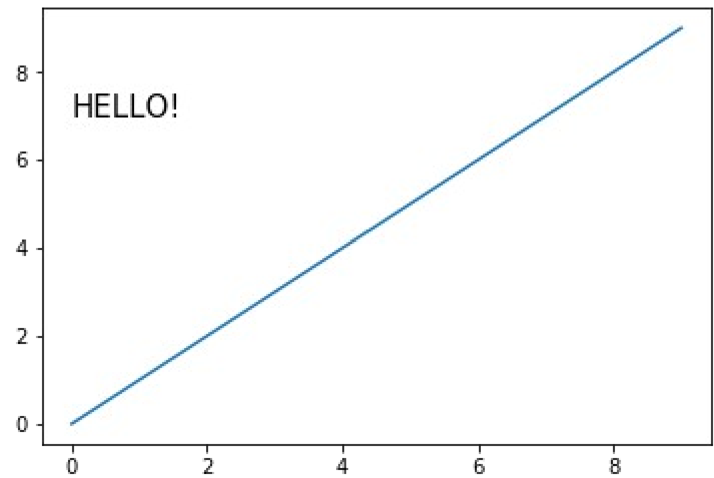


Рисунок 3.15 — Текстовый блок (пример 1)

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))
```

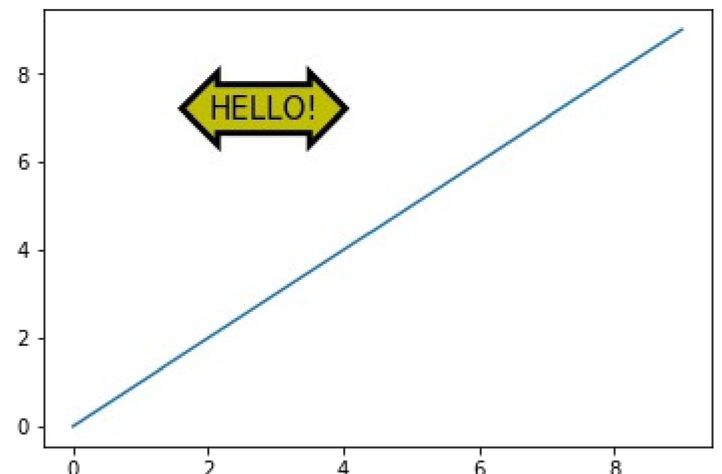


Рисунок 3.16 — Текстовый блок (пример 2)

3.3.4 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 её 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 работу с функцией `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)
```

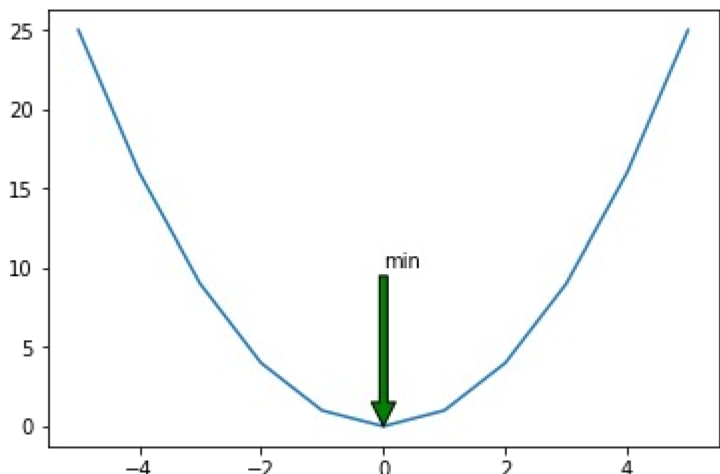


Рисунок 3.17 — Демонстрация работы с функцией `annotation()`

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: декартовая. Работа ведётся в пространстве поля графика.
`'polar'`	Тип координатной системы: полярная. Работа ведётся 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

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

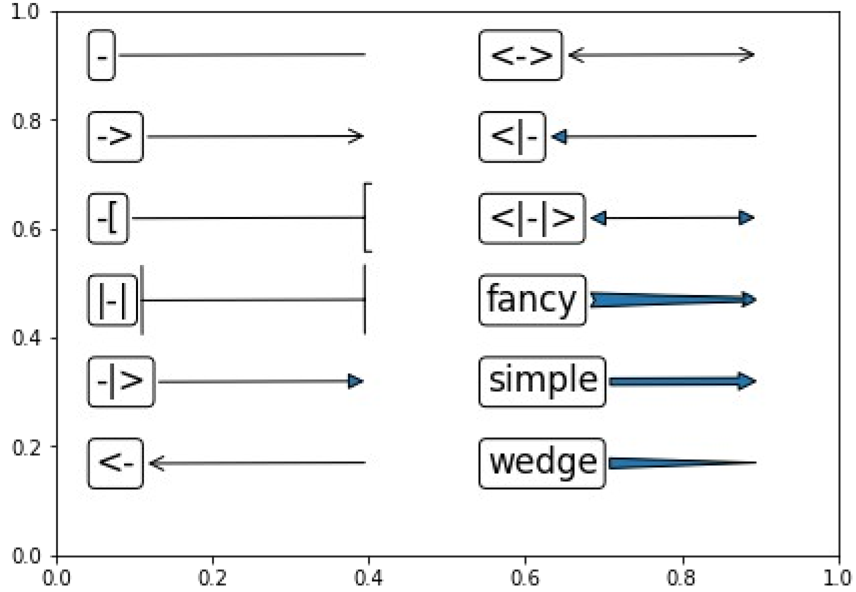


Рисунок 3.18 — Стили стрелок аннотации

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`

Данный параметр задаёт стиль линии, которая соединяет точки xy и `xycoords`. В качестве значения может принимать объект класса `ConnectionStyle` или строку, в которой указывается стиль линии соединения с параметрами, перечисленными через запятую.

Таблица 3.5 — 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, демонстрирующий 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 отображение текста

Параметры, отвечающие за отображение текста, позволяют настроить прозрачность, цвет, шрифт и т.п.:

- `alpha`: `float`
  - Уровень прозрачности надписи. Параметр задаётся числом в диапазоне от 0 до 1. 0 — полная прозрачность, 1 — полная непрозрачность.
- `color`: один из доступных способов задания цвета (см. раздел "*2.3.2 Цвет линии*")
  - 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`
  - Стиль шрифта, задаётся из набора: {`'normal'`, `'italic'`, `'oblique'`}.
- `fontvariant` (или `variant`): `str`
  - Начертание шрифта, задаётся из набора: {`'normal'`, `'small-caps'`}.
- `fontweight` (или `weight`): `str`
  - Насыщенность шрифта, задаётся 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))
```

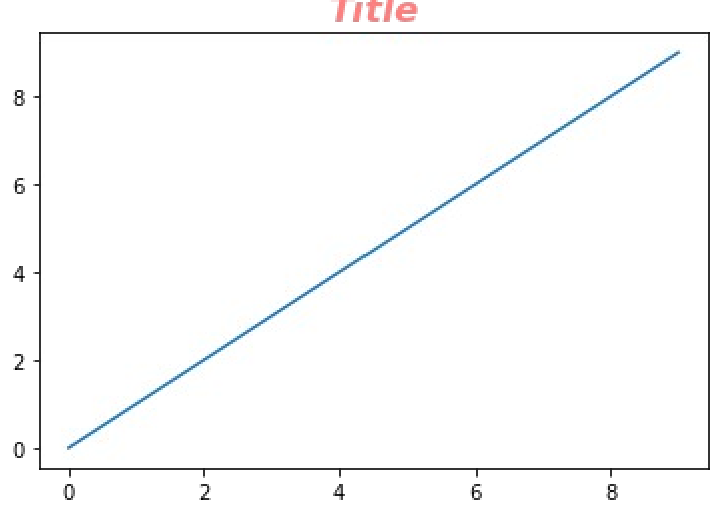


Рисунок 3.20 — Пример использования свойств класса Text

Для группового задания свойств можно использовать параметр `fontproperties` или `font_properties`, которому в качестве значения передаётся 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)
```

Параметры конструктора:

- `family`: `str`
    - Имя шрифта.
- `style` `str`
    - Стиль шрифта.
- `variant` `str`
    - Начертание.
- `stretch` `str`
    - Ширина шрифта.
- `weight` `str`
    - Насыщенность шрифта.
- `size` `str`
    - Размер шрифта.

Перед тем как использовать `FontProperties` не забудьте его импортировать:

```
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))
```

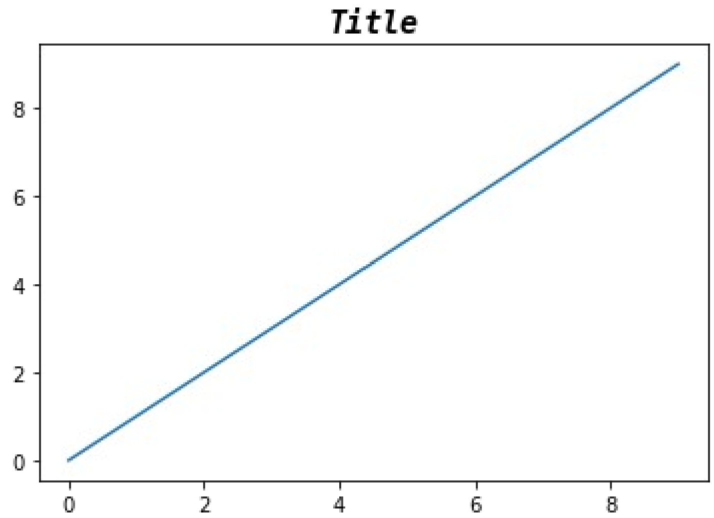


Рисунок 3.21 — Пример работы с параметром `fontproperties`

3.4.2 Параметры, отвечающие за расположение надписи

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

- `horizontalalignment` (или `ha`): `str`
  - Горизонтальное выравнивание. Задаётся из набора: `{'center', 'right', 'left'}`.
- `verticalalignment` (или `va`): `str`
  - Вертикальное выравнивание. Задаётся 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 очерёдность 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 очерёдность 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))
```

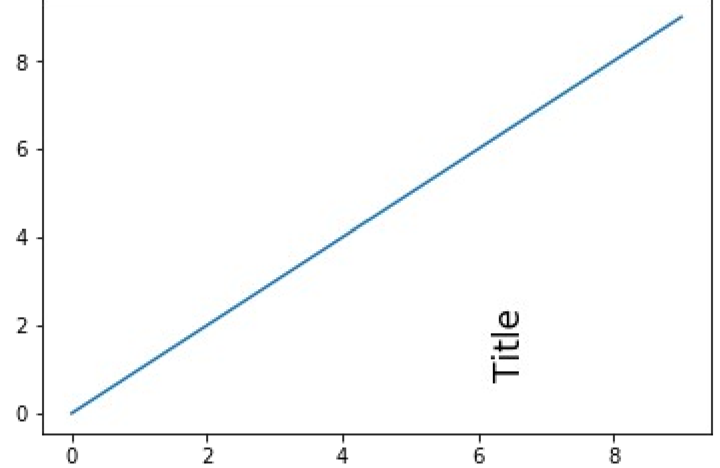


Рисунок 3.22 — Пример использования параметров, задающих расположение

3.4.3 Параметры, отвечающие за настройку заднего фона надписи

За настройку заднего фона надписи отвечает параметр:

- `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 3.6).

Таблица 3.6 — Свойства класса `patches.FancyBboxPatch`

Свойство	Тип значения	Описание
`boxstyle`	`str` или `matplotlib.patches.BoxStyle`	Стиль рамки. См. Таблицу 3.7.
`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`	`True` — использовать заливку, `False` — 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 3.7 — Параметры boxstyle

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

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))
```

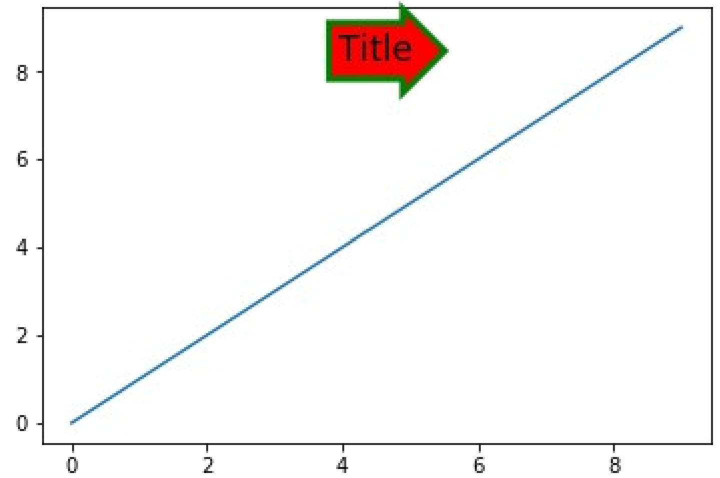


Рисунок 3.23— Пример настройки заднего фона надписи

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 понадобится аналог легенды, который в *Matplotlib* называется *colorbar*. Создадим случайное распределение с помощью `np.random.rand()` и отобразим его через `pcolor()`:

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

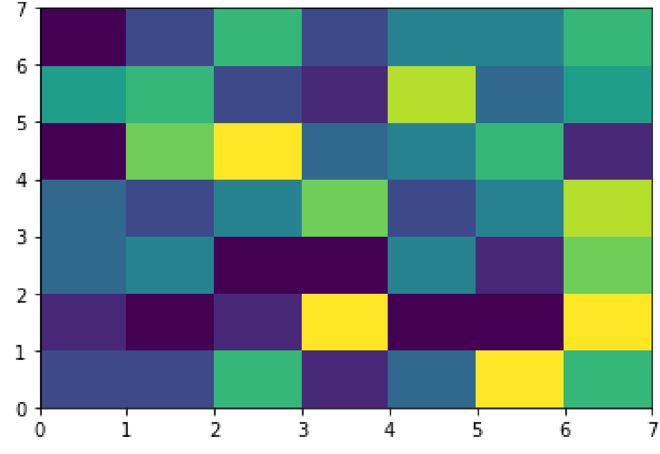


Рисунок 3.24 — Цветовое распределение

Для данного набора построим *colorbar*:

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

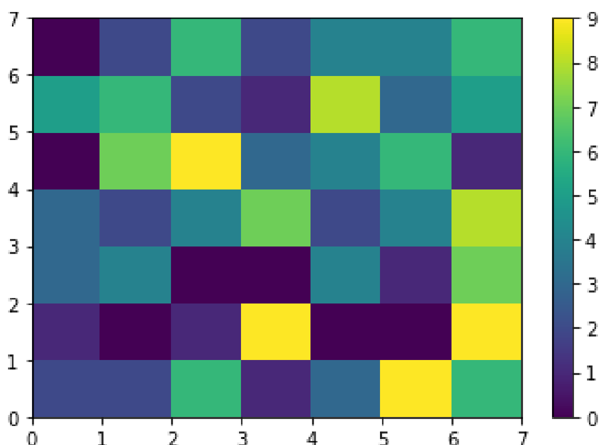


Рисунок 3.25 — Цветовая полоса для заданного цветового распределения

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

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

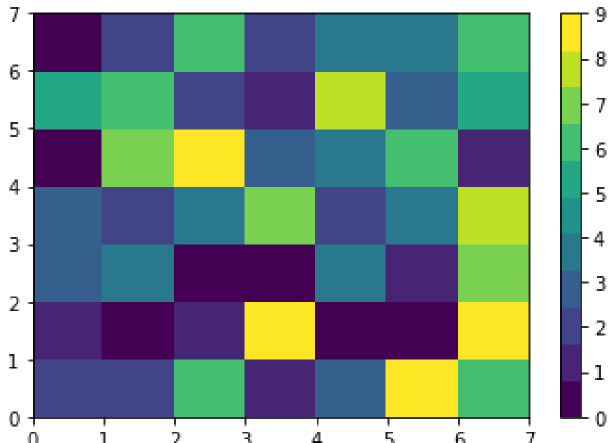


Рисунок 3.26 — Цветовая полоса с дискретным разделением цветов

3.5.1 Общая настройка с использованием `inset_locator()`

Один 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
`parent_axes`	`Axes`	Родительский `Axes` объект.
`width`	`float` или `str`	Ширина объекта. Задаётся в процентах от родительского объекта либо абсолютным значением в виде числа.
`height`	`float` или `str`	Высота объекта. Задаётся в процентах от родительского объекта либо абсолютным значением в виде числа.
`loc`	`int` или `string`, `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. Задаётся 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.

Продемонстрируем 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)
```

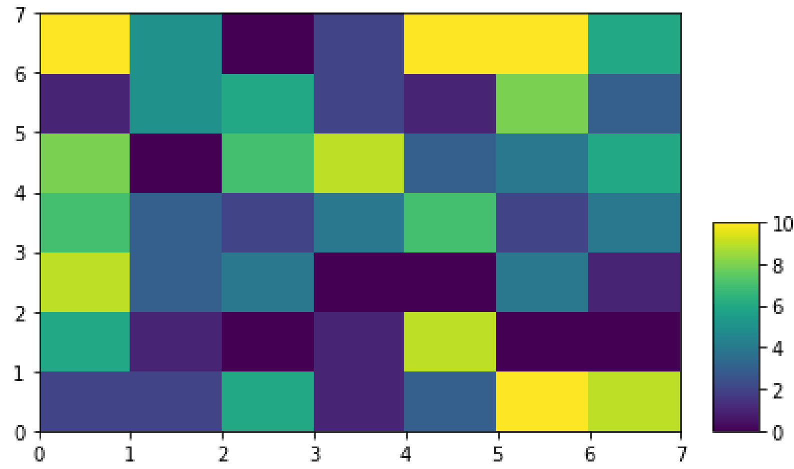


Рисунок 3.27 — Цветовая полоса, построенная с использованием inset_axes()

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 список со значениями элементов шкалы в функцию `colorbar()` через параметр `ticks`. Надпись на шкале устанавливается с помощью параметра `label` функции `colorbar()`.

Модифицируем последнюю строку из предыдущего примера следующим образом:

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

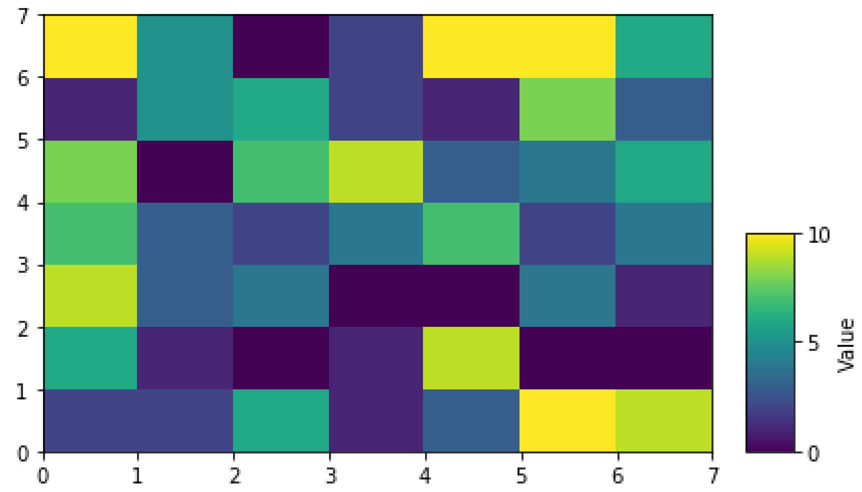


Рисунок 3.28 — Цветовая полоса с собственной шкалой

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'])
```

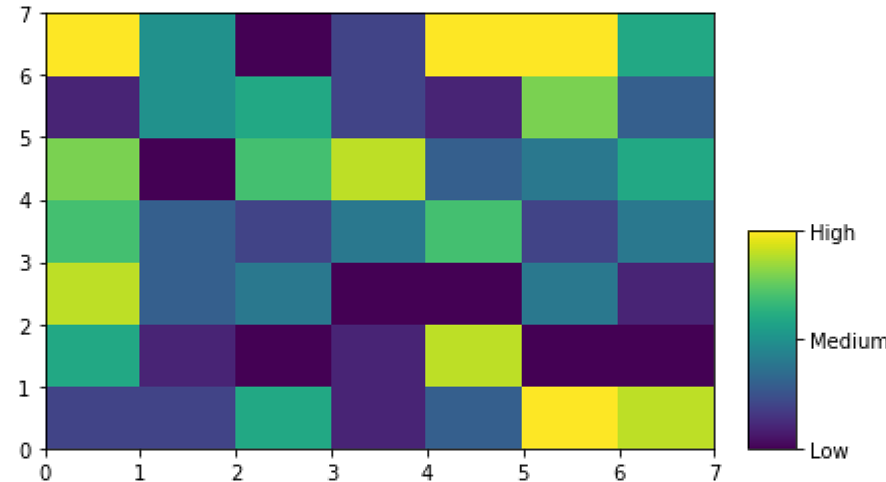


Рисунок 3.29 — Цветовая полоса с текстовыми надписями

3.5.3 Дополнительные параметры настройки цветовой полосы

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

Таблица 3.8 — Параметры функции `colorbar()`

Свойство	Тип	Описание
`orientation`	`vertical` или `horizontal`	Ориентация.
`shrink`	`float`	Масштабирование цветовой полосы.
`extend`	`[ 'neither' | 'both' | 'min' | 'max' ]`	Положение указателя продления шкалы.

extendfrac	[None | 'auto' | length | lengths]	Размер указателя продления шкалы.
drawedges	bool	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)
```

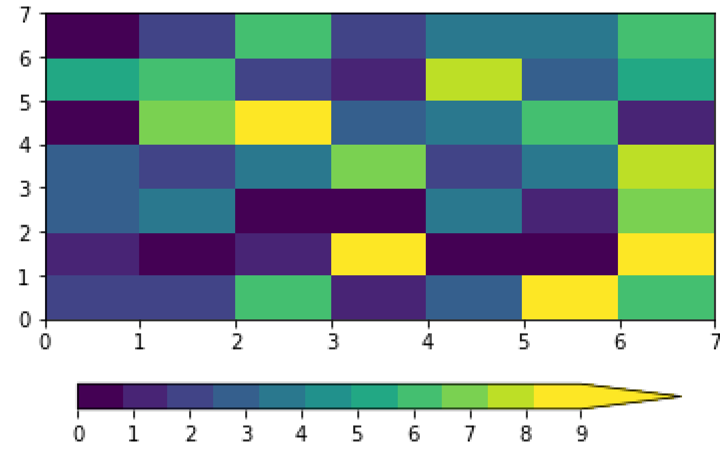


Рисунок 3.30 — Цветовая полоса с дополнительными параметрами

Глава 4. Визуализация данных

4.1 Линейный график

Линейный график — это один из наиболее часто используемых видов графиков для визуализации данных. Он 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 таким образом: `plot(y)`, то мы получим график, у которого по оси ординат (ось *y*) будут отложены значения из переданного списка, по оси абсцисс (ось *x*) — индексы элементов массива.

Рассмотрим аргументы функции `plot()`:

- `x, x2, ...`: массив
  - Наборы данных для оси абсцисс (ось *x*) для первого, второго и т.д. графика.
- `y, y2, ...`: массив
  - Наборы данных для оси ординат (ось *y*) для первого, второго и т.д. графика.

- `fmt`: `str`
  - Формат графика. Задаётся 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
`alpha`	`float`	Прозрачность.
`color` или `c`	`color`	Цвет.
`fillstyle`	`{'full', 'left', 'right', 'bottom', 'top', 'none'}`	Стиль заливки.
`label`	`object`	Текстовая метка.
`linestyle` или `ls`	`{'-', '--', '-.', ':', '', (offset, on-off-seq), ...}`	Стиль линии.
`linewidth` или `lw`	`float`	Толщина линии.
`marker`	`matplotlib.markers`	Стиль маркера.
`markeredgecolor` или `mec`	`color`	Цвет границы маркера.
`markeredgewidth` или `mew`	`float`	Толщина границы маркера.
`markerfacecolor` или `mfc`	`color`	Цвет заливки маркера.
`markersize` или `ms`	`float`	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 (*point marker*).
`','`	Пиксель (*pixel marker*).
`'o'`	Окружность (*circle marker*).
`'v'`	Треугольник, направленный вниз (*triangle_down marker*).
`'^'`	Треугольник, направленный вверх (*triangle_up marker*).
`'<'`	Треугольник, направленный влево (*triangle_left marker*).
`'>'`	Треугольник, направленный вправо (*triangle_right marker*).
`'1'`	Треугольник, направленный вниз (*tri_down marker*).
`'2'`	Треугольник, направленный вверх (*tri_up marker*).
`'3'`	Треугольник, направленный влево (*tri_left marker*).
`'4'`	Треугольник, направленный вправо (*tri_right marker*).
`'s'`	Квадрат (*square marker*).
`'p'`	Пятиугольник (*pentagon marker*).
`'*'`	Звезда (*star marker*).
`'h'`	Шестиугольник (*hexagon1 marker*).
`'H'`	Шестиугольник (*hexagon2 marker*)
`'+'`	Плюс (*plus marker*).
`'x'`	*X*-образный маркер (*x marker*).
`'D'`	Ромб (*diamond marker*).

<table>
<tr><td><code>'d'</code></td><td>Ромб (thin_diamond marker).</td></tr>
<tr><td><code>'|'</code></td><td>Вертикальная линия (vline marker).</td></tr>
<tr><td><code>'_'</code></td><td>Горизонтальная линия (hline marker).</td></tr>
</table>

- `line`: `str`
    - Стиль линии.

Таблица 4.3 — Стиль линии

Символ	Описание
`'-'`	Сплошная линия (*solid line style*).
`'--'`	Штриховая линия (*dashed line style*).
`'-.'`	Штрихпунктирная линия (*dash-dot line style*).
`':'`	Штриховая линия (*dotted line style*).

- `color`
    - Цвет графика.

Таблица 4.4 — Цвет графика

Символ	Описание
`'b'`	Синий.
`'g'`	Зелёный.
`'r'`	Красный.
`'c'`	Бирюзовый.
`'m'`	Фиолетовый (пурпурный).
`'y'`	Жёлтый.
`'k'`	Чёрный.
`'w'`	Белый.

Продемонстрируем 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)
```

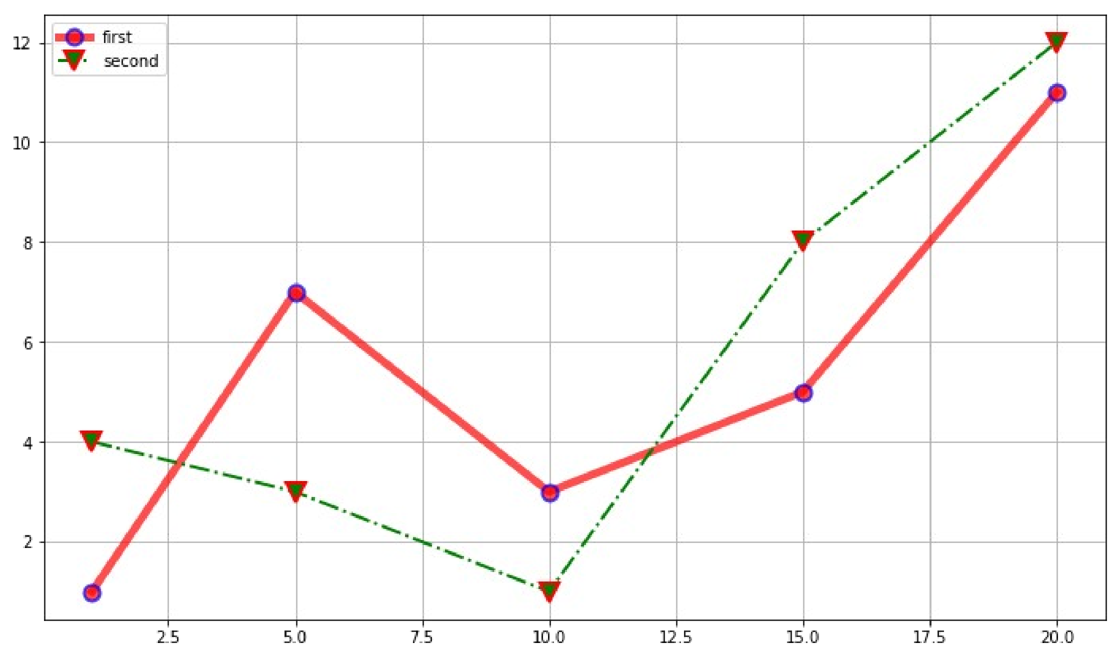


Рисунок 4.1 — Графики, построенные с помощью `plot()`

Рассмотрим различные варианты работы с линейным графиком.

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 (ось *x*).
- y1 : массив длины *N* или скалярное значение
  - Набор данных для оси ординат (ось *y*) — первая кривая.
- y2 : массив длины *N* или скалярное значение
  - Набор данных для оси ординат (ось *y*) — вторая кривая.
- `where`: массив `bool` элементов (длины *N*), `optional`, значение по умолчанию: None
  - Задает 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 разделов).
- `**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)
```

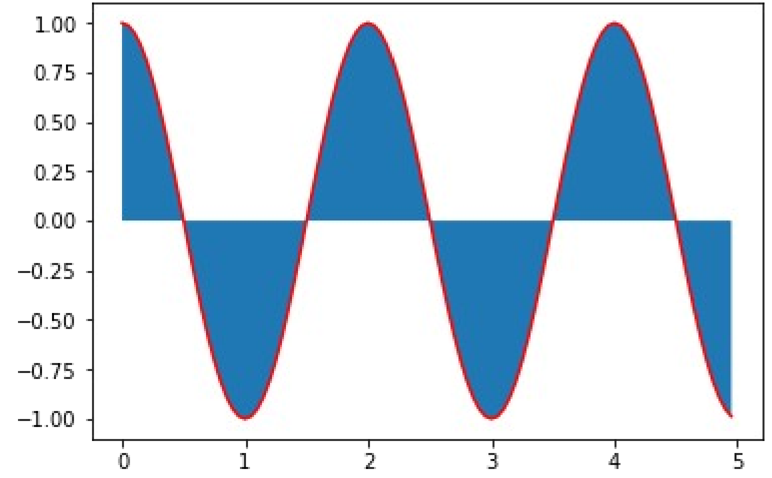


Рисунок 4.2 — График с заливкой (пример 1)

Изменим правила заливки:

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

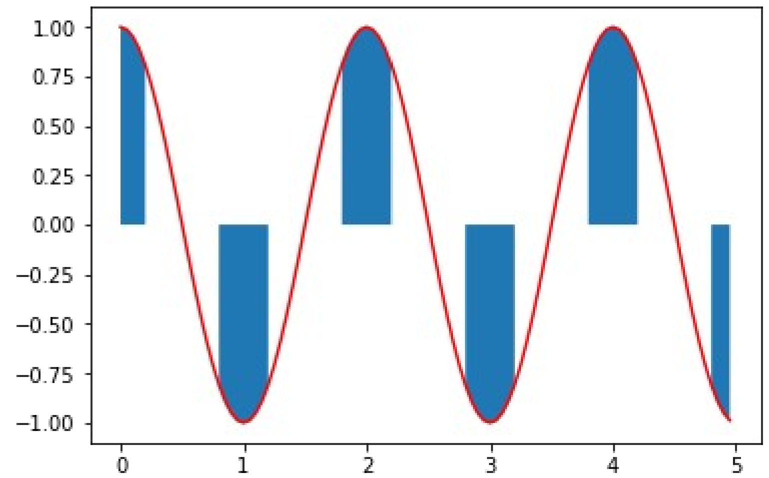


Рисунок 4.3 — График с заливкой (пример 2)

Используя параметры y1 и y2, можно формировать более сложные решения. Заливка области между 0 и *y* при условии, что *y* >= 0:

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

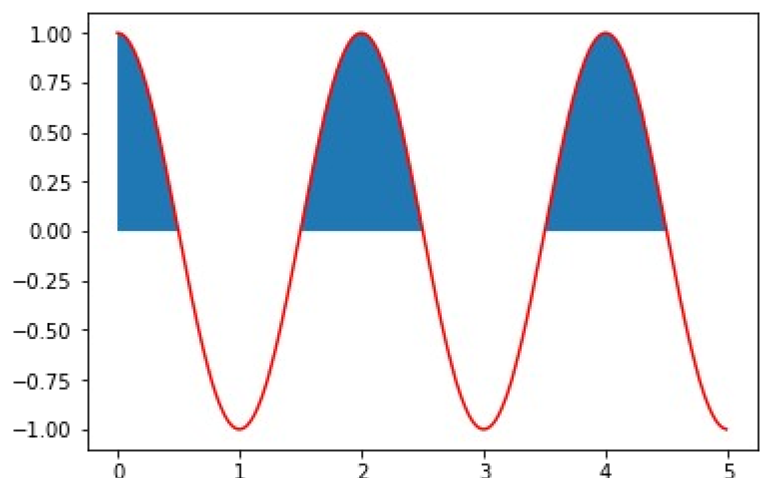


Рисунок 4.4 — График с заливкой (пример 3)

Заливка области между 0.5 и *y* при условии, что *y* >= 0.5:

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

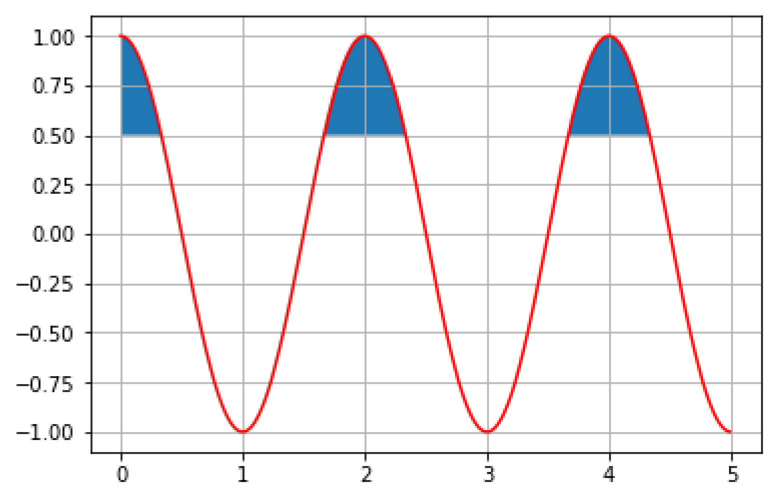


Рисунок 4.5 — График с заливкой (пример 4)

Заливка области между *y* и 1:

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

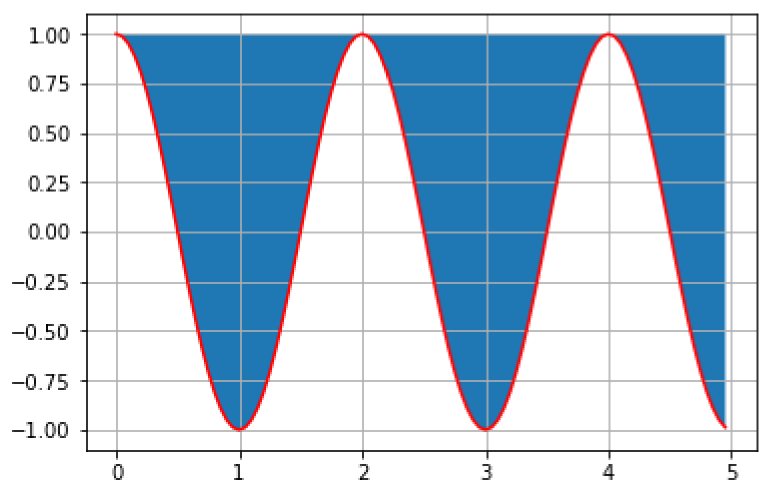


Рисунок 4.6 — График с заливкой (пример 5)

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)
```

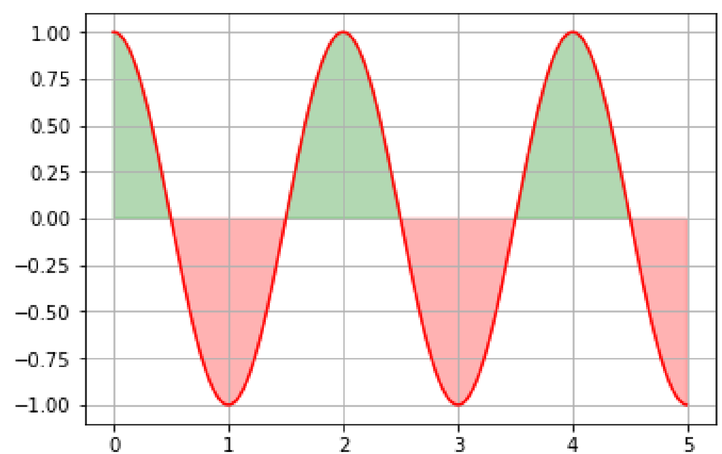


Рисунок 4.7 — График с заливкой (пример 6)

4.1.3 Настройка маркировки графиков

В начале этого раздела мы приводили пример работы с маркерами при отображении графиков. Сделаем это ещё раз, но уже в упрощённом 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')
```

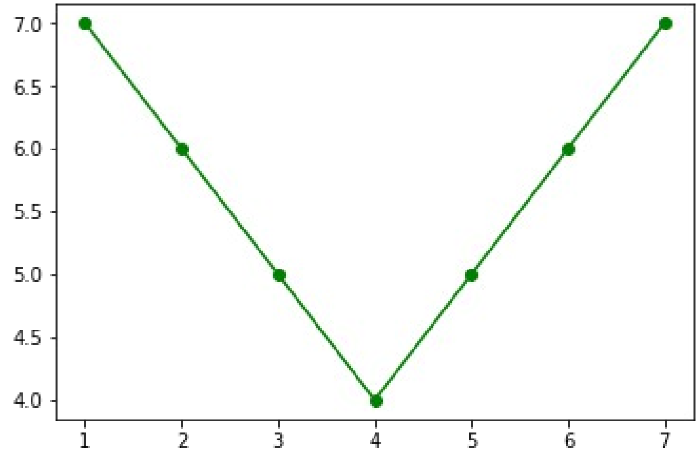


Рисунок 4.8 — График с маркировкой

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: произойдёт наложение точек:

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

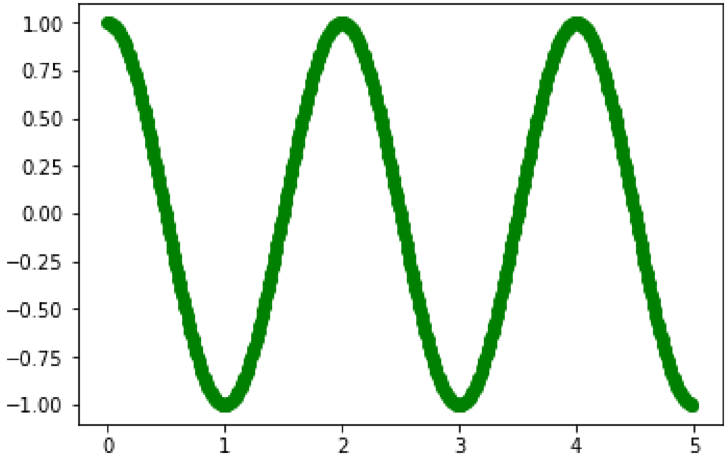


Рисунок 4.9 — График с большим количеством маркеров

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)
```

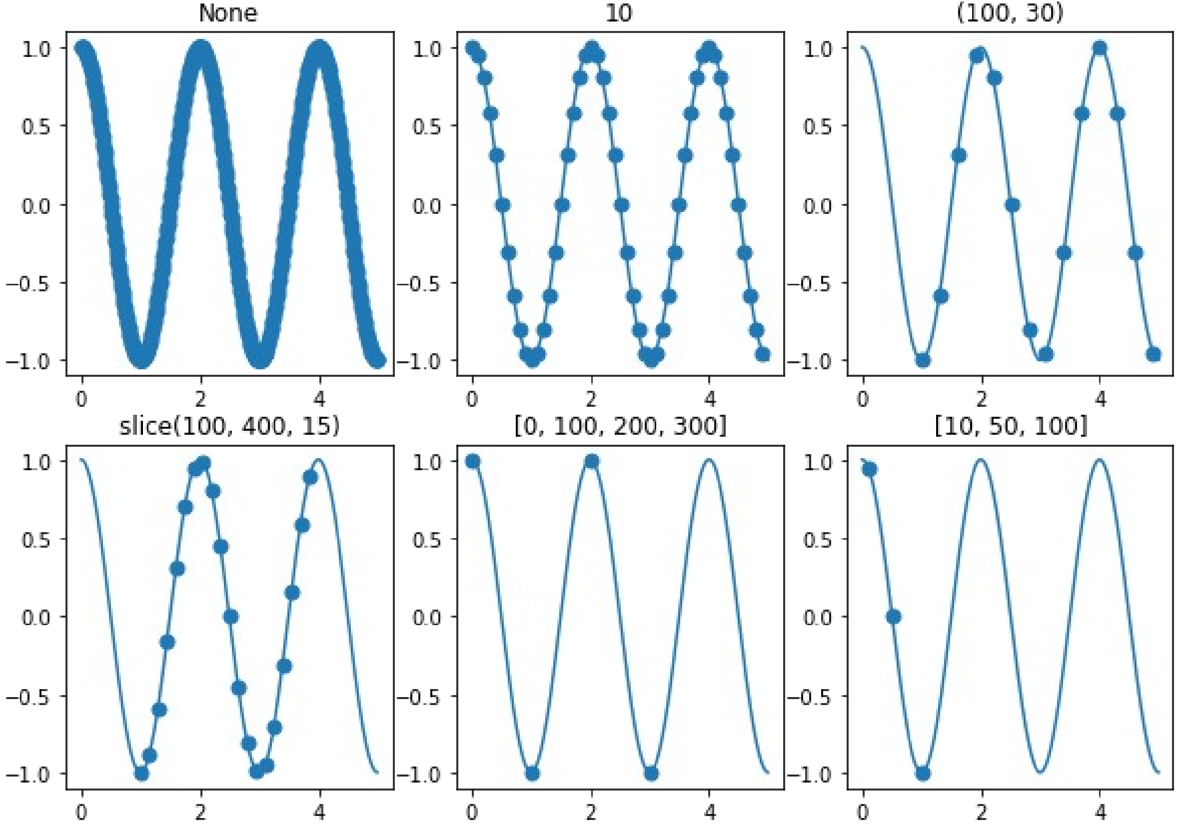


Рисунок 4.10 — Различные варианты маркировки

4.1.4 Обрезка графика

Для того, чтобы отобразить только часть графика, которая отвечает определённому 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)
```

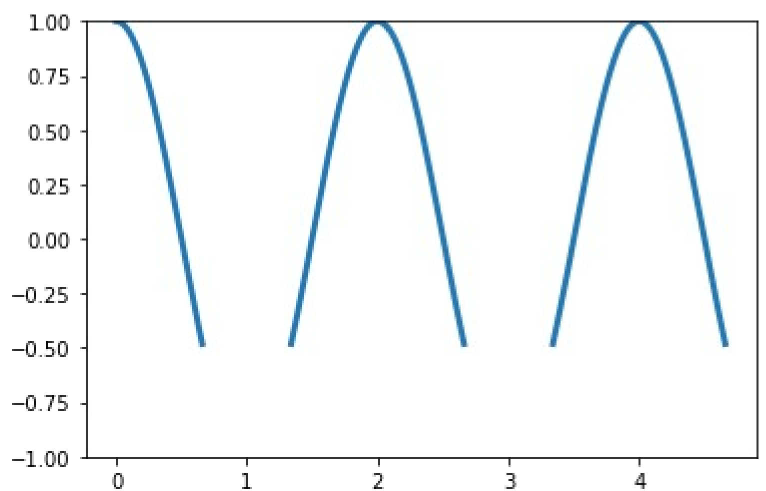


Рисунок 4.11 — Пример обрезки графика

4.2 Ступенчатый, стековый, точечный и другие графики

4.2.1 Ступенчатый график

Ступенчатый график строится с помощью функции `step()`:

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

Параметры функции:

- x: массив
  - Набор данных для оси абсцисс (ось *x*).
- y: массив
  - Набор данных для оси ординат (ось *y*).
- `fmt`: `str`, `optional`
  - Формат линии (см. функцию `plot()`).
- `data`: индексируемый 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 4.12 — 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')
```

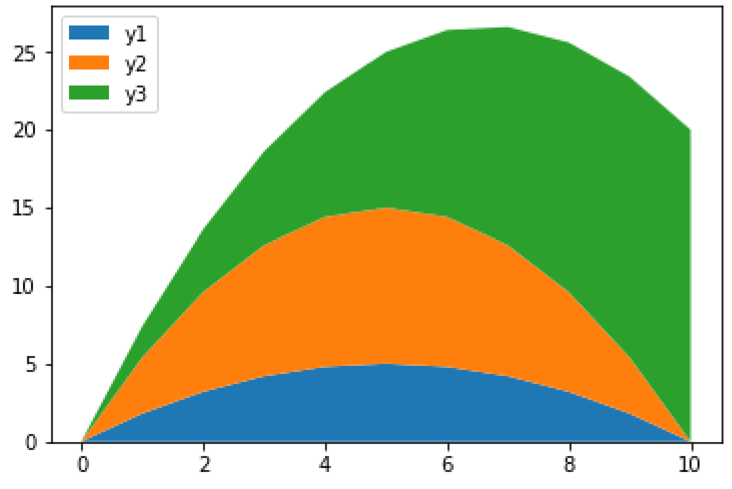


Рисунок 4.13 — Стековый график

Верхний край области *y2* определяется как сумма значений из наборов *y1* и *y2*, *y3* — соответственно сумма *y1*, *y2* и *y3*.

4.2.3 *Stem*-график

Визуально *stem*-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)
```

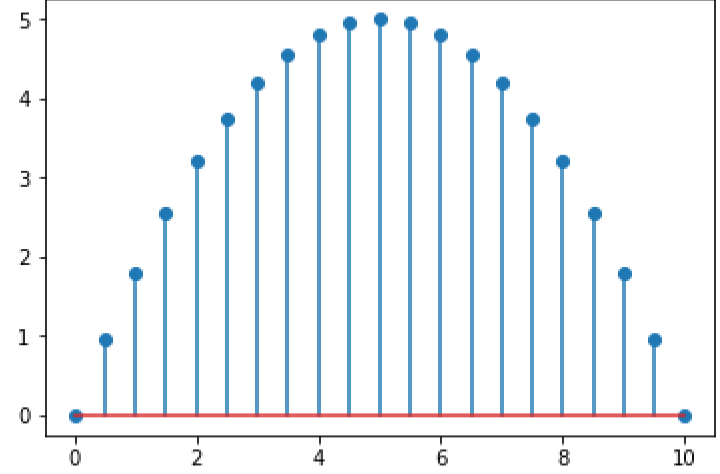


Рисунок 4.14 — *Stem*-график

Дополнительные параметры функции `stem()`:

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

Таблица 4.5 — Стиль вертикальной линии

Символ	Стиль линии
`'-'`	Сплошная линия (*solid line style*).
`'--'`	Штриховая линия (*dashed line style*).
`'-.'`	Штрихпунктирная линия (*dash-dot line style*).
`':'`	Штриховая линия (*dotted line style*).

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

Таблица 4.6 — Формат маркера

Значение	Описание
`'o'`	Круг (*Circle*).
`'+'`	Знак плюс (*Plus sign*).
`'*'`	Звездочка (*Asterisk*).
`'.'`	Точка (*Point*).
`'x'`	Крест (*Cross*).
`'square'` или `'s'`	Квадрат (*Square*).
`'diamond'` или `'d'`	Ромб (*Diamond*).
`'^'`	Треугольник, направленный вниз (*triangle_down*).
`'v'`	Треугольник, направленный вверх (*triangle_up*).
`'<'`	Треугольник, направленный влево (*triangle_left*).
`'>'`	Треугольник, направленный вправо (*triangle_right*).
`'pentagram'` или `'p'`	Пятиугольник (*Five-pointed star (pentagram)*).

'hexagram' или 'h'	Шестиугольник (*Six-pointed star (hexagram)*).
'none'	Нет маркера (*No markers*).

- `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)
```

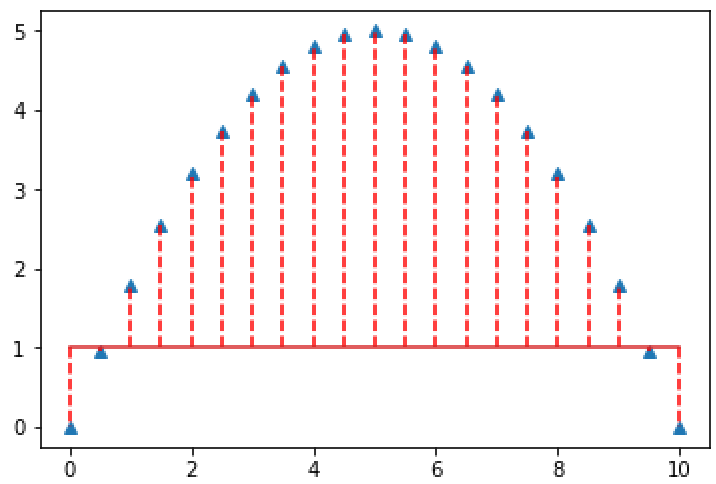


Рисунок 4.15 — Модифицированный *Stem*-график

4.2.4 Точечный график (Диаграмма рассеяния)

Для построения диаграммы рассеяния используется функция `scatter()`. В простейшем виде её можно получить, передав функции `scatter()` координаты *x* и *y*:

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

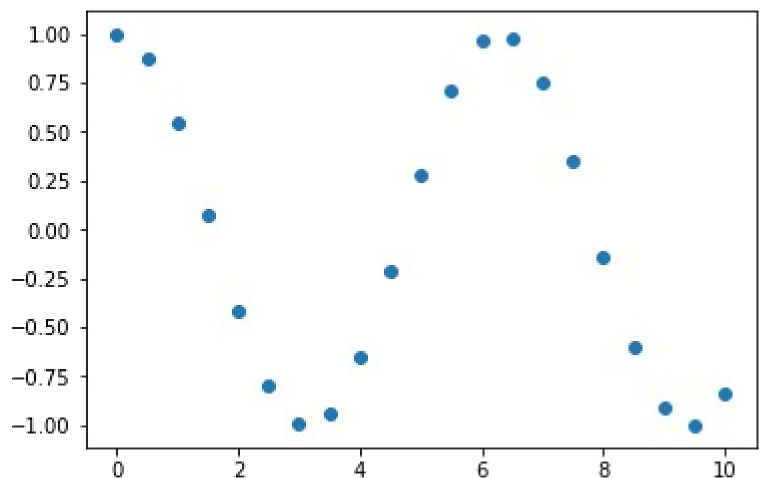


Рисунок 4.16 — Диаграмма распределения (пример 1)

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

Сигнатура вызова функции:

```
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 её параметров:

- x: массив, `shape(n, )`
  - Набор данных для оси абсцисс (ось *x*).

- `y`: массив, `shape(n, )`
    - Набор данных для оси ординат (ось *y*).
- `s`: скалярная величина или массив, `shape(n, )`, `optional`
    - Масштаб точек.
- `c`: цвет[6] или набор цветовых элементов, `optional`
    - Цвет.
- `marker`: `MarkerStyle, optional`
    - Стиль точки.
- `cmap: str, Colormap`[7], `optional`, значение по умолчанию: `None`
    - Цветовая карта (см. “*4.4.1 Цветовые карты (colormaps)*“).
- `norm`: `Normalize`[8], `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}`, цвет[6] или набор цветовых элементов, `optional`.
    - Цвет границы.

6 Один из доступных способов задания цвета (см. раздел “*2.3.2 Цвет линии*”)
7 https://matplotlib.org/api/_as_gen/matplotlib.colors.Colormap.html
8 https://matplotlib.org/api/_as_gen/matplotlib.colors.Normalize.html

Пример работы с параметрами 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')
```

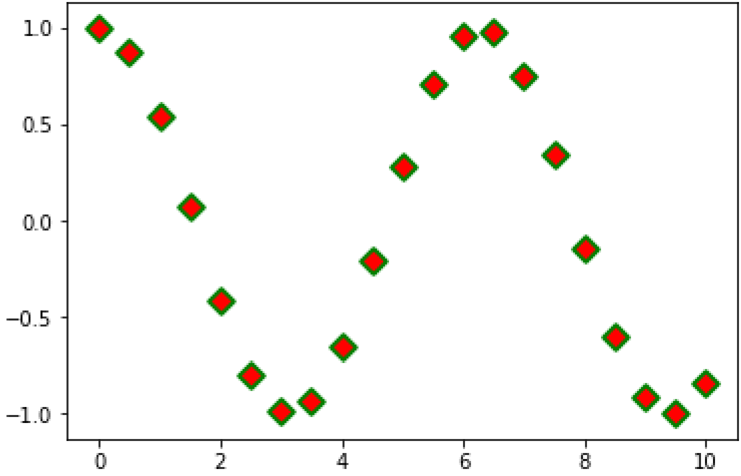


Рисунок 4.17 — Диаграмма распределения (пример 2)

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)
```

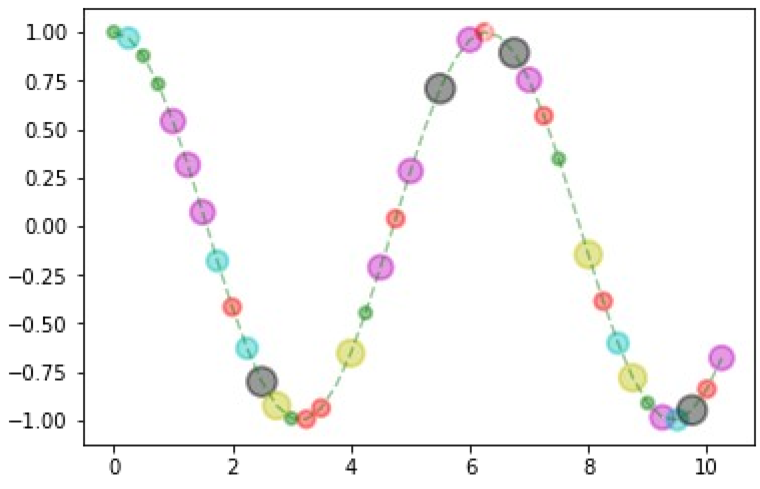


Рисунок 4.18 — Модифицированный точечный график (пример 3)

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)
```

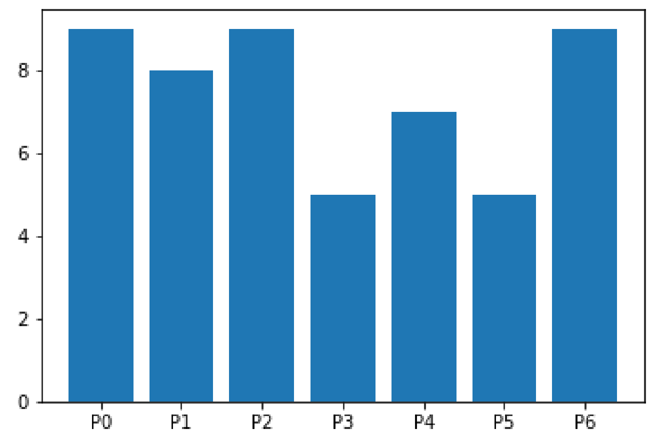


Рисунок 4.19 — Вертикальная столбчатая диаграмма

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

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

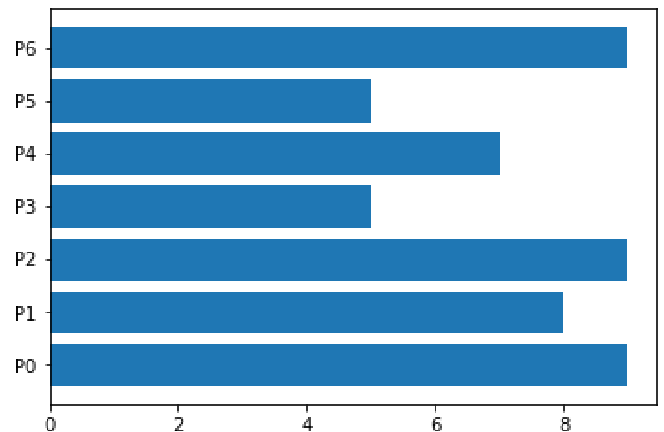


Рисунок 4.20 — Горизонтальная столбчатая диаграмма

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: цвет[9], набор цветовых элементов или optional
    - Цвет столбцов диаграммы.

9 Один из доступных способов задания цвета (см. раздел “*2.3.2 Цвет линии*”)

- `edgecolor`: цвет[10], набор цветовых элементов 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`: цвет[10], набор цветовых элементов 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.

10 Один из доступных способов задания цвета (см. раздел “*2.3.2 Цвет линии*”)

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)
```

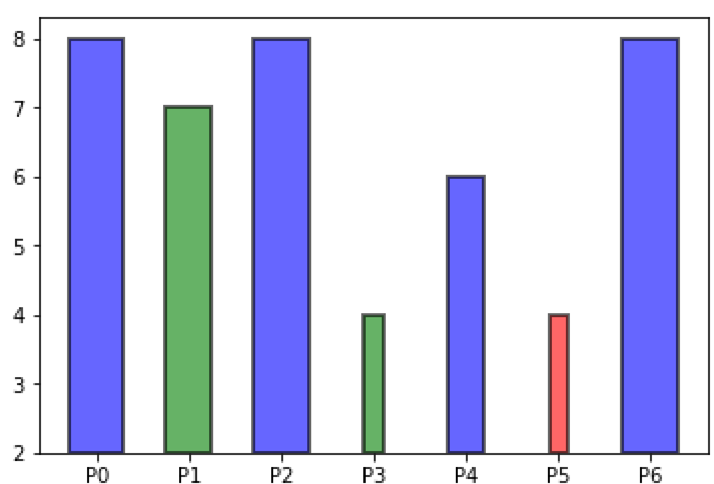


Рисунок 4.21 — Модифицированная столбчатая диаграмма

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()
```

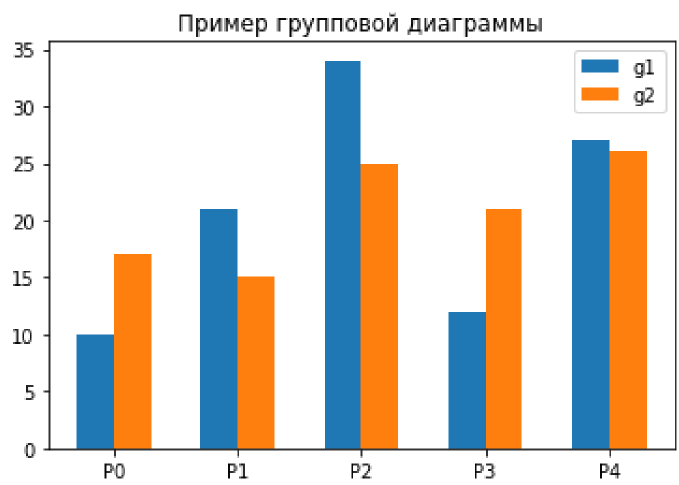


Рисунок 4.22 — Групповая столбчатая диаграмма

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)
```

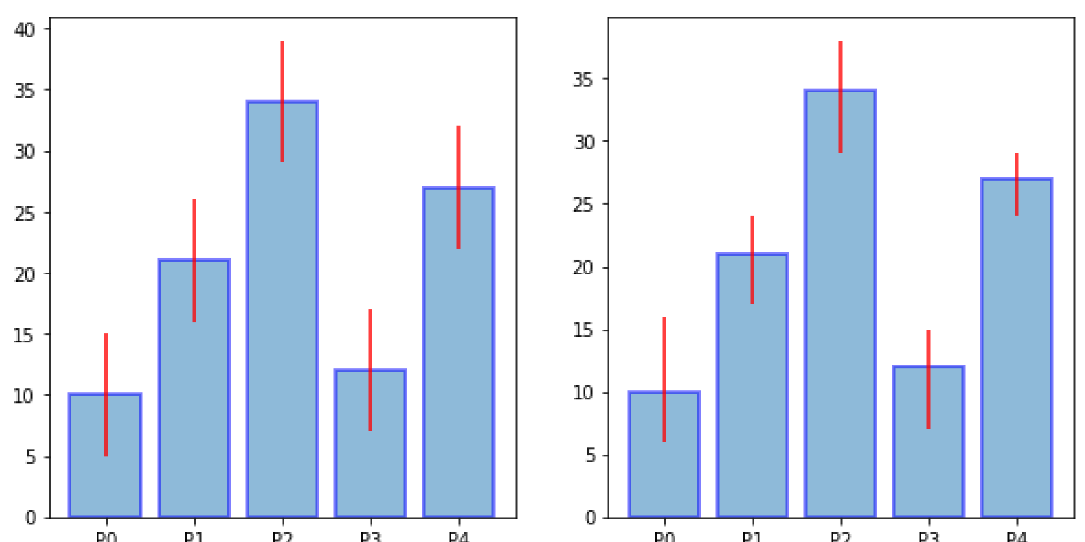


Рисунок 4.23 — Столбчатая диаграмма с *errorbar* элементом

4.3.2 Круговые диаграммы

4.3.2.1 Классическая круговая диаграмма

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

Пример 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')
```

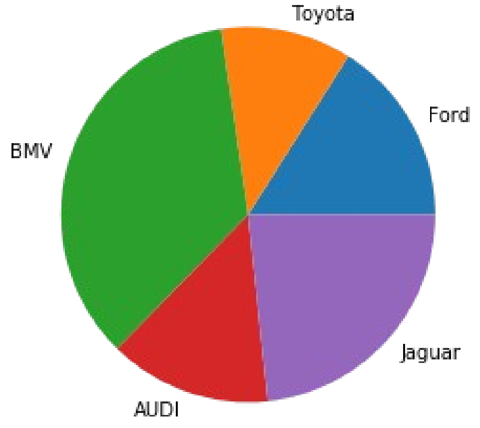


Рисунок 4.24 — Круговая диаграмма

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 такой диаграммы:

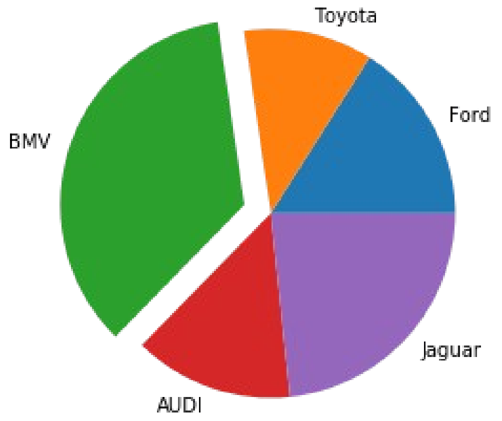


Рисунок 4.25 — Круговая диаграмма с отделенным сектором

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

- `colors`: массив цветовых элементов[11], `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`
  - 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`
  - 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`

11 Один из доступных способов задания цвета (см. раздел “*2.3.2 Цвет линии*”)

- Словарь параметров, определяющих внешний вид текстовых меток (см. класс `matplotlib.text.Text`).

- `center`: список 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')
```

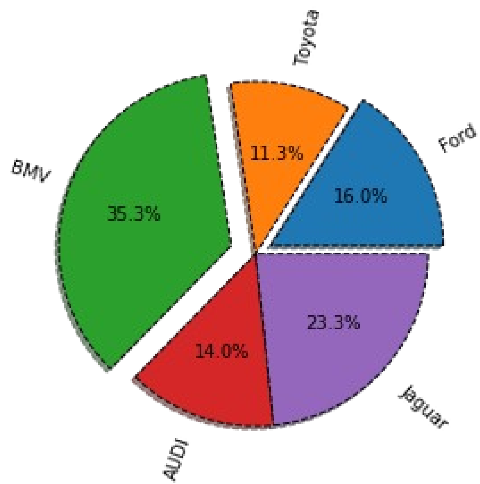


Рисунок 4.26 — Модифицированная круговая диаграмма

4.3.2.2 Вложенные круговые диаграммы

Вложенная круговая диаграмма состоит из двух компонентов: внутренняя её 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 4.27 — Вложенная круговая диаграмма

4.3.2.3 Круговая диаграмма с отверстием

Для того чтобы построить круговую диаграмму с отверстием, необходимо параметру `wedgeprops`, отвечающему за внешний вид долей, задать значение `dict(width=0.5)`:

```
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))
```

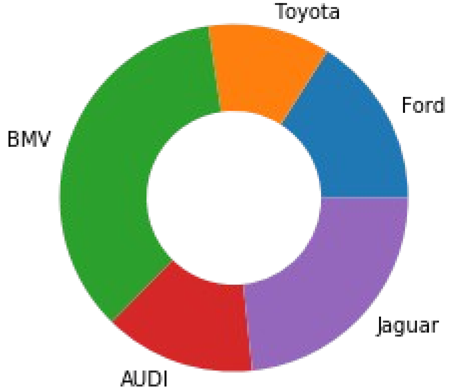


Рисунок 4.28 — Круговая диаграмма с отверстием

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 можно использовать для визуализации наборов данных. Подробное руководство по цветовым картам вы можете найти на официальном сайте Matplotlib[12]. Такую карту можно создать самостоятельно, если среди существующих нет подходящей. На рисунке 4.29 представлены примеры некоторых цветовых карт из библиотеки *Matplotlib*.

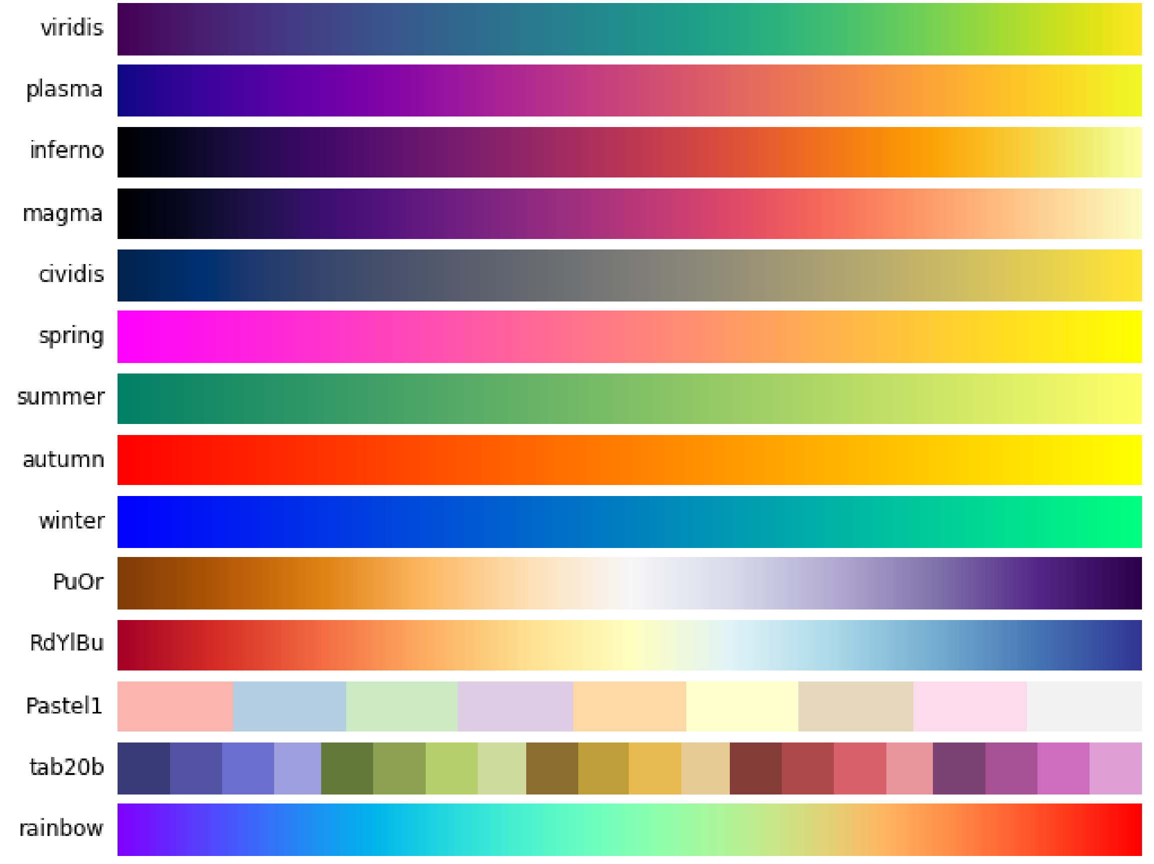


Рисунок 4.29 — Цветовые карты

12 https://matplotlib.org/3.1.0/tutorials/colors/colormaps.html

4.4.2 Построение цветовой сетки

Для построения цветовой сетки можно воспользоваться функцией `imshow()` или `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)
```

Результатом её работы будет изображение логотипа *Matplotlib*.

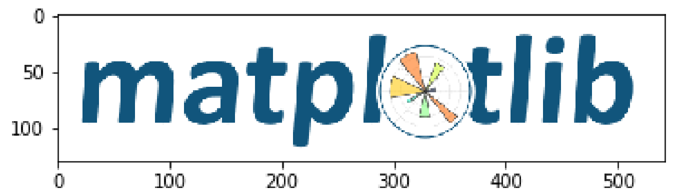


Рисунок 4.30 — Изображение логотипа *Matplotlib*

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

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

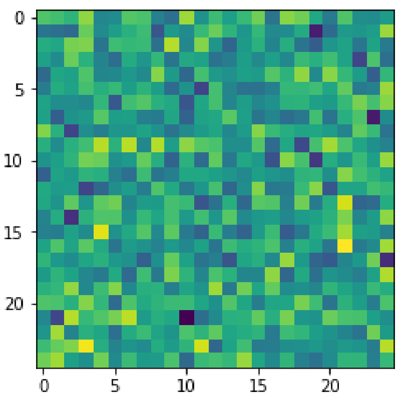


Рисунок 4.31 — Визуализация двумерного набора данных с использованием `imshow()`

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 для изображения (см. "*4.4.1 Цветовые карты (colormaps)*")
- `norm`: `Normalize`, `optional`
    - Нормализация — приведение скалярных данных к диапазону [0,1] перед наложением цветовой карты. 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`: численное значение, `optional`
  - Прозрачность. Задаётся в диапазоне от 0 до 1. Параметр игнорируется для *RGBA*.
- `vmin`, `vmax`: численное 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 начала 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 приведённого 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])
```

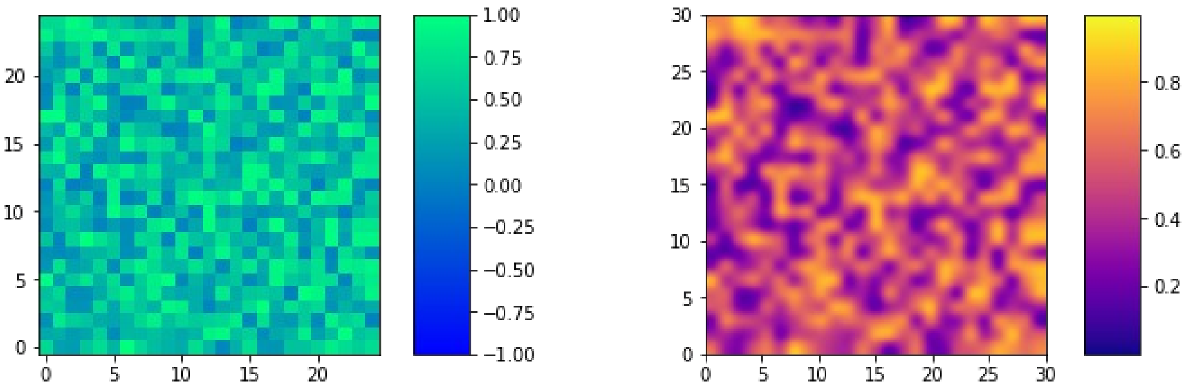


Рисунок 4.32 — Варианты визуализации двумерного набора данных

`pcolormesh()`

Следующая функция для визуализации *2D*-наборов данных, которую мы рассмотрим, будет `pcolormesh()`. В библиотеке *Matplotlib* есть ещё один инструмент с аналогичным функционалом — `pcolor()`, в отличие от неё, 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`: численное значение, `optional`; значение по умолчанию: `None`
  - См. `vmin`, `vmax` в `imshow()`.
- `edgecolors`: `{'none', None, 'face',` цвет[13]`,` массив цветовых элементов`}`, `optional`; значение по умолчанию: `'none'`
  - Цвет границы. Возможны следующие варианты:
    - `'none'` или `''`: без отображения границы;
    - `None`: чёрный цвет;
    - `'face':` используется цвет ячейки;
    - Один из доступных способов задания цвета (см. раздел “*2.3.2 Цвет линии*”).
- `alpha`: численное значение, `optional`; значение по умолчанию: `None`
  - См. `alpha` в `imshow()`.
- `shading`: `{'flat', 'gouraud'}`, `optional`
  - Стиль 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.

13 Один из доступных способов задания цвета (см. раздел “*2.3.2 Цвет линии*”)

Пример использования функции pcolormesh():

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

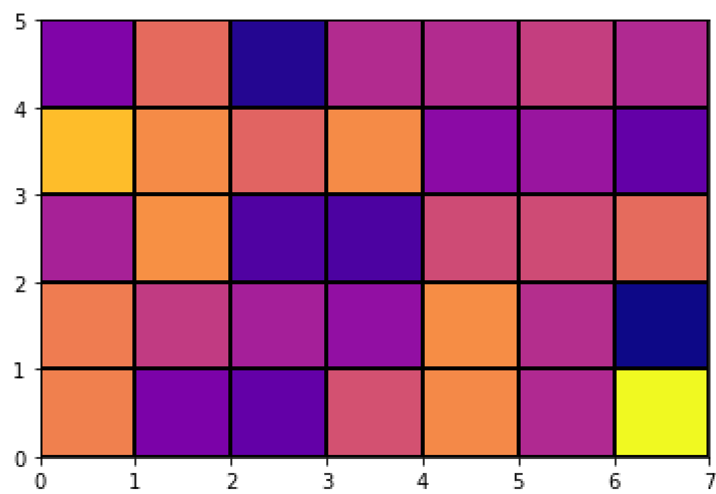


Рисунок 4.33 — Визуализация двумерного набора данных с использованием pcolormesh()

Глава 5. Построение *3D*-графиков. Работа с *mplot3d Toolkit*

До этого момента все графики, которые мы строили, были двумерные, а *Matplotlib* позволяет строить также *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()` из `Axes3D`:

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

Параметры функции *Axes3D.plot*:

- `xs`, `ys`: *1D*-массивы
  - Координаты точек по осям *x* и *y*.
- `zs`: число или *1D*-массив
  - *z* координаты. Если передано скалярное значение, то оно будет присвоено всем точкам графика.
- `zdir`*:* `{'x', 'y', 'z'}`; значение по умолчанию: `'z'`
  - 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')
```

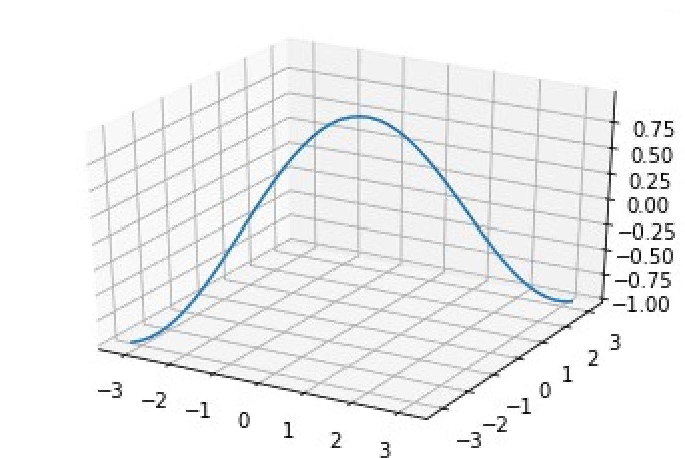


Рисунок 5.1 — Демонстрация работы функции `Axes3D.plot()`

5.2 Точечный график (диаграмма рассеяния)

Для построения диаграммы рассеяния используется функция `scatter()` из `Axes3D`:

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

Параметры функции `Axes3D.scatter()`:

- `xs`, `ys`: *1D*-массив
  - Координаты точек по осям *x* и *y*.
- `zs`: число или *1D*-массив, `optional`; значение по умолчанию: 0
  - Координаты точек по оси *z*. Если передано скалярное значение, то оно будет присвоено всем точкам графика.

- `zdir`*:* `{'x', 'y', 'z', '-x', '-y', '-z'}`, `optional`; значение по умолчанию `'z'`
  - Ось, которая будет принята за *z* направление.
- `s`: число или массив, `optional`; значение по умолчанию 20
  - Размер маркера.
- `c`: цвет[14], массив чисел, массив цветовых элементов, `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 `cmap` 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)
```

14 Один из доступных способов задания цвета (см. раздел “*2.3.2 Цвет линии*”)

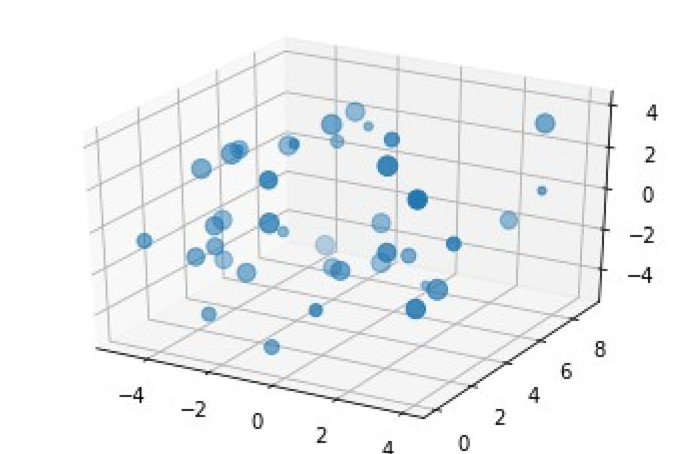


Рисунок 5.2 — Демонстрация работы функции `Axes3D.scatter()`

5.3 Каркасная поверхность

Для построения каркасной поверхности используется функция `plot_wireframe()` из `Axes3D`:

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

Параметры функции `Axes3D.plot_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, которые являются параметрами конструктора класса `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()
```

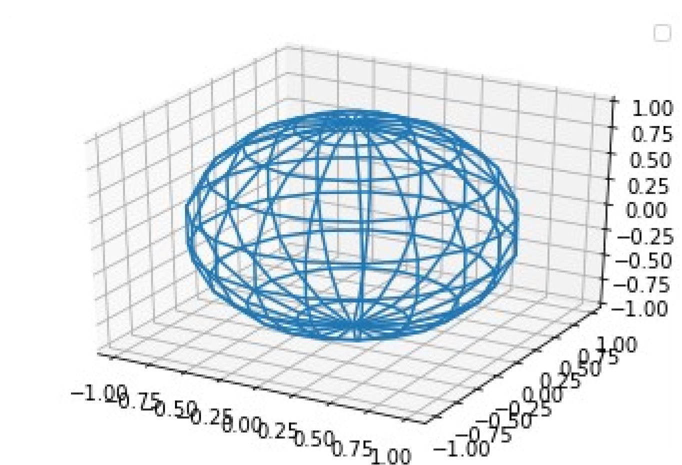


Рисунок 5.3 — Демонстрация работы функции `Axes3D.plot_wireframe()`

5.4 Поверхность

Для построения поверхности используется функция `plot_surface()` из `Axes3D`:

```
Axes3D.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` из “*5.3 Каркасная поверхность*”.
- `rstride`, `cstride` *:* `int`
    - см.`rstride`, `cstride` из “*5.3 Каркасная поверхность*”.
- `color`*:* один из доступных способов задания цвета (см. раздел “*2.3.2 Цвет линии*”).
    - Цвет для элементов поверхности.
- `cmap`: `str` или `Colormap`*,* `optional`
    - Цветовая карта для поверхности (см. "*4.4.1 Цветовые карты (colormaps)*")
- `facecolors`: массив цветовых элементов
    - 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, которые являются параметрами конструктора класса `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()
```

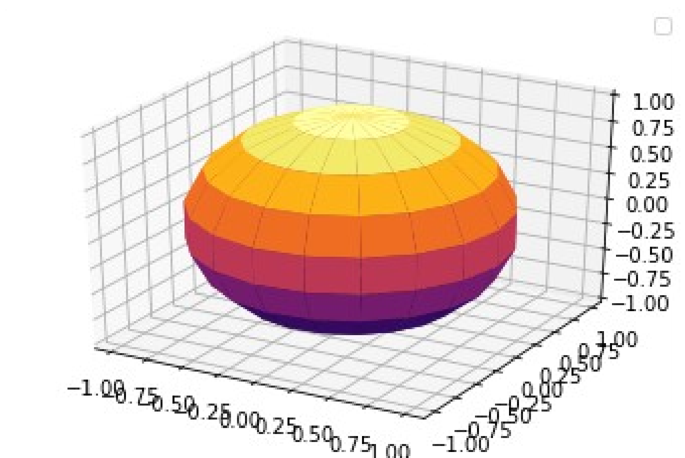


Рисунок 5.4 — Демонстрация работы функции `Axes3D.plot_surface()`

Часть II. Библиотека Seaborn

Введение

Seaborn — это библиотека для решения задач визуализации с ориентацией на работу в области статистики. Базой *seaborn* является библиотека *Matplotlib*, о которой подробно было рассказано в первой части книги.

Seaborn предоставляет:

- удобный *API* для изучения связей в наборах данных;
- автоматический расчёт и отображение моделей линейной регрессии;
- инструменты для исследования внутренней структуры данных;
- инструменты для управления расположением графиков разных типов на одном поле;
- темы оформления графиков.

Инструменты для визуализации данных, которые предоставляет *seaborn* можно разделить на пять групп:

- визуализация отношений в данных;
- визуализация категориальных данных;
- визуализация распределений;
- визуализация линейной регрессии;
- визуализация матричных наборов данных.

Дизайн *API*-функций визуализации данных в *seaborn* спроектирован таким образом, что аргументы различных функций, имеющие одинаковое назначение, имеют одинаковые имена. Потому при изучении

различных инструментов визуализации мы заранее будем давать описание общих аргументов, чтобы не возвращаться к этому вопросу при рассмотрении самих функций. При первом прочтении эти разделы можно пропускать и обращаться к ним в процессе изучения функционала того или иного инструмента.

Глава 6. Быстрый старт

6.1 Установка

6.1.1 Варианты установки *seaborn*

Библиотека *seaborn* входит в состав пакета *Anaconda,* если она у вас установлена, то можете пропустить этот раздел. Руководство по установке *Anaconda* есть на devpractice.ru[15].

6.1.2 Установка *seaborn* через менеджеры *pip* и *conda*

Для установки библиотеки можете воспользоваться пакетным менеджером *pip*:

```
pip install seaborn
```

или *conda*:

```
conda install seaborn
```

В обоих перечисленных вариантах вы получите последний стабильный релиз. Если вас интересует самая свежая, на текущий момент, версия (*development version*), то можно установить *seaborn* напрямую из *github*:

```
pip install git+https://github.com/mwaskom/seaborn.git#egg=seaborn
```

Для своей работы *seaborn* требует наличия следующих модулей:

- *Python* 3.6+
- *numpy*
- *scipy*
- *pandas*
- *matplotlib*
- *statsmodels*

---

15 https://devpractice.ru/python-lesson-1-install/

Если какие-то из них не установлены, то они будут загружены и развёрнуты автоматически.

6.1.3 Проверка корректности установки

Для проверки корректности установки *seaborn* запустите интерпретатор *Python* в командном режиме и введите следующие команды:

```
>>> import seaborn
>>> seaborn.__version__
'0.9.0'
```

Если в результате отобразится номер версии (в примере выше это 0.9.0), то можно считать, что библиотека установлена и ей можно пользоваться.

6.2 Быстрый старт

Перед началом работы необходимо импортировать набор нужных библиотек:

```
import pandas as pd
import numpy as np
import seaborn as sns
```

При импорте *seaborn* ей задается псевдоним *sns*, он является общепринятым среди пользователей этой библиотеки, рекомендуем вам придерживаться этого варианта.

Для экспериментов будем использовать данные, загружаемые с помощью функции `load_dataset()`. В качестве аргумента она принимает имя нужного набора данных.

6.2.1 Построение точечного графика

Загрузим набор данных *mpg*:

```
mpg = sns.load_dataset("mpg")
```

Функция `load_dataset()` формирует набор данных и возвращает его в виде объекта `pandas.DataFrame`. В наборе *mpg* содержится информация о характеристиках ряда автомобилей. Более подробную информацию об нём можете прочитать в autompg-dataset[16].

Для визуальной оценки содержимого набора данных воспользуемся методом `head()` объекта класса `DataFrame`:

```
mpg.head()
```

	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin	name
0	18.0	8	307.0	130.0	3504	12.0	70	usa	chevrolet chevelle malibu
1	15.0	8	350.0	165.0	3693	11.5	70	usa	buick skylark 320
2	18.0	8	318.0	150.0	3436	11.0	70	usa	plymouth satellite
3	16.0	8	304.0	150.0	3433	12.0	70	usa	amc rebel sst
4	17.0	8	302.0	140.0	3449	10.5	70	usa	ford torino

Построим зависимость ускорения (*acceleration*) от количества лошадиных сил (*horsepower*), при этом размер точки будет определяться количеством цилиндров:

```
sns.relplot(x="horsepower", y="acceleration", size="cylinders", data=mpg)
```

16 https://www.kaggle.com/uciml/autompg-dataset

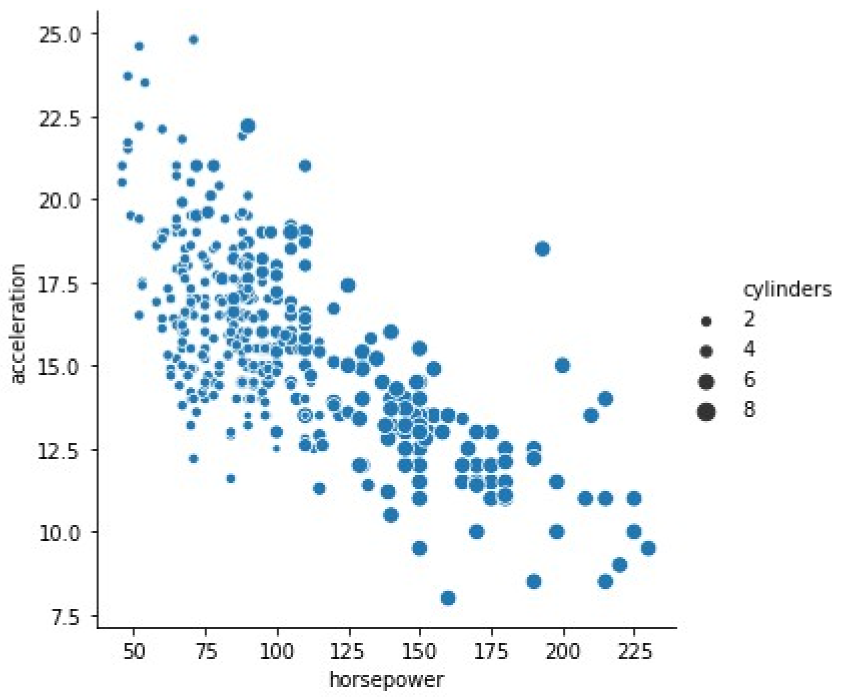


Рисунок 6.1 — Демонстрация работы функции `relplot()`, вариант: диаграмма рассеяния

6.2.2 Построение линейного графика

Для демонстрации работы функции построения линейного графика загрузим набор данных *flights*, содержащий информацию о количестве пассажиров, которые воспользовались авиатранспортом:

```
flights = sns.load_dataset("flights")
flights.head()
```

	year	month	passengers
0	1949	January	112
1	1949	February	118
2	1949	March	132
3	1949	April	129
4	1949	May	121

Построим зависимость количества перевезённых пассажиров (*passengers*) от года (*year*):

```
sns.relplot(x="year", y="passengers", kind="line", legend="full",
data=flights);
```

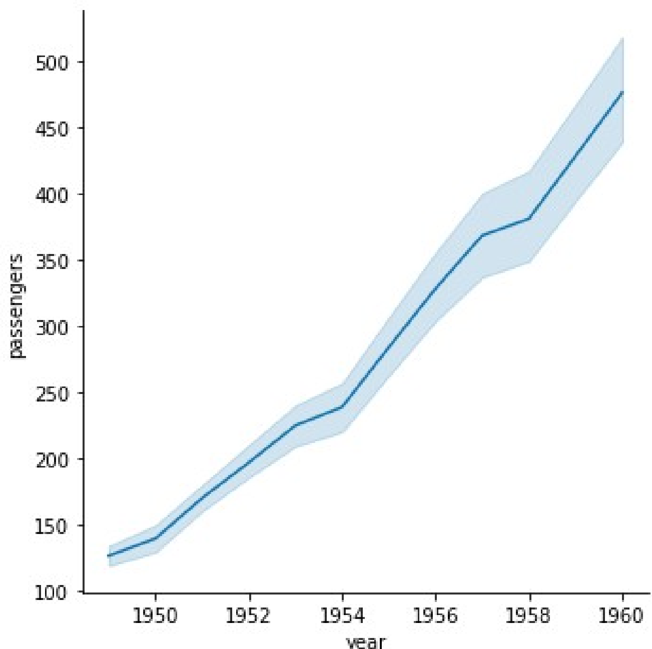


Рисунок 6.2 — Демонстрация работы функции `relplot()`, вариант: линейный график

6.2.3 Работа с категориальными данными

Рассмотрим работу с категориальными данными на примере набора *iris* (Ирисы Фишера):

```
iris = sns.load_dataset("iris")
iris.head()
```

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

Загруженный набор данных является эталонным для изучения алгоритмов классификации, он представляет собой информацию о 150 экземплярах ириса по 50 на каждый отдельный вид: Ирис щетинистый (*setosa*), Ирис Виргинский (*virginica*) и Ирис разноцветный (*versicolor*). Для каждого экземпляра определены следующие параметры:

- Длина наружной доли околоцветника (*sepal_length*);
- Ширина наружной доли околоцветника (*sepal_width*);
- Длина внутренней доли околоцветника (*petal_length*);
- Ширина внутренней доли околоцветника (*petal_width*).

Для визуализации этого набора воспользуемся функцией `catplot()`:

```
sns.catplot(x="species", y="sepal_length", kind="swarm", data=iris);
```

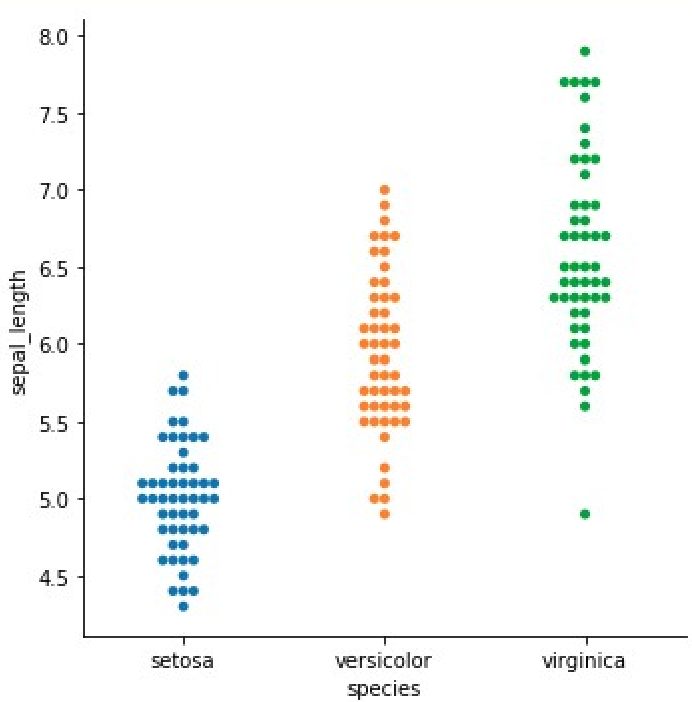


Рисунок 6.3 — Демонстрация работы функции `catplot()`

При работе с категориальными данными часто используется диаграмма “ящик с усами”, она строится с помощью функции `boxplot()`:

```
sns.boxplot(x="species", y="sepal_length", data=iris)
```

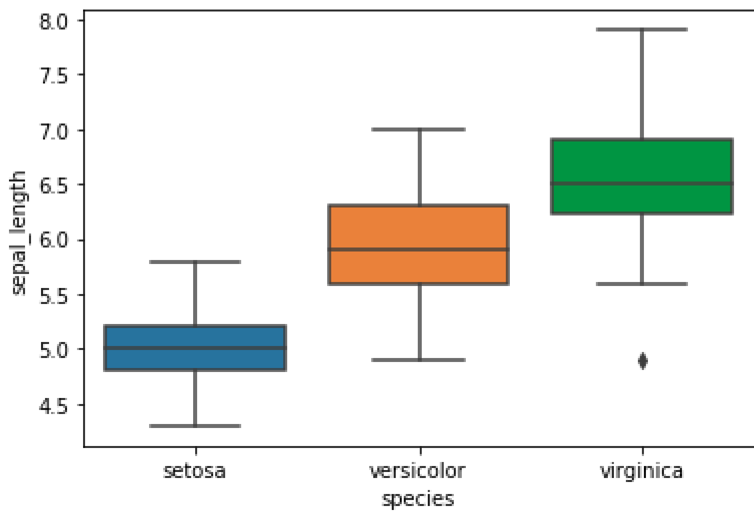


Рисунок 6.4 — Демонстрация работы функции boxplot()

На этом закончим краткий обзор библиотеки *seaborn*. Ключевая её особенность состоит в том, что, используя минимум настроек, можно получить графики и диаграммы с прекрасным визуальным оформлением. Далее мы подробно рассмотрим инструменты для визуализации данных и настройки внешнего вида графиков, которые предоставляет *seaborn*.

Глава 7. Настройка внешнего вида графиков

Библиотека *seaborn* предоставляет инструменты для работы с цветовым оформлением, настройки стилей, сетки и осей графиков, компоновки графиков, работы с легендой и шрифтами.

7.1 Стили *seaborn*

Особенностью *seaborn* является то, что графики и диаграммы, которые с помощью неё можно построить, имеют красивый внешний вид. Разработчики постарались сделать так, чтобы пользователю требовалось минимум усилий для настройки оформления результата своей работы. Самый простой и быстрый способ задать оформление — это использовать один из заранее подготовленных стилей. Также есть возможность создавать и использовать свои стили с индивидуальным оформлением.

Для задания стиля используется функция `set_style()`, которая имеет следующую сигнатуру:

```
set_style(style=None, rc=None)
```

Параметры функции:

- `style`: `dict`, `None` или значение из набора `{'darkgrid', 'whitegrid', 'dark', 'white', 'ticks'}`
    - Словарь с параметрами или имя стиля из подготовленного набора.
- `rc`: `dict`, `optional`
    - Отвечает за переопределение параметров в переданном через аргумент `style` стиле.

Получить список параметров, отвечающих за оформление графика можно с помощью функции axes_style():

```
axes_style(style=None, rc=None)
```

Список и функциональное назначение аргументов этой функции совпадает с приведённым для set_style().

Для начала познакомимся с доступным набором стилей, который можно использовать. Для демонстрации воспользуемся набором данных *flights*. Загрузим его:

```
flights = sns.load_dataset("flights")
```

Будем строить зависимость количества пассажиров, выбравших авиатранспорт для передвижения от года с помощью функции lineplot().

Seaborn предоставляет следующий набор стилей: darkgrid, whitegrid, dark, white, ticks.

Стиль darkgrid:

```
sns.set_style("darkgrid")
sns.lineplot(x='year', y='passengers', data=flights)
```

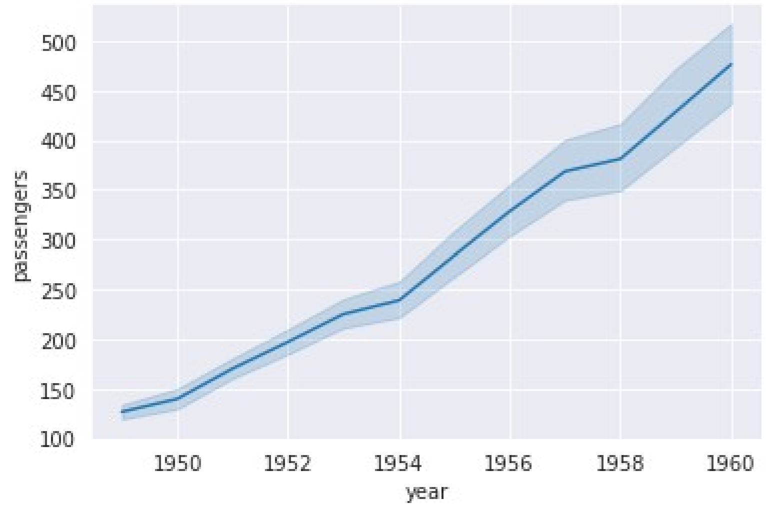


Рисунок 7.1 — Стиль оформления darkgrid

Стиль `whitegrid`:

```
sns.set_style("whitegrid")
sns.lineplot(x='year', y='passengers', data=flights)
```

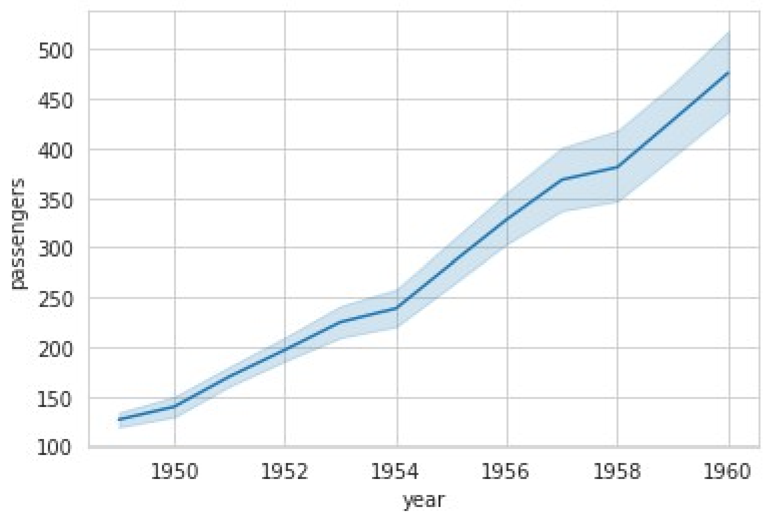


Рисунок 7.2 — Стиль оформления `whitegrid`

Стиль `dark`:

```
sns.set_style("dark")
sns.lineplot(x='year', y='passengers', data=flights)
```

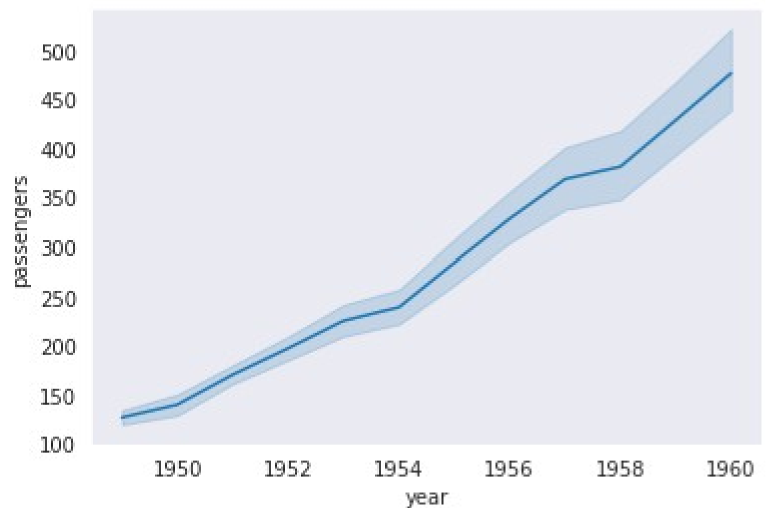


Рисунок 7.3 — Стиль оформления dark

Стиль `white`:

```
sns.set_style("white")
sns.lineplot(x='year', y='passengers', data=flights)
```

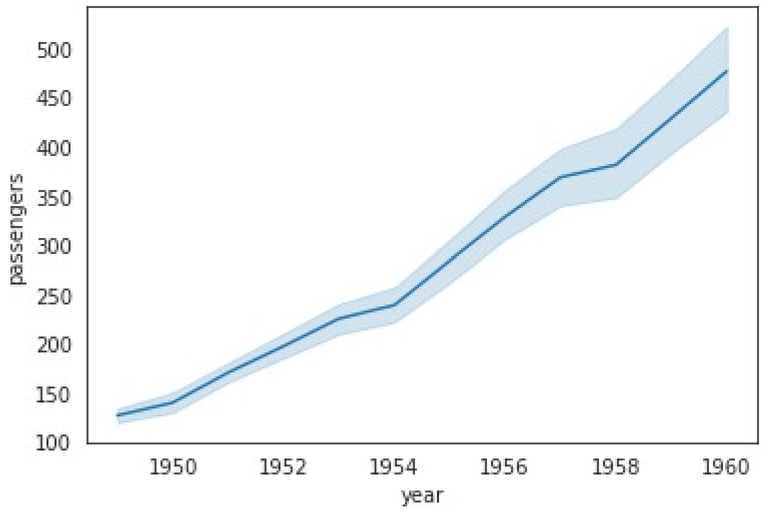


Рисунок 7.4 — Стиль оформления white

Стиль `ticks`:

```
sns.set_style("ticks")
sns.lineplot(x='year', y='passengers', data=flights)
```

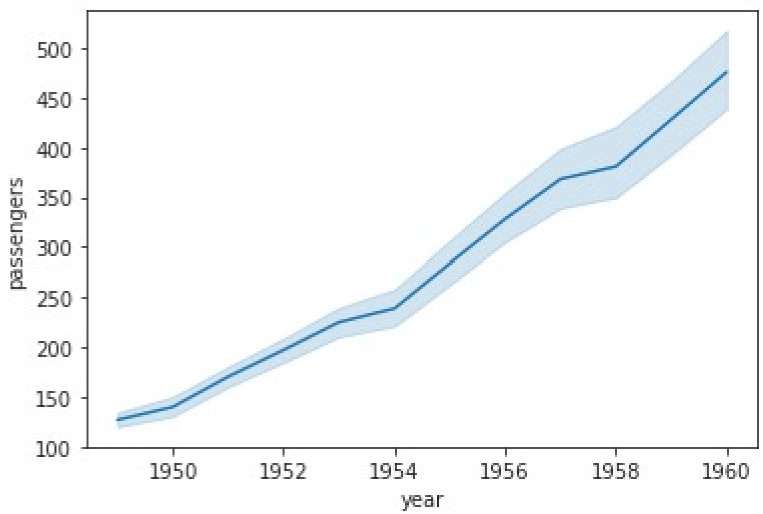


Рисунок 7.5 — Стиль оформления ticks

Список доступных для изменения параметров можно получить с помощью функции axes_style():

```
>>> sns.axes_style()
{'axes.axisbelow': True,
 'axes.edgecolor': 'white',
 'axes.facecolor': '#EAEAF2',
 'axes.grid': True,
 'axes.labelcolor': '.15',
 'axes.spines.bottom': True,
 'axes.spines.left': True,
 'axes.spines.right': True,
 'axes.spines.top': True,
 'figure.facecolor': 'white',
 'font.family': ['sans-serif'],
 'font.sans-serif': ['Arial',
  'DejaVu Sans',
  'Liberation Sans',
  'Bitstream Vera Sans',
  'sans-serif'],
 'grid.color': 'white',
 'grid.linestyle': '-',
 'image.cmap': 'rocket',
 'lines.solid_capstyle': 'round',
 'patch.edgecolor': 'w',
 'patch.force_edgecolor': True,
 'text.color': '.15',
 'xtick.bottom': False, 'xtick.color': '.15',
 'xtick.direction': 'out', 'xtick.top': False,
 'ytick.color': '.15', 'ytick.direction': 'out',
 'ytick.left': False, 'ytick.right': False}
```

Все параметры имеют имена, указывающие на их функциональное назначение. Модифицируем ряд параметров стиля `whitegrid`:

```
sns.set_style("whitegrid", {'axes.labelcolor':"b", 'axes.edgecolor':'r',
'xtick.color':'g'})
sns.lineplot(x='year', y='passengers', data=flights)
```

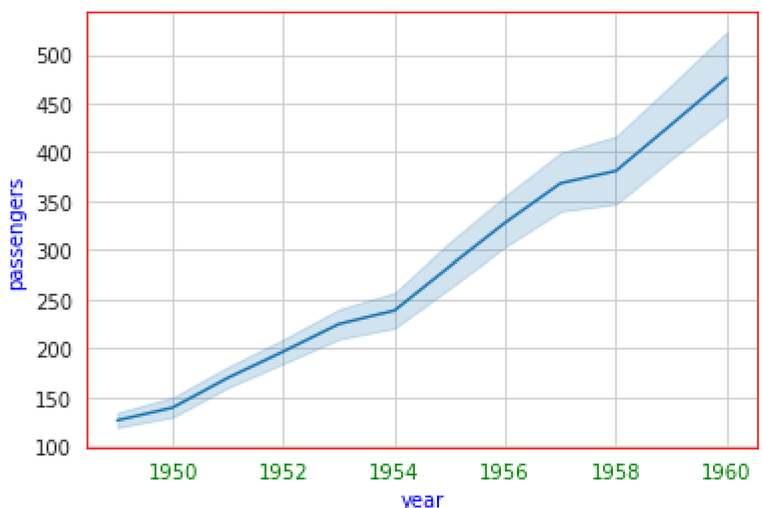


Рисунок 7.6 — Модифицированный стиль оформления `whitegrid`

7.2 Контексты *seaborn*

Контексты в *seaborn* используются для управления масштабом изображения. В зависимости от того, где будет использоваться график: станет он частью статьи или презентации, выбирается тот или иной масштаб его элементов.

Для задания контекста используется функция `set_context()`:

```
set_context(context=None, font_scale=1, rc=None)
```

Рассмотрим параметры функции:

- `context`: `dict`, параметр из набора: `{'paper', 'notebook', 'talk', 'poster'}`, `None`
  - Словарь с параметрами либо символьное имя контекста.

- `font_scale`: `float`, `optional`
  - Масштабный коэффициент для изменения размера шрифта.
- `rc`: `dict`, `optional`
  - Словарь с параметрами для переопределения свойств, заданного через аргумент `context`, контекста.

Для получения списка параметров контекста используется функция `plotting_context()`:

```
plotting_context(context=None, font_scale=1, rc=None)
```

Назначение параметров функции совпадает с тем, что приведено для `set_context()`.

Для демонстрации работы с контекстами воспользуемся набором данных *iris*:

```
iris = sns.load_dataset("iris")
```

Контекст paper:

```
sns.set_context("paper")
sns.scatterplot(x='sepal_length', y='petal_length', data=iris)
```

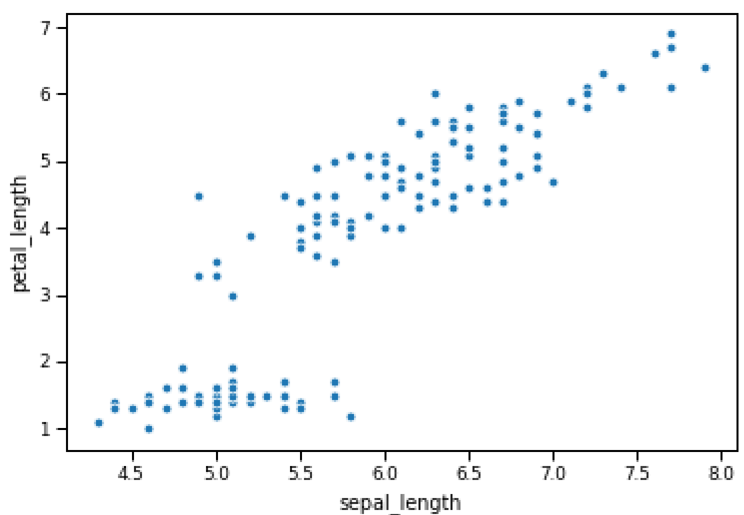


Рисунок 7.7 — Контекст paper

Контекст `notebook`:

```
sns.set_context("notebook")
sns.scatterplot(x='sepal_length', y='petal_length', data=iris)
```

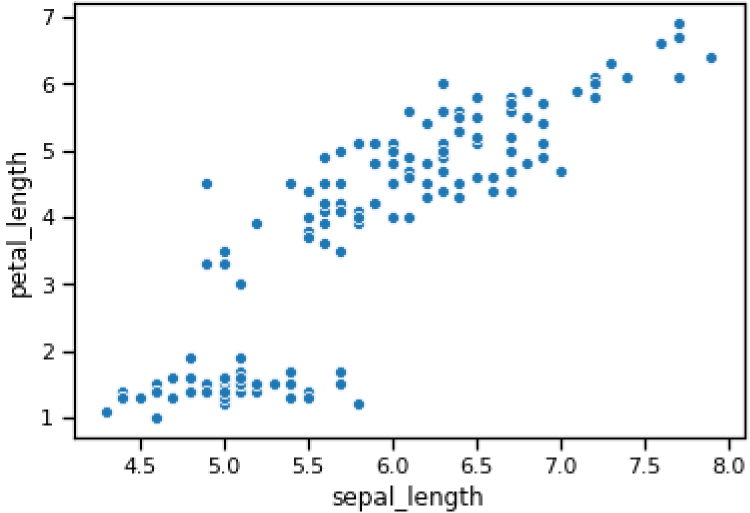


Рисунок 7.8 — Контекст notebook

Контекст `talk`:

```
sns.set_context("talk")
sns.scatterplot(x='sepal_length', y='petal_length', data=iris)
```

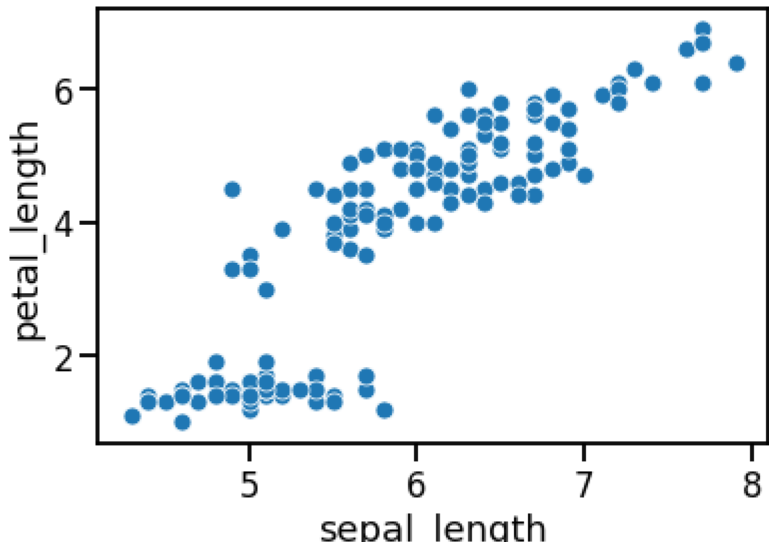


Рисунок 7.9 — Контекст talk

Контекст poster:

```
sns.set_context("poster")
sns.scatterplot(x='sepal_length', y='petal_length', data=iris)
```

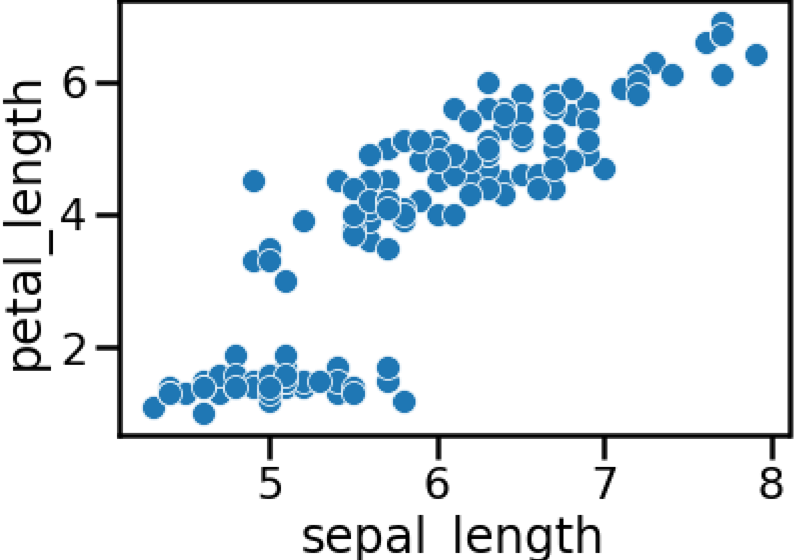


Рисунок 7.10 — Контекст poster

Посмотрим на список параметров контекста, которые можно изменять. Если нужно узнать настройки конкретного контекста, то его название нужно передать в качестве аргумента:

```
>>> sns.plotting_context("notebook")
{'axes.labelsize': 12,
 'axes.linewidth': 1.25,
 'axes.titlesize': 12,
 'font.size': 12,
 'grid.linewidth': 1,
 'legend.fontsize': 11,
 'lines.linewidth': 1.5,
 'lines.markersize': 6,
 'patch.linewidth': 1,
 'xtick.labelsize': 11,
 'xtick.major.size': 6,
 'xtick.major.width': 1.25,
```

```
'xtick.minor.size': 4,
'xtick.minor.width': 1,
'ytick.labelsize': 11,
'ytick.major.size': 6,
'ytick.major.width': 1.25,
'ytick.minor.size': 4,
'ytick.minor.width': 1}
```

Изменим некоторые из параметров:

```
sns.set_context("notebook", font_scale=1.5, rc={'lines.markersize':15,
'xtick.labelsize': 15.0, 'ytick.labelsize': 15.0})
sns.scatterplot(x='sepal_length', y='petal_length', data=iris)
```

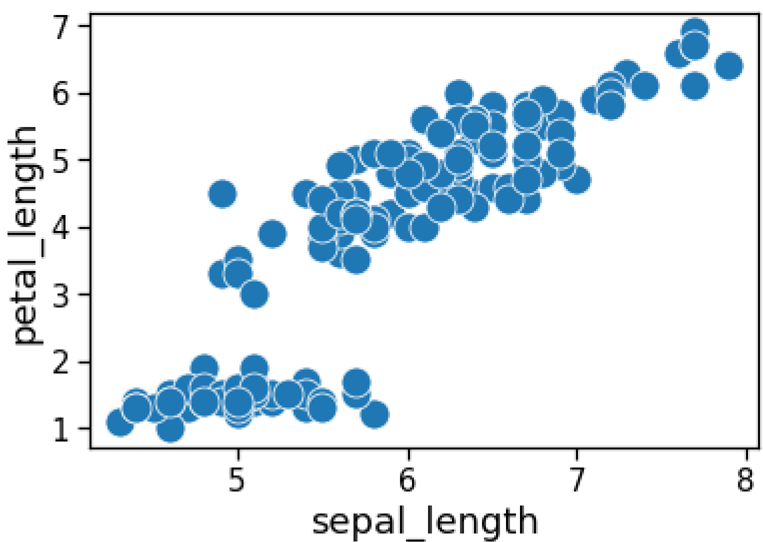


Рисунок 7.11 — Контекст notebook с модифицированными параметрами

7.3 Настройка сетки и осей

Как вы могли видеть в примерах, демонстрирующих работу со стилями и контекстами: визуальное представление сетки на поле графика и осей, изменяется с выбором стиля (контекста). В этом разделе будут рассмотрены варианты более тонкой настройки этих атрибутов графика.

7.3.1 Сетка

За настройку сетки отвечают параметры стиля, представленные в таблице 7.1 и параметры контекста из таблицы 7.2.

Таблица 7.1 — Параметры стиля для настройки сетки

Параметр	Описание
`axes.grid`	Отвечает за отображение сетки на поле графика. `True` — отобразить сетку, `False` — нет.
`grid.color`	Цвет линии сетки.
`grid.linestyle`	Стиль линии сетки.

Таблица 7.2 — Параметры контекста для настройки сетки

Параметр	Описание
`grid.linewidth`	Толщина линии сетки.

Рассмотрим работу с сеткой на примерах. Для начала установим стиль `whitegrid` и контекст notebook:

```
sns.set_style("whitegrid")
sns.set_context("notebook")
```

Построим диаграмму рассеяния для набора *iris*:

```
sns.scatterplot(x='sepal_length', y='petal_length', data=iris)
```

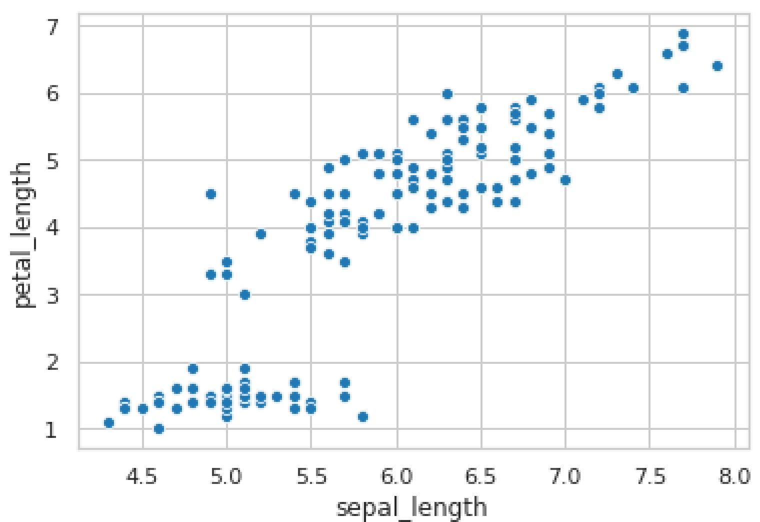


Рисунок 7.12 — Внешний вид графика с примененным стилем `whitegrid` и контекстом `notebook`

Теперь изменим параметры сетки:

```
sns.set_style("whitegrid", rc={'grid.color': '#ff0000', 'grid.linestyle': '--'})
sns.set_context("notebook", rc={'grid.linewidth': 3.0})
```

Посмотрим, что получилось:

```
sns.scatterplot(x='sepal_length', y='petal_length', data=iris)
```

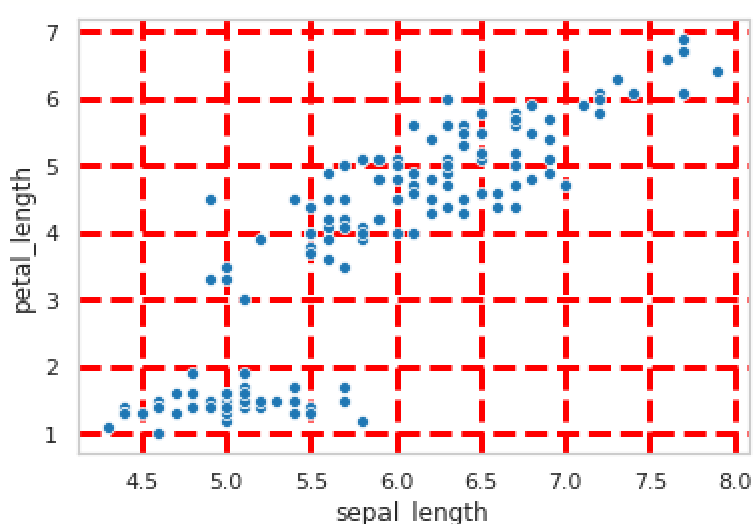


Рисунок 7.13 — Внешний вид графика с изменёнными параметрами сетки

7.3.2 Поле и оси графика

Параметры стиля и контекста, отвечающие за настройку поля и осей графика, начинаются с приставки `'axes'` (см. таблицы 7.3 и 7.4).

Таблица 7.3 — Параметры стиля для настройки сетки

Параметр	Описание
`axes.axisbelow`	Размещение сетки под (`True / 'line'`) или над (`False`) диаграммой.
`axes.edgecolor`	Цвет границы поля графика.
`axes.facecolor`	Цвет поля графика.
`axes.labelcolor`	Цвет подписей осей.
`axes.spines.bottom`	Размещение оси в нижней части поля (`True`).
`axes.spines.left`	Размещение оси в левой части поля (`True`).
`axes.spines.right`	Размещение оси в правой части поля (`True`).
`axes.spines.top`	Размещение оси в верхней части поля (`True`).

Таблица 7.4 — Параметры контекста для настройки сетки

Параметр	Описание
`axes.linewidth`	Толщина осей графика.
`axes.titlesize`	Размер заголовка.

Приведём пример использования этих параметров:

```
sns.set_style("whitegrid", rc={'axes.axisbelow': 'line',
'axes.edgecolor': 'red',
'axes.facecolor': 'lightgreen',
'axes.labelcolor': 'red',
'axes.spines.bottom': True,
'axes.spines.left': True,
'axes.spines.right': False,
'axes.spines.top': False})

sns.set_context("notebook", rc={'axes.labelsize': 15.0,
'axes.linewidth': 2.5,
'axes.titlesize': 20.0})

sp = sns.scatterplot(x='sepal_length', y='petal_length', data=iris)
sp.set_title("Axes tune")
```

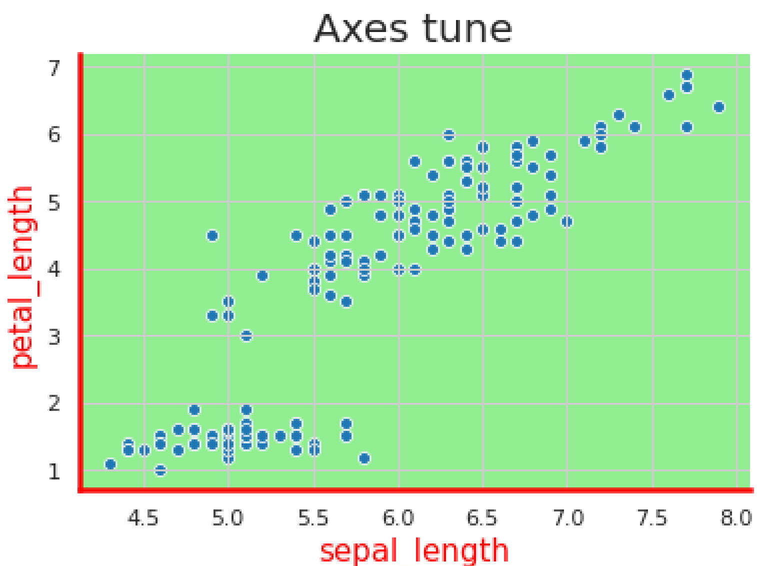


Рисунок 7.14 — Внешний вид графика с изменёнными параметрами поля и осей графика

Ещё одним важным элементом являются **тики —** отметки, которые наносятся на оси графика. Для работы с ними используются параметры с приставками `'xtick'` и `'ytick'`, см. таблицы 7.5 и 7.6.

Таблица 7.5 — Параметры стиля для настройки тиков

Параметр	Описание
`xtick.bottom`	Размещение тиков на нижней оси (`True`).
`xtick.top`	Размещение тиков на верхней оси (`True`).
`xtick.direction`	Направление линий тиков наружу (`'out'`) или вовнутрь (`'in'`) на оси *x*.
`xtick.color`	Цвет линий тиков на оси *x*.
`ytick.left`	Размещение тиков на левой оси (`True`).
`ytick.right`	Размещение тиков на правой оси (`True`).
`ytick.direction`	Направление линий тиков наружу (`'out'`) или вовнутрь (`'in'`) на оси *y*.
`ytick.color`	Цвет линий тиков на оси *y*.

Таблица 7.6 — Параметры контекста для настройки тиков

Параметр	Описание
`xtick.labelsize`	Размер меток тиков на оси *x*.
`xtick.major.size`	Длинна меток основных тиков на оси *x*.
`xtick.major.width`	Ширина меток основных тиков на оси *x*.
`xtick.minor.size`	Длинна меток дополнительных тиков на оси *x*.

xtick.minor.width	Ширина меток дополнительных тиков на оси *x*.
ytick.labelsize	Размер меток тиков на оси *y*.
ytick.major.size	Длинна меток основных тиков на оси *y*.
ytick.major.width	Ширина меток основных тиков на оси *y*.
ytick.minor.size	Длинна меток дополнительных тиков на оси *y*.
ytick.minor.width	Ширина меток дополнительных тиков на оси *y*.

Для отображения вспомогательной линейки (*minor*) необходимо подключить локатор с помощью функции `set_minor_locator()`. Такая манипуляция — это уже работа на уровне *Matplotlib* (*seaborn* реализована поверх неё), обычно, при работе с *seaborn*, такого делать не приходится.

Для демонстрации оформления вспомогательной линейки приведём пример:

```
sns.set_style("whitegrid", rc={'xtick.bottom': True,
'xtick.color': 'red', 'xtick.direction': 'in',
'xtick.top': True,
'ytick.color': 'red', 'ytick.direction': 'in',
'ytick.left': True,
'ytick.right': True})

sns.set_context("notebook", rc={'xtick.labelsize': 15.0,
'xtick.major.size': 6.0, 'xtick.major.width': 1.25,
'xtick.minor.size': 4.0, 'xtick.minor.width': 2.0,
'ytick.labelsize': 15.0,
```

```
'ytick.major.size': 6.0, 'ytick.major.width': 1.25,
'ytick.minor.size': 4.0, 'ytick.minor.width': 1.0})
import matplotlib as mpl

sp = sns.scatterplot(x='sepal_length', y='petal_length', data=iris)
sp.get_xaxis().set_minor_locator(mpl.ticker.AutoMinorLocator())
sp.get_yaxis().set_minor_locator(mpl.ticker.AutoMinorLocator())
sp.grid(b=True, which='minor', color='lightgreen', linewidth=0.5)
```

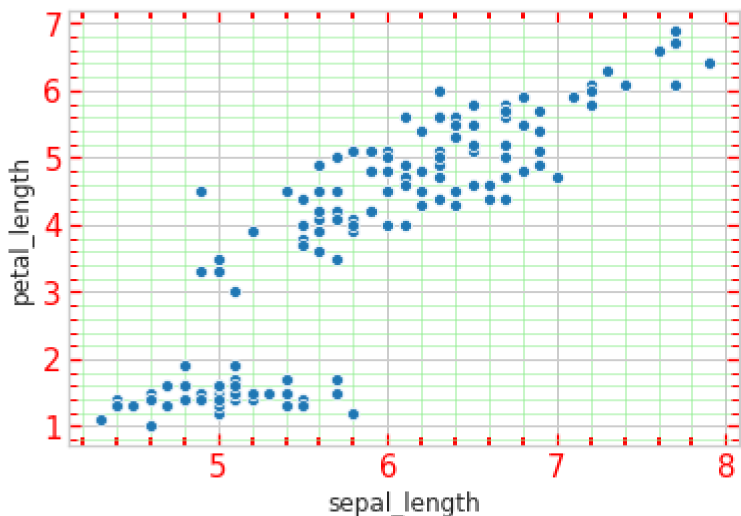


Рисунок 7.15 — График с измененным оформлением тиков

7.4 Легенда

Легенда на графике отображается автоматически, если вы используете какой-то дополнительный параметр для группировки данных по тем или иным признакам. Для выделения групп можно использовать цвет (параметр hue) или размер (параметр `size`).

Приведём пример графика с легендой:

```
sns.scatterplot(x='sepal_length', y='petal_length', hue="species",
data=iris)
```

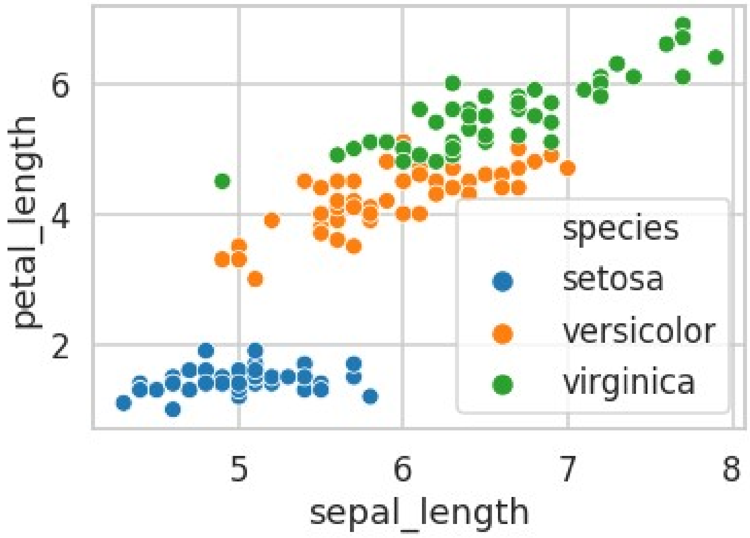


Рисунок 7.16 — График с легендой

Непосредственно сама библиотека *seaborn* практически не предоставляет инструментов для настройки визуального оформления легенды. Единственный параметр — это `legend.fontsize` для изолированного управления размером шрифта легенды из группы параметров настройки контекста:

```
sns.set_style("whitegrid")
sns.set_context("talk", rc={'legend.fontsize': 10.0})
```

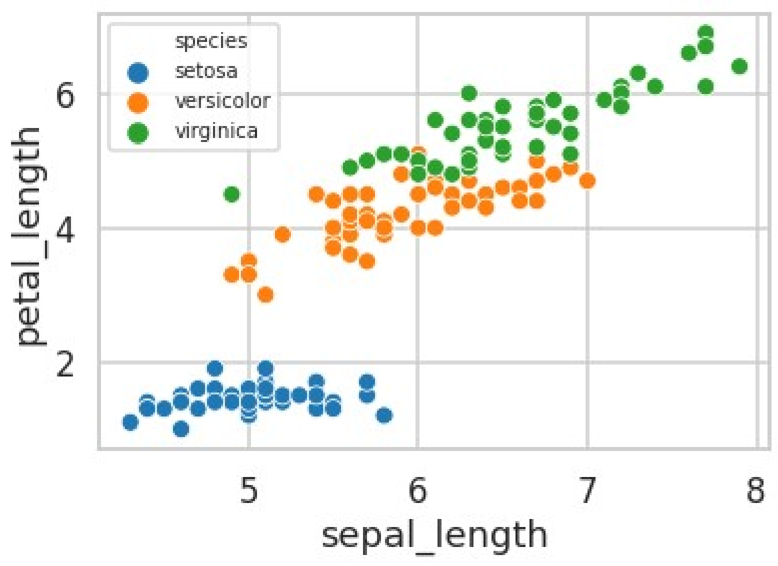


Рисунок 7.17 — Легенда с измененным размером шрифта

Если требуется более тонкая настройка: расположение легенды, её размер и т.п., то для этого следует воспользоваться инструментами, которые предлагает *Matplotlib*:

```
sns.set_style("whitegrid")
sns.set_context("notebook")
sp = sns.scatterplot(x='sepal_length', y='petal_length', hue="species", data=iris)
sp.legend(loc='center right', bbox_to_anchor=(1.35, 0.5), ncol=1)
```

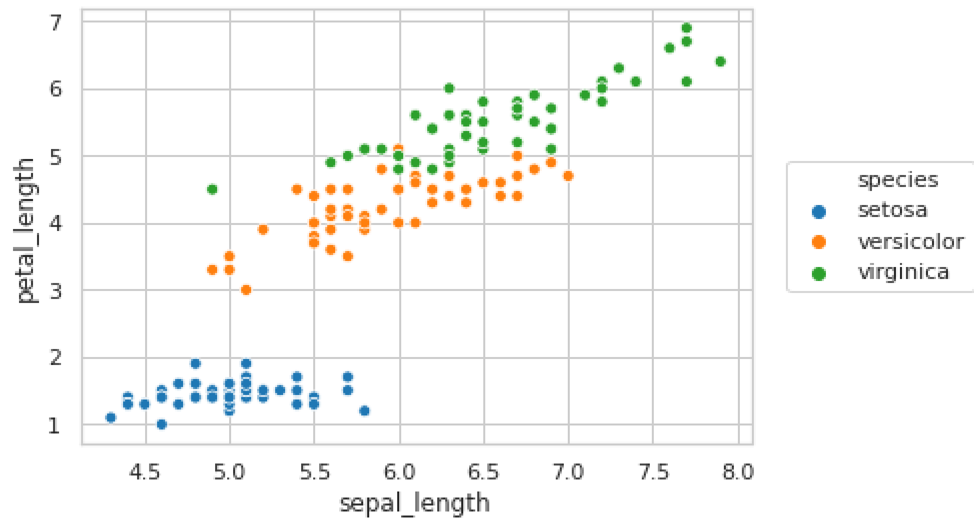


Рисунок 7.18 — Изменение расположения легенды с помощью метода из библиотеки *matplotlib*

7.5 Шрифт

За настройку шрифта отвечают параметры `font.family` (из стиля) и `font_scale` (из контекста):

```
sns.set_style("whitegrid", rc={ 'font.family': ['fantasy']})
sns.set_context("notebook", font_scale=1.5)
sp = sns.scatterplot(x='sepal_length', y='petal_length', hue="species", data=iris)
sp.legend(loc='center right', bbox_to_anchor=(1.5, 0.5), ncol=1)
```

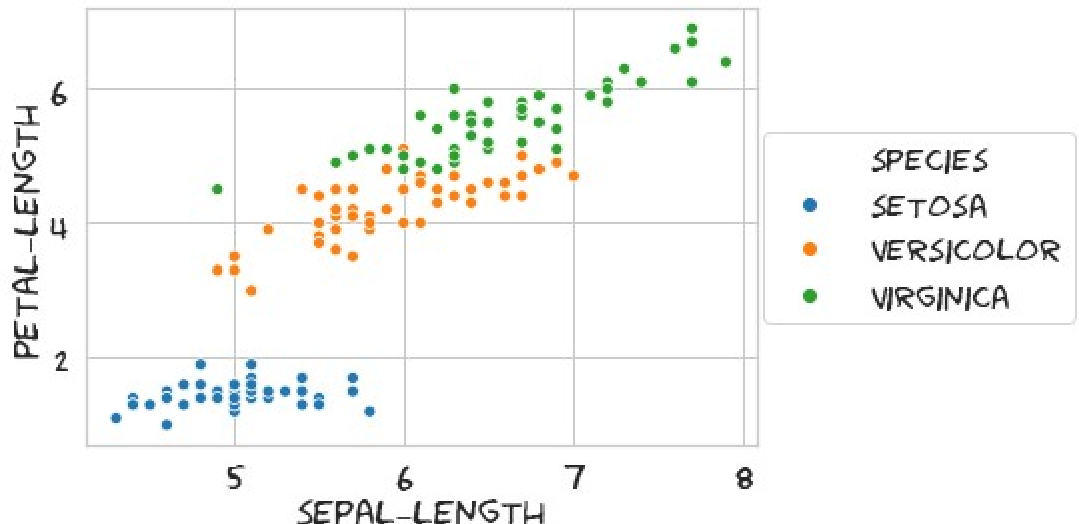


Рисунок 7.19 — Изменение шрифта на графике

7.6 Работа с цветом

Seaborn использует цветовые схемы, которые предоставляет *Matplolib*, про это подробно написано в разделе "*2.3.2 Цвет линии*". Для работы с цветовым оформлением библиотека предоставляет ряд функций, которые подробно рассмотрены далее в этом разделе.

Функция `color_palette()`

Возвращает список цветов.

Прототип функции:

```
color_palette(palette=None, n_colors=None, desat=None)
```

Параметры функции:

- `palette`: `None`, `string`, `list`, `optional`
  - Имя палитры или набор цветов. Если значение равно `None`, то будет возвращена текущая палитра.
- `n_colors`: `int`, `optional`
  - Количество цветов в палитре.

- `desat`: `float`, `optional`
    - Коэффициент управления насыщенностью, 1 — это исходное представление цвета.

Функция `set_palette()`

Устанавливает цветовую палитру в качестве текущей. Назначение параметров совпадает с указанными для функции `color_palette()`.

Прототип функции:

```
set_palette(palette, n_colors=None, desat=None, color_codes=False)
```

Дополнительные параметры:

- `color_codes`: `bool`
    - Если параметр равен `True`, то производится переопределение цветов, связанных с короткими именами (`'r'`, `'g'`, …) в соответствии с задаваемой палитрой.

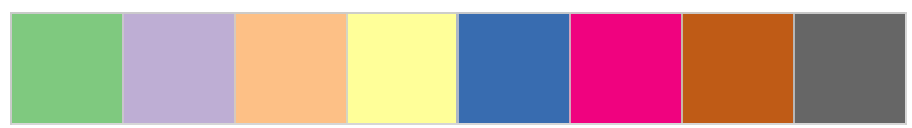
Посмотрим выборочно на некоторые наборы цветов.

Текущая цветовая схема:

```
sns.palplot(sns.color_palette())
```

Палитра `Accent`:

```
sns.set_palette("Accent")
sns.palplot(sns.color_palette())
```

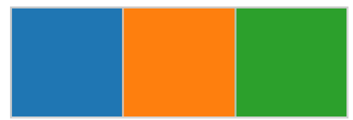


Из палитры `Accent` возьмём только три цвета:

```
sns.set_palette("Accent", n_colors=3)
sns.palplot(sns.color_palette())
```

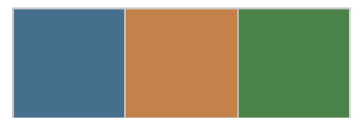
Палитра `tab10`:

```
sns.set_palette("tab10", n_colors=3, desat=1)
sns.palplot(sns.color_palette())
```



Изменим насыщенность:

```
sns.set_palette("tab10", n_colors=3, desat=0.5)
sns.palplot(sns.color_palette())
```



```
sns.set_palette("tab10", n_colors=3, desat=0.1)
sns.palplot(sns.color_palette())
```

Функция `set_color_codes()`

Изменяет яркость и насыщенность если для задания цвета, используются однобуквенные сокращения.

Прототип функции:

```
set_color_codes(palette='deep')
```

Параметр функции:

- palette: {'deep', 'muted', 'pastel', 'dark', 'bright', 'colorblind'}
    - Имя палитры из библиотеки *seaborn*.

Приведём несколько примеров.

Палитра deep:

```
sns.set_color_codes("deep")
sns.barplot(x='species', y='petal_length', data=iris, palette=['r', 'g', 'b'])
```

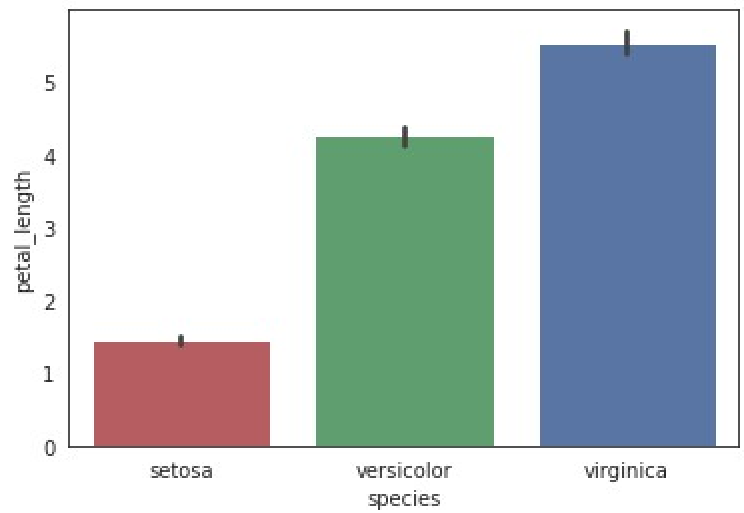


Рисунок 7.20 — Палитра deep

Палитра muted:

```
sns.set_color_codes("muted")
sns.barplot(x='species', y='petal_length', data=iris, palette=['r', 'g', 'b'])
```

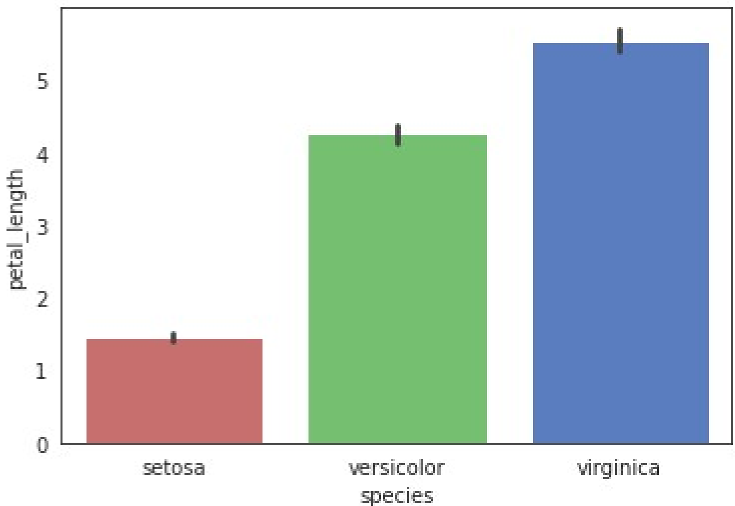


Рисунок 7.21 — Палитра muted

Палитра dark:

```
sns.set_color_codes("dark")
sns.barplot(x='species', y='petal_length', data=iris, palette=['r', 'g', 'b'])
```

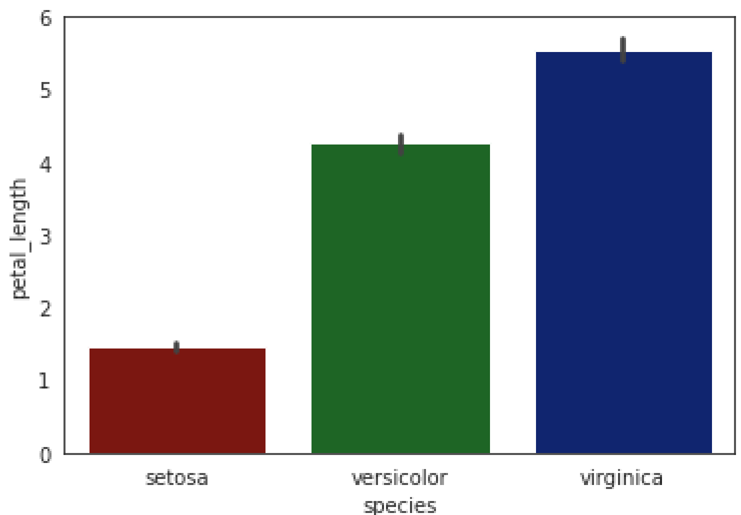


Рисунок 7.22 — Палитра dark

Глава 8. Визуализация отношений в данных

Seaborn предоставляет функции (см. таблицу 8.1) для визуализации отношений в данных в виде линейных графиков и диаграмм рассеяния.

Таблица 8.1 — Функции *seaborn* для визуализации отношений в данных

Функция	Описание
`relplot`	Общий интерфейс для визуализации отношений с параметрами для настройки компоновки.
`lineplot`	Линейный график с семантической группировкой.
`scatterplot`	Диаграмма рассеяния с семантической группировкой.

Представленный набор функций условно можно разделить на две группы: первая — специализированные функций для построения графиков определённого типа, к ним относятся `lineplot()` и `scatterplot()`; вторая группа — функции уровня фигуры (подложки? на которой отображаются графики): `relplot()`, она обеспечивает функционал инструментов из первой группы и дополнительно предоставляет возможности по компоновке графиков: позволяет выводить графики в отдельных полях в зависимости от значения того или иного признака. Можно сказать, что `relplot()` является более высокоуровневым интерфейсом для `lineplot()` и `scatterplot()`.

8.1 Общие параметры функций

Рассмотрим параметры, которые являются общими для функций визуализации отношений в данных.

8.1.1 Базовые аргументы

Базовые аргументы функций:

- `x`, `y`: имена переменных из набора `data`, `optional`
  - Связывают ось *x* и *y* с конкретными признаками из набора данных, переданного через параметр `data`. Данные должны иметь числовой тип. Для функций `lineplot()` и `scatterplot()` допустимо передавать вектора с данными напрямую. Параметры могут иметь значение `None`, в этом случае, будет визуализирован весь набор данных из `data`.
- `data`: `DataFrame`, `optional`
  - Набор данных в формате `pandas.DataFrame`, в котором столбцы — это имена переменных, строки — значения. Имена столбцов, данные из которых нужно визуализировать, передаются в параметры `x` и `y`.

8.1.2 Параметры для повышения информативности графиков

Для придания большей информативности графикам функции визуализации отношений в данных предоставляют следующие параметры:

- `ci`: `int`, `'sd'` или `None`, `optional`
  - Определяет размер отображаемого доверительного интервала. Если значение равно `'sd'`, то вместо доверительного интервала будет отображено стандартное отклонение.
- `hue`: имя переменной из набора `data`, `optional`
  - Задаёт признак в наборе данных, который будет использован для цветового разделения данных. Визуально группы будут представлены в виде отдельных линий (точек), отличающихся

цветом. Функции `lineplot()` и `scatterplot()` позволяют передавать вектора с данными в параметр напрямую, для `relplot()` это недопустимо.

- `palette`: имя палитры, `list`, `dict`, `optional`
  - Палитра, которая будет использована для цветового разделения набора данных по значениями признака, указанного в `hue`. *Seaborn* предоставляет набор различных цветовых схем, которые можно использовать, более подробно о них рассказано в разделе "*7.5 Работа с цветом*". Можно указать свою цветовую схему через словарь, в котором определяется соответствие значений признака, переданного в `hue`, и цвета из библиотеки *Matplotlib*.
- `hue_order`: `list`, `optional`
  - Задаёт порядок применения цветов для данных из набора, переданного через параметр `hue`.
- `hue_norm`: `tuple` или объект класса `Normalize`, `optional`
  - Нормализация данных может применятся, если набор данных, указанный в `hue` параметре, имеет числовой тип. Для категориальных данных нормализация не используется.
- `size`: имя переменной из набора `data`, `optional`
  - Задаёт признак в наборе данных, который будет использован для разделения данных по размеру. Визуально группы будут представлены в виде отдельных линий (точек), отличающихся шириной (размером). Функции `lineplot()` и `scatterplot()` позволяют передавать вектора с данными в параметр напрямую, для `relplot()` это не допустимо.
- `size_order`: `list`, `optional`

    - Задаёт порядок распределения толщины линии между элементами из набора данных, заданного через параметр `size`.
- `size_norm`: `tuple`, объект класса `Normalize`, `optional`
    - Определяет нормализацию для данных для набора, заданного через параметр `size`. Может использоваться только для численных значений.
- `style`: имя переменной из набора `data`, `optional`
    - Задаёт признак в наборе данных, который будет использован для разделения данных по стилю. Визуально группы будут представлены в виде отдельных линий (точек), отличающихся стилем линии и/или видом маркера. Функции `lineplot()` и `scatterplot()` позволяют передавать вектора с данными в параметр напрямую, для `relplot()` это не допустимо.
- `markers`: `bool`, `list`, `dict`, `optional`
    - Определяет тип маркеров. Если параметр равен `False`, то маркеры использоваться не будут, если `True`, то будут использованы маркеры по умолчанию. Тип маркера можно задать через словарь, устанавливающий соответствие между значениями из набора данных, переданного через параметр `style` и кодом маркера. Коды маркеров соответствуют используемым в *Matplotlib*.
- `style_order` : `list`, `optional`
    - Задаёт порядок применения стилей.

8.2 Линейный график. Функция `lineplot()`

Для визуализации отношений в данных в виде линейного графика применяется функция `lineplot()`. Такой тип графика чаще всего используется для визуализации временных рядов и зависимостей между

переменными, имеющими непрерывный характер, например, такими как температура, давление, стоимость и т.д.

8.2.1 Знакомство с функцией `lineplot()`

Начнём с построения простой зависимости одной переменной от другой. Для начала импортируем необходимые библиотеки:

```
import seaborn as sns
import numpy as np
import pandas as pd
```

Установим стиль оформления графиков (см. раздел "*7.1 Стили seaborn*"):

```
sns.set_style("darkgrid")
```

Сгенерируем набор случайных данных и воспользуемся функцией `lineplot()` для отображения зависимости между ними:

```
np.random.seed(123)
x = [i for i in range(10)]
y = np.random.randint(10, size=len(x))
sns.lineplot(x, y)
```

В первой строке задаётся зерно генератора, это нужно сделать для того, чтобы каждый раз при запуске программы генерировались одни и те же данные. После этого создаём наборы данных для оси абсцисс (ось *x*) — последовательность от 0 до 9, оси ординат (ось *y*) — случайные числа в диапазоне от 0 до 9 в количестве, равном числу элементов в списке *x*. Далее вызываем функцию `lineplot()` для отображения зависимости между созданными наборами. В нашем случае, мы воспользовались вариантом, когда данные передаются в функцию напрямую, в виде векторов. Результат работы программы представлен на рисунке 8.1.

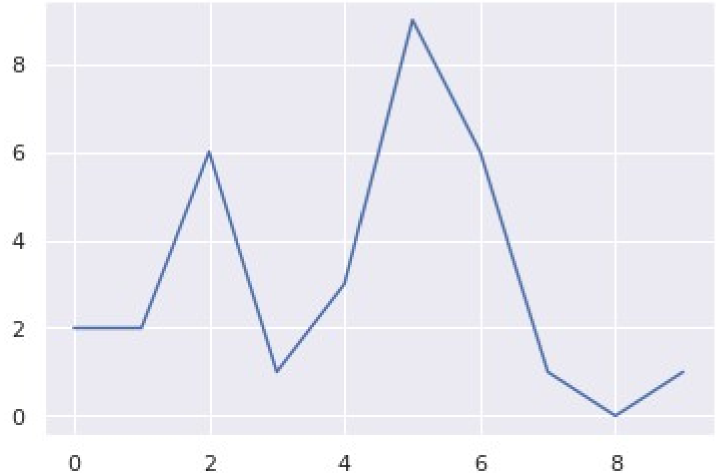


Рисунок 8.1 — Результат работы функции `lineplot()`

На практике данные для анализа чаще всего подготавливаются в виде структур `Series` или `DataFrame` библиотеки `pandas`. Создадим ещё несколько наборов данных:

```
np.random.seed(123)
sample = [i for i in range(10)]
y = np.random.randint(10, size=len(sample))
z = np.random.randint(4, size=len(sample))
data = [sample, y, z]
```

Объединим их в один `DataFrame`, это можно сделать так:

```
df = pd.DataFrame(data).transpose()
df.columns = ['sample', 'y_val', 'z_val']
```

либо следующим образом:

```
df = pd.DataFrame({'sample':sample, 'y_val': y, 'z_val': z})
```

Посмотрим на получившийся набор данных: содержимое структуры `df` представлено в таблице 8.2

Таблица 8.2 — Содержимое структуры df

sample	y_val	z_val
0	2	2
1	2	3
2	6	1
3	1	0
4	3	2
5	9	0
6	6	3
7	1	1
8	0	3
9	1	2

Отобразим зависимость `y_val` от `sample` из набора `df` с помощью функции `lineplot()`:

```
sns.lineplot(x='sample', y='y_val', data=df)
```

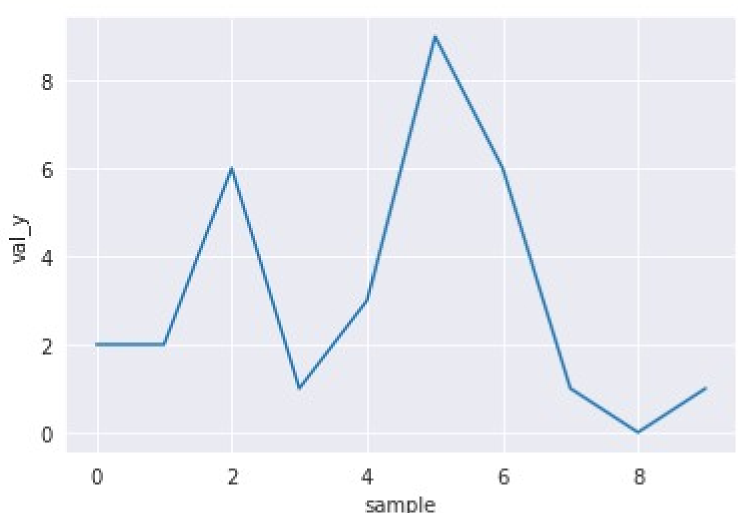


Рисунок 8.2 — Визуализация зависимости признака `y_val` от `sample`

Через параметр x задаётся имя признака из набора данных для оси абсцисс, через y — для оси ординат, data — это DataFrame, из которого данные будут извлечены.

Если необходимо отобразить все элементы из DataFrame набора, то можно его передать в lineplot() через параметр data, при этом не задавая x и y:

```
sns.lineplot(data=df)
```

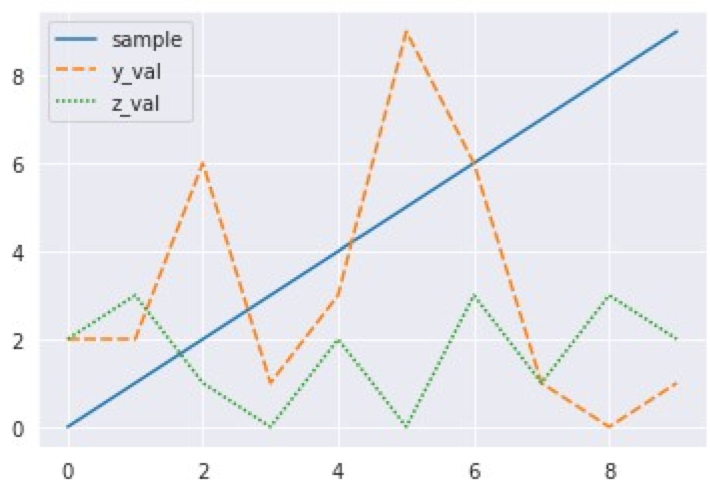


Рисунок 8.3 — Отображение всего набора данных df

8.2.2 Отображение математического ожидания и доверительных интервалов

Обратите внимание, что в предыдущих примерах, значения из набора данных, который мы указывали в качестве параметра для оси абсцисс (ось x), не повторялись. Построим график зависимости val_y от val_z. Для этих данных указанное выше свойство уже не будет выполняться: одному и тому же значению val_z (например, 0) будет соответствовать несколько значений val_y (в нашем случае — это 1 и 9).

Построим линейный график:

```
sns.lineplot(x='z_val', y='y_val', data=df)
```

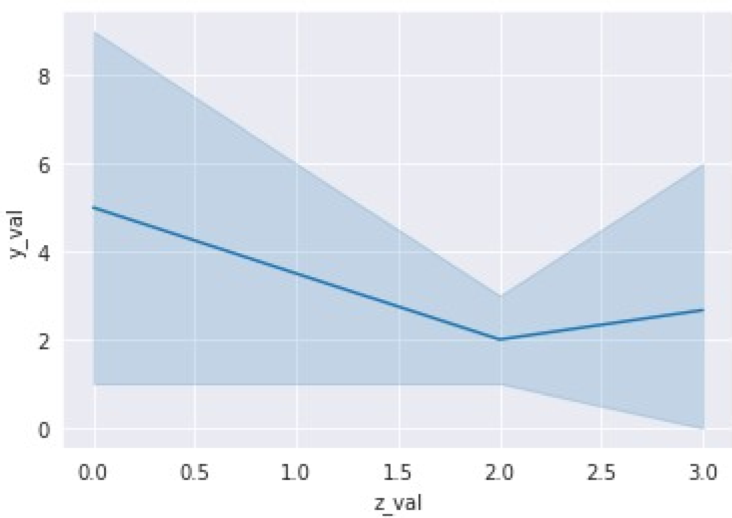


Рисунок 8.4 — Зависимость параметра `y_val` от `z_val`

В этом случае по оси ординат будут откладываться не значения параметра *y*, а его математическое ожидание. Например для `z_val=0`, `y_val=(1+9)/2=5`, что мы и видим на графике. Более светлая область вокруг линии — это 95% доверительный интервал, если вместо него необходимо отобразить стандартное отклонение, то следует присвоить параметру `ci` функции `lineplot()` значение `'sd'`:

```
sns.lineplot(x='z_val', y='y_val', ci='sd', data=df)
```

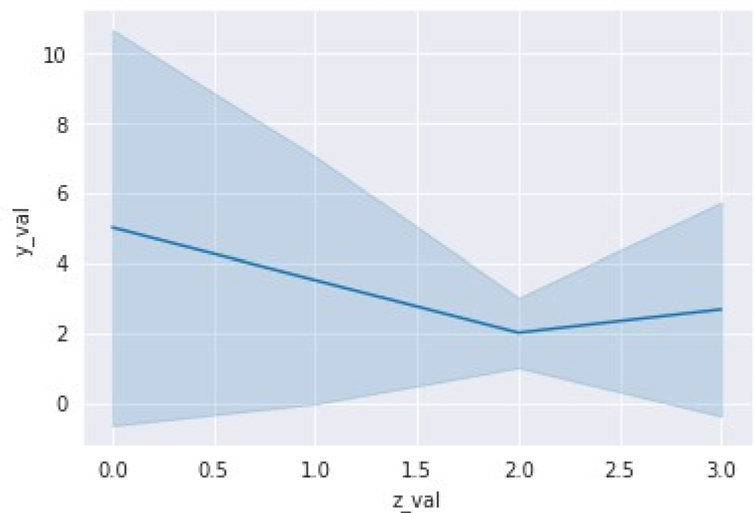


Рисунок 8.5 — График зависимости с отображением стандартного отклонения

Отображение доверительного интервала (или стандартного отклонения) можно отключить, передав в `ci` значение None:

```
sns.lineplot(x='z_val', y='y_val', ci=None, data=df)
```

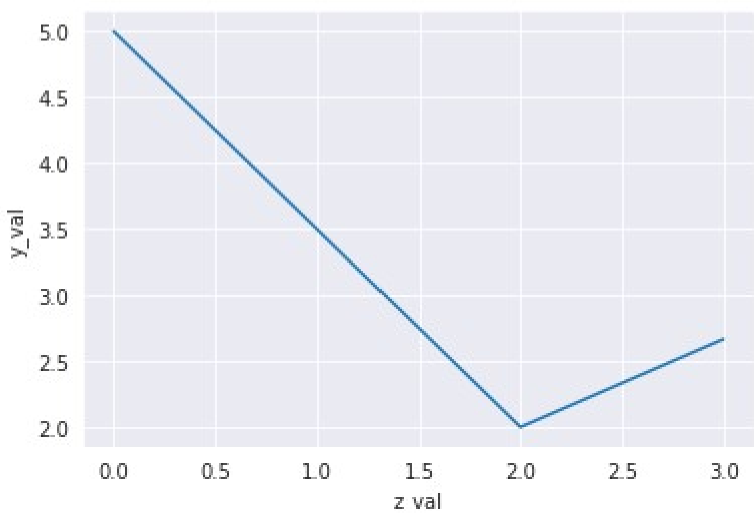


Рисунок 8.6 — Отключение отображения доверительного интервала

В главе шеской "*Быстрый старт*", мы приводили пример, иллюстрирующий зависимость пассажиропотока от года из набора данных *flights*, приведём его ещё раз:

```
flights = sns.load_dataset("flights")
sns.lineplot(x='year', y='passengers', data=flights)
```

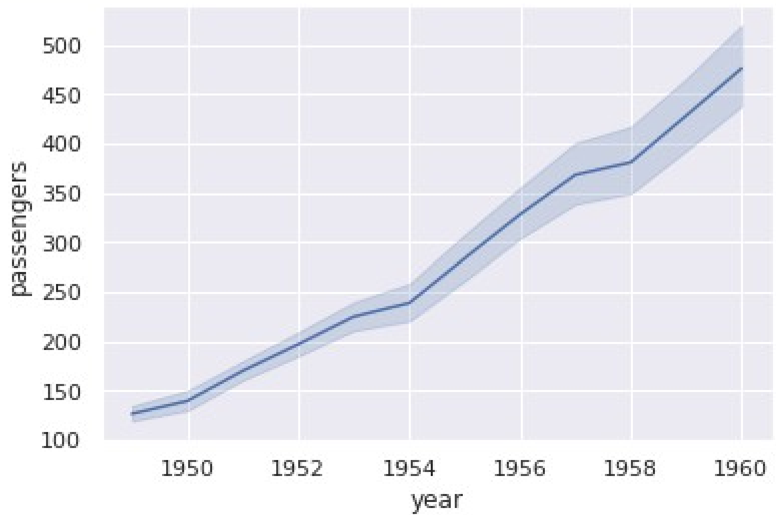


Рисунок 8.7 — Зависимость пассажиропотока от года из набора *flights*

Для управления видом отображения доверительного интервала (сплошная заливка, метки) используется параметр err_style, который может принимать значения 'band', 'bars' или None. Значение 'band' определяет сплошную заливку, как это представлено на рисунке выше. Значение 'bars' отображает доверительный интервал в виде отрезков:

```
sns.lineplot(x='year', y='passengers', err_style='bars', data=flights)
```

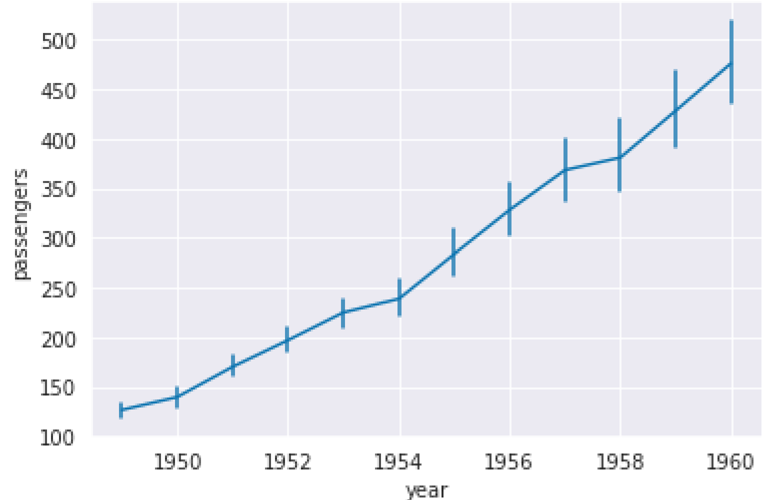


Рисунок 8.8 — Отображение доверительного интервала в виде отрезков

Если установить для параметра err_style значение None, то доверительный интервал отображён не будет:

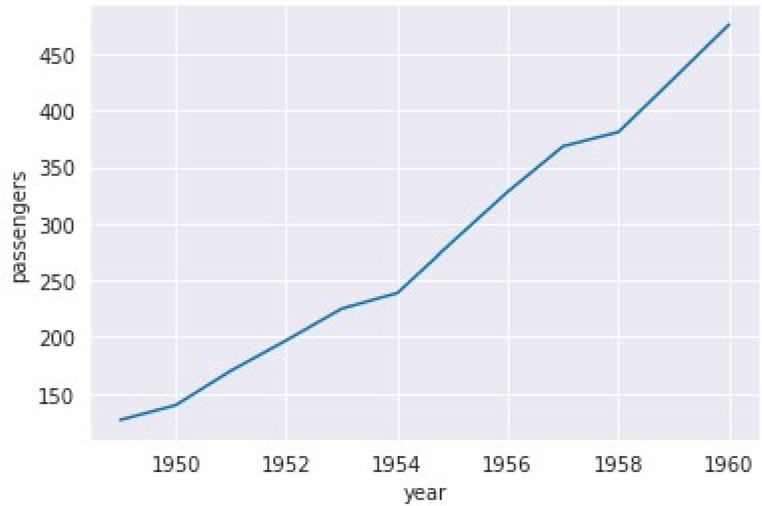


Рисунок 8.9 — Отключение отображение доверительного интервала с помощью параметра err_style

8.2.3 Повышение информативности графика

Функция `lineplot()` предоставляет три параметра, используя которые можно повысить информативность графика, связав дополнительные признаки набора с такими визуальными аспектами как цвет, стиль линии и размер. За это отвечают параметры: `hue` (цвет), `style` (стиль линии) и `size` (размер). Рассмотрим работу с ними на примерах.

Загрузим необходимый набор библиотек и настроим стиль:

```
import seaborn as sns
import pandas as pd
sns.set_style("darkgrid")
```

Загрузим набор данных для работы:

```
df = sns.load_dataset("mpg")
```

Отобразим зависимость мощности автомобиля от года выпуска:

```
sns.lineplot(x='model_year', y='horsepower', data=df)
```

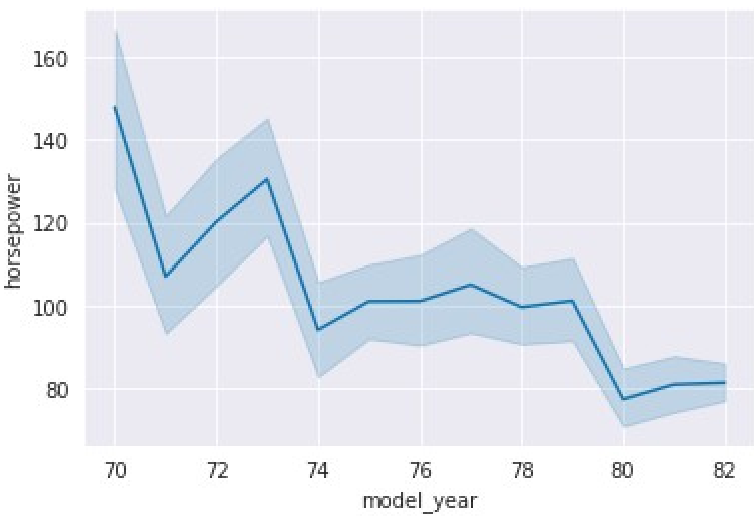


Рисунок 8.10 — График зависимости мощности автомобиля от года выпуска

8.2.3.1 Настройка цветовой схемы

Если стоит задача изучить зависимость мощности автомобиля от года выпуска отдельно для стран США и Японии, то можно воспользоваться параметром hue для выделения стран цветом:

```
sns.lineplot(x='model_year', y='horsepower', hue='origin',
data=df[df['origin'] != 'europe'])
```

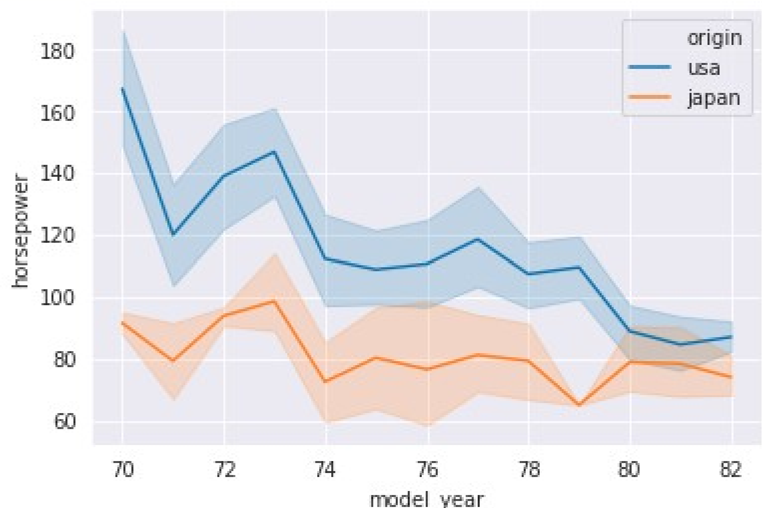


Рисунок 8.11 — Использование параметра hue для разделения данных по странам

Для дополнительной настройки цветового разделения можно использовать следующие параметры функции `lineplot()`: `palette`, `hue_order`, `hue_norm`.

Изменим цвет и зададим порядок:

```
df_usa_jp = df[df['origin'] != 'europe']
sns.lineplot(x='model_year', y='horsepower', hue='origin',
palette={'usa':'r', 'japan':'b'}, hue_order=['japan', 'usa'],
data=df_usa_jp)
```

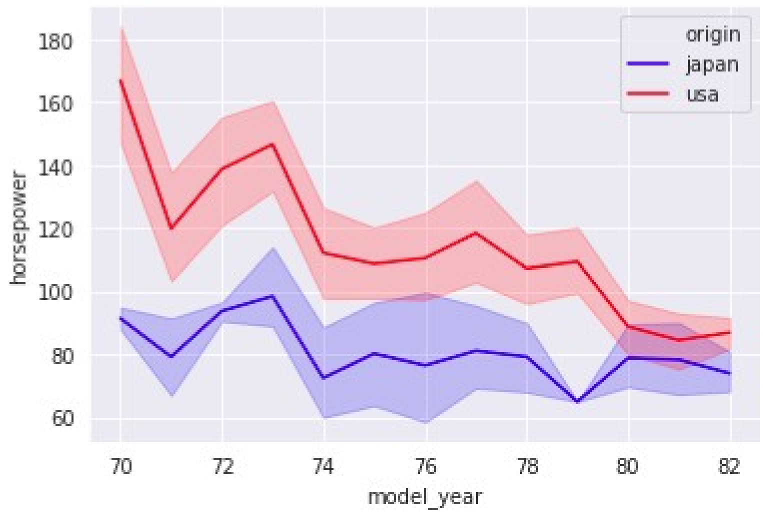


Рисунок 8.12 — Пример использования параметров `palette` и `hue_order`

8.2.3.2 Настройка стиля

Для сегментации данных с помощью стиля линии используется параметр `style`, через него задаётся признак, по которому будет производиться разделение:

```
sns.lineplot(x='model_year', y='horsepower', style='origin',
data=df_usa_jp)
```

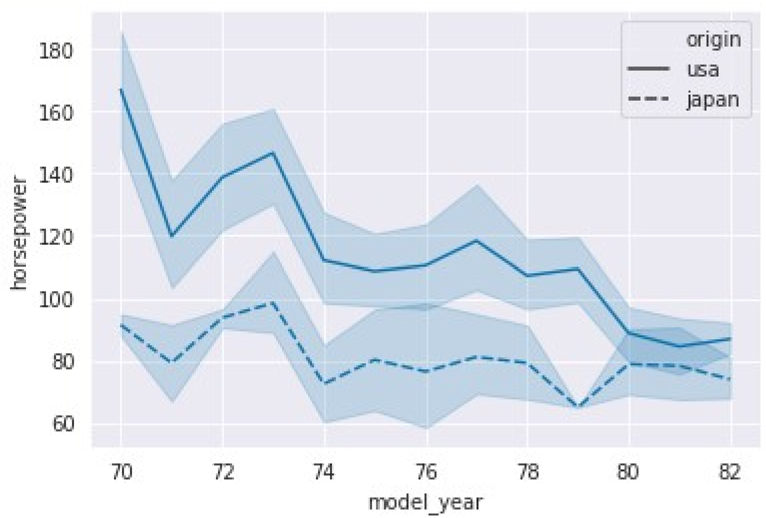


Рисунок 8.13 — Использование параметра `style` для разделения данных по странам

Настройку стиля линии можно произвести с помощью следующих параметров: `dashes`, `markers`, `style_order`. Описание для двух последних переведено в разделе "*8.1.2 Параметры для повышения информативности графиков*", информация по `dashes` представлена ниже:

- `dashes`: `bool`, `list`, `dict`, `optional`
    - Определяет тип штриховки. Если параметр равен `False`, то будет использована сплошная линия, если `True`, то типы штриховки по умолчанию. Тип применяемой штриховки для конкретных значений из набора данных, переданного в параметр `style`, можно определить самостоятельно, для этого нужно задать соответствие значений параметра и кодов штриховки. Штриховка задаётся либо *tuple*'ами следующего вида: `(длина сегмента, размер зазора)`, либо пустой строкой, что определяет сплошную линию.

Рассмотрим работу с параметром `dashes` на примере:

```
sns.lineplot(x='model_year', y='horsepower', style='origin',
dashes={'usa': (2, 2), 'japan': (5, 2)}, data=df_usa_jp)
```

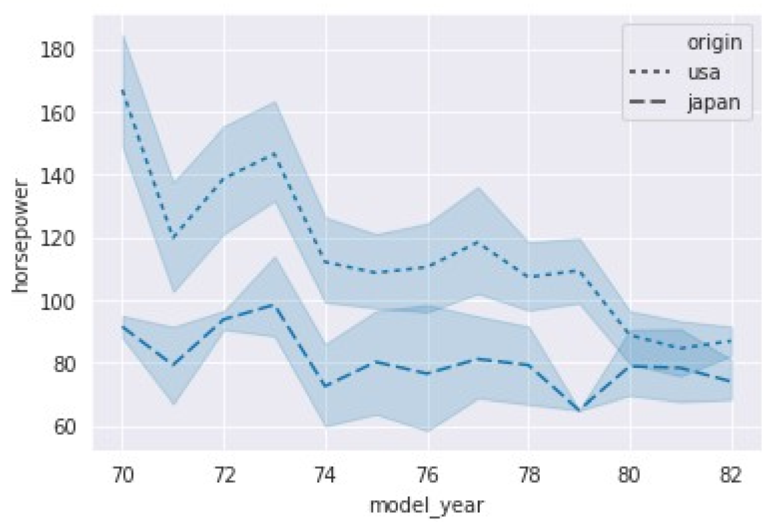


Рисунок 8.14 — Демонстрация работы с параметром dashes

Добавим маркеры:

```
sns.lineplot(x='model_year', y='horsepower', style='origin',
dashes=False, markers={'usa': '^', 'japan': 'o'}, data=df_usa_jp)
```

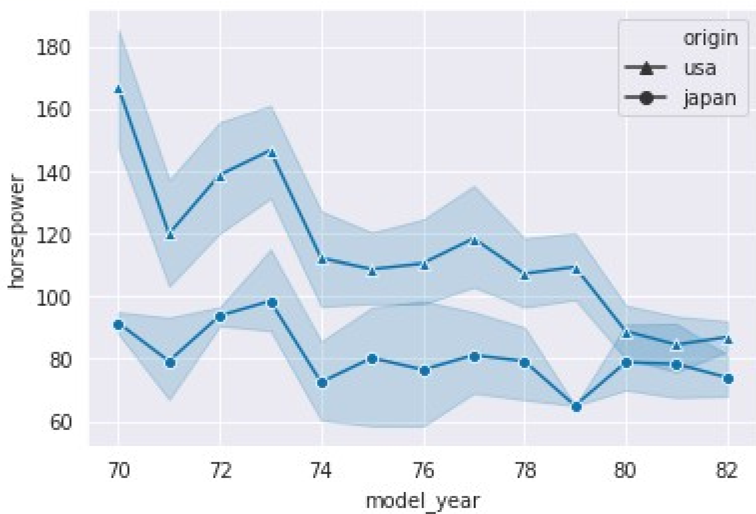


Рисунок 8.15 — Демонстрация работы с параметром `markers`

8.2.3.3 Настройка толщины линии

Следующим способом сегментации данных является задание толщины линии, за это отвечает параметр `size`. Через него передаётся набор данных, который будет использован для разделения основного набора на линии различной толщины.

Для демонстрации произведём модификацию набора *mpg*, с которым мы работали в предыдущих разделах:

```
fn_filter = lambda x: True if x in [4, 8] else False
fn_mod = lambda x: {4: 'four', 8: 'eight'}[x]
df_mod1 = df[df['cylinders'].map(fn_filter)].copy()
df_mod1['cylinders'] = df_mod1['cylinders'].map(fn_mod)
```

Отобразим на графике разной толщиной линии автомобили с восемью и четырёхцилиндровым двигателями:

```
sns.lineplot(x='model_year',      y='horsepower',      size='cylinders',
data=df_mod1)
```

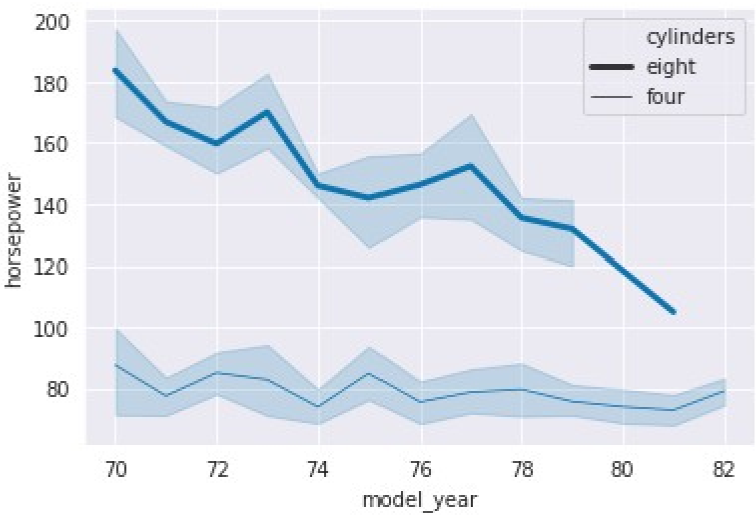


Рисунок 8.16 — Демонстрация использования параметра `size` для разделения данных по количеству цилиндров

Дополнительно для работы с толщиной линии можно использовать параметры `size_order` и `size_norm` (см. раздел "*8.1.2 Параметры для повышения информативности графиков*").

Изменим порядок задания толщины линии:

```
sns.lineplot(x='model_year', y='horsepower', size='cylinders',
size_order=('four', 'eight'), data=df_mod1)
```

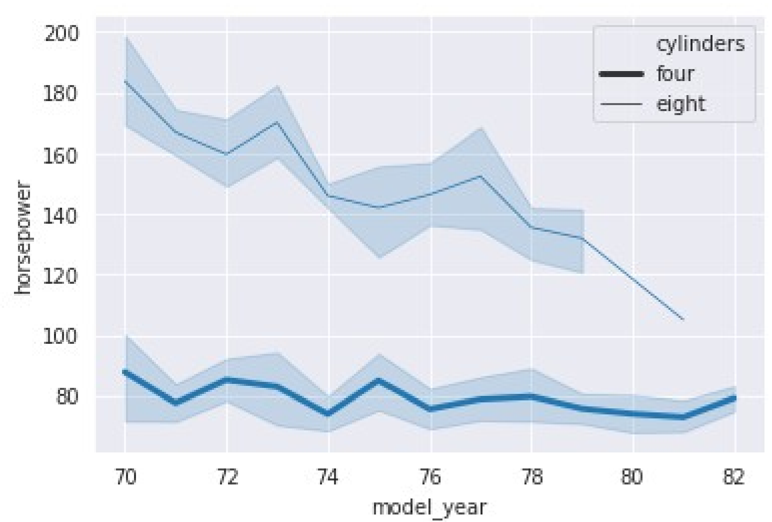


Рисунок 8.17 — Демонстрация работы с параметром `size_order`

8.2.4 Визуализация временных рядов

Библиотека *pandas* предоставляет мощные инструменты для работы с временными рядами. Воспользуемся ими для того, чтобы создать набор данных, у которых в качестве индексов будут временные метки с января 2018 по январь 2019 с периодом в один месяц, а значения для этих меток сгенерируем случайным образом:

```
date_index = pd.date_range(start='2018', freq='M', periods=12)
print(date_index)
```

```
DatetimeIndex(['2018-01-31', '2018-02-28', '2018-03-31', '2018-04-30',
               '2018-05-31', '2018-06-30', '2018-07-31', '2018-08-31',
               '2018-09-30', '2018-10-31', '2018-11-30', '2018-12-31'],
              dtype='datetime64[ns]', freq='M')
```

Набор данных для работы:

```
np.random.seed(123)
data_set = np.random.randint(5, size=len(date_index))
```

Построим структуру `DataFrame`:

```
df = pd.DataFrame(data=data_set, index=date_index, columns=['value'])
```

Первые пять элементов структуры:

```
df.head()
```

	value
2018-01-31	2
2018-02-28	4
2018-03-31	2
2018-04-30	1
2018-05-31	3

Представим данные из df в виде линейного графика с помощью *seaborn*:

```
sns.lineplot(data=df)
```

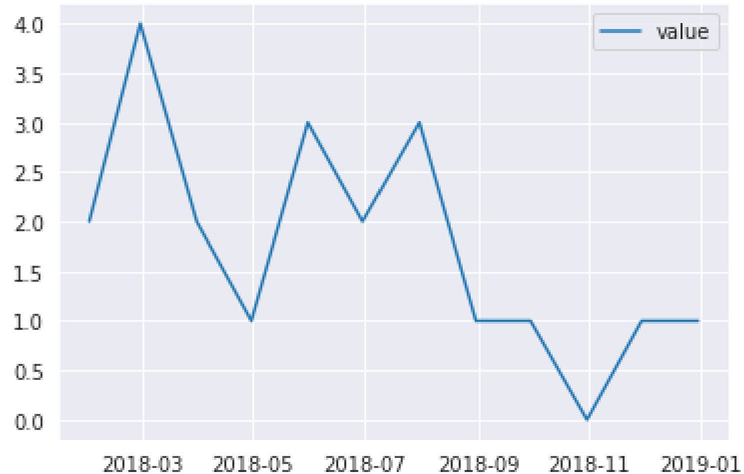


Рисунок 8.18 — Визуализация временного ряда

Обратите внимание, что индексы из набора df были использованы в качестве данных для оси *x*.

8.3 Диаграмма рассеяния. Функция scatterplot()

Диаграмма рассеяния (*scatterplot*) является неотъемлемым инструментом аналитика при работе с данными наряду с линейными графиками, столбчатыми диаграммами и т.д. Для построения такой диаграммы *seaborn* предоставляет функцию scatterplot().

8.3.1 Знакомство с функцией scatterplot()

Перед началом работы импортируйте набор библиотек, указанный в разделе "6*.2 Быстрый старт*". Зададим стиль darkgrid:

```
sns.set_style("darkgrid")
```

Загрузим набор данных *iris*:

```
df = sns.load_dataset("iris")
```

Отобразим зависимость параметра `sepal_length` от `petal_length` в виде точечного графика:

```
sns.scatterplot(x='sepal_length', y='petal_length', data=df)
```

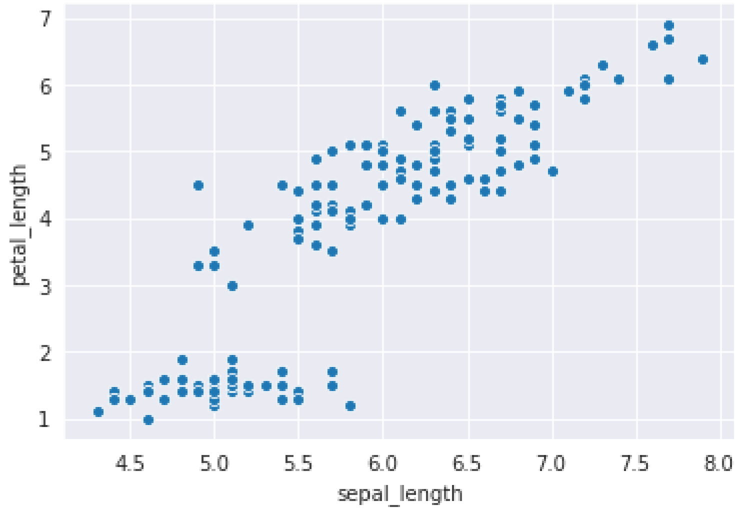


Рисунок 8.19 — Демонстрация работы функции `scatterplot()`

8.3.2 Повышение информативности графика *scatterplot*

Для повышения информативности точечного графика можно использовать параметры, аналогичные тем, что были рассмотрены нами при работе с `lineplot()` – это hue для задания оттенка (цвета), `size` – размера маркера, `style` – стиля маркера.

Для экспериментов воспользуемся уже известным нам набором данных *mpg*:

```
mpg = sns.load_dataset("mpg")
```

Построим точечный график зависимости дальности пробега (*displacement*) автомобиля от расхода топлива (*mpg*):

```
sns.scatterplot(x='mpg', y='displacement', data=mpg )
```

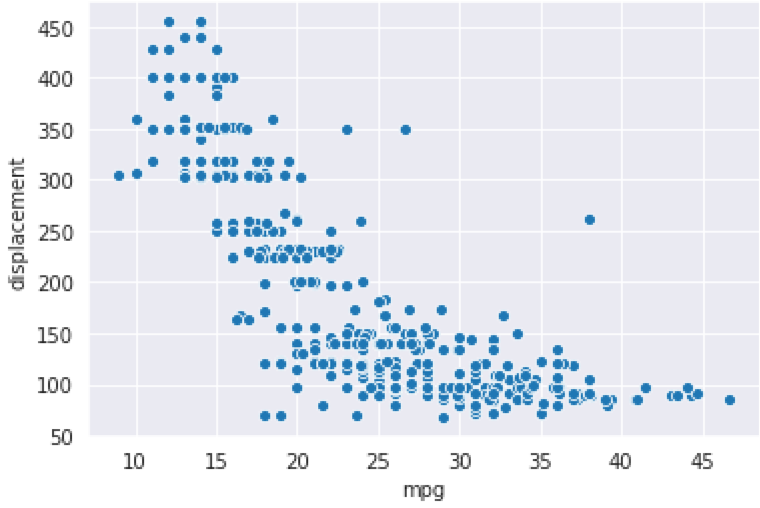


Рисунок 8.20 — Зависимость параметра *displacement* от *mpg*

8.3.2.1 Настройка цветовой схемы

Выделим цветом страну производителя:

```
sns.scatterplot(x='mpg', y='displacement', hue='origin', data=mpg )
```

Рисунок 8.21 — Использование параметра hue для разделения данных по странам

За дополнительную настройку цветовой схемы отвечают параметры `palette`, `hue_order` и `hue_norm`, их назначение и порядок использования аналогичен одноимённым параметрам функции `lineplot()` (см. раздел

"*8.2.3.1 Настройка цветовой схемы линейного графика*"). Для демонстрации работы с ними приведём несколько примеров. Изменим цветовую палитру:

```
sns.scatterplot(x='mpg', y='displacement', hue='origin',
palette='plasma', data=mpg)
```

Рисунок 8.22 — Пример использование цветовой палитры `plasma`

Зададим свой набор цветов:

```
cmap={'usa':'y', 'japan':'g', 'europe': 'r'}
sns.scatterplot(x='mpg', y='displacement', hue='origin', palette=cmap,
data=mpg)
```

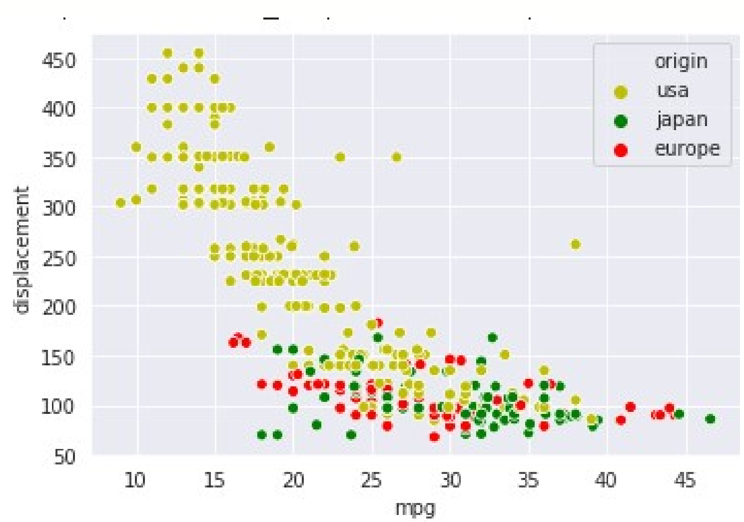


Рисунок 8.23 — Пример работы с предварительно подготовленной цветовой схемой

Изменим порядок применения цветов:

```
order=['japan', 'europe', 'usa']
sns.scatterplot(x='mpg', y='displacement', hue='origin', hue_order=order,
data=mpg)
```

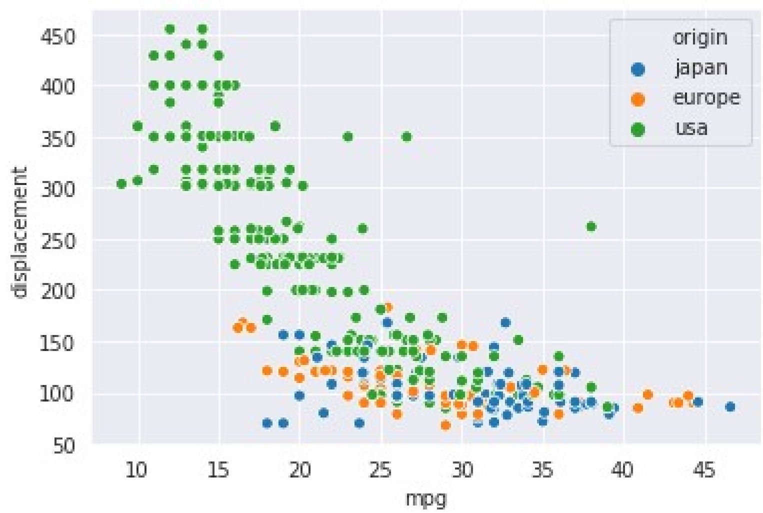


Рисунок 8.24 — Изменение порядка применения цветов

Обратите внимание на легенду, в отличие от предыдущих примеров перечисление стран теперь идёт в порядке, указанном в переменной `order`.

8.3.2.2 Настройка стиля маркеров

За стиль маркеров в нашем примере будет отвечать признак количество цилиндров (*cylinders*):

```
sns.scatterplot(x='mpg', y='displacement', hue='origin',
style='cylinders', data=mpg)
plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
```

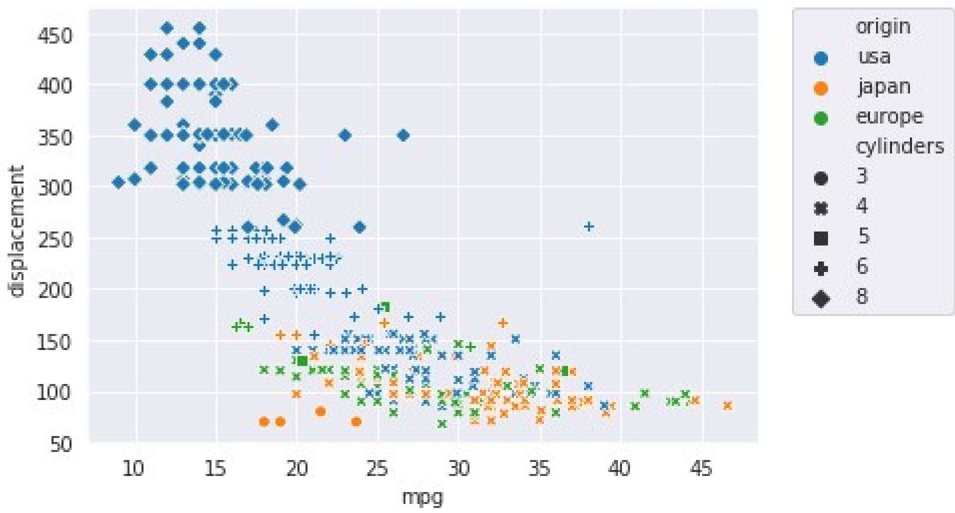


Рисунок 8.25 — Демонстрация работы с параметром `style`

Дополнительно нам пришлось вынести легенду за поле графика, т.к. она стала занимать уже значительное пространство.

Для сравнения приведём пример использования параметра `style` для набора *iris*:

```
sns.scatterplot(x='sepal_length', y='petal_length', style='species',
data=iris)
```

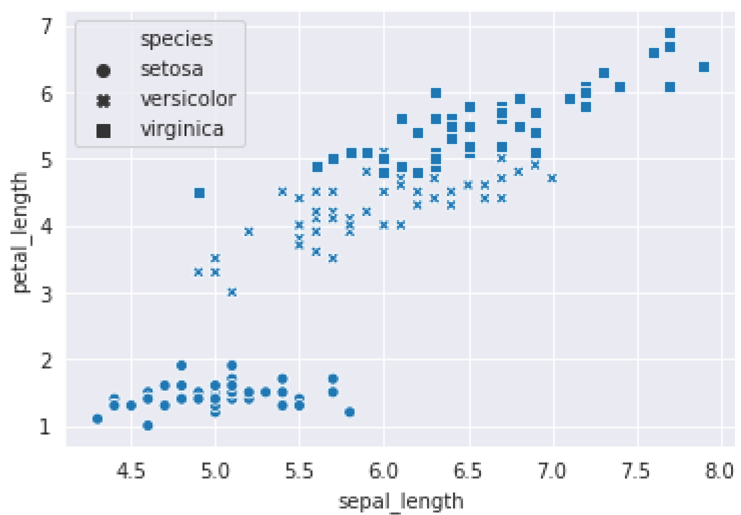


Рисунок 8.26 — Демонстрация работы с параметром `style` для набора данных *iris*

Так как данные в этом наборе визуально разделимы, то даже без дополнительной сегментации по цвету видны различия между объектами.

Для дополнительной настройки стиля маркеров можно воспользоваться параметрами `markers` и `style_order`, назначение которых совпадает с одноимёнными для функции `lineplot()`. Воспользуемся ими для более детальной настройки изображения: изменим стиль маркеров и порядок представления признаков:

```
mrks={'virginica':'o', 'setosa':'D', 'versicolor':'X'}
order = ['virginica', 'setosa', 'versicolor']
sns.scatterplot(x='sepal_length', y='petal_length', style='species',
data=iris, style_order=order, markers=mrks)
```

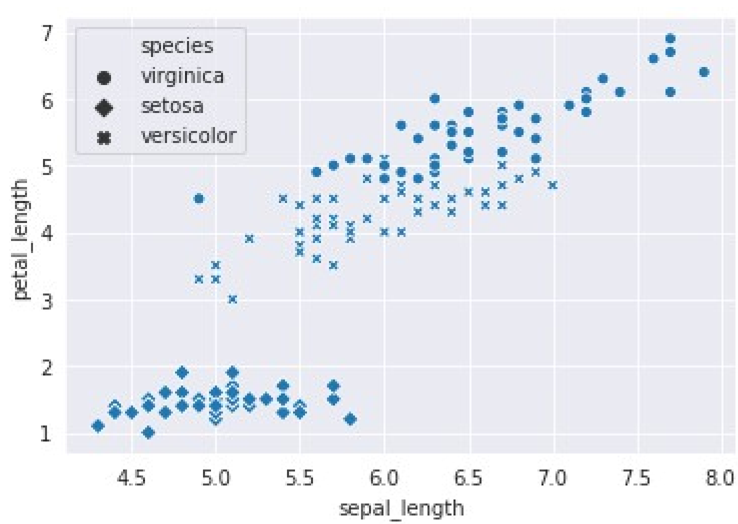


Рисунок 8.27 — Демонстрация работы с параметрами `markers` и `style_order`

8.3.2.3 Настройка размера маркера

Для демонстрации работы с параметром `size` вернёмся к набору данных *mpg,* будем использовать вес автомобиля (признак *weigh*) для управления размером маркеров:

```
sns.scatterplot(x='mpg', y='displacement', hue='origin', size='weight',
data=mpg)
```

Рисунок 8.28 — Демонстрация работы с параметром size

Параметры size_order и size_norm, отвечают за порядок и нормализацию при задании размера маркера, sizes определяет размер маркеров:

```
sns.scatterplot(x='mpg', y='displacement', hue='origin', size='weight',
sizes=(10, 150), data=mpg)
```

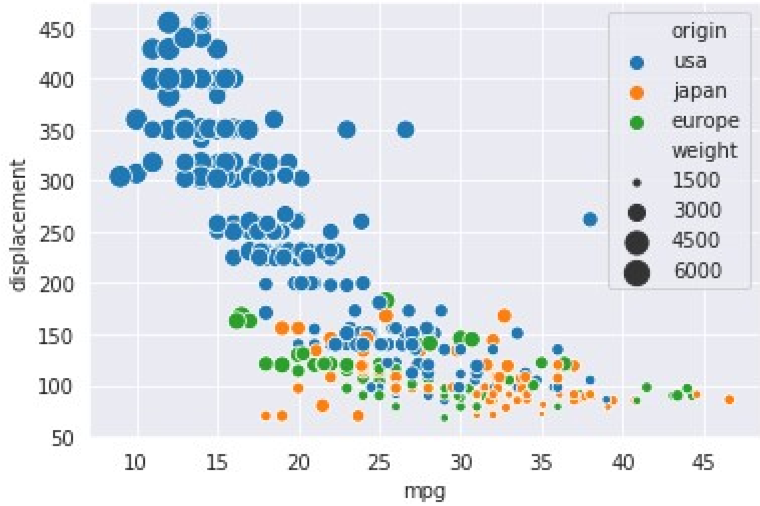


Рисунок 8.29 — Демонстрация работы с параметром sizes

Воспользуемся всеми тремя инструментами: hue, size и style одновременно:

```
sns.scatterplot(x='mpg', y='displacement', hue='origin', size='weight',
style='cylinders', data=mpg)
plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
```

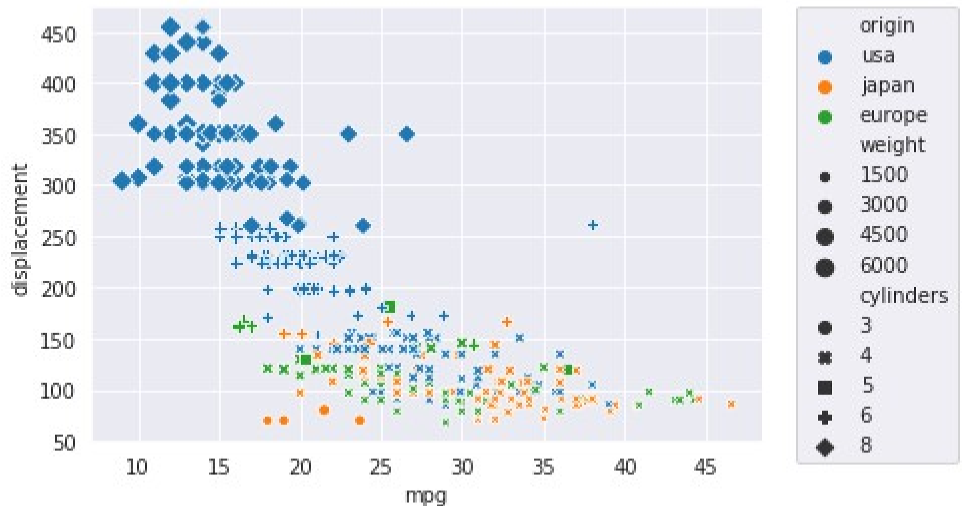


Рисунок 8.30 — Демонстрация одновременного применения параметров `hue`, `size` и `style`

8.4 Настройка внешнего вида элементов поля графика

Рассмотрим несколько полезных инструментов для настройки элементов поля линейного графика, а именно: легенды и подписей осей. Более подробно про эти и другие инструменты можете прочитать в главе 7 "*Настройка внешнего вида графиков*".

8.4.1 Легенда

За отображение легенды отвечает параметр `legend`, он может принимать следующие значения: `'brief'` — будет использован

сокращённый набор данных, `'full'` — будет использован полный набор данных, `False` — легенда отображаться не будет.

Рассмотрим пример: в наборе данных *mgp* признак *cylinders* принимает значения из множества `{3, 4, 5, 6, 8}`:

```
df = sns.load_dataset("mpg")
set(df["cylinders"])
{3, 4, 5, 6, 8}
```

Отобразим график с цветовым разделением по этому признаку с сжатым вариантом представления легенды:

```
sns.lineplot(x='model_year', y='horsepower', hue='cylinders', data=df,
legend='brief')
```

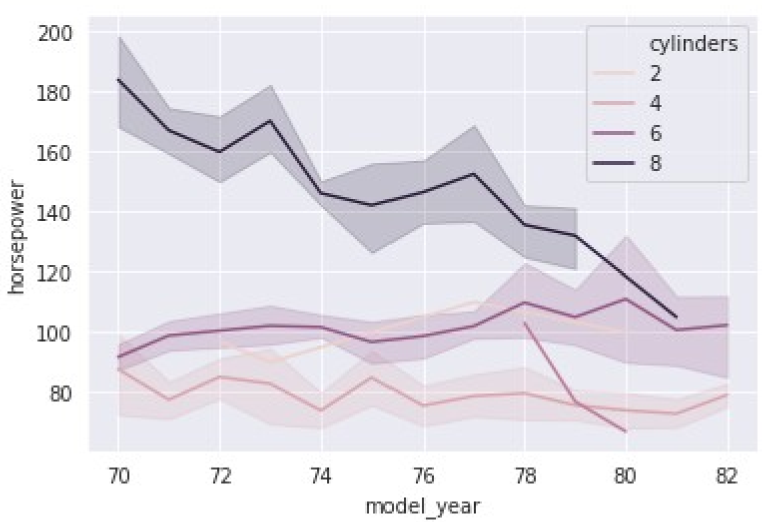


Рисунок 8.31 — Представление сокращённого набора значений признака в легенде

Как вы можете видеть, в легенде отображён диапазон значений от 2-х до 8-ми. Теперь построим тот же график, но параметру `legend` присвоим значение `'full'`:

```
sns.lineplot(x='model_year', y='horsepower', hue='cylinders', data=df,
legend='full')
```

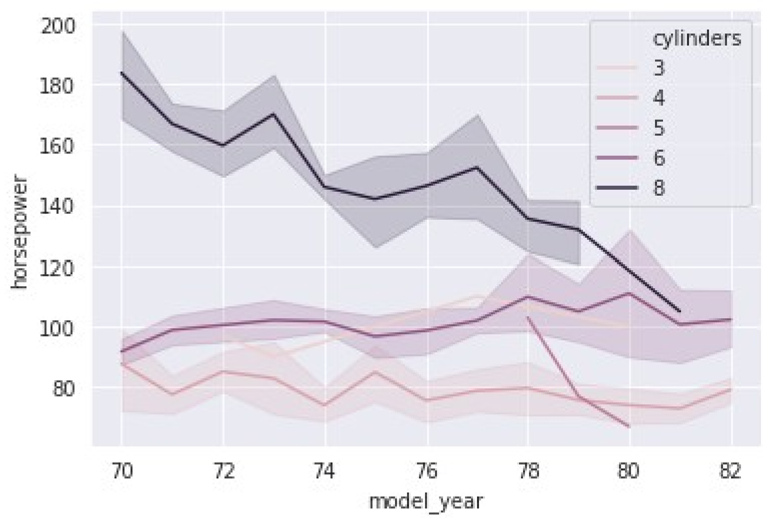


Рисунок 8.32 — Представление полного набора значений признака в легенде

Теперь в легенде перечислены все значения признака *cylinders*. Уберём легенду с поля графика, для этого параметру `legend` присвоим значение `False`:

```
sns.lineplot(x='model_year', y='horsepower', hue='cylinders', data=df,
legend=False)
```

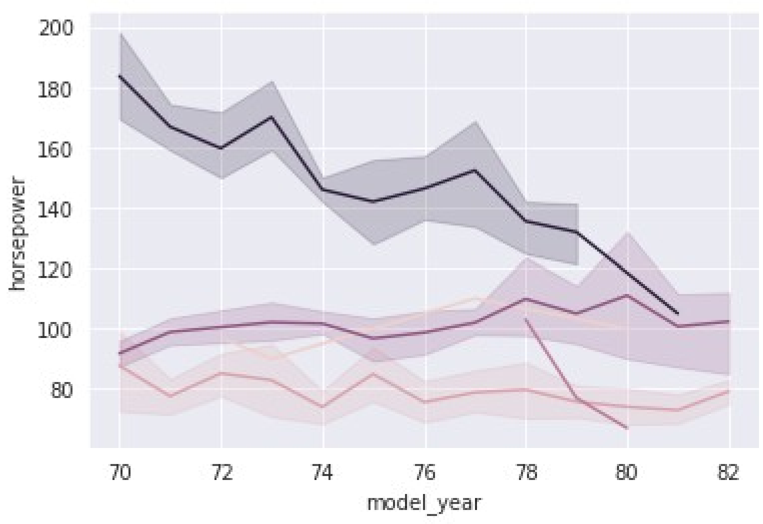


Рисунок 8.33 — Поле графика без легенды

8.4.2 Подписи осей

Библиотека *seaborn* построена на базе *Matplotlib*, поэтому часть настроек можно производить через неё. Это касается задания подписей для осей координат графика.

Обратимся к примеру из раздела "*8.2.4 Визуализация временных рядов*":

```
np.random.seed(123)
date_index = pd.date_range(start='2018', freq='M', periods=12)
data_set = np.random.randint(5, size=len(date_index))
df = pd.DataFrame(data=data_set, index=date_index, columns=['value'])
```

Для задания подписей воспользуемся функциями `xlabel()` и `ylabel()` из `matplotlib.pyplot`:

```
import matplotlib.pyplot as plt
sns.lineplot(data=df)
plt.xlabel("Event day", fontsize=14)
plt.ylabel("Count", fontsize=14)
```

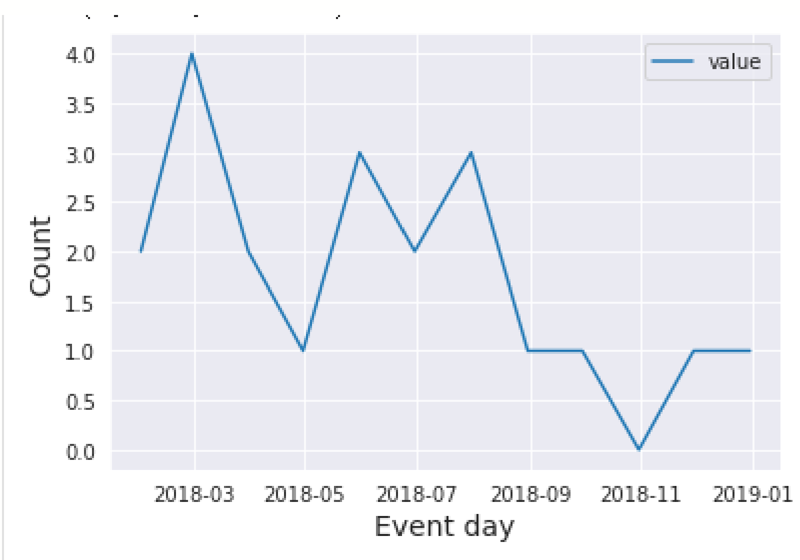


Рисунок 8.34 — Задание подписей для осей графика

8.4.3 Сортировка набора данных

По умолчанию, перед тем как отобразить график, *seaborn* производит сортировку набора данных, эту опцию можно отключить, задав параметру `sort` значение `False`.

Построим зависимость мощности двигателя от года выпуска автомобиля:

```
sns.lineplot(x='model_year', y='horsepower', data=df)
```

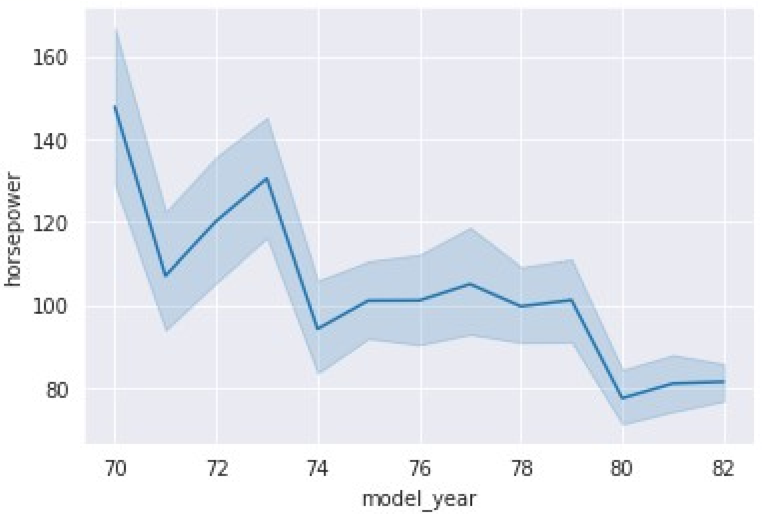


Рисунок 8.35 — График зависимости параметра *horsepower* от *model_year* до перемешивания набора данных

Перемешаем набор данных *mpg*:

```
df = sns.load_dataset('mpg')
df = df.sample(frac=1)
```

Снова построим график зависимости выбранных параметров:

```
sns.lineplot(x='model_year', y='horsepower', data=df)
```

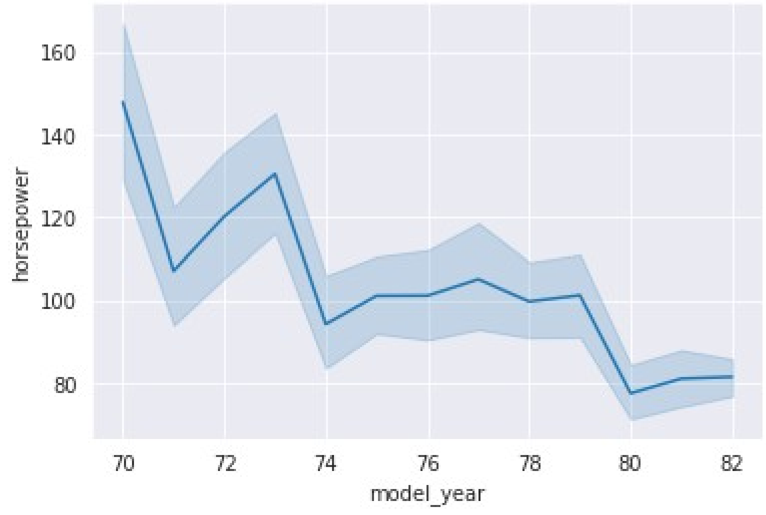


Рисунок 8.36 — График зависимости параметра *horsepower* от *model_year* после перемешивания набора данных

Как вы можете видеть ничего не поменялось в сравнении с результатом, который мы получали без перемешивания (см. рисунок 8.36). Отключим опцию предварительной сортировки:

```
sns.lineplot(x='model_year', y='horsepower', data=df, sort=False)
```

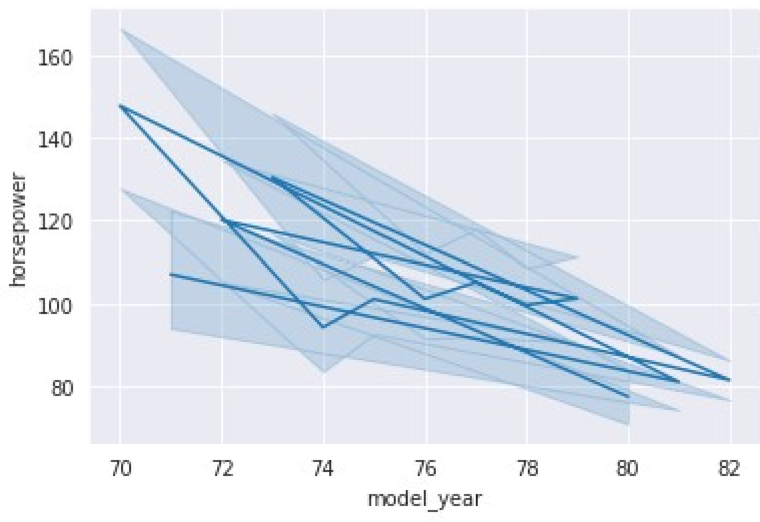


Рисунок 8.37 — График зависимости параметра *horsepower* от *model_year* с отключённой опцией сортировки

В этом случае график уже будет отличаться от приведённого выше варианта.

8.5 Визуализация отношений с настройкой подложки. Функция `relplot()`

Библиотека *seaborn* предоставляет ещё одну функцию для визуализации отношений в данных: `relplot()`. В отличие от `lineplot()` и `scatterplot()`, она предоставляет возможность настраивать не только внешний вид непосредственно самого графика, но и фигуру — подложку, на которой располагаются все графические компоненты.

Параметры, отвечающие за настройку внешнего вида, аналогичны приведённым для функций `lineplot()` и `scatterplot()`. Ниже приставлены аргументы, которые можно использовать для настройки фигуры:

- `row`, `col`: `str`
  - Имена признаков, по которым будет производиться разделение фигуры на строки и столбцы. Могут использоваться только категориальные признаки.
- `col_wrap` : `int`, `optional`
  - Количество столбцов для объединения.
- `row_order`, `col_order`: список сторок, `optional`
  - Порядок строк и/или столбцов согласно перечисленным значениям признака.
- `kind`: `str`, `optional`
  - Тип отображаемого графика: `'line'` – для вывода линейного графика, `'scatter'` — точечного.
- `height`: `int`, `float`, `optional`
  - Высота поля графика в дюймах.
- `aspect` : `int`, `float`, `optional`
  - Соотношение сторон поля с графиком, ширина рассчитывается по формуле: `aspect * height`.

- `facet_kws`: `dict`, `optional`
    - Словарь с дополнительными аргументами для `FacetGrid` (класс для управления фигурой).

Рассмотрим работу с `relplot()` на примерах. Будем использовать стиль `whitegrid`:

```
sns.set_style('whitegrid')
```

Для начала поработаем с набором *iris*:

```
iris = sns.load_dataset('iris')
sns.relplot(x='sepal_length', y='petal_length', kind='scatter',
data=iris)
```

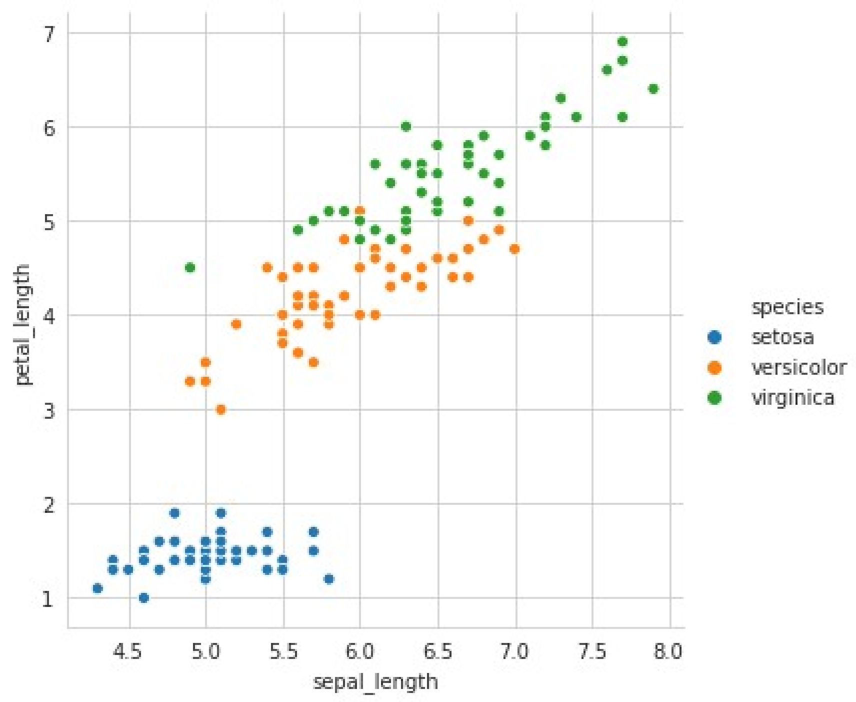


Рисунок 8.38 — Демонстрация работы функции `relplot()`

График практически не отличается от полученного нами в разделе "*8.3 Диаграмма рассеяния. Функция* `scatterplot()`".

Теперь построим диаграммы на трёх отдельных полях, в качестве разделяющего параметра будем использовать тип ириса (*species*):

```
sns.relplot(x='sepal_length', y='petal_length', hue='species',
kind='scatter', data=iris, col='species')
```

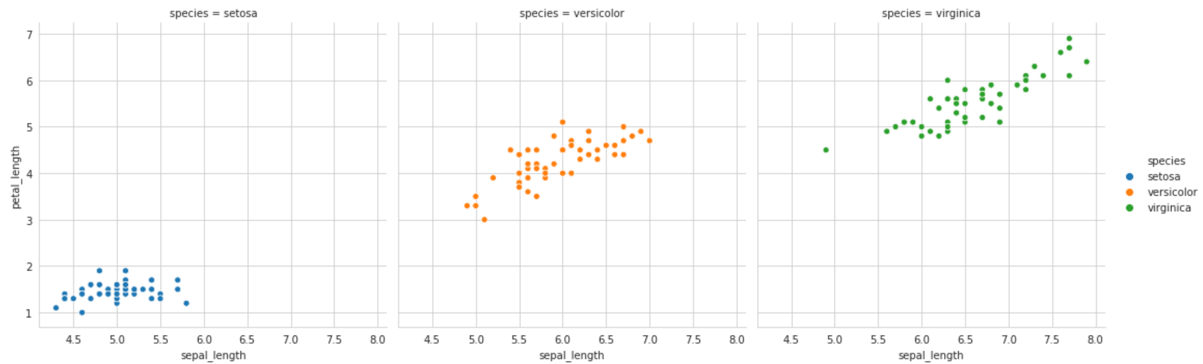


Рисунок 8.39 — Демонстрация работы с параметром `col` функции `relplot()`

Произведём разделение по строкам с помощью параметра `row` и зададим размеры полей через `height` и `aspect`:

```
sns.relplot(x='sepal_length', y='petal_length', hue='species',
kind='scatter', data=iris, row='species', height=3, aspect=3)
```

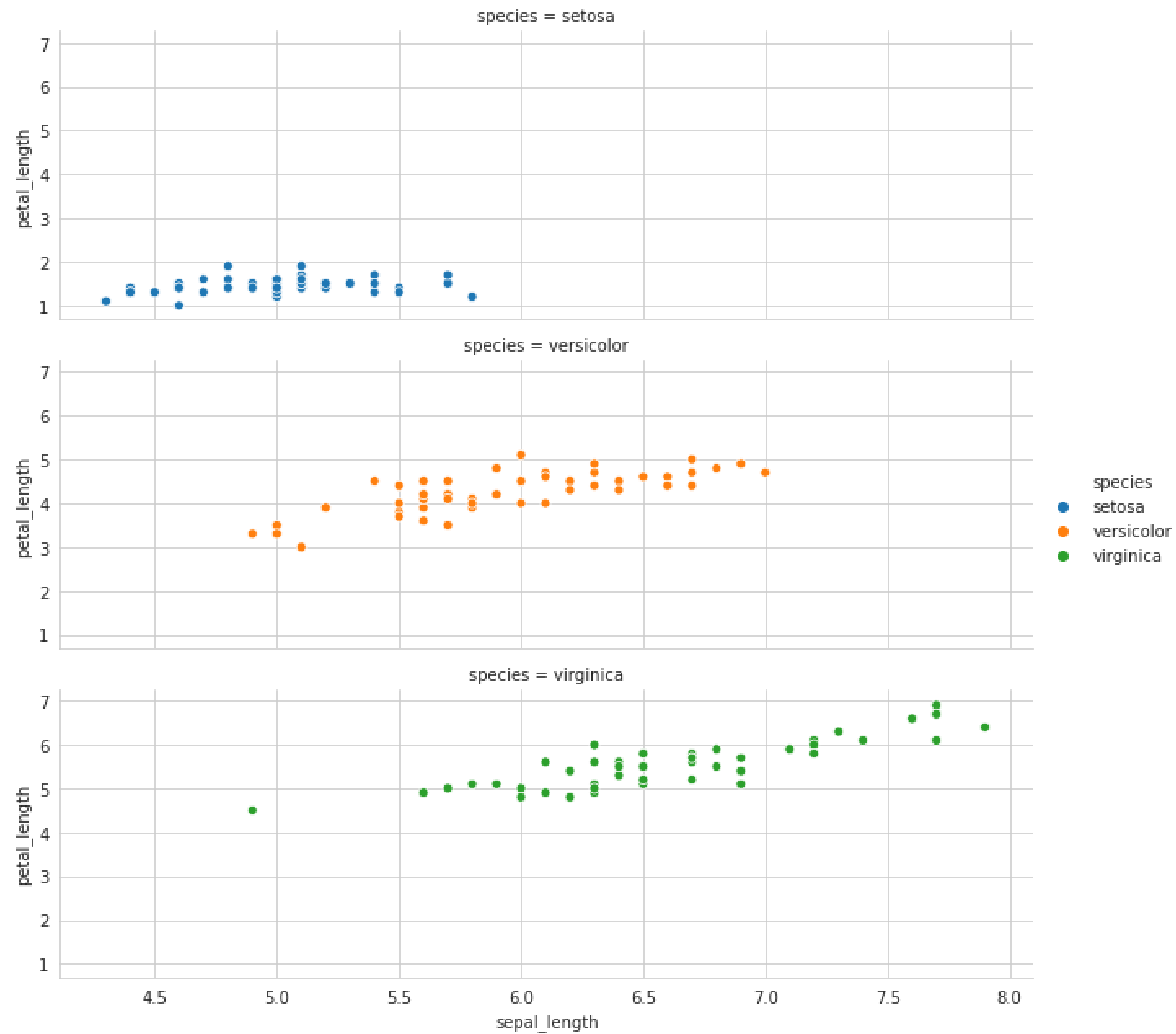


Рисунок 8.40 — Демонстрация работы с параметрами `row`, `height`, `aspect` функции `relplot()`

Глава 9. Визуализация категориальных данных

При анализе данных часто приходится работать с категориальными признаками, их особенность состоит в том, что они принимают значения из неупорядоченного множества. К этой группе можно отнести такие признаки как цвет (автомобиля, фрукта), форма (квадрат, круг, треугольник) и т.п. Важным является то, что для такого типа данных не вводится отношение порядка, т.е. нельзя применить операцию больше или меньше для их сравнения.

Есть более частный вид категориальных признаков — это порядковые признаки. Они по своей природе не являются числами, но для них уже можно определить операцию сравнения, примерами таких признаков могут быть тип автомобиля (легковой, грузовой), образование (бакалавр, магистр, кандидат наук, доктор наук).

Seaborn предоставляет инструменты для визуализации наборов категориальных данных в виде точечных диаграмм и их оценки, и средства представления распределений категориальных данных. Функция `catplot()` является общим интерфейсом для всех функций работы с категориальными данными с возможностью модифицировать фигуру (подложку), идейно она повторяет функцию `relplot()` из раздела "*8.5 Визуализация отношений с настройкой подложки. Функция* `relplot()`*"*. Список функций приведён в таблице 9.1.

Таблица 9.1 — Функции для визуализации категориальны данных

<table>
<tr><th>Имя функции</th><th>Описание</th></tr>
<tr><td>`catplot()`</td><td>Общий интерфейс для работы с категориальными данными с возможностью модификации подложки.</td></tr>
<tr><td>`stripplot()`</td><td rowspan="2">Инструменты для визуализации категориальных данных в виде точечных диаграмм.</td></tr>
<tr><td>`swarmplot()`</td></tr>
<tr><td>`boxplot()`</td><td rowspan="3">Инструменты для визуализации распределений категориальных данных.</td></tr>
<tr><td>`violinplot()`</td></tr>
<tr><td>`boxenplot()`</td></tr>
<tr><td>`pointplot()`</td><td rowspan="3">Инструменты для визуализации оценки категориальных данных.</td></tr>
<tr><td>`barplot()`</td></tr>
<tr><td>`countplot()`</td></tr>
</table>

9.1 Общие параметры функций

Перечисленные в таблице 9.1 функции имеют общий набор параметров для настройки внешнего вида диаграмм, их описание приведено далее в этом разделе.

9.1.1 Базовые параметры

- `x`, `y`: имена переменных из набора `data`, `optional`
  - Связывают ось *x* и *y* с конкретными признаками из набора данных, переданного через параметр `data`. Для функций отличных от `catplot()` допустимо передавать вектора с данными напрямую, для `catplot()` нет. Параметры могут иметь значение `None` в случае, если необходимо визуализировать весь набор `data`.

- `data`: `DataFrame`
    - Набор данных типа `pandas.DataFrame`, в котором столбцы — это имена признаков, строки — значения. Имена столбцов, данные из которых необходимо визуализировать, передаются через параметры *x* и *y*.

9.1.2 Параметры для повышения информативности графиков

Для повышения информативности графиков могут использованы параметры:

- `hue`: имя переменной из набора `data`, `optional`
    - Задаёт признак в наборе данных, который будет использован для цветового разделения данных. Визуально группы будут представлены в виде отдельных элементов, отличающихся цветом. Все функции за исключением `catplot()` позволяют передавать вектора с данными в параметр напрямую, для `catplot()` это недопустимо.
- `order`, `hue_order`: список строковых значений
    - Порядок отображения элементов (или задания цвета).
- `orient`: `'v' | 'h'`, `optional`
    - Ориентация графика, `'v'` – вертикальная, `'h'` – горизонтальная.
- `color`: *Matplotlib*-цвет[17], `optional`
    - Цвет для всех отображаемых элементов или зерно для градиента палитры.
- `palette`: имя палитры, `list` или `dict`
    - Палитра, которая будет использоваться для цветового разделения данных по значениями признака, указанного в `hue`.

17 Один из доступных способов задания цвета (см. раздел “*2.3.2 Цвет линии*”)

9.2 Визуализация категориальных данных в виде точечных диаграмм

К первой группе инструментов для визуализации категориальных данных относятся функции `stripplot()` и `swarmplot()`.

Данные функции имеют более широкий набор общих параметров по сравнению с приведённым в начале главы. Рассмотрим их более подробно:

- `size`: `float`, `optional`
  - Диаметр маркеров.
- `edgecolor`: *Matplotlib*-цвет[18], `'gray'`, `optional`
  - Цвет границы маркеров. Для значения `'gray'` яркость будет определяться в зависимости от используемой цветовой схемы.
- `linewidth` : `float`, `optional`
  - Ширина граничной линии.
- `dodge` : `bool`, `optional`
  - Если параметр равен `True` и используется `hue`, то на диаграмме данные будут представлены в виде визуально разделимых групп.

9.2.1 Функция `stripplot()`

Начнём наш обзор с функции `stripplot()`. Эта функция строит точечную диаграмму. С похожим инструментом — функцией `scatterplot()`, мы познакомились в главе "*8.3 Диаграмма рассеяния. Функция scatterplot()*".

18 Один из доступных способов задания цвета (см. раздел “*2.3.2 Цвет линии*”)

Загрузим набор данных *iris*:

```
iris = sns.load_dataset("iris")
```

Построим простую диаграмму распределения:

```
sns.stripplot(x='species', y='sepal_length', data=iris)
```

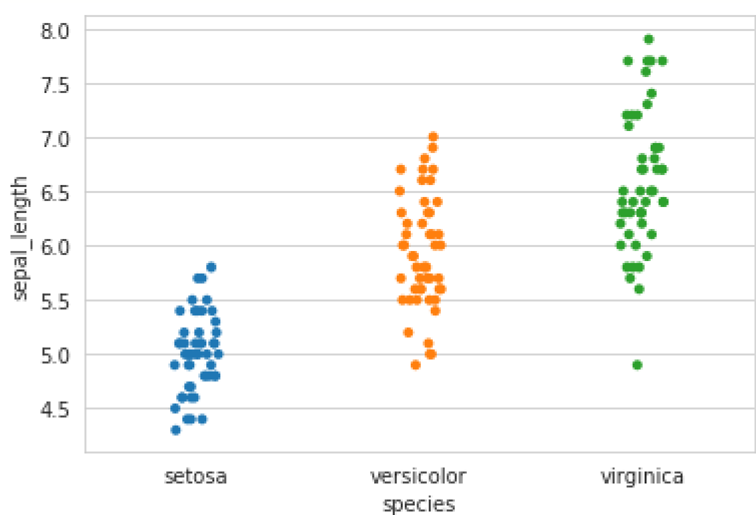


Рисунок 9.1 — Диаграмма, построенная с помощью функции `stripplot()`

Если необходимо построить точечную диаграмму для набора данных, элементы которого представляют собой самостоятельные массивы, то можно их передать напрямую через параметры x, y, не формируя предварительно `DataFrame`:

```
np.random.seed(321)
x_vals = np.random.randint(3, size=200)
y_vals = np.random.randn(1, len(x_vals))[0]
sns.stripplot(x=x_vals, y=y_vals)
```

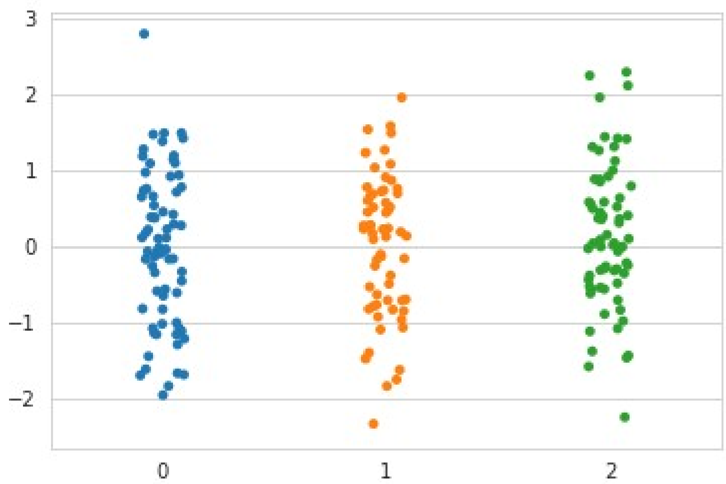


Рисунок 9.2 — Диаграмма, построенная с помощью функции `stripplot()` по данным, переданным напрямую через параметры x и y

Для повышения информативности диаграммы можно использовать параметры для настройки цветовой схемы, размера и цвета обводки маркеров.

Загрузим набор данных *tips*:

```
tips = sns.load_dataset("tips")
```

Выведем пять первых строк с помощью метода `head()`, результат выполнения функции приведён в таблице 9.2:

```
tips.head()
```

Таблица 9.2 — Первые пять строк набора данных *tips*

total_bill	*tip*	*sex*	*smoker*	*day*	*time*	*size*
16.99	1.01	*Female*	*No*	*Sun*	*Dinner*	2
10.34	1.66	*Male*	*No*	*Sun*	*Dinner*	3
21.01	3.50	*Male*	*No*	*Sun*	*Dinner*	3
23.68	3.31	*Male*	*No*	*Sun*	*Dinner*	2
24.59	3.61	*Female*	*No*	*Sun*	*Dinner*	4

В этом наборе содержатся данные о размерах оставленных чаевых, разделённые по следующим признакам: *total_bill* — общий счёт (в долларах США), *tip* — размер чаевых (в долларах США), *sex* — пол, *smoker* — курит клиент или нет, *time* — время: обед или ланч, *size* — размер компании.

Построим точечный график зависимости размера чаевых (*tips*) от количества человек в отдыхающей компании (*size*):

```
sns.stripplot(x="size", y='tip', data=tips)
```

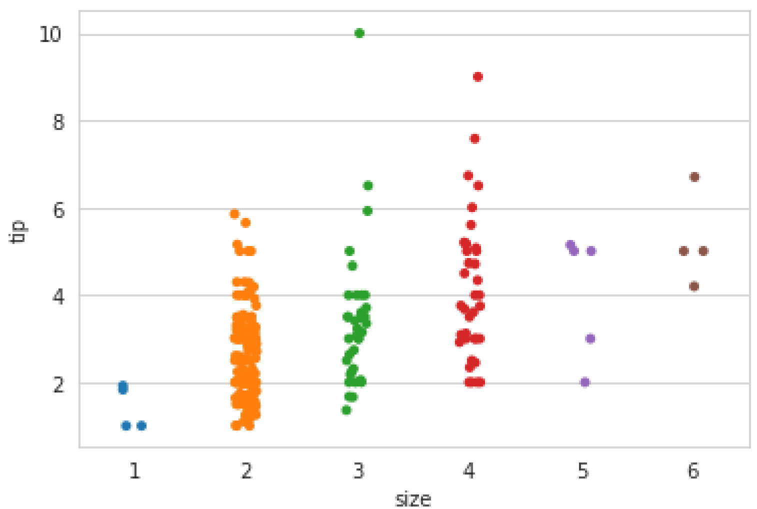


Рисунок 9.3 — Зависимость размера чаевых (*tips*) от количества человек в отдыхающей компании (*size*)

Как вы можете видеть, группы маркеров для разных значений `size` выделены разными цветами, если необходимо, чтобы цвет использовался один и тот же, то его можно задать через параметр `color`:

```
sns.stripplot(x="size", y='tip', color="g", data=tips)
```

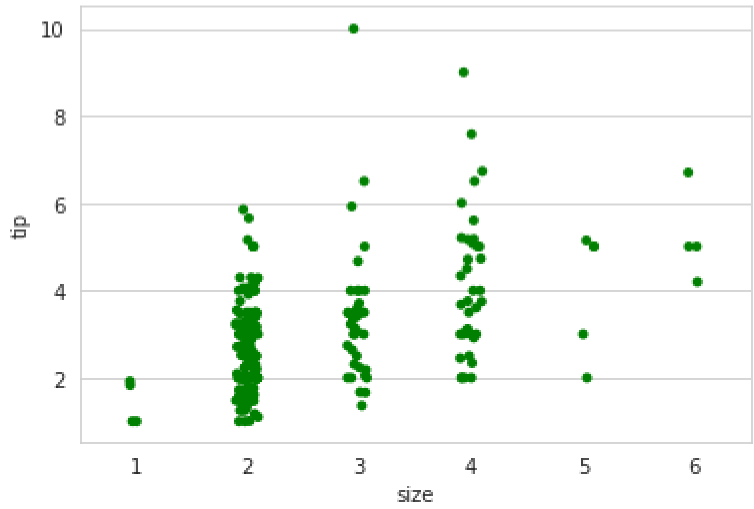


Рисунок 9.4 — Демонстрация работы с параметром color

Теперь выделим цветом пол людей, для этого присвоим параметру hue значение *sex*:

```
sns.stripplot(x="size", y='tip', hue="sex", data=tips)
```

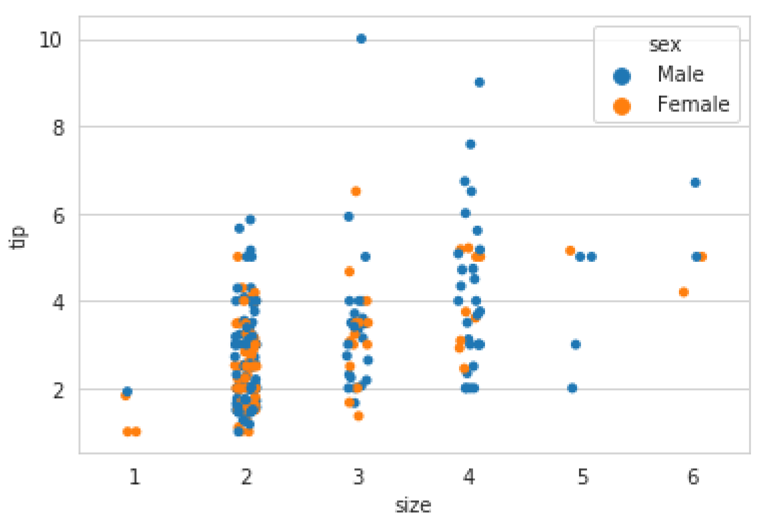


Рисунок 9.5 — Демонстрация использования параметра hue (пример 1)

Так как *'size'* — это вещественный признак, то корректировать для него порядок бессмысленно. Если бы мы построили диаграмму зависимости чаевых от пола клиента, тогда можно было бы задать желаемое представление.

По умолчанию мы получим следующий результат:

```
sns.stripplot(x="time", y='tip', hue='sex', data=tips)
```

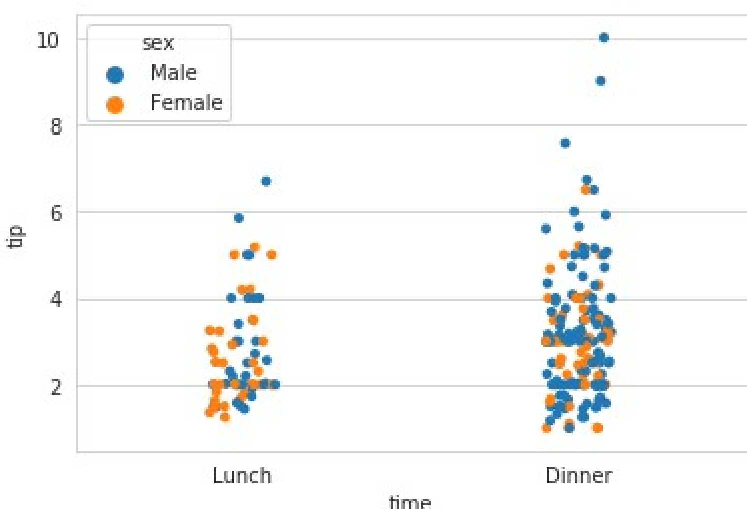


Рисунок 9.6 — Демонстрация использования параметра hue (пример 2)

Изменим порядок задания цвета:

```
sns.stripplot(x="time", y='tip', hue='sex', hue_order=['Female', 'Male'],
data=tips)
```

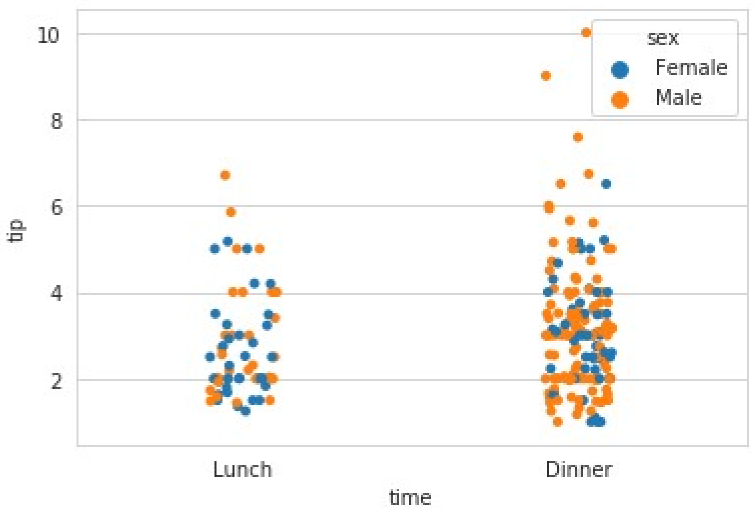


Рисунок 9.7 — Демонстрация работы с параметром `hue_order`

Изменим цветовую палитру через параметр `palette`:

```
color_palette = {"Male":"r", "Female":"g"}
sns.stripplot(x="size", y='tip', hue="sex", palette=color_palette,
data=tips)
```

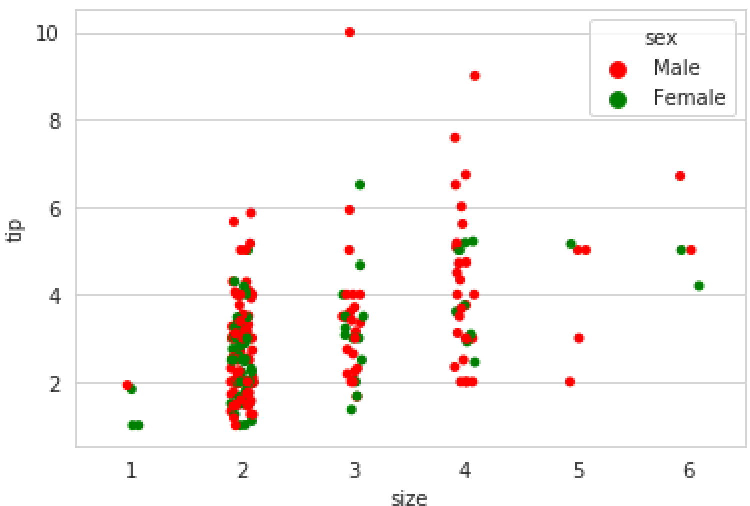


Рисунок 9.8 — Пример изменения цветовой схемы через параметр `palette`

Увеличим размер маркеров, зададим им цвет и ширину границы:

```
sns.stripplot(x="time", y='tip', size=10, edgecolor="gray", linewidth=1,
data=tips.sample(frac=1, random_state=123)[:30])
```

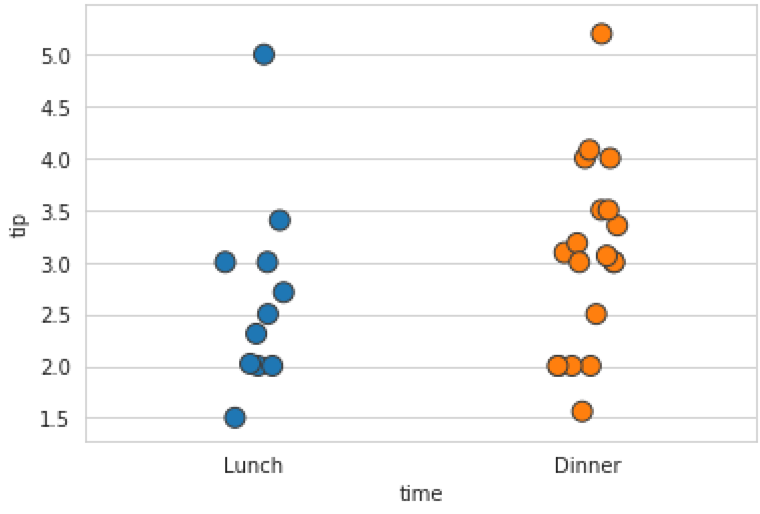


Рисунок 9.9 — Настройка внешнего вида маркеров через параметры `size`, `edgecolor` и `linewidth`

Поработаем с параметрами, которые отвечают за задание ориентации диаграммы, порядок отображения значений признака, расщепление основного набора данных по значению дополнительного признака.

Изменим порядок вывода признаков: поменяем *Dinner* и *Lunch* местами:

```
sns.stripplot(x="time", y='tip', hue_order=["Dinner", "Lunch"],
data=tips)
```

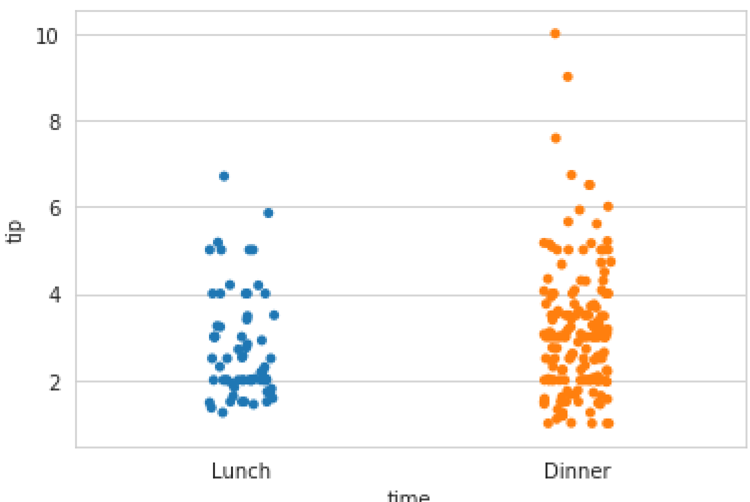


Рисунок 9.10 — Демонстрация работы с параметром hue_order

Ориентация диаграммы задаётся через параметр `orient`:

```
sns.stripplot(x="tip", y='time', orient='h', data=tips)
```

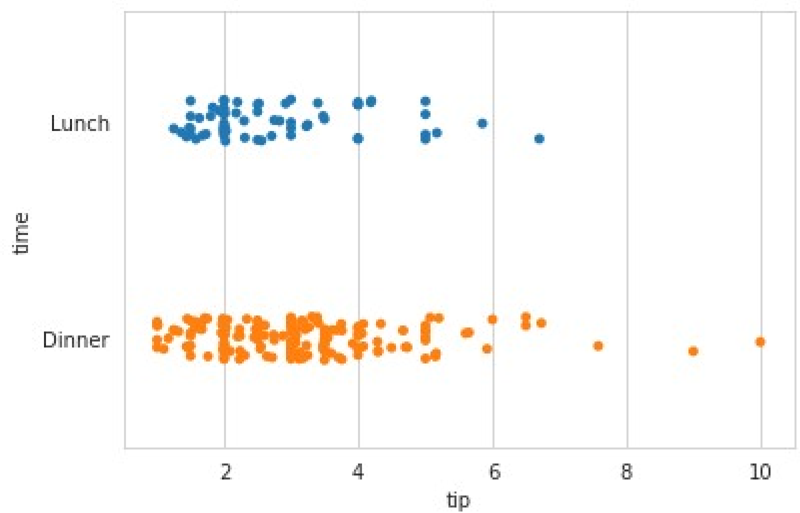


Рисунок 9.11 — Задание горизонтальной ориентации диаграммы

Обратите внимание, что имена признаков для x и y тоже были переставлены местами. Для улучшения наглядности диаграммы, в которой используется цветовое разделение через параметр hue, можно представить полученные наборы в виде визуально различимых групп с помощью параметра dodge:

```
sns.stripplot(x="time", y='tip', hue="sex", dodge=True, data=tips)
```

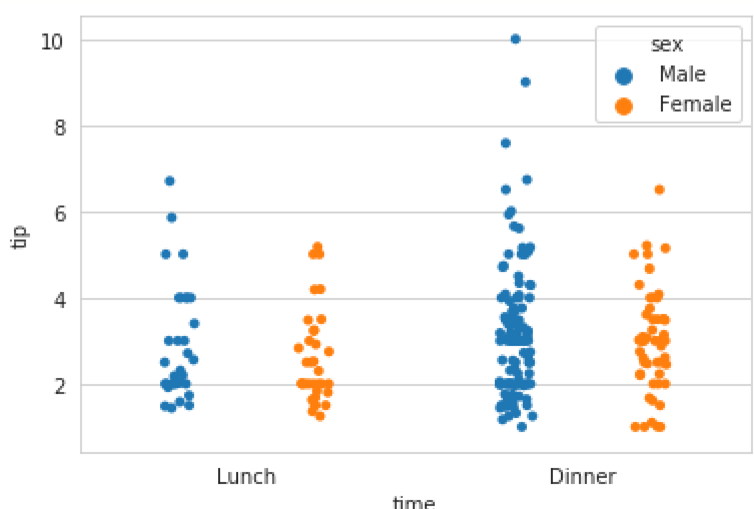


Рисунок 9.12 — Демонстрация работы с параметром dodge

Кучность представляемых наборов регулируется параметром `jitter`. Эта опция может быть полезна, если необходимо устранить перекрытие между наборами данных. Пример с настройками по умолчанию:

```
sns.stripplot(x="size", y='tip', data=tips)
```

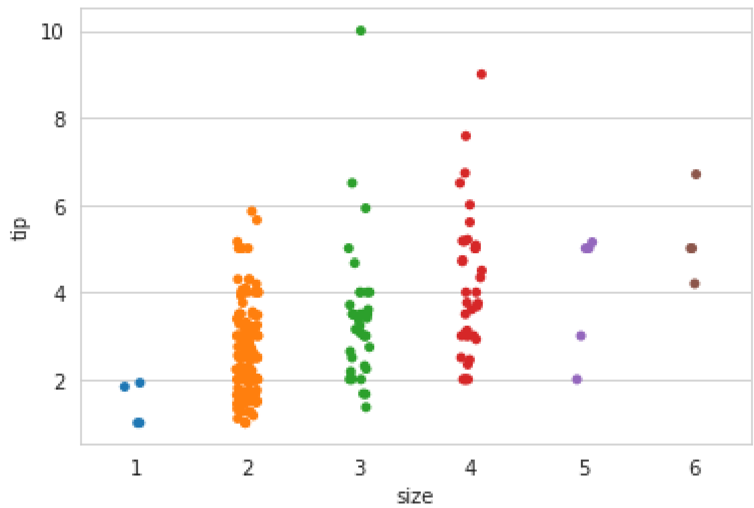


Рисунок 9.13 — Диаграмма со значением по умолчанию для параметра `jitter`

Изменим значение `jitter` на 0.03:

```
sns.stripplot(x="size", y='tip', size=5, jitter=0.03, data=tips)
```

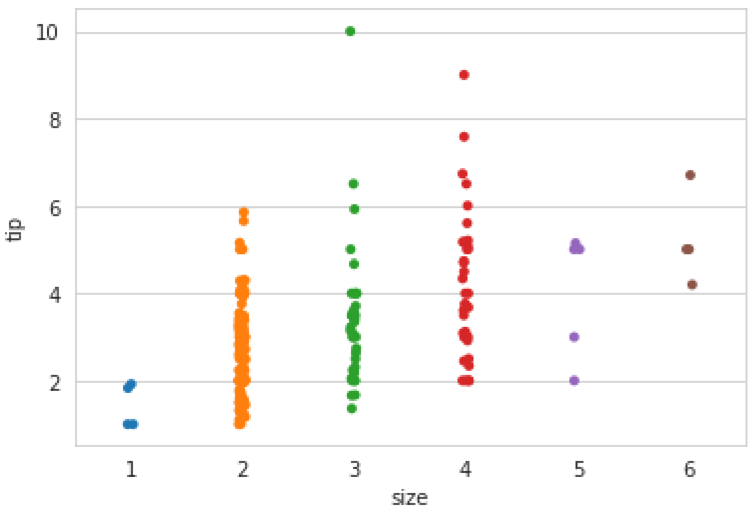


Рисунок 9.14 — Диаграмма со значением `jitter` равным 0.03

Если `jitter` присвоить `True`, то будет автоматически подобрано наиболее оптимальное значение.

9.2.2 Функция `swarmplot()`

Функция `swarmplot()` по своему функционалу и параметрам для настройки отображения подобна функции `stripplot()`, за исключением параметра `jitter`, который у неё отсутствует. Идея `swarmplot()` в том, что отображаемые точки на диаграмме не перекрываются, это позволяет делать выводы о преобладании тех или иных значений в наборах данных по их визуальному распределению.

Рассмотрим несколько примеров, демонстрирующих работу с `swarmplot()`:

```
sns.swarmplot(x="time", y='tip', data=tips)
```

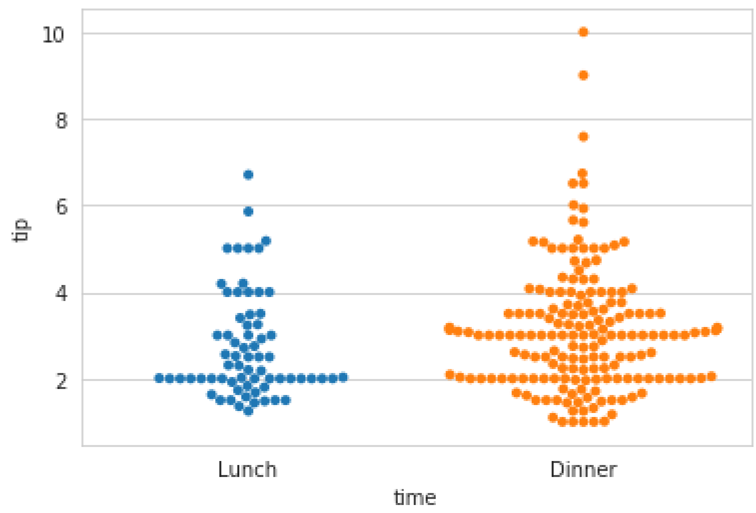


Рисунок 9.15 — Диаграмма, построенная функцией `swarmplot()`

Из представленной диаграммы можно сделать вывод, что за ланчем чаще всего оставляют два доллара в качестве чаевых, а за обедом от двух до четырёх долларов.

Для задания цветовой схемы будем использовать признак, отвечающий за то, курили при этом клиенты или нет:

```
sns.swarmplot(x="time", y='tip', hue="smoker", data=tips)
```

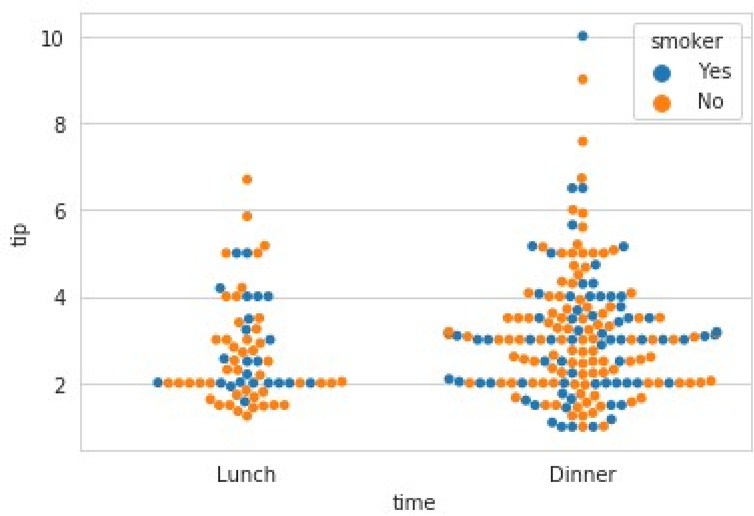


Рисунок 9.16 — Демонстрация цветового разделения по признаку *smoker*

Возьмём подвыборку из набора в количестве 50 штук и дополнительно настроим размер маркеров, их цвет и ширину границы:

```
df = tips.sample(frac=1, random_state=123)[:50]
sns.swarmplot(x="time", y='tip', size=10, edgecolor="gray", linewidth=2, data=df)
```

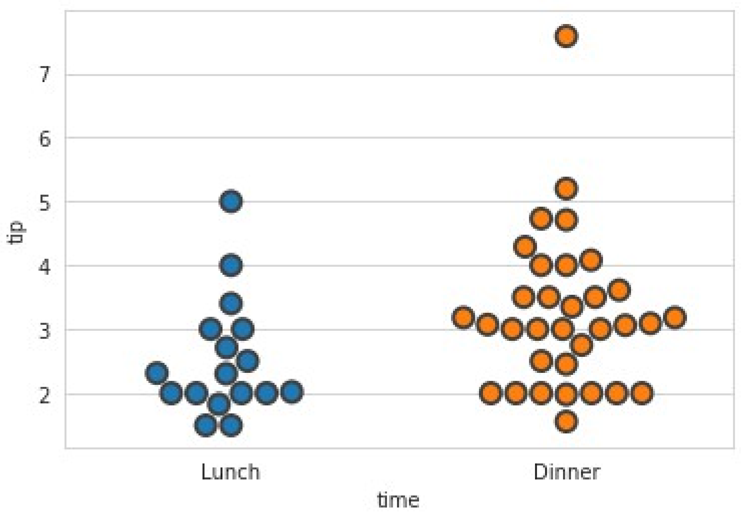


Рисунок 9.17 — Диаграмма с измеренными размером, цветом и шириной границы маркеров

Зададим самостоятельного цветовую палитру:

```
color_palette = {"Male":"g", "Female":"y"}
sns.swarmplot(x="time", y='tip', hue="sex", size=10, edgecolor="gray",
palette=color_palette, linewidth=2, data=df)
```

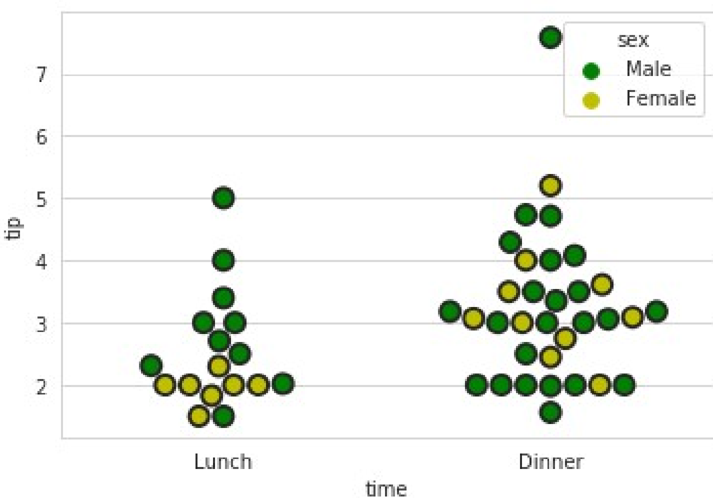


Рисунок 9.18 — Диаграмма с изменённой цветовой палитрой

За ориентацию диаграммы отвечает параметр `orient`, за порядок — `order`, параметр `dodge` управляет визуальным разделением данных. Проиллюстрируем на примерах их использование. Изменим ориентацию и порядок отображения значений признака *time*:

```
sns.swarmplot(x="tip", y='time', order=['Dinner', 'Lunch'], orient='h',
hue="sex", data=tips)
```

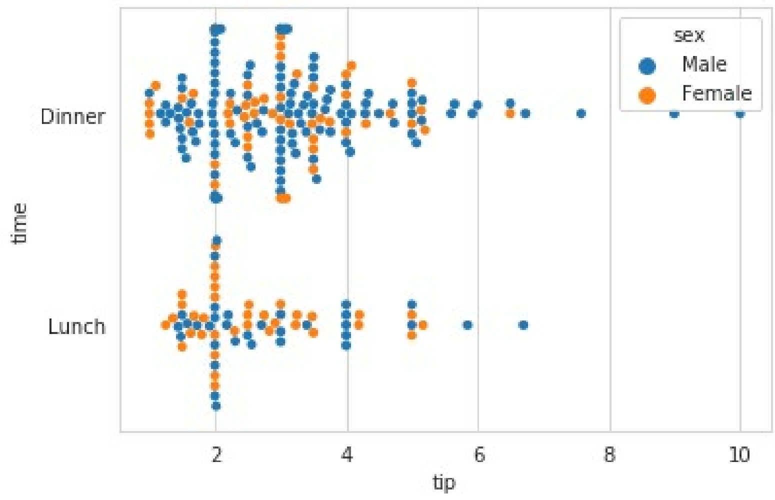


Рисунок 9.19 — Демонстрация работы с параметрами `order` и `orient`

Разделим с помощью параметра dodge выбору на группы:

```
sns.swarmplot(x="time", y='tip', hue="smoker", dodge=True, data=tips)
```

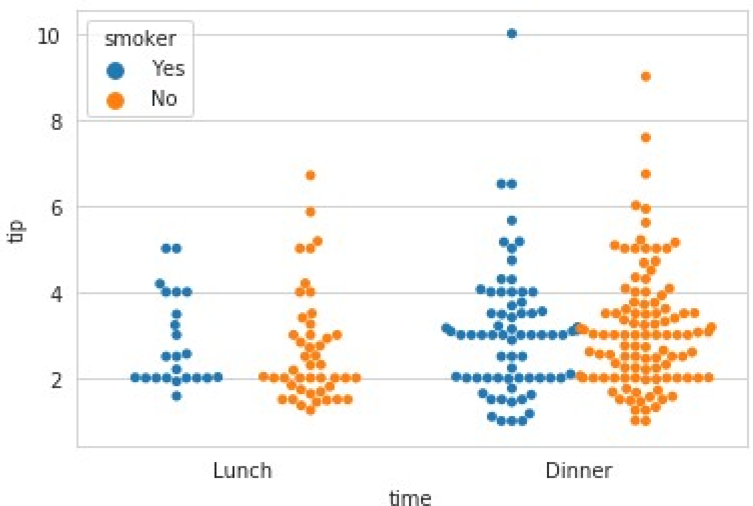


Рисунок 9.20 — Демонстрация работы с параметром dodge

9.3 Визуализации распределений категориальных данных

На практике, при решении задач статистики, чаще приходится строить не диаграммы рассеяния, а диаграммы распределений. По ним можно получить представление об изучаемых наборах данных по ряду числовых характеристик, значения которых отображаются на такого типа диаграммах.

Seaborn предоставляет три функции для визуализации распределений категориальных данных:

- `boxplot()` – строит диаграмму типа "ящик с усами", иногда её называют через транскрипцию: боксплот, на ней отображаются медианное значение, квартили и выбросы;
- `violinplot()` – строит диаграмму похожую на "ящик с усами" с оценкой плотности ядра;

- `boxenplot()` — строит диаграмму из прямоугольников, хорошо подходит для визуализации больших наборов данных.

Для настройки внешнего вида диаграмм используются параметры, перечисленные в начале главы. В дополнение к ним функции имеют общий набор аргументов и индивидуальные элементы настройки. Рассмотрим общий набор дополнительных параметров для функций визуализации распределений категориальных данных:

- `saturation`: `float`, `optional`
  - Пропорция насыщенности цвета.
- `width`: `float`, `optional`
  - Ширина элемента диаграммы (например ящика с усами).
- `dodge`: `bool`, `optional`
  - Если параметр равен `True` и используется параметр `hue`, то на диаграмме данные будут представлены в виде визуально разделимых групп.
- `linewidth`: `float`, `optional`
  - Ширина рамки элементов.

9.3.1 Функция `boxplot()`

Для начала разберёмся с тем, какую информацию мы можем получить от диаграммы *boxplot*. Её внешний вид представлен на рисунке 9.21. На диаграмме отображена медиана (2-ой квартиль), 1-ый и 3-ий квартили; которые формируют межквартильный размах; если отложить вверх и вниз по полтора межквартильных размаха, то последние значения признака на этих промежутках формируют границы усов, значения вне этих границ будут определяться как выбросы, они отображаются отдельными точками.

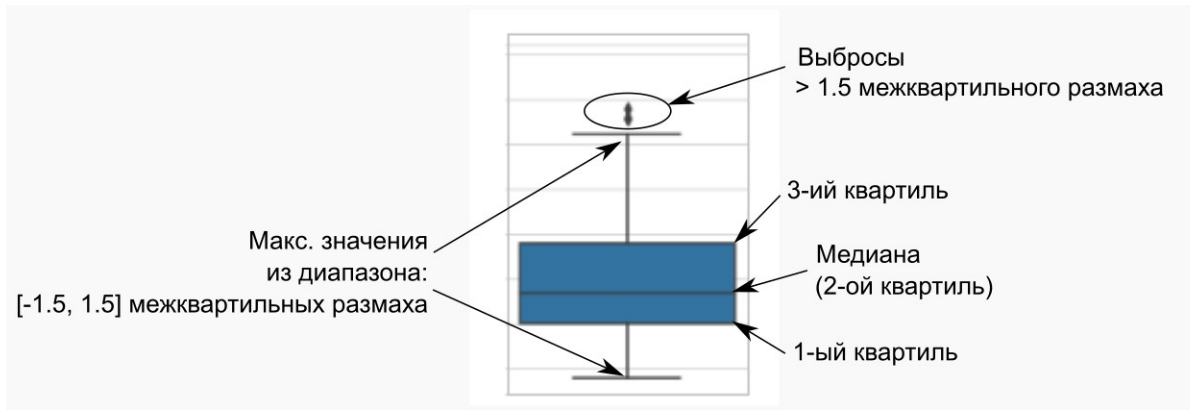


Рисунок 9.21 — Диаграмма *boxplot*

Помимо параметров общих для всех функций визуализации данных *seaborn* (см. *8.1 Общие параметры функций*) и инструментов для работы с категориальными данными, функция `boxplot()` имеет ряд уникальных параметров:

- `fliersize`: `float`, `optional`
  - Размер маркеров, с помощью которых обозначаются выбросы.
- `whis`: `float`, `optional`
  - Величина межквартильного размаха, задающего длину усов диаграммы.

Простроим диаграмму для набора данных *mpg*, отобразим распределение количества миль, которое автомобиль проезжает на одном галлоне топлива для США и Европы:

```
mpg = sns.load_dataset("mpg")
mpg_mod = mpg[mpg["origin"] != "japan"]
sns.boxplot(x="origin", y="mpg", data=mpg_mod)
```

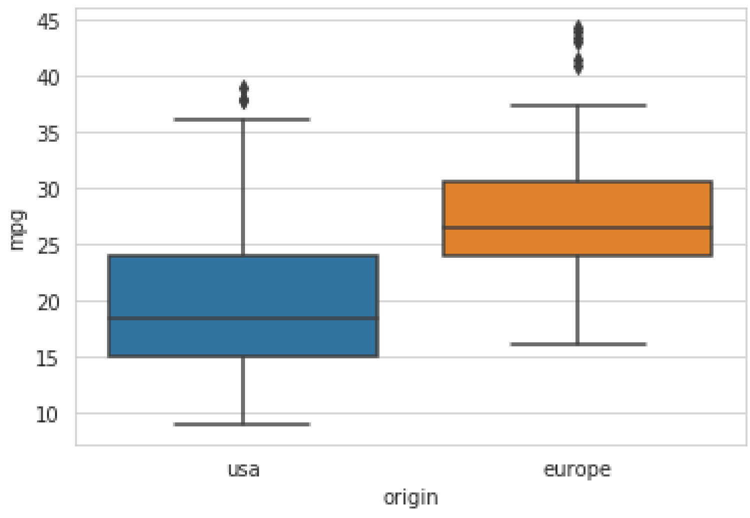


Рисунок 9.22 — Демонстрация работы функции `boxplot()`

Сделаем цветовую заливку монотонной:

```
sns.boxplot(x="origin", y="mpg", color='g', data=mpg_mod)
```

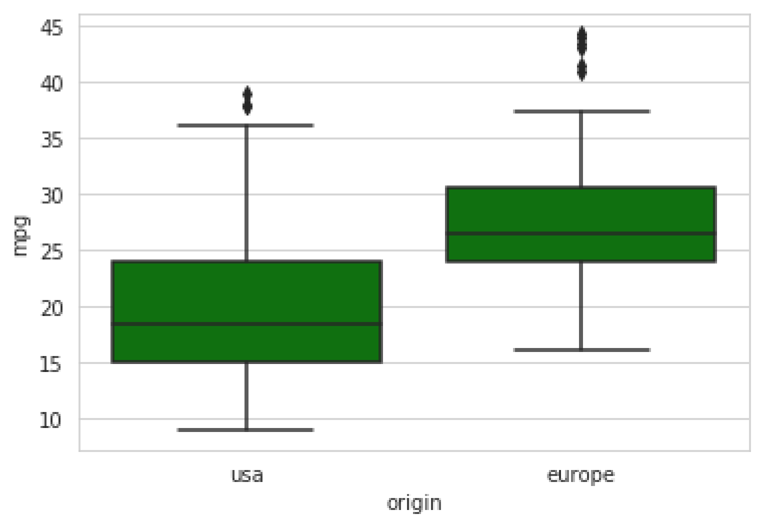


Рисунок 9.23 — Диаграмма *boxplot* с заливкой зелёного цвета

Насыщенность цвета можно изменить через параметр `saturation`, по умолчанию это значение равно 0.75, т.е. изображение выводится с заниженной насыщенностью, как правило, такое решение даёт более приятное визуальное отображение цвета. Сравним, как будет выглядеть диаграмма при разных значениях параметра `saturation`:

```
plt.figure(figsize=(15, 5))
sat_list = [1, 0.75, 0.5, 0.25]
for i, s in enumerate(sat_list):
    plt.subplot(1, len(sat_list), i+1)
    sns.boxplot(x="origin", y="mpg", saturation=s, data=mpg_mod)
```

Рисунок 9.24 — Демонстрация работы с параметром saturation

Ещё одним инструментом для изменения визуального оформления диаграммы является параметр `linewidth`, через него задаётся толщина линий "ящика с усами":

```
plt.figure(figsize=(15, 5))
lw_list = [6, 3, 1]
for i, lw in enumerate(lw_list):
    plt.subplot(1, len(lw_list), i+1)
    sns.boxplot(x="origin", y="mpg", linewidth=lw, data=mpg_mod)
```

Рисунок 9.25 — Демонстрация работы с параметром linewidth

Добавим дополнительное цветовое разделение по количеству цилиндров в двигателе автомобиля с помощью параметра hue:

```
sns.boxplot(x="origin", y="mpg", hue="cylinders", data=mpg_mod)
```

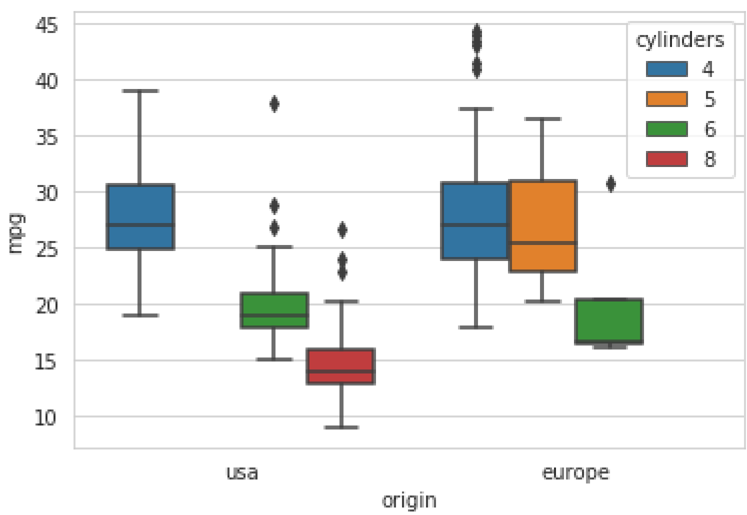


Рисунок 9.26 — Демонстрация работы с параметром hue

Зададим цветовую схему через параметр palette:

```
sns.boxplot(x="origin", y="mpg", hue="cylinders", palette='Pastel1',
data=mpg_mod)
```

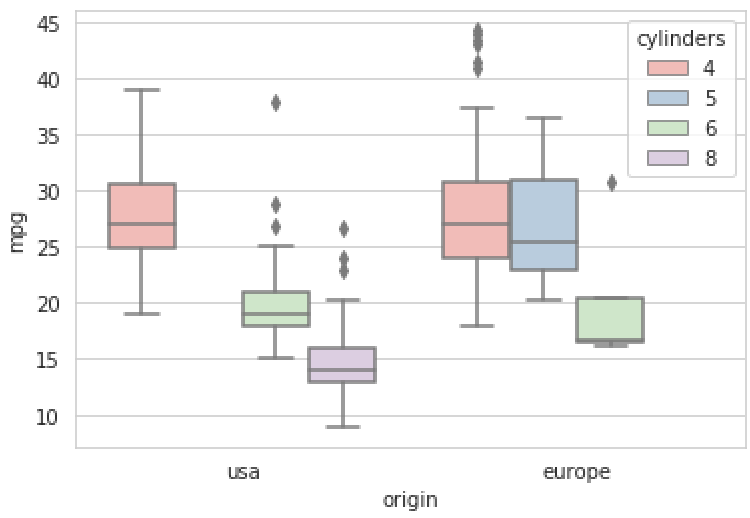


Рисунок 9.27 — Демонстрация работы с параметром palette

Порядок вывода значений основного признака определяется через параметр `order`, применение цвета — через `hue_order`. Ориентация диаграммы, также как у всех функций *seaborn*, управляется с помощью `orient`:

```
sns.boxplot(x="mpg", y="origin", orient='h', data=mpg_mod)
```

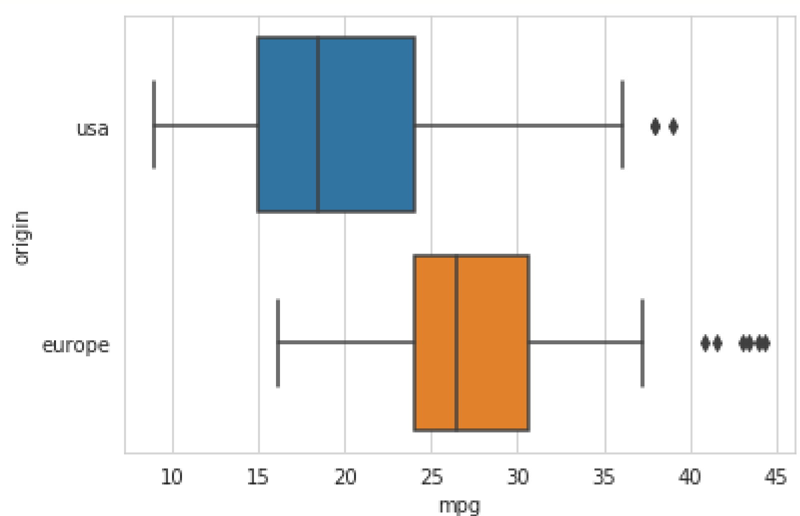


Рисунок 9.28 — Демонстрация работы с параметром orient

Ширина (или высота, если `orient='h'`) "ящика с усами" задаётся через параметр `width`. При значении равном 1 ящики занимают все предоставленное пространство, например, если признак принимает только два значения, то все поле графика будет поделено между двумя ящиками, увеличение или уменьшение `width` изменяет размер ящика:

```
plt.figure(figsize=(15, 5))
w_list = [1.25, 1, 0.75, 0.5]
for i, w in enumerate(w_list):
    plt.subplot(1, len(w_list), i+1)
    sns.boxplot(x="origin", y="mpg", width=w, data=mpg_mod)
```

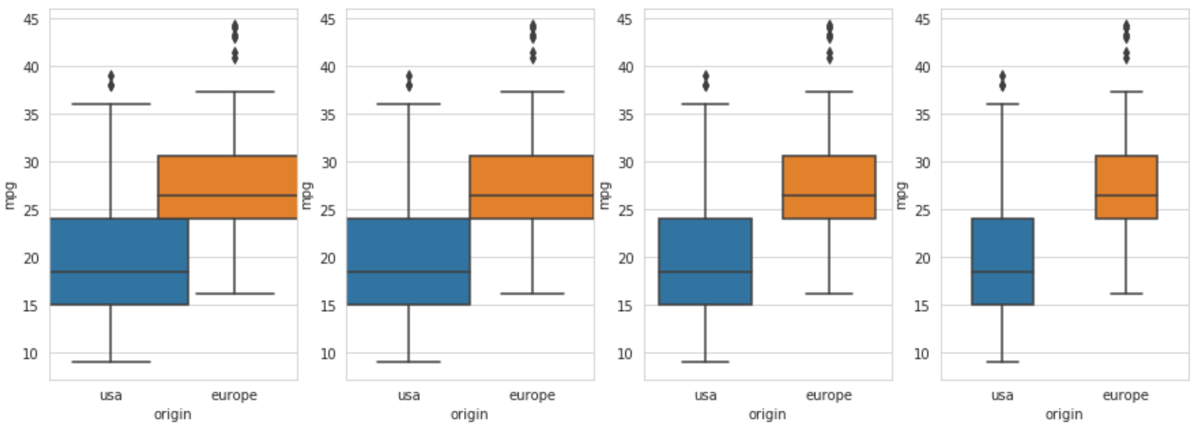


Рисунок 9.29 — Демонстрация работы с параметром width

За размер маркеров, обозначающих выбросы, отвечает параметр `fliersize`:

```
plt.figure(figsize=(10, 5))
plt.subplot(121)
sns.boxplot(x="origin", y="mpg", data=mpg_mod)
plt.subplot(122)
sns.boxplot(x="origin", y="mpg", fliersize=10, data=mpg_mod)
```

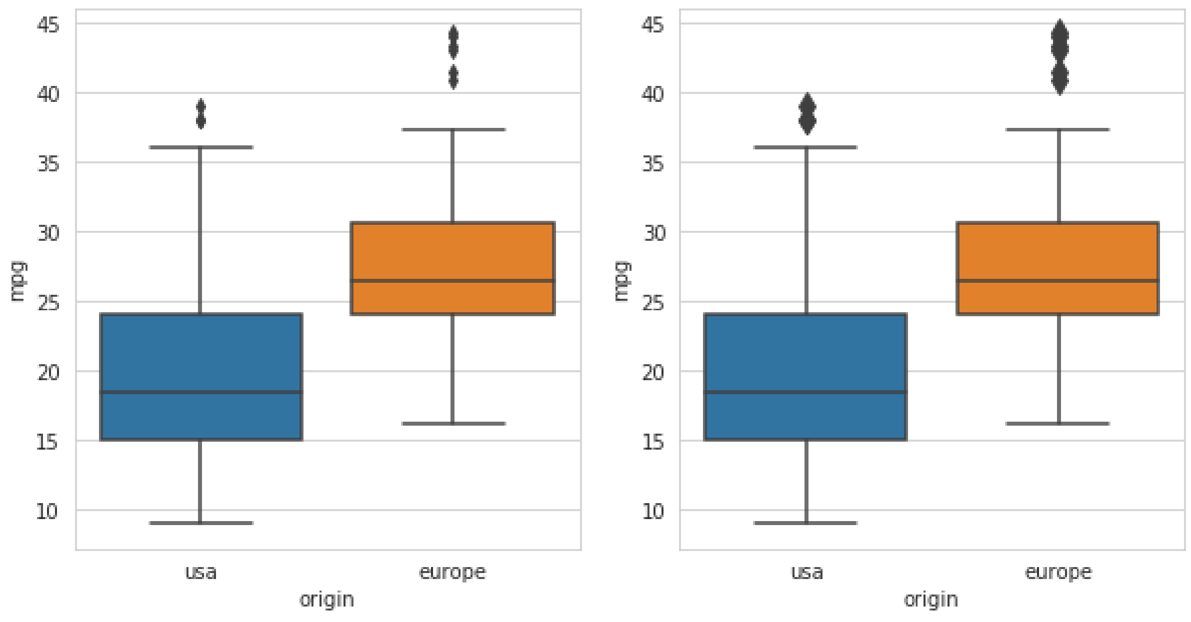


Рисунок 9.30 — Демонстрация работы с параметром fliersize

Как было сказано в начале главы, вертикальный размер ящика (его высота) равен интервалу между 1-м и 3-м квартилем, полтора таких интервала определяют длину усов, все значения из набора данных, которые больше этого интервала обозначаются как выбросы. За величину межквартильного размаха отвечает параметр whis, проверим, что по умолчанию, это значение равно 1.5:

```
plt.figure(figsize=(10, 5))
plt.subplot(121)
sns.boxplot(x="origin", y="mpg", data=mpg_mod)
plt.subplot(122)
sns.boxplot(x="origin", y="mpg", whis=1.5, data=mpg_mod)
```

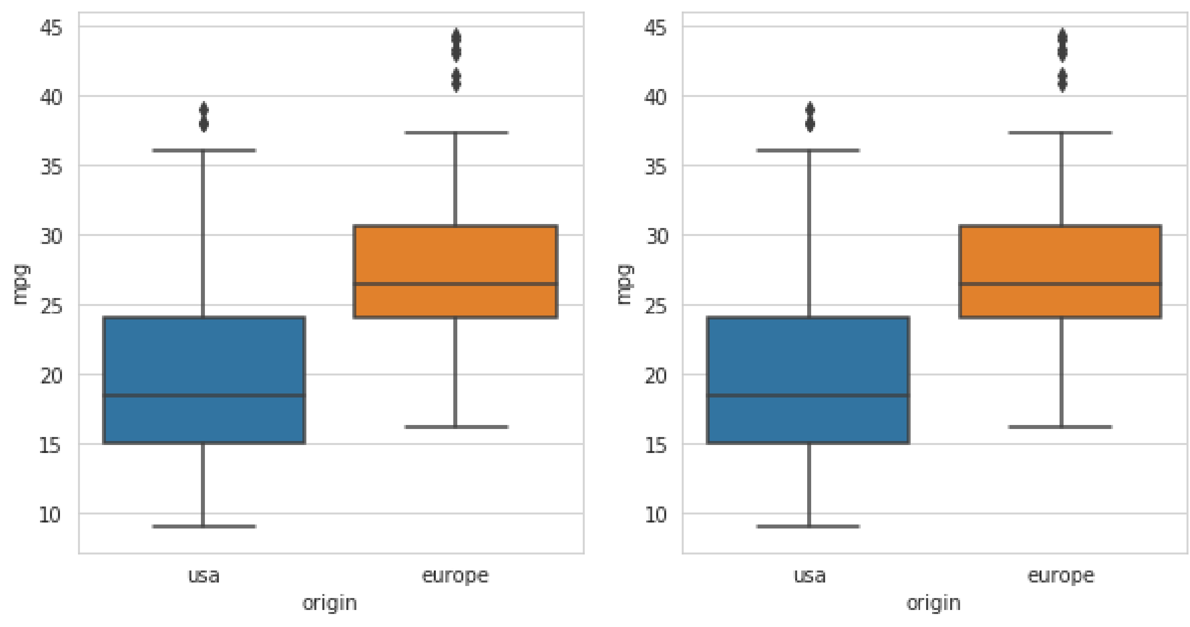


Рисунок 9.31 — Демонстрация работы с параметром whis=1.5

Теперь изменим эти величину, построим диаграмму, в которой размах равен двум:

```
sns.boxplot(x="origin", y="mpg", whis=2, data=mpg_mod)
```

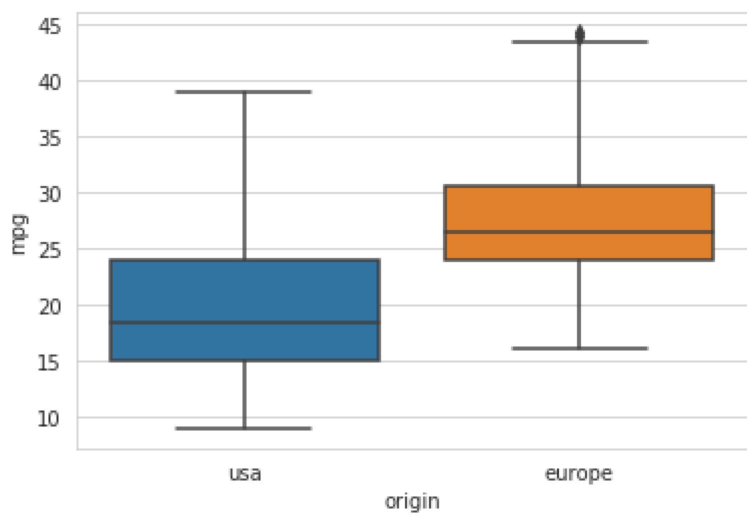


Рисунок 9.32 — Демонстрация работы с параметром `whis=2`

9.3.2 Функция violin()

Следующей функцией для визуализации категориальных данных, которую мы рассмотрим, будет `violin()`. По своему функциональному назначению и возможностям она похожа на рассмотренную ранее `boxplot()`, дополнительно на ней отображается оценка плотности ядра. Будем работать с тем же набором данных, что был использован нами при изучении `boxplot()`:

```
mpg = sns.load_dataset("mpg")
```

Построим диаграммы *boxplot* и *violine* рядом друг с другом для сравнения.

```
mpg_mod = mpg[mpg["origin"] != "japan"]
plt.figure(figsize=(10, 5))
plt.subplot(121)
sns.boxplot(x="origin", y="mpg", data=mpg_mod)
```

```
plt.subplot(122)
sns.violinplot(x="origin", y="mpg", data=mpg_mod)
```

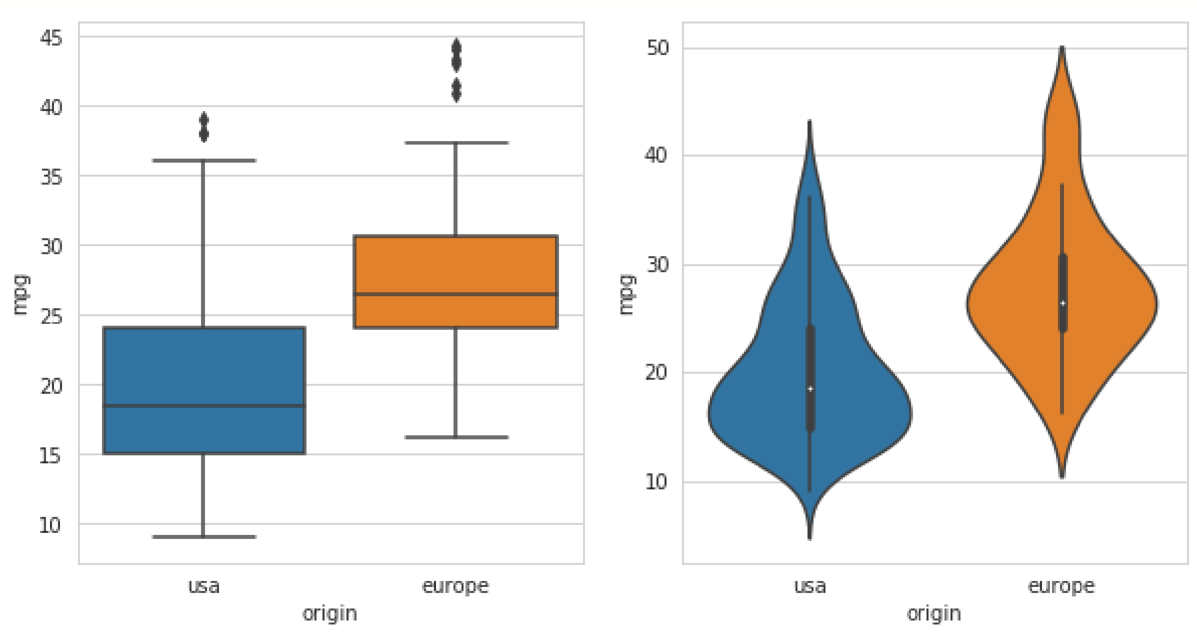


Рисунок 9.33 — Сравнение диаграмм *boxplot* и *violinplot*

Помимо уже знакомых нам параметров для работы с цветовой схемой и представлением `violineplot()` имеет ряд уникальных аргументов, некоторые из них приведены ниже:

- `bw`: `{'scott', 'silverman', float}`, `optional`
  - Ширина ядра (*kernel bandwidth*).
- `scale`: `{'area', 'count', 'width'}`, `optional`
  - Метод масштабирования ширины диаграммы.
- `split`: `bool`, `optional`
  - Если разделение набора данных через параметр hue производится на две группы, то при `split=True` будет отображена только половина диаграммы.

Цветовое оформление диаграммы

Единая цветовая гамма задаётся через параметр color:

```
sns.violinplot(x="origin", y="mpg", color='yellow', data=mpg_mod)
```

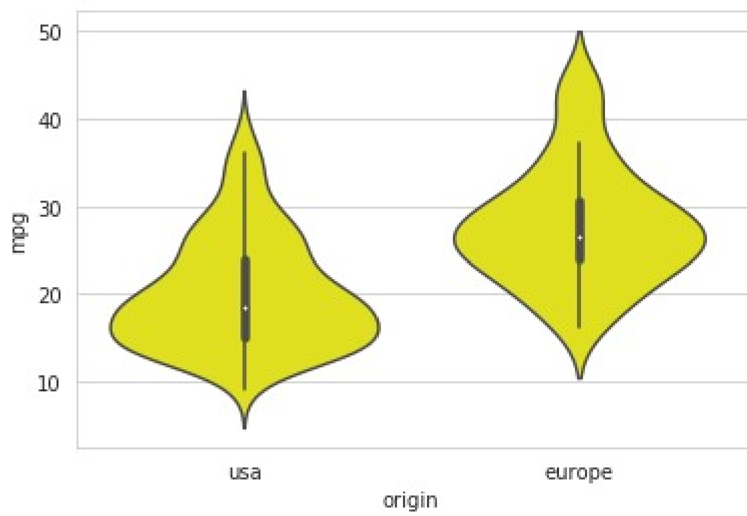


Рисунок 9.34 — Демонстрация работы с параметром color функции violinplot()

Выделим из набора данных *mpg* автомобили, произведённые в США и Японии, у которых количество цилиндров равно 4 или 6:

```
fn_filter = lambda x: True if x in [4, 6] else False
fn_mod = lambda x: {4: 'four', 6: 'six', 8: 'eight'}[x]
mpg_country = mpg[mpg["origin"] != "europe"]
mpg_demo = mpg_country[mpg_country['cylinders'].map(fn_filter)].copy()
mpg_demo['cylinders'] = mpg_demo['cylinders'].map(fn_mod)
```

Построим диаграмму с дополнительным цветовым разделением по количеству цилиндров:

```
sns.violinplot(x="origin", y="mpg", hue="cylinders", data=mpg_demo)
```

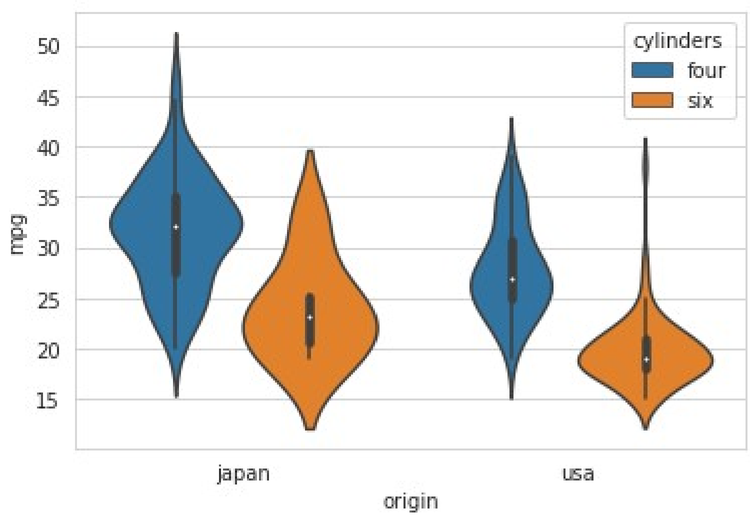


Рисунок 9.35 — Демонстрация работы с параметром hue функции violinplot()

Определим цветовую схему и порядок присвоения цветов для *cylinders*:

```
color_scheme = {"four": "y", "six":"violet"}
color_order = ["six", "four"]
sns.violinplot(x="origin", y="mpg", hue="cylinders",
hue_order=color_order, palette=color_scheme, data=mpg_demo)
```

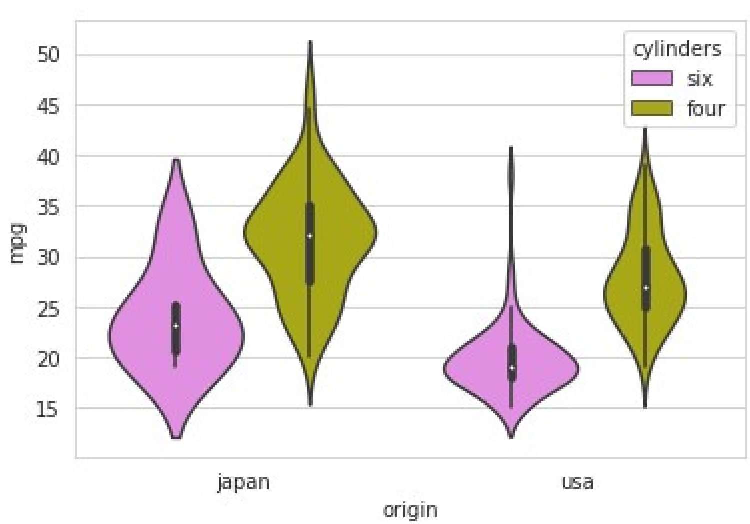


Рисунок 9.36 — Демонстрация работы с параметрами hue_order и palette функции violinplot()

Насыщенность цвета задаётся через параметр `saturation`:

```
plt.figure(figsize=(15, 5))
s_list = [0.25, 0.5, 0.75, 1]
for i, s in enumerate(s_list):
    plt.subplot(1, len(s_list), i+1)
    sns.violinplot(x="origin", y="mpg", hue="cylinders", saturation=s,
data=mpg_demo)
```

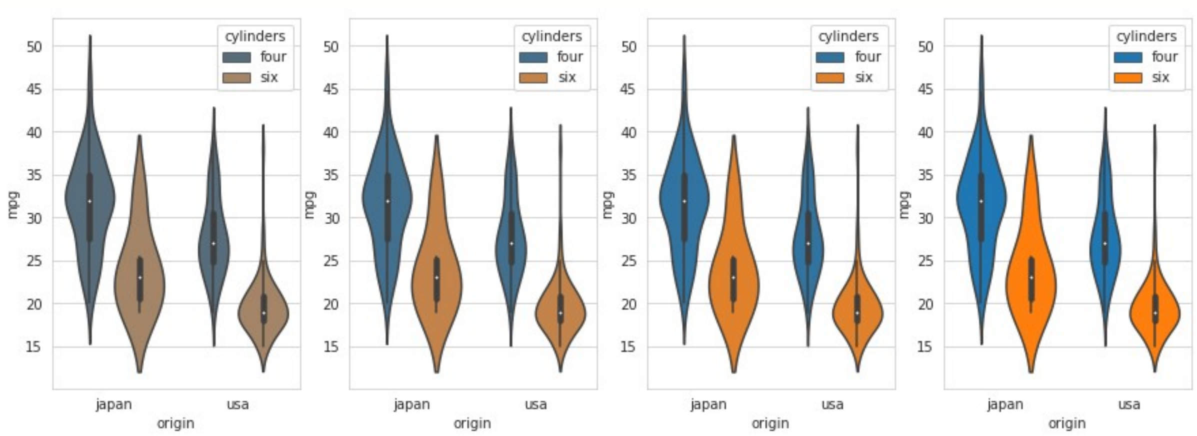


Рисунок 9.37 — Демонстрация работы с параметром `saturation` функции `violinplot()`

Заметим, что эти диаграммы имеют симметричную форму, поэтому для того, чтобы судить о характере распределения достаточно только её половины, можно объединить их — левую часть взять от распределения с количеством цилиндров равным шести, правую — с количеством равным четырём:

```
sns.violinplot(x="origin", y="mpg", hue="cylinders",
hue_order=color_order, palette=color_scheme, split=True, data=mpg_demo)
```

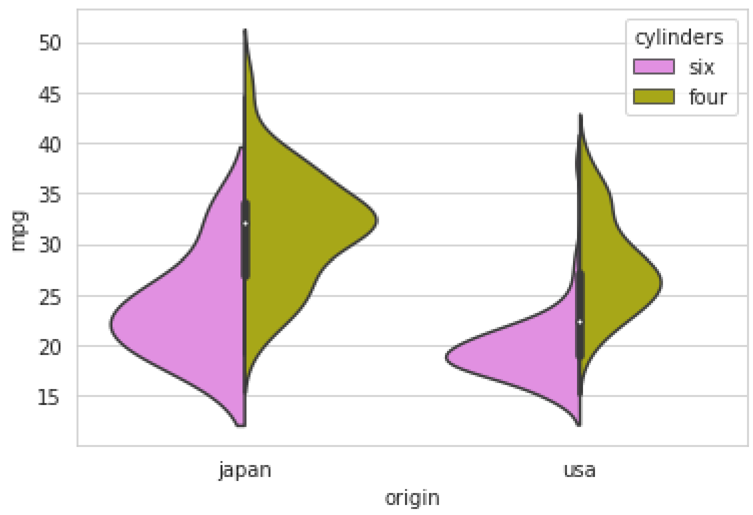


Рисунок 9.38 — Демонстрация работы с параметром `split` функции `violinplot()`

Толщина линии контура диаграммы задаётся через `linewidth`:

```
plt.figure(figsize=(15, 5))
lw_list = [6, 3, 1]
for i, lw in enumerate(lw_list):
    plt.subplot(1, len(lw_list), i+1)
    sns.violinplot(x="origin", y="mpg", linewidth=lw, data=mpg_demo)
```

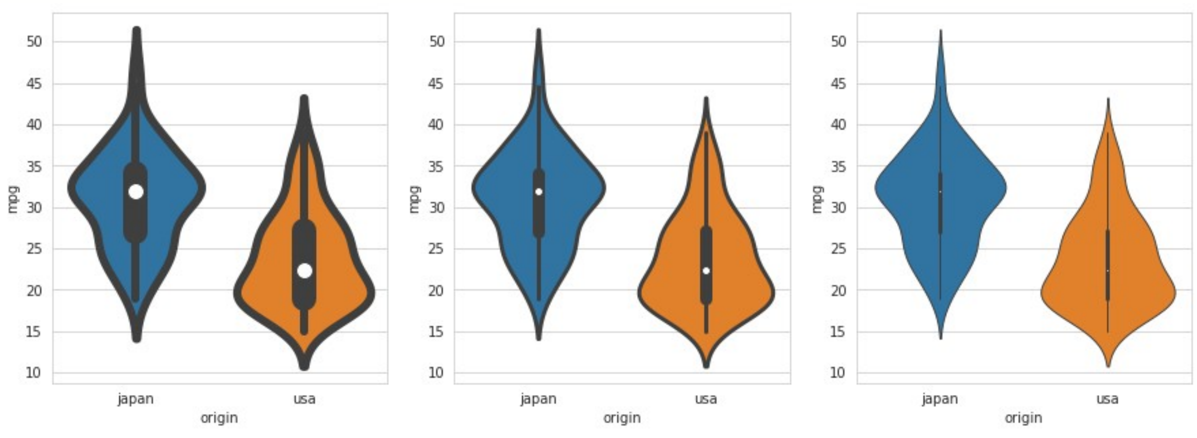


Рисунок 9.39 — Демонстрация работы с параметром `linewidth` функции `violinplot()`

Ширина ядра устанавливается через параметр bw:

```
bw_values = ["scott", "silverman", 0.25, 1]
plt.figure(figsize=(15, 5))
for i, b in enumerate(bw_values):
    plt.subplot(1, len(bw_values), i+1)
    plt.title(str(b))
    sns.violinplot(x="origin", y="mpg", bw=b, data=mpg_demo)
```

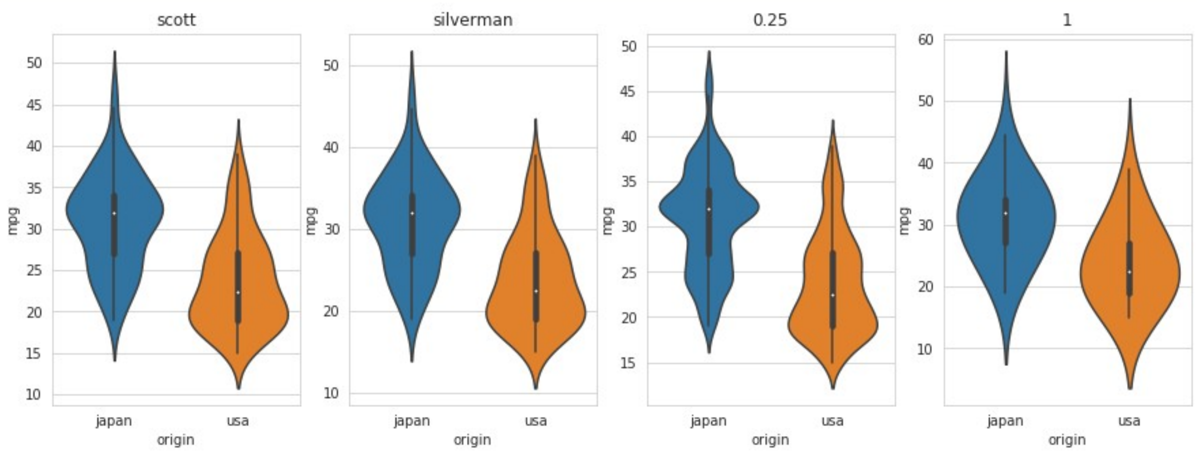


Рисунок 9.40 — Демонстрация работы с параметром bw функции violinplot()

Масштаб диаграммы можно задать через `scale`, этот параметр принимает одно из трех значений: `{'area', 'count', 'width'}`:

```
scale_values = ["area", "count", "width"]
plt.figure(figsize=(15, 5))
for i, s in enumerate(scale_values):
    plt.subplot(1, len(scale_values), i+1)
    plt.title(str(s))
    sns.violinplot(x="origin", y="mpg", scale=s, data=mpg_demo)
```

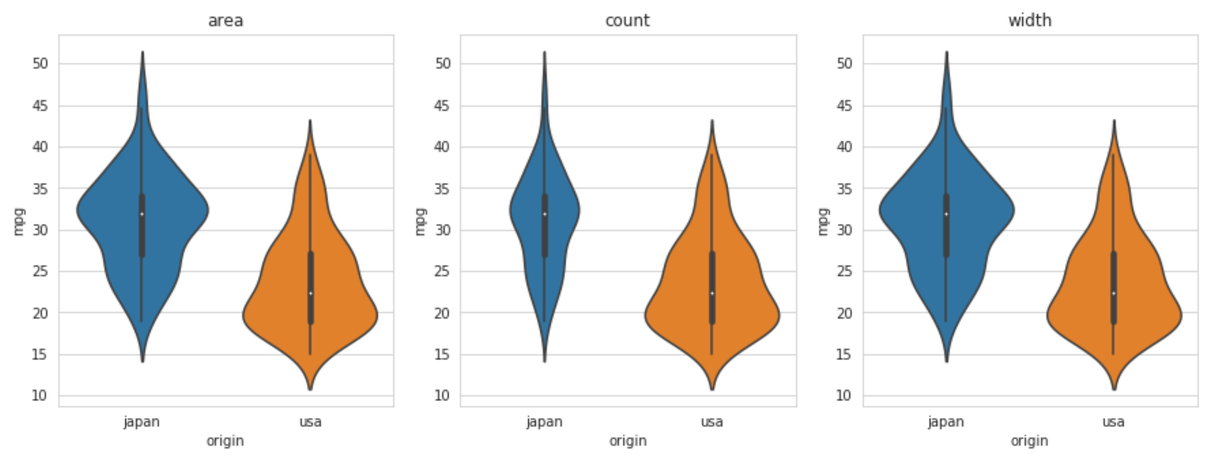


Рисунок 9.41 — Демонстрация работы с параметром `scale` функции `violinplot()`

За порядок и ориентацию отвечают параметры `order` и `orient`:

```
sns.violinplot(x="mpg", y="origin", orient='h',
order=["usa", "japan"], data=mpg_demo)
```

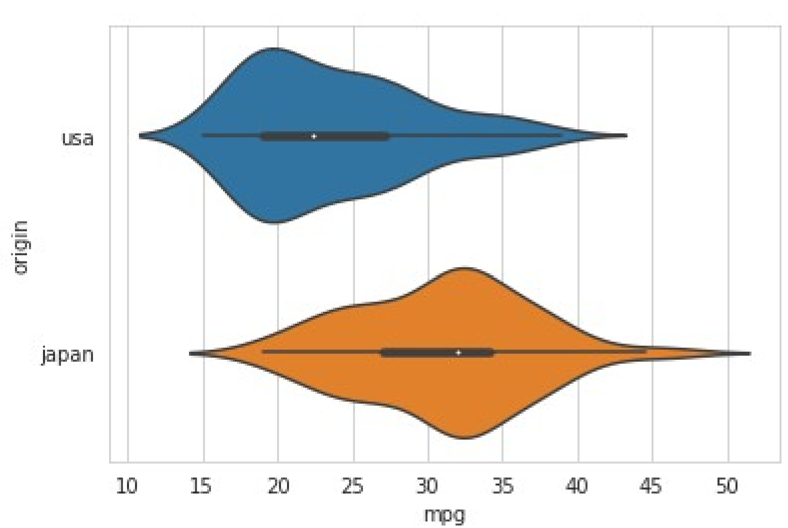


Рисунок 9.42 — Демонстрация работы с параметрами `order` и `orient` функции `violinplot()`

9.4 Визуализация оценок категориальных данных

В данную группу входят функции `pointplot()`, `barplot()` и `countplot()`, их назначение — это визуализация различных обобщенных характеристик наборов данных (будем их называть оценками), например, количество элементов, относящихся к той или ной

группе, величина стандартного отклонения и т.п. Группы параметров этих функций частично пересекаются между собой, но мы не будем их рассматривать в виде отдельного списка.

9.4.1 Функция pointplot()

Функция `pointplot()` отображает оценку какого-либо набора данных как точку на поле графика и доверительный интервал в виде линии, центр которой лежит на указанной точке.

Демонстрацию будем проводить на наборе данных *dots*:

```
dots = sns.load_dataset("dots")
```

Построим диаграмму `pointplot`, в качестве категориального признака выберем *align*, оценку будем считать по *firing_rate*:

```
sns.pointplot(x='align', y='firing_rate', data=dots)
```

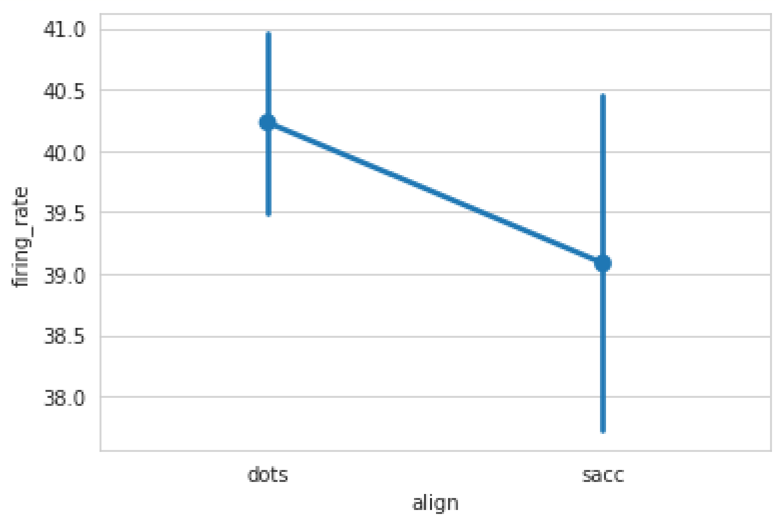


Рисунок 9.43 — Диаграмма *pointplot*

Как вы можете видеть из рисунка: среднее значение *firing_rate*, для *align=dots* приблизительно равно 40.25, для *align=sacc* составляет 39.1. Получим точные значения этих величин:

```
dots[dots["align"]=='dots']['firing_rate'].mean()
```

```
40.23124948122005
```

```
dots[dots["align"]=='sacc']['firing_rate'].mean()
```

```
39.083297066051394
```

Вертикальными линиями обозначен 95% доверительный интервал.

Добавим ещё один уровень разделения — по признаку *choice*, выделим его с помощью параметра hue:

```
sns.pointplot(x='align', y='firing_rate', hue='choice', data=dots)
```

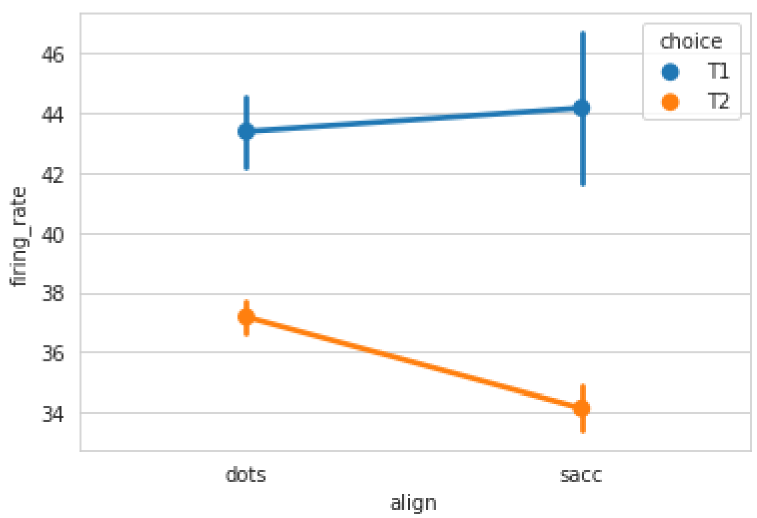


Рисунок 9.44 — Демонстрация разделения данных с помощью параметра hue функции pointplot()

Изменим цветовую палитру:

```
sns.pointplot(x='align', y='firing_rate', hue='choice', palette='GnBu',
data=dots)
```

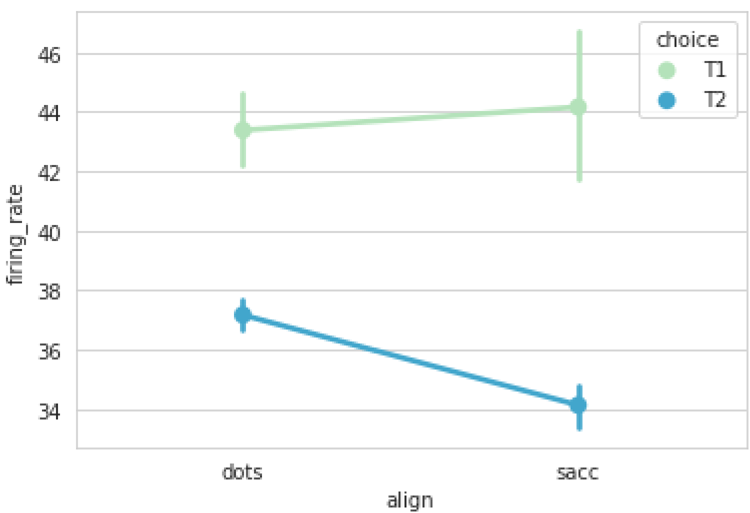


Рисунок 9.45 — Демонстрация изменения цветовой палитры с помощью параметра `palette` функции `pointplot()`

За алгоритм вычисления оценки отвечает параметр `estimator`, через который задаётся функция расчёта статистики, принимающая вектор и возвращающая скалярное значение. Построим диаграммы с различным способом определения оценки:

```
from numpy import mean, median, min, max
estimator = [mean, median, min, max]
plt.figure(figsize=(15, 5))
for i, es in enumerate(estimator):
    plt.subplot(1, len(estimator), i+1)
    plt.title(es.__name__)
    sns.pointplot(x='align', y='firing_rate', estimator=es, data=dots)
```

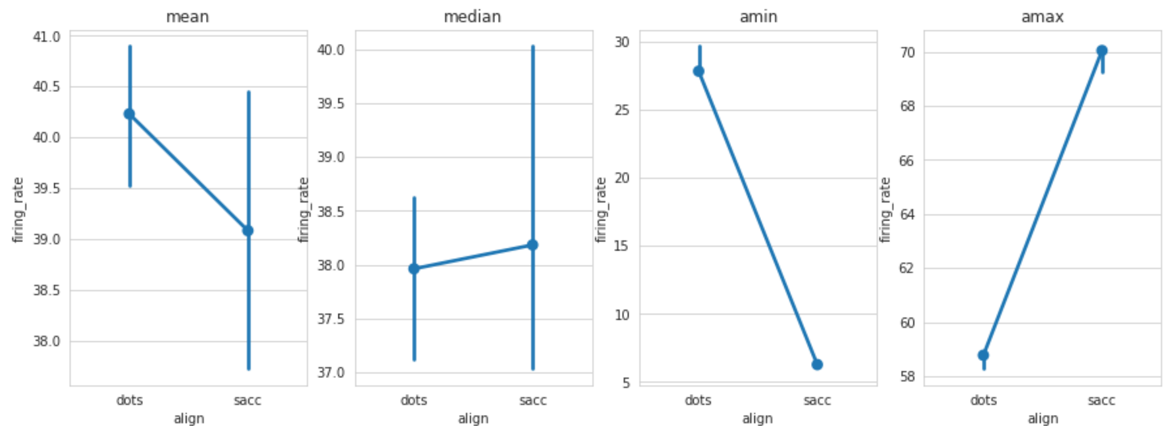


Рисунок 9.46 — Демонстрация работы с параметром `estimator` функции `pointplot()`

Для задания доверительного интервала используется параметр `ci` (см. раздел "*8.1.2 Параметры для повышения информативности графиков*"). Помимо числового значения, явно определяющего величину доверительного интервала, в него можно передать `'sd'`, если требуется отобразить стандартное отклонение, или `None` – в этом случае вертикальные линии отображаться не будут:

```
cis = [None, 95, 'sd']
plt.figure(figsize=(15, 5))
for i, c in enumerate(cis):
    plt.subplot(1, len(cis), i+1)
    plt.title(f"ci = {c}")
    sns.pointplot(x='align', y='firing_rate', ci=c, data=dots)
```

Рисунок 9.47 — Демонстрация работы с параметром `ci` функции `pointplot()`

Стиль маркеров и соединяющей линии задаётся через параметры `markers` и `linestyle`:

- `markers`: строка или список строк, `optional`
  - Маркеры, которые будут использоваться для каждого значения признака, переданного через hue.
- `linestyles`: строка или список строк, `optional`
  - Стили линий, которые будут использованы для значений признака, переданного через hue.

```
ms = ["s", "^"]
ls = ["--", "-."]
sns.pointplot(x='align', y='firing_rate', hue='choice', markers=ms,
linestyles=ls, data=dots)
```

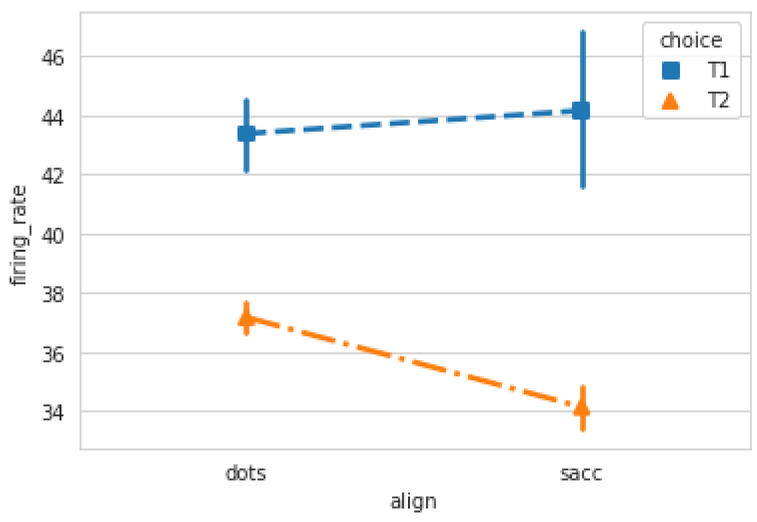


Рисунок 9.48 — Демонстрация работы с параметрами `markers` и `linestyles` функции `pointplot()`

Дополнительно для настройки внешнего вида диаграммы могут быть полезны следующие параметры функции `pointplot()`:

- `scale`: `float`, `optional`
  - Множитель, определяющий размер элементов диаграммы.

- `errwidth`: `float`, `optional`
    - Толщина линий, представляющих доверительные интервалы и их выносные элементы.
- `capsize`: `float`, `optional`
    - Ширина выносных элементов линий, представляющих доверительный интервал

Пример использования параметров `errwidth`, `capsize`:

```
errwidthes = [1, 2, 4]
capsizes = [0, 0.5, 1]


plt.figure(figsize=(15, 5))
for i, p in enumerate(zip(errwidthes, capsizes)):
    plt.subplot(1, len(errwidthes), i+1)
    plt.title(f"errwidth = {p[0]}, capsize = {p[1]}")
    sns.pointplot(x='align', y='firing_rate', errwidth=p[0],
capsize=p[1], data=dots)
```

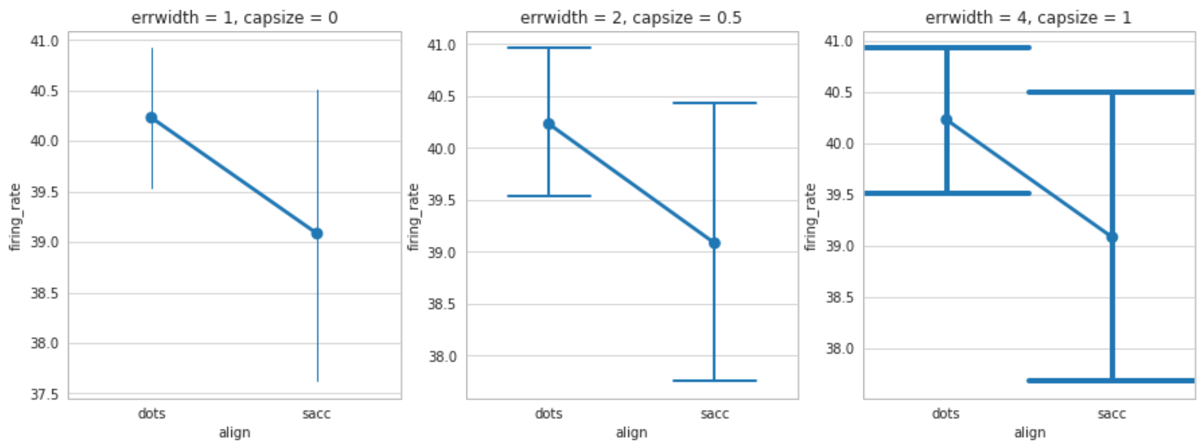


Рисунок 9.49 — Демонстрация работы с параметрами `errwidth` и `capsize` функции pointplot()

Демонстрация работы с параметром `scale`:

```
scales = [0.5, 1, 2]

plt.figure(figsize=(15, 5))
for i, s in enumerate(scales):
    plt.subplot(1, len(scales), i+1)
    plt.title(f"scale = {s}")
    sns.pointplot(x='align', y='firing_rate', scale=s, data=dots)
```

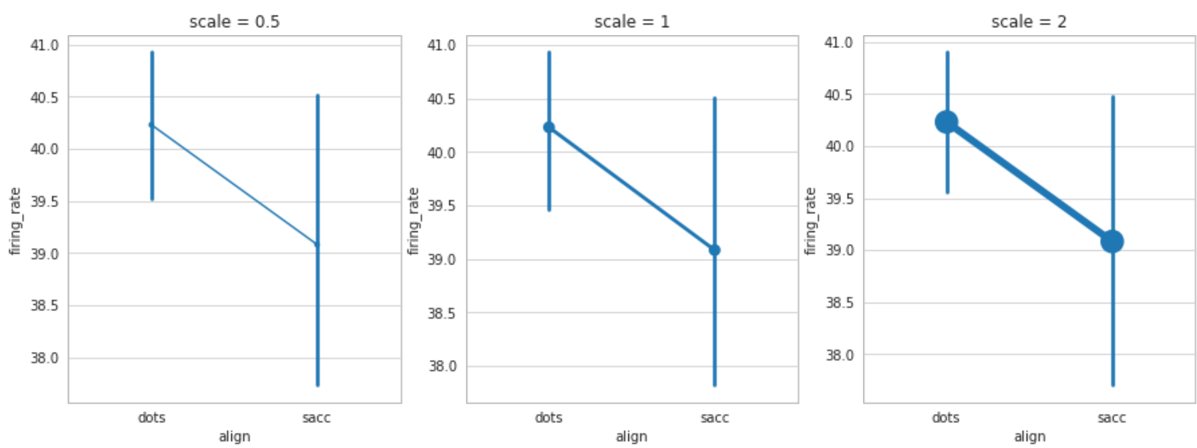


Рисунок 9.50 — Демонстрация работы с параметром `scale` функции `pointplot()`

9.4.2 Функция barplot()

Функция `barplot()` строит столбчатую диаграмму: высота бара (столбца) определяет численное значение оценки признака (математическое ожидание, медиана и т.п.), линия, пересекающая в верхнюю границу бара — доверительный интервал.

Загрузим набор данных *mpg* для исследования возможностей функции `barplot()`:

```
mpg = sns.load_dataset("mpg")
```

Построим диаграмму оценки мощности автомобилей в зависимости от страны производителя:

```
sns.barplot(x='origin', y='horsepower', data=mpg)
```

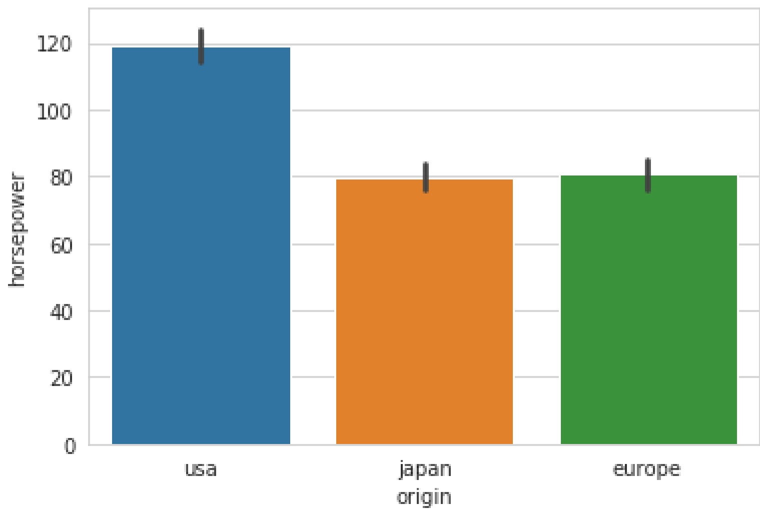


Рисунок 9.51 — Диаграмма barplot

Доверительный интервал изображается в виде линий темно-серого цвета в верхней части каждого бара. Для задания единого цвета всем барам можно воспользоваться параметром color:

```
sns.barplot(x='origin', y='horsepower', color='orange', data=mpg)
```

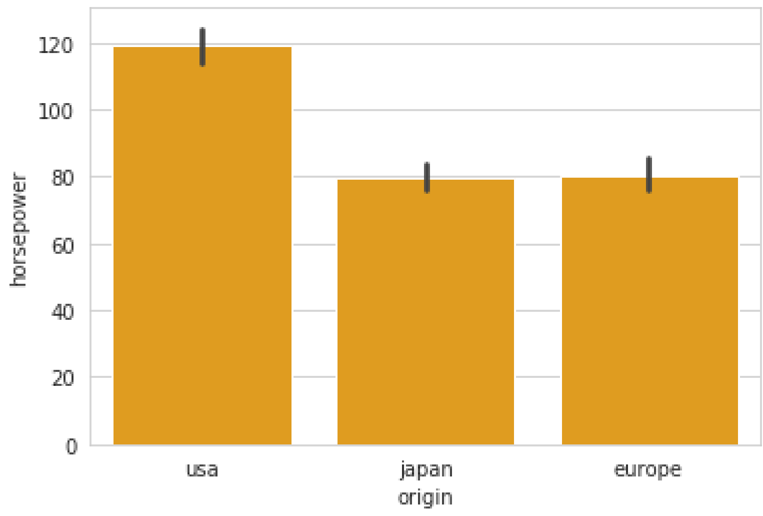


Рисунок 9.51 — Демонстрация работы с параметром color функции barplot()

Через аргумент hue можно задать ещё один уровень разделения данных, воспользуемся этим для дополнительной сегментации по количеству цилидров в двигателе автомобиля:

```
sns.barplot(x='origin', y='horsepower', hue='cylinders', data=mpg)
```

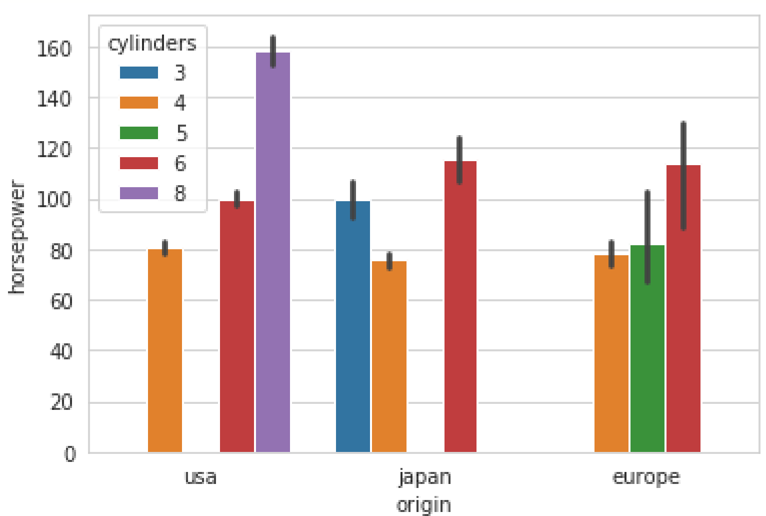


Рисунок 9.52 — Демонстрация работы с параметром `hue` функции `barplot()`

Изменим цветовую палитру:

```
sns.barplot(x='origin', y='horsepower', hue='cylinders', palette='Set2',
data=mpg)
```

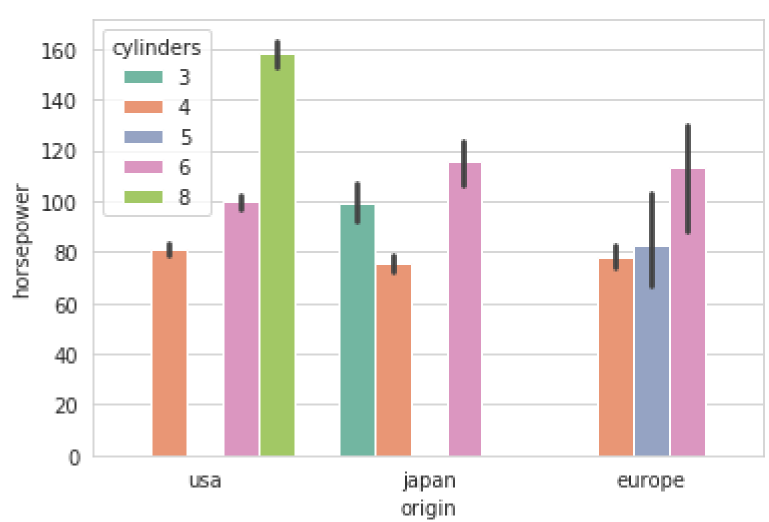


Рисунок 9.53 — Демонстрация работы с параметром `palette` функции `barplot()`

Зададим свой порядок вывода значений категориального признака *origin*:

```
order=['europe', 'usa', 'japan']
sns.barplot(x='origin', y='horsepower', hue='cylinders', palette='Set2',
order=order, data=mpg)
```

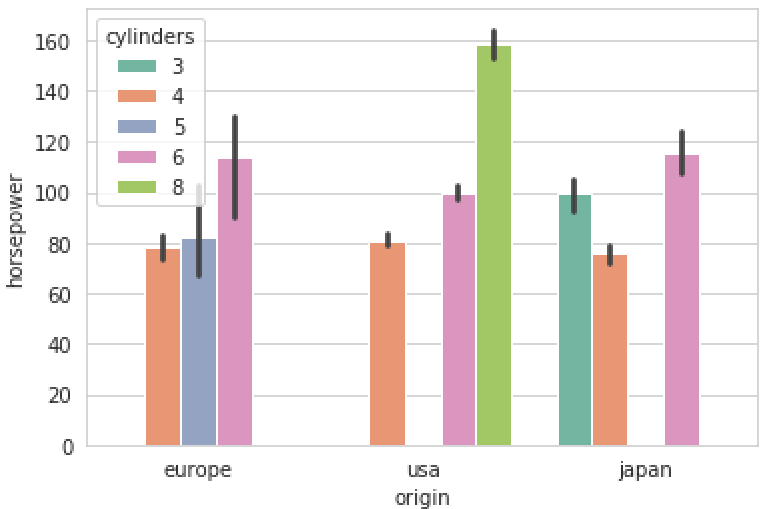


Рисунок 9.54 — Демонстрация работы с параметром order функции barplot()

Построим диаграммы с различными способами расчёта оценки:

```
estimator = [mean, median, min, max]


plt.figure(figsize=(15, 5))
for i, es in enumerate(estimator):
    plt.subplot(1, len(estimator), i+1)
    plt.title(es.__name__)
    sns.barplot(x='origin', y='horsepower', estimator=es, data=mpg)
```

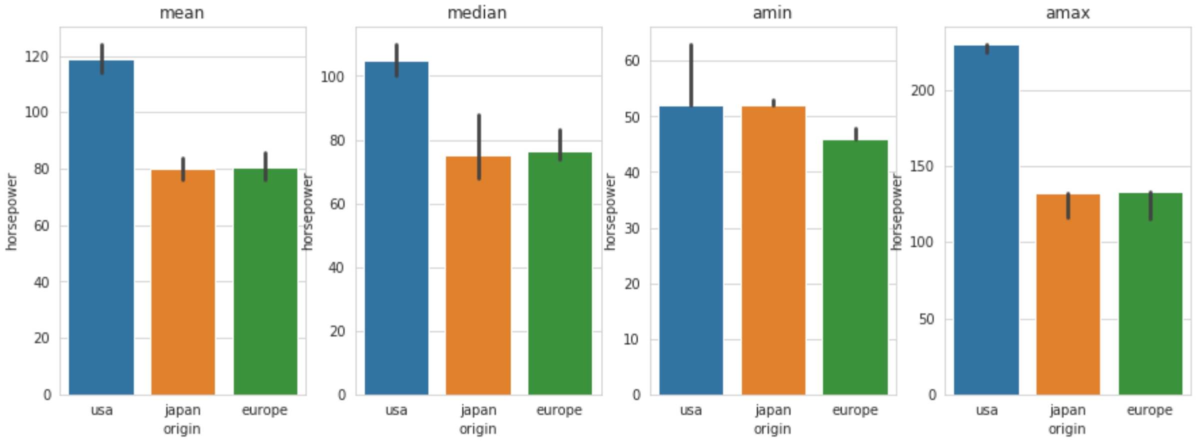


Рисунок 9.55 — Демонстрация работы с параметром `estimator` функции `barplot()`

Задание доверительного интервала (или стандартного отклонения) производится через параметр `ci`:

```
cis = [None, 80, 'sd']
plt.figure(figsize=(15, 5))
for i, c in enumerate(cis):
    plt.subplot(1, len(cis), i+1)
    plt.title(f"ci = {c}")
    sns.barplot(x='origin', y='horsepower', ci=c, data=mpg)
```

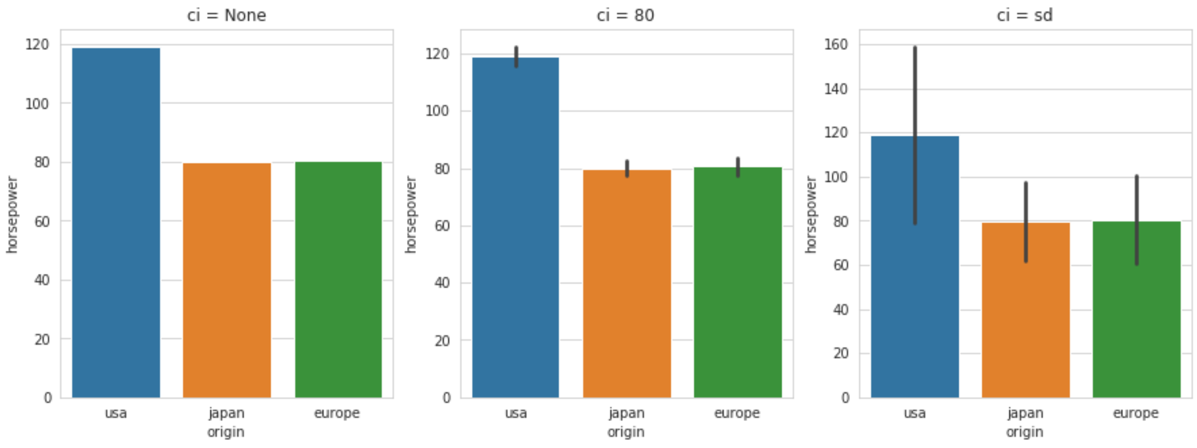


Рисунок 9.56 — Демонстрация работы с параметром `ci` функции `barplot()`

Параметры `errwidth`, `capsize` (см. "*9.4.1 Функция* `pointplot()`") и `errcolor` отвечают за настройку свойств линии. С первыми двумя мы уже знакомы, определимся с `errcolor`:

- `errcolor`: *Matplotlib*-цвет
    - Цвет линии, обозначающей доверительный интервал.

Ниже представлен пример работы с этими параметрами:

```
errcolor = ['r', 'b', 'y']
errwidthes = [2, 3, 4]
capsizes = [0, 0.3, 0.7]
plt.figure(figsize=(15, 5))
for i, p in enumerate(zip(errcolor, errwidthes, capsizes)):
    plt.subplot(1, len(errcolor), i+1)
    plt.title(f"errcolor = {p[0]}, errwidth = {p[1]}, capsize = {p[2]}")
    sns.barplot(x='origin', y='horsepower', errcolor=p[0], errwidth=p[1],
capsize=p[2], data=mpg)
```

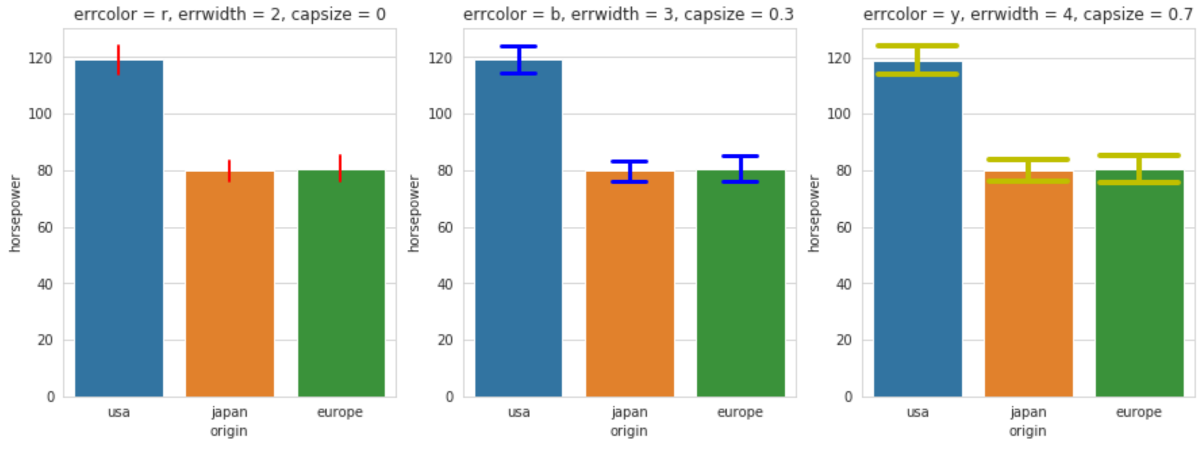


Рисунок 9.57 — Демонстрация работы с параметрами `errwidth`, `capsize` и `errcolor` функции `barplot()`

9.4.3 Функция countplot()

Функция `countplot()` определяет количество элементов из набора данных, которые относятся к той или иной категории, и отображает полученное значение в виде столбчатой диаграммы.

Для демонстрации работы с `countplot()` воспользуемся набором данных *tips*:

```
tips = sns.load_dataset("tips")
```

Построим диаграмму распределения, в качестве признака для оси *x* укажем *day*:

```
sns.countplot(x="day", data=tips)
```

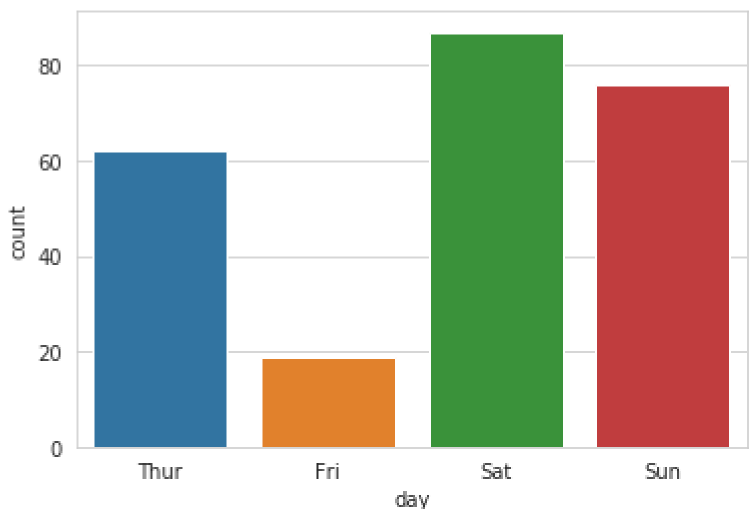


Рисунок 9.58 — Диаграмма countplot()

Зададим для всех баров зелёный цвет:

```
sns.countplot(x="day", color="g", data=tips)
```

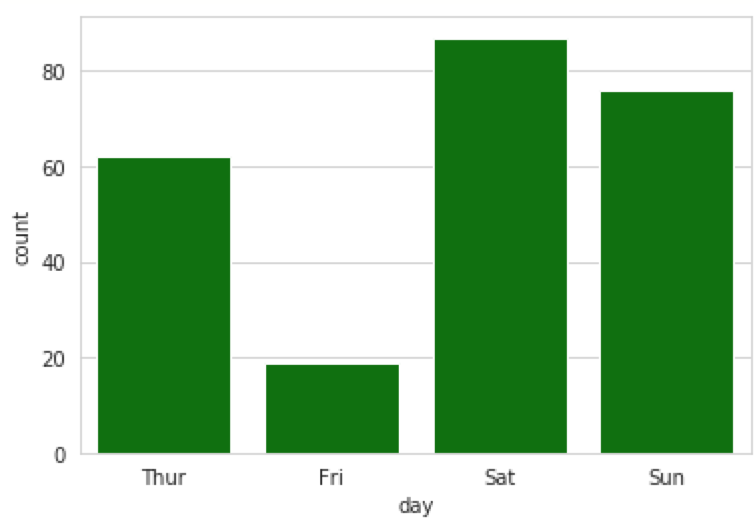


Рисунок 9.59 — Демонстрация работы с параметром color функции countplot()

Введём ещё один параметр — пол человека:

```
sns.countplot(x='day', hue='sex', data=tips)
```

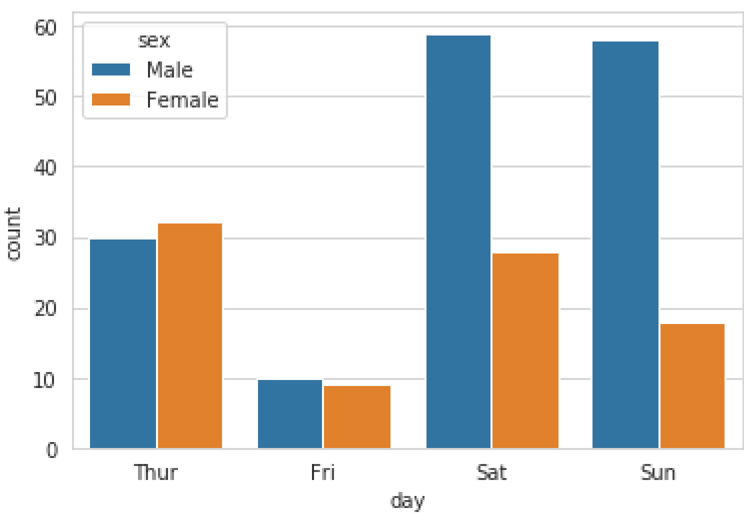


Рисунок 9.60 — Демонстрация работы с параметром `hue` функции `countplot()`

Изменим цветовую палитру:

```
sns.countplot(x='day', hue='sex', palette="PuOr", data=tips)
```

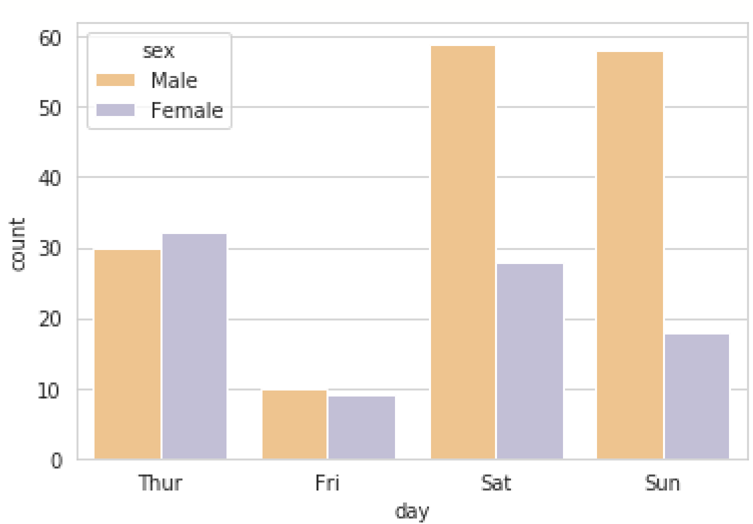


Рисунок 9.61 — Демонстрация работы с параметром `palette` функции `countplot()`

У функции `countplot()` есть параметр `saturation`, он отвечает за насыщенность выбранной цветовой гаммы:

```
plt.figure(figsize=(15, 5))
s_list = [0.25, 0.5, 0.75, 1]
for i, s in enumerate(s_list):
    plt.subplot(1, len(s_list), i+1)
    plt.title(f"saturation = {s}")
    sns.countplot(x='day', hue='sex', saturation=s, palette="Set1", 
data=tips)
```

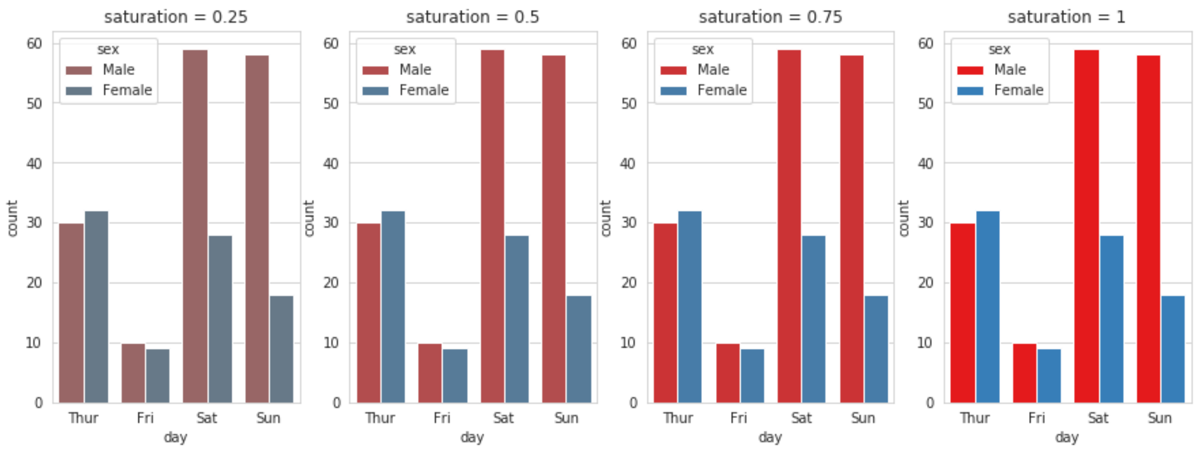


Рисунок 9.62 — Демонстрация работы с параметром `saturation` функции `countplot()`

9.5 Работа на уровне фигуры. Функция `catplot()`

По функциональным возможностям и назначению функция `catplot()` подобна функции `relplot()` из набора инструментов для визуализации отношений в данных (см. раздел "*8.5 Визуализация отношений с настройкой подложки. Функция* `relplot()`"). Её особенность заключается в том, что она предоставляет общий интерфейс для всех функций из группы визуализации категориальных данных, также она позволяет работать с параметрами уровня фигуры, которые управляют

размещением диаграмм. Уровень фигуры в данном случае определяется объектом класса `FacetGrid`, через который происходит непосредственно управление размещением.

Познакомимся с этим инструментом поближе, предварительно зададим стиль и контекст:

```
sns.set_style("whitegrid")
sns.set_context("notebook")
```

Загрузим набор данных *tips* для экспериментов:

```
tips = sns.load_dataset("tips")
```

Построим точечную диаграмму распределения значений категориальной переменной *day*:

```
sns.catplot(x='day', y='total_bill', kind='strip', data=tips)
```

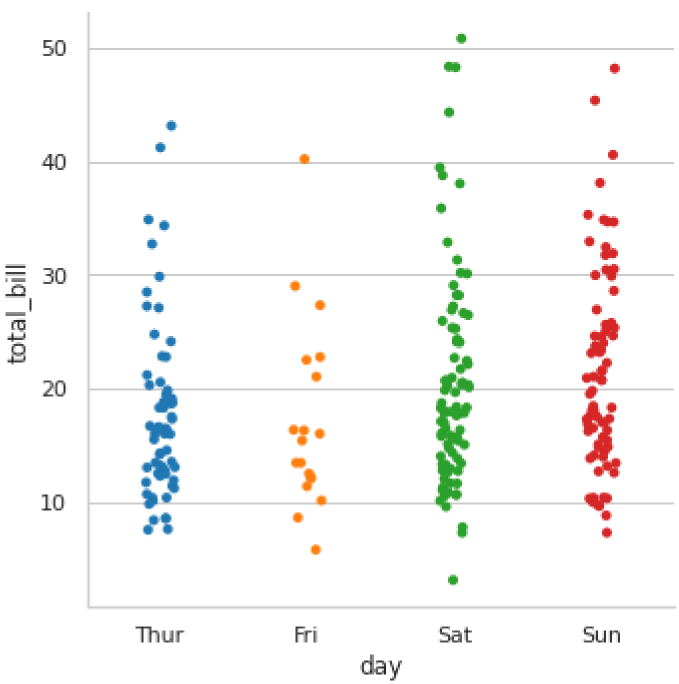


Рисунок 9.63 — Демонстрация работы функции `catplot()`

Как вы можете видеть из рисунка, мы получили диаграмму, аналогичную той, что создаётся функцией `stripplot()`. Параметром `catplot()`, через который задаётся тип диаграммы, является `kind`:

- `kind`: `str`, `optional`
  - Тип диаграммы. В таблице 9.5 приведено соответствие значения параметра `kind` и функции из набора инструментов для визуализации категориальных данных. Значение по умолчанию *'strip'*.

Таблица 9.4 — Соответствие значений параметра `kind` и функций *seaborn*

Значение параметра `kind`	**Функция *seaborn***
strip	`stripplot()`
swarm	`swarmplot()`
box	`boxplot()`
violin	`violinplot()`
boxen	`boxenplot()`
point	`pointplot()`
bar	`barplot()`
count	`countplot()`

Назначение параметров `x`, `y`, `data`, `hue`, `estimator`, `ci`, `order`, `order_hue`, `orient`, `color` и `palette` аналогично одноимённым из рассмотренных ранее функций визуализации категориальных данных. На них мы не будем останавливаться. Перейдём непосредственно к

аргументам, через которые мы можем управлять расположением диаграмм. Начнем с `col` и `row`:

- `row`, `col`: имена переменных из набора `data`, `optional`
  - Категориальные признаки, по которым будет производиться распределение на строки и столбцы.

Построим диаграммы распределения количества оставленных чаевых от дня недели, с разделением на столбцы:

```
sns.catplot(x='day', y='total_bill', col='sex', kind='strip', data=tips)
```

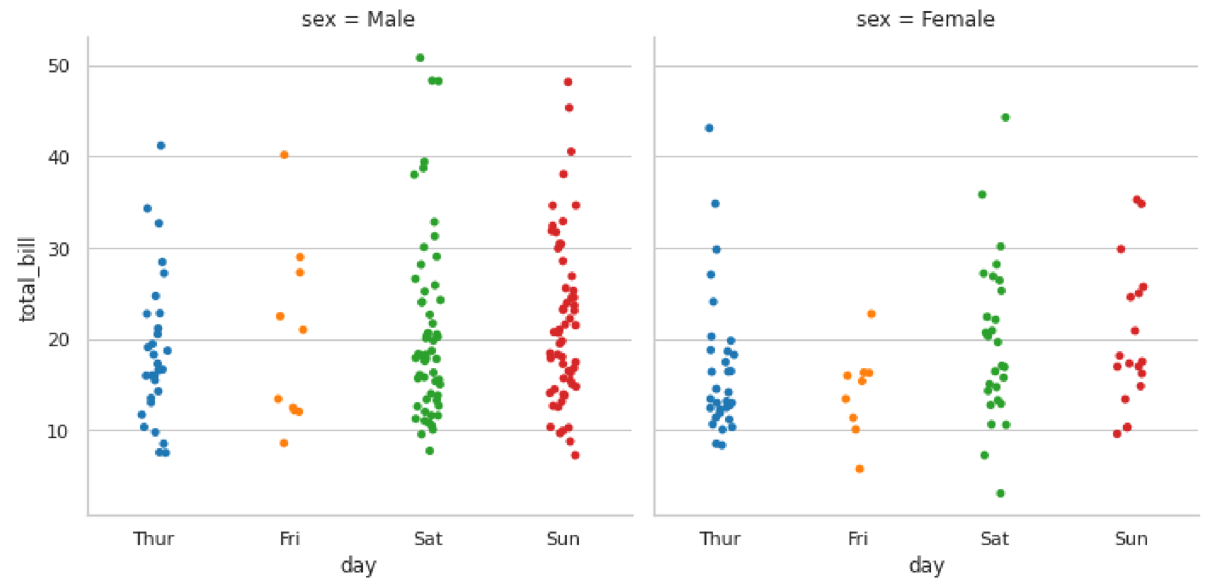


Рисунок 9.64 — Демонстрация работы с параметром `col` функции `catplot()`

Изменим этот пример так, чтобы разделение производилось по строкам:

```
sns.catplot(x='day', y='total_bill', row='sex', kind='strip', data=tips)
```

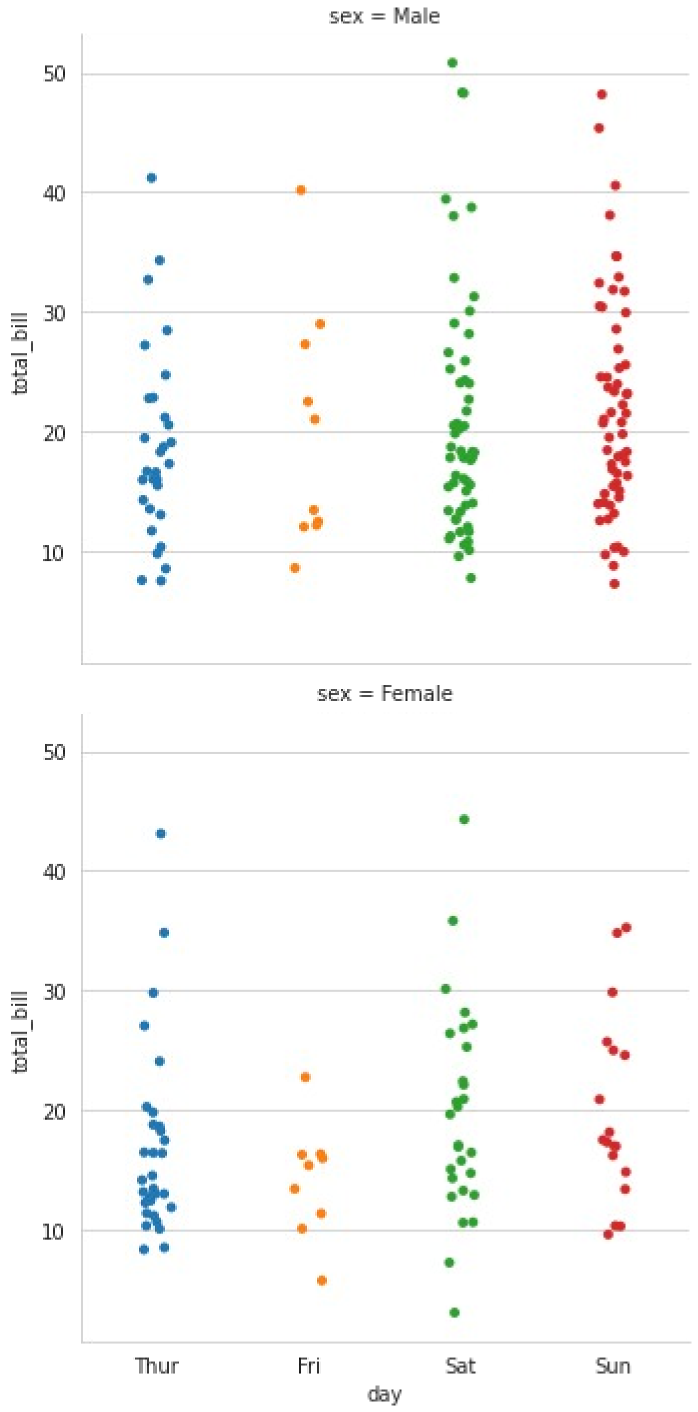


Рисунок 9.65 — Демонстрация работы с параметром `row` функции `catplot()`

Для управления размером диаграмм и соотношением сторон используйте параметры `height` и `aspect`:

- `height`: число, `optional`
  - Высота диаграммы. Задаётся в дюймах.

- `aspect`: число, `optional`
    - Коэффициент, задающий соотношение сторон, ширина диаграммы вычисляется следующим образом: `aspect * height`

Приведём несколько примеров, демонстрирующих работу с `height` и `aspect`:

```
sns.catplot(x='day', y='total_bill', col='sex', height=5, aspect=0.5,
kind='strip', data=tips)
```

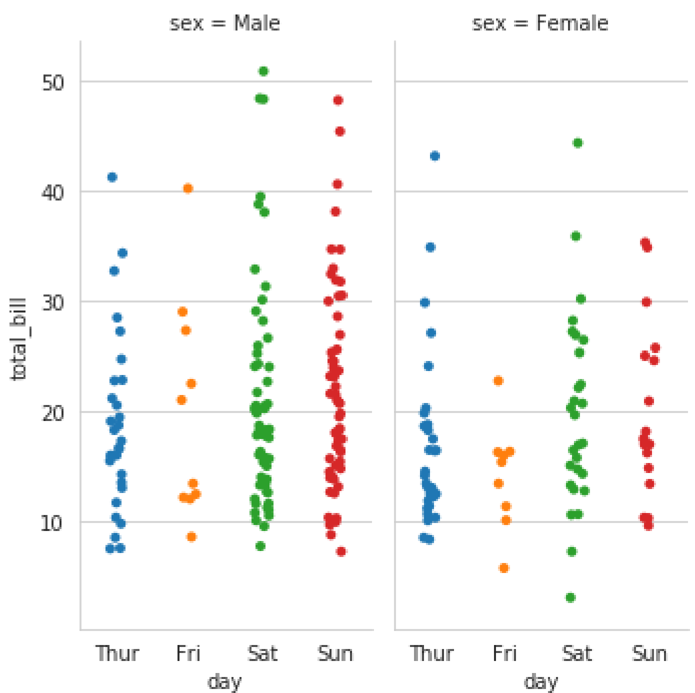


Рисунок 9.66 — Демонстрация работы с параметрами `height` и `aspect` функции `catplot()` (пример 1)

```
sns.catplot(x='day', y='total_bill', col='sex', height=5, aspect=1.5,
kind='strip', data=tips)
```

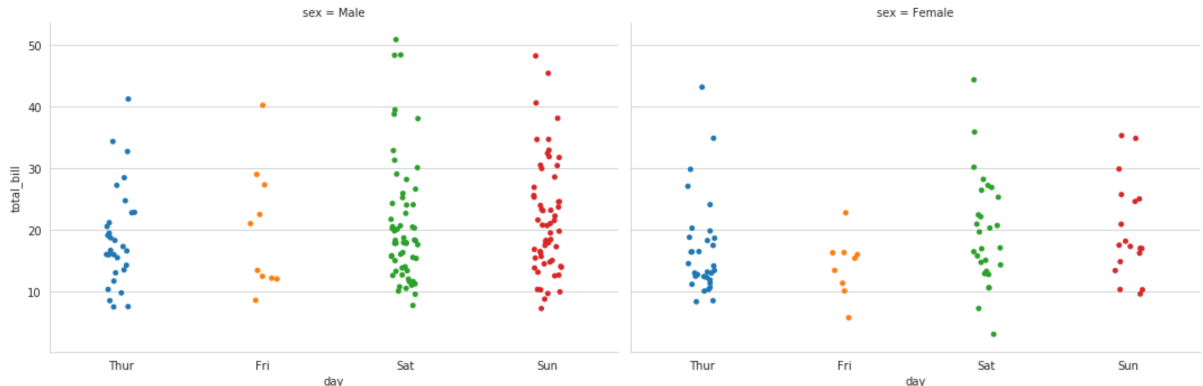


Рисунок 9.67 — Демонстрация работы с параметрами `height` и `aspect` функции `catplot()` (пример 1)

Управление расположением легенды осуществляется через `legend` и `legend_out`:

- `legend`: `bool`, `optional`
  - Легенда будет отображена, если параметр равен `True`, в противном случае её на диаграмме не будет.
- `legend_out`: `bool`, `optional`
  - Расположение легенды: если параметр равен `True`, то легенда будет находиться вне диаграмм (в правой части фигуры), `False` — будет отображена непосредственно на одной из диаграмм.

Поместим легенду на диаграмму:

```
sns.catplot(x='day', y='total_bill', col='sex', hue='smoker',
legend_out=False, kind='strip', data=tips)
```

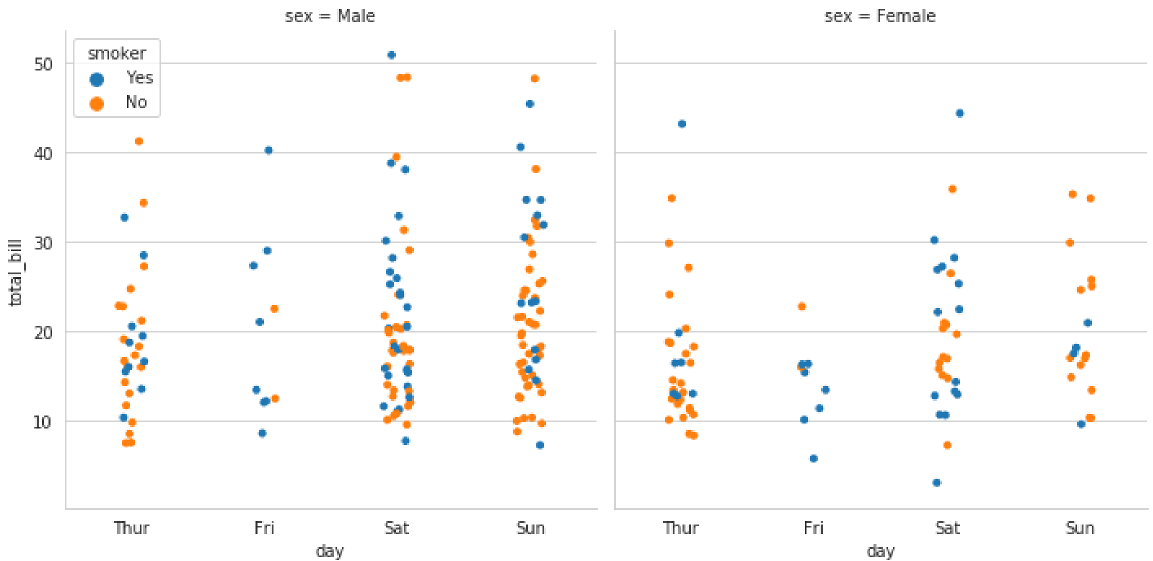


Рисунок 9.68 — Демонстрация работы с параметром `legend_out` функции `catplot()`

Уберём легенду с фигуры:

```
sns.catplot(x='day', y='total_bill', col='sex', hue='smoker',
legend=False, kind='strip', data=tips)
```

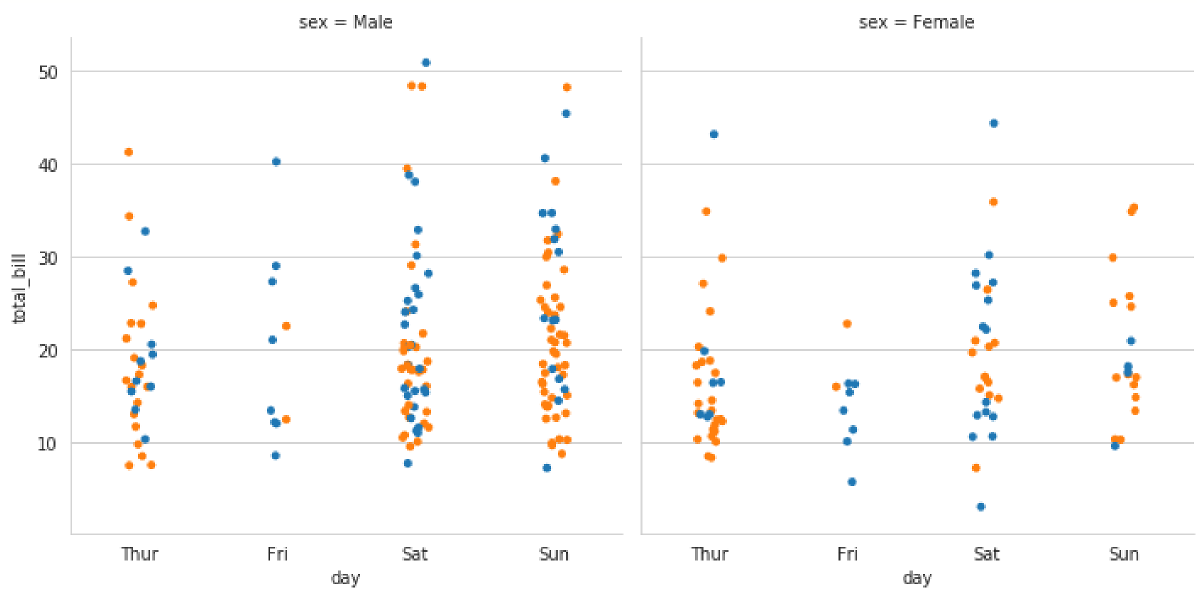


Рисунок 9.69 — Демонстрация работы с параметром `legend` функции `catplot()`

Поместим легенду вне поля диаграмм:

```
sns.catplot(x='day', y='total_bill', col='sex', hue='smoker',
kind='strip', data=tips)
```

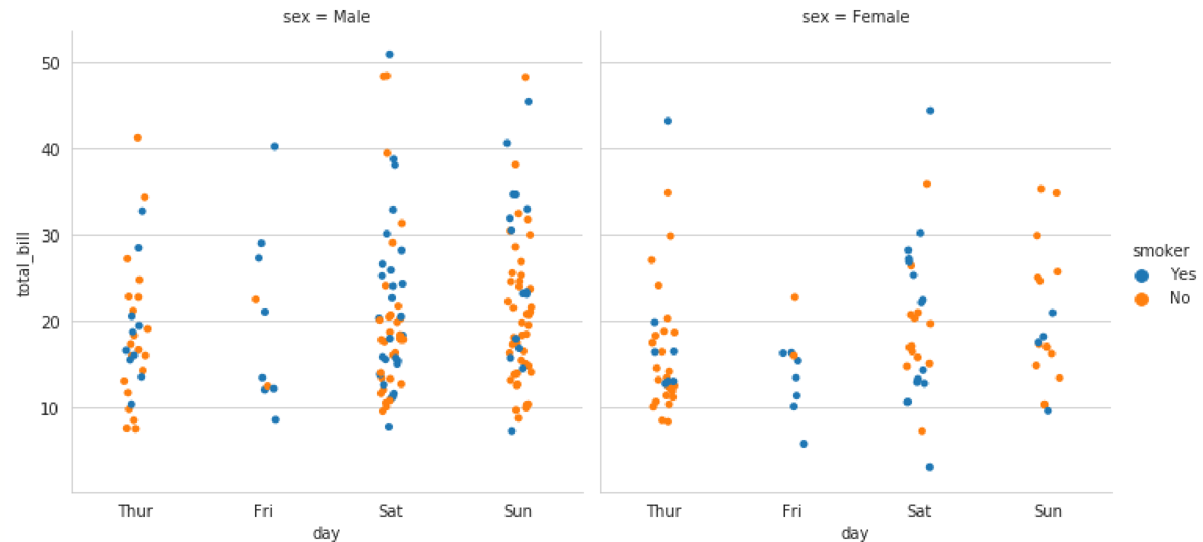


Рисунок 9.70 — Демонстрация размещения легенды вне поля графика

Диаграмма с `margin_titles=False` представлена на рисунке 9.71:

```
sns.catplot(x='day', y='total_bill', col='sex', row='smoker',
margin_titles=False, height=3, kind='strip', data=tips)
```

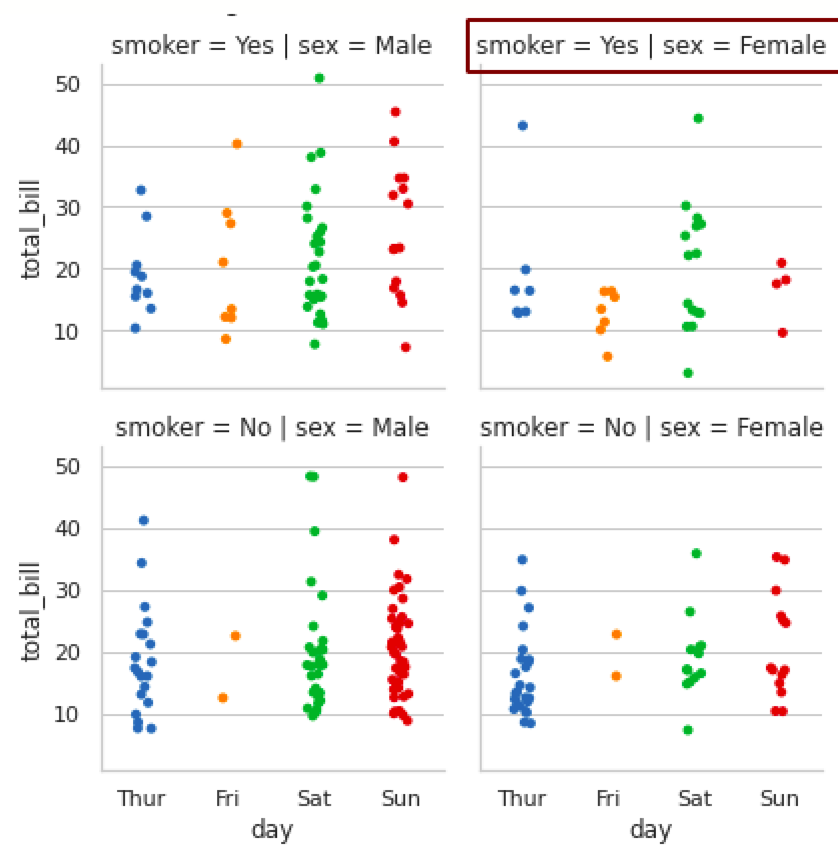


Рисунок 9.71 — Демонстрация работы с параметром `margin_titles=False` функции `catplot()`

Сравните диаграмму с рисунка 9.71 с вариантом, когда `margin_titles=True`:

```
sns.catplot(x='day', y='total_bill', col='sex', row='smoker', height=3,
margin_titles=True, kind='strip', data=tips)
```

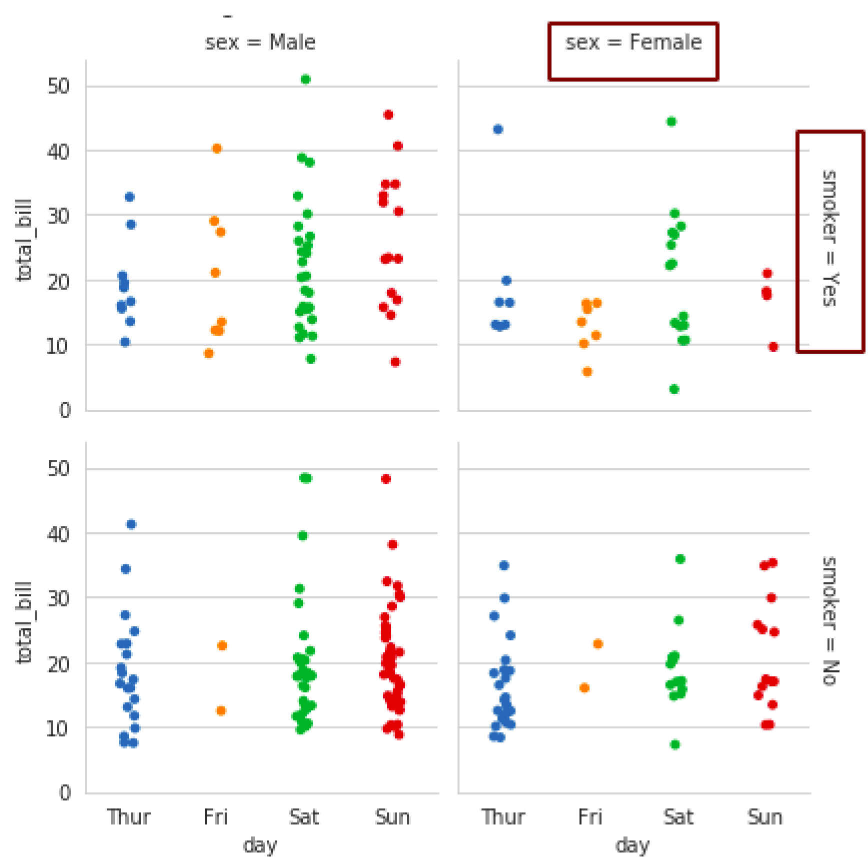


Рисунок 9.72 — Демонстрация работы с параметром `margin_titles=True` функции `catplot()`

Для более тонкой настройки фигуры можно воспользоваться параметром `facet_kws`, через него передаются аргументы конструктора класса `FacetGrid` в виде словаря, ключами которого являются имена аргументов.

Глава 10. Визуализация распределений в данных

Набор инструментов для визуализации распределений в данных состоит из трёх функций, представленных в таблице 10.1.

Таблица 10.1 — Функции для визуализации распределений в данных

Функция	**Описание**
`distplot()`	Диаграмма, состоящая из результатов работы функций: `hist()`, `kdeplot() и rugplot()`.
`kdeplot()`	Отображает одномерную либо двумерную ядерную оценку плотности.
`rugplot()`	Отображает элементы данных в виде линии рядом с осью координат.

10.1 Функция distplot()

Функция `distplot()` предназначена для визуализации распределений одномерных наборов данных. Диаграмма, которую она строит, может состоять из следующих компонентов:

- результат работы функции `hist()` из *Matplotlib*;
- результат работы функции `kdeplot()` из *seaborn*;
- результат работы функции `rugplot()` из *seaborn.*

Эти компоненты являются опциональными: вы можете выбрать, какой набор функций использовать.

Рассмотрим наиболее важные параметры функции `distplot()`:

- a: `Series`, *1D*-массив или список.
    - Набор данных для построения диаграммы может быть структурой `Series`, либо одномерным массивом.

- `bins`: `int`, список, `str`, `None`, `optional`
  - Количество бинов для функции `hist()` библиотеки *Matplotlib*.
- `hist`: `bool`, `optional`
  - Если параметр равен `True`, то гистограмма будет отображена на поле графика, если равен `False`, то нет.
- `kde`: `bool`, `optional`
  - Если параметр равен `True`, то диаграмма KDE (результат работы функции `kdeplot()`) будет отображена на поле графика, если равен `False`, то нет.
- `rug`: `bool`, `optional`
  - Если параметр равен `True`, то результат работы функции `rugplot()` будет отображён на поле графика, если равен `False`, то нет.

Построим набор данных для экспериментов и визуализируем его:

```
np.random.seed(123)
x = np.random.chisquare(2,500)
sns.distplot(x)
```

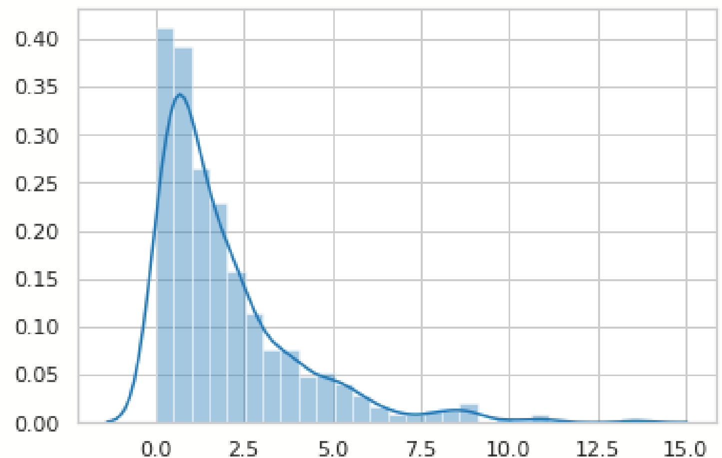


Рисунок 10.1 — Демонстрация работы функции `distplot()`

Построим `pandas.Series` на базе *x* и передадим полученный объект в `distplot()`:

```
s = pd.Series(x)
sns.distplot(s)
```

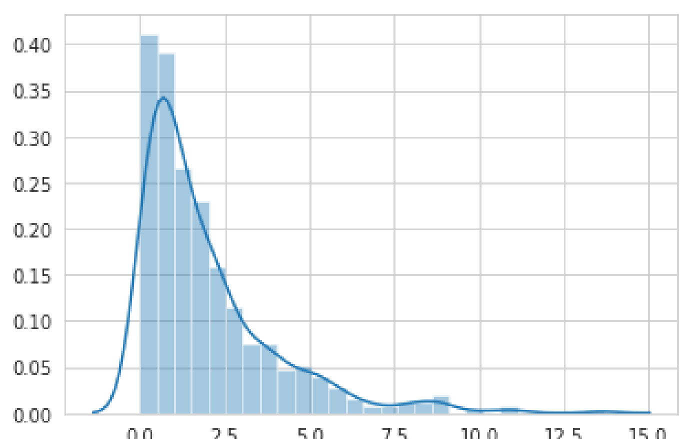


Рисунок 10.2 — Демонстрация работы функции `distplot()` с аргументом типа `pandas.Series`

Продемонстрируем как различные значения параметра `bins` влияют на внешний вид диаграммы:

```
bs = [None, 1, 3, 5, 15]
plt.figure(figsize=(15, 5))
for i, b in enumerate(bs):
    plt.subplot(1, len(bs), i+1)
    plt.title(f"bins = {b}")
    sns.distplot(s, bins=b)
```

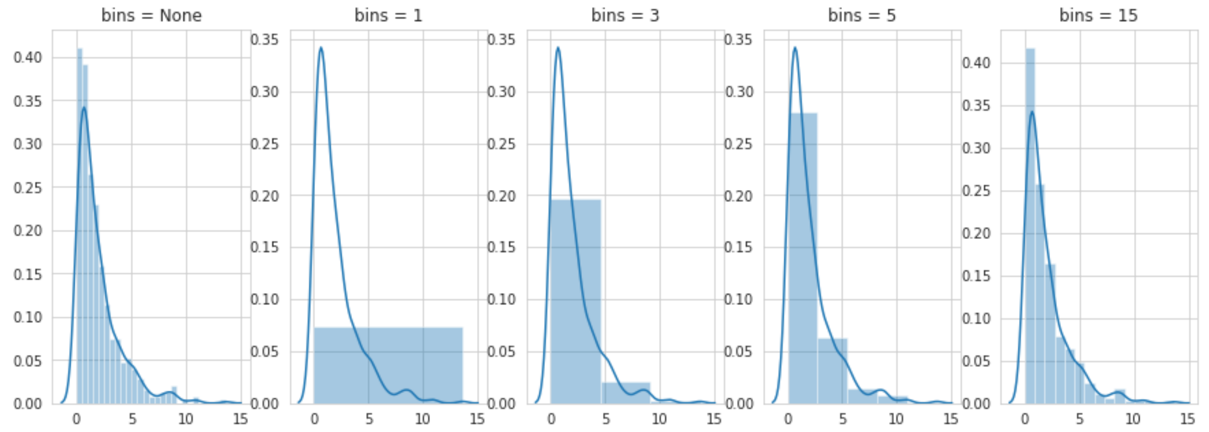


Рисунок 10.3 — Демонстрация работы с параметром `bins` функции `distplot()`

Для того чтобы убрать с диаграммы гистограмму, присвойте параметру `hist` значение `False`:

```
sns.distplot(s, hist=False)
```

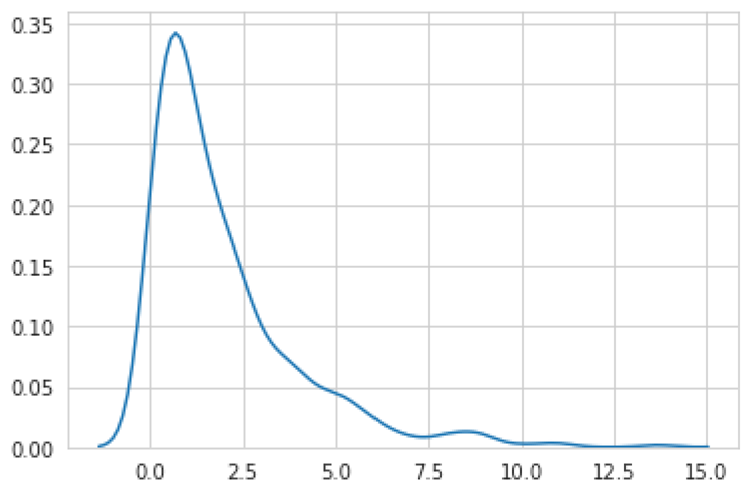


Рисунок 10.4 — Диаграмма без гистограммы

За отображение графика ядерной оценки плотности отвечает параметр `kde`:

```
sns.distplot(s, kde=False)
```

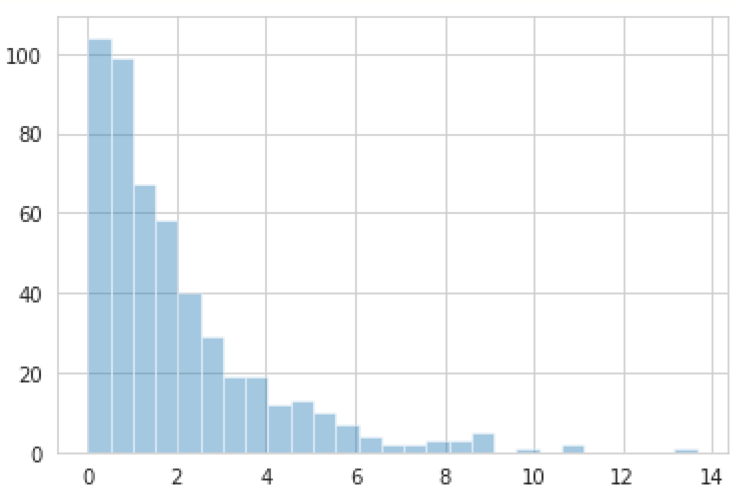


Рисунок 10.5 — Диаграмма без kde

Добавим на диаграмму результат работы `rugplot()`:

```
sns.distplot(s, rug=True)
```

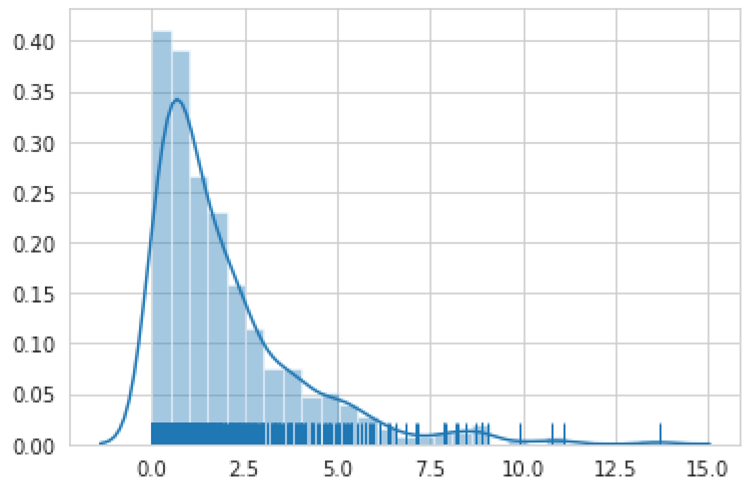


Рисунок 10.6 — Диаграмма с добавлением распределения

Для дополнительной настройки представления можно воспользоваться следующими параметрами:

- `color`: *Matplotlib*-цвет, `optional`
  - Цвет элементов диаграммы. Для разных элементов будут выбраны разные оттенки.
- `vertical`: `bool`, `optional`
  - Ориентация графиков: `True` — вертикальная, `False` (или `None`) — горизонтальная.
- `norm_hist`: `bool`, `optional`
  - Если значение равно `True`, то при построении гистограммы высота столбцов будет обозначать величину плотности, а не количество элементов в наборе данных.
- `axlabel`: `str`, `False`, `None`, `optional`
  - Имя оси абсцисс (ось x). Если значение равно `None`, то будет взято значение параметра `Name` структуры, переданной через аргумент `a` (если таковой у неё есть), иначе подпись не будет поставлена.

- `label`: `string`, `optional`
    - Метка для легенды.

Поработаем с этими параметрами на практике. Изменим цвет графиков на зелёный:

```
sns.distplot(s, rug=True, color="g")
```

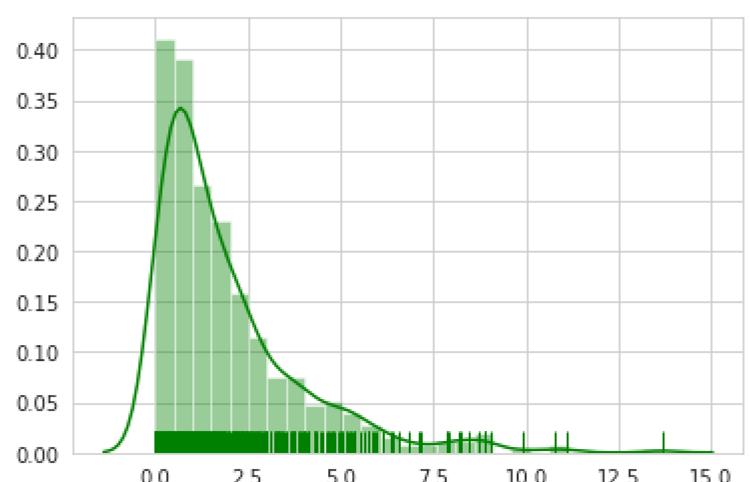


Рисунок 10.7 — Демонстрация работы функции `distplot()` с параметром `color`

Изменим ориентацию:

```
sns.distplot(s, rug=True, vertical=True)
```

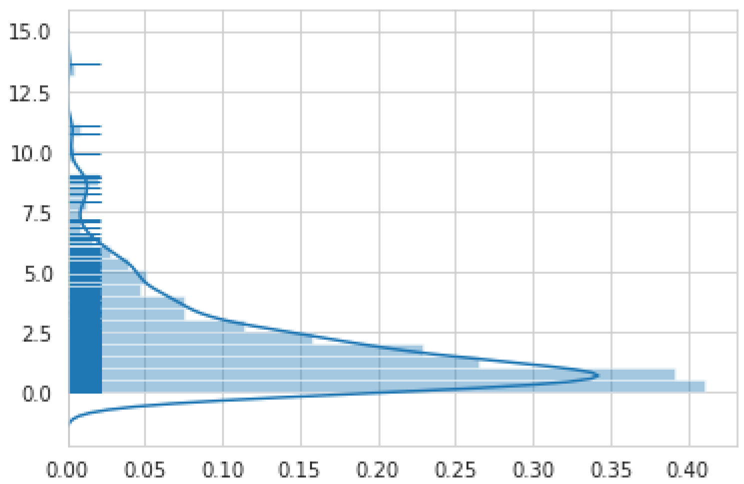


Рисунок 10.8 — Демонстрация работы функции `distplot()` с параметром `vertical`

Зададим подпись оси *x*:

```
sns.distplot(s, rug=True, axlabel="chi-square values")
```

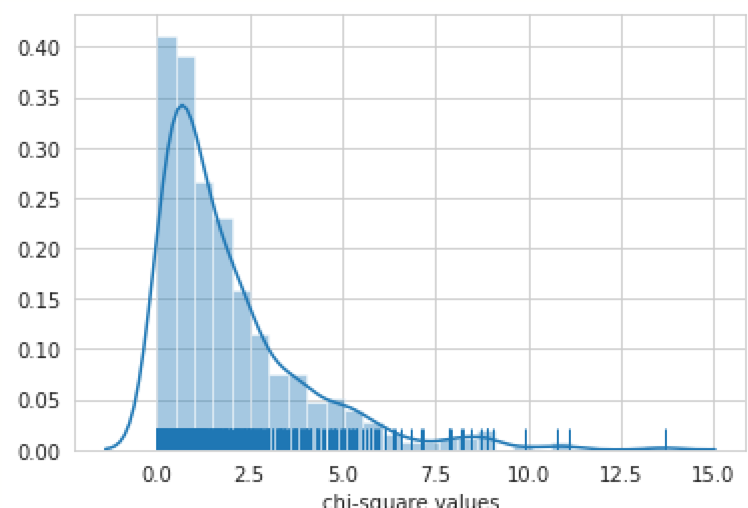


Рисунок 10.9 — Демонстрация работы функции `distplot()` с параметром `axlabel`

Для более тонкой настройки представления графиков используйте параметры `hist_kws`, `kde_kws` и `rug_kws`, через которые задаются параметры соответствующих функций:

- `hist_kws`: `dict`, `optional`
  - Аргументы функции `hist()`.
- `kde_kws`: `dict`, `optional`
  - Аргументы функции `kdeplot()`.
- `rug_kws`: `dict`, `optional`
  - Аргументы функции `rugplot()`.

Продемонстрируем работу с ними на примере:

```
h_kws={"alpha": 0.3, "color": "r"}
k_kws={"shade": True, "color": "g"}
r_kws={"height":0.1}
sns.distplot(s, bins=10, rug=True, hist_kws=h_kws, kde_kws=k_kws,
rug_kws=r_kws)
```

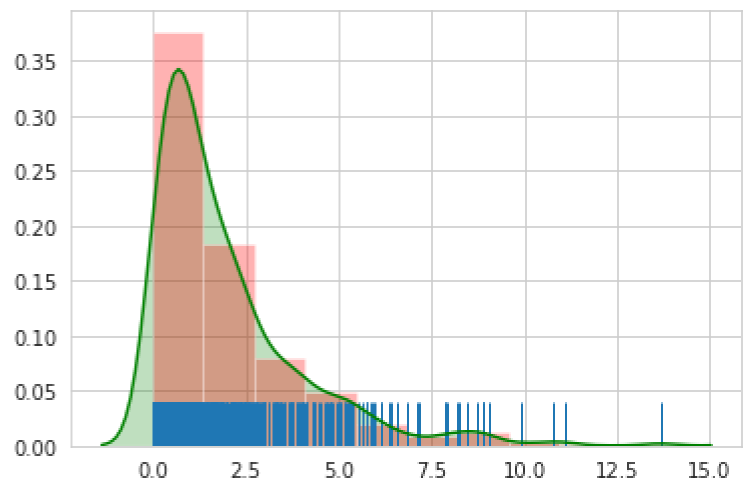


Рисунок 10.10 — Демонстрация работы с параметрами `hist_kws`, `kde_kws` **и** `rug_kws` **функции** `distplot()`

10.2 Функция `kdeplot()`

Функция `kdeplot()` отображает одномерную либо двумерную ядерную оценку плотности — *kernel density estimate* (далее *KDE*). Данные по которым строятся графики передаются через параметры `data` и `data2` (в случае двумерной оценки):

- `data`: одномерный массив
  - Набор данных для построения графика.
- `data2`: одномерный массив, `optional`
  - Дополнительный набор данных, если необходимо построить двумерную *KDE*.

Загрузим наборы данных *tips* и *mpg*:

```
tips = sns.load_dataset("tips")
mpg = sns.load_dataset("mpg")
```

Построим *KDE* для параметра *displacement*:

```
sns.kdeplot(mpg["displacement"])
```

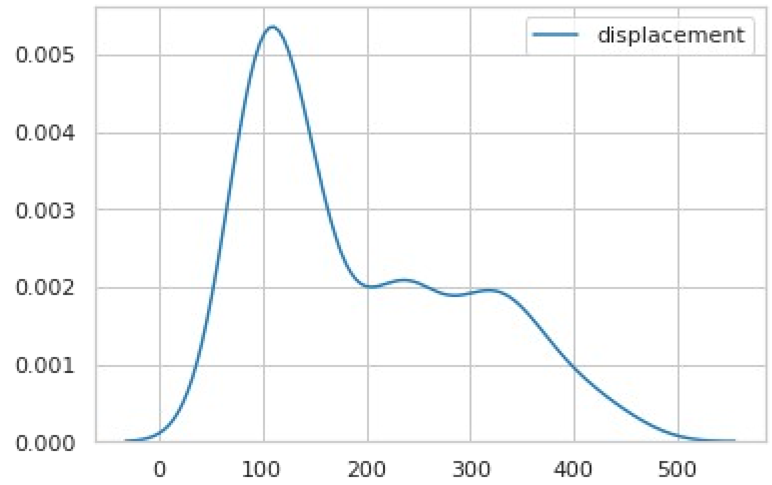


Рисунок 10.11 — Демонстрация работы функции `kdeplot()`. Одномерный вариант

Пример двумерной *KDE*:

```
sns.kdeplot(tips["total_bill"], tips["tip"])
```

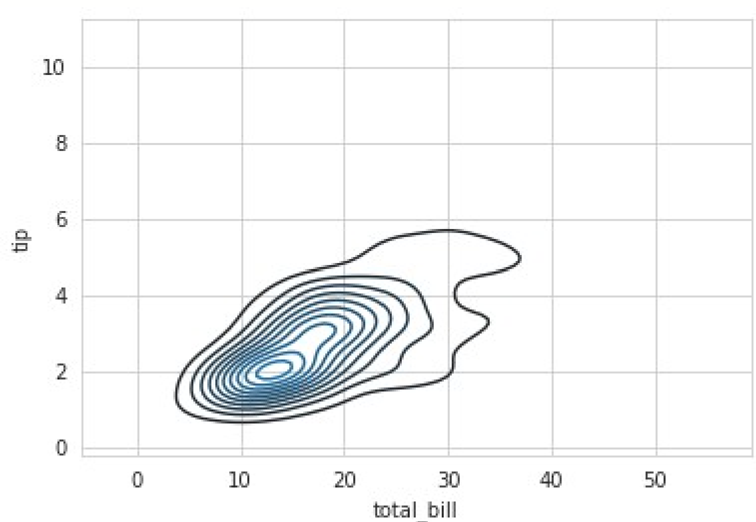


Рисунок 10.12 — Демонстрация работы функции `kdeplot()`. Двумерный вариант

Построим диаграмму, аналогичную приведённой на рисунке 10.12, для набора *mpg*:

```
x = mpg["cylinders"]
y = mpg["displacement"]
sns.kdeplot(x, y)
```

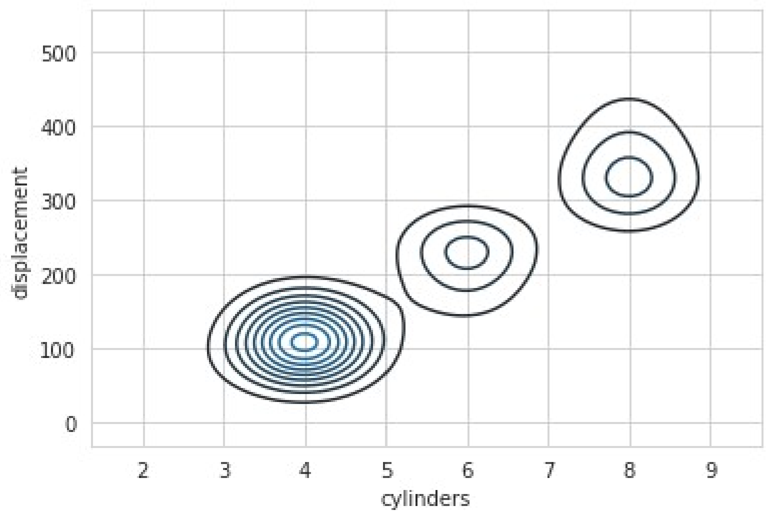


Рисунок 10.13 — Демонстрация работы функции `kdeplot()` на наборе данных *mpg*

Для настройки внешнего вида диаграммы можно воспользоваться следующими параметрами:

- `shade`: `bool`, `optional`
    - Определяет наличие (`True`) или отсутствие (`False`) заливки под кривой *KDE* для одномерного варианта или заливку контуров для двумерного варианта.
- `vertical`: `bool`, `optional`
    - Расположение графика. Если значение равно `True`, то численное значение параметра откладывается по оси *x*, *KDE* — по оси *y*, если `False`, то наоборот.
- `color`: *Matplotlib*-цвет, `optional`
    - Цвет графика и заливки.
- `cmap`: `str` или `Colormap`, `optional`
    - Цветовая карта для диаграммы (см. раздел "*4.4.1 Цветовые карты (colormaps)*".
- `legend`: `bool`, `optional`
    - Если параметр равен `True`, то легенда будет добавлена на диаграмму.

- `cbar`: `bool`, `optional`
  - Если параметр равен `True`, то на диаграмму будет добавлена цветовая полоса (используется для двумерной *KDE*).
- `shade_lowest`: `bool`, `optional`
  - Управляет заливкой. Если параметр равен `True`, то будет скрыта областью с наименьшим значением *KDE*. Используется только для двумерной *KDE.*
- `cumulative`: `bool`, `optional`
  - Если равно `True`, то будет отображена функция распределения.
- `gridsize`: `int`, `optional`
  - Количество точек в сетке.

Рассмотрим на практике работу с перечисленными выше параметрами. Отобразим заливку для одномерного варианта *KDE*:

```
sns.kdeplot(y, shade=True)
```

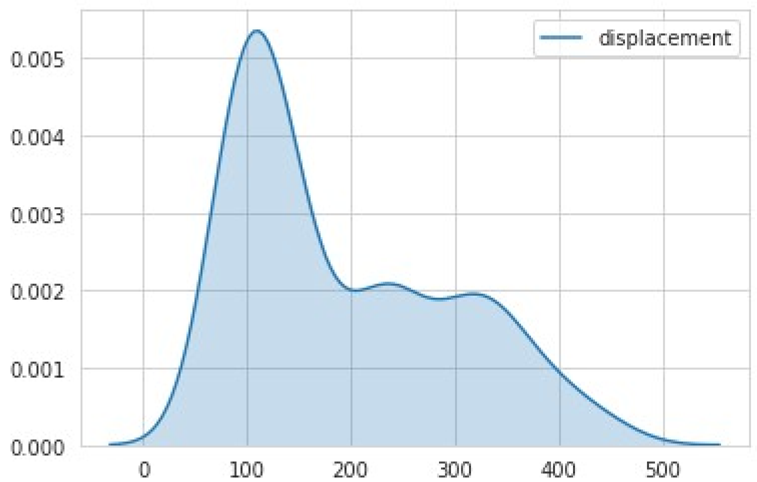


Рисунок 10.14 — Демонстрация работы с параметром `shade` функции `kdeplot()`

Уберём с поля легенду:

```
sns.kdeplot(y, shade=True, legend=False)
```

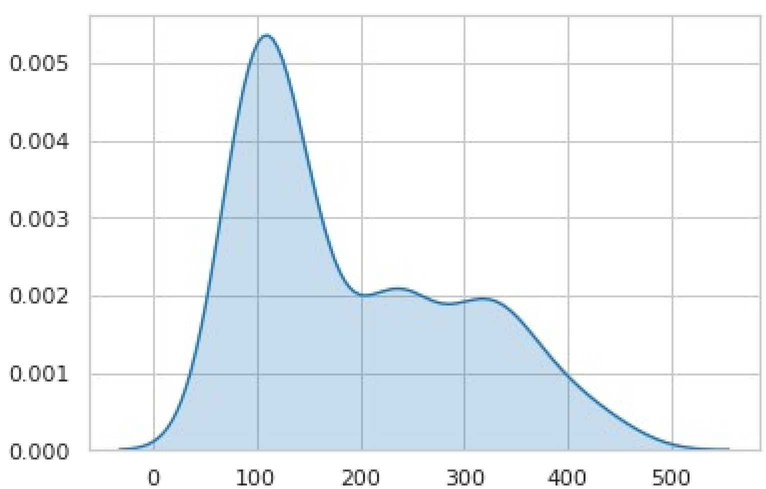


Рисунок 10.15 — Демонстрация работы с параметром `legend` функции `kdeplot()`

Изменим цвет:

```
sns.kdeplot(y, shade=True, color='r')
```

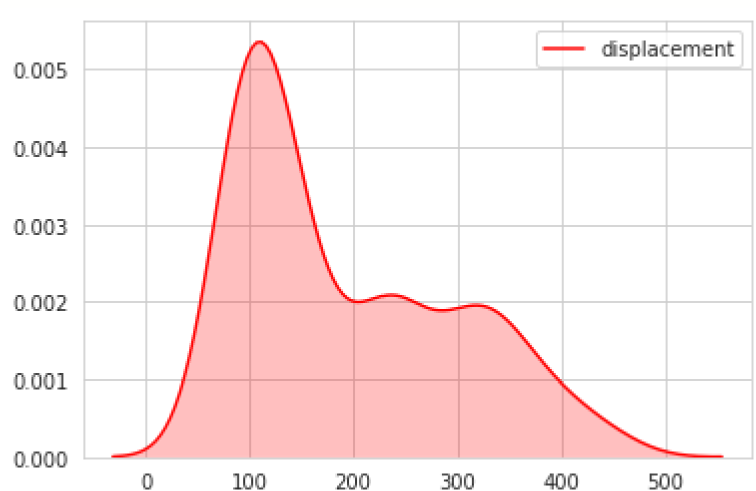


Рисунок 10.16 — Демонстрация работы с параметром `color` функции `kdeplot()`

Пример работы с заливкой для двумерного варианта:

```
sns.kdeplot(x, y, shade=True)
```

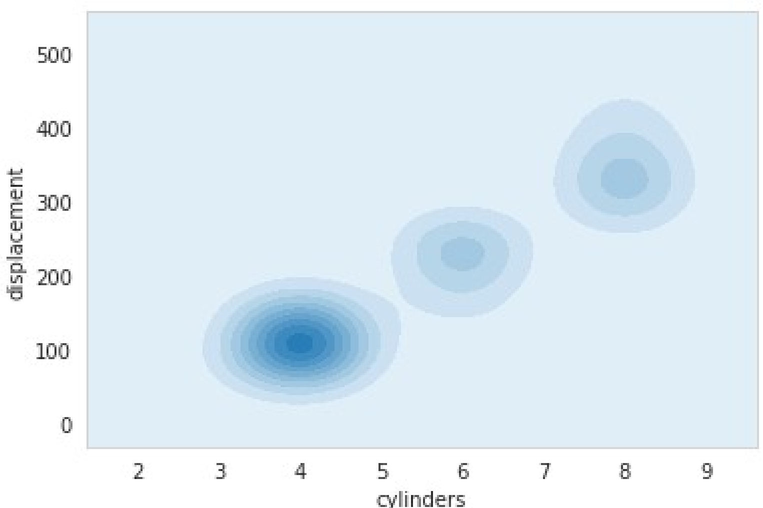


Рисунок 10.17 — Демонстрация работы с параметром `shade` функции `kdeplot()` (двумерный набор данных)

Зададим количество уровней для отображения:

```
sns.kdeplot(x, y, shade=True, n_levels=3)
```

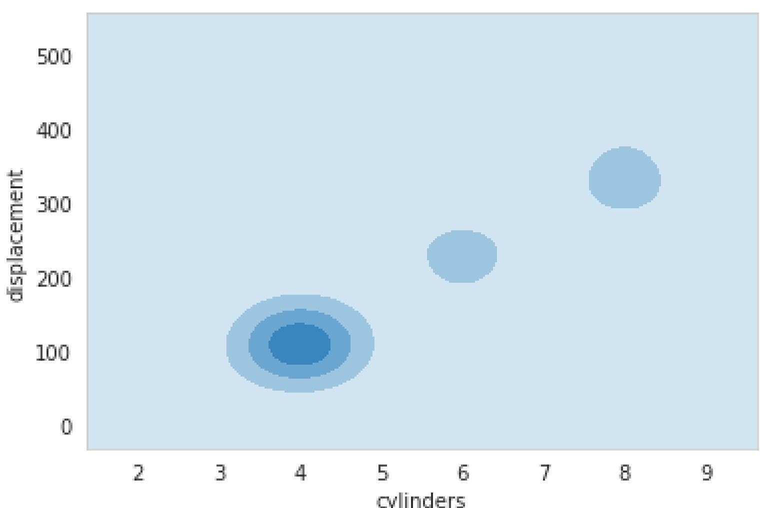


Рисунок 10.18 — Демонстрация работы с параметром `n_levels` функции `kdeplot()` (двумерный набор данных)

Для лучшего визуального представления изменим цветовую палитру на '*plasma*':

```
sns.kdeplot(x, y, shade=True, cmap='plasma')
```

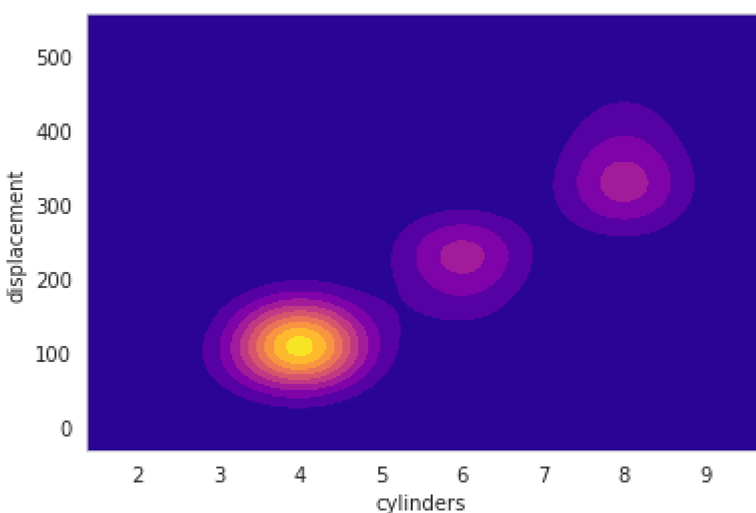


Рисунок 10.19 — Демонстрация работы с параметром `cmap` функции `kdeplot()` (двумерный набор данных)

Уберём с диаграммы заливку уровня с наименьшим значением *KDE* и добавим *colorbar*:

```
sns.kdeplot(x, y, shade=True, cmap='plasma', shade_lowest=False,
cbar=True)
```

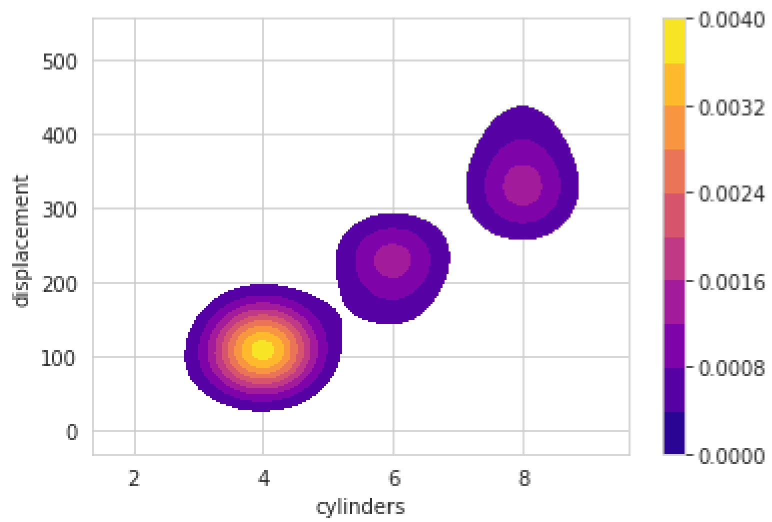


Рисунок 10.20 — Демонстрация работы с параметром `shade_lowest` и `cbar` функции `kdeplot()` (двумерный набор данных)

Ещё вариант палитры:

```
sns.kdeplot(x, y, shade=True, cmap='rainbow', cbar=True)
```

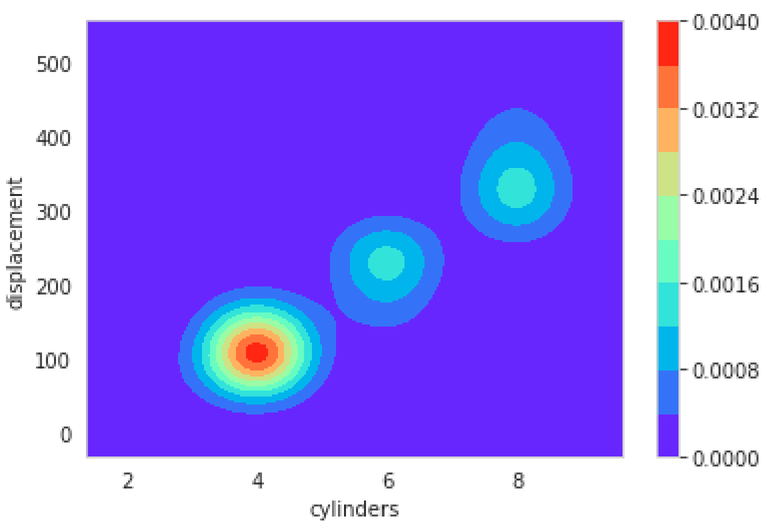


Рисунок 10.21 — Палитра *rainbow*

Построим функцию распределения:

```
sns.kdeplot(y, shade=True, cumulative=True)
```

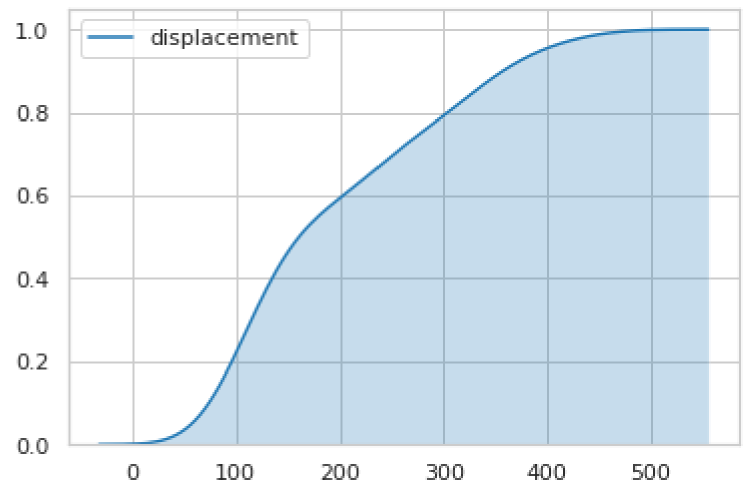


Рисунок 10.22 — Демонстрация работы с параметром `cumulative` функции `kdeplot()`

Для управления вычислениями могут быть использованы параметры `kernel`, `bw`, `gridsize` и `cut`:

- `kernel`: `{'gau' | 'cos' | 'biw' | 'epa' | 'tri' | 'triw' }`, `optional`
  - Функция ядра. Для двумерного варианта используется только Гауссово ядро (`gau`)
- `bw`: `{'scott' | 'silverman' }`, численное значение, пара чисел (для двумерного варианта), `optional`
  - Ширина ядра. Если оставить значение по умолчанию, то будет подобрана такая величина, которая даст наилучшее визуальное представление.

```
sns.kdeplot(y, shade=True, kernel='cos')
sns.kdeplot(y, shade=True, kernel='gau')
```

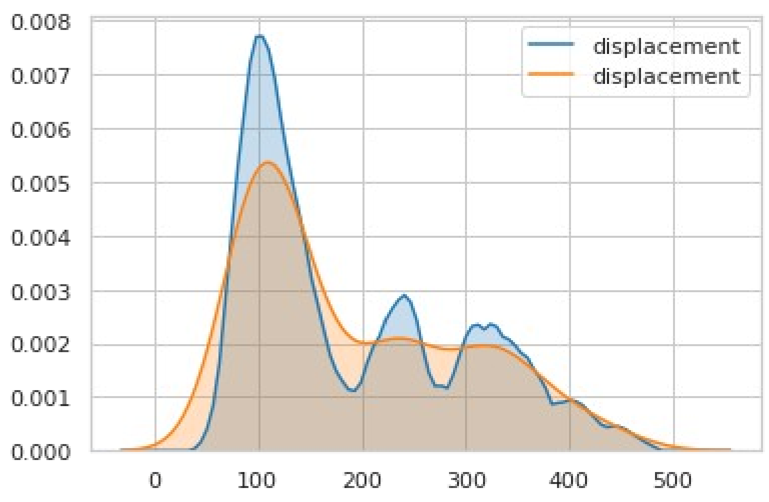


Рисунок 10.23 — Демонстрация работы с параметром `kernel` функции `kdeplot()`

```
sns.kdeplot(y, shade=True, label="default")
sns.kdeplot(y, shade=True, bw=10, label="bw=10")
sns.kdeplot(y, shade=True, bw=70, label="bw=70")
```

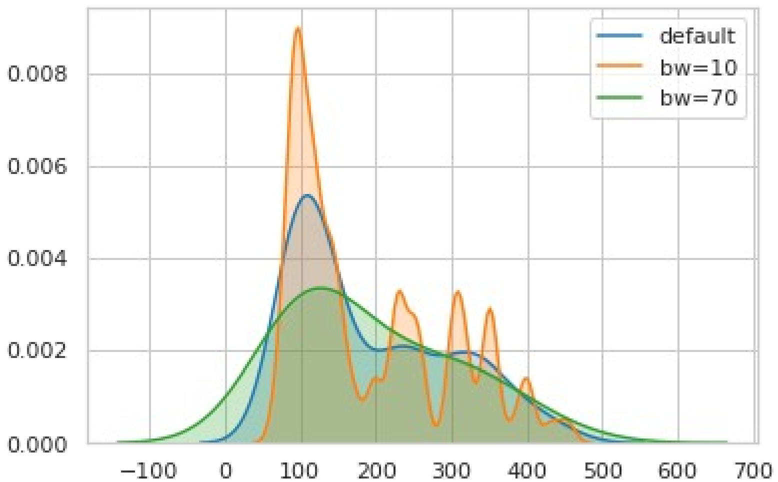


Рисунок 10.24 — Демонстрация работы с параметром bw функции `kdeplot()`

10.3 Функция `rugplot()`

Функция `rugplot()` отображает элементы данных в виде линии рядом с осью координат. Параметры функции:

- a: `vector`
  - Одномерный массив данных.
- `height`: численное значение, `optional`
  - Высота линии.
- `axis`:`{'x' | 'y'}`, `optional`
  - Ось, на которой будет отображена диаграмма.

Будем работать с набором данных *mpg*:

```
mpg = sns.load_dataset('mpg')
```

Построим диаграмму *rugplot*:

```
sns.rugplot(mpg["displacement"])
```

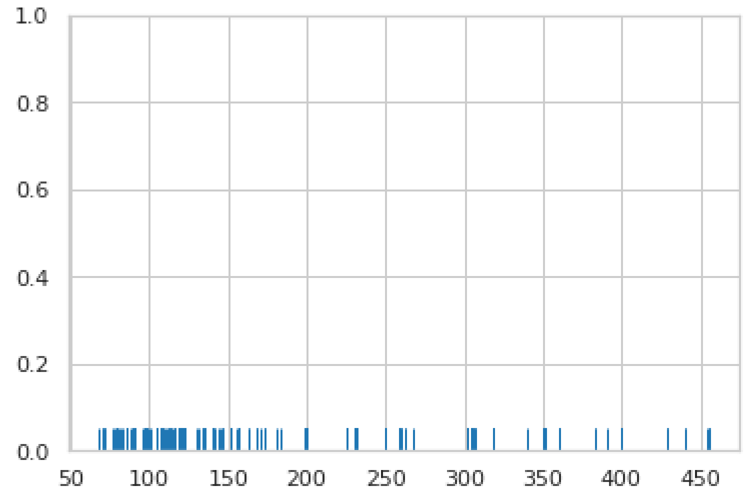


Рисунок 10.25 — Демонстрация работы функции `rugplot()`

Изменим цвет и высоту линий:

```
sns.rugplot(mpg["displacement"], height=0.5, color='orange')
```

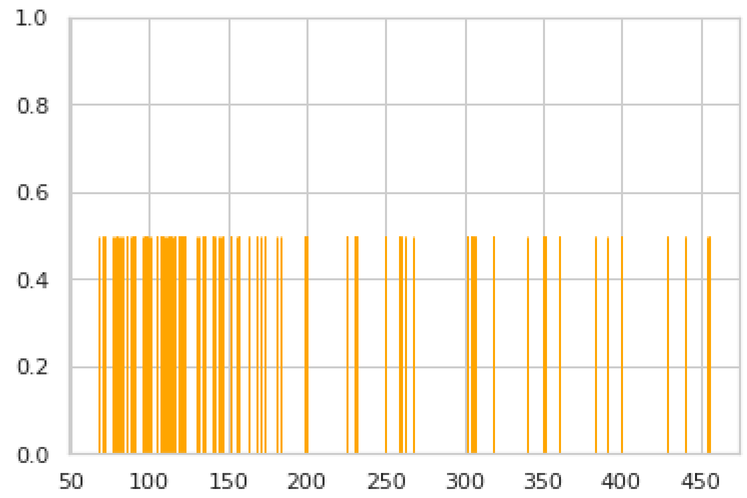


Рисунок 10.26 — Демонстрация работы с параметром `height` и `color` функции `kdeplot()`

Выберем ось *y* для представления значений параметра:

```
sns.rugplot(mpg["displacement"], height=0.3, color='r', axis='y')
```

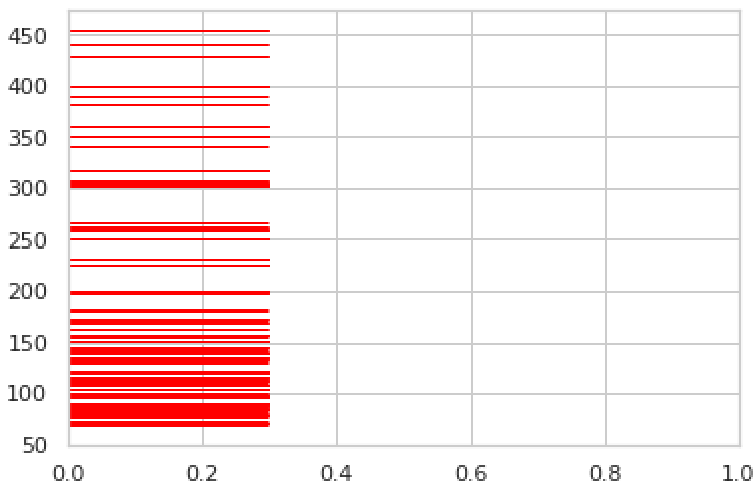


Рисунок 10.27 — Демонстрация работы с параметром `axis` функции `kdeplot()`

Дополнительно для управления представлением можно использовать следующие параметры:

- `ax`: `matplotlib axes`, `optional`
    - Поле (объект класса `Axes`), на котором будет выведена диаграмма.
- `kwargs`: `dict`
    - Словарь с параметрами, которые будут переданы в конструктор класса `LineCollection`.

Глава 11. Визуализация модели линейной регрессии

Инструменты этой группы обладают очень интересным функционалом: они строят модель линейной регрессии по переданным данным и отображают ее вместе с исходным набором. Функции представлены в таблице 11.1.

Таблица 11.1 — Функции для визуализации модели линейной регрессии

Функция	Описание
`lmplot()`	Отображает набор данных и линию регрессии, построенную по ним, с возможностью управлять компоновкой полей с графиками на подложке.
`regplot()`	Отображает набор данных и линию регрессии, построенную по ним.
`residplot()`	Отображает отклонения элементов исходного набора данных от регрессионной модели, построенной по ним, в виде диаграммы рассеяния.

11.1 Общие параметры функций

Рассмотрим некоторые из общих параметров функций визуализации модели линейной регрессии:

- `x`, `y`: имена переменных из набора `data`, `optional`
  - Связывают ось *x* и *y* с конкретными признаками из набора данных, переданного через параметр `data`. Для функций, отличных от `catplot()`, допустимо передавать вектора с данными напрямую, для `catplot()` — нет. Параметры могут иметь значение `None`, в случае если необходимо визуализировать весь набор `data`.

- `data`: `DataFrame`
    - Набор данных в формате `pandas.DataFrame`, в котором столбцы — это признаки, строки — значения. Имена столбцов, данные из которых необходимо визуализировать, передаются в параметры *x* и *y*.
- lowess: bool, optional
    - Если параметр равен `True`, то будет использована модель из *statsmodels* для построения локально взвешенной линейной регрессии. Доверительный интервал для такой модели не отображается.
- robust: bool, optional
    - Если параметр равен `True`, то будет использована модель из *statsmodels* для оценки робастной регрессии.
- `{scatter,line}_kws`: `dict`
    - Определяют дополнительные аргументы функций `plt.scatter()` и `plt.plot()`.

11.2 Функция `regplot()`

Функция `regplot()` отображает набор данных и линию регрессии, построенную по ним. Загрузим наборы данных *iris* и *mpg*:

```
mpg = sns.load_dataset("mpg")
iris = sns.load_dataset("iris")
```

Построим линию регрессии зависимости расстояния, которое может проехать автомобиль, от его мощности. Через параметры x и y передаём названия признаков, через `data` – набор данных.

```
sns.lmplot(x="horsepower", y="displacement", data=mpg)
```

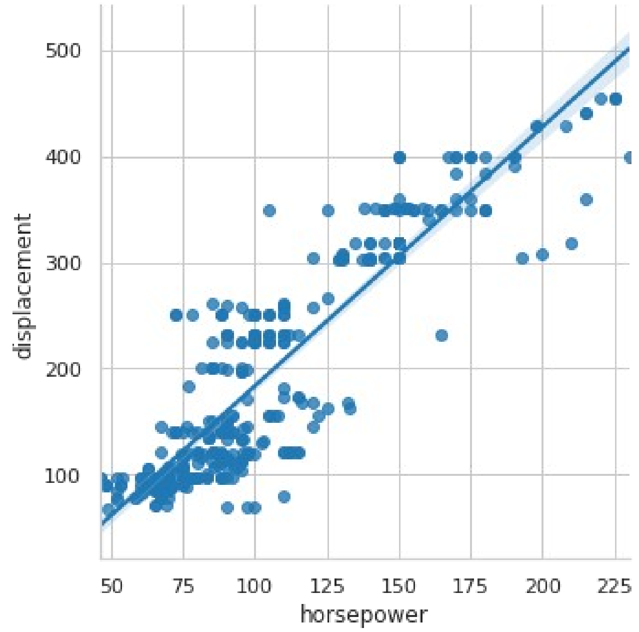


Рисунок 11.1 — Демонстрация работы функции `regplot()`

Из набора *iris* выберем только те ирисы, которые относятся к классу *setosa*:

```
iris_mod = iris[iris["species"] == "setosa"]
```

Построим зависимость ширины от длины наружной доли околоцветника:

```
sns.regplot(x="sepal_length", y="sepal_width", data=iris_mod)
```

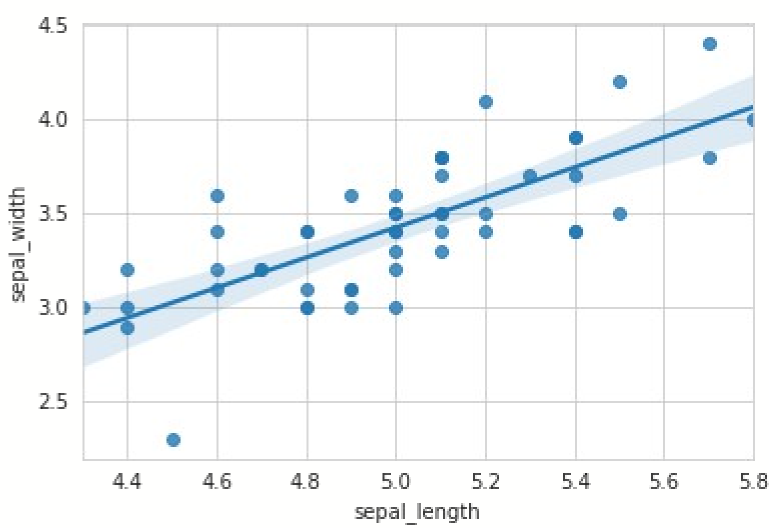


Рисунок 11.2 — Визуализации зависимости параметров *sepal_length* и *sepal_width* из набора данных *iris*

С поля графика можно убрать либо точки (параметр `scatter`), либо регрессионную прямую (параметр `fit_reg`). Продемонстрируем это, присвоим параметру `scatter` значение `False`:

```
sns.regplot(x="sepal_length", y="sepal_width", data=iris_mod,
scatter=False)
```

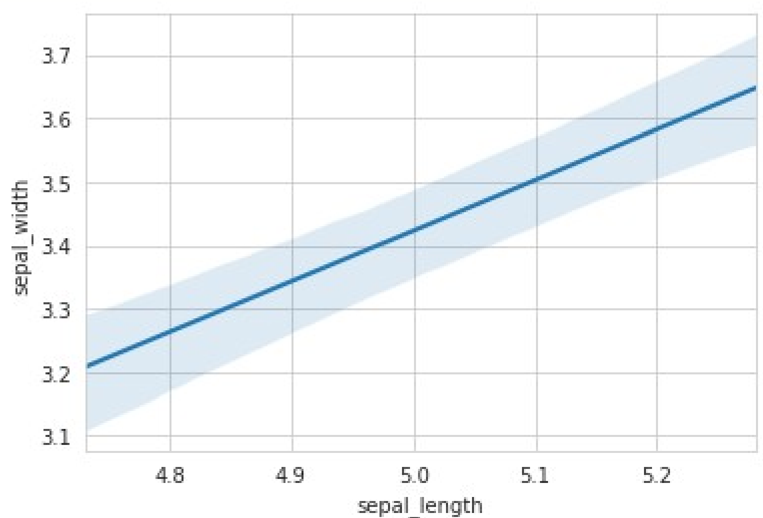


Рисунок 11.3 — Демонстрация работы с параметром `scatter` функции `regplot()`

Зададим параметру `fit_reg` значение `False`:

```
sns.regplot(x="sepal_length", y="sepal_width", data=iris_mod,
fit_reg=False)
```

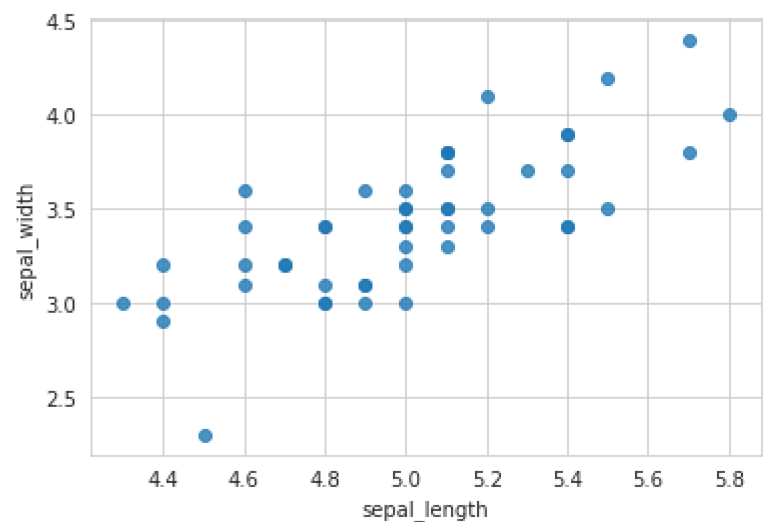


Рисунок 11.4 — Демонстрация работы с параметром `fit_reg` функции `regplot()`

Для настройки внешнего вида графиков можно воспользоваться параметрами:

- `color`: *Matplotlib*-цвет, `optional`
  - Цвет отображаемых элементов (точки и линия).
- `marker`: код маркера из *Matplotlib*
  - Стиль маркера, которым отображаются точки.

```
sns.regplot(x="sepal_length", y="sepal_width", data=iris_mod, color='r')
```

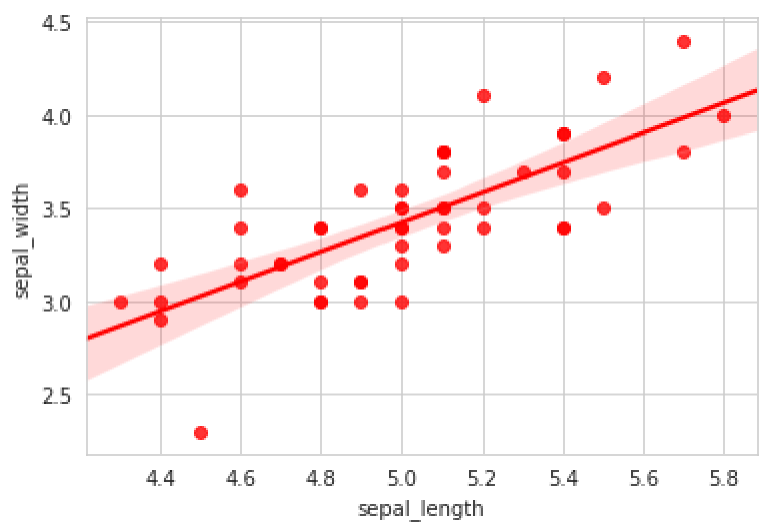


Рисунок 11.5 — Демонстрация работы с параметром `color` функции `regplot()`

```
markers = ["^", "*", "o"]
plt.figure(figsize=(15, 5))
for i, m in enumerate(markers):
    plt.subplot(1, len(markers), i+1)
    plt.title(f"marker = {m}")
    sns.regplot(x="sepal_length", y="sepal_width", data=iris_mod,
color='r', marker=m)
```

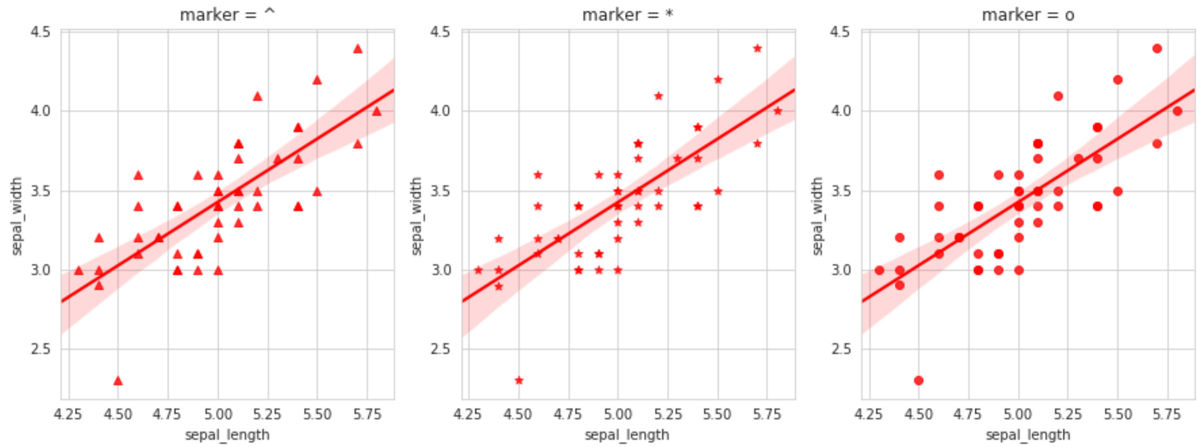


Рисунок 11.6 — Демонстрация работы с параметром marker функции regplot()

Для более тонкой настройки используйте параметры scatter_kws, line_kws:

```
s_kws = {"linewidths":2, "edgecolors":'g', 'color':'r', "s": 50}
p_kws = {'ls':'-.', 'lw':3}
sns.regplot(x="sepal_length", y="sepal_width", data=iris_mod,
scatter_kws=s_kws, line_kws=p_kws)
```

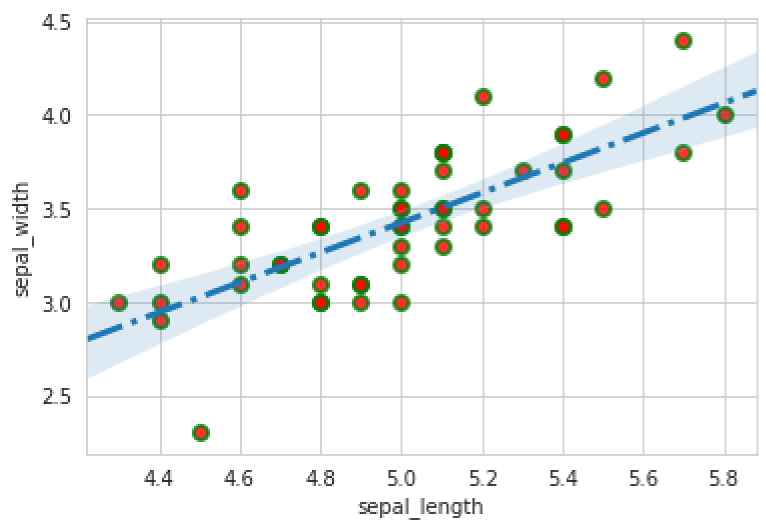


Рисунок 11.7 — Демонстрация работы с параметром scatter_kws и line_kws функции regplot()

Функция replog() предоставляет альтернативу точечной диаграмме — диаграмму со значениями оценок и доверительными интервалами. Для настройки этого режима используются параметры:

- `x_estimator`: функция, `optional`
    - Функция для вычисления значения оценки.
- `x_bins`: `int` или `vector`, `optional`
    - Определяет количество групп, на которые будет разбито исходное множество значений.
- `x_ci`: `'ci'`, `'sd'`, `int` в диапазоне `[0, 100]` или None, `optional`
    - Размер доверительного интервала. Если значение равно `'sd'`, то вместо доверительного интервала будет использоваться стандартное отклонение.

Построим диаграмму с медианной оценкой:

```
from numpy import median
sns.regplot(x="sepal_length", y="sepal_width", data=iris_mod,
x_estimator=median)
```

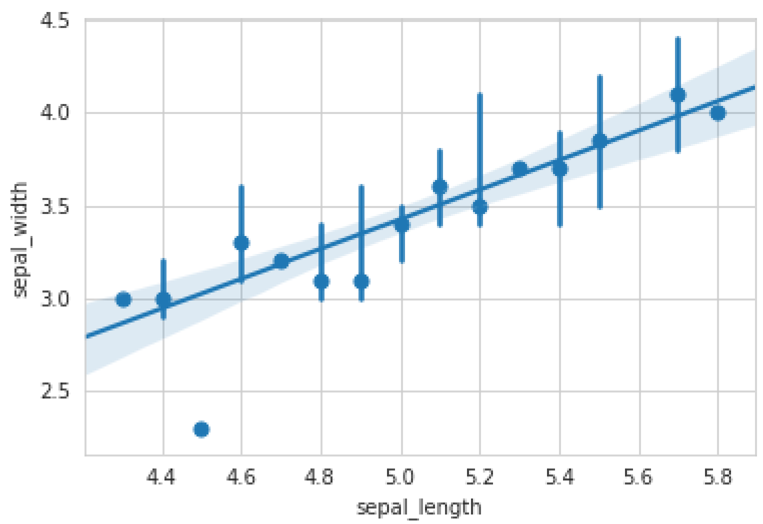


Рисунок 11.8 — Диаграмма с медианной оценкой

Пример работы с параметром `x_bins`:

```
x_bins = [5, 10, None]
plt.figure(figsize=(15, 5))
for i, b in enumerate(x_bins):
    plt.subplot(1, len(x_bins), i+1)
    plt.title(f"x_bins = {b}")
    sns.regplot(x="sepal_length", y="sepal_width", data=iris_mod,
x_bins=b)
```

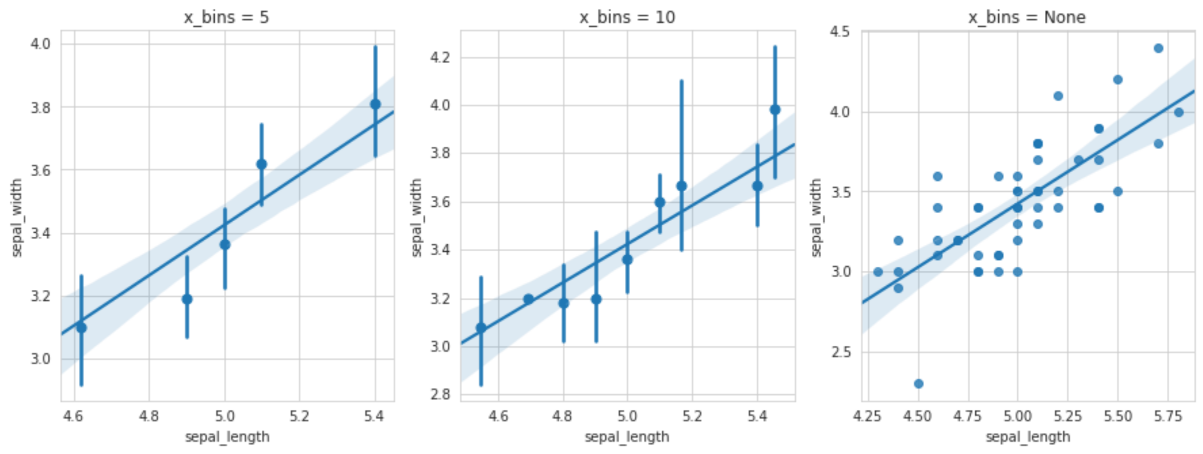


Рисунок 11.9 — Демонстрация работы с параметром x_bins функции `regplot()`

Пример работы с параметром `x_ci`:

```
x_ci = [50, 80, "sd"]
plt.figure(figsize=(15, 5))
for i, xc in enumerate(x_ci):
    plt.subplot(1, len(x_ci), i+1)
    plt.title(f"x_ci = {xc}")
    sns.regplot(x="sepal_length", y="sepal_width", data=iris_mod,
x_ci=xc, x_bins=10)
```

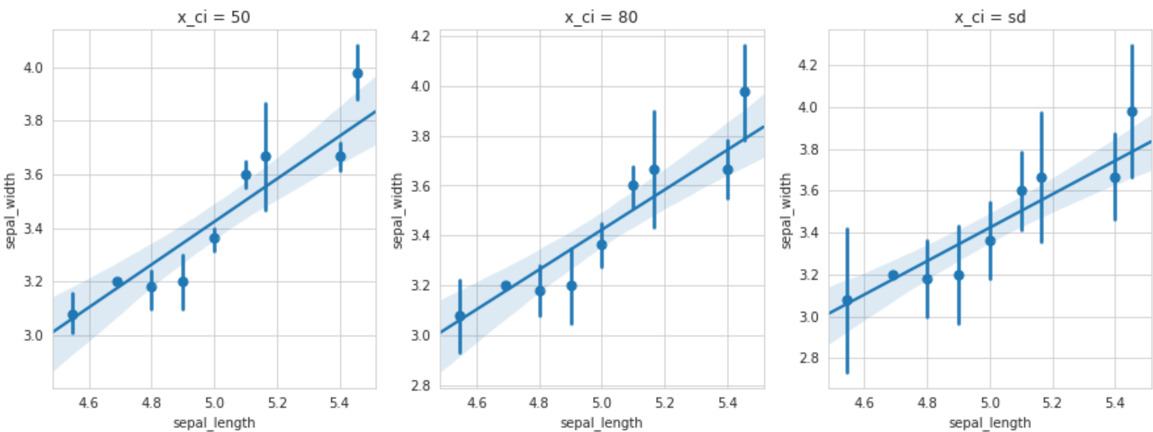


Рисунок 11.10 — Демонстрация работы с параметром `x_ci` функции `regplot()`

Как вы уже могли заметить, вокруг линии регрессии есть область того же цвета, что и основная линия, но более светлого оттенка. Она представляет доверительный интервал, расчётом этой области управляют следующие параметры:

- `ci`: `int` в диапазоне [0, 100] или `None`, `optional`
    - Размер доверительного интервала.
- `n_boot`: `int`, `optional`
    - Количество образцов для оценки `ci`.

Посмотрим как будет выглядеть диаграмма с различными значениями `ci`:

```
ci = [50, 99, None]
plt.figure(figsize=(15, 5))
for i, c in enumerate(ci):
    plt.subplot(1, len(ci), i+1)
    plt.title(f"ci = {c}")
    sns.regplot(x="sepal_length", y="sepal_width", data=iris_mod, ci=c)
```

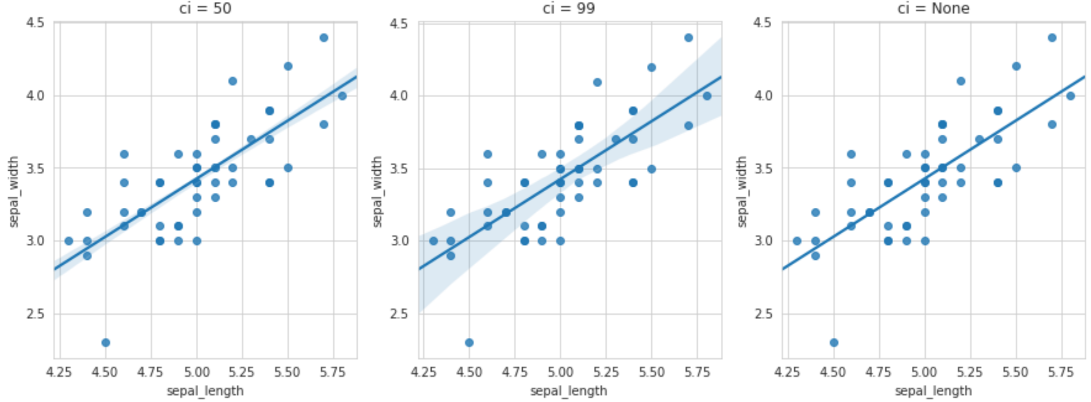


Рисунок 11.11 — Демонстрация работы с параметром ci функции `regplot()`

Рассмотрим некоторые из параметров функции `regplot()`:

- `logistic`: `bool`, `optional`
  - Если параметр равен `True`, то будет принято предположение, что *y* является бинарной переменной и для построения модели логистической регрессии будет использоваться модель из *statsmodels*.

Добавим в набор данных *iris* ещё один признак: *is_setosa*, который принимает значение `True`, если ирис относится к классу *setosa*, иначе — значение `False`:

```
iris["is_setosa"] = iris["species"] == 'setosa'
sns.regplot(x="sepal_length", y="is_setosa", logistic=True, data=iris)
```

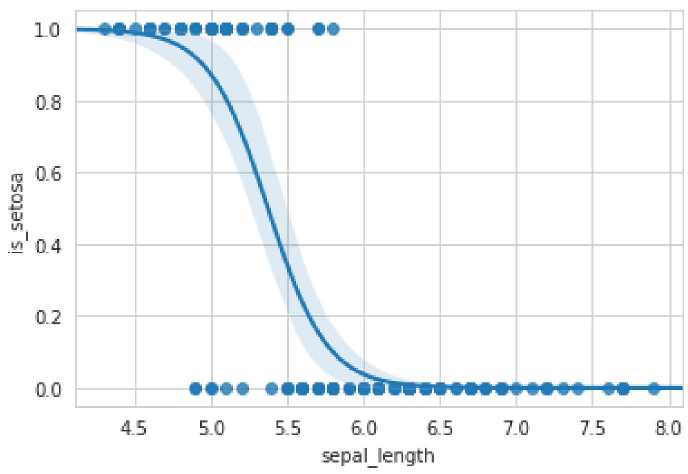


Рисунок 11.12 — Демонстрация работы с параметром `logistic` функции `regplot()`

- robust: bool, optional
  - Если параметр равен True, то будет использована модель из statsmodels для оценки робастной регрессии.

Пример работы с параметром robust:

```
sns.regplot(x="sepal_length", y="sepal_width", robust=True,
data=iris_mod)
```

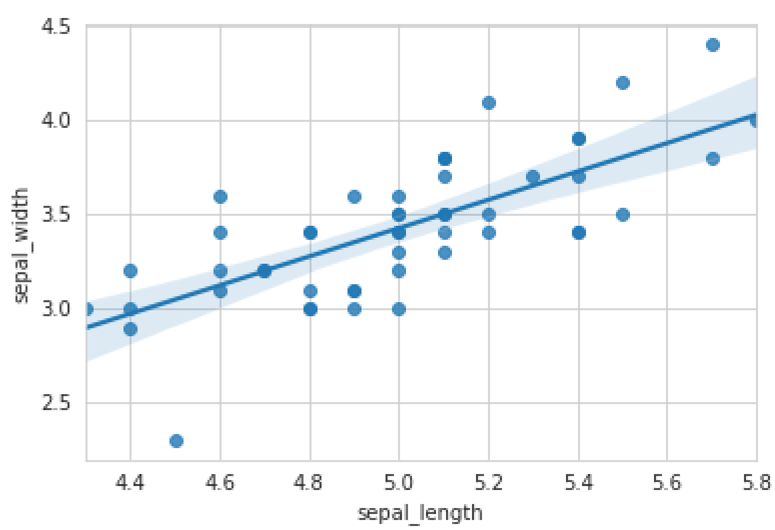


Рисунок 11.13 — Демонстрация работы с параметром robust функции regplot()

- logx: bool, optional
  - Если значение равно True, то оценивается линейная регрессия вида *y ~ log(x)*. Значения параметра x должны быть больше нуля.

Пример работы с параметром logx:

```
sns.regplot(x="mpg", y="displacement", logx=True, data=mpg)
```

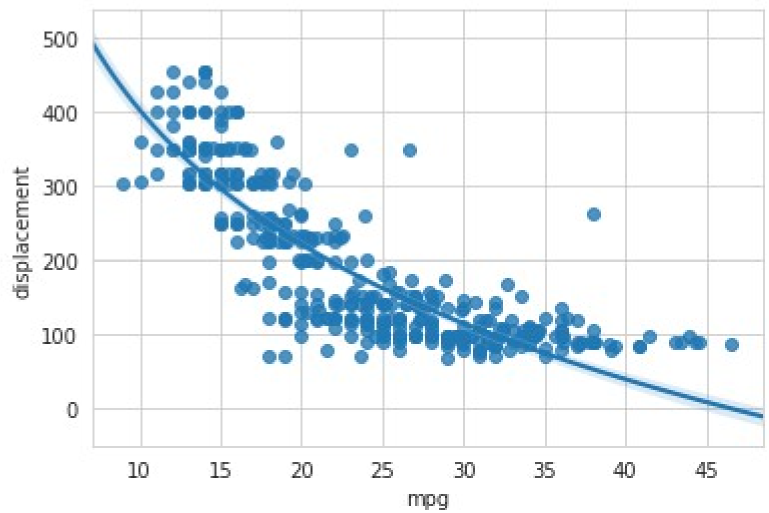


Рисунок 11.14 — Демонстрация работы с параметром `logx` функции `regplot()`

- `lowess`: `bool`, `optional`
  - Если параметр равен `True`, то будет построена локально взвешенная линейная регрессия. Доверительный интервал для такой модели не отображается.

Пример работы с параметром `lowess`:

```
sns.regplot(x="mpg", y="displacement", lowess=True, data=mpg)
```

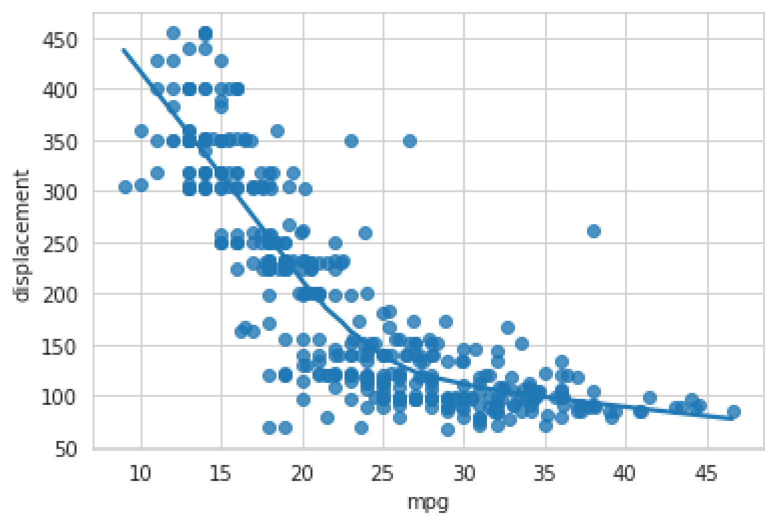


Рисунок 11.15 — Демонстрация работы с параметром `lowess` функции `regplot()`

- `order`: `int`, `optional`
    - Если значение параметра больше 1, то будет использована функция `numpy.polyfit` для построения полиномиальной регрессии.

Пример работы с параметром `order`:

```
order=[1, 2, 3]
plt.figure(figsize=(15, 5))
for i, o in enumerate(order):
    plt.subplot(1, len(order), i+1)
    plt.title(f"order = {o}")
    sns.regplot(x="mpg", y="displacement", order=o,  data=mpg)
```

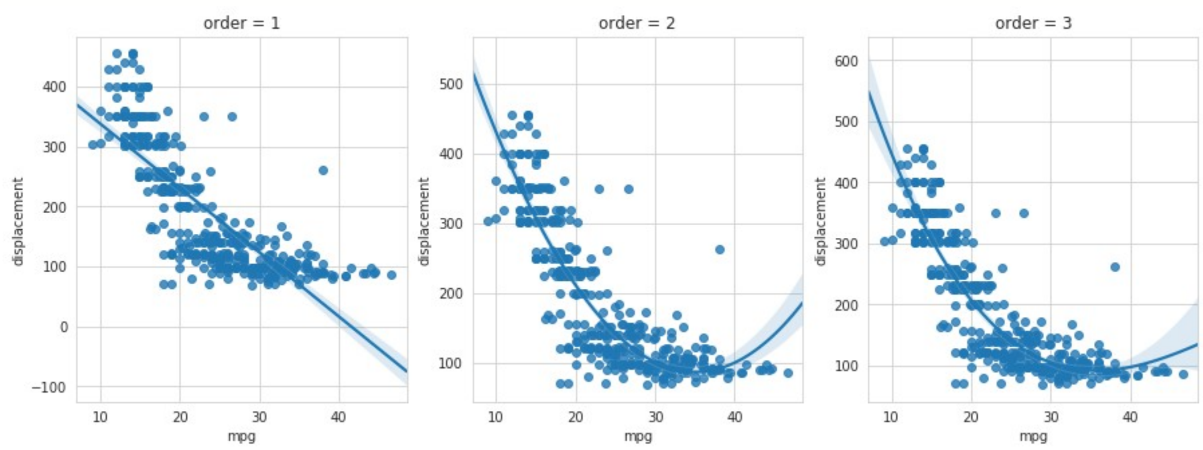


Рисунок 11.16 — Демонстрация работы с параметром `order` функции `regplot()`

- `truncate`: `bool`, `optional`
    - По умолчанию линия регрессии отображается для всего диапазона оси x. Если параметр равен `True`, то линия будет ограничена крайними значениями набора данных.

Пример работы с параметром `truncate`:

```
sns.regplot(x="mpg", y="displacement", truncate=True, data=mpg)
```

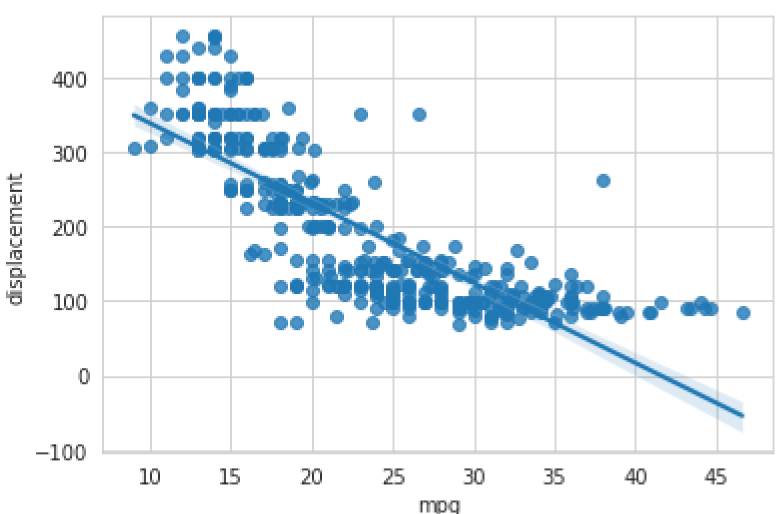


Рисунок 11.17 — Демонстрация работы с параметром `truncate` функции `regplot()`

11.3 Функция `residplot()`

Функция `residplot()` отображает отклонения элементов исходного набора данных от регрессионной модели, построенной по ним, в виде диаграммы рассеяния. Каждая точка такой диаграммы — это разность между значением элемента исходного набора и значением, которое выдаст модель регрессии в этой точке. Параметры функции `residplot()` совпадают с набором, приведённым в разделе "*11.1 Общие параметры функций*".

Приведём несколько примеров её использования. Будем работать с набором *mpg*:

```
mpg = sns.load_dataset("mpg")
```

Построим модель линейной регрессии с помощью функции `regplot()`:

```
sns.regplot(x="mpg", y="displacement", data=mpg)
```

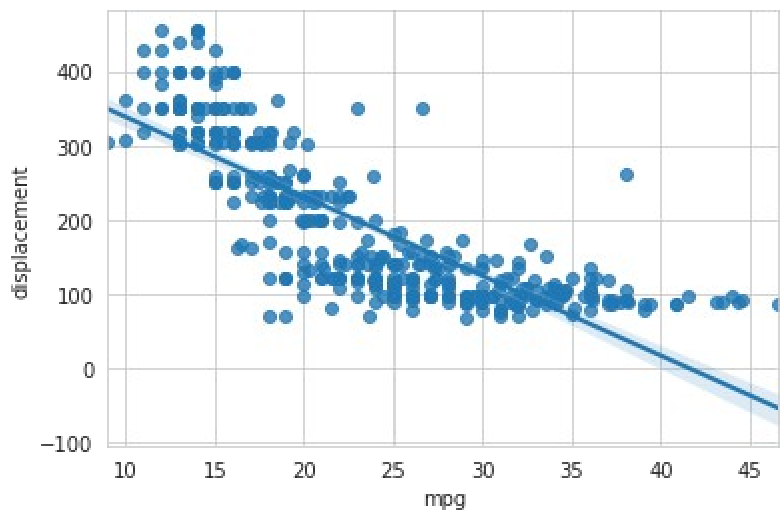


Рисунок 11.18 — График модели линейной регрессии, построенный с помощью `regplot()`

Теперь построим диаграмму рассеяния отклонений значений набора данных от модели:

```
sns.residplot(x="mpg", y="displacement", color='g', data=mpg)
```

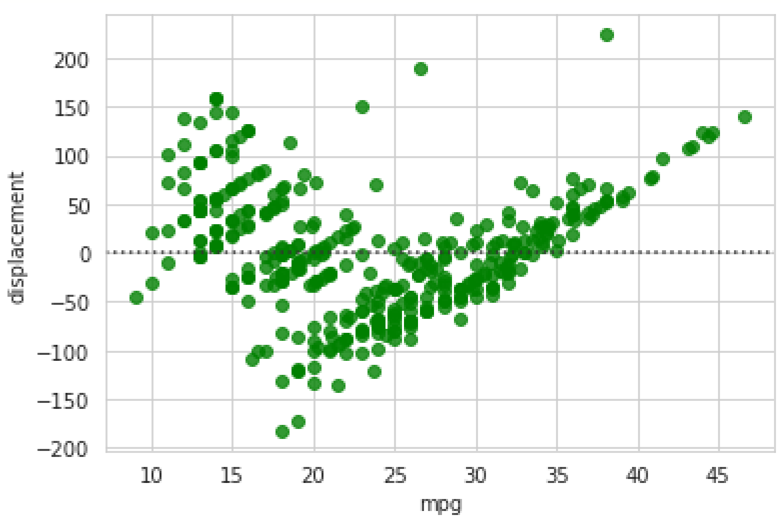


Рисунок 11.19 — Диаграмма рассеяния отклонений значений набора данных от модели, построенная с помощью `residplot()`

Пример модели третьего порядка:

```
sns.regplot(x="mpg", y="displacement", order=3, data=mpg)
```

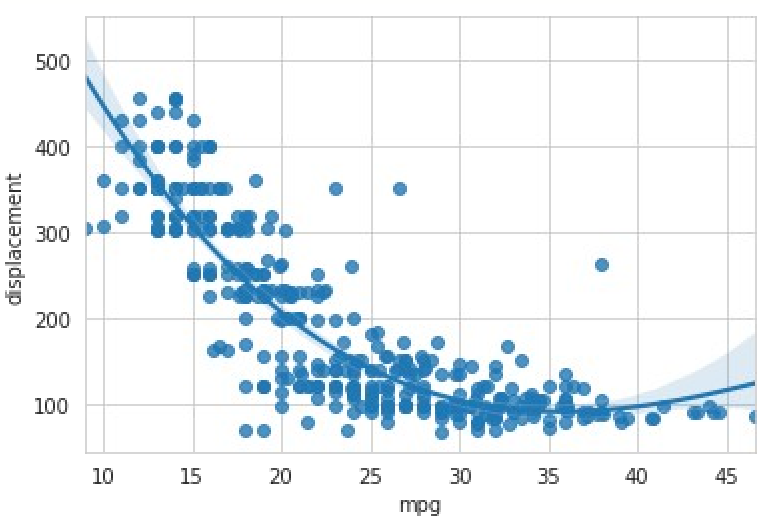


Рисунок 11.20 — График модели третьего порядка

Для такой модели распределение отклонений будет таким:

```
sns.residplot(x="mpg", y="displacement", order=3, color='g', data=mpg)
```

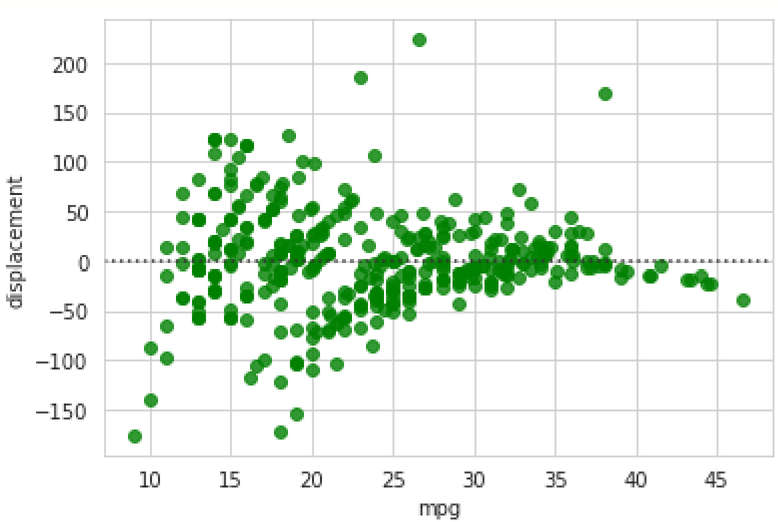


Рисунок 11.21 — Распределение отклонений для модели третьего порядка

11.4 Функция lmplot()

Функция lmplot() по своему назначению аналогична regplot() (см. раздел "*11.2 Функция* regplot()") с возможностью управлять компоновкой полей с графиками на подложке.

Загрузим наборы данных *mpg* и *iris*:

```
mpg = sns.load_dataset("mpg")
iris = sns.load_dataset("iris")
```

Если пока оставить в стороне параметры, отвечающие на настройку подложки, то в остальном, аргументы функции lmplot() практически полностью совпадают по названию и назначению с рассмотренными для функции regplot(). К ним относятся x, y, data, order, x_bins, units, ci, x_ci, n_boot, {x,y}_jitter, {x, y}_partial, x_estimator, scatter, fit_reg, logistic, robust, logx, lowess, truncate, {scatter, line}_kws, dropna.

Параметры, отвечающие за разделение данных на группы и управление отображением диаграмм функции lmplot() совпадают с аргументами функций relplot() и catplot(), к ним относятся hue, row, col, pallete, hue_order, legend, legend_out, height, aspect. Поэтому мы не будем приводить полное их описание, ограничимся примером работы с ними.

Построим график с моделью регрессии с помощью функции lmplot(), такой же как мы могли бы получить с помощью функции regplot().

```
sns.lmplot(x="horsepower", y="displacement", data=mpg)
```

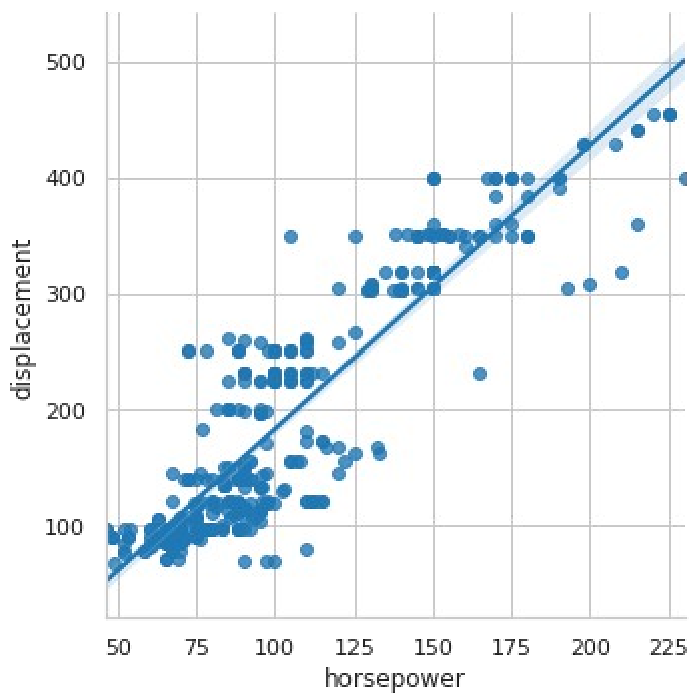


Рисунок 11.22 — График модели линейной регрессии, построенный с помощью `lmplot()`

Для разделения данных по категориальному признаку с последующим построением для каждой группы своей модели регрессии в функции `lmplot()` используются параметры:

- `hue`: для цветового разделения;
- `row`: для вертикального представления диаграмм;
- `col`: для горизонтального представления диаграмм.

За настройку цветовой схемы отвечает параметр `pallete`.

Построим диаграмму для набора *iris* с разделением по типу ириса (признак *species*):

```
sns.lmplot(x="sepal_width", y="petal_width", hue="species", data=iris)
```

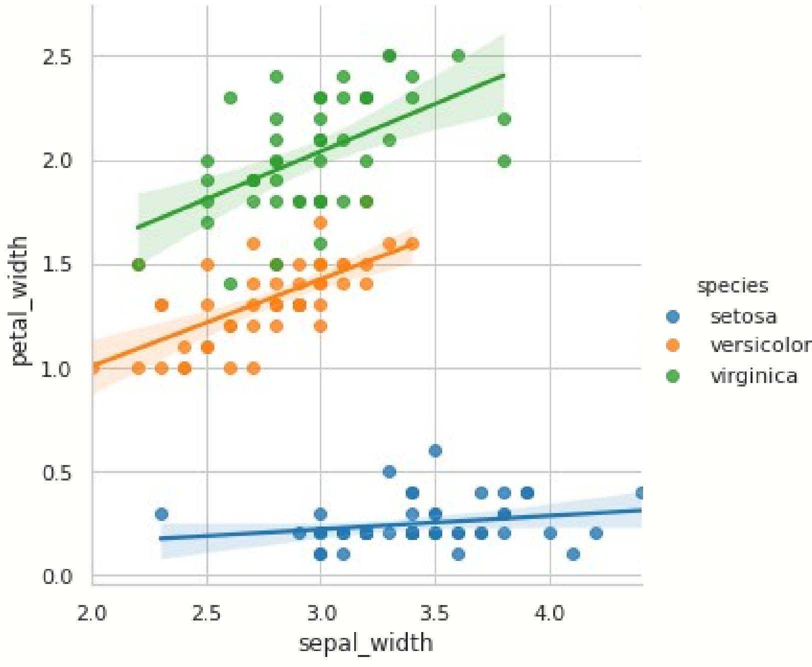


Рисунок 11.23 — Демонстрация работы с параметром hue функции `lmplot()`

Изменим палитру:

```
sns.lmplot(x="sepal_width", y="petal_width",hue="species",
palette="Dark2", data=iris)
```

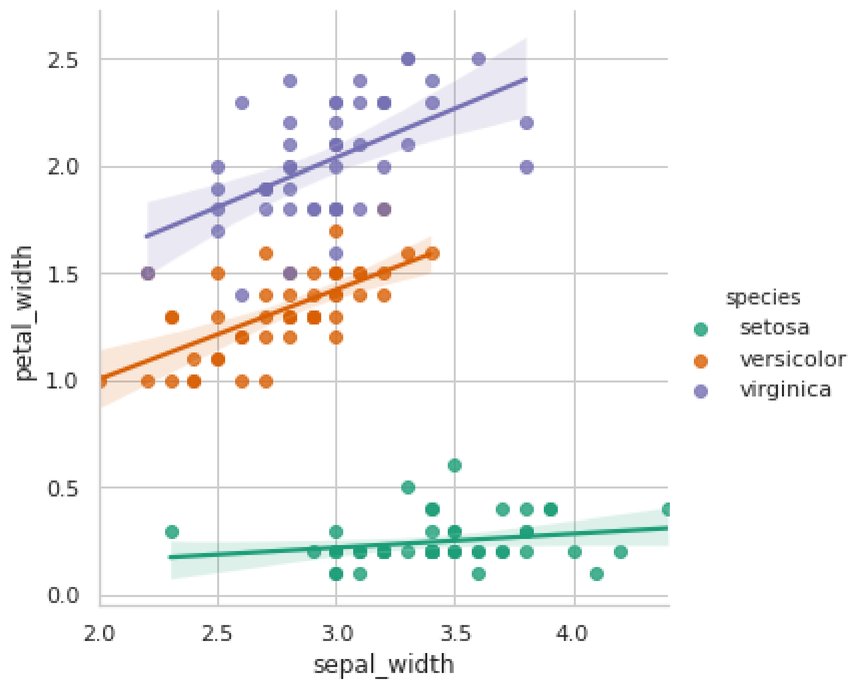


Рисунок 11.24 — Демонстрация работы с параметром `palette` функции `lmplot()`

Представим графики для каждого отдельного значения *species* на разных полях с горизонтальным разделением:

```
sns.lmplot(x="sepal_length", y="sepal_width", hue="species",
palette="Dark2", col="species", data=iris)
```

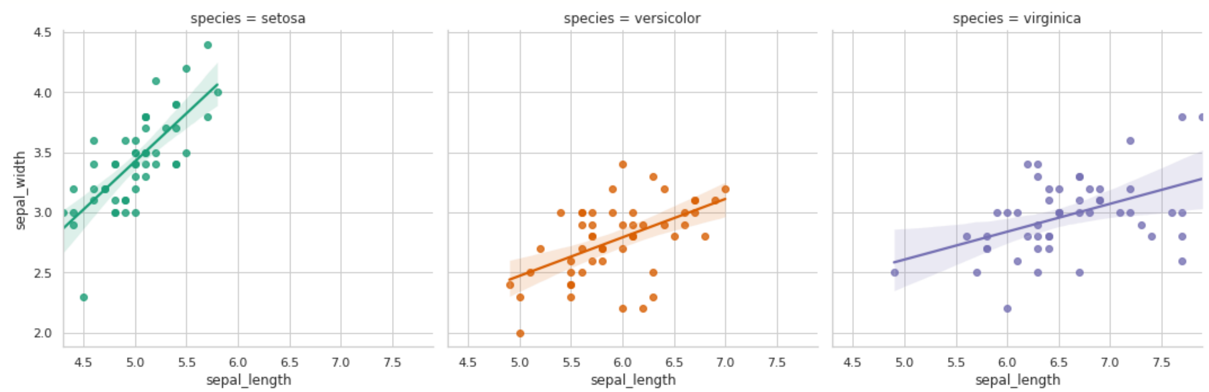


Рисунок 11.25 — Демонстрация работы с параметром `col` функции `lmplot()`

С вертикальным разделением:

```
sns.lmplot(x="sepal_length", y="sepal_width", hue="species",
palette="Dark2", row="species", data=iris)
```

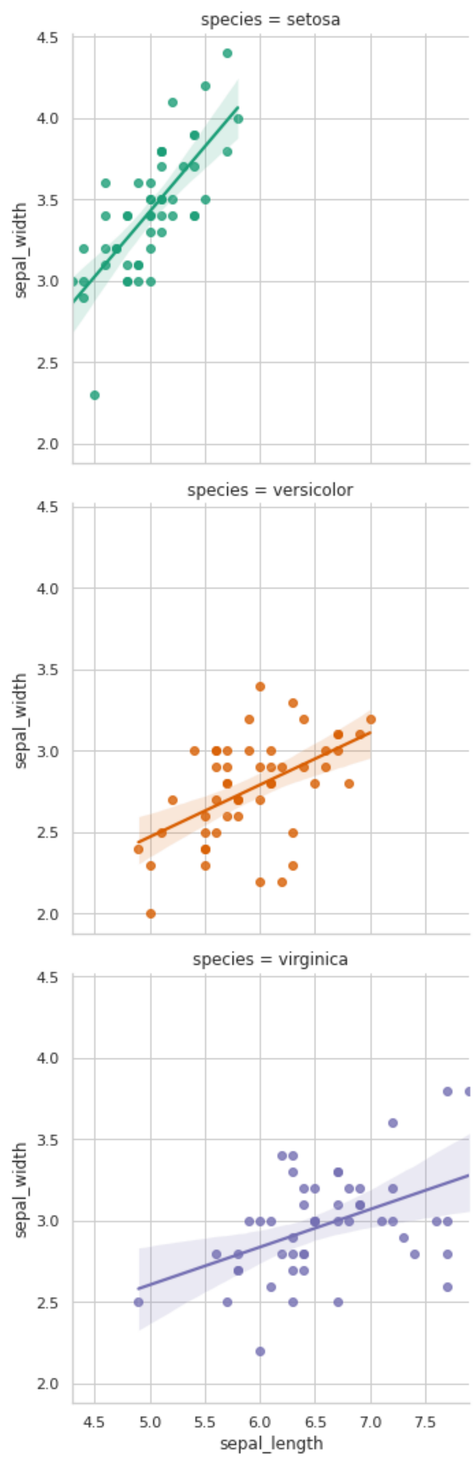


Рисунок 11.26 — Демонстрация работы с параметром `row` функции `lmplot()`

Для управления размером диаграмм используйте параметры `height` (высота) и `aspect` (соотношение сторон):

```
sns.lmplot(x="sepal_length", y="sepal_width", hue="species",
palette="Dark2", col="species", height=3, aspect=1.5, data=iris)
```

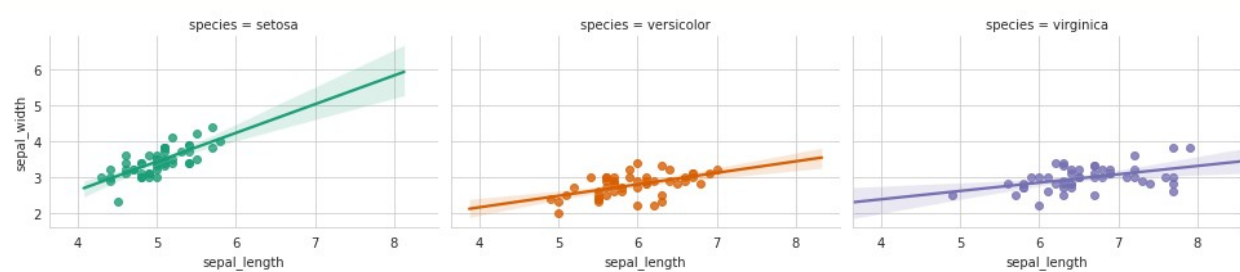


Рисунок 11.27 — Демонстрация работы с параметром `height` и `aspect` функции `lmplot()`

Параметры `sharex` и `sharey` отвечают за отображение осей `x` и `y` на всех диаграммах либо только на крайней левой (при горизонтальном расположении) или на нижней (при вертикальном расположении):

```
sns.lmplot(x="sepal_length", y="sepal_width", col="species",
sharey='col', data=iris)
```

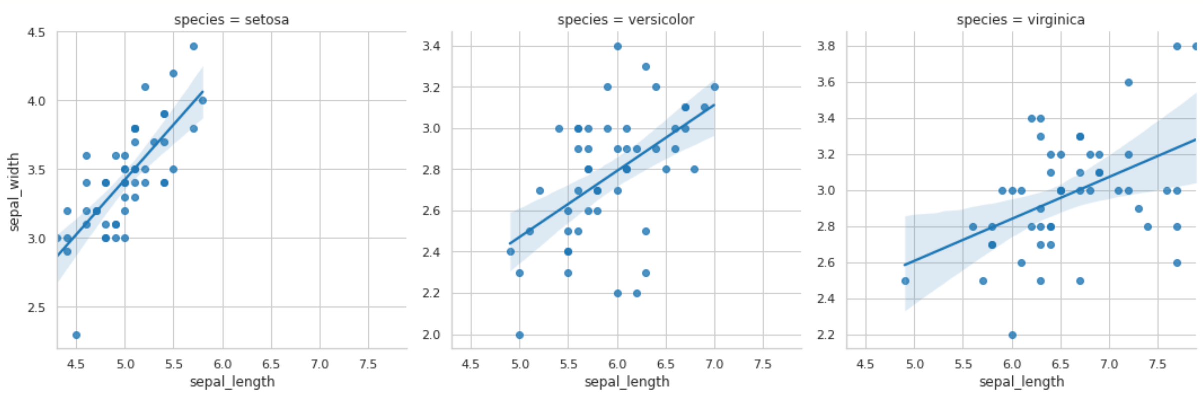


Рисунок 11.28 — Демонстрация работы с параметром `sharey` функции `lmplot()`

Порядок задания цвета (если вы используете параметр `hue`) или порядок вывода диаграмм (если для разделения на группы используется `col` или

row) задаётся с помощью hue_order, col_order, row_order. Приведём пример для hue_order:

```
h_order = ["virginica", "setosa", "versicolor"]
sns.lmplot(x="sepal_width", y="petal_width", hue="species",
hue_order=h_order, data=iris)
```

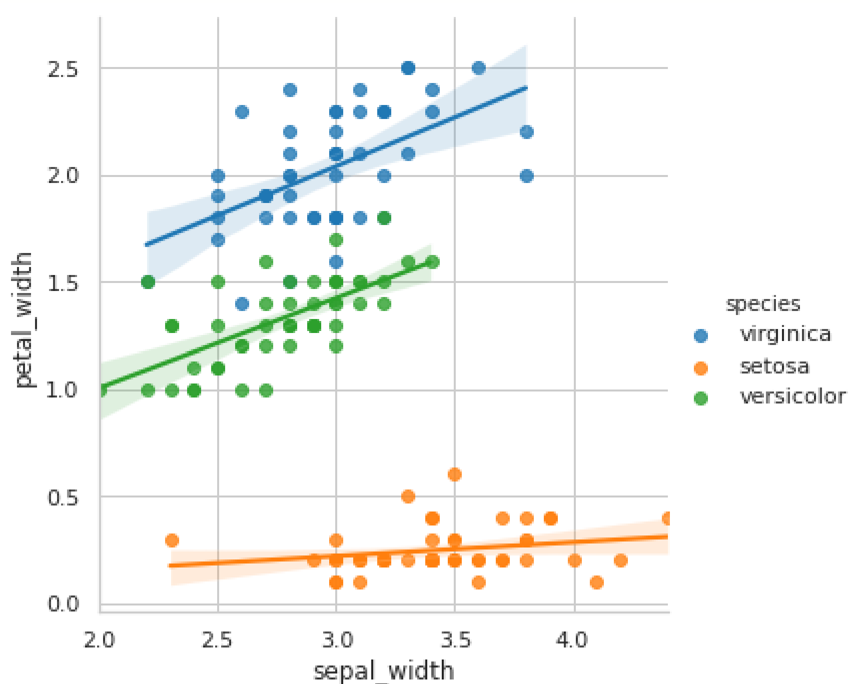


Рисунок 11.29 — Демонстрация работы с параметром hue_order функции lmplot()

За отображение легенды отвечает параметр legend, за вынос её за пределы поля с графиками — legend_out.

Глава 12. Управление компоновкой диаграмм

Seaborn предоставляет три набора инструментов для управления компоновкой диаграмм:

- *Facet*-сетка: классическая сетка для размещения графиков, построенных с помощью функций из наборов для визуализации отношений в данных (`scatterplot()`, `lineplot()`), визуализации категориальных данных (`stripplot()`, `boxplot()` и т.д.) и визуализации модели линейной регрессии (`regplot()`, `residplot()`).
- *Pair*-сетка: сетка для представления попарных соотношений в данных.
- *Joint*-сетка: отображает диаграмму для двух переменных с дополнительной визуализацией их распределений.

12.1 *Facet*-сетка

С классом `FacetGrid` мы уже встречались, когда знакомились с функциями `relplot()`, `catplot()`, помимо реализации общего интерфейса для соответствующих групп функций, они позволяли работать с компоновкой графиков. Так как класс `FacetGrid` доступен для использования напрямую, то можно самостоятельно, на базе соответствующего объекта, создавать компоновку.

С большей частью параметров конструктора `FacetGrid` вы должны быть уже знакомы, если читали описание `relplot()` и `catplot()`, к ним относятся `data`, `row`, `col`, `hue`, `col_wrap`, `sharex`, `sharey`, `height`, `aspect`, `palette`, `row_order`, `col_order`, `hue_order`, `dropna`, `legend_out`, `margin_titles`.

Для отображения графиков необходимо вызвать метод `map()` объекта класса `FacetGrid`. Он имеет следующий набор аргументов:

- `func`: `callable`
  - Функция построения графика (например, `scatter()`).
- `args`: `strings`
  - Имена столбцов из набора, который был передан через параметр `data` при создании объекта `FacetGrid`.
- `kwargs`: `keyword arguments`
  - Параметры функции, переданной через аргумент `func`.

Загрузим набор данных *dots*:

```
dots = sns.load_dataset("dots")
```

Извлечём первые 250 элементов:

```
dots_mod = dots.sample(frac=1)[:250]
```

Для разделения графиков по столбцам будем использовать параметр *choice*, который может принимать значения *T1* или *T2*, по строкам — *align*, принимающий значения *succ*, *dots*. Создадим объект класса `FacetGrid`:

```
fg = sns.FacetGrid(dots_mod, col="choice", row="align")
```

Теперь в каждой ячейке получившейся сетки выведем диаграмму рассеяния для признаков *time* и *firing_rate*:

```
fg.map(plt.scatter, "firing_rate", "time")
```

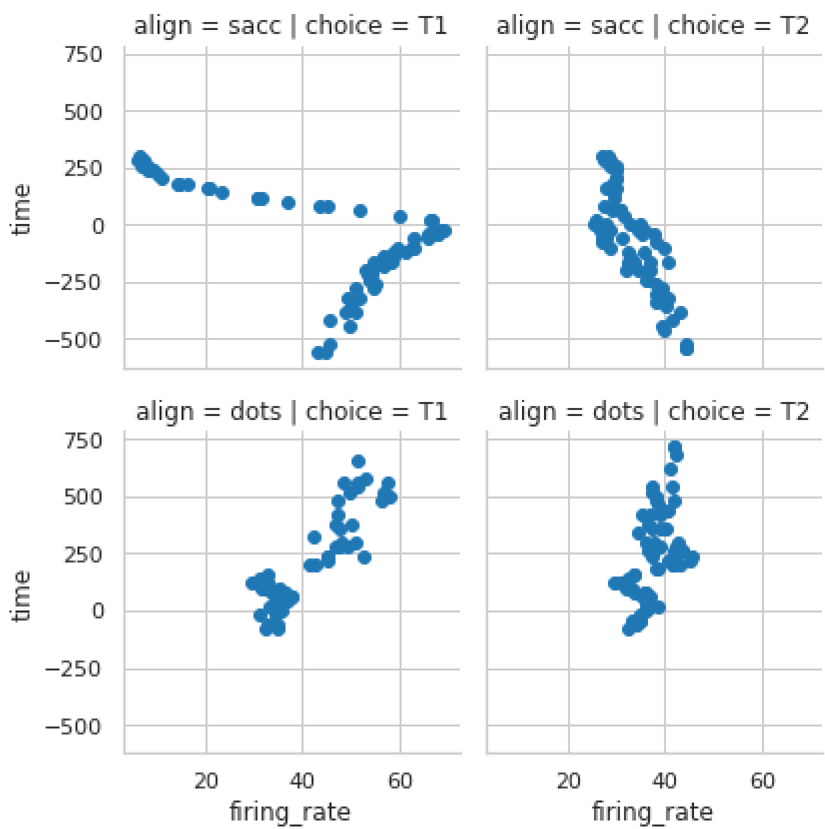


Рисунок 12.1 — Демонстрация работы с параметром `col` и `row` класса `FacetGrid`

Вместо диаграммы рассеяния построим гистограммы распределения признака *coherence*:

```
fg = sns.FacetGrid(dots_mod, col="choice", row="align")
fg.map(plt.hist, "coherence")
```

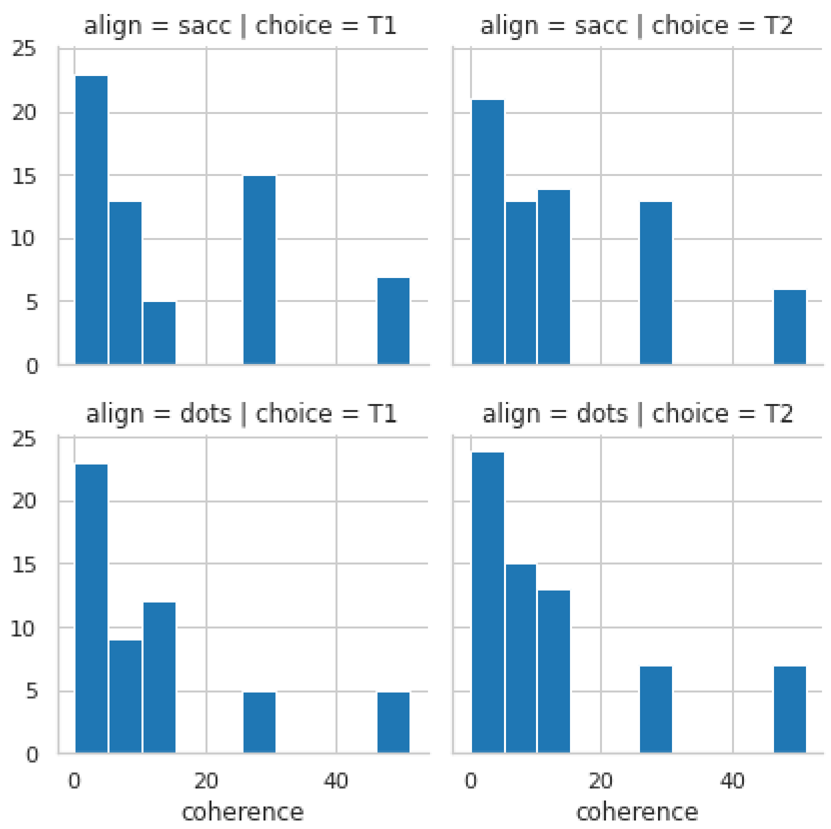


Рисунок 12.2 — Демонстрация работы с параметрами `col` и `row` класса `FacetGrid`

Для дополнительного цветового разделения можно использовать параметр hue:

```
fg = sns.FacetGrid(dots_mod, col="choice", hue="align")
fg.map(plt.scatter, "firing_rate", "time")
```

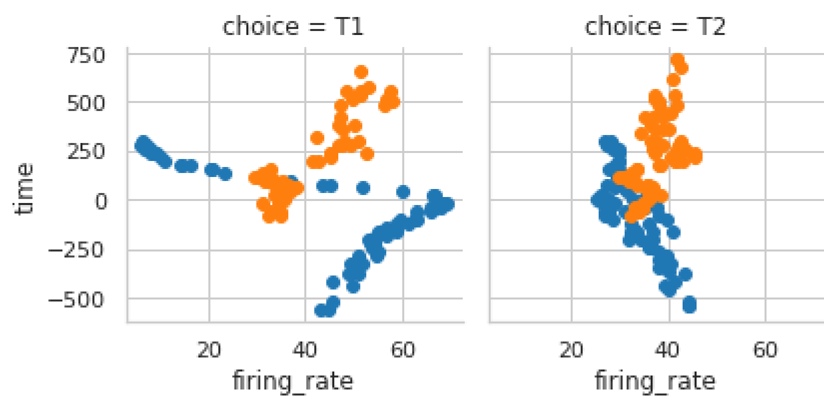


Рисунок 12.3 — Демонстрация работы с параметром `hue` класса `FacetGrid`

Изменим палитру:

```
fg = sns.FacetGrid(dots_mod, col="choice", hue="align", palette="Set2")
fg.map(plt.scatter, "firing_rate", "time")
```

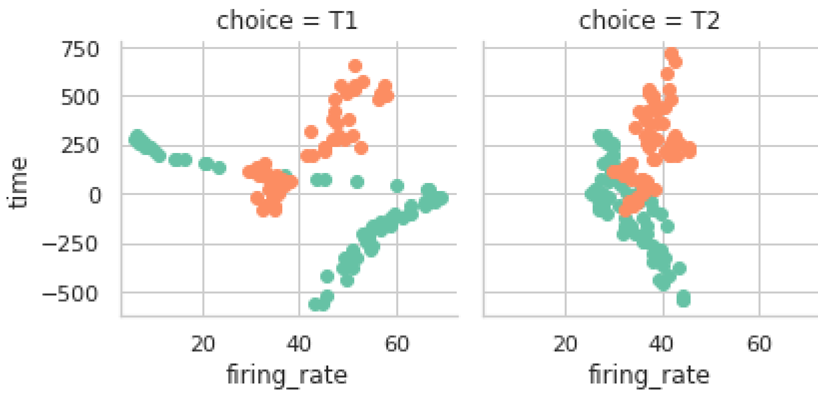


Рисунок 12.4 — Демонстрация работы с параметром `palette` класса `FacetGrid`

Зададим размеры диаграмм через параметры `height` и `aspect`:

```
fg = sns.FacetGrid(dots_mod, col="choice", hue="align", height=5,
aspect=0.5)
fg.map(plt.scatter, "firing_rate", "time")
```

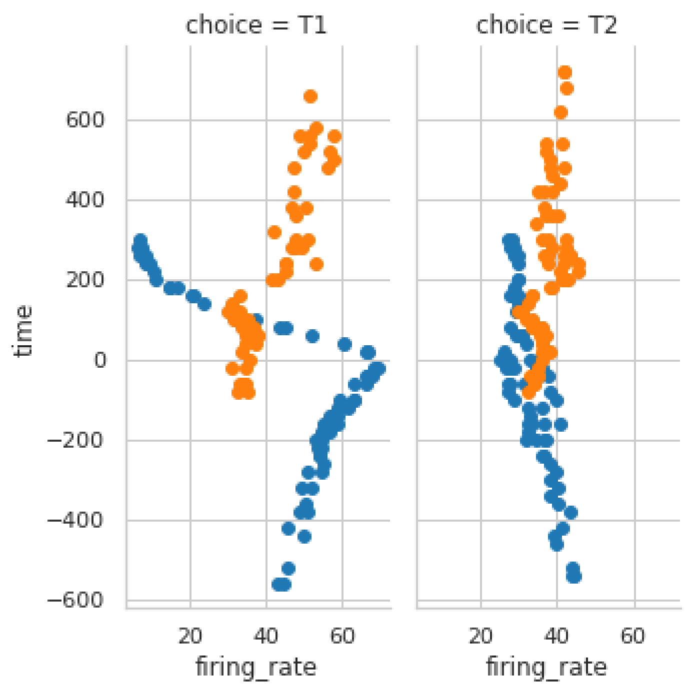


Рисунок 12.5 — Демонстрация работы с параметрами `height` и `aspect` класса `FacetGrid`

Изменим порядок отображения графиков и добавим легенду:

```
col_ord = ["T1", "T2"]
fg = sns.FacetGrid(dots_mod, col="choice", hue="align",col_order=col_ord)
fg.map(plt.scatter, "firing_rate", "time").add_legend()
```

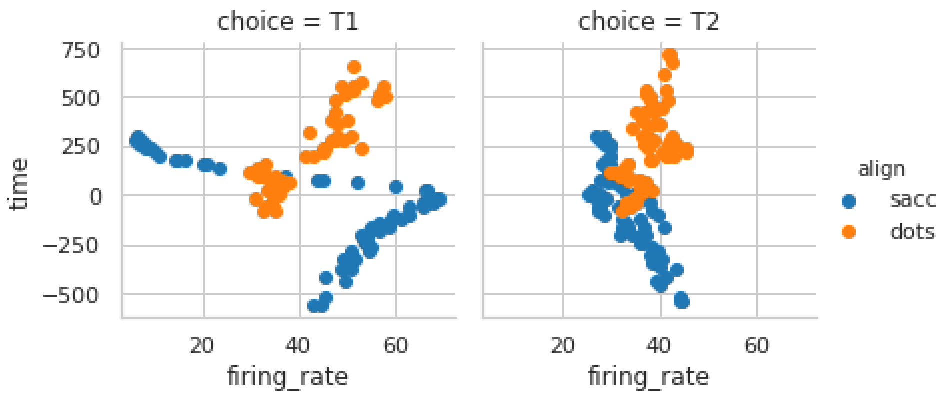


Рисунок 12.6 — Демонстрация работы с параметром `col_order` класса `FacetGrid`

Легенду можно поместить на поле с графиком, присвоив параметру `legend_out` значение `False`. Параметры `xlim`, `ylim` задают диапазоны для осей *x* и *y*:

```
ylim = (-1000, 1000)
xlim = (0, 100)
fg = sns.FacetGrid(dots_mod, col="choice", hue="align", xlim=xlim,
ylim=ylim)
fg.map(plt.scatter, "firing_rate", "time")
```

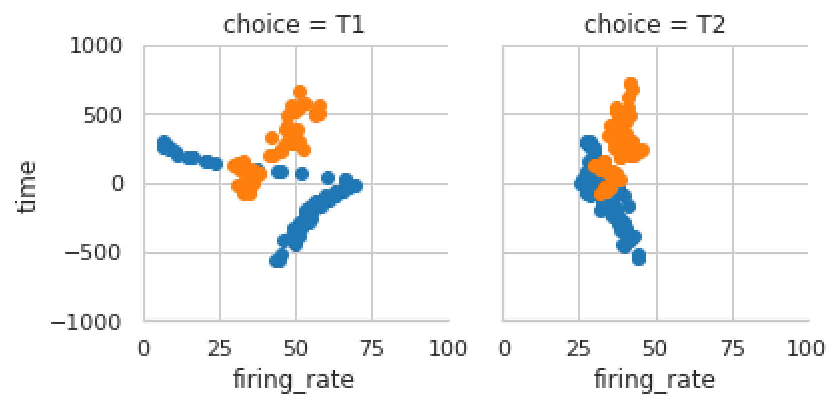


Рисунок 12.7 — Демонстрация работы с параметром `xlim` и `ylim` класса `FacetGrid`

Приведём несколько примеров работы с осями координат и сеткой. Ещё раз представим вариант диаграммы с параметрами по умолчанию:

```
fg.map(plt.scatter, "firing_rate", "time")
```

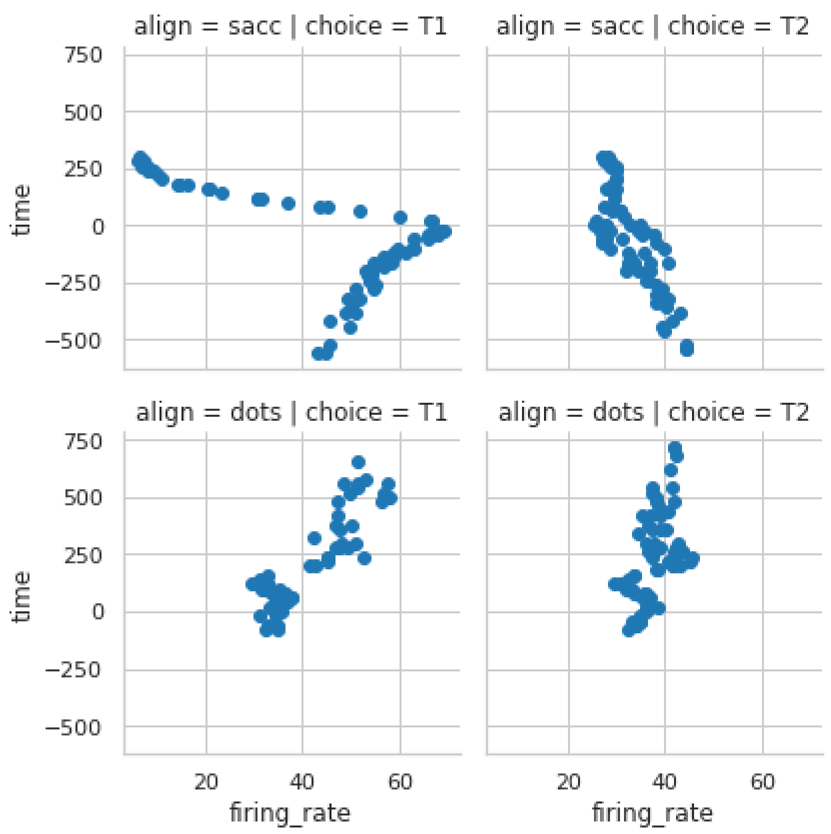


Рисунок 12.8 — Диаграмма с параметрами по умолчанию

Обратите внимание: ось *y* является общей для всех графиков строки, а *x* — для всех графиков столбца. Можно принудительно включить отображение отметок на осях для каждого графика с помощью параметров `sharex` и `sharey` передав им значения `False`:

```
fg = sns.FacetGrid(dots_mod, col="choice", row="align", sharex=False,
sharey=False)
fg.map(plt.scatter, "firing_rate", "time")
```

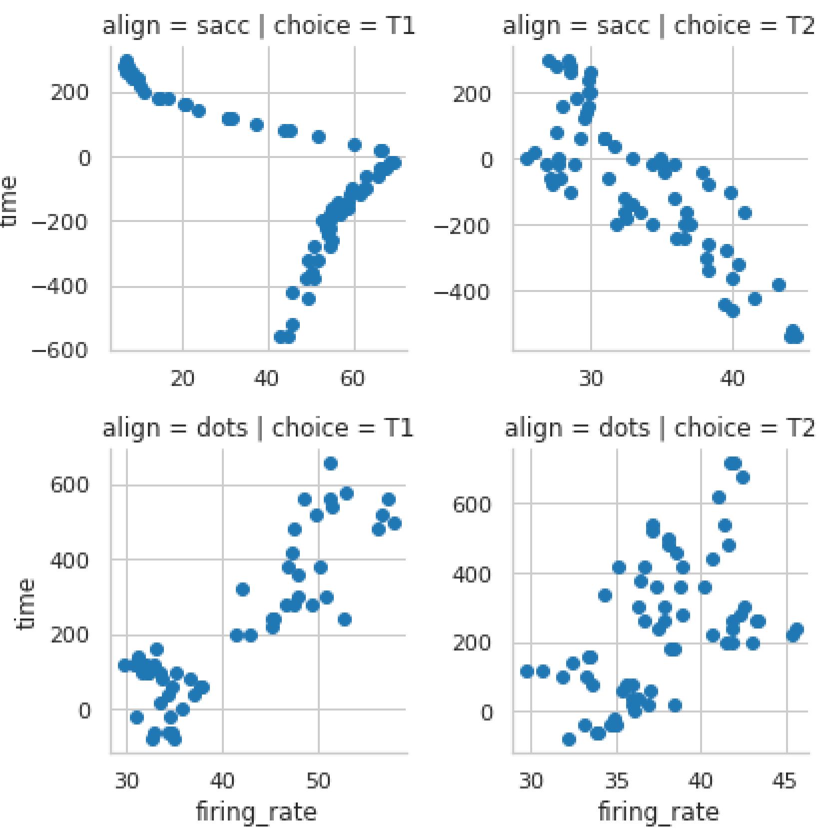


Рисунок 12.9 — Демонстрация работы с параметрами `sharex` и `sharey` класса `FacetGrid`

Если параметр `despine` оставить со значением по умолчанию, то мы получим открытые правую и верхнюю стороны поля графика:

```
fg = sns.FacetGrid(dots_mod, col="choice", hue="align")
fg.map(plt.hist, "firing_rate")
```

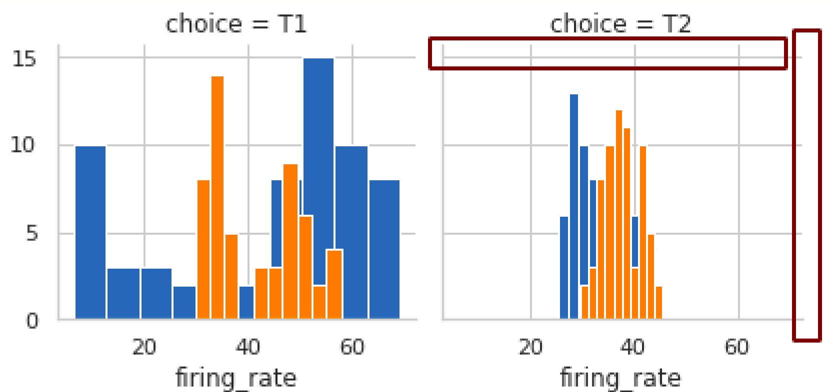


Рисунок 12.10 — Демонстрация работы с параметром `despine` класса `FacetGrid`

Присвоим ему значение `False`:

```
fg = sns.FacetGrid(dots_mod, col="choice", hue="align", despine=False)
fg.map(plt.hist, "firing_rate")
```

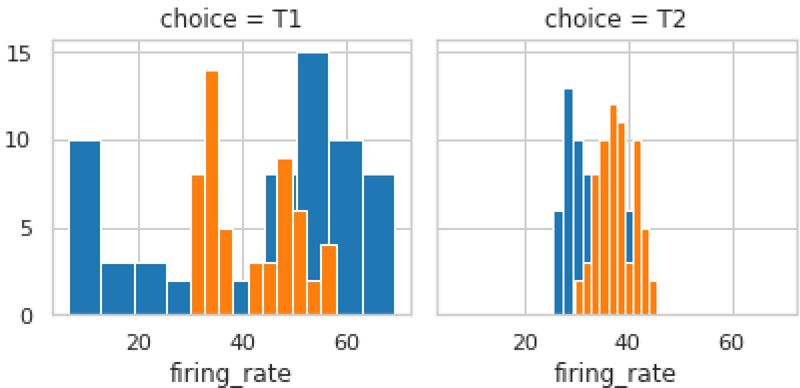


Рисунок 12.11 — Демонстрация работы с параметром `despine=False` класса `FacetGrid`

Как вы можете видеть, поля с графиками стали ограничеными со всех сторон.

12.2 *Pair*-сетка

Следующий инструмент управления компоновкой, который мы рассмотрим — это `PairPlot`. Класс `PairPlot` формирует сетку для построения диаграмм попарного сравнения выбранных признаков из переданного набора данных. *Seaborn* предоставляет удобный инструмент для работы с такого типа компоновкой — функцию `pairplot()`. Если же требуется более тонкая настройка, то в этом случае следует воспользоваться непосредственно классом `PairPlot` и его методами. Начнём наше знакомство с функции `pairplot()`.

12.2.1 Функция pairplot()

Функция `pairplot()` строит сетку элементами, которой являются графики попарного сравнения заданного набора признаков. Для лучшего визуального представления применим стиль `ticks`:

```
sns.set(style="ticks")
```

Загрузим набор данных *mpg* и извлечём из него подвыборку, содержащую только признаки `mpg`, `horsepower`, `displacement`, `origin`:

```
mpg = sns.load_dataset("mpg")
mpg_mod = mpg[["mpg", "horsepower", "displacement", "origin"]]
```

Построим диаграмму с помощью функции `pairplot()`:

```
sns.pairplot(mpg_mod)
```

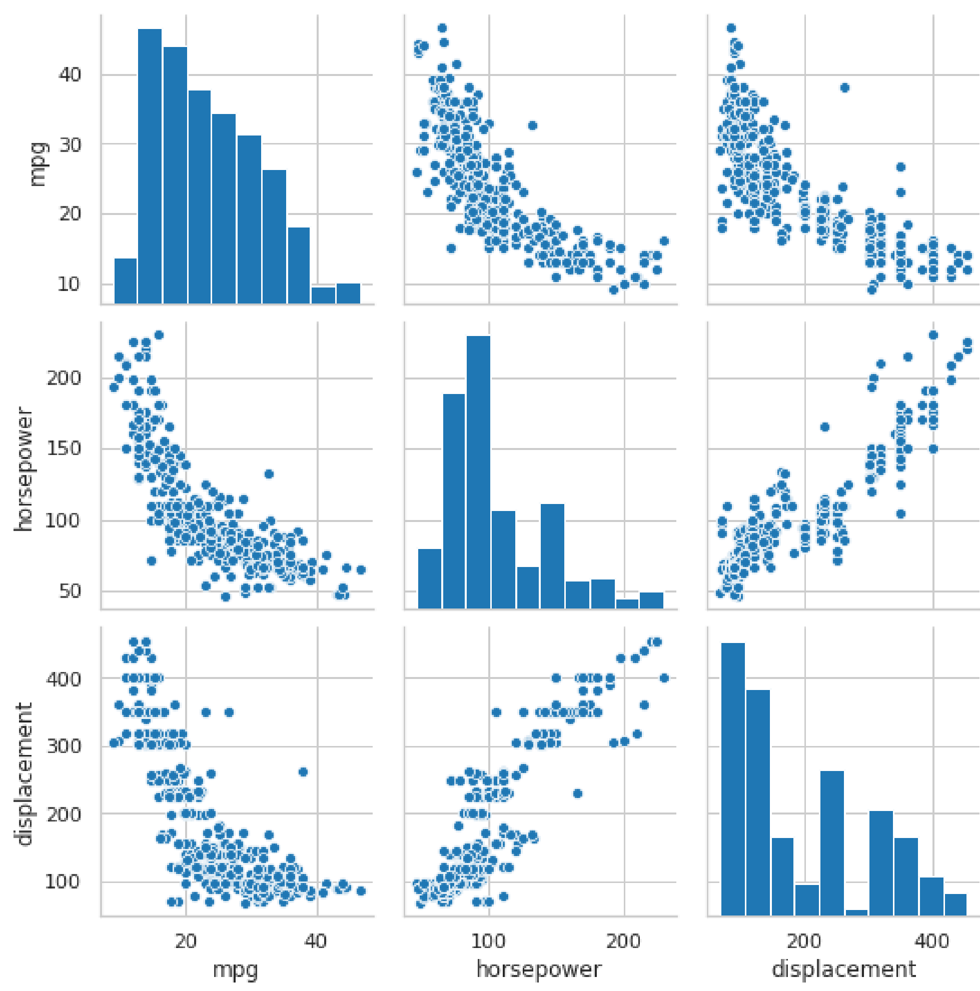


Рисунок 12.12 — Диаграмма, построенная с функцией pairplot()

Большинство параметров функции вам уже должны быть известны: `data`, `hue`, `hue_order`, `palette`, `markers`, `height`, `aspect`, `dropna`, их мы встречали при изучении других функций.

Приведём несколько примеров работы с ними. Добавим разделение по категориальному признаку *origin*, для точечной диаграммы зададим треугольный маркер, а также укажем высоту и соотношение сторон:

```
sns.pairplot(mpg_mod, hue="origin", markers="^", height=2, aspect=1.1)
```

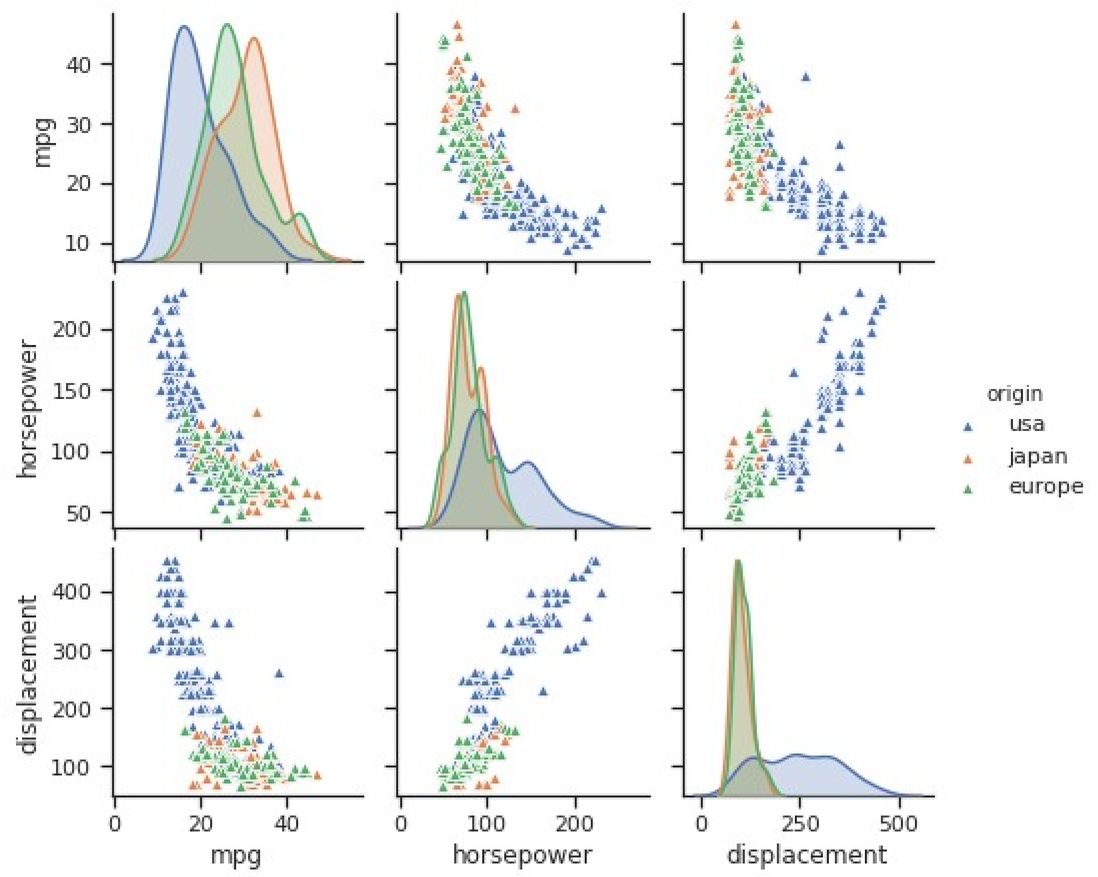


Рисунок 12.13 — Демонстрация работы с параметрами `hue`, `markers`, `height` и `aspect` функции `pairplot()`

Помимо перечисленных, функция `pairplot()` содержит следующие полезные параметры:

- `vars`: `list`, `optional`
  - Имена признаков из набора `data`, которые будут использоваться для отображения.

- `{x, y}_vars`: `list`, `optional`
  - Имена признаков из набора `data`, которые будут представлены на строках (`x_vars`) и столбцах (`y_vars`).
- `kind:{'scatter', 'reg'}`, `optional`
  - Тип графика для диаграмм, расположённых вне главной диагонали.
- `diag_kind`: `{'auto', 'hist', 'kde', None}`, `optional`
  - Тип графика на элементах главной диагонали.
- `corner`: `bool`, `optional`
  - Если параметр равен `True`, то диаграммы в правой верхней части (все что выше главной диагонали) отображаться не будут.

Выберем для отображения признаки *mpg* и *horsepower* из набора *mpg_mod* с помощью параметра `vars`:

```
sns.pairplot(mpg_mod, hue="origin", vars=["mpg", "horsepower"])
```

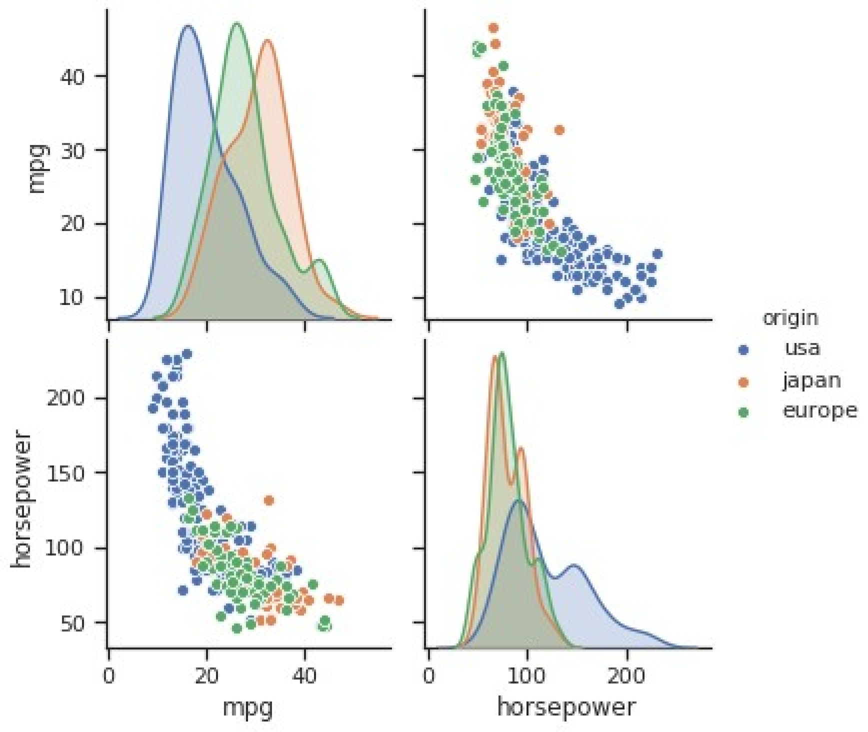


Рисунок 12.14 — Демонстрация работы с параметром `vars` функции `pairplot()`

Воспользуемся параметрами x_vars и y_vars для указания наборов признаков для осей *x* и *y*:

```
sns.pairplot(mpg_mod, hue="origin", x_vars=["mpg", "horsepower"],
y_vars=["displacement"])
```

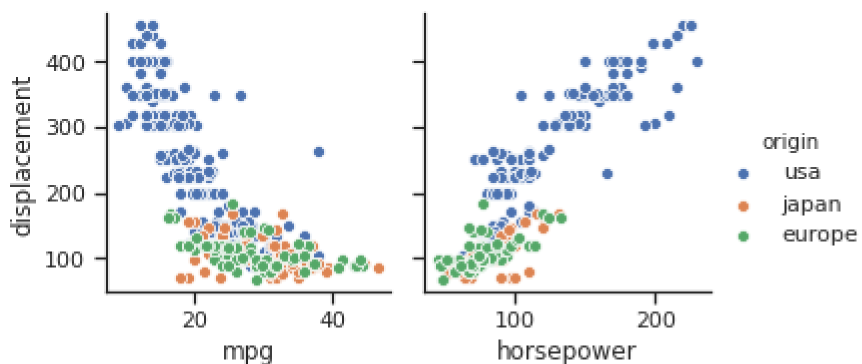


Рисунок 12.15 — Демонстрация работы с параметрами `x_vars` и `y_vars` функции `pairplot()`

Построим график с моделью регрессии вместо диаграммы рассеяния:

```
sns.pairplot(mpg_mod, vars=["horsepower", "displacement"], kind='reg')
```

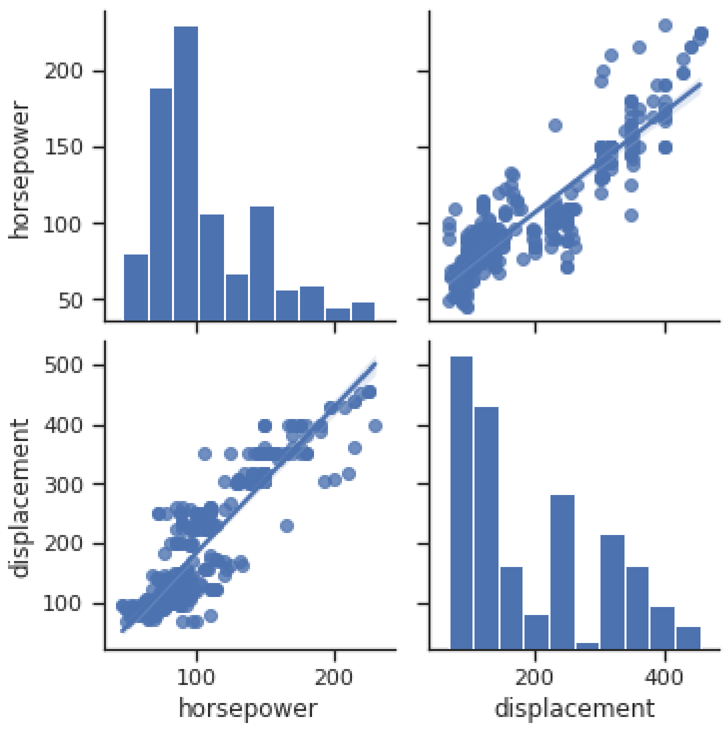


Рисунок 12.16 — Демонстрация работы с параметром `kind` функции `pairplot()`

Изменим тип диаграммы на главной диагонали с гистограммы на *KDE*:

```
sns.pairplot(mpg_mod, vars=["horsepower", "displacement"],
diag_kind='kde')
```

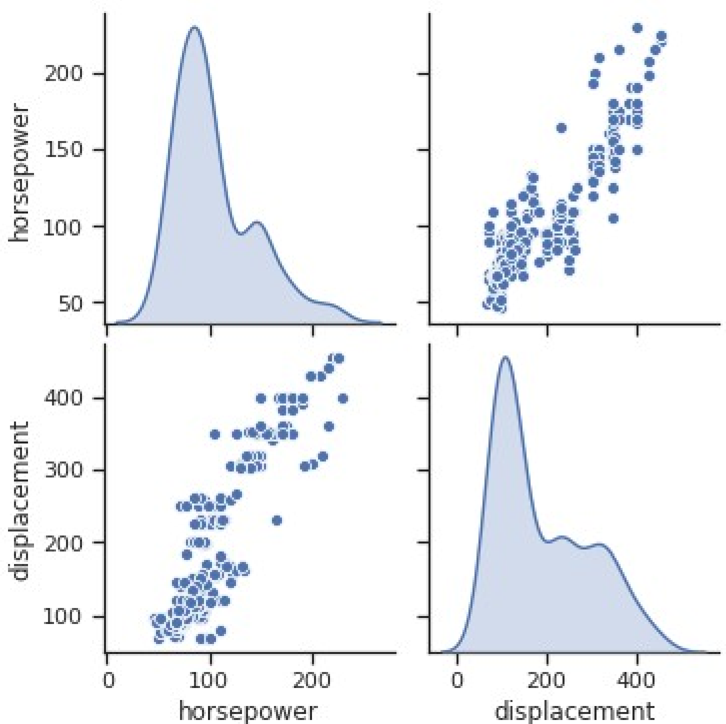


Рисунок 12.17 — Демонстрация работы с параметром diag_kind функции pairplot()

Уберём все диаграммы выше главной диагонали, для этого присвоим параметру corner значение True:

```
sns.pairplot(mpg_mod, vars=["horsepower", "displacement"], corner=True)
```

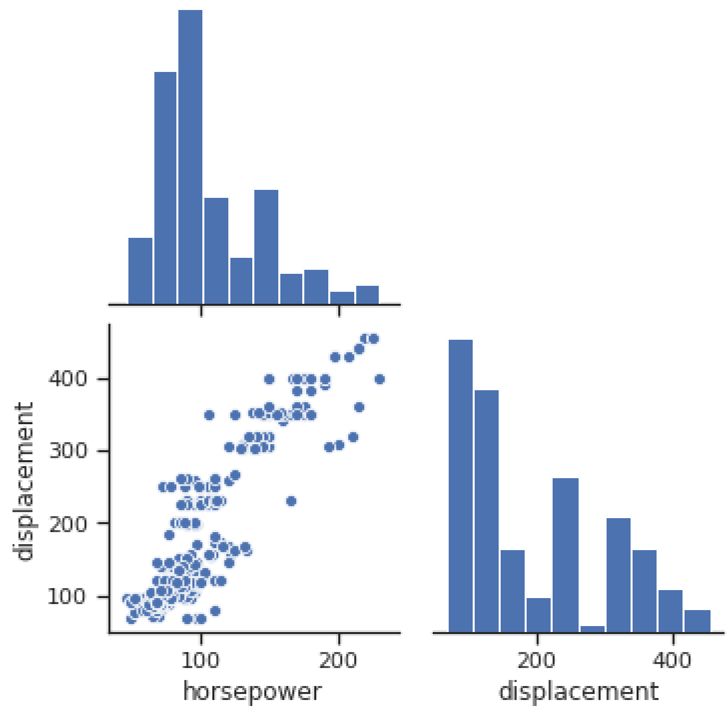


Рисунок 12.18 — Демонстрация работы с параметром `corner` функции `pairplot()`

12.2.2 Класс `PairPlot`

Конструктор класса `PairPlot` содержит параметры, частично совпадающие с теми, что были нами рассмотрены для функции `pairplot()`, к ним относятся: `data`, `hue`, `hue_order`, `palette`, `hue_kws`, `vars`, `{x, y}_vars`, `corner`, `height`, `aspect`, `dropna`, дополнительно есть два параметра для настройки подложки:

- `layout_pad`: `int` или `float`, `optional`
  - Расстояние между диаграммами.
- `Despine: bool`, `optional`
  - Если параметр равен `True`, то будут убраны ограничительные линии в верхней и правой частях поля графика.

Для построения диаграммы, по аналогии с классом `FacetGird`, используется метод `map()`, в качестве аргумента ему передаётся

функция для построения графика. PairPlot предоставляет ещё ряд функций, для управления представлением диаграмм на главной диагонали, верхней и нижней частях сетки: map_diag(), map_offdiag(), map_lower(), map_upper().

Подготовим набор данных для работы из уже загруженного ранее *mpg*:

```
mpg_mod = mpg[["mpg", "weight", "displacement", "origin"]]
```

Воспользуемся методом map() объекта класса PairPlot для построения диаграммы:

```
pg = sns.PairGrid(mpg_mod)
pg.map(plt.scatter)
```

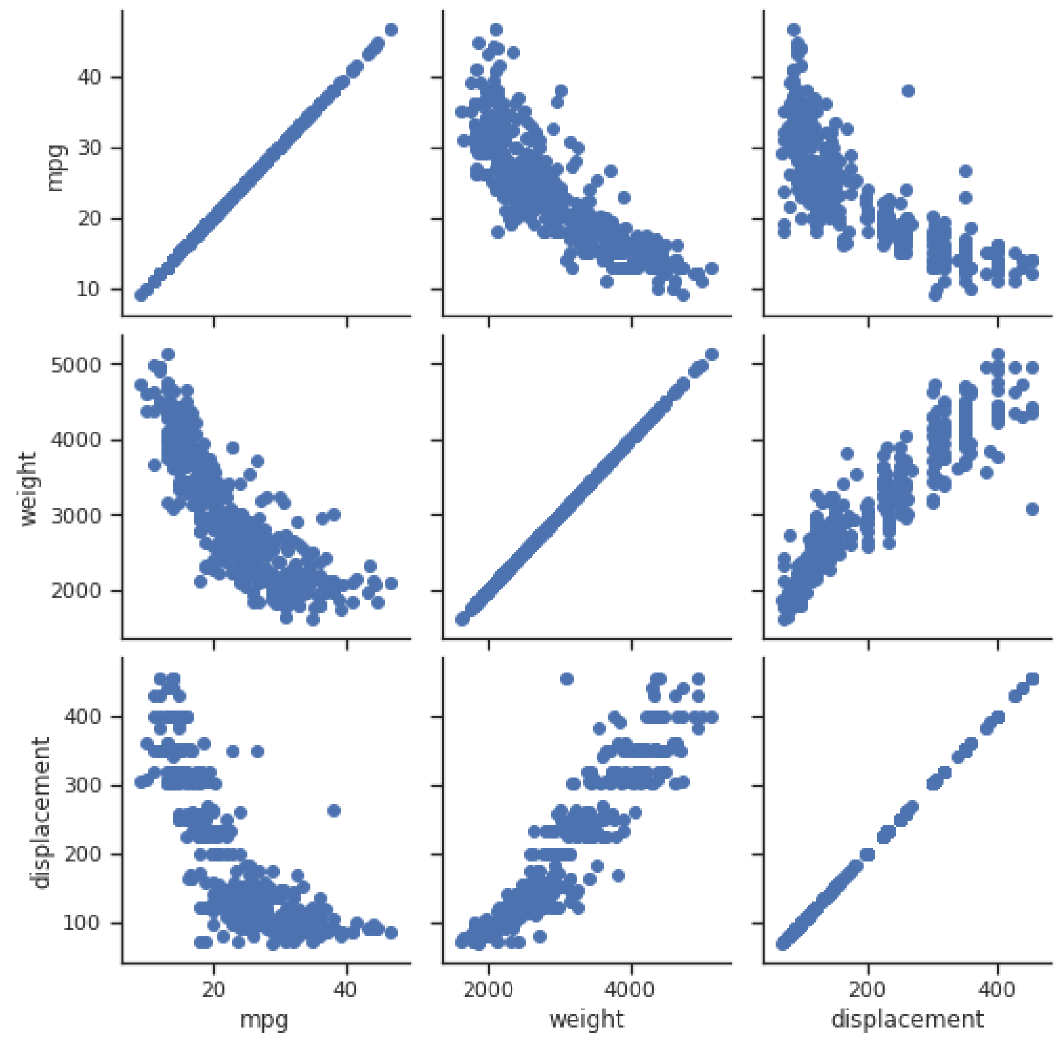


Рисунок 12.19 — Демонстрация работы с классом PairGrid

Добавим разделение по признаку *origin*:

```
pg = sns.PairGrid(mpg_mod, hue='origin')
pg.map(plt.scatter)
```

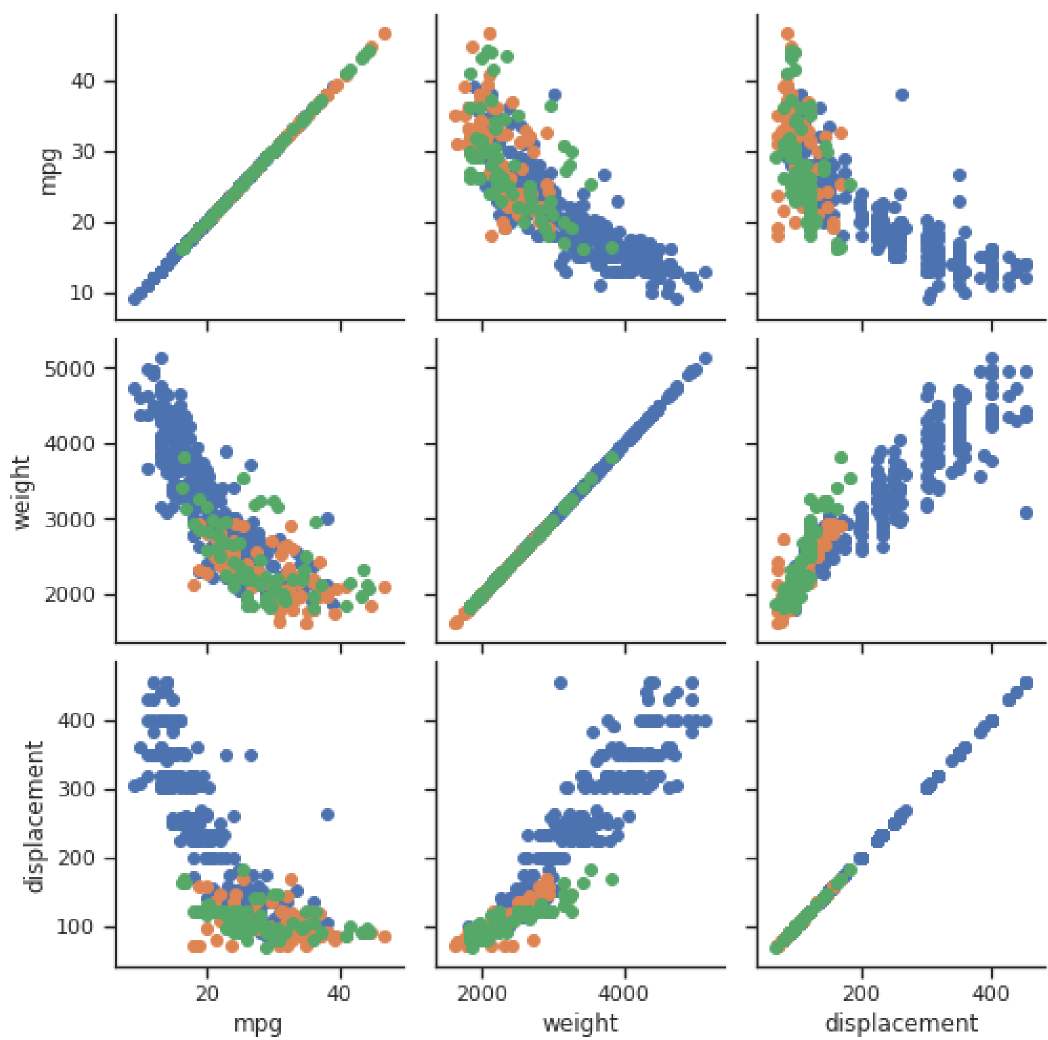


Рисунок 12.20 — Демонстрация работы с параметром hue класса `PairGrid`

Воспользуемся функциями `map_diag()` и `map_offdiag()` для задания типов диаграмм на главной диагонали и вне ее:

```
pg = sns.PairGrid(mpg_mod)
pg.map_diag(sns.kdeplot)
pg.map_offdiag(plt.scatter)
```

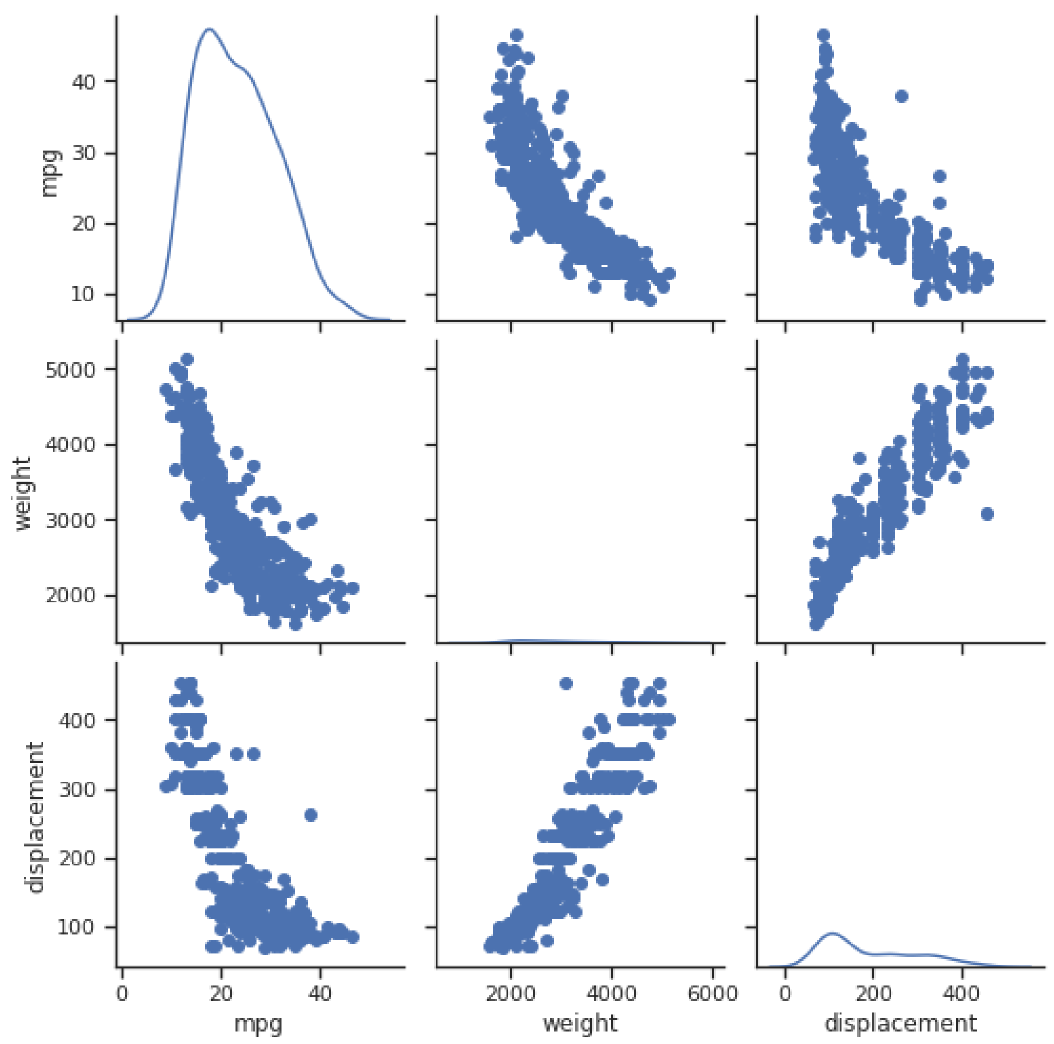


Рисунок 12.21 — Демонстрация работы с функциями `map_diag()` и `map_offdiag()` класса `PairGrid`

Вариант работы с `map_diag()`, `map_lower()` и `map_upper()`:

```
pg = sns.PairGrid(mpg_mod)
pg.map_diag(plt.hist)
pg.map_lower(plt.scatter)
pg.map_upper(sns.kdeplot)
```

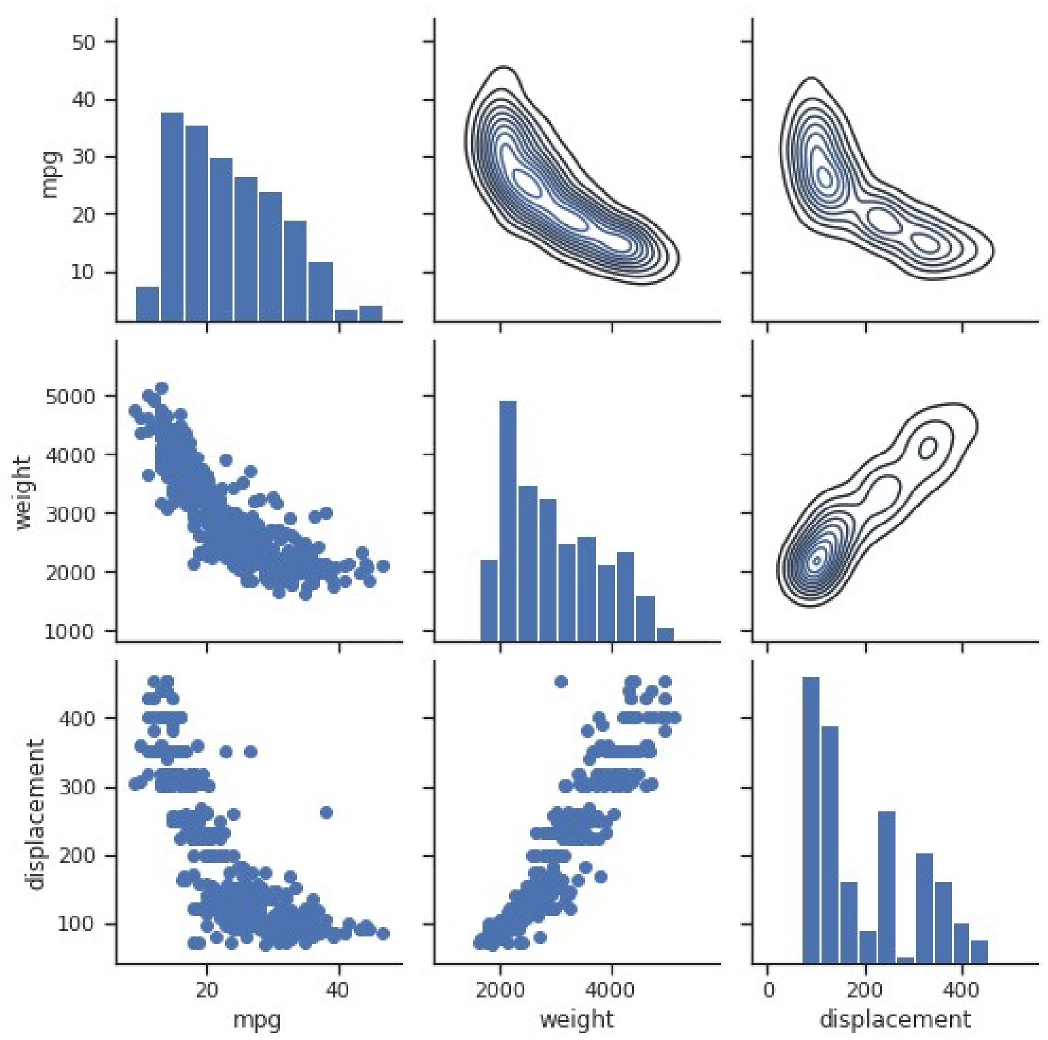


Рисунок 12.22 — Демонстрация работы с функциями `map_diag()`, `map_lower()` и `map_upper()` класса `PairGrid`

12.3 *Joint*-сетка

Завершающей темой в рамках обзора инструментов для управления компоновкой будет обзор возможностей класса `JointGrid`. Этот класс строит сетку для представления двумерного распределения на основной диаграмме и двух отдельных диаграмм, визуализирующих одномерные распределения каждого из параметров по отдельности. Также как в случае с `PairGrid` для упрощения работы с `JointGrid` библиотека *seaborn* предоставляет функцию `jointplot()`, для более тонкой настройки используйте объект класса `JointGrid` непосредственно.

12.3.1 Функция `jointplot()`

Функция `jointplot()` содержит следующий набор уже знакомых вам параметров: `x`, `y`, `data`, `color`, `height`, `dropna`, `{x, y}lim`.

Загрузим набор данных *iris*:

```
iris = sns.load_dataset("iris")
```

Построим диаграмму с помощью функции `jointplot()`:

```
sns.jointplot(x='sepal_length', y='sepal_width', data=iris)
```

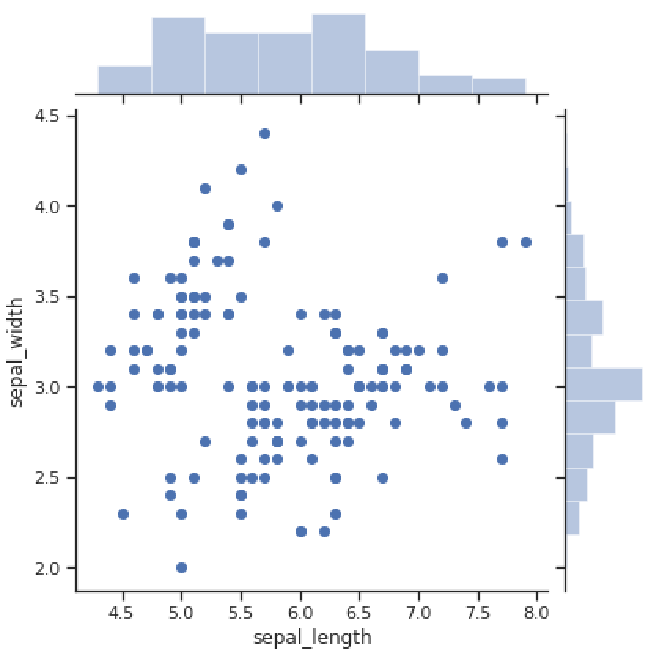


Рисунок 12.23 — Демонстрация работы функции `jointplot()`

Изменим цвет и размер диаграммы:

```
sns.jointplot(x='sepal_length', y='sepal_width', data=iris,
color='orange', height=5)
```

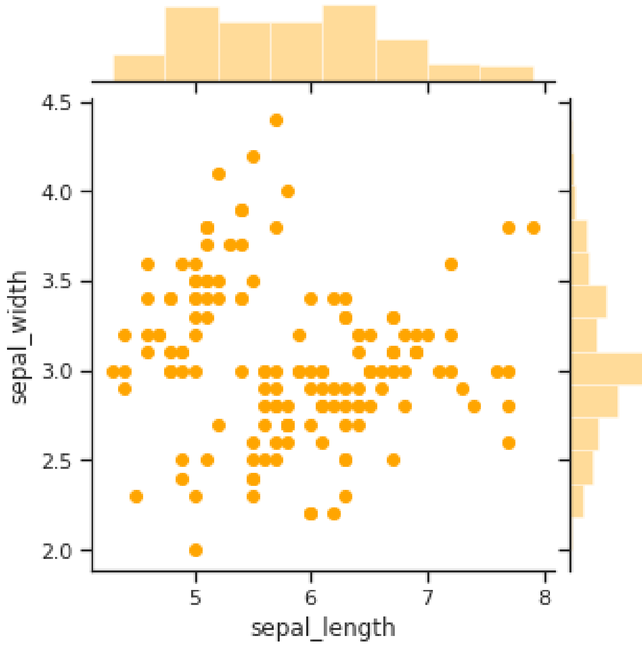


Рисунок 12.24 — Демонстрация работы с параметрами `color` и `height` функции `jointplot()`

Укажем диапазоны для осей координат:

```
sns.jointplot(x='sepal_length', y='sepal_width', data=iris, xlim=(0, 10),
ylim=(0, 10))
```

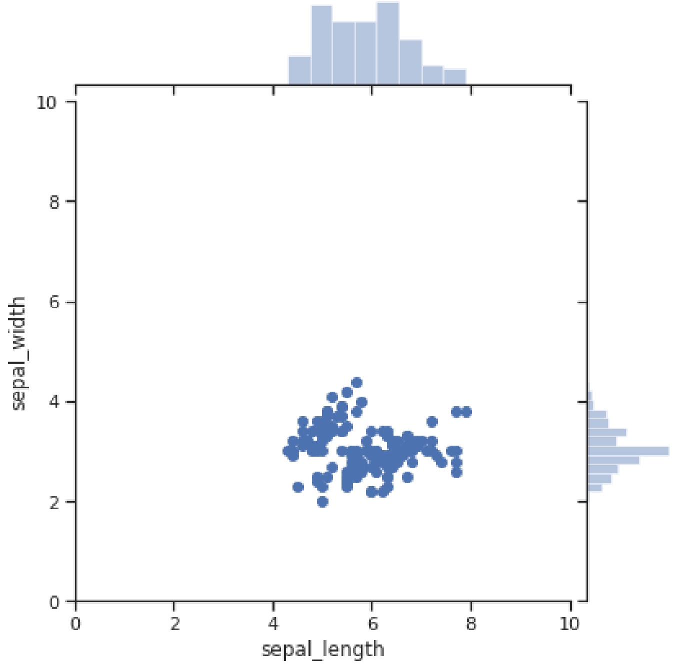


Рисунок 12.25 — Демонстрация работы с параметрами `xlim` и `ylim` функции `jointplot()`

Для задания типа центральной диаграммы используется параметр `kind`, он может иметь одно из следующих значений `{'scatter' | 'reg' | 'resid' | 'kde' | 'hex'}`.

Ниже приведены примеры, демонстрирующие внешний вид диаграмм, полученных при различных значениях `kind`. Вариант с `kind='scatter'` мы видели на предыдущих примерах.

Диаграмма с параметром `kind='reg'`:

```
sns.jointplot(x='sepal_length', y='petal_length', data=iris, kind="reg")
```

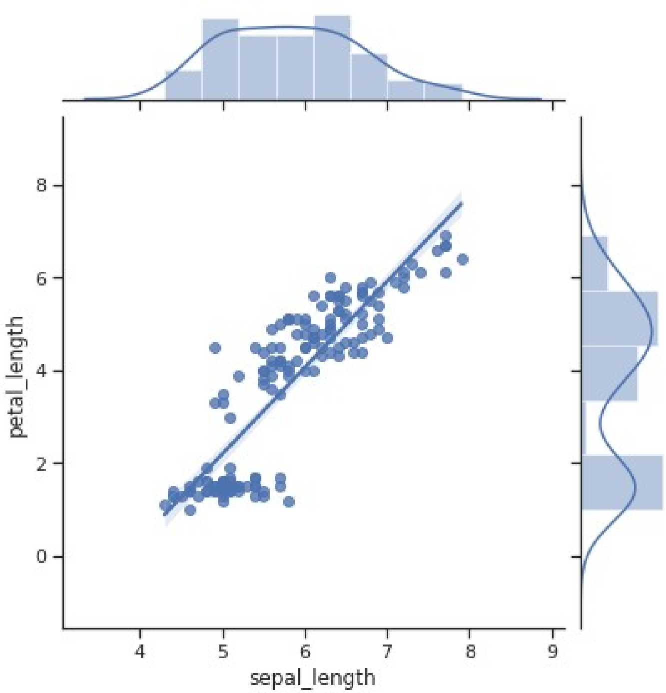


Рисунок 12.26 — Демонстрация работы с параметром `kind='reg'` функции `jointplot()`

Диаграмма с параметром kind='kde':

```
sns.jointplot(x='sepal_length', y='petal_length', data=iris, kind="kde")
```

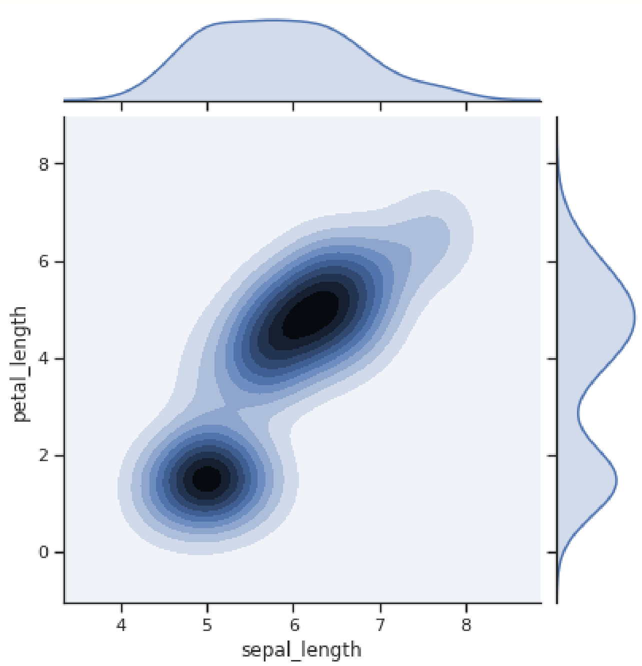


Рисунок 12.27 — Демонстрация работы с параметром kind='kde' функции jointplot()

Диаграмма с параметром kind='hex':

```
sns.jointplot(x='sepal_length', y='sepal_width', data=iris, kind="hex")
```

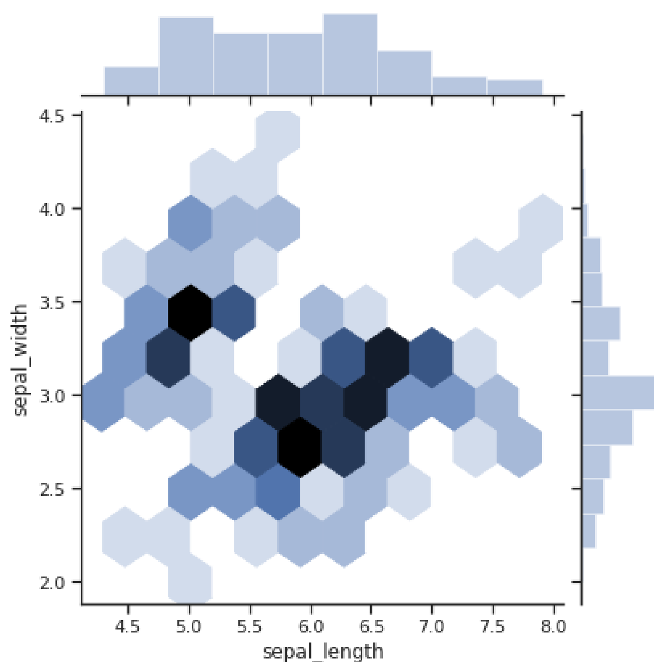


Рисунок 12.28 — Демонстрация работы с параметром kind='hex' функции jointplot()

Диаграмма с параметром `kind='resid'`:

```
sns.jointplot(x='sepal_length', y='sepal_width', data=iris, kind="resid")
```

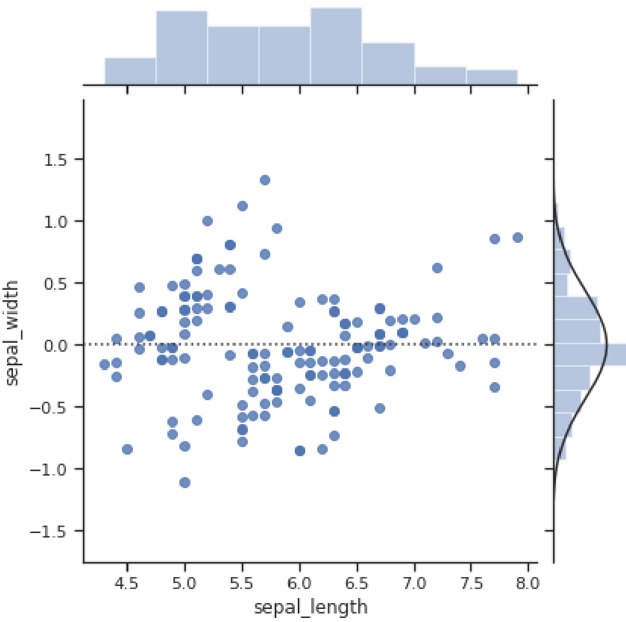


Рисунок 12.29 — Демонстрация работы с параметром `kind='resid'` функции `jointplot()`

За управление соотношением размеров центральной и боковых диаграмм отвечает параметр `ratio`:

```
sns.jointplot(x='sepal_length', y='sepal_width', data=iris, ratio=2)
```

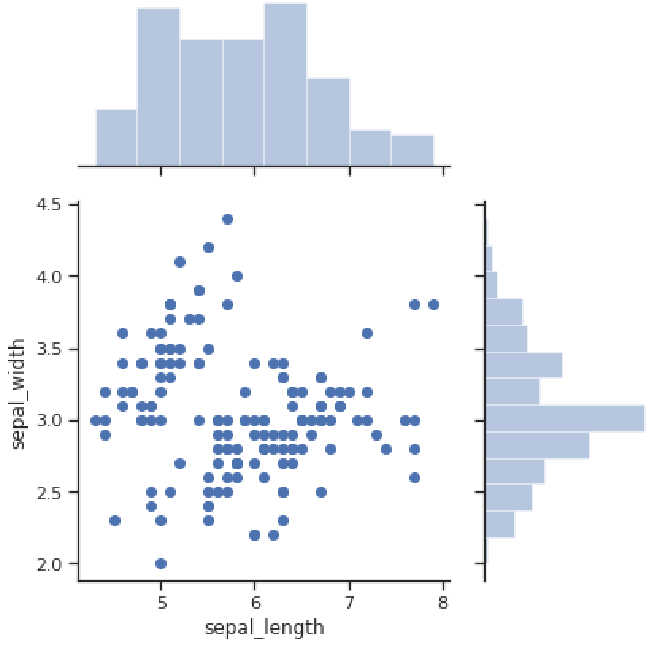


Рисунок 12.30 — Демонстрация работы с параметром `ratio` функции `jointplot()`

Параметр `space` определяет зазор между центральной и боковыми диаграммами:

```
sns.jointplot(x='sepal_length', y='sepal_width', data=iris, space=2)
```

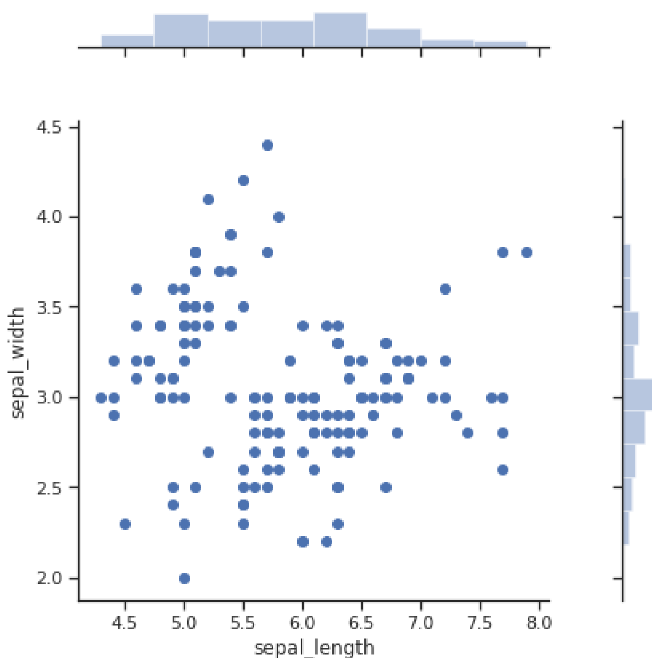


Рисунок 12.31 — Демонстрация работы с параметром `space` функции `jointplot()`

12.3.2 Класс JointPlot

Параметры конструктора класса JointPlot совпадают с частью аргументов функции jointplot(): x, y, data, height, ratio, space, dropna, {x,y}lim. Для вывода диаграммы используйте метод plot() либо методы plot_joint(), plot_marginal(). В метод plot() в качестве параметров передаются функции для построения центральной и боковых диаграмм:

```
jg = sns.JointGrid(x='sepal_length', y='sepal_width', data=iris)
jg.plot(sns.scatterplot, sns.kdeplot)
```

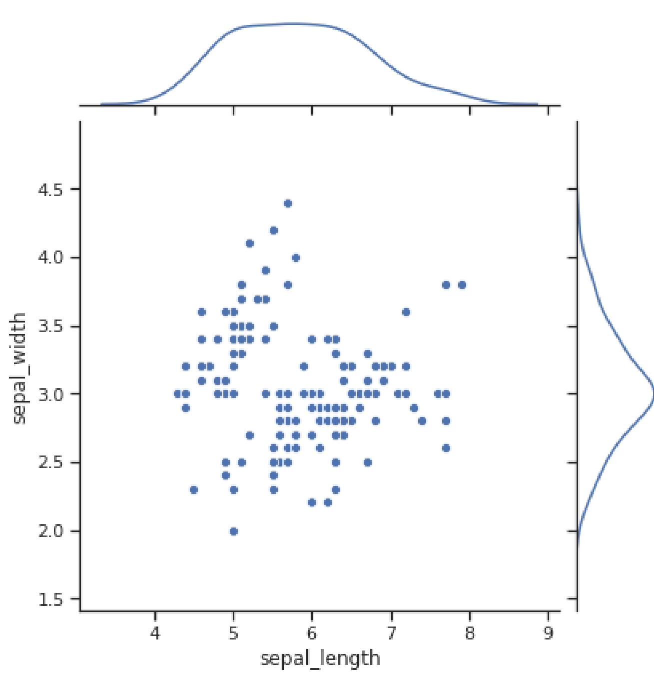


Рисунок 12.32 — Демонстрация работы функции plot() класса JointGrid

Функции plot_joint() и plot_marginals() позволяют изолированно указать и передать параметры для функций построения центральной и боковых диаграмм:

```
jp = sns.JointGrid(x='sepal_length', y='sepal_width', data=iris)
jp.plot_joint(sns.regplot, color="g")
jp.plot_marginals(sns.distplot, color='r')
```

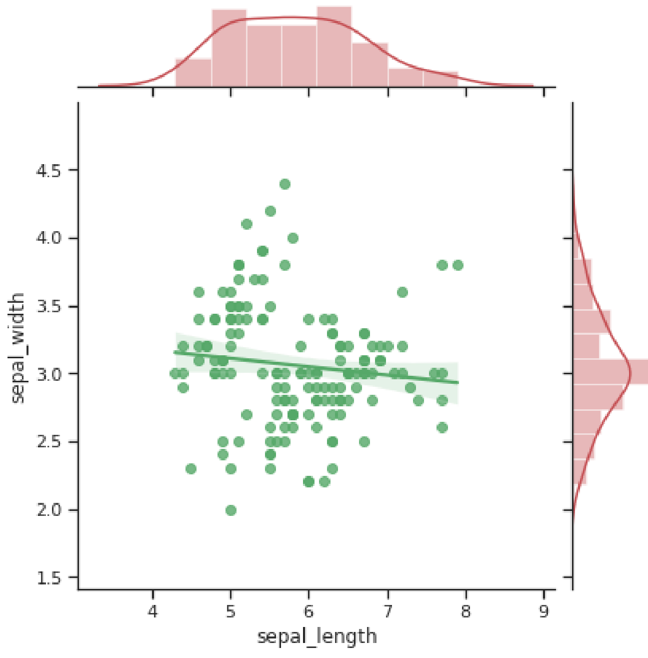


Рисунок 12.33 — Демонстрация работы с функциями `plot_joint()` и `plot_marginals()`

```
jp = sns.JointGrid(x='sepal_length', y='sepal_width', data=iris)
jp.plot_joint(sns.regplot, color="g")
jp.plot_marginals(sns.kdeplot, shade=True, color='r')
```

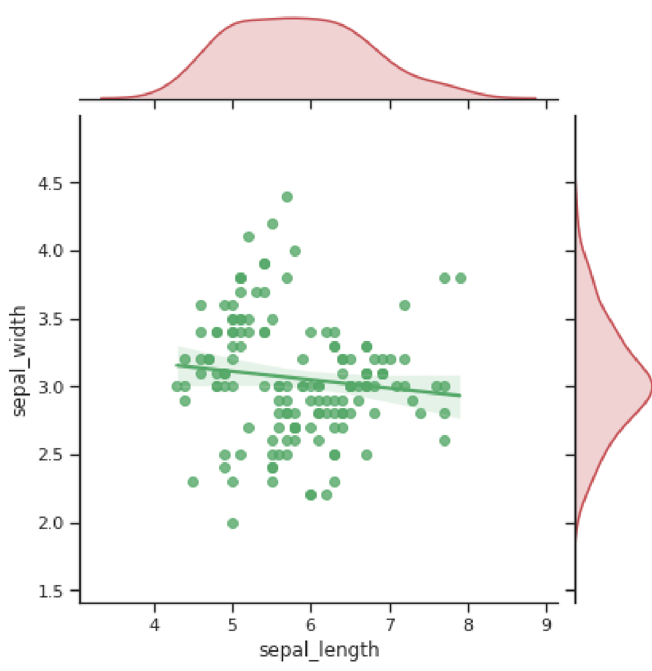


Рисунок 12.34 — Демонстрация работы с функциями `plot_joint()` и `plot_marginals()`

Часть III. Библиотека *Mayavi*

Введение

Mayavi представляет собой набор инструментов для решения задач *2D*/*3D*-визуализации. В состав *Mayavi* входит инструмент с графическим интерфейсом, через который удобно управлять *2D/3D* представлением данных, он называется *mayavi2*. Вторым крупным компонентом является набор пакетов и модулей для языка *Python*, через который можно управлять созданием, настройкой и запуском рендеринга *2D/3D*-моделей. Идейно принципы работы с этой библиотекой похожи на те, которым мы следовали, когда изучали *Matplotlib* или *seaborn*. *Mayavi* является обёрткой над *VTK* — *Visualization Toolkit* — мощной открытой библиотекой для визуализации.

Mayavi предоставляет следующие возможности:

- визуализация скалярных, векторных и тензорных данных в виде *2D* или *3D* представления;
- поддержка языка *Python*;
- возможность расширения функционала, за счёт разработки собственных модулей, фильтров и т.п.;
- поддержка различных форматов файлов: *VTK* (*XML* и более старые варианты), *PLOT3D*;
- сохранение результатов работы в графических файлах различных форматов;
- удобные инструменты для быстрого построения графиков через `mlab`.

Суть работы с *Mayavi* заключается в построении потоков данных (*pipelines*). На первом шаге данные загружаются в специальные объекты (*data sources*). Можно использовать как непосредственно *VTK*-файлы, так и генерировать *VTK*-структуры из `numpy` массивов (или других структур, для которых это возможно). Далее происходит обработка данных (процессинг) с помощью фильтров, результаты обработки, передаются в модуль визуализации. Область, в которой выводится результат рендеринга модели, называется сценой (*Scene*).

Глава 13. Быстрый старт

13.1 Установка

Для установки *Mayavi* можно воспользоваться пакетным менеджером *pip* либо *conda* (если у вас установлена *Anaconda*), или собрать библиотеку из исходников.

Для установки через *pip* воспользуйтесь следующими командами:

```
pip install mayavi
pip install PyQt5
```

Для установки из исходников:

```
git clone https://github.com/enthought/mayavi.git
cd mayavi
pip install -r requirements.txt
pip install PyQt5
python setup.py install
```

Чтобы проверить корректно ли *Mayavi* установилась на ваш ПК, откройте консоль и введите в ней команду:

```
mayavi2
```

В результате должна загрузиться оболочка, внешний вид которой представлен на рисунке 13.1.

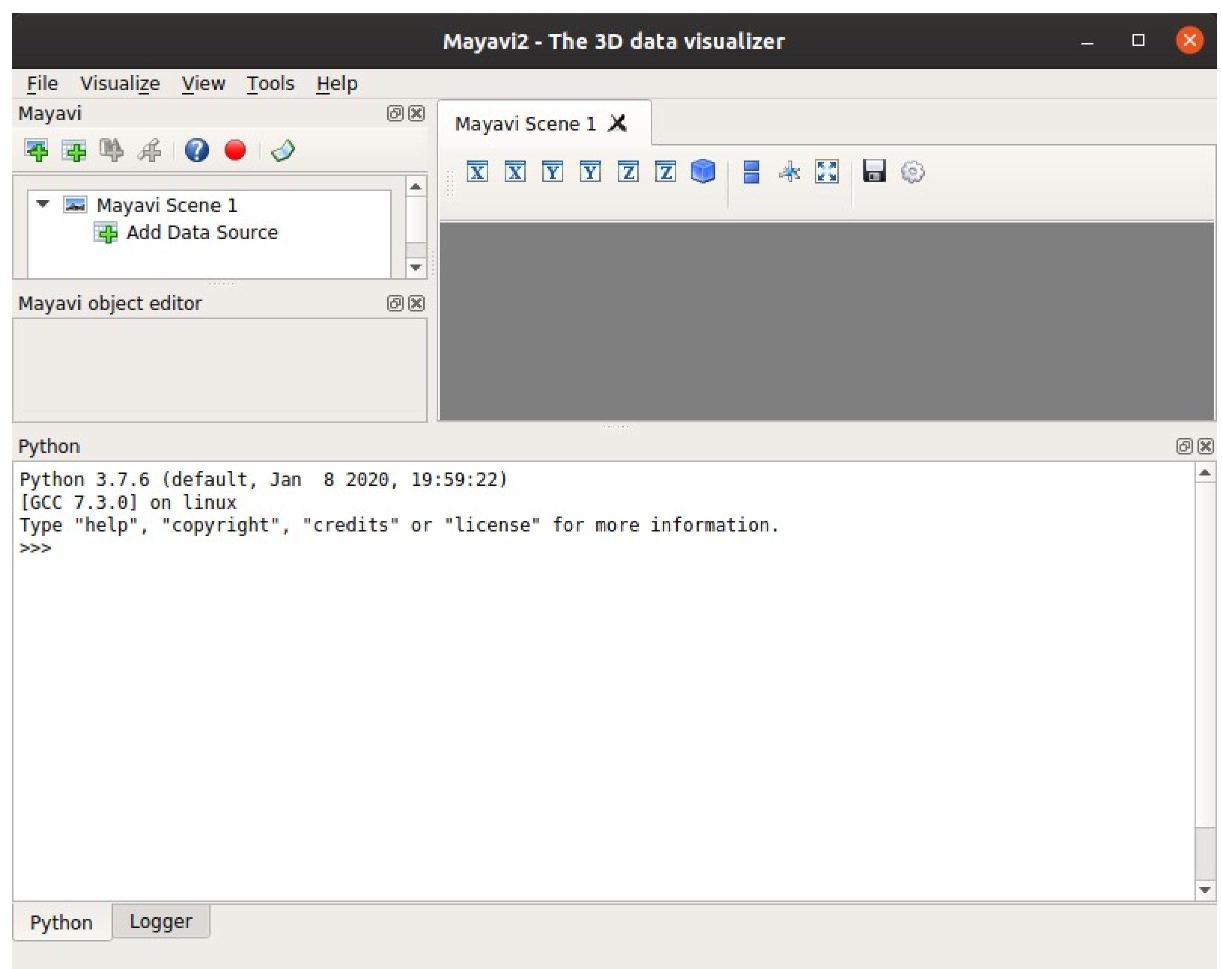


Рисунок 13.1 — Программа *mayavi2* с графическим интерфейсом для *2D/3D*-визуализации

13.2 Быстрый старт

Mayavi предполагает несколько способов работы с ней:

1. Работа с *GUI* приложением *mayavi2.*
2. Разработка программ на языке *Python*, использующих библиотеку *Mayavi*. Тут возможны два варианта:
    1. Разработка *Python*-модулей.
    2. Разработка *GUI* приложений со встраиваемым движком *Mayavi*.
    3. Использование *Jupyter notebook*.

Разработку *GUI* приложений со встраиваемым движком *Mayavi* мы рассматривать не будем. Если кратко, то суть этой возможности заключается в том, что вы можете создавать собственные приложения с графическим интерфейсов с внедрёнными в них элементами *Mayavi* для визуализации данных. Таким образом можно получать решения, подготовленные под определённые задачи. Для внедрения элементов *Mayavi* используется *Traits*[19]. Более подробно о том, как это делать можете прочитать в официальной документации[20].

Начнём наше знакомство с *GUI* приложения *Mayavi2*.

13.2.1 Работа с GUI приложением Mayavi2

Для запуска *Mayavi2* откройте терминал и введите:

```
mayavi2
```

В результате появится окно, представленное на рисунке 13.2.

19 https://docs.enthought.com/traits/

20 https://docs.enthought.com/mayavi/mayavi/building_applications.html

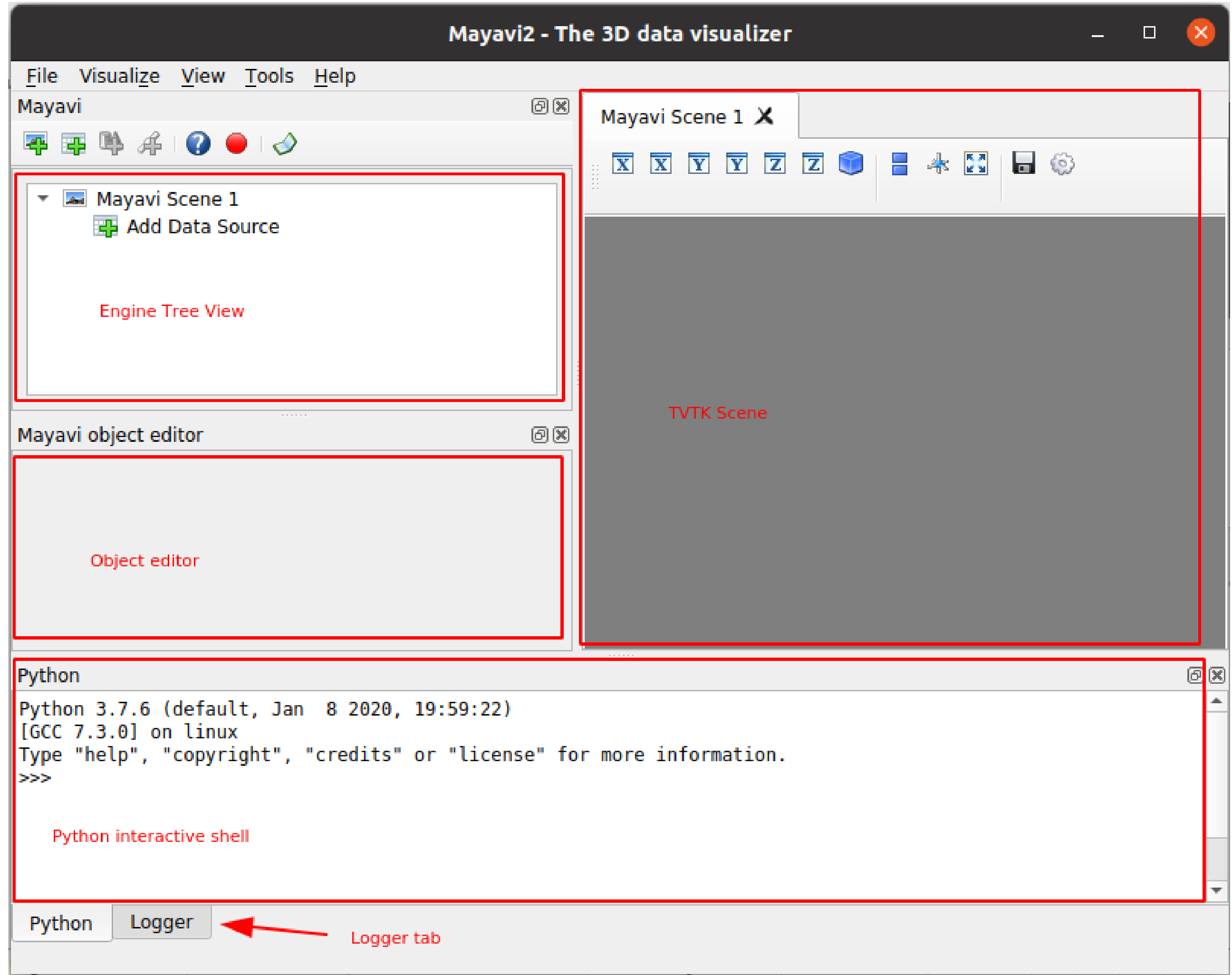


Рисунок 13.2 — Mayavi2 с элементами окна программы

На рисунке указаны основные элементы окна программы:

- *Engine Tree View;*
- *TVTK Scene;*
- *Object Editor;*
- *Python Interactive Shell;*
- *Logger tab.*

Для того чтобы поэкспериментировать с программой, загрузим набор демо данных, для этого на панели меню выберете *File->Load data -> Create Parametric Surface Source*. После этого в области "*Engine Tree View*" будет добавлен источник данных:

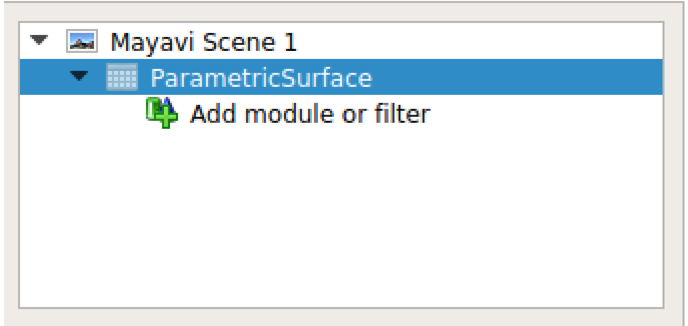


Рисунок 13.3 — *Engine Tree View* с источником данных

Выберете "*Add module or filter*" в дереве элементов, после этого в "*Object Editor*" вы увидите список модулей и фильтров, разделённый на две вкладки. Выберете модуль *Surface* из вкладки "*Visualization modules*" (нужно два раза щёлкнуть по элементу левой кнопкой мыши (см. рисунок 13.4).

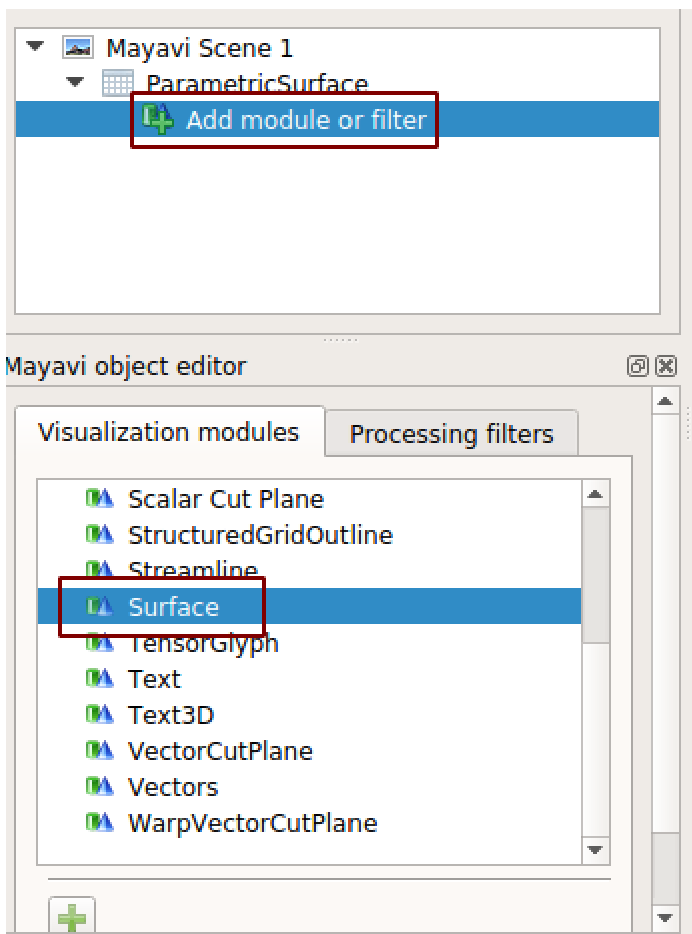


Рисунок 13.4 — Добавление модуля *Surface*

Если вы все сделали правильно, то на сцене должно появиться изображение как на рисунке 13.5.

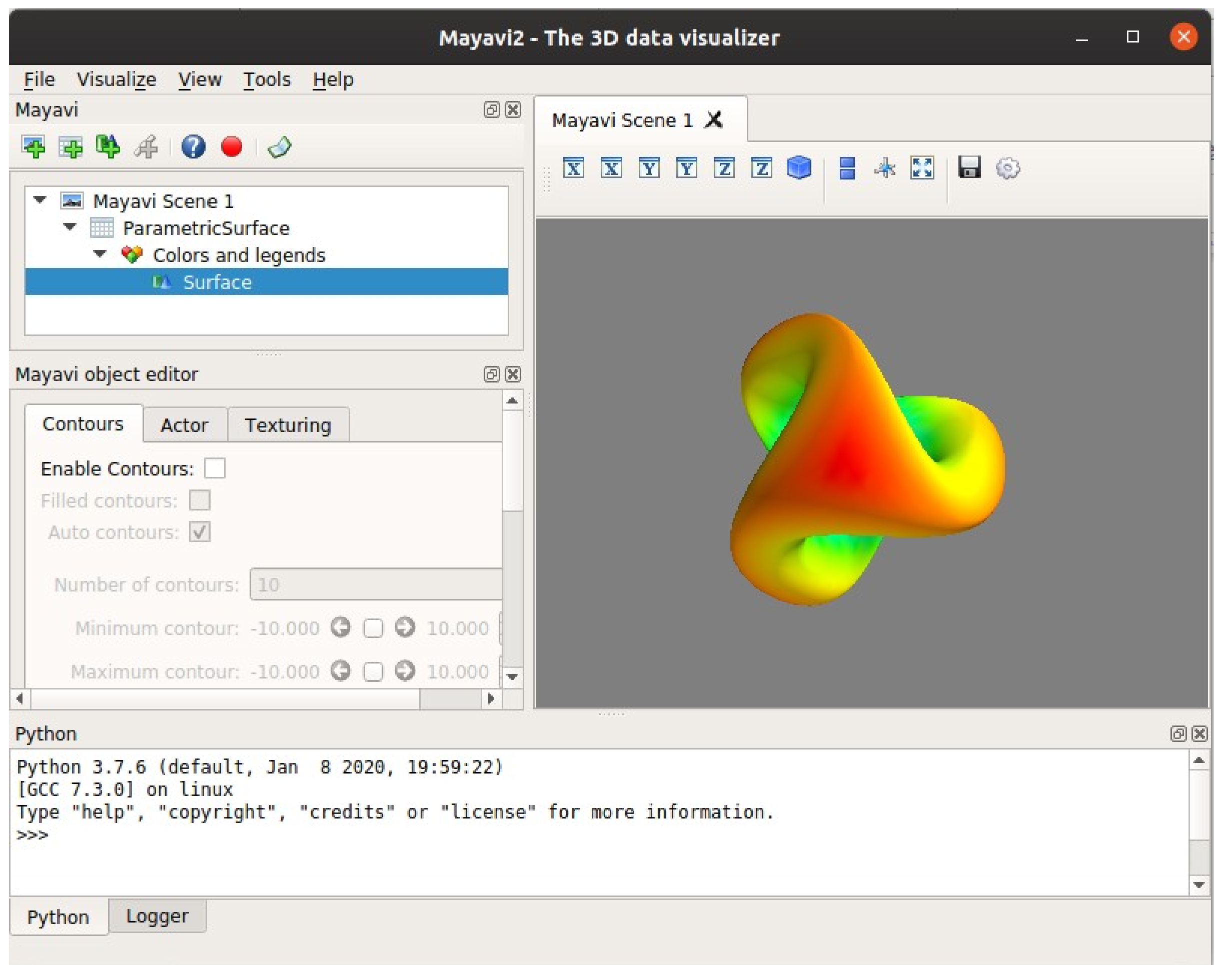


Рисунок 13.5 — Визуализация загруженного набора демо-данных

Вернувшись к элементу "*ParametricSurface*" в "*Engine Tree View*", можно поменять набор данных, по которому строится *3D*-модель.

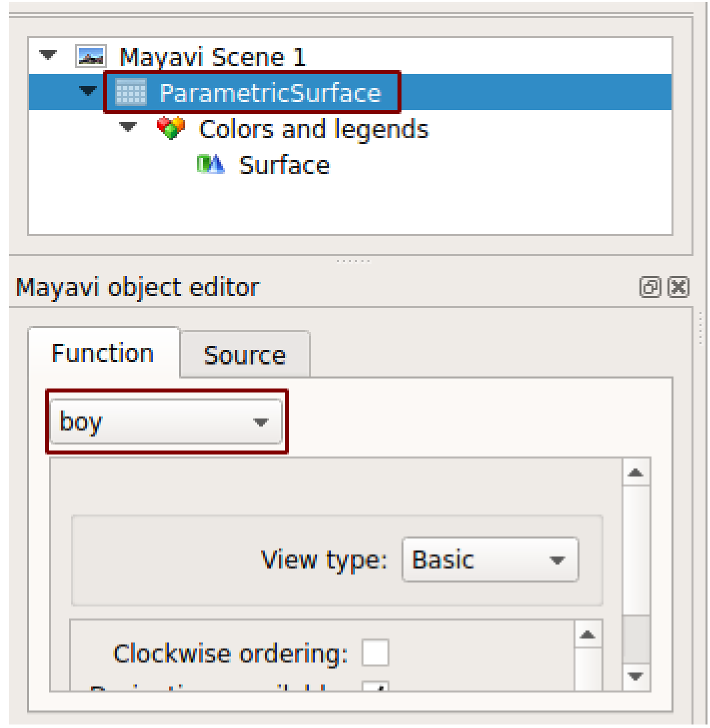


Рисунок 13.6 — Изменение набора данных

В нашем случае выбран вариант *boy*, поменяем его на *dini*:

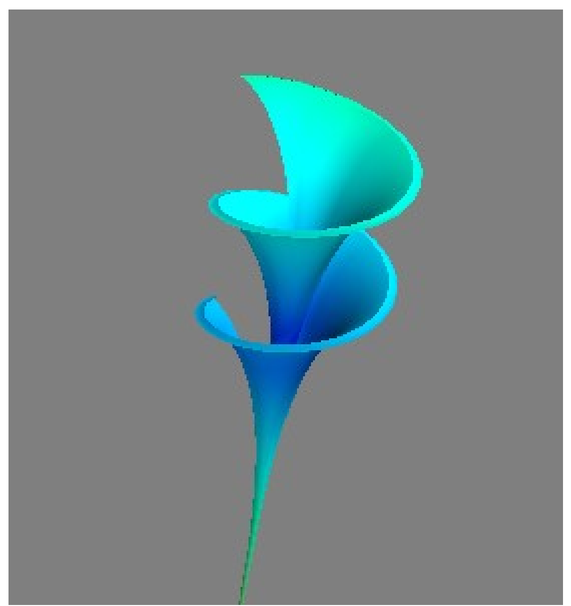


Рисунок 13.7 — Визуализация набора данных *dini* на сцене

Пример с лентой Мебиуса (*mobius*):

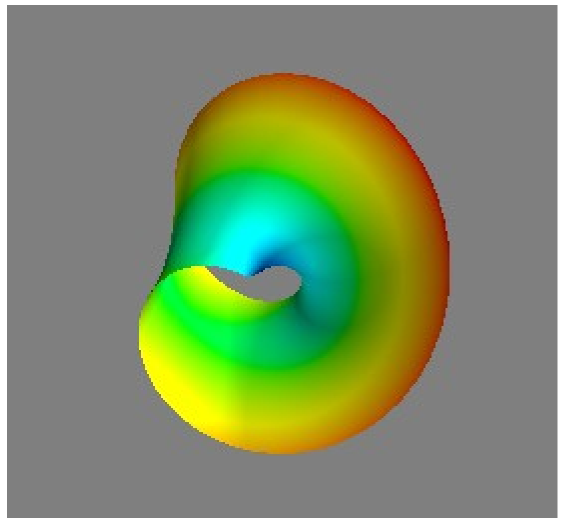


Рисунок 13.8 — Визуализация набора данных *mobius* на сцене

Что касается самого модуля поверхности (*Surface*), то у него тоже есть ряд параметров, которые можно использовать для изменения внешнего вида фигуры на сцене. Например, верните обратно набор данных *boy*, выделите *Surface* в окне "*Engine Tree View*", перейдите на вкладку "*Actor*" и поменяйте параметр "*Representation*" на значение *wireframe.* В результате изменится тип поверхности на представление в виде каркаса.

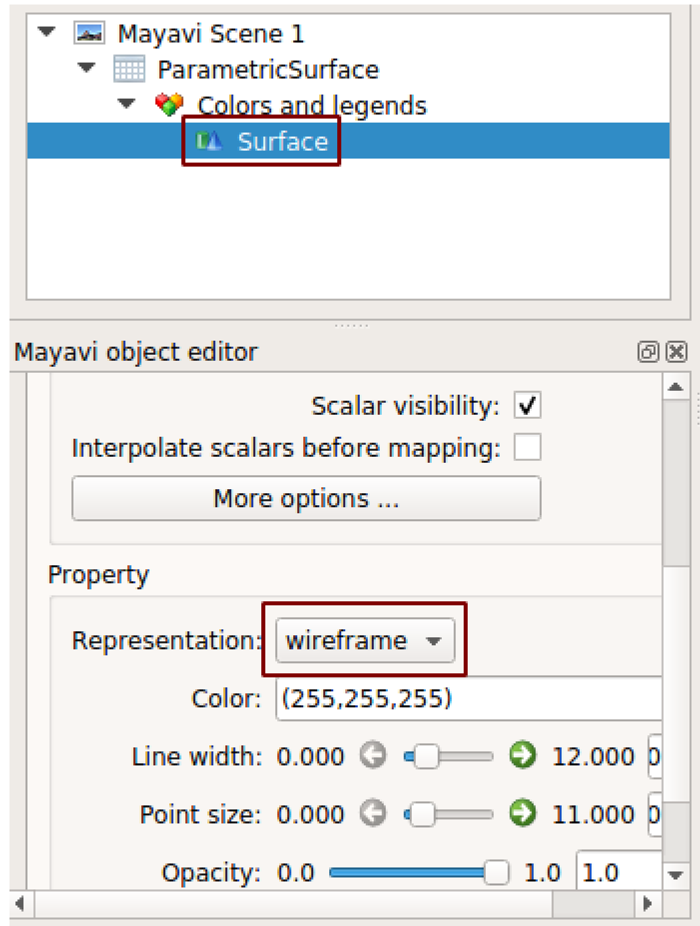


Рисунок 13.9 — Изменение типа представления поверхности на wireframe

В результате получите изображение, представленное на рисунке 13.10.

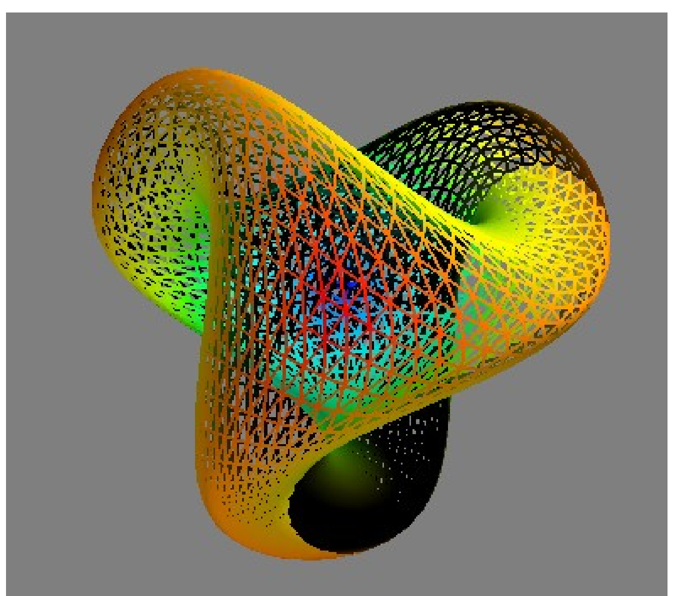


Рисунок 13.10 — Поверхность с параметром *Representation=wireframe*

Таким образом, работа с *Mayavi2* довольно проста. В области "*Engine Tree View*" создаются и настраиваются компоненты в следующем порядке: ***Источник данных -> Фильтры -> Модель***. В области "*Object Editor*" изменяются параметры компонентов, на сцене представляется результат работы.

13.2.2 Разработка Python-модулей, использующих Mayavi

Создайте в вашем текстовом редакторе документ с именем *demo.py*, в котором мы напишем код, демонстрирующий работу с библиотекой *Mayavi*.

Для начала импортируем нужные библиотеки:

```
import numpy as np
from mayavi import mlab
```

С `numpy` вы ознакомились в предыдущих частях этой книги. Из библиотеки `mayavi` мы импортируем модуль `mlab`. Он предоставляет инструменты для построения *2D/3D*-моделей в *Matplotlib* стиле. Вы можете использовать `mlab` как в коде своих собственных модулей, так и в интерактивной среде.

Построим набор точек от *-5* до *5* в количестве *100* штук, для этого воспользуемся функцией `linspace` из `numpy`:

```
t = np.linspace(-5, 5, 100)
```

Создадим массивы с координатами точек *x*, *y*, *z* для трёхмерной спирали через параметрические уравнения:

```
x = 3 * np.cos(t)
y = 3 * np.sin(t)
z = t / np.pi
```

Добавим код для построения *3D*-изображения:

```
s = mlab.plot3d(x, y, z, z, tube_radius=0.1)
mlab.show()
```

Приведём весь код, который должен быть в вашем файле *demo.py*:

```
import numpy as np
from mayavi import mlab
t = np.linspace(-5, 5, 100)
x = 3 * np.cos(t)
y = 3 * np.sin(t)
z = t / np.pi
s = mlab.plot3d(x, y, z, z, tube_radius=0.1)
mlab.show()
```

Откройте терминал, перейдите в каталог с модулем *demo.py* и запустите приложение:

```
python demo.py
```

В результате должно открыться окно "*Mayavi Scene*" с моделью, представленной на рисунке 13.11.

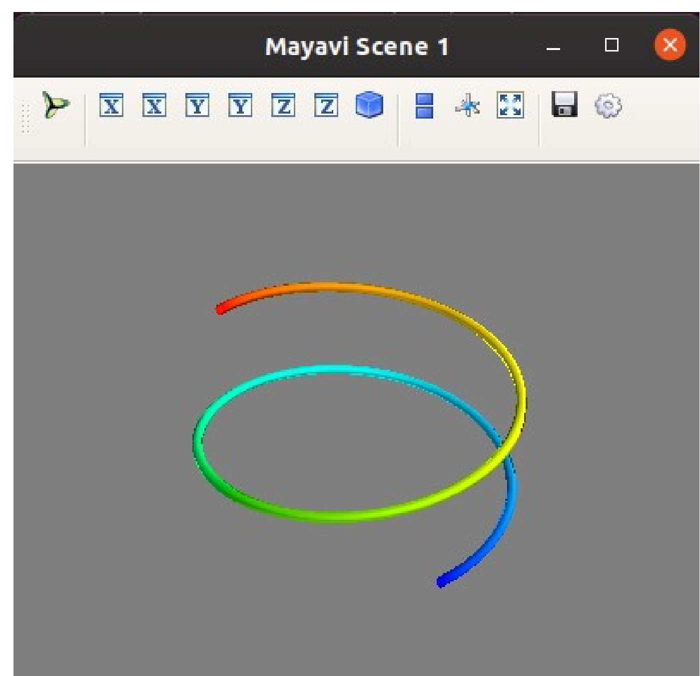


Рисунок 13.11 — Пример построения *3D*-модели в *Python* с использованием *Mayavi*

Mayavi Scene является интерактивной средой, в ней вы можете вращать изображения, приводить к изометрическому виду, сохранять и настраивать его.

13.2.3 Работа с Mayavi в Jupyter notebook

Для отображения результатов работы *Mayavi* в *Jupyter notebook* необходимо указать *backend*, который будет использоваться для представления модели. В зависимости от того, какой вы хотите использовать *backend*, вам может понадобиться установить дополнительные компоненты.

Для работы с *backend'ом 'ipy'* (рекомендуемый вариант) необходимо установить пакеты *ipywidgets* и *ipyevents*:

```
conda install -c conda-forge ipyevents
conda install -c conda-forge ipywidgets
```

Backend 'png' представляет результат рендеринга в виде статичного изображения. С ним не получится поработать в интерактивном режиме. Ещё один вариант *backend'a* это *'x3d'*, который поддерживает *X3D*-элементы, для его использования выполните команду:

```
jupyter nbextension install --py mayavi --user
```

После установки всех необходимых компонентов запустите *Jupyter notebook*. Импортируем `numpy` и `mayavi`, они нам понадобятся для примера:

```
import numpy as np
from mayavi import mlab
```

Следующий важный шаг — это сконфигурировать *Mayavi*, передав информацию о том, что мы работаем с *jupyter notebook*, для этого нужно вызвать функцию `init_notebook()`. При вызове без параметров по

умолчанию будет выбран *backend 'ipy'*, если хотите указать какой-нибудь другой, то передайте его имя в качестве аргумента:

```
mlab.init_notebook()
```

После выполнения этой команды вы должны получить следующее сообщение:

```
…
Notebook initialized with x3d backend.
```

В следующей ячейке введите код:

```
t = np.linspace(-5, 5, 100)
x = 3 * np.cos(t)
y = 3 * np.sin(t)
z = t / np.pi
s = mlab.plot3d(x, y, z, z, tube_radius=0.1)
```

Далее выполните ячейку с переменной s:

```
In [5]: t = np.linspace(-5, 5, 100)
        x = 3 * np.cos(t)
        y = 3 * np.sin(t)
        z = t / np.pi

        s = mlab.plot3d(x, y, z, z, tube_radius=0.1)

In [6]: s

Out[6]:
```

Рисунок 13.12 — Пример *mayavi* в *Jupyter Notebook*

Если вы используете *'ipy'* или *'x3d' backend*, то модель (изображение) будет интерактивной, её можно будет вращать, масштабировать и т.п.

Глава 14. Настройка представления

Самым высокоуровневым элементом *Mayavi* при работе с ней в *Python*-скрипте является фигура (*Figure*). В дальнейшем понятия сцена (*Scene*) и фигура (*Figure*), если не оговариваются отдельно, будут пониматься как синонимы. На сцене происходит отрисовка *2D/3D*-изображения. Помимо самого изображения, можно отдельно настроить окружение, которое включает настройку осей координат, настройку и размещение элемента *colorbar*. Отдельно можно выделить управление камерой на сцене. Рассмотрим эти инструменты более подробно.

14.1 Управление Фигурой/Сценой

Все инструменты, которые предоставляет *Mayavi* для управления сценой, располагаются в пакете `mayavi.mlab`.

Функция `clf()`

Выполняет очистку сцены.

Прототип функции:

```
clf(figure=None)
```

Параметры функции:

- `figure`
  - Сцена, содержимое которой необходимо очистить. Если значение равно `None`, то будет очищена текущая сцена.

Функция `close()`

Закрывает сцену.

Прототип функции:

```
close(scene=None, all=False)
```

Параметры функции:

- `scene`
    - Идентификатор фигуры, которую нужно закрыть, может быть числом, строкой либо объектом *Scene*. Если значение равно `None`, то будет закрыта текущая сцена.
- `all`
    - Если параметр равен `True`, то будут закрыты все существующие сцены.

Функция `draw()`

Выполняет принудительную перерисовку сцены.

Прототип функции:

```
draw(figure=None)
```

Параметры функции:

- `figure`
    - Сцена, которую нужно перерисовать. Если значение равно `None`, то будет перерисована текущая сцена.

Функция `gcf()`

Возвращает указатель на текущую сцену.

Прототип функции:

```
gcf(engine=None)
```

Параметры функции:

- `engine`
    - Движок, из которого будет получена сцена, используется если вы работаете с несколькими движками.

Функция `savefig()`

Сохраняет содержимое сцены в виде изображения. Функция предоставляет возможность сохранить изображение в одном из следующих форматов: *png, jpg, bmp, tiff, ps, eps, pdf, rib (renderman), oogl (geomview), iv (OpenInventor), vrml, obj (wavefront)*.

Прототип функции:

```
savefig(filename, size=None, figure=None, magnification='auto', **kwargs)
```

Параметры функции:

- `filename`
  - Имя файла для сохранения.
- `size`
  - Размер сохраняемого изображения.
- `figure`
  - Сцена, которую нужно сохранить как изображение.
- `magnification`
  - Масштабный коэффициент для перевода изображения с экрана в сохраняемое изображение.

Функция `screenshot()`

Возвращает содержимое сцены (изображение) как массив.

Прототип функции:

```
screenshot(figure=None, mode='rgb', antialiased=False)
```

Параметры функции:

- `figure`
  - Сцена, изображение с которой будет взято.
- `mode`
  - Цветовой режим изображения `{'rgb', 'rgba'}`.

- `Antialiased`
  - Если значение равно `True`, то при рендеринге скриншота будет использоваться антиализинг.

Функция `figure()`

Создаёт новую сцену или возвращает текущую.

Прототип функции:

```
figure(figure=None, bgcolor=None, fgcolor=None, engine=None, size=(400,
350))
```

Параметры функции:

- `figure`
  - Если параметр равен `None`, то будет возвращена текущая сцена, иначе будет создана новая.
- `bgcolor`
  - Цвет заднего фона сцены.
- `fgcolor`
  - Цвет текстовых элементов на сцене.
- `engine`
  - Движок *Mayavi* для управления фигурой.
- `size`
  - Размер сцены. Значение по умолчанию *(400, 350)*.

Рассмотрим на примерах работу с сценой. Для начала импортируем нужные пакеты:

```
import numpy as np
from mayavi import mlab
import matplotlib.pyplot as plt
```

Создадим массив, содержащий координаты уже знакомой нам *3D*-спирали:

```
t = np.linspace(-5, 5, 100)
x = 3 * np.cos(t)
y = 3 * np.sin(t)
z = t / np.pi
```

Настроим для новой сцены светло-зеленый задний фон (параметр `bgcolor`), красный цвет для текстовых надписей (параметр `fgcolor`) и размер *300x300*:

```
mlab.figure("test_fig", bgcolor=(0.56, 0.93, 0.56), fgcolor=(1, 0, 0),
size=(300, 300))
```

Построим *3D*-спираль и отобразим сцену:

```
s = mlab.plot3d(x, y, z, z, tube_radius=0.1)
mlab.title("Test Fig")
mlab.show()
```

В результате получим сцену следующего вида:

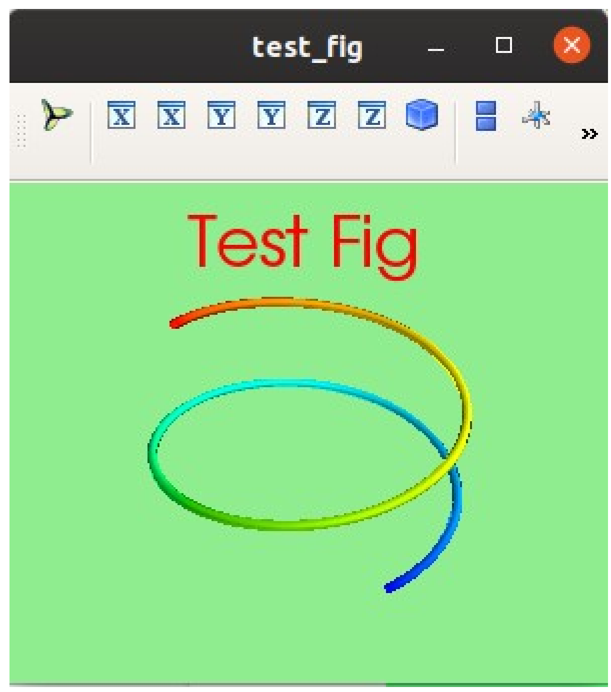


Рисунок 14.1 — Демонстрация возможностей *mayavi* для настройки сцены

Если вы хотите сохранить сцену как изображение, то нужно добавить перед функцией `mlab.show()` либо вместо неё, если не хотите, чтобы сцена отображалась на экране, строчку:

```
mlab.savefig("test.png")
```

Содержимое сцены можно получить как изображение в виде массива для дальнейшей обработки. Удалите или закомментируйте строчку:

```
mlab.show()
```

Добавьте следующий код:

```
f = mlab.gcf()
f.scene._lift()

img = mlab.screenshot()
plt.imshow(img)
plt.show()
```

Первые две строки нужны для того, чтобы функция `screenshot` нормально работала. На момент написания книги в *Mayavi* была ошибка, которая не позволяла без них получать скриншот. Для визуализации изображения используется функция `imshow()` из библиотеки *Matplotlib*. В результате получим окно, представленное на рисунке 14.2.

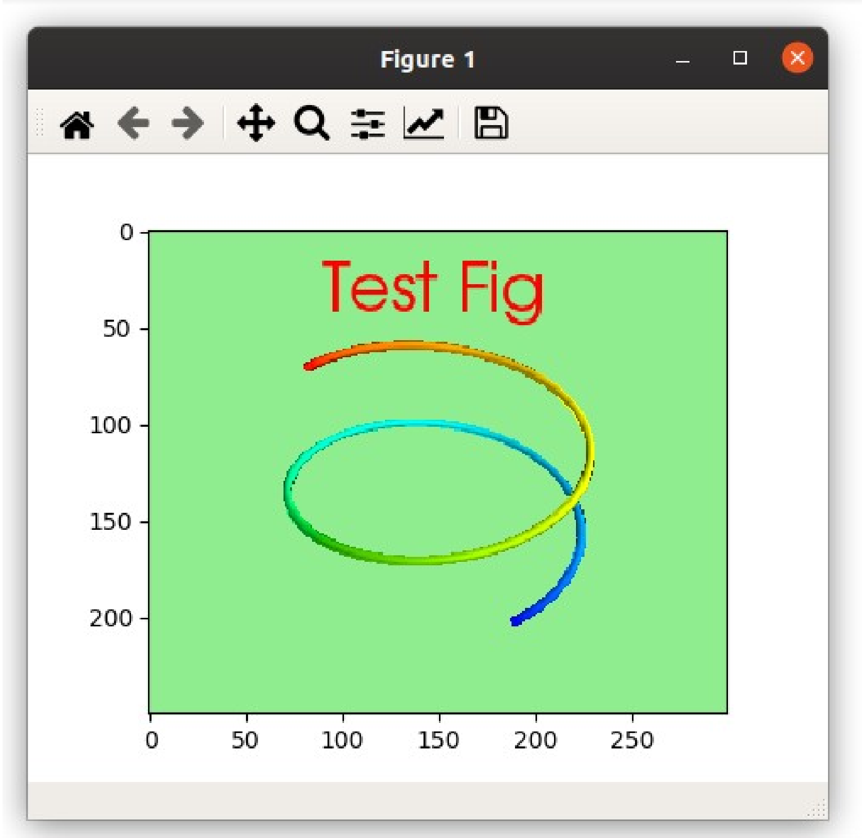


Рисунок 14.2 — Модель в виде изображения

14.2 Настройка элементов оформления

Под элементами оформления будем понимать графические элементы, которые не являются непосредственно самим графиком. К ним относятся заголовок, оси координат, метки, элементы *colorbar* и т.п. Все функции для управления элементами оформления располагаются в пакете `mlab`.

Список функций представлен в таблице 14.1.

Таблица 14.1 — Функции управления элементами оформления

Функция	Описание
`title()`	Заголовок сцены.
`outline()`	Внешний контур модели.
`axes()`	Управляет внешним видом координатных осей.
`xlabel()`	Подпись оси x.
`ylabel()`	Подпись оси y.
`zlabel()`	Подпись оси z.
`colorbar()`	Цветовая полоса.

14.2.1 Заголовок сцены

Начнём наше знакомство с заголовка сцены. С ним мы уже встречались в предыдущем разделе, когда создавали и настраивали фигуру.

Прототип функции для управления заголовком имеет вид:

```
title(*args, **kwargs)
```

Параметры функции:

- `color`
  - Цвет заголовка, задаётся как кортеж из трёх элементов, каждый из которых является числом в диапазоне от 0 до 1.
- `figure`
  - Сцена, на которой будет отображён заголовок. Если параметр равен `None`, то надпись будет выведена на текущий сцене.
- `height`
  - Высота, на которой следует расположить заголовок, задаётся как пропорция от размера фигуры.
- `line_width`
  - Ширина линии символов. Значение по умолчанию: 2.

- `opacity`
  - Прозрачность, значение по умолчанию 1.
- `size`
  - Размер заголовка.
- `name`
  - Имя создаваемого объекта.

Пример работы с заголовком:

```
import numpy as np
from mayavi import mlab

t = np.linspace(-5, 5, 100)
x = 3 * np.cos(t)
y = 3 * np.sin(t)
z = t / np.pi

s = mlab.plot3d(x, y, z, z, tube_radius=0.1)
mlab.title("Test Fig", color=(0,1,0), height=0.1, opacity=0.5, size=1.5)
mlab.show()
```

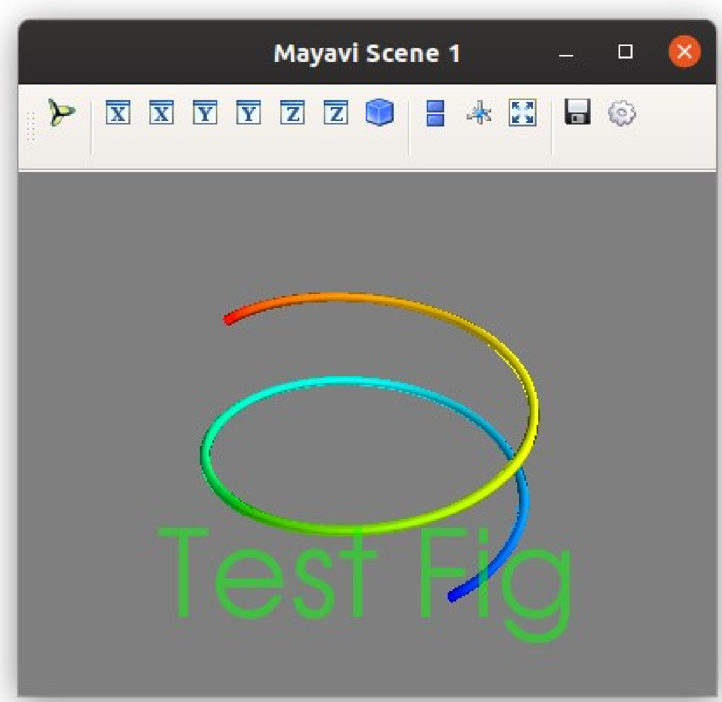


Рисунок 14.3 — Демонстрация работы с функцией `title()`

14.2.2 Внешний контур модели

На сцене можно разместить контур вокруг модели, для этого используется функция `outline()`:

```
outline(*args, **kwargs)
```

Параметры функции:

- `color`
  - Цвет линии, задаётся как кортеж из трёх элементов, каждый из которых является числом в диапазоне от 0 до 1.
- `extent`
  - Геометрия контура `[xmin, xmax, ymin, ymax, zmin, zmax]`, по умолчанию будут взяты размеры модели.
- `figure`
  - Сцена, на которой будет отображён внешний контур. Если параметр равен `None`, то надпись будет выведена на текущей сцене.
- `line_width`
  - Ширина линии. Значение по умолчанию: 2.
- `name`
  - Имя создаваемого объекта.
- `opacity`
  - Прозрачность, значение по умолчанию 1.

Рассмотрим работу с `outline()` на примере. Код для импорта нужных библиотек и создания массивов с координатами: *t, x, y, z* приводить не будем, можете его взять из раздела "*14.2.1 Заголовок сцены*". Ниже представлена программа для отрисовки рамки вокруг *3D* спирали:

```
s = mlab.plot3d(x, y, z, z, tube_radius=0.1)
mlab.outline(color=(0.9, 0.9, 0.9), line_width=3, opacity=0.5)
mlab.show()
```

В результате будет построена сцена, представленная на рисунке 14.4.

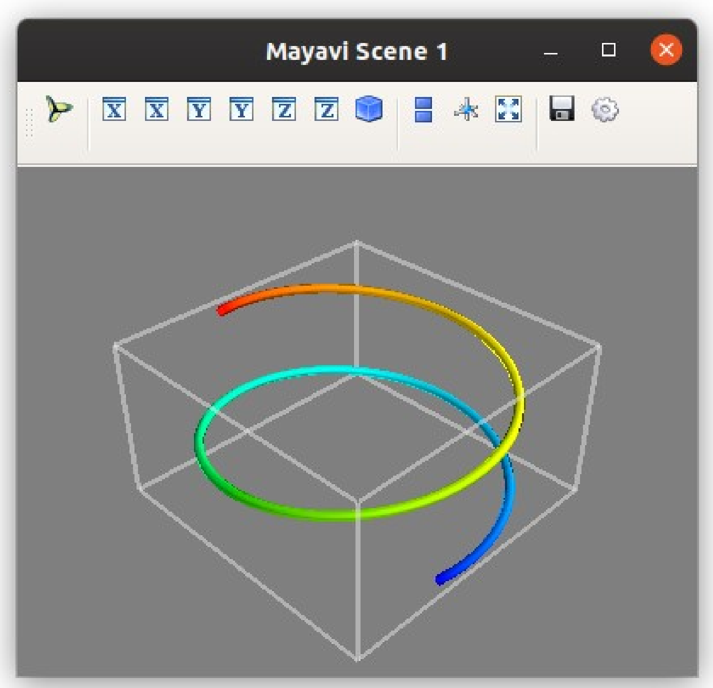


Рисунок 14.4 — Демонстрация работы с функцией `outline()`

14.2.3 Настройка осей координат

За управление внешним видом координатных осей отвечает функция `axes()`, подписи осей можно задавать с помощью `xlable()`, `ylabel()`, `zlabel()`. Рассмотрим их более подробно.

Функция axes()

Управляет внешним видом координатных осей.

Прототип функции:

```
axes(*args, **kwargs)
```

Параметры функции:

- `color`
  - Цвет, задаётся как кортеж из трёх элементов, каждый из которых число в диапазоне от 0 до 1.
- `extent`
  - Геометрия `[xmin, xmax, ymin, ymax, zmin, zmax]`, по умолчанию будут взяты размеры модели.
- `figure`
  - Сцена, на которой будут отображены оси координат. Если параметр равен `None`, то надпись будет выведена на текущий сцене.
- `line_width`
  - Ширина линии. Значение по умолчанию: 2.
- `name`
  - Имя создаваемого объекта.
- `nb_labels`
  - Количество меток вдоль каждого направления осей.
- `opacity`
  - Прозрачность, значение по умолчанию 1.
- `ranges`
  - Диапазоны меток, отображаемые на осях: `[xmin, xmax, ymin, ymax, zmin, zmax]`, по умолчанию берутся размеры модели.
- `x_axis_visibility`
  - Отображать или нет ось *x.*
- `xlabel`
  - Подпись оси *x*.
- `y_axis_visibility`
  - Отображать или нет ось *y.*

- `ylabel`
  - Подпись оси *y*.
- `z_axis_visibility`
  - Отображать или нет ось *z*.
- `zlabel`
  - Подпись оси *z*.

Пример работы с функцией `axes()`, данные для построения модели следует взять из примера, представленного в разделе "*14.2.1 Заголовок сцены*":

```
s = mlab.plot3d(x, y, z, z, tube_radius=0.1)
mlab.axes(color=(0,1,0), nb_labels=5, ranges=[0, 10, 0, 10, 0, 10],
z_axis_visibility=False, ylabel="y_ax", zlabel="z_ax"
)
mlab.show()
```

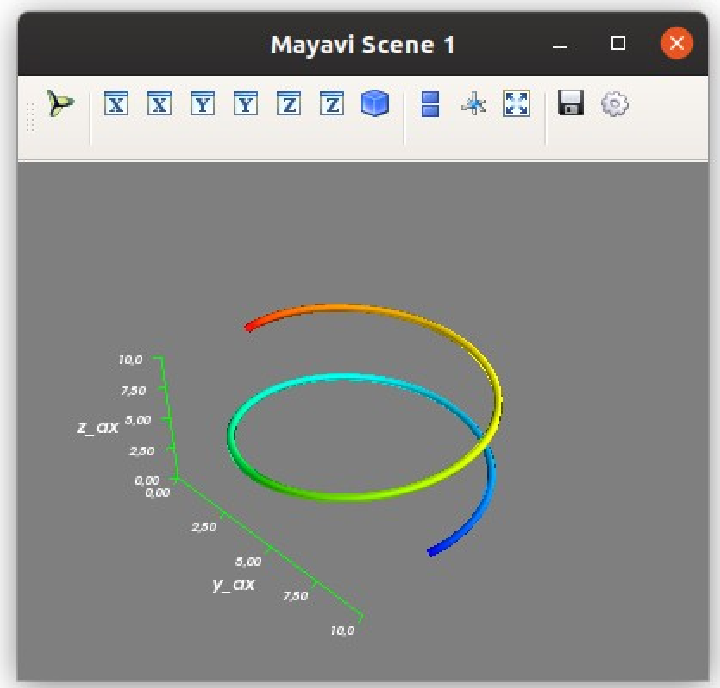


Рисунок 14.5 — Пример работы с функцией `axes()`

Для задания подписей для осей координат можно воспользоваться функциями `xlabel()`, `ylable()`, `zlable()`. Перечисленные функции имеют одинаковый набор аргументов, в качестве примера приведём прототип `xlabel()`:

```
xlabel(text, object=None)
```

Параметры функции:

- `text`
  - Подпись оси.
- `object`
  - Объект, к которому будет применими данная настройка.

Пример работы с функциями `xlabel()`, `ylable()`, `zlable()`:

```
s = mlab.plot3d(x, y, z, z, tube_radius=0.1)
mlab.xlabel("x_ax")
mlab.ylabel("y_ax")
mlab.zlabel("z_ax")
mlab.show()
```

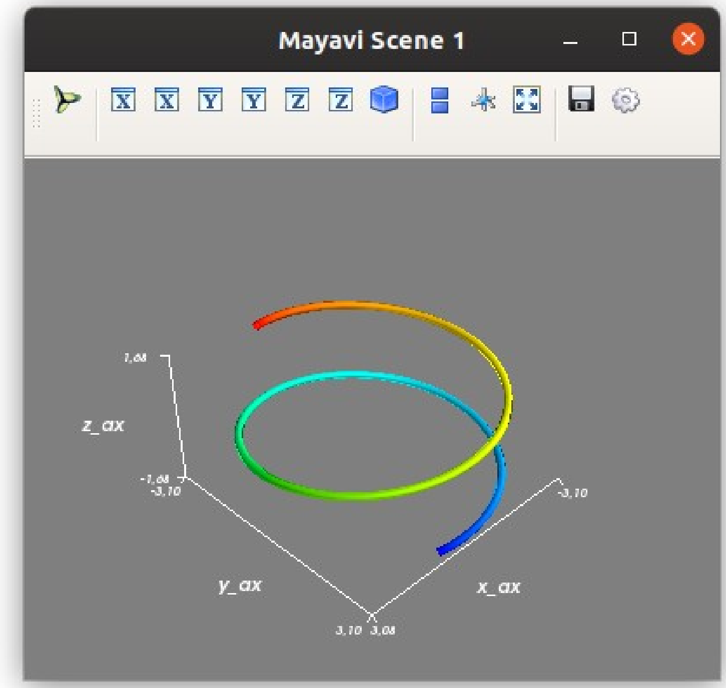


Рисунок 14.6 — Пример работы с функциями `xlabel()`, `ylable()`, `zlable()`

14.2.4 Настройка цветовой полосы (*colorbar*)

Для размещения на сцене цветовой полосы воспользуйтесь функцией `colorbar()`, если вы хотите явно указать, что цветовая полоса нужна для скалярных данных, то используйте `scalarbar()`, для векторных `vectorbar()`.

Прототип функции:

```
colorbar(object=None, title=None, orientation=None, nb_labels=None, nb_colors=None, label_fmt=None)
```

Параметры функции:

- `object`
  - Объект, для которого будет строиться `colorbar`.
- `title`
  - Заголовок элемента `colorbar`.
- `orientation`
  - Ориентация: `'horizontal'` – горизонтальная, `'vertical'` – вертикальная.
- `nb_lables`
  - Количество меток, которое будет отображено на цветовой полосе.
- `label_fmt`
  - Шаблон текстового оформления метки.
- `nb_colors`
  - Максимальное количество цветов, которое будет отображено на цветовой полосе.

Пример использования colorbar():

```
s = mlab.plot3d(x, y, z, z, tube_radius=0.1)
mlab.colorbar(s, title="Bar v1", orientation="vertical",
nb_labels=5,label_fmt="%.1f")
mlab.show()
```

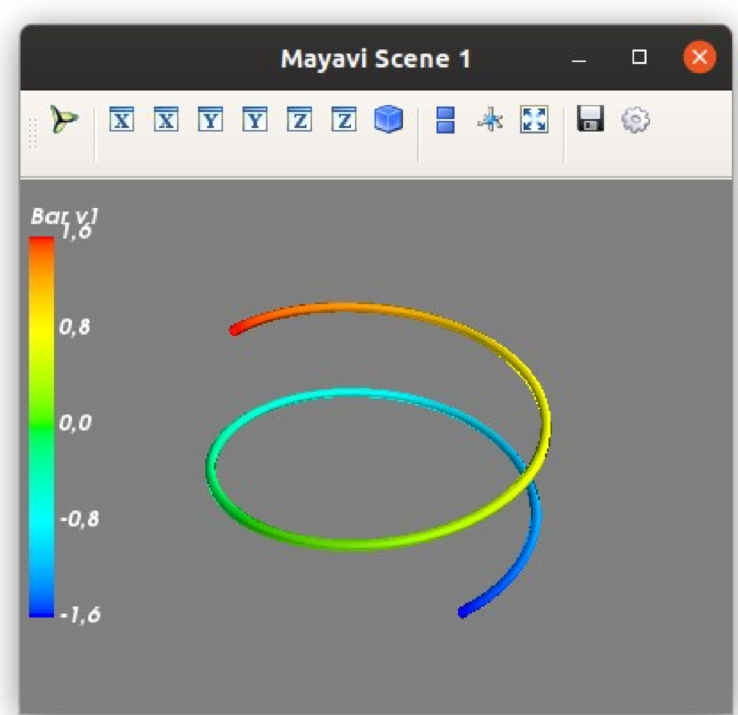


Рисунок 14.7 — Пример работы с функцией colorbar()

Ограничим количество цветов на *colorbar*'е:

```
s = mlab.plot3d(x, y, z, z, tube_radius=0.1)
mlab.colorbar(s, title="Bar v2", orientation="vertical", nb_labels=5,
label_fmt="%.1f", nb_colors=10)
mlab.show()
```

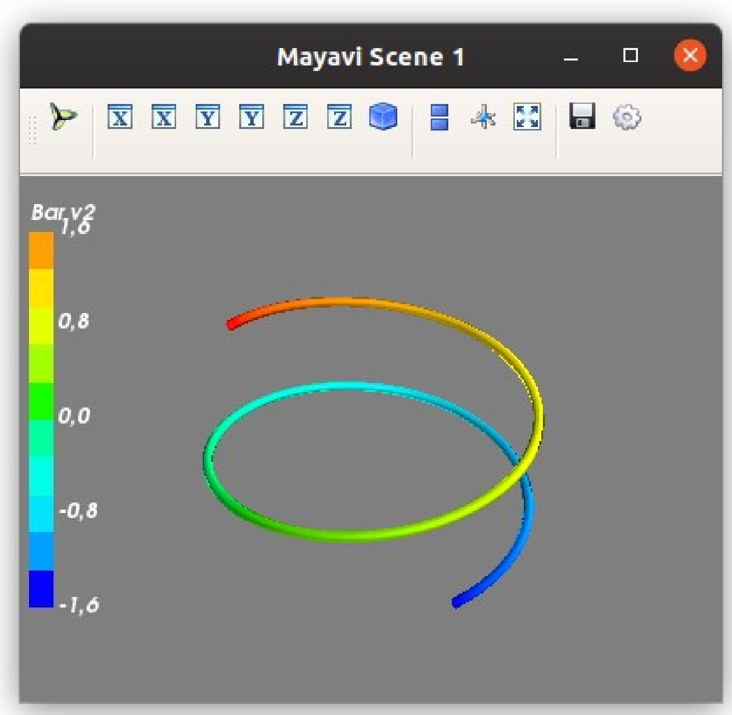


Рисунок 14.8 — Ограничение количества цветов на элементе `colorbar`

14.3 Управление камерой

Mayavi предоставляет ряд инструментов для управления положением камеры, с которой происходит обзор *3D*-сцены, список функций представлен в таблице 14.2.

Таблица 14.2 — Функции управления камерой

Функция	Описание
`move`	Перемещает камеру и фокус.
`pitch`	Вращает камеру вокруг оси, которая соответствует “правому' направлению.
`roll`	Задаёт / возвращает абсолютный угол крена камеры.
`view`	Задаёт / возвращает точку обзора камеры.

yaw	Вращает камеру вокруг оси, которая соответствует “верхнему” направлению.

Более подробно обо всех эти функциях управления камерой вы можете прочитать в официальной документации[21].

Для примера рассмотрим работу с функцией `view()`. Прототип функции:

```
view(azimuth=None, elevation=None, distance=None, focalpoint=None,
roll=None, reset_roll=True, figure=None)
```

Параметры функции:

- `azimuth`
  - Азимут, задаётся в градусах (от 0 до 360).
- `elevation`
  - Подъём, задаётся в градусах (от 0 до 180).
- `distance`
  - Расстояние, от фокуса до камеры. Если поставить значение `'auto'`, то будет подобрано наилучшее значение.
- `focalpoint`
  - Фокус камеры. Если поставить значение `'auto'`, то будет подобрано наилучшее значение.
- `roll`
  - Задаёт вращение камеры вокруг оси.
- `reset_roll`
  - Если значение равно `True`, то параметр `roll` не учитывается.
- `figure`
  - Сцена, на которой будет производиться управление камерой.

21 https://docs.enthought.com/mayavi/mayavi/auto/mlab_camera.html

Пример работы с функцией `view()`:

```
v = mlab.view()
mlab.view(azimuth=0, elevation=0, distance=20, focalpoint=[1,0,0])
mlab.show()
```

В результате получим следующее изображение:

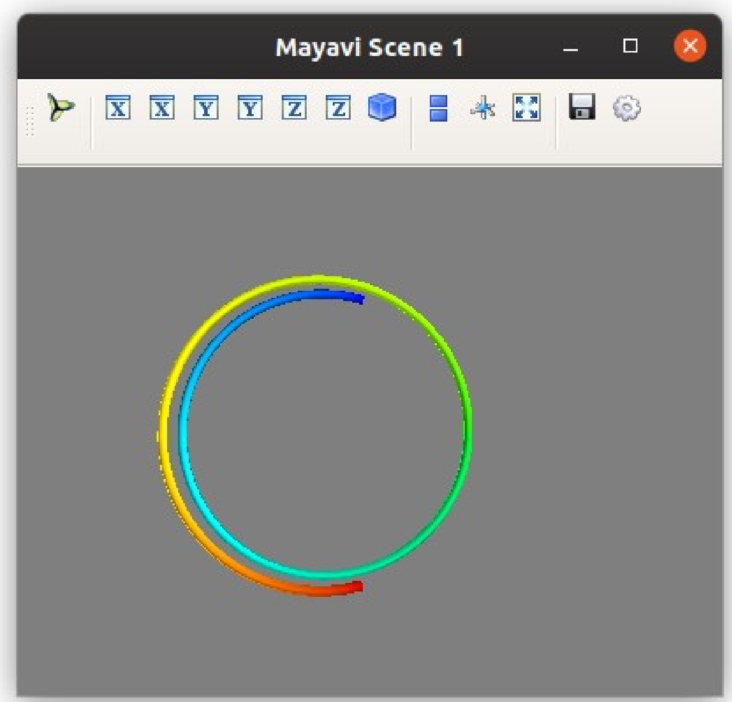


Рисунок 14.9 — Пример работы с функцией `view()`

Глава 15. Визуализация данных

В рамках данного раздела дано описание и примеры работы с функциями для *2D/3D*-визуализации, которые предоставляет *Mayavi*. В следующем разделе: “*Глава 16. Инструменты Mayavi*” будет рассмотрен более общий подход к решению данной задачи с использованием конвейеров обработки данных.

Все функции *Mayavi* для визуализации данных можно разделить по размерности массивов данных, с которыми они работают, а именно, это могут быть одномерные (*1D*) вектора, двумерные (*2D*) таблицы данных и трехмерные (*3D*) наборы. Все функции, представленные в данном разделе, располагаются в пакете `mlab`.

Общие параметры для функций визуализации:

- `color`
  - Цвет. Задаёт единый цвет для всех элементов, в виде кортежа из трёх элементов, каждый из которых — число в диапазоне от 0 до 1.
- `colormap`
  - Цветовая палитра. Если используется аргумент `s` или `f`, то цвет элементов будет выбираться в зависимости от их значений.
- `extent`
  - Размер модели. Задаётся в виде списка `[xmin, xmax, ymin, ymax, zmin, zmax]`. По умолчанию используется размер из массивов `x`, `y`, `z`.
- `figure`
  - Сцена, на которой будет размещена модель.

- `line_width`
  - Ширина линии. Значение по умолчанию: 2.0.
- `name`
  - Имя модели.
- `opacity`
  - Прозрачность.
- `reset_zoom`
  - Сброс масштабирования.
- `vmax`, `vmin`
  - Максимальное и минимальное значения цветовой шкалы.

15.1 Функции для работы с одномерными наборами данных

Для визуализации одномерных наборов данных в *3D*-пространстве доступны функции, представленные в таблице 15.1.

Таблица 15.1 — Функции *Mayavi* для визуализации одномерных наборов данных

Функция	Описание
`points3d`	Визуализация набора данных в виде точек.
`plot3d`	Визуализация набора данных в виде линии.
`quiver3d`	Визуализация набора данных в виде векторного поля (стрелок).

15.1.1 Функция `points3d()`

Функция `points3d()` отображает в виде специальных изображений (глифов) точки, координаты которых передаются через аргументы `x`, `y` и `z`. В дополнение к перечисленным в главе 15 "*Визуализация данных*" для функции `points3d()` доступны следующие параметры:

- `x`, `y`, `z`: *numpy*-массив, `list`
  - Координаты точек.
- `s`: *numpy*-массив, `list`
  - Массив, размер которого должен совпадать с `x`, `y`, `z`. Содержит числа, характеризующие каждую точку. Может использоваться для управления цветом или размером глифов.
- `f`: `callabel`
  - Функция `f(x, y, z)`, которая возвращает характеристическое число для каждой точки исходного набора данных. Может использоваться для управления цветом или размером глифов.
- `mask_points`
  - Определяет шаг извлечения данных из переданных наборов `x`, `y`, `z`. Используется для уменьшения количества отображаемых элементов при работе с большим набором данных.
- `mode`
  - Задаёт стиль глифов: `'2darrow'`, `'2dcircle'`, `'2dcross'`, `'2ddash'`, `'2ddiamond'`, `'2dhooked_arrow'`, `'2dsquare'`, `'2dthick_arrow'`, `'2dthick_cross'`, `'2dtriangle'`, `'2dvertex'`, `'arrow'`, `'axes'`, `'cone'`, `'cube'`, `'cylinder'`, `'point'`, `'sphere'`
- `resolution`
  - Разрешение глифа.

Рассмотрим на примерах работу с `points3d()`. Загрузим нужные библиотеки и создадим набор данных для экспериментов:

```
import numpy as np
from mayavi import mlab

t = np.linspace(0, 2*pi, 15)
x = 3 * np.cos(t)
y = 3 * np.sin(t)
z = 0 * t
s = 2 * np.cos(t)
```

Визуализируем набор данных в виде сфер в *3D*-пространстве:

```
mlab.points3d(x, y, z)
mlab.show()
```

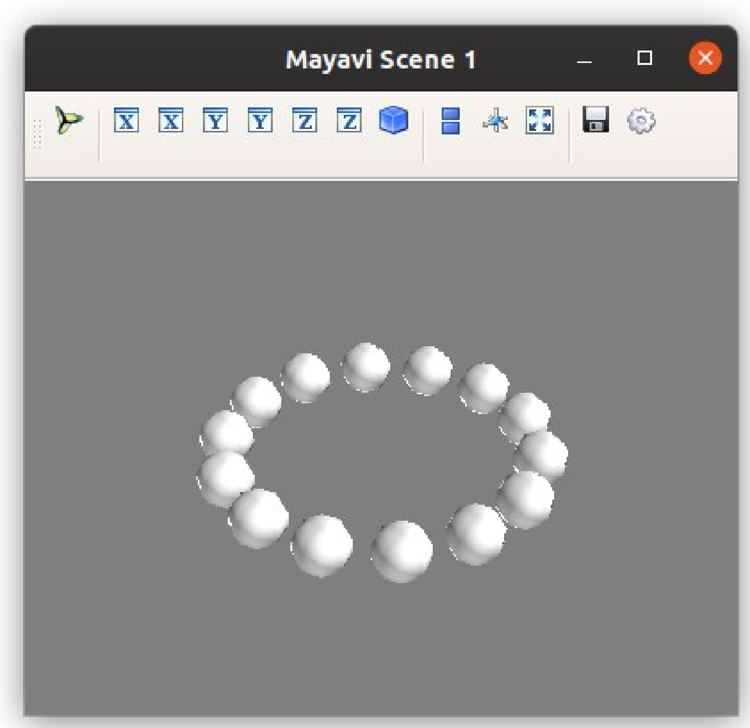


Рисунок 15.1 — Демонстрация работы функции `points3d()`

Если передать массив s в качестве четвёртого параметра, то значения из него будут использоваться для определения цвета и размера глифов:

```
mlab.points3d(x, y, z, s)
```

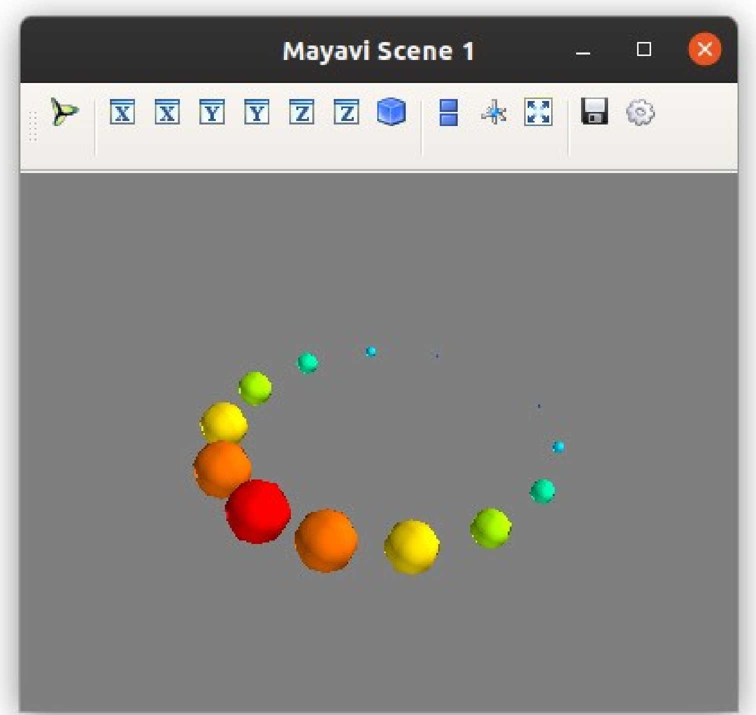


Рисунок 15.2 — Демонстрация работы с параметром s функции points3d()

Вместо s можно использовать f:

```
f_attr = lambda x, y, z: np.array([2*np.random.rand() for _ in
range(len(x))])
mlab.points3d(x, y, z, f_attr)
```

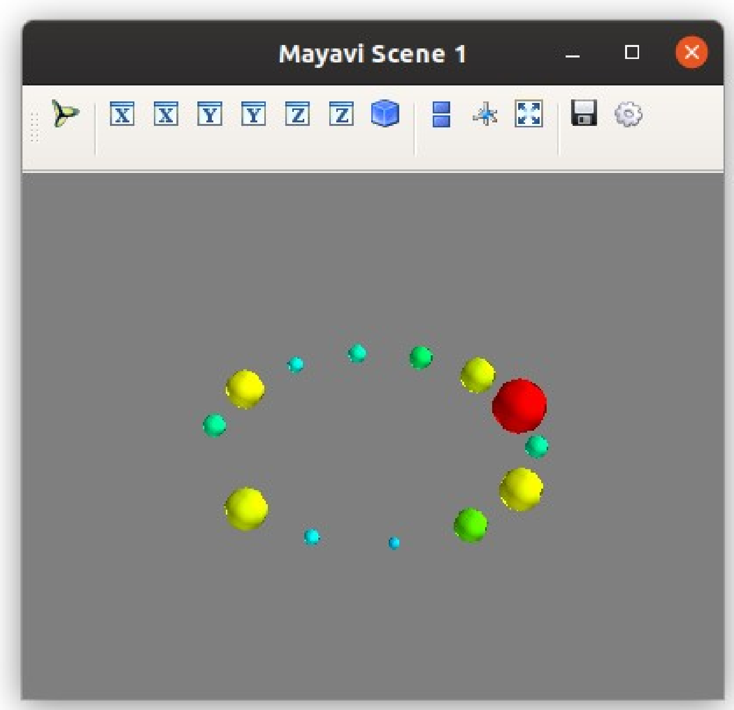


Рисунок 15.3 — Демонстрация работы с параметром `f` функции `points3d()`

Зададим всем глифам одинаковый цвет, также изменим значение параметра `resolution`, вначале присвоим ему значение 4, потом 16, код приведён для варианта 4:

```
mlab.points3d(x, y, z, color=(0,1,0), resolution=4)
```

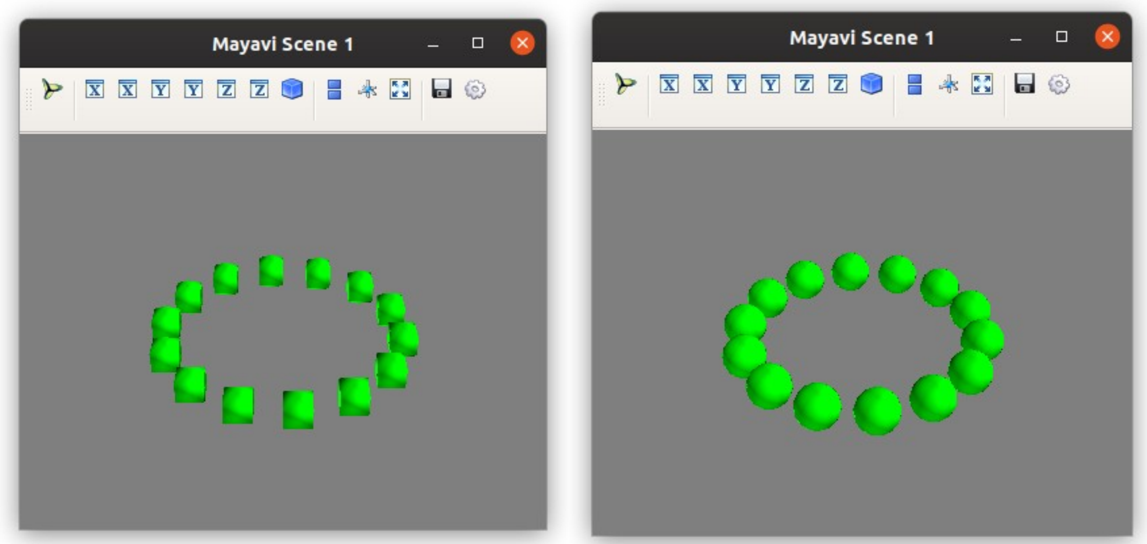


Рисунок 15.4 — Демонстрация работы с параметром `resolution` функции `points3d()`

Обратите внимание на качество глифов во втором варианте (при большом значении параметра `resolution`). Изменим используемую цветовую палитру, внешний вид глифов и прозрачность:

```
mlab.points3d(x, y, z, s, colormap="plasma", mode='cylinder',
opacity=0.7)
```

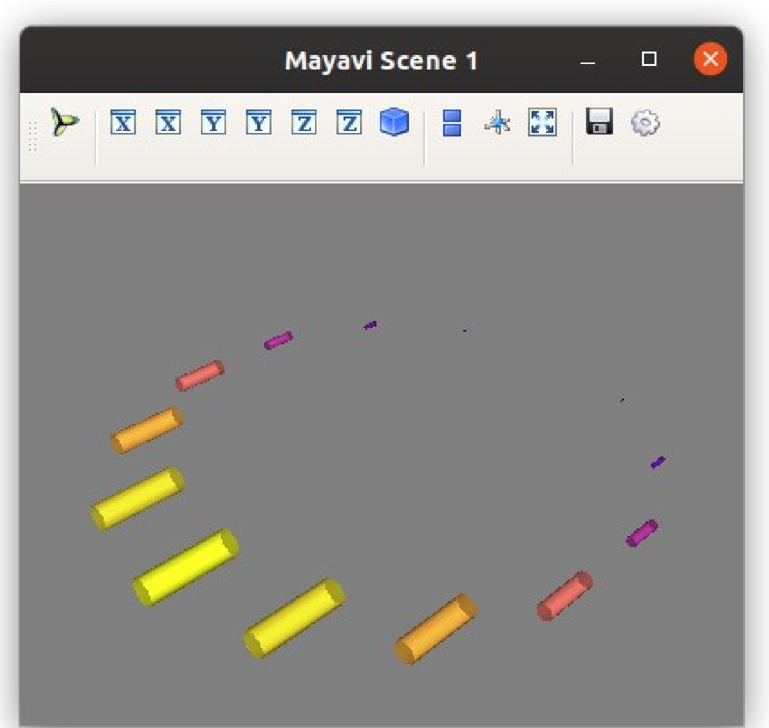


Рисунок 15.5 — Демонстрация работы с параметрами `colormap`, `mod`, `opacity` функции `points3d()`

15.1.2 Функция `plot3d()`

Функция `plot3d()` строит линии по переданным одномерным массивам данных в *3D*-пространстве. Помимо перечисленных в начале главы общих параметров, `plot3d()` имеет ряд уникальных аргументов:

- x, y, z: *numpy*-массив, `list`
  - Координаты точек.
- `representation`
  - Тип поверхности модели: `'surface'`, `'wireframe'`, `'points'`.

- `tube_radius`
    - Радиус трубы, которая используется для представления линии.
- `tube_slides`
    - Количество компонентов трубы, которое используется для представления линии. Значение по умолчанию 6.

Набор данных для работы:

```
t = np.linspace(0, 5*np.pi, 100)
x = t * np.cos(t)
y = t * np.sin(t)
z = t
```

Построим простую модель с помощью `plot3d()`:

```
s = mlab.plot3d(x, y, z)
mlab.show()
```

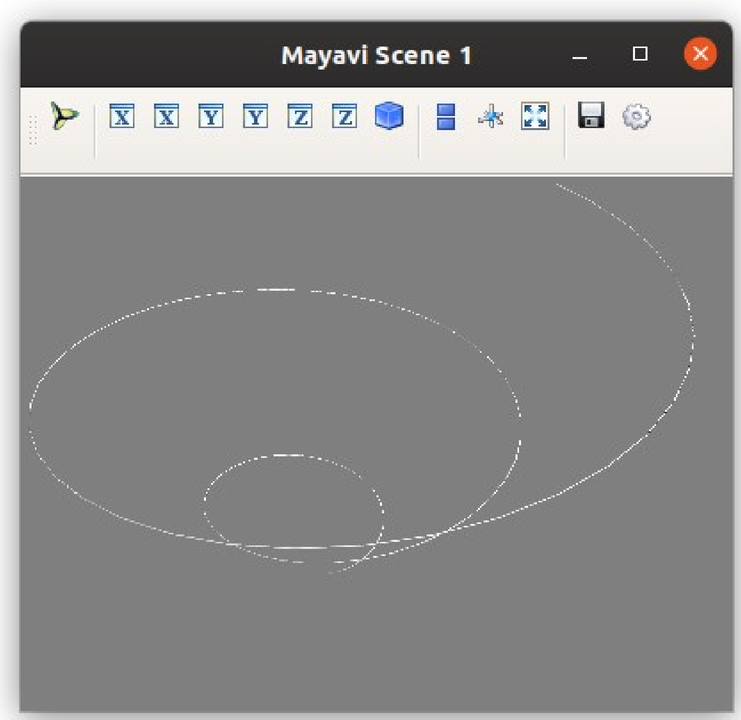


Рисунок 15.6 — Демонстрация работы функции `plot3d()`

Построим модель в виде трубы с радиусом 0.5, дополнительно укажем параметр s для задания цвета разным участкам трубы, в качестве цветовой карты выберем rainbow:

```
s = mlab.plot3d(x, y, z, z, colormap="rainbow", tube_radius=0.5,
tube_sides=3)
```

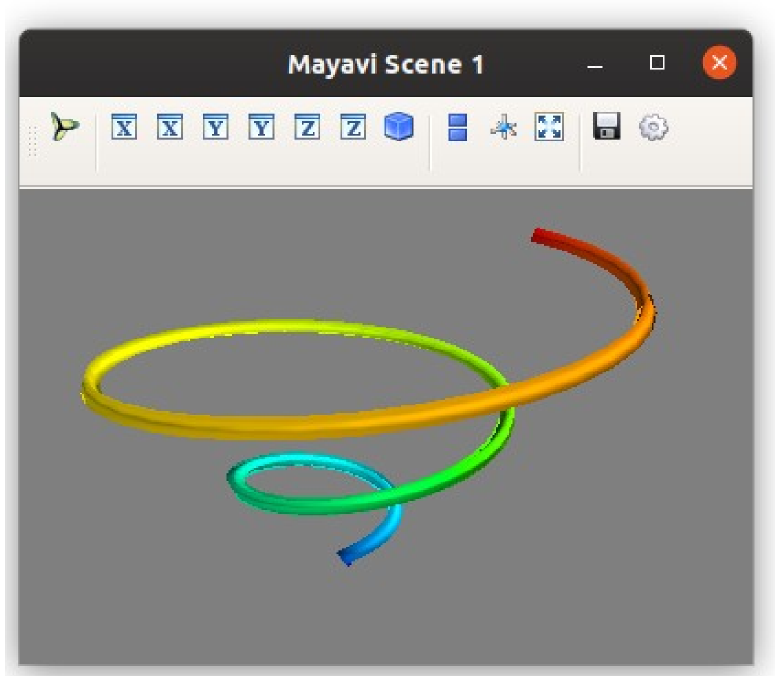


Рисунок 15.7 — Демонстрация работы с параметрами colormap, tube_radius, tube_sides функции plot3d()

Параметр tube_sides, как уже было указано выше, отвечает за количество граней трубы, в данном примере мы построили треугольную трубу. Это хорошо видно при ракурсе, представленном на рисунке 15.8.

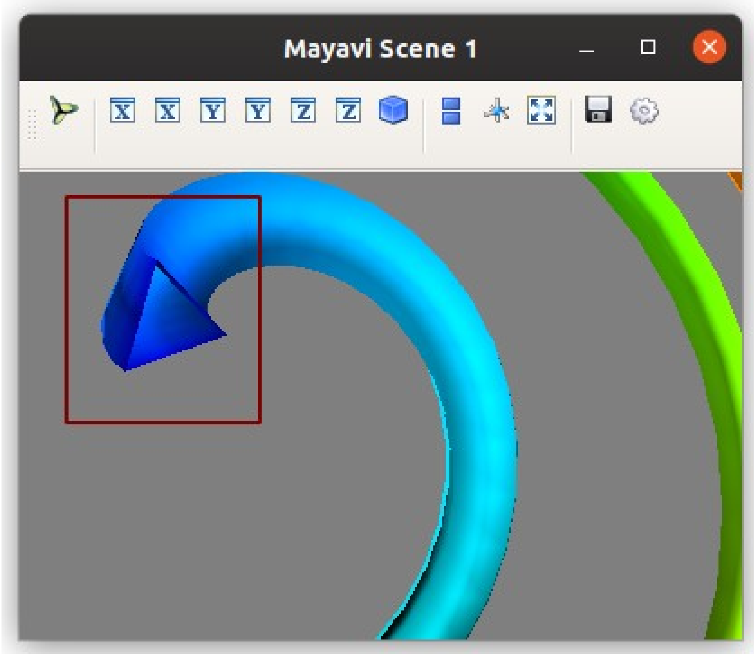


Рисунок 15.8 — Структура трубы

Присвоим параметру `tube_sides` значение `6`, теперь труба имеет форму шестиугольника:

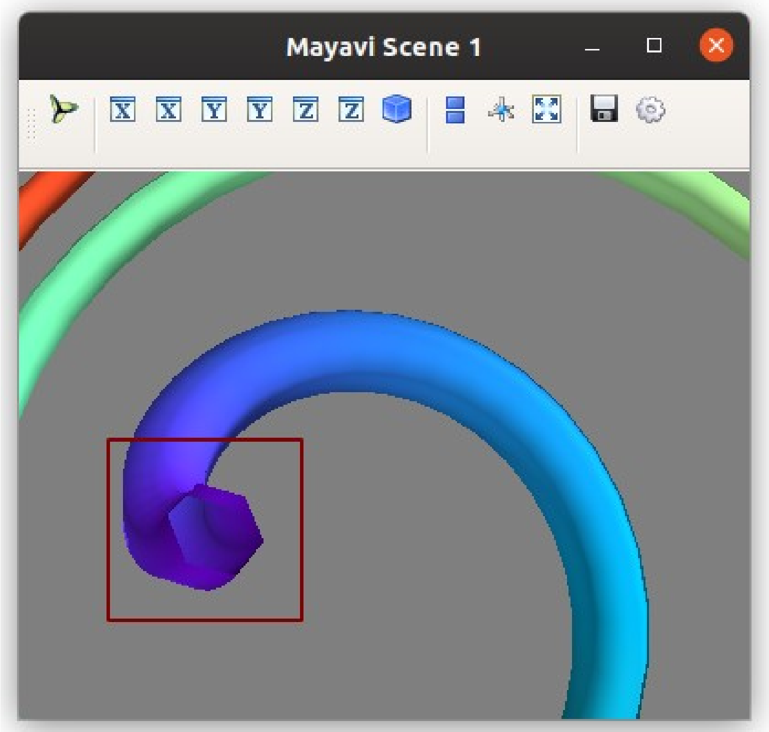


Рисунок 15.9 — Демонстрация работы с параметром `tube_sides=6` функции `plot3d()`

Увеличим размер трубы и поменяем стиль поверхности на `wireframe`:

```
s = mlab.plot3d(x, y, z, z, tube_radius=1.5, tube_sides=10,
representation='wireframe' )
```

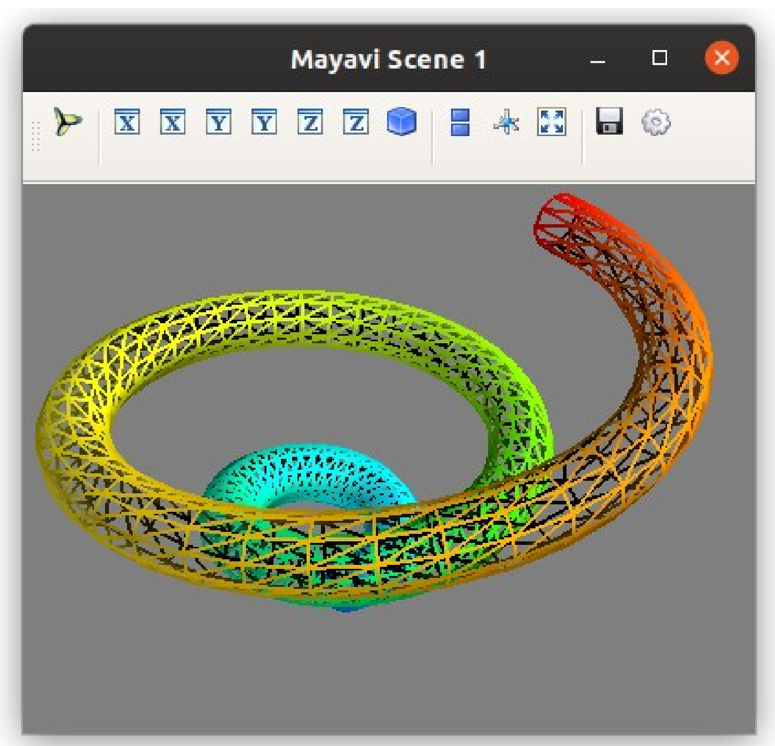


Рисунок 15.10 — Демонстрация работы с параметрами `representation='wireframe'` функции `plot3d()`

15.2 Функции для работы с двумерными наборами данных

Для визуализации двумерных наборов данных в *3D*-пространстве доступны функции, представленные в таблице 15.2.

Таблица 15.2 — Функции *Mayavi* для визуализации двумерных наборов данных

Функция	Описание
`imshow`	Визуализирует набор данных в виде изображения.
`surf`	Строит поверхность по переданному *2D* массиву высот.

`contour_surf`	Строит контурные линии для каждого уровня высоты из переданного *2D* массива данных.
`mesh`	Строит поверхность, по переданным *2D* массивам данных.

15.2.1 Функция `imshow()`

Функция `imshow()` представляет переданный ей двумерный массив в виде изображения.

В дополнение к перечисленным в главе 15 “*Визуализация данных*” `imshow()` имеет следующий набор аргументов:

- `interpolate`
  - Если значение равно `True`, то включается режим интерполяции.

Построим набор данных для визуализации:

```
x, y = np.mgrid[-2:2:0.1, -2:2:0.1]
z = np.cos(x*y) * np.sin(x*y)
```

Отобразим матрицу `z` в виде изображения:

```
mlab.imshow(z)
mlab.show()
```

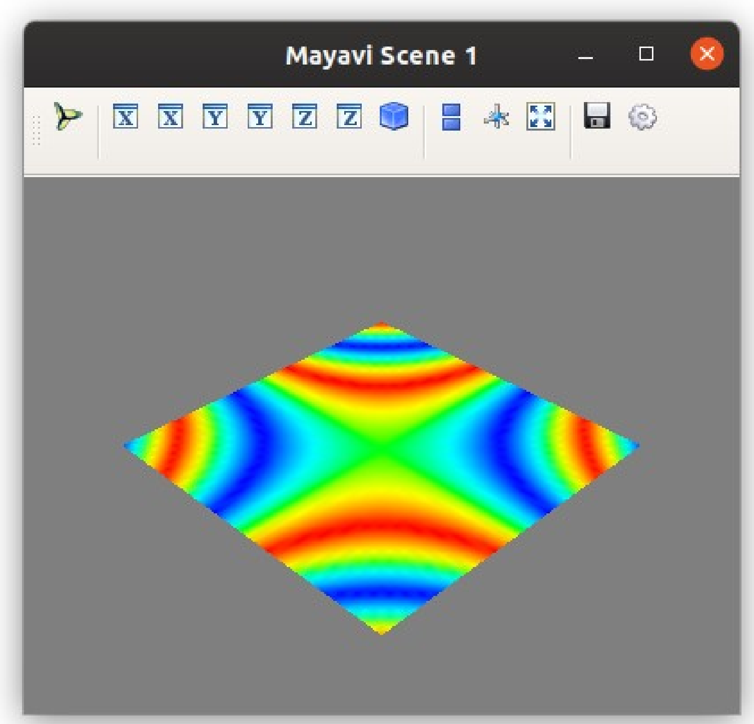


Рисунок 15.11 — Демонстрация работы функции `imshow()`

Выключим режим интерполяции и поменяем цветовую палитру:

```
mlab.imshow(z, interpolate=False, colormap='winter')
mlab.show()
```

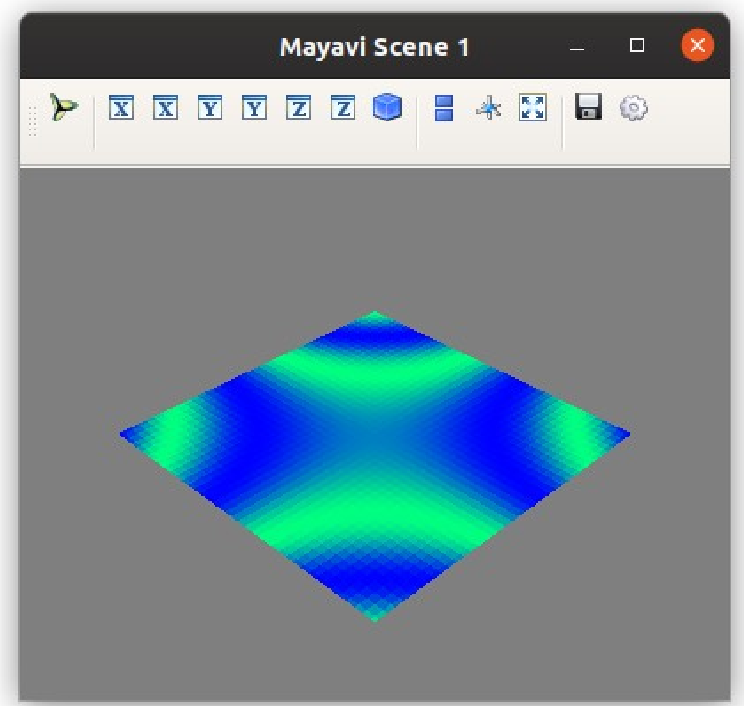


Рисунок 15.12 — Демонстрация работы с параметрами `interpolate` и `colormap` функции `imshow()`

15.2.2 Функция `surf()`

Функция строит поверхность по переданному *2D*-массиву высот. Наиболее часто используемые варианты вызова `surf()` имеют следующий вид:

```
surf(s, ...)
surf(x, y, s, ...)
surf(x, y, f, ...)
```

Аргументы `x`, `y`, `s`, `f` являются позиционными (то есть их нельзя передавать с указанием имени параметра). Параметр `s` – это *2D*-массив данных, каждый элемент которого является значением высоты поверхности; `x`, `y` – могут иметь размерность *1D* или *2D*, они используются для индексации массива `s`; вместо непосредственно *2D*-массива с данными можно передать функцию `f`, которая для наборов координат `x` и `y` будет выдавать значения высоты уровня поверхности. Остальные аргументы функции `surf()` являются именованными. В дополнение к указанным в начале главы (см. “*Глава 15. Визуализация данных*”) функция предоставляет следующие параметры:

- `mask`
  - Массив `bool` значений для подавления элементов из исходного набора данных.
- `representation`
  - Тип поверхности, может быть: `'surface'`, `'wireframe'` или `'points'`.
- `warp_scale`
  - Масштабирование по оси `z`. Если значение равно `'auto'`, то будет подобрано значение масштаба так, чтобы обеспечить наилучшее визуальное представление.

Будем работать со следующим набором данных:

```
x, y = mgrid[-2:2:0.1, -2:2:0.1]
z = 5*cos(x*y) * sin(x*y)
```

Построим поверхность:

```
mlab.surf(z)
mlab.show()
```

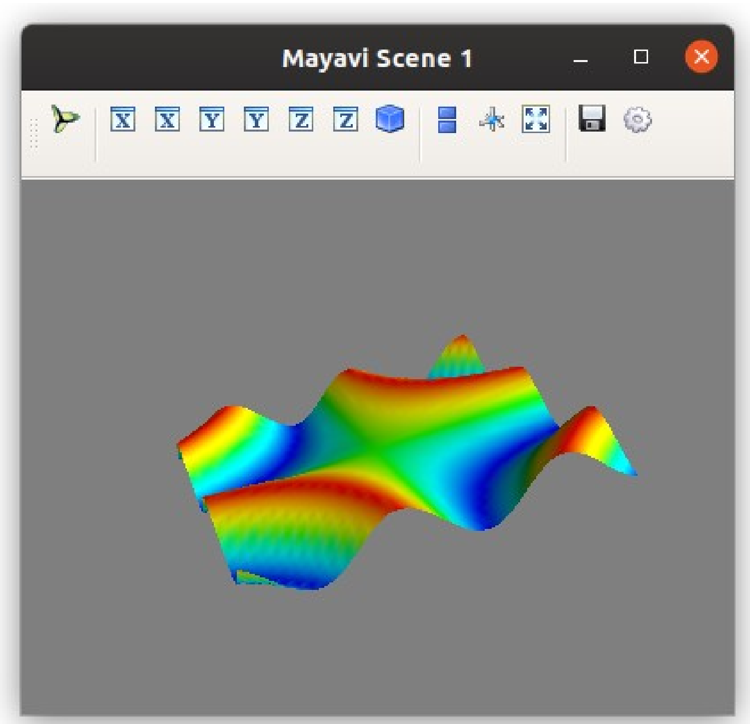


Рисунок 15.13 — Демонстрация работы функции `surf()`

Передадим в `surf()` помимо z массивы `x` и `y`:

```
mlab.surf(x, y, z)
mlab.show()
```

Получим следующее изображение, представленное на рисунке 15.14.

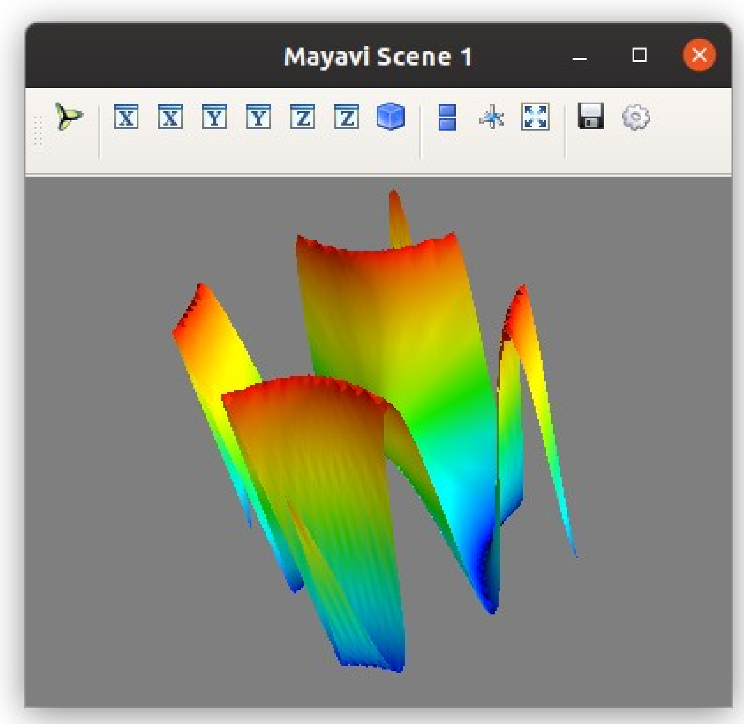


Рисунок 15.14 — Демонстрация работы с параметрами x, y, z функции `surf()`

Так произошло потому, что в случае, когда мы передавали только массив `z`, в качестве значений координат `(x, y)` принимались индексы этого массива, во втором варианте, в дополнение к `z` были переданы предварительно сформированные массивы x и y. Обратите внимание, что шаг между соседними значениями в этих массивах 0.1, а не 1, как было в первом варианте, поэтому высоты и впадины имеют более выраженный вид.

Для поверхности можно задать однотонную расцветку через параметр `color`. Также полезным для настройки визуального представления является параметр `warp_scale`, управляющий масштабом по оси `z`. Сравните высоту поверхности из предыдущего примера, когда мы передавали x и y, с вариантом при `warp_scale='auto'`:

```
mlab.surf(x, y, z, color=(0,0,1), warp_scale='auto')
```

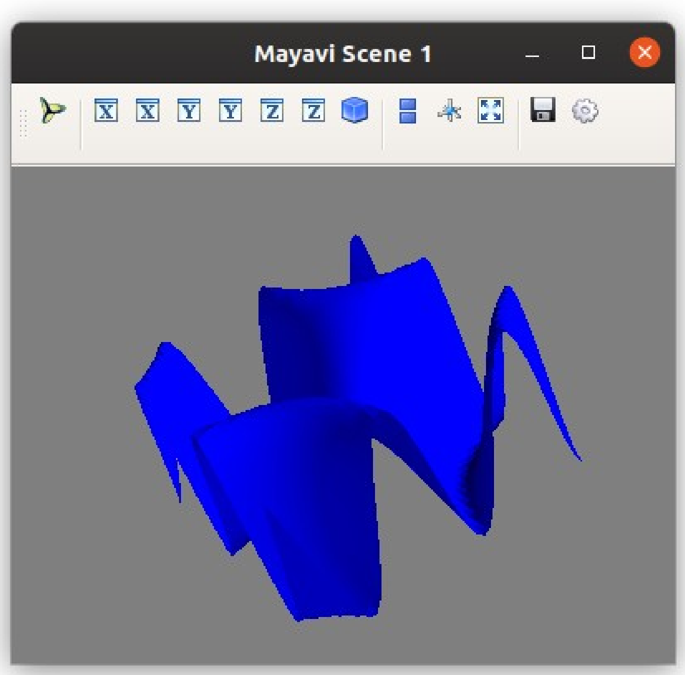


Рисунок 15.15 — Демонстрация работы с параметрами color, warp_scale функции surf()

Изменим палитру на 'spring', зададим тип поверхности 'wireframe', прозрачность установим в 50%:

```
mlab.surf(z, colormap='spring', opacity=0.5, representation='wireframe',
warp_scale='auto')
```

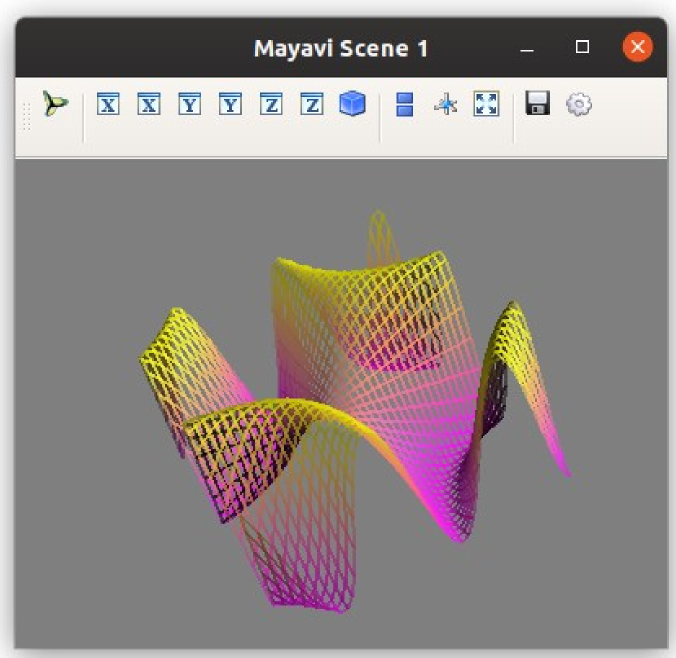


Рисунок 15.16 — Демонстрация работы с параметрами opacity, representation и colormap функции surf()

15.2.3 Функция `contour_surf()`

Функция `contour_surf()` похожа на рассмотренную ранее `surf()`, в отличие от неё она строит не поверхность, а контурные линии для каждого уровня высоты из переданного *2D*-массива данных.

Варианты вызова функции:

```
contour_surf(s, ...)
contour_surf(x, y, s, ...)
contour_surf(x, y, f, ...)
```

Параметры `x`, `y`, `s`, `f` являются позиционными, по назначению совпадают с аналогичными для `surf()`. Функция `contour_surf()` поддерживает параметры из общего набора (см. главу 15 “*Визуализация данных*”) и в дополнение к ним:

- `warp_scale`
  - Масштаб по оси `z`, в отличии от функции `surf()` не поддерживает значение `'auto'`.
- `contours`
  - Список контуров для отображения.

Будем работать с уже знакомым набором данных:

```
x, y = np.mgrid[-2:2:0.1, -2:2:0.1]
z = np.cos(x*y) * np.sin(x*y)
```

Построим контурную поверхность:

```
mlab.contour_surf(x, y, z)
mlab.show()
```

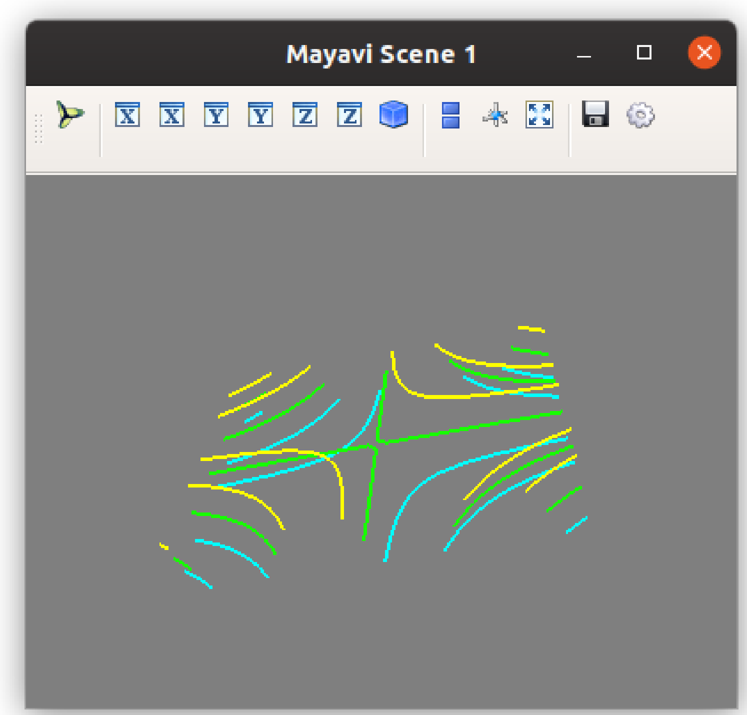


Рисунок 15.17 — Демонстрация работы функции contour_surf()

Увеличим количество отображаемых контуров с помощью параметра contours, поменяем палитру и зададим масштаб для оси z:

```
mlab.contour_surf(z, colormap='autumn', contours=11, warp_scale=10)
```

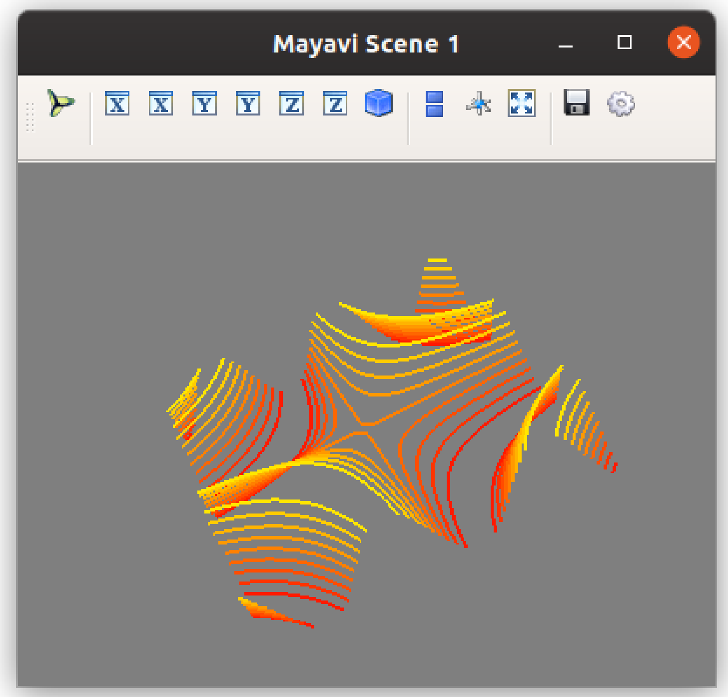


Рисунок 15.18 — Демонстрация работы с параметрами contours, colormap и warp_scale функции contour_surf()

15.2.4 Функция `mesh()`

Функция `mesh()` строит поверхность, по переданным *2D*-массивам данных. Сигнатура функции имеет следующий вид:

`mesh(x, y, z, ...)`

Где x, y, z – это *2D* наборы данных, по которым будет построена поверхность. После них можно добавить параметры, указанные в начале главы, или из списка, представленного ниже (список не полный):

- `representation`
  - Тип поверхности, может быть: `'surface'`, `'wireframe'`, `'points'`, `'mesh'` или `'fancymesh'`. Если выбран тип `'points'`, то можно использовать параметры для работы с глифами.
- `mode`
  - Тип глифа.
- `scale_factor`
  - Масштабный коэффициент.

Создадим набор данных, задающий ¾ поверхности тора:

```
u, v = np.mgrid[-np.pi:np.pi/2:0.05, -np.pi:np.pi+0.1:0.1]
x=np.cos(u)*(np.cos(v)+3)
y=np.sin(u)*(np.cos(v)+3)
z=np.sin(v)
```

Построим поверхность с помощью функции `mesh()`:

```
mlab.mesh(x, y, z)
mlab.show()
```

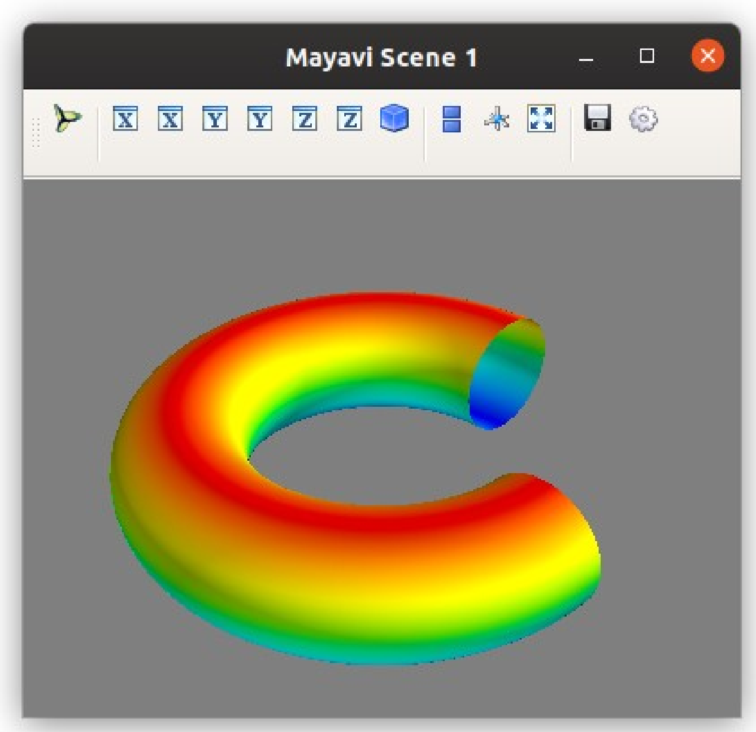


Рисунок 15.19 — Демонстрация работы функции `mesh()`

Изменим цветовую схему и тип поверхности на `'wireframe'`:

```
mlab.mesh(x, y, z, colormap="spring", representation="wireframe")
```

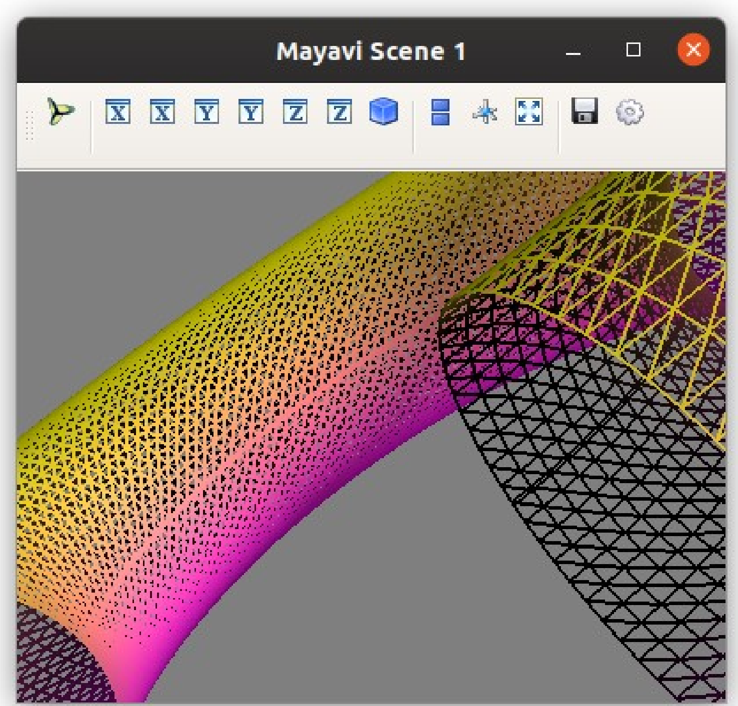


Рисунок 15.20 — Демонстрация работы с параметром `representation='wireframe'` функции `mesh()`

Вариант поверхности 'mesh':

```
mlab.mesh(x, y, z, colormap="summer", representation="mesh")
```

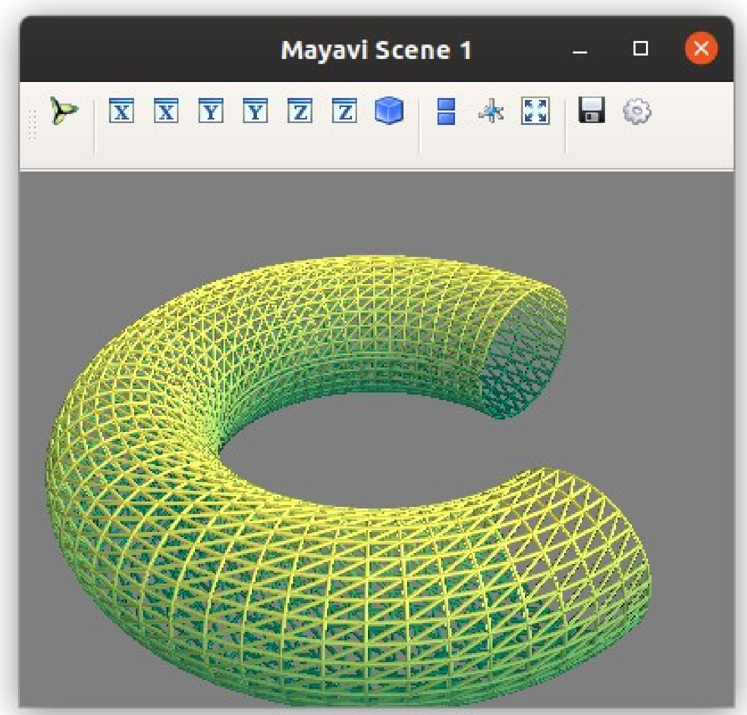


Рисунок 15.21 — Демонстрация работы с параметром representation='mesh' функции mesh()

15.3 Функции для работы с трехмерными наборами данных

Для визуализации трехмерного набора данных в *3D*-пространстве доступны функции, представленные в таблице 15.3.

Таблица 15.3 — Функции *Mayavi* для визуализации двумерных наборов данных

Функция	Описание
contour3d	Строит изоповерхности по переданному *3D*-набору данных
quiver3d	Строит векторное поле по переданным координатам векторов.
volume_slice	Строит срезы для переданных *3D*-наборов данных.

Общие параметры для функций работы с трехмерными наборами данных:

- `color`
  - Цвет. Задаёт единый цвет в виде кортежа из трёх элементов, каждый из которых — число в диапазоне от 0 до 1.
- `colormap`
  - Цветовая палитра. Если используется аргумент `s` или `f`, то цвет элементов будет выбираться в зависимости от их значений.
- `extent`
  - Размер модели. Задаётся в виде списка `[xmin, xmax, ymin, ymax, zmin, zmax]`. По умолчанию используется размер из массивов `x`, `y`, `z`.
- `figure`
  - Сцена, на которой будет размещена модель.
- `line_width`
  - Ширина линии. Значение по умолчанию: 2.0.
- `name`
  - Имя модели.
- `opacity`
  - Прозрачность.
- `reset_zoom`
  - Сброс масштабирования.
- `vmax`, `vmin`
  - Максимальное и минимальное значения цветовой шкалы.

15.3.1 Функция contour3d()

Функция `contour3d()` строит изоповерхности по переданному *3D*-набору данных. Идейно она похожа на `contour_surf()`, которая строит

изолинии (см. раздел "*15.2.3 Функция* contour_surf()"). Возможны два основных варианта вызова:

- без задания координат: contour3d(scalars, …).
- с заданием координат: contour3d(x, y, z, scalars, …).

Параметр scalars – это *3D*-массив данных, по которому будут строиться изоповерхности. Остальные параметры полностью совпадают с contour_surf(), за исключением warp_scale, он у функции contour3d() отсутствует.

Построим массив данных на основе уравнения конуса в пространстве:

```
x, y, z = np.ogrid[-7:7:0.1, -7:7:0.1, -1:7:0.1]
scalars = x*x + y*y - z*z
```

Модель со значениями параметров по умолчанию:

```
mlab.contour3d(scalars)
```

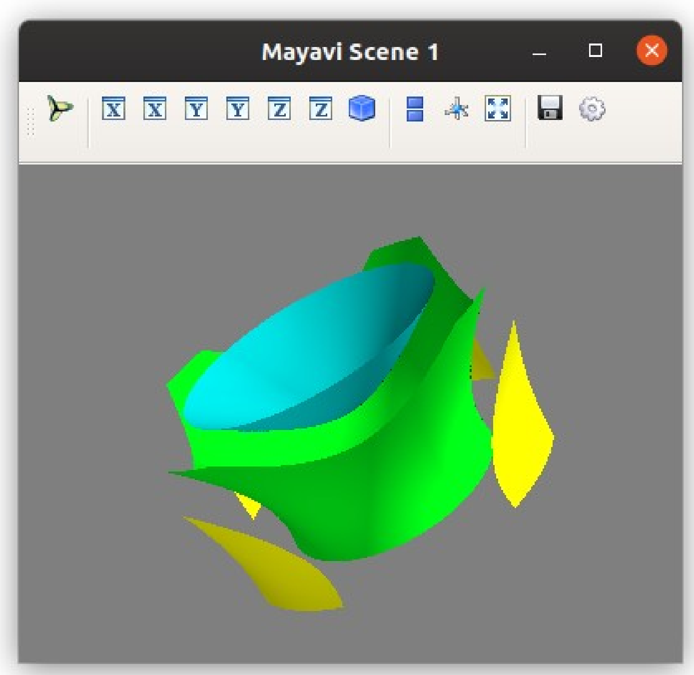


Рисунок 15.22 — Демонстрация работы функции contour3d()

Увеличим количество изоповерхностей:

```
mlab.contour3d(scalars, contours=10)
```

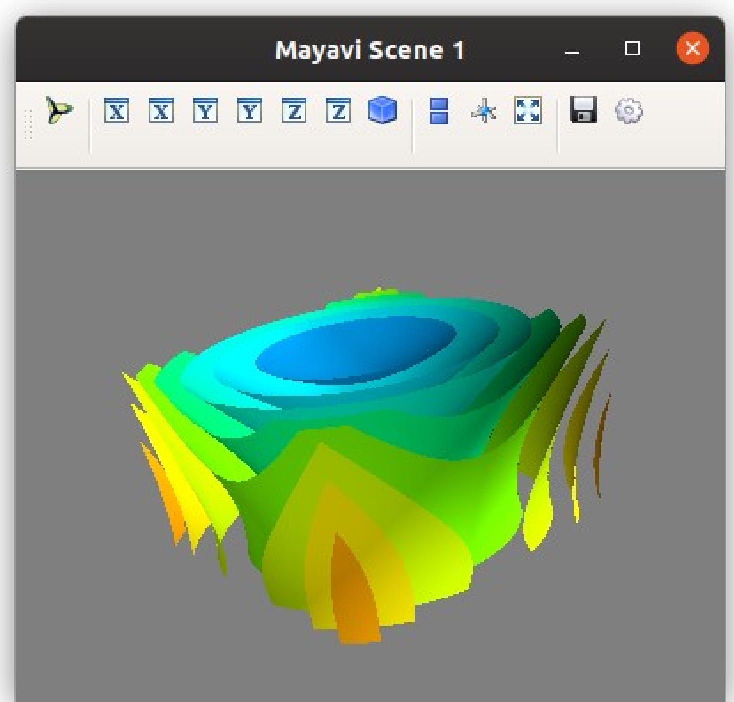


Рисунок 15.23 — Демонстрация работы с параметром `contrours` функции `contour3d()`

Продемонстрируем работу с прозрачностью, возможны два варианта — это задать прозрачность всем элементам модели, или прозрачность будет определяться величиной скаляра из набора данных. За первый вариант отвечает параметр `opacity`:

```
mlab.contour3d(scalars, contours=10, opacity=0.5)
```

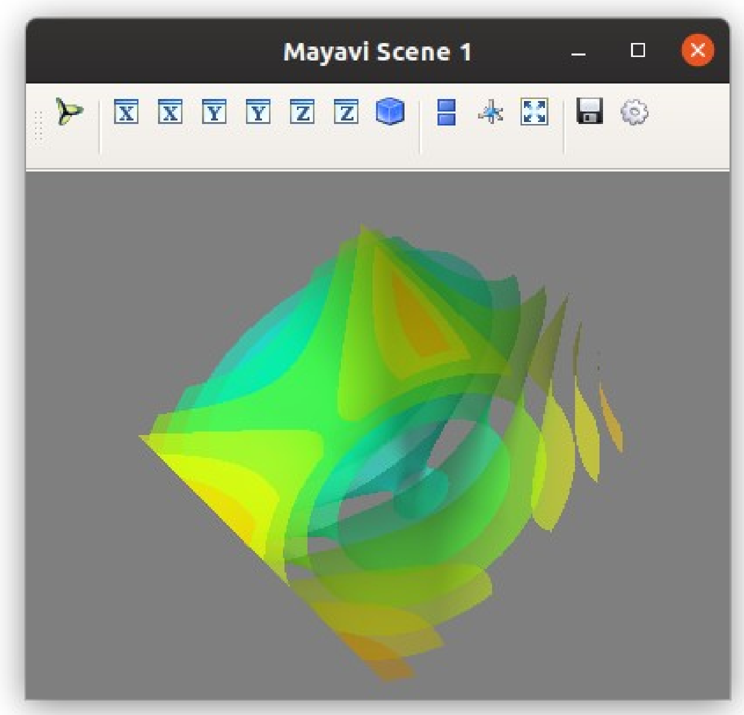


Рисунок 15.24 — Демонстрация работы с параметром `opacity` функции `contour3d()`

Второй вариант активируется через параметр `transparent`:

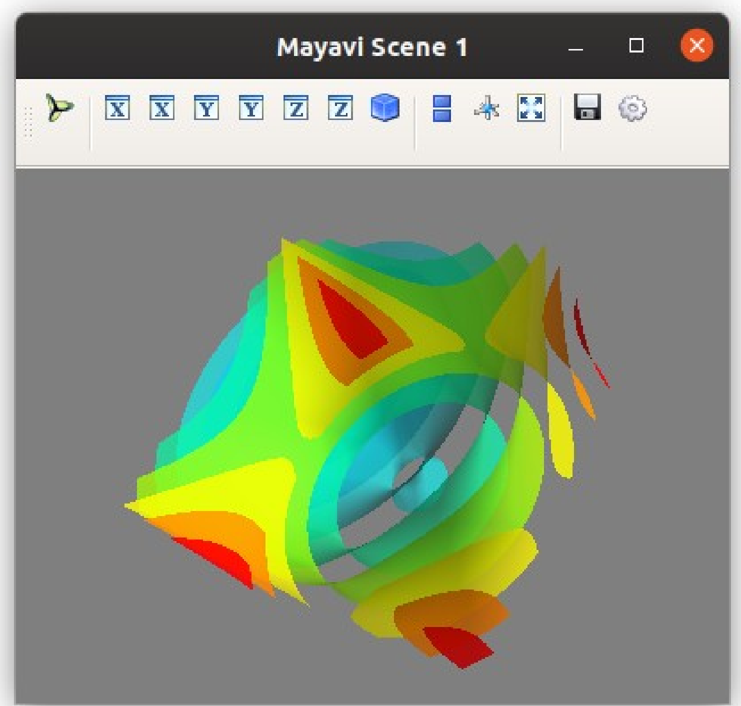


Рисунок 15.25 — Демонстрация работы с параметром `transparent` функции `contour3d()`

Обратите внимание, что области с наименьшими значениями из переданного набора (голубой цвет) имеют большую прозрачность по сравнению с областями с наибольшими значениями (красный цвет).

15.3.2 Функция `quiver3d()`

Функция `quiver3d()` строит векторное поле по переданным координатам векторов. Библиотека предоставляет три основных варианта работы с функцией:

```
quiver3d(u, v, w, ...)
quiver3d(x, y, z, u, v, w, ...)
quiver3d(x, y, z, f, ...)
```

В первом, в `quiver3d()` передаются только компоненты вектора, во втором координаты и компоненты вектора, в третьем координаты и функция, которая их принимает и возвращает компоненты вектора.
Компоненты и координаты — это *3D* наборы данных.

Параметры функции `quiver3d()` совпадают с параметрами `contour3d()` с дополнительным аргументом `scalars`, через который можно передать массив скалярных значений.

Подготовим набор данных и функцию, которая по координатам, возвращает компоненты вектора:

```
x, y, z = np.mgrid[0:3, 0:3, 0:3]
u = x * x
v = y
w = z


def f(x, y, z):
    return x*x, y, z
```

Векторное поле можно построить, вызвав функцию `quiver3d()` с переданными компонентами и координатами:

```
mlab.quiver3d(x, y, z, u, v, w, scale_factor=0.2)
mlab.show()
```

либо с координатами и функцией `f`:

```
mlab.quiver3d(x, y, z, f, scale_factor=0.2)
mlab.show()
```

В обоих случаях получим модель следующего вида:

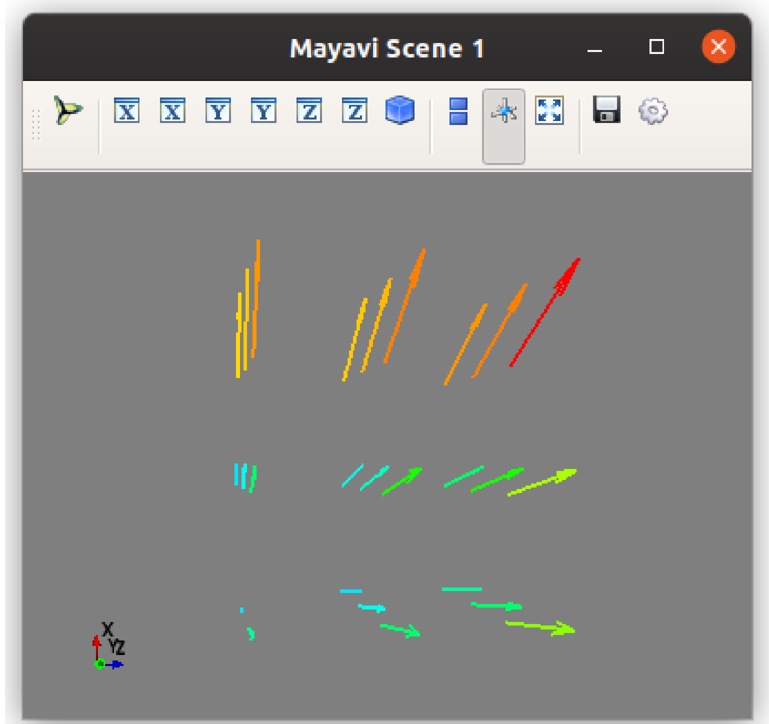


Рисунок 15.26 — Демонстрация работы функции `quiver3d()`

15.3.3 Функция `volume_slice()`

Функция `volume_slice()` строит срезы для переданных *3D*-наборов данных. Возможны два основных варианта вызова:

- с передачей набора скаляров:

  `volume_slice(scalars, ...)`

- с указанием координат и скаляров:

  `volume_slice(x, y, z, scalars, ...)`

Дополнительно к параметрам, перечисленным в начале главы, для `volume_slice()` доступны:

- `plane_opacity`
  - Прозрачность плоскости среза.
- `plane_orientation`
  - Положение плоскости среза, значение по умолчанию `'x_axes'`.
- `slice_index`
  - Индекс, на котором будет располагаться плоскость среза.

Набор данных для демонстрации:

```
x, y, z = ogrid[-7:7:0.1, -7:7:0.1, -1:7:0.1]
scalars = x*x + y*y - z*z
```

Построим срез по оси *x*:

```
mlab.volume_slice(scalars, plane_orientation='x_axes')
mlab.show()
```

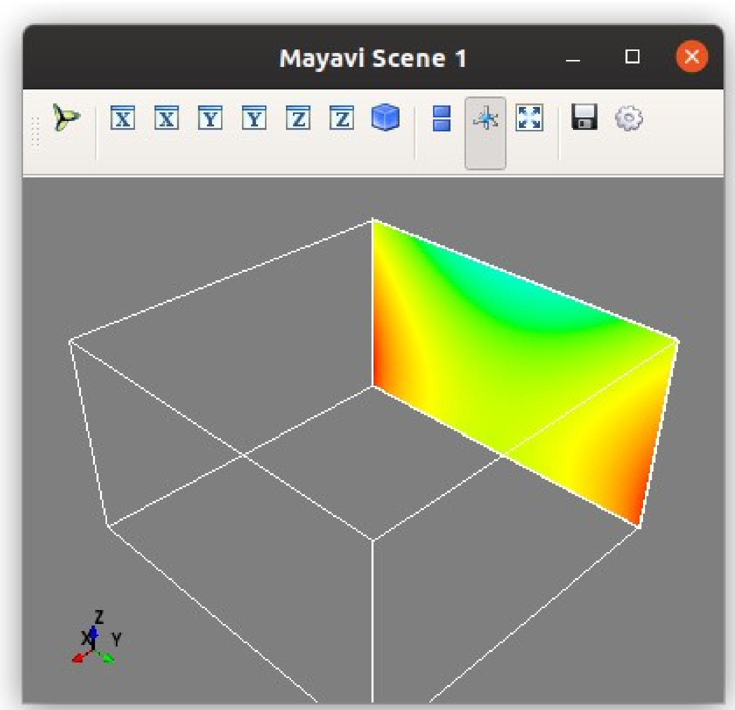


Рисунок 15.27 — Демонстрация работы функции `volume_slice()`

Добавим плоскость среза на ещё одну ось и подвинем её на середину:

```
mlab.volume_slice(scalars, slice_index=round(len(x)/2),
plane_orientation='x_axes')
mlab.volume_slice(scalars, plane_orientation='y_axes')
mlab.outline()
mlab.show()
```

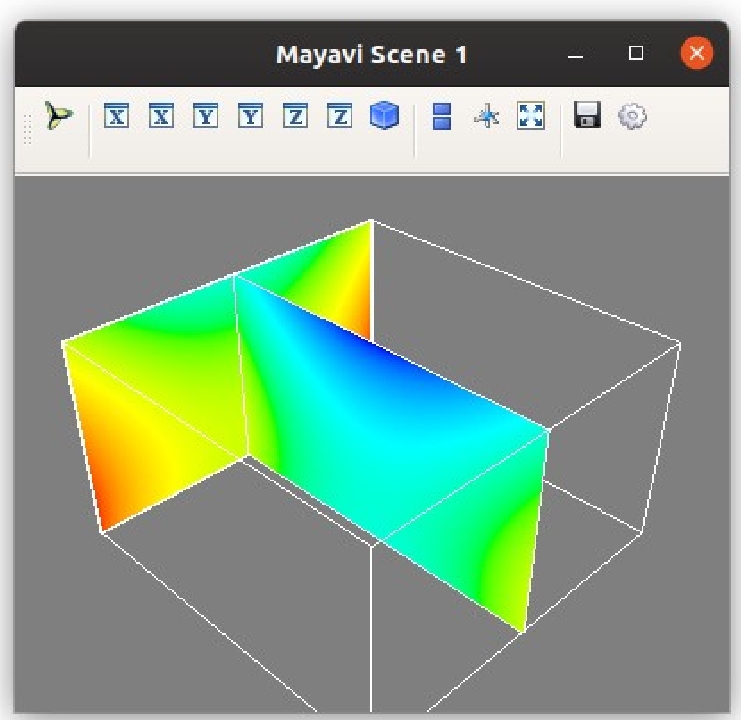


Рисунок 15.28 — Демонстрация работы с двумя плоскостями среза

Глава 16. Работа с *pipeline*

Вариант работы с библиотекой *Mayavi*, которого мы придерживались в рамках главы 15 "*Визуализация данных*", на самом деле является довольно ограниченным и не позволяет использовать все возможности, которые предоставляет этот инструмент. По сути, функции, с которыми мы познакомились, являются заранее заданными сценариями (шаблонами) работы, дающими определённый результат. Например, если мы работаем с функцией `surf()`, то передаём ей в качестве входных данных `numpy`-массивы, а получаем поверхность, которая строится определенным образом по переданным данным и т.п. На более глубоком уровне работа *Mayavi* представляет собой выполнение ряда операций в конвейерном (*pipeline*) режиме. *Pipeline* предполагает последовательность из трёх этапов:

- загрузка данных из определённого источника в объект *Data source*;
- трансформация данных с помощью инструментов из набора *Filters*[22] — это опциональный этап;
- построение модели с помощью модуля из набора *Modules*[23].

Для построения *pipeline*'а библиотека *Mayavi* представляет инструмент `pipeline`, который находится в пакете `mlab`.

Когда вы вызываете любую функцию из рассмотренных в главе 15, *Mayavi* строит соответствующий *pipeline*, который включает в себя объекты, обеспечивающие работу с нужным источником данных и модуль, который строит модель соответствующего вида.

22 https://docs.enthought.com/mayavi/mayavi/filters.html#filters

23 https://docs.enthought.com/mayavi/mayavi/modules.html#modules

Для того чтобы посмотреть на структуру конвейера, нажмите на кнопку *“Mayavi pipeline”* на панели инструментов сцены (см. рисунок 16.1). Для примера посмотрим на *pipeline*, который формируется для функции `imshow()`. Запустите код программы, приведённый ниже:

```
import numpy as np
from mayavi import mlab

x, y = np.mgrid[-2:2:0.1, -2:2:0.1]
z = np.cos(x*y) * np.sin(x*y)

mlab.imshow(z)
```

В результате откроется сцена, представленная на рисунке ниже.

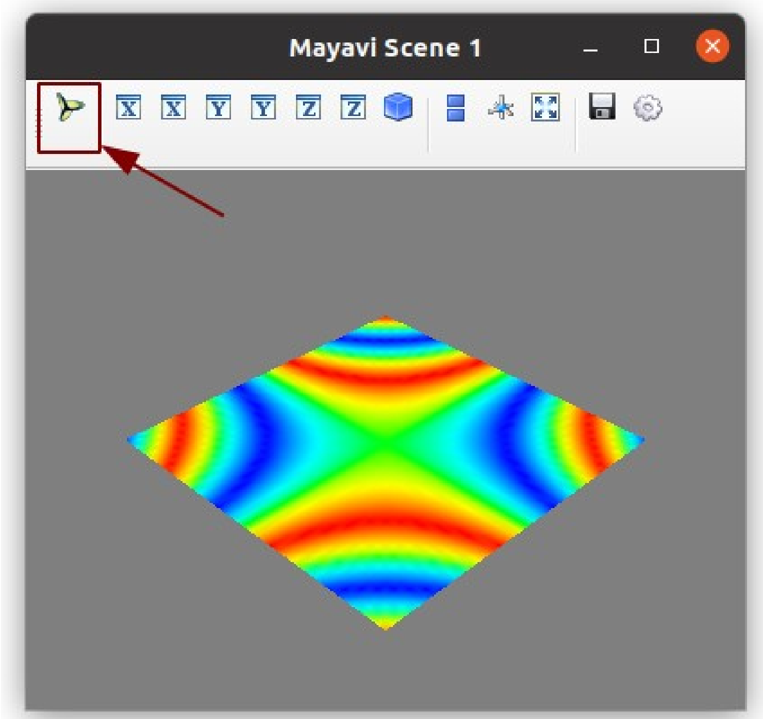


Рисунок 16.1 — Модель, построенная с помощью функции `imshow()`

Нажмите на указанную кнопку на панели инструментов. В окне "*Mayavi pipeline*" конвейер представляется в виде дерева в левой части окна.

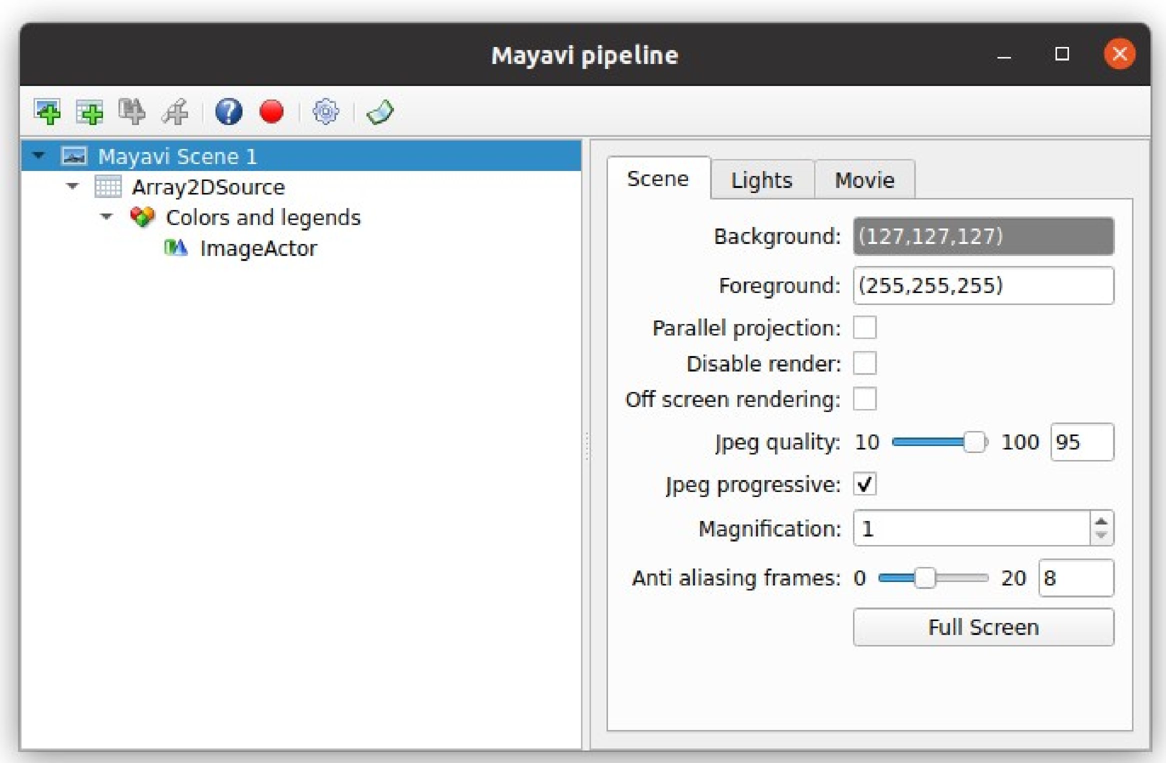


Рисунок 16.2 — Окно *Mayavi pipeline*

В нашем случае конвейер состоит из двух элементов:

- *Array2DSource* — создаёт объект *data source*;
- *ImageActor* — создаёт модель для отображения на сцене.

Построим соответствующие *pipeline*“ы вручную:

```
src = mlab.pipeline.array2d_source(z)
img = mlab.pipeline.image_actor(src)
mlab.show()
```

Результат выполнения этого кода, будет аналогичный варианту с `mlab.imshow()`. Фактически строки:

```
mlab.imshow(z)
```

и

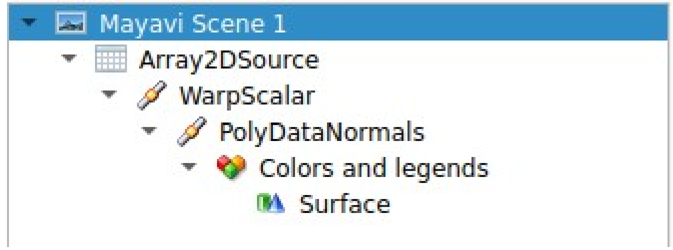
```
src = mlab.pipeline.array2d_source(z)
img = mlab.pipeline.image_actor(src)
```

эквиваленты.

Посмотрим на более сложный *pipeline*, для этого передадим в функцию `surf()` набор данных *z*:

```
mlab.surf(z)
mlab.show()
```

Конвейер будет выглядеть так:



Реализуем его в коде:

```
src = mlab.pipeline.array2d_source(z)
warp = mlab.pipeline.warp_scalar(src)
norm = mlab.pipeline.poly_data_normals(warp)
surf = mlab.pipeline.surface(norm)
mlab.show()
```

16.1 Структура *pipline*

Конвейер *Mayavi* состоит из нескольких уровней. На самом верхнем находится движок (*Engine*). Он не отображается в дереве элементов *pipeline*, движок отвечает за создание и удаление сцены, на которой отображается модель. Далее идёт сцена (*Scene*), она содержит в себе источники данных (*Data source*), фильтры (*Filters*), менеджер модулей (*Module Manager*) и модули визуализации (*Modules*). *Engine* и *Scene* присутствуют в единственном экземпляре, цепочек со следующей структурой: ***источник данных -> фильтры (от нуля до n штук) -> Module Manager -> модуль визуализации*** может быть несколько в рамках одной сцены.

Источники данных предназначены для загрузки данных из *numpy*-массивов, списков и т.п. в специальные объекты. Фильтры позволяют выполнить над данными различные преобразования, менеджер модулей (отображается как *Colors and legends* в *Mayavi pipeline*) управляет цветом и представлением данных. Модули визуализации непосредственно формируют представление на сцене в виде линий, поверхностей, точек и т.п.

Pipeline можно собрать вручную через пользовательский интерфейс либо написать программу, которая вызывает функции создания элементов конвейера и передаёт в них нужные аргументы. Третий вариант — это воспользоваться заранее подготовленной функцией из библиотеки *Mayavi*, которая построит *pipeline* для решения конкретной задачи, например отображение поверхности, линии и т.п. Каждый элемент, входящий в *pipeline*, имеет ссылку на родительский элемент, данные из которого он получает.

Напишем пример для вывода родительских элементов функции `surf()`:

```
obj = mlab.surf(z)
while(True):
    print(type(obj))
    if hasattr(obj, "parent"):
        obj = obj.parent
    else:
        break
```

Запустим эту программу:

```
<class 'mayavi.modules.surface.Surface'>
<class 'mayavi.core.module_manager.ModuleManager'>
<class 'mayavi.filters.poly_data_normals.PolyDataNormals'>
<class 'mayavi.filters.warp_scalar.WarpScalar'>
```

```
<class 'mayavi.sources.array_source.ArraySource'>
<class 'mayavi.core.scene.Scene'>
<class 'mayavi.core.engine.Engine'>
```

Обратите внимание, когда вы программно строите *pipeline*, то явно указывать *ModuleManager* не нужно, также как *Scene* и *Engine*.

16.2 Работа с источниками данных

При формировании *pipeline* для решения задачи визуализации первое, что нужно сделать — это определиться с источником данных, который будет использоваться в конвейере. Данные, которые визуализируются с помощью *Mayavi* можно разделить на две большие группы — это связанные и несвязанные.

Набор несвязанных данных (*unconnected points*) представляет собой набор точек, координаты которых могут быть случайно распределены в пространстве, расстояния между ними и численные значения параметров в точках также могут иметь случайных характер. В таком наборе о конкретном элементе данных ничего нельзя сказать, зная значения соседних.

Набор связанных данных (*connected data points*) представляет собой структурированный, равномерно распределенный блок данных. Особенность его заключается в том, что соседние элементы (точки) функционально связаны друг с другом (отсюда и название), можно сказать, что они являются результатом интерполяции. Такие наборы ещё называют полем. Связанные данные разделяют на явно (*explicit*) и неявно (*implicit*) связанные. В первом случае предполагается, что данные представляются расположенными в решетчатой структуре, так,

например, описываются *2D*-изображения. Во втором связность между точками задаётся явно, это позволяет проводить линии в данных, определять объем фигур и т.п.

Наиболее часто используемые функции для построения источников несвязанных данных представлены ниже:

- `scalar_scatter()`
- `vector_scatter()`

Для построения связанных наборов используются следующие группы функции:

- создающие неявно связанные:
  - `scalar_field()`
  - `vector_field()`
  - `array2d_source()`
- создающие явно связанные:
  - `line_source()`
  - `triangular_mesh_source()`

В качестве исходных наборов данных, представленные выше функции, принимают *numpy*-массивы.

Рассмотрим эти функции более подробно.

Функция `scalar_scatter()` создаёт разреженный несвязанный источник данных, имеет следующие возможные сигнатуры вызова:

```
scalar_scatter(s, ...)
scalar_scatter(x, y, z, s, ...)
scalar_scatter(x, y, z, f, ...)
```

Параметр s – определяет набор данных: массив, который будет визуализирован. Если параметры x, y, z не заданы, то вместо них, в качестве координат, будут использоваться индексы из набора s. Вместо s можно указать функцию f, которая будет принимать координаты и возвращать соответствующее им значение.

Пример работы с scalar_scatter():

```
s = [1, 2, 3, 4, 5]
src_data = mlab.pipeline.scalar_scatter(s)
mlab.pipeline.glyph(src_data)
mlab.show()
```

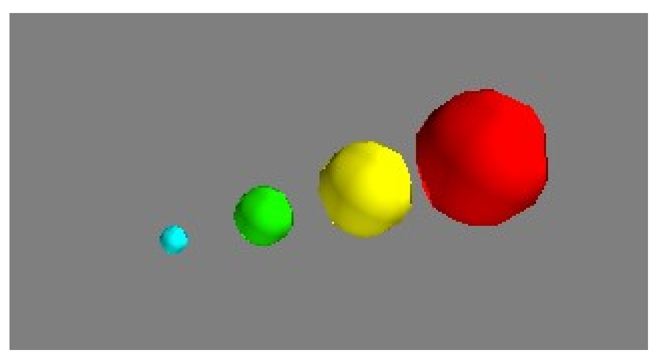


Рисунок 16.3 — Демонстрация работы функции mlab.pipeline.scalar_scatter()

Функция vector_scatter() создаёт разреженный источник векторных данных. Сигнатуры функции:

```
vector_scatter(u, v, w, ...)
vector_scatter(x, y, z, u, v, w, ...)
vector_scatter(x, y, z, f, ...)
```

Параметры u, v, w – это компоненты векторов, также как в функции scalar_scatter(), если не заданы x, y, z, то в качестве координат будут приняты индексы из наборов u, v, w. Функция f должна возвращать компоненты вектора для заданного набора координат.

Функция `scalar_field()` создаёт скалярное поле, сигнатура функции:

```
scalar_field(s, ...)
scalar_field(x, y, z, s, ...)
scalar_field(x, y, z, f, ...)
```

Параметры `s`, `x`, `y`, `z` – это *2D*, *3D*-наборы данных, которые, например, могут быть построены с помощью функций `numpy.ogird()`, `numpy.mgird()`.

Пример использования:

```
x, y, z = np.ogrid[0:5, 0:5, 0:5]
s = x*y*z

src = mlab.pipeline.scalar_field(s)
volume = mlab.pipeline.volume(src, vmin=0, vmax=10)
mlab.show()
```

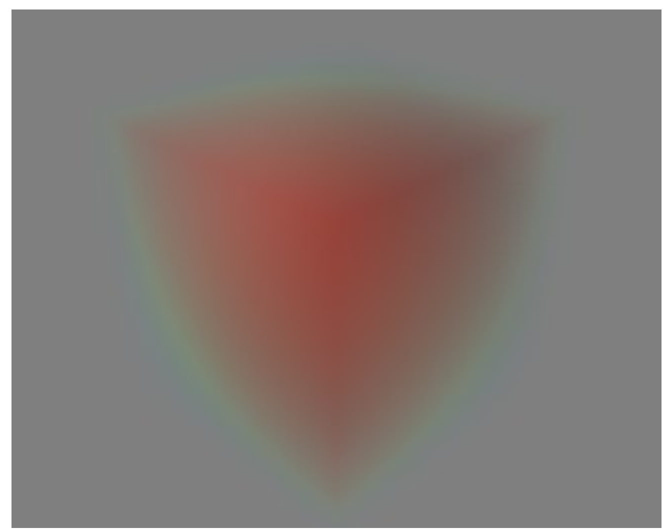


Рисунок 16.4 — Демонстрация работы функций `scalar_field()` и `volume()`

Функция `vector_filed()` аналогична функции `vector_scatter()` только для векторных данных.

Функция `array2d_source()` создаёт *2D*-источник связанных данных из *2D* исходного массива. Сигнатуры функций:

```
array2d_source(s, ...)
array2d_source(x, y, s, ...)
array2d_source(x, y, f, ...)
```

Параметры `s`, `x`, `y` — *2D*-массивы, подобные тем, что могут быть построены с помощью `numpy.ogird()`, `numpy.mgird()`.

Пример использования:

```
x, y = np.ogrid[-2:2:0.1, -2:2:0.1]
z = np.cos(x*y)

src = mlab.pipeline.array2d_source(z)
img = mlab.pipeline.image_actor(src)
mlab.show()
```

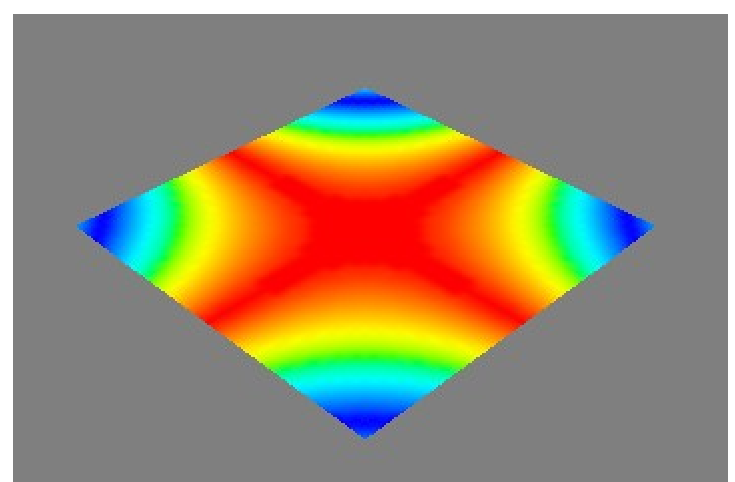


Рисунок 16.5 — Демонстрация работы функций `array2d_source()` и `image_actor()`

Функция `line_source()` формирует линейный набор данных. Сигнатуры функций:

```
line_source(x, y, z, ...)
line_source(x, y, z, s, ...)
```

```
line_source(x, y, z, f, ...)
```

Параметры x, y, z – координаты точек линии, s – массив со значениями, определяющими свойства линии.

```
t = np.linspace(0, 5*np.pi, 100)
x = t * np.cos(t)
y = t * np.sin(t)
z = t

line = mlab.pipeline.line_source(x, y, z)
tube = mlab.pipeline.tube(line, tube_radius=1)
surf = mlab.pipeline.surface(tube)
mlab.show()
```

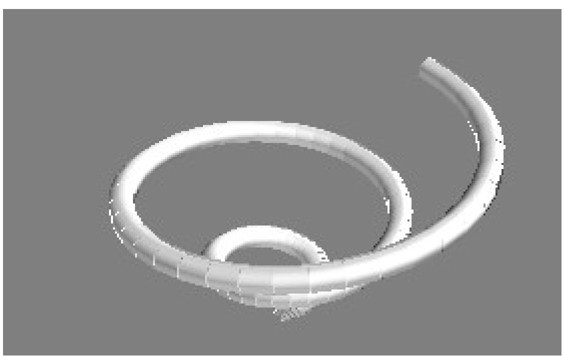


Рисунок 16.6 — Демонстрация работы функций `line_source()` и `surface()`

16.3 Работа с фильтрами

Фильтры в *pipeline*'е *Mayavi*[24] предназначены только для трансформации данных, они не производят визуализацию данных. Рассмотрим некоторые из фильтров, которые могут быть полезны в работе.

24 https://docs.enthought.com/mayavi/mayavi/filters.html#filters

CellDerivatives

Вычисляет производные для входного набора данных (скалярные либо векторные данные), строит набор ячеек, значениями в которых будут производные.

Пример:

```
x, y = np.mgrid[-2:2:0.1, -2:2:0.1]
z = np.cos(x*y) * np.sin(x*y)

src = mlab.pipeline.array2d_source(z)
cell_drv = mlab.pipeline.cell_derivatives(src)
mlab.pipeline.glyph(cell_drv, mode='cube')
mlab.pipeline.image_actor(src)
```

Для изображения, представленного на рисунке 16.7, будет построено поле со значениями производных, изображённое на рисунке 16.8.

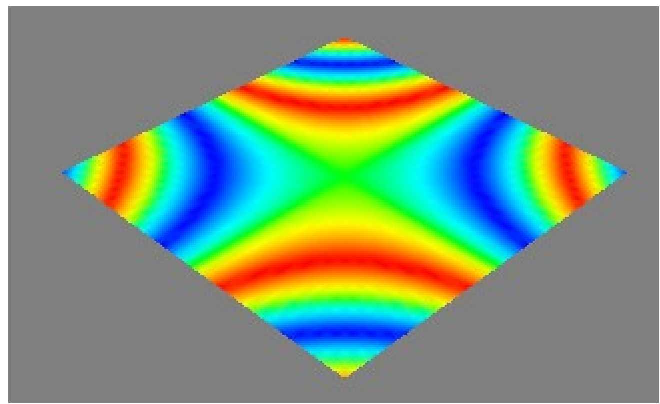


Рисунок 16.7 — Исходное изображение

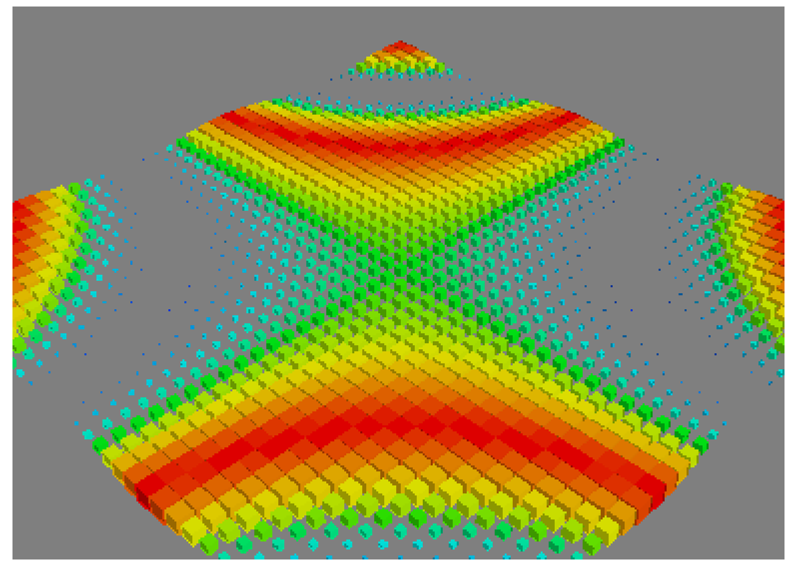


Рисунок 16.8 — Демонстрация работы фильтра *CellDerivatives*

CellToPointData

Преобразует *cell* атрибут с данными в *point* атрибут, значение для *point* определяется как среднее ячеек, к которым эта точка принадлежит. Существует обратный к нему фильтр *PointToCellData.*

Contour

Контурный фильтр для создания изоповерхностей.

CutPlane

Интерактивный фильтр с *3D*-инструментом на сцене, который позволяет строить срезы для наборов данных.

Пример:

```
x, y, z = np.ogrid[-7:7:0.5, -7:7:0.5, -1:7:0.5]
scalars = x*x + y*y - z*z

src = mlab.pipeline.scalar_field(scalars)
cut_pl = mlab.pipeline.cut_plane(src)
mlab.pipeline.surface(cut_pl)
mlab.pipeline.iso_surface(src, contours=3, transparent=True)
```

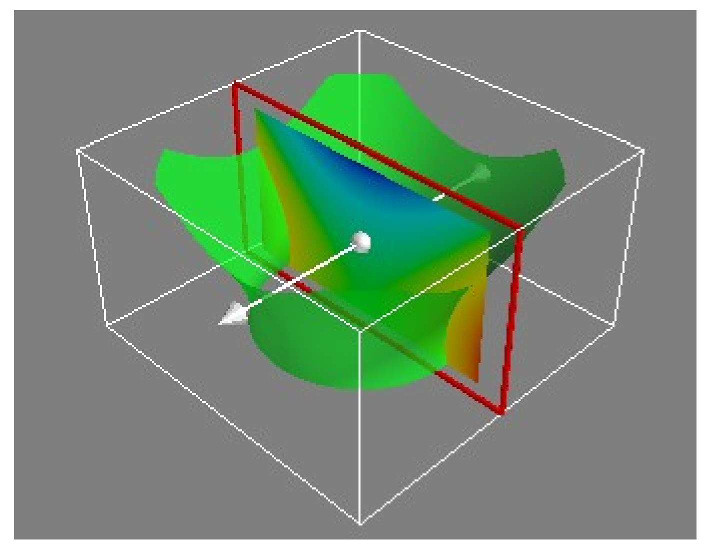


Рисунок 16.9 — Демонстрация работы фильтра *CutPlane*

DataSetClipper

Фильтр закрепляет набор данных в определённой области:

```
x, y, z = np.ogrid[-7:7:0.5, -7:7:0.5, -1:7:0.5]
scalars = x*x + y*y - z*z

src = mlab.pipeline.scalar_field(scalars)
dsc = mlab.pipeline.data_set_clipper(src)
mlab.pipeline.glyph(dsc)
```

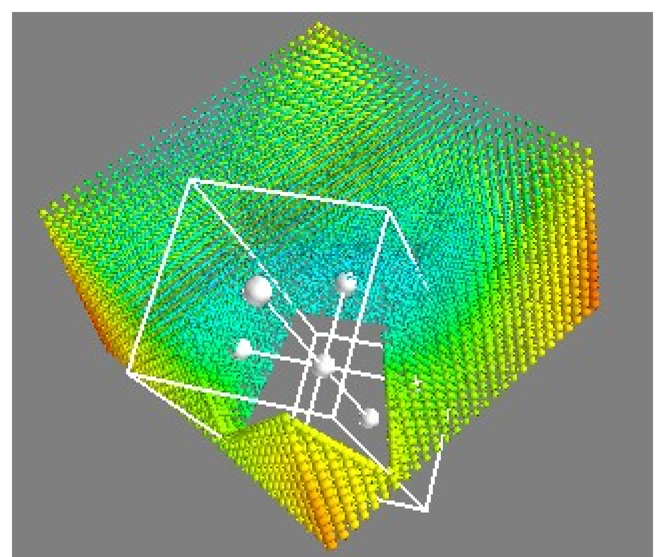


Рисунок 16.10 — Демонстрация работы фильтра *DataSetClipper*

ExtractEdges

Извлекает границы ячеек из набора данных:

```
x, y, z = np.ogrid[-7:7:0.5, -7:7:0.5, -1:7:0.5]
scalars = x*x + y*y - z*z
src = mlab.pipeline.scalar_field(scalars)
edges = mlab.pipeline.extract_edges(src)
mlab.pipeline.surface(edges)
mlab.show()
```

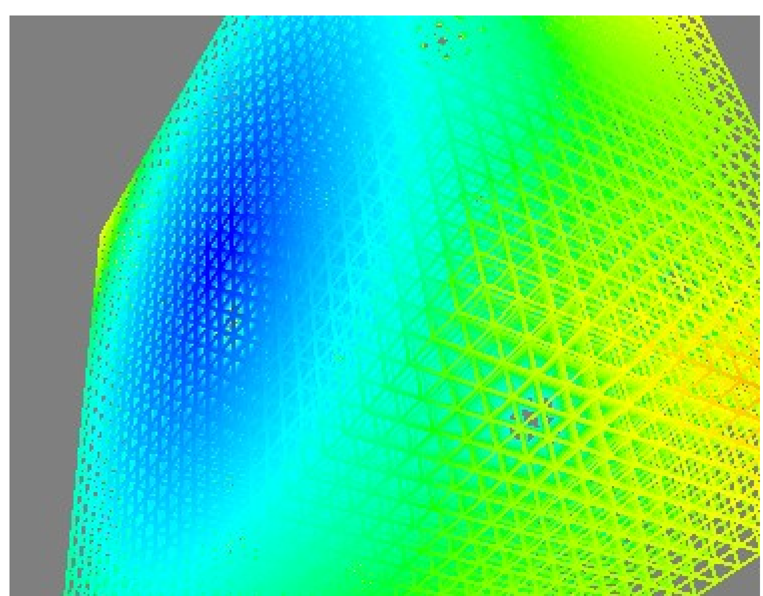


Рисунок 16.11 — Демонстрация работы фильтра *ExtractEdges*

ExtractGrid

Позволяет извлечь часть данных из исходного набора:

```
x, y, z = np.ogrid[-7:7:0.5, -7:7:0.5, -1:7:0.5]
scalars = x*x + y*y - z*z
src = mlab.pipeline.scalar_field(scalars)
grid = mlab.pipeline.extract_grid(src)
grid.x_min = 5
grid.x_max = 20
grid.y_min = 5
grid.y_max = 20
mlab.pipeline.surface(grid)
mlab.show()
```

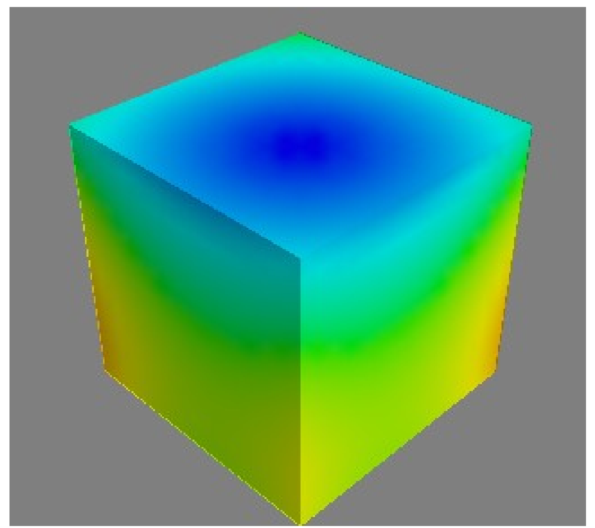


Рисунок 16.12 — Демонстрация работы фильтра *ExtractGrid*

ExtractVectorNorm

Фильтр вычисляет норму (длину) вектора. Применяется при работе с векторными данными. Например, если визуализировать векторные данные без построения длин, то результат будет таким:

```
x, y, z = np.mgrid[0:3, 0:3, 0:3]
u = x * x
v = y
w = z
src = mlab.pipeline.vector_scatter(u, v, w)
mlab.pipeline.glyph(src, mode="2darrow")
mlab.show()
```

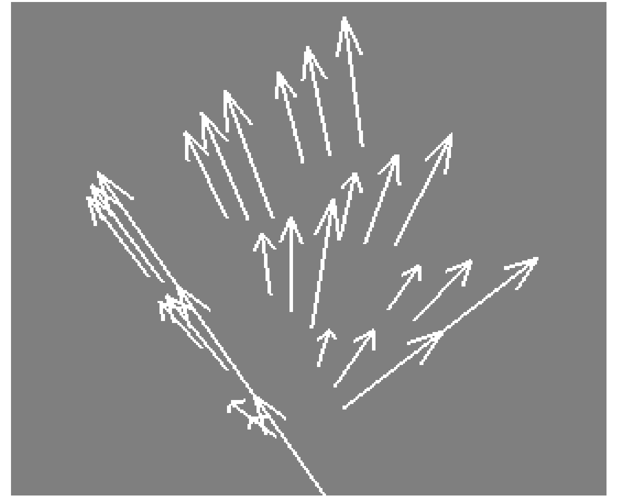


Рисунок 16.13 — Исходная модель

В этом случае мы видим только направления векторов. Добавим вычисление длин векторов:

```
src = mlab.pipeline.vector_scatter(u, v, w)
norms = mlab.pipeline.extract_vector_norm(src)
mlab.pipeline.glyph(norms, mode="2darrow")
mlab.show()
```

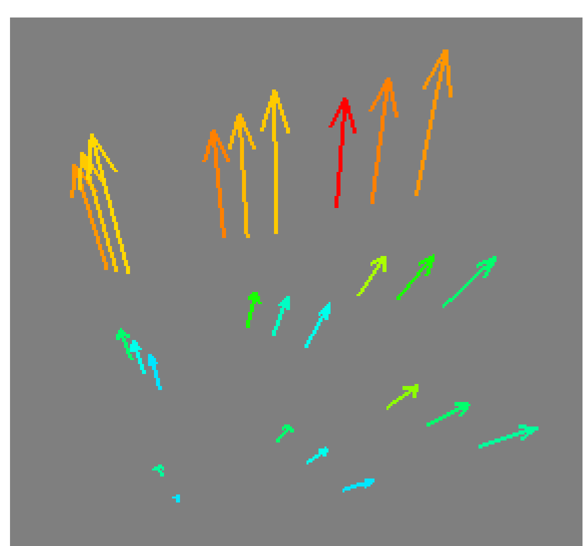


Рисунок 16.14 — Демонстрация работы фильтра *ExtractVectorNorm*

ExtractVectorComponents

Извлекает компоненты векторов:

Извлечём из переданных векторов *y*-компоненты:

```
x, y, z = np.mgrid[0:3, 0:3, 0:3]
u = x * x
v = y
w = z

src = mlab.pipeline.vector_scatter(u, v, w)
comps = mlab.pipeline.extract_vector_components(src)
comps.component = 'y-component'
mlab.pipeline.glyph(comps, mode="2darrow")

mlab.show()
```

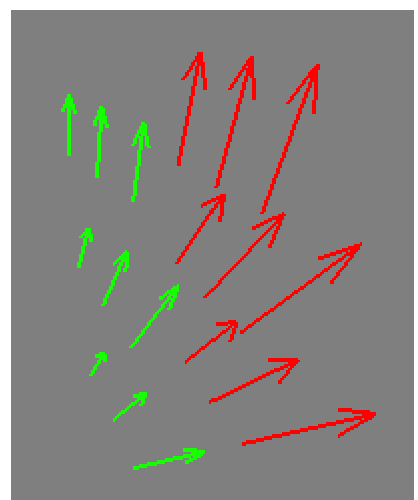


Рисунок 16.15 — Демонстрация работы фильтра *ExtractVectorComponents*

GaussianSplatter

Фильтр размещает точки в объеме с эллиптическим гауссовым распределением. Это полезно для оценки поля плотности по набору рассеянных точек.

Для набора, который выглядит так, как показано на рисунке 16.16.

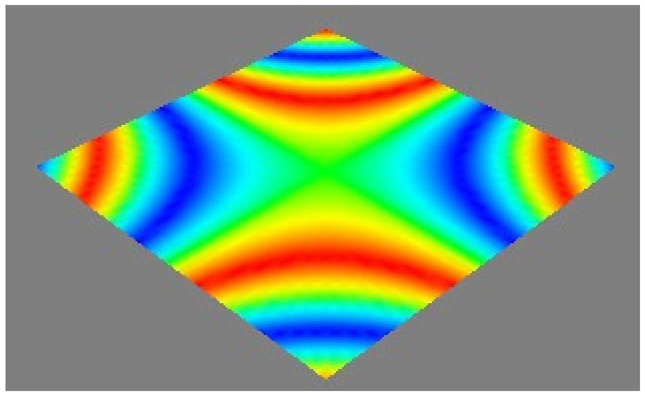


Рисунок 16.16 — Исходная модель

Проведём предварительную обработку фильтром *GaussianSplatter*:

```
src = mlab.pipeline.array2d_source(z)
gsp = mlab.pipeline.gaussian_splatter(src)
mlab.pipeline.glyph(gsp, mode='cone')
mlab.show()
```

Получим следующую модель:

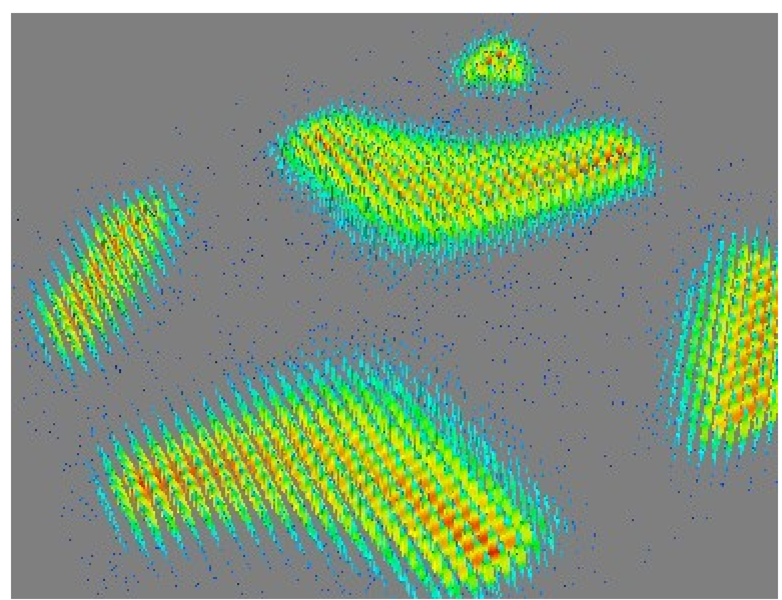


Рисунок 16.17 — Демонстрация работы фильтра *GaussianSplatter*

Используя другие модули для визуализации можно получить довольно интересные модели:

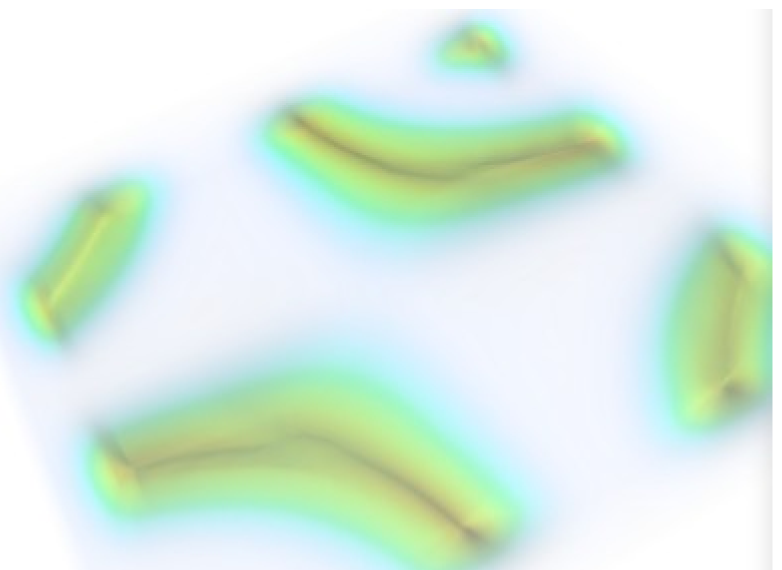


Рисунок 16.18 — Демонстрация работы с параметром `mode='Volume'` фильтра *GaussianSplatter*

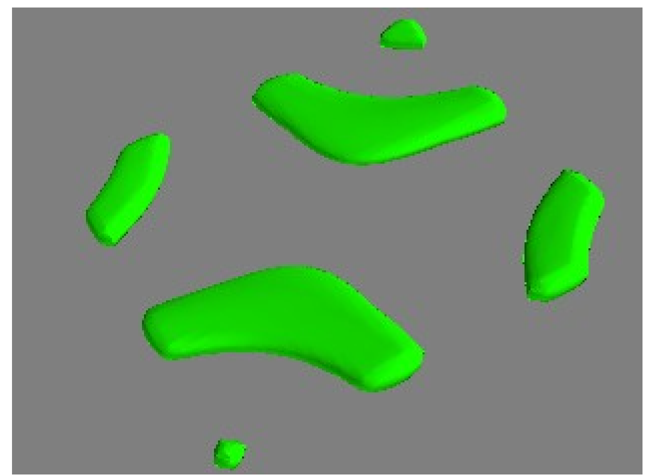


Рисунок 16.19 — Демонстрация работы с параметром `mode='isosurface'` фильтра *GaussianSplatter*

GreedyTerrainDecimation

Аппроксимирует *2D*-изображение набором треугольников, сохраняя их количество минимальным:

```
x, y = np.mgrid[-2:2:0.1, -2:2:0.1]
z = np.cos(x*y) * np.sin(x*y)

src = mlab.pipeline.array2d_source(z)
gtd = mlab.pipeline.greedy_terrain_decimation(src)
mlab.pipeline.surface(gtd)

mlab.show()
```

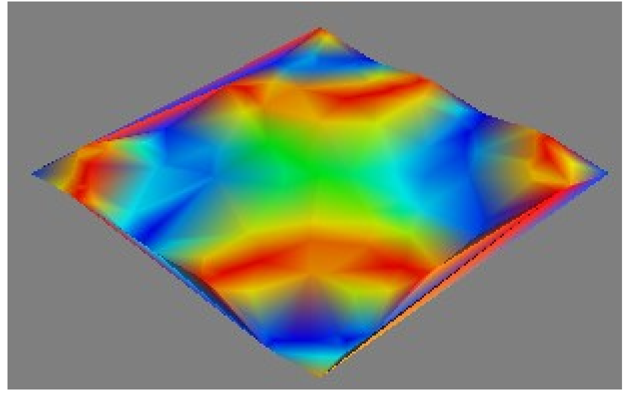


Рисунок 16.20 — Демонстрация работы фильтра *GreedyTerrainDecimation*

ImageChangeInformation

Фильтр позволяет задать расположение координатных осей, растяжение по осям и т.п. без модификации самих данных.

MaskPoints

Фильтр для формирования подвыборок данных.

PolyDataNormals

Вычисляет нормали для входных данных, используется для сглаживания изображений и т.п.

Для примера построим модель с большим количеством неровностей, для этого воспользуемся фильтром *WarpScalar*:

```
z = np.random.rand(15,15)

src = mlab.pipeline.array2d_source(z)
warp = mlab.pipeline.warp_scalar(src)
mlab.pipeline.surface(warp)
mlag.show()
```

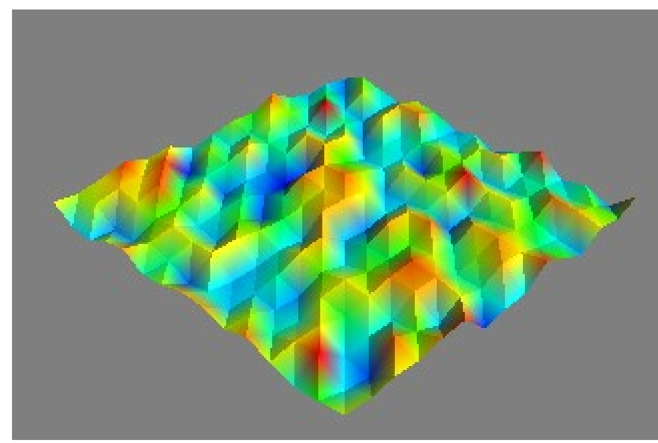


Рисунок 16.21 — Демонстрация работы фильтра *WarpScalar*

Теперь сгладим неровности, добавив ещё один уровень фильтрации, с помощью *PolyDataNormals*:

```
z = np.random.rand(15,15)
src = mlab.pipeline.array2d_source(z)
warp = mlab.pipeline.warp_scalar(src)
poly = mlab.pipeline.poly_data_normals(warp)
mlab.pipeline.surface(poly)
mlab.show()
```

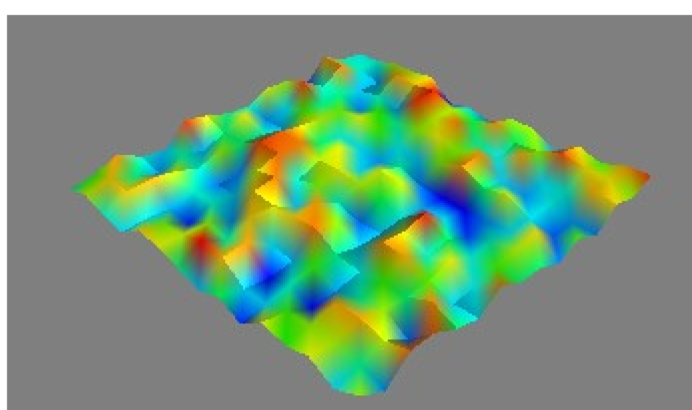


Рисунок 16.22 — Демонстрация работы фильтра *PolyDataNormals*

QuadricDecimation

Уменьшает количество треугольников в исходной сетке, формируя лучшее представление модели.

SelectOutput

Используется при работе с мультиблоковыми источниками данных для выбора выхода.

Stripper

Фильтр применяется для заполнения разрывов поверхностей. Создадим трубку вокруг линии без фильтра *stripper*:

```
t = np.linspace(0, 5*np.pi, 100)
x = t * np.cos(t)
y = t * np.sin(t)
z = t
```

```
src = mlab.pipeline.line_source(x,y,z, z)
tb = mlab.pipeline.tube(src, tube_radius=1.5, tube_sides=10)
mlab.pipeline.surface(tb)
mlab.show()
```

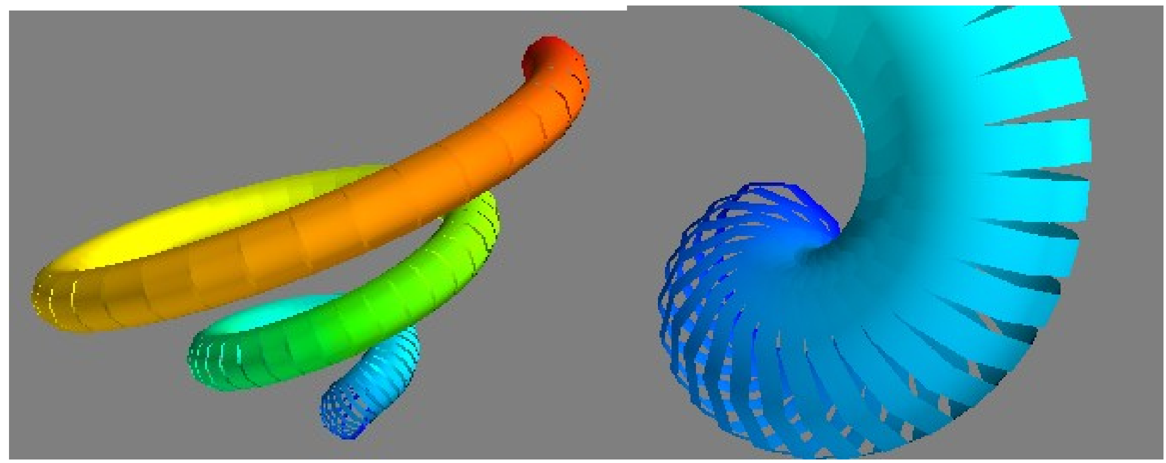


Рисунок 16.23 — Модель без фильтра *Stripper*

```
src = mlab.pipeline.line_source(x,y,z, z)
stp = mlab.pipeline.stripper(src)
tb = mlab.pipeline.tube(stp, tube_radius=1.5, tube_sides=10)
mlab.pipeline.surface(tb)
mlab.show()
```

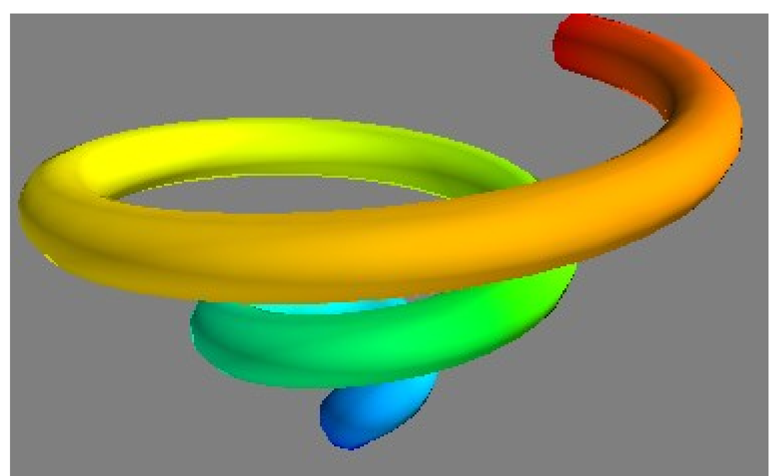


Рисунок 16.24 — Демонстрация работы фильтра *Stripper*

Threshold

Фильтр для задания пороговых значений для входных данных.

TransformData

Осуществляет линейные преобразования входных данных.

Tube

Превращает линии в трубки.

UserDefined

Позволяет использовать фильтры, спроектированные пользователем.

Vorticity

Вычисляет завихрённость входного векторного поля.

WarpScalar

Изменяет (искривляет) входные данные вдоль определённого направления с заданным масштабом.

WarpVector

Изменяет (искривляет) входные данные вдоль вектора.

16.4 Работа с модулями

Модули в *pipeline Mayavi* отвечают за визуализацию данных и отображение дополнительной информации на сцене. Данные могут быть переданы в модуль как напрямую, так и через ряд фильтров, если требуется выполнить дополнительные преобразования. Сгруппируем модули по функциональному признаку, так их будет проще изучать.

Модули для размещения аннотаций на сцене.

Особенность этих модулей состоит в том, что они располагаются в пакете `mlab`, а не в `mlab.pipeline`. К этим модулям относятся:

- *Axes*
    - Добавляет оси координат к модели.

- *Outline*
    - Строит вокруг модели параллелепипед.
- *OrientationAxes*
    - Добавляет на сцену элемент с направлениями осей.
- *Text*, *Text3D*
    - Добавляет на сцену текст.

```
x, y, z = np.ogrid[-7:7:0.1, -7:7:0.1, -1:7:0.1]
scalars = x*x + y*y - z*z

src = mlab.pipeline.scalar_field(scalars)
mlab.pipeline.iso_surface(src)
mlab.axes()
mlab.outline()
mlab.orientation_axes()
mlab.text(0, 0, "Low point")
```

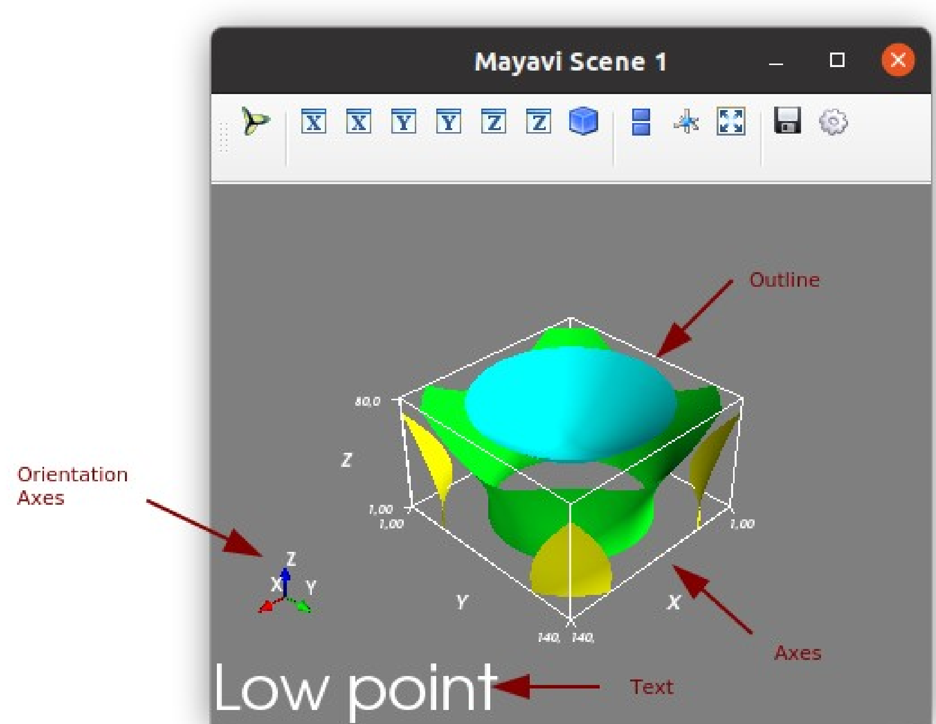


Рисунок 16.25 — Модули *Mayavi*

Модули для работы со скалярными данными.

Glyph

Визуализирует данные в виде глифов:

```
x, y, z = np.ogrid[-7:7:0.5, -7:7:0.5, -1:7:0.5]
scalars = x*x + y*y - z*z

src = mlab.pipeline.scalar_field(scalars)
mlab.pipeline.glyph(src)
mlab.show()
```

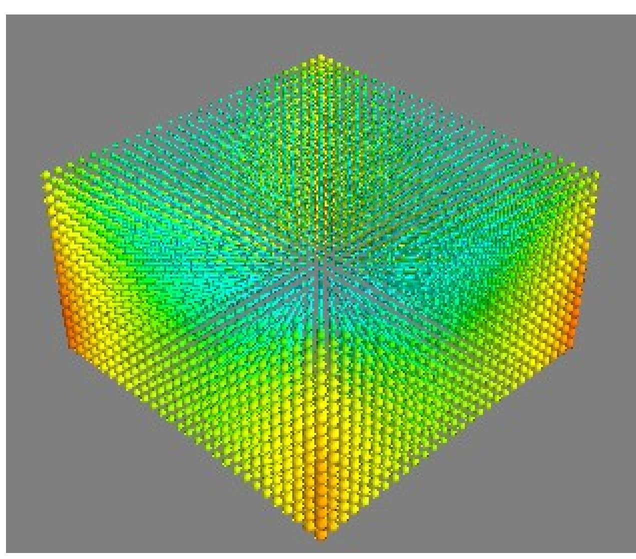


Рисунок 16.26 — Демонстрация работы фильтра *Glyph*

IsoSurface

Представление данных в виде изоповерхностей:

```
src = mlab.pipeline.scalar_field(scalars)
mlab.pipeline.iso_surface(src)
mlab.show()
```

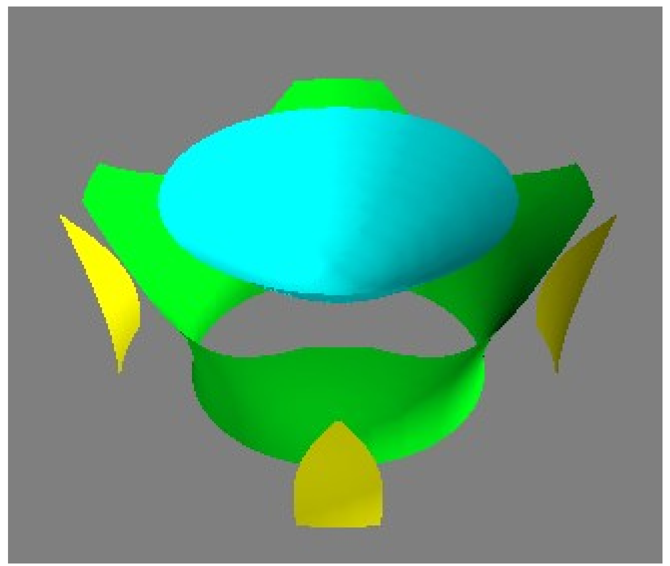


Рисунок 16.27 — Демонстрация работы фильтра *IsoSurface*

ScalarCutPlane

Представляет только часть набора данных, которая выделяется секущей плоскостью:

```
src = mlab.pipeline.scalar_field(scalars)
mlab.pipeline.scalar_cut_plane(src)

mlab.show()
```

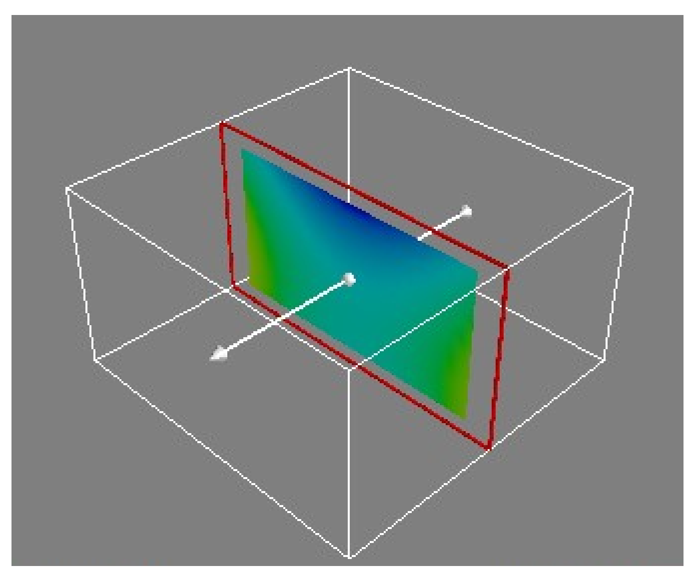


Рисунок 16.28 — Демонстрация работы модуля *ScalarCutPlane*

Surface

Строит поверхность по переданному набору данных:

```
src = mlab.pipeline.scalar_field(scalars)
mlab.pipeline.surface(src)
mlab.show()
```

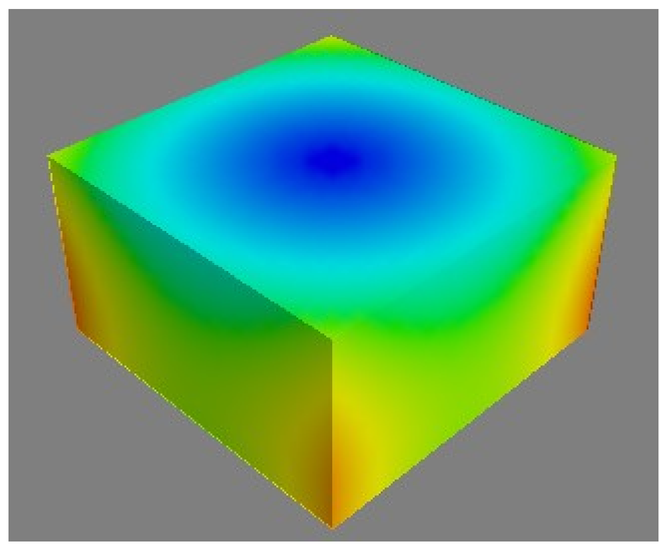


Рисунок 16.29 — Демонстрация работы модуля *Surface*

Модули для визуализации векторных данных.

VectorCutplane

Строит срезы для векторных данных:

```
x, y, z = np.mgrid[0:3:0.5, 0:3:0.5, 0:3:0.5]
u = x * x
v = y
w = z

src = mlab.pipeline.vector_field(u,v,w)
mlab.pipeline.vector_cut_plane(src)
mlab.show()
```

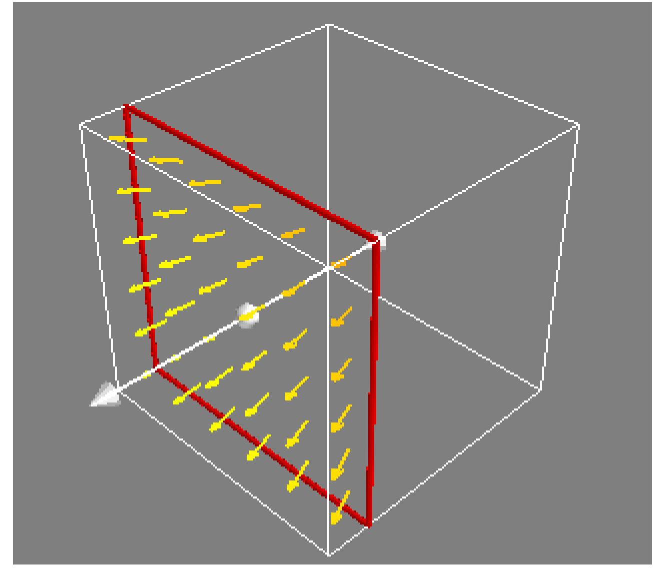


Рисунок 16.30 — Демонстрация работы модуля *VectorCutplane*

Vectors

Визуализирует переданный набор векторов:

```
x, y, z = np.mgrid[0:3:0.5, 0:3:0.5, 0:3:0.5]
u = x * x
v = y
w = z
src = mlab.pipeline.vector_field(u,v,w)
mlab.pipeline.vectors(src)
mlab.show()
```

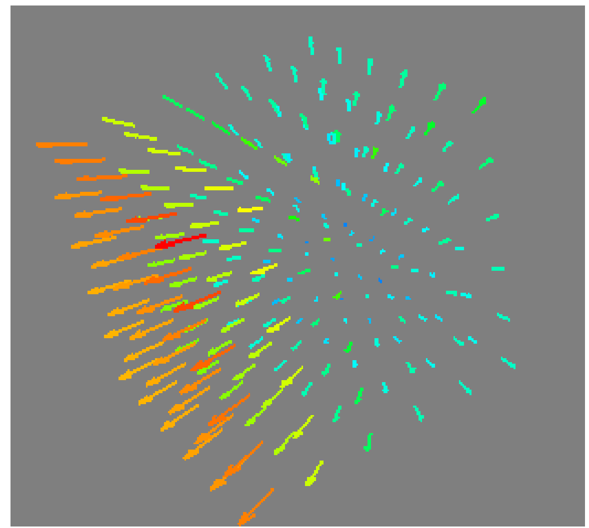


Рисунок 16.31 — Демонстрация работы модуля *Vectors*

Streamline

Интерактивный инструмент, который позволяет изучать линии потока, построенные по переданному набору векторов:

```
x, y, z = np.mgrid[0:3:0.5, 0:3:0.5, 0:3:0.5]
u = x * x
v = y
w = z

src = mlab.pipeline.vector_field(u,v,w)
mlab.pipeline.streamline(src, seedtype='plane')
mlab.show()
```

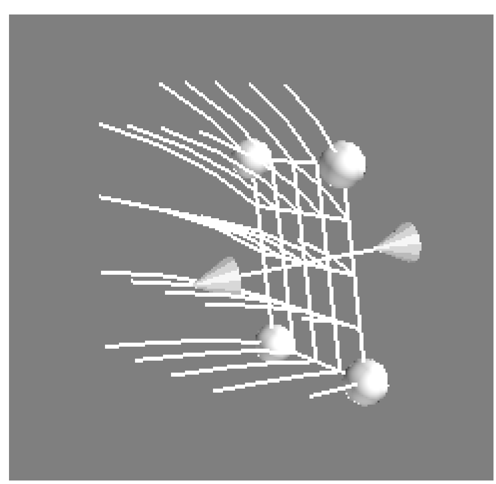


Рисунок 16.32 — Демонстрация работы модуля *Streamline*

Модули для работы с данными.

GridPlane

Модуль располагает плоскость с сеткой на модели:

```
x, y, z = np.ogrid[-7:7:0.5, -7:7:0.5, -1:7:0.5]
scalars = x*x + y*y - z*z

src = mlab.pipeline.scalar_field(scalars)
mlab.pipeline.grid_plane(src)
mlab.show()
```

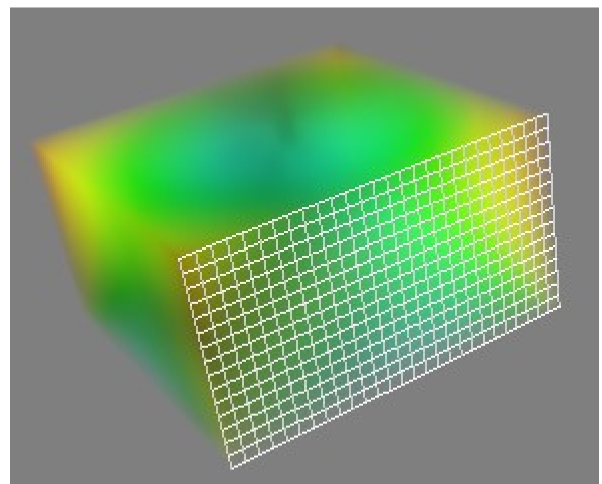


Рисунок 16.33 — Демонстрация работы модуля *GridPlane*

ContourGridPlane

Плоскость с набором контуров для заданного набора данных.

CustomGridPlane

Позволяет создавать настраиваемую пользователем плоскость с сеткой.

ImagePlaneWidget

Строит интерактивную секущую плоскость для заданного набора данных:

```
x, y, z = np.ogrid[-7:7:0.5, -7:7:0.5, -1:7:0.5]
scalars = x*x + y*y - z*z

src = mlab.pipeline.scalar_field(scalars)
mlab.pipeline.image_plane_widget(src)
mlab.pipeline.volume(src)
mlab.show()
```

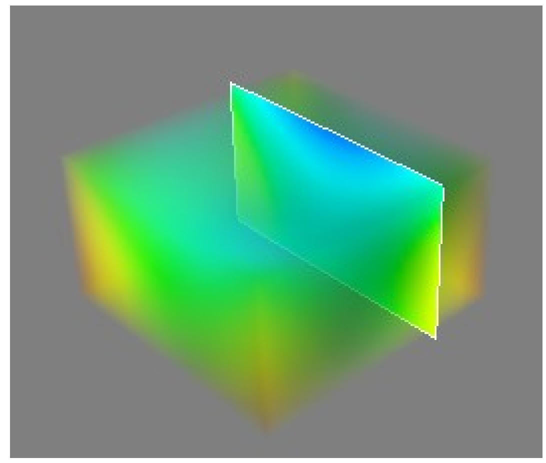


Рисунок 16.34 — Демонстрация работы модуля *ImagePlaneWidget*

ImageActor

Представляет набор данных как двухмерное изображение:

```
x, y = np.mgrid[-2:2:0.1, -2:2:0.1]
z = np.cos(x*y) * np.sin(x*y)

src = mlab.pipeline.scalar_field(z)
mlab.pipeline.image_actor(src)
mlab.show()
```

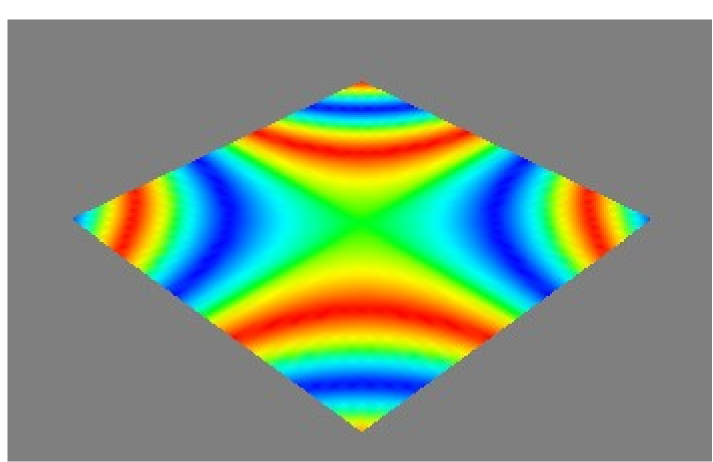


Рисунок 16.35 — Демонстрация работы модуля *ImageActor*

Volume

Строит объёмную модель:

```
x, y, z = np.ogrid[-7:7:0.5, -7:7:0.5, -1:7:0.5]
scalars = x*x + y*y - z*z

src = mlab.pipeline.scalar_field(scalars)
mlab.pipeline.volume(src)
mlab.show()
```

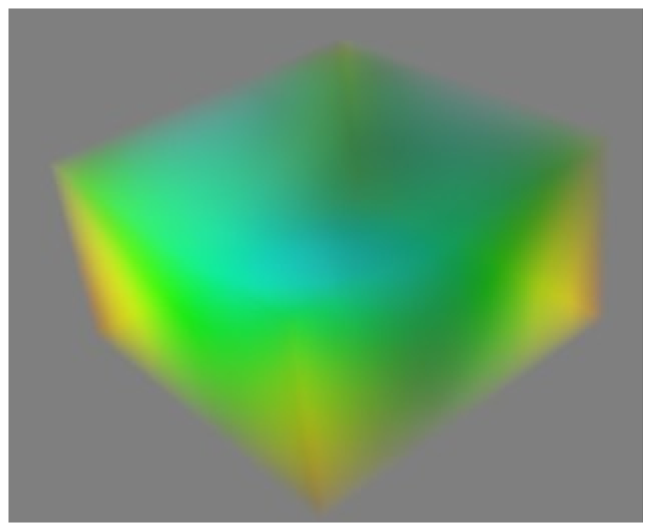


Рисунок 16.36 — Демонстрация работы модуля *Volume*

Заключение

Мы постарались как можно более полно рассказать о библиотеках *Matplotlib*, *Seaborn* и *Mayavi*, надеемся, что информация, которую вы встретили на страницах этой книги, оказалась полезной. Если у вас есть замечания или пожелания по содержанию, то пишите нам на devpractice.mail@gmail.com, мы будем рады обратной связи.