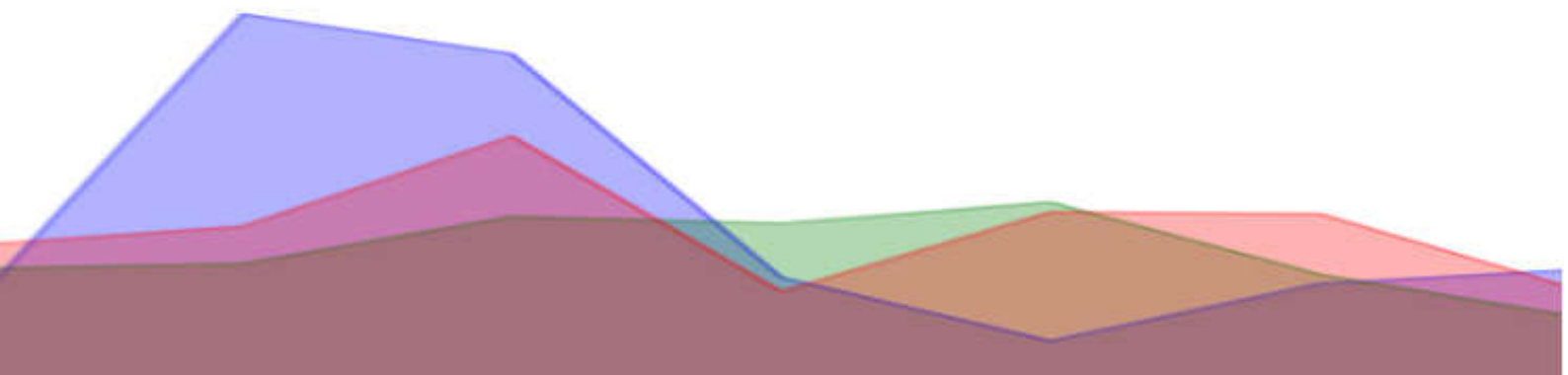
Pandas

Работа с данными

Devpractice Team

Второе издание



УДК 004.6
ББК: 32.973

Devpractice Team. Pandas. Работа с данными. 2-е изд. - devpractice.ru. 2020. - 170 с.: ил.

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

УДК 004.6
ББК: 32.973

Материал составил и подготовил:
Абдрахманов М.И.

Оглавление

Введение

Что такое *pandas*?

Pandas - это библиотека, которая предоставляет очень удобные, с точки зрения использования, инструменты для хранения и работы с данными. Если вы занимаетесь анализом данных или машинным обучением и при этом используете язык *Python*, то знание *pandas* значительно упростит вам работу.

Pandas входит в группу проектов, спонсируемых *numfocus* (*https://www.numfocus.org/*). *Numfocus* - это организация, которая поддерживает различные проекты и программное обеспечение научной тематики.

Официальный сайт *pandas* находится здесь *http://pandas.pydata.org/*. Стоит отметить, что у данного продукта хорошая документация, можете перейти по ссылке (*http://pandas.pydata.org/pandas-docs/stable/*) чтобы ознакомиться с ней.

Особенность *pandas* состоит в том, что эта библиотека очень быстрая и гибкая, а т.к. она используется с языком *Python*, который не отличается высокой производительностью, вопрос быстродействия, при работе с большими объемами данных, становится одним из ключевых. *Pandas* прекрасно подходит для работы с одномерными и двумерными таблицами данных, хорошо интегрирован с внешним миром: есть возможность работать с файлами *CSV*, таблицами *Excel*, может стыковаться с языком *R*.

Установка *pandas*

Для проведения научных расчетов, анализа данных и построения моделей в рамках машинного обучения, для языка *Python* существуют прекрасное решение - *Anaconda*. *Anaconda* - это пакет, который содержит в себе большой набор различных библиотек, интерпретатор языка *Python* и несколько сред для разработки.

Pandas присутствует в стандартной поставке *Anaconda*. Если же его там нет, то его можно установить отдельно. Для этого стоит воспользоваться пакетным менеджером, который входит в состав *Anaconda*, который называется *conda*. Для его запуска необходимо перейти в каталог *[Anaconda install path]\Scripts* в *Windows*. В операционной системе *Linux*, после установки *Anaconda* менеджер *conda* должен быть доступен везде.

Введите командной строке:

```
conda install pandas
```

В случае, если требуется конкретная версия *pandas*, то ее можно указать при установке:

```
conda install pandas=0.13.1
```

При необходимости, можно воспользоваться пакетным менеджером *pip*, входящим в состав дистрибутива *Python*:

```
pip install pandas
```

Если вы используете *Linux*, то есть ещё одни способ установить *pandas* - это воспользоваться пакетным менеджером самой операционной системы. Для *Ubuntu* это выглядит так:

```
sudo apt-get install python-pandas
```

После установки необходимо проверить, что *pandas* установлен и корректно работает. Для этого запустите интерпретатор *Python* и введите в нем следующие команды (набор символов >>> вводить не нужно, это приглашение интерпретатора *Python*):

```
>>> import pandas as pd
>>> pd.test()
```

В результате, в окне терминала должен появиться следующий текст:

```
Running unit tests for pandas
pandas version 0.18.1
numpy version 1.11.1
pandas is installed in c:\Anaconda3\lib\site-packages\pandas
Python version 3.5.2 |Anaconda 4.1.1 (64-bit)| (default, Jul 5 2016,
11:41:13) [MSC v.1900 64 bit (AMD64)]
nose version 1.3.7
..........
----------------------------------------------------------------
Ran 11 tests in 0.422s
OK
```

Это означает, что *pandas* установлен и его можно использовать.

Глава 1. Структуры данных в *Pandas*: *Series* и *DataFrame*

Библиотека *pandas* предоставляет две структуры: *Series* и *DataFrame* для быстрой и удобной работы с данными (на самом деле их три, есть еще одна структура - *Panel*, но в данный момент она имеет статус *deprecated* и в будущем будет исключена из состава библиотеки *pandas*).

Series - это маркированная одномерная структура данных, ее можно представить, как таблицу с одной строкой (или столбцом). С *Series* можно работать как с обычным массивом (обращаться по номеру индекса), и как с ассоциированным массивом, в этом случае можно использовать ключ для доступа к элементам данных.

DataFrame - это двумерная маркированная структура. Идейно она очень похожа на обычную таблицу, что выражается в способе ее создания и работе с ее элементами.

Panel - представляет собой трехмерную структуру данных. О *Panel* мы больше говорить не будем. В рамках этой части мы остановимся на вопросах создания и получения доступа к элементам данных структур *Series* и *DataFrame*.

1.1 Структура данных *Series*

Для того чтобы начать работу со структурами данных из *pandas* требуется предварительно импортировать соответствующий модуль. Убедитесь, что библиотека *pandas* установлена на вашем компьютере, о том, как это сделать, можно прочитать во введении книги. Также будем считать, что вы знакомы с языком *Python*.

Помимо самого *pandas* нам понадобится библиотека *numpy*. Для экспериментов рекомендуем интерактивную оболочку *IPython*, ее можно запускать как отдельное приложение или использовать в рамках какой-нибудь *IDE* (*Spyder*, *PyCharm*), либо воспользоваться IDLE, но она не подойдет для работы с графиками.

Если строки кода будут содержать префикс в виде символа >>>, то это означает, что данные команды мы вводим в интерактивной оболочке, в ином случае, это будет означать, что код написан в редакторе.

Пора переходить к практике! Для начала импортируем нужные нам библиотеки:

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

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

- словарь (dict) *Python*;
- список (list) *Python*;
- массив ndarray (из библиотеки numpy);
- скалярная величина.

Конструктор класса *Series* выглядит следующим образом:

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

Опишем некоторые из параметров конструктора:

- `data`*:* массив, скалярное значение, `dict`; значение по умолчанию: `None`
  - Структура, на базе которой будет построен *Series.*
- `index`: одномерный массив; значение по умолчанию: `None`
  - Список меток, который будет использоваться для доступа к элементам *Series*. Длина списка должна быть равна длине `data`.
- `dtype`*:* `numpy.dtype`; значение по умолчанию: `None`
  - Объект, определяющий тип данных.
- `copy`*:* `bool`; значение по умолчанию: `False`
  - Если параметр равен `True`, то будет создана копия массива данных.

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

1.1.1 Создание *Series* из списка *Python*

Самый простой способ создать *Series* - это передать в качестве единственного параметра в конструктор список *Python*:

```
>>> s1 = pd.Series([1, 2, 3, 4, 5])
>>> s1
0    1
1    2
2    3
3    4
4    5
dtype: int64
```

В примере была создана структура *Series* на базе списка из языка *Python*. Для доступа к элементам *Series*, в данном случае, можно использовать только положительные целые числа - левый столбец чисел, начинающийся с нуля - это как раз и есть индексы элементов структуры, которые представлены в правом столбце.

Передадим в качестве второго элемента список строк (в нашем случае - это отдельные символы). Это позволит обращаться к элементам структуры *Series* не только по численному индексу, но и по метке, что сделает работу с таким объектом, похожей на работу со словарем:

```
>>> s2 = pd.Series([1, 2, 3, 4, 5], ['a', 'b', 'c', 'd', 'e'])
>>> print(s2)
a    1
b    2
c    3
d    4
e    5
dtype: int64
```

Обратите внимание на левый столбец, в нем содержатся метки, которые мы передали в качестве `index`-параметра при создании структуры. Правый столбец - это по-прежнему элементы структуры.

1.1.2 Создание *Series* из `ndarray` массива из `numpy`

Создадим массив `ndarray` из пяти чисел, аналогичный списку из предыдущего раздела. Библиотеки `pandas` и `numpy` должны быть предварительно импортированы:

```
>>> ndarr = np.array([1, 2, 3, 4, 5])
>>> type(ndarr)
<class 'numpy.ndarray'>
```

Теперь создадим *Series* с буквенными метками:

```
>>> s3 = pd.Series(ndarr, ['a', 'b', 'c', 'd', 'e'])
>>> print(s3)
a    1
b    2
c    3
d    4
e    5
dtype: int32
```

Как вы можете видеть, результат аналогичен тому, чтобы был получен с использованием списков *Python*.

1.1.3 Создание *Series* из словаря (`dict`)

Еще один способ создать структуру *Series* - это использовать словарь для одновременного задания меток и значений:

```
>>> d = {'a':1, 'b':2, 'c':3}
>>> s4 = pd.Series(d)
>>> print(s4)
a    1
b    2
c    3
dtype: int64
```

Ключи (`keys`) из словаря `d` станут метками структуры `s4`, а значения (`values`) словаря — значениями.

1.1.4 Создание *Series* с использованием константы

Рассмотрим еще один способ создания *Series*. На этот раз значения в ячейках структуры будут одинаковыми:

```
>>> a = 7
>>> s5 = pd.Series(a, ['a', 'b', 'c'])
>>> print(s5)
a    7
b    7
c    7
dtype: int64
```

В полученной структуре имеется три элемента с одинаковым содержанием.

1.1.5 Работа с элементами *Series*

Индексации и работе с элементами *Series* и *DataFrame* будет посвящена отдельная глава, сейчас рассмотрим основные подходы, которые предоставляет *pandas*.

К элементам *Series* можно обращаться по численному индексу, при таком подходе работа со структурой не отличается от работы со списками в *Python:*

```
>>> s6 = pd.Series([1, 2, 3, 4, 5], ['a', 'b', 'c', 'd', 'e'])
>>> s6[2]
3
```

Можно использовать метку, тогда работа с *Series* будет похожа на работу со словарем (`dict`) в *Python:*

```
>>> s6['d']
4
```

Доступен синтаксис работы со срезами:

```
>>> s6[:2]
a    1
b    2
dtype: int64
```

В поле для индекса можно поместить условное выражение:

```
>>> s6[s6 <= 3]
a    1
b    2
c    3
dtype: int64
```

Со структурами *Series* можно работать как с векторами: складывать, умножать вектор на число и т.п.:

```
>>> s7 = pd.Series([10, 20, 30, 40, 50], ['a', 'b', 'c', 'd', 'e'])
```

При сложении структур, их элементы складываются между собой:

```
>>> s6 + s7
a    11
b    22
c    33
d    44
e    55
dtype: int64
```

При умножении структуры на число, все элементы структуры умножаются на данный множитель:

```
>>> s6 * 3
a     3
b     6
c     9
d    12
e    15
dtype: int64
```

1.2 Структура данных *DataFrame*

Если *Series* представляет собой одномерную структуру, которую для себя можно представить, как таблицу с одной строкой, то *DataFrame* - это уже двумерная структура - полноценная таблица с множеством строк и столбцов.

Конструктор класса *DataFrame* выглядит так:

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

Параметры конструктора:

- `data`: `ndarray`, `dict` или *DataFrame*; значение по умолчанию: `None`
  - Данные, на базе которых будет создан *DataFrame*.
- `index`*:* одномерный массив; значение по умолчанию: `None`
  - Список меток для записей (имена строк таблицы).
- `columns`*:* одномерный массив; значение по умолчанию: `None`
  - Список меток для полей (имена столбцов таблицы).
- `dtype`*:* `numpy.dtype`; значение по умолчанию: `None`
  - Объект, определяющий тип данных.

- `copy`*:* `bool`; значение по умолчанию: `False`
  - Если параметр равен `True`, то будет создана копия массива данных.

Структуру *DataFrame* можно создать на базе следующих типов данных:

- словарь (`dict`), в качестве элементов которого могут выступать: одномерные `ndarray`, списки, другие словари, структуры *Series*;
- двумерный `ndarray`;
- структура *Series*;
- другой *DataFrame*.

Рассмотрим на практике различные подходы к созданию *DataFrame'ов*.

1.2.1 Создание *DataFrame* из словаря

Для создания *DataFrame* будем использовать словарь, элементами которого могут быть списки, структуры *Series* и т.д. Начнем со варианта, когда элементы — это структуры *Series:*

```
>>> d = {'price':pd.Series([1, 2, 3], index=['v1', 'v2', 'v3']),'count':
pd.Series([10, 12, 7], index=['v1', 'v2', 'v3'])}
>>> df1 = pd.DataFrame(d)
>>> print(df1)
    price  count
v1      1     10
v2      2     12
v3      3      7
```

Индексы созданного *DataFrame*:

```
>>> print(df1.index)
Index(['v1', 'v2', 'v3'], dtype='object')
```

Столбцы созданного *DataFrame*:

```
>>> print(df1.columns)
Index(['price', 'count'], dtype='object')
```

Построим аналогичный словарь, но на элементах ndarray*:*

```
>>> d2 = {'price':np.array([1, 2, 3]), 'count': np.array([10, 12, 7])}
>>> df2 = pd.DataFrame(d2, index=['v1', 'v2', 'v3'])
>>> print(df2)
    price  count
v1      1     10
v2      2     12
v3      3      7

>>> print(df2.index)
Index(['v1', 'v2', 'v3'], dtype='object')

>>> print(df2.columns)
Index(['price', 'count'], dtype='object')
```

Как видно - результат аналогичен предыдущему. Вместо ndarray можно использовать обычный список *Python*.

1.2.2 Создание *DataFrame* из списка словарей

До этого мы создавали *DataFrame* из словаря, элементами которого были структуры *Series*, списки и массивы, сейчас мы создадим *DataFrame* из списка, элементами которого являются словари:

```
>>> d3 = [{'price': 3, 'count':8}, {'price': 4, 'count': 11}]
>>> df3 = pd.DataFrame(d3)
>>> print(df3)
   count  price
0      8      3
1     11      4
```

Для получения сводной информации по созданному *DataFrame* можно использовать функцию `info()`. Она выводит данные о типе структуры, количестве записей, количестве `non-null` элементов в столбцах, типы и количество хранимых элементов и объем используемой памяти:

```
>>> print(df3.info())
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2 entries, 0 to 1
Data columns (total 2 columns):
count    2 non-null int64
price    2 non-null int64
dtypes: int64(2)
memory usage: 112.0 bytes
None
```

1.2.3 Создание *DataFrame* из двумерного массива

Создать *DataFrame* можно также и из двумерного массива, в нашем примере это будет `ndarray` из библиотеки `numpy`:

```
>>> nda1 = np.array([[1, 2, 3], [10, 20, 30]])
>>> df4 = pd.DataFrame(nda1)
>>> print(df4)
    0   1   2
0   1   2   3
1  10  20  30
```

1.2.4 Работа с элементами *DataFrame*

Работа с элементами *DataFrame* - тема достаточно обширная, и она будет раскрыта в главе 3. Сейчас рассмотрим наиболее часто используемые подходы для доступа к элементам структуры. Основные способы представлены в таблице 1.1.

Таблица 1.1 - Работа с элементами *DataFrame*

Операция	**Синтаксис**	**Возвращаемый результат**
Выбор столбца	`df[col]`	*Series*
Выбор строки по метке	`df.loc[label]`	*Series*
Выбор строки по индексу	`df.iloc[loc]`	*Series*
Срез по строкам	`df[0:4]`	*DataFrame*
Выбор строк, отвечающих условию	`df[bool_vec]`	*DataFrame*

Рассмотрим работу с данными операциями на практике. Для начала создадим *DataFrame:*

```
>>> d = {'price':np.array([1, 2, 3]), 'count': np.array([10, 20, 30])}
>>> df = pd.DataFrame(d, index=['a', 'b', 'c'])
>>> print(df)
   price  count
a      1     10
b      2     20
c      3     30
```

Операция: выбор столбца:

```
>>> df['count']
a    10
b    20
c    30
Name: count, dtype: int32
```

Операция: выбор строки по метке:

```
>>> df.loc['a']
price     1
count    10
Name: a, dtype: int32
```

Операция: выбор строки по индексу:

```
>>> df.iloc[1]
price     2
count    20
Name: b, dtype: int32
```

Операция: срез по строкам:

```
>>> df[0:2]
   price  count
a      1     10
b      2     20
```

Операция: выбор строк, отвечающих условию:

```
>>> df[df['count'] >= 20]
   price  count
b      2     20
c      3     30
```

Глава 2. Доступ к данным в структурах *pandas*

2.1 Два подхода получения доступа к данным в *pandas*

При работе со структурами *Series* и *DataFrame*, как правило, используют два основных способа получения значений элементов.

Первый способ основан на использовании меток, в этом случае работа ведется через метод `.loc`. Если вы обращаетесь к отсутствующей метке, то будет сгенерировано исключение `KeyError`. Такой подход позволяет использовать:

- метки в виде отдельных символов `['a']` или чисел `[5]`, числа используются в качестве меток, если при создании структуры не был указан список с метками;
- список меток `['a', 'b', 'c']`;
- срез меток `['a':'c']`;
- массив логических переменных;
- *callable*-функция с одним аргументом.

Второй способ основан на использовании целых чисел для доступа к данных, он предоставляется через метод `.iloc`. Если вы обращаетесь к несуществующему элементу, то будет сгенерировано исключение `IndexError`. Логика использования `.iloc` очень похожа на работу с `.loc`. При таком подходе можно использовать:

- отдельные целые числа для доступа к элементам структуры;
- массивы целых чисел `[0, 1, 2]`;
- срезы целых чисел `[1:4]`;
- массивы логических переменных;
- *callable*-функцию с одним аргументом.

В зависимости от типа структуры, будет меняться форма `.loc`:

- для *Series*, она выглядит так: `s.loc[indexer]`;
- для *DataFrame* так: `df.loc[row_indexer, column_indexer]`.

Создадим объекты типов *Series* и *DataFrame*, которые в дальнейшем будут использованы нами для экспериментов.

Структура *Series*:

```
>>> s = pd.Series([10, 20, 30, 40, 50], ['a', 'b', 'c', 'd', 'e'])
>>> s['a']
10
>>> s
a    10
b    20
c    30
d    40
e    50
dtype: int64
```

Структура *DataFrame*:

```
>>> d = {'price':[1, 2, 3], 'count': [10, 20, 30], 'percent': [24, 51,
71]}
>>> df = pd.DataFrame(d, index=['a', 'b', 'c'])
>>> df
   price  count  percent
a      1     10       24
b      2     20       51
c      3     30       71
```

2.2 Доступ к данным структуры *Series*

2.2.1 Доступ с использованием меток

При использовании меток для доступа к данным можно применять один из следующих подходов:

- первый, когда вы записываете имя переменной типа *Series* и в квадратных скобках указываете метку, по которой хотите обратиться (пример: `s['a']`);
- второй, когда после имени переменной пишите `.loc` и далее указываете метку в квадратных скобках (пример: `s.loc['a']`).

Обращение по отдельной метке.

Элемент с меткой `'a'`:

```
>>> s['a']
10
```

Обращение по массиву меток.

Элементы с метками `'a'`, `'c'` и `'e'`:

```
>>> s[['a', 'c', 'e']]
a    10
c    30
e    50
dtype: int64
```

Обращение по срезу меток.

Элементы структуры с метками от `'a'` до `'e'`:

```
>>> s['a':'c']
a    10
b    20
c    30
dtype: int64
```

2.2.2 Доступ с использованием целочисленных индексов

При работе с целочисленными индексами, индекс можно ставить сразу после имени переменной в квадратных скобках (пример: `s[1]`) или воспользоваться `.iloc` (пример: `s.iloc[1]`).

Обращение по отдельному индексу.

Элемент с индексом 1:

```
>>> s[1]
20
```

Обращение с использованием списка индексов.

Элементы с индексами 1, 2 и 3:

```
>>> s[[1, 2 ,3]]
b    20
c    30
d    40
```

Обращение по срезу индексов.

Получение первых трех элементов структуры:

```
>>> s[:3]
a    10
b    20
c    30
dtype: int64
```

2.2.3 Доступ с использованием *callable*-функции

При таком подходе в квадратных скобках указывается не индекс или метка, а функция (как правило, это *lambda*-функция), которая используется для выборки элементов структуры.

Элементы, значение которых больше либо равно 30:

```
>>> s[lambda x: x>= 30]
c    30
d    40
e    50
dtype: int64
```

2.2.4 Доступ с использованием логического выражения

Данный подход похож на работу с *callable*-функцией: в квадратных скобках записывается логическое выражение, согласно которому будет произведен отбор.

Элементы, значение которых больше 30:

```
>>> s[s > 30]
d    40
e    50
dtype: int64
```

2.3 Доступ к данным структуры *DataFrame*

2.3.1 Доступ с использованием меток

Рассмотрим различные варианты использования меток, которые могут быть как именами столбцов таблицы, так и именами строк.

Обращение к конкретному столбцу.

Элементы столбца `'count'`:

```
>>> df['count']
a    10
b    20
c    30
Name: count, dtype: int64
```

Обращение с использованием массива столбцов.

Элементы столбцов `'count'` и `'price'`:

```
>>> df[['count','price']]
   count  price
a     10      1
b     20      2
c     30      3
```

Обращение по срезу меток.

Элементы с метками от `'a'` до `'b'`:

```
>>> df['a':'b']
   price  count  percent
a      1     10       24
b      2     20       51
```

2.3.2 Доступ с использованием *callable*-функции

Подход в работе с *callable*-функцией для *DataFrame* аналогичен тому, что используется для *Series*, только при формировании условий необходимо дополнительно указывать имя столбца.

Получим все элементы, у которых значение в столбце `'count'` больше 15:

```
>>> df[lambda x: x['count'] > 15]
   price  count  percent
b      2     20       51
c      3     30       71
```

2.3.3 Доступ с использованием логического выражения

При формировании логического выражения необходимо указывать имена столбцов, также как и при работе с *callable*-функциями, по которым будет производиться выборка.

Получим все элементы, у которых `'price'` больше либо равен 2:

```
>>> df[df['price'] >= 2]
   price  count  percent
b      2     20       51
c      3     30       71
```

2.4 Использование атрибутов для доступа к данным

Для доступа к данным можно использовать атрибуты структур, в качестве которых выступают метки. Начнем со структуры *Series*. Воспользуемся уже знакомой нам структурой:

```
>>> s = pd.Series([10, 20, 30, 40, 50], ['a', 'b', 'c', 'd', 'e'])
```

Для доступа к элементу через атрибут необходимо указать его через точку после имени переменной:

```
>>> s.a
10
>>> s.c
30
```

Т.к. структура *s* имеет метки *'a'*, `'b'`, `'c'`, `'d'`, `'e'`, то для доступа к элементу с меткой `'a'` используется синтаксис `s.a`.

Этот же подход можно применить для переменной типа *DataFrame*:

```
>>> d = {'price':[1, 2, 3], 'count': [10, 20, 30], 'percent': [24, 51,
71]}
>>> df = pd.DataFrame(d, index=['a', 'b', 'c'])
```

Получим доступ к столбцу 'price':

```
>>> df.price
a    1
b    2
c    3
Name: price, dtype: int64
```

2.5 Получение случайного набора из структур *pandas*

Библиотека *pandas* предоставляет возможность получить случайный набор данных из уже существующей структуры. Такой функционал предоставляет как *Series,* так и *DataFrame*. Случайная подвыборка, извлекается с помощью метода sample().

Для начала разберем работу с этим методом на примере структуры *Series*.

Для выбора случайного элемента из *Series* используется следующий синтаксис:

```
>>> s.sample()
a    10
dtype: int64
```

Можно сделать выборку из нескольких элементов, для этого нужно передать количество элементов через параметр n:

```
>>> s.sample(n=3)
c    30
a    10
d    40
dtype: int64
```

Есть возможность указать долю от общего числа объектов в структуре, за это отвечает параметр `frac`:

```
>>> s.sample(frac=0.3)
d    40
e    50
dtype: int64
```

Дополнительно, в качестве аргумента, мы можем передать вектор весов, длина которого должна быть равна количеству элементов в структуре, а сумма элементов вектора - единице. Вес, в данном случае, это вероятность появления элемента в выборке.

В нашей тестовой структуре пять элементов, сформируем вектор весов для нее и сделаем выборку из трех элементов:

```
>>> w = [0.1, 0.2, 0.5, 0.1, 0.1]
>>> s.sample(n = 3, weights=w)
c    30
b    20
a    10
dtype: int64
```

Возможности метода `sample()` доступны и для структуры *DataFrame*:

```
>>> d = {'price':[1, 2, 3, 5, 6], 'count': [10, 20, 30, 40, 50],
'percent': [24, 51, 71, 25, 42]}
>>> df = pd.DataFrame(d)
>>> df.sample()
   price  count  percent
2      3     30       71
```

При работе с *DataFrame* можно указать ось. Для выбора столбца случайным образом задайте параметру `axis` значение равное 1:

```
>>> df.sample(axis=1)
   count
0     10
1     20
2     30
3     40
4     50
```

Выбор двух столбцов случайным образом:

```
>>> df.sample(n=2, axis=1)
   percent  count
0       24     10
1       51     20
2       71     30
3       25     40
4       42     50
```

Выбор двух строк случайным образом:

```
>>> df.sample(n=2)
   price  count  percent
0      1     10       24
1      2     20       51
```

2.6 Индексация с использованием логических выражений

На практике часто приходится делать подвыборку из существующего набора данных. Например: получить все товары, скидка на которые больше пяти процентов, или выбрать из базы информацию о сотрудниках мужского пола старше 30 лет. Это очень похоже на процесс фильтрации при работе с таблицами или получение выборки из базы

данных. Похожий функционал реализован в *pandas*, и мы уже касались этого вопроса, когда рассматривали различные подходы к индексации: условное выражение должно быть записано вместо индекса в квадратных скобках при обращении к элементам структуры.

При работе с *Series* возможны следующие варианты использования:

```
>>> s = pd.Series([10, 20, 30, 40, 50, 10, 10], ['a', 'b', 'c', 'd', 'e',
'f', 'g'])
>>> s[s>30]
d    40
e    50
dtype: int64
```

Получим все элементы структуры, значение которых равно 10:

```
>>> s[s==10]
a    10
f    10
g    10
dtype: int64
```

Элементы структуры *s*, значения которых находятся в интервале [30, 50):

```
>>> s[(s>=30) & (s<50)]
c    30
d    40
dtype: int64
```

При работе с *DataFrame* необходимо указывать столбец, по-которому будет производиться фильтрация (выборка):

```
>>> d = {'price':[1, 2, 3, 5, 6], 'count': [10, 20, 30, 40, 50],
'percent': [24, 51, 71, 25, 42], 'cat': ['A', 'B', 'A', 'A', 'C']}
>>> df = pd.DataFrame(d)
```

```
>>> df
   price  count  percent cat
0      1     10       24   A
1      2     20       51   B
2      3     30       71   A
3      5     40       25   A
4      6     50       42   C
```

Выделим список строк таблицы, у которых значение в поле `'price'` больше, чем 3:

```
>>> df[df['price'] > 3]
   price  count  percent cat
3      5     40       25   A
4      6     50       42   C
```

В качестве логического выражения можно использовать довольно сложные конструкции с применением функций `map`, `filter`, *lambda*-выражений и т.п.

Получим список индексов структуры `df`, у которых значение поля `'cat'` равно `'A'`:

```
>>> fn = df['cat'].map(lambda x: x == 'A')
>>> df[fn]
   price  count  percent cat
0      1     10       24   A
2      3     30       71   A
3      5     40       25   A
```

2.7 Использование `isin` при работы с данными в *pandas*

По структурам данных *pandas* можно строить массивы с элементами типа `bool`, по которому можно проверить наличие или отсутствие того или иного элемента в исходной структуре.

Будем работать со следующей структурой:

```
>>> s = pd.Series([10, 20, 30, 40, 50, 10, 10], ['a', 'b', 'c', 'd', 'e',
'f', 'g'])
```

Построим новую структуру с элементами типа `bool`, такую, что если значение элемента исходной структуры находится в списке [10, 20], то значение элемента равно `True`, в противном случае – `False`:

```
>>> s.isin([10, 20])
a     True
b     True
c    False
d    False
e    False
f     True
g     True
dtype: bool
```

Работа с *DataFrame* аналогична работе со структурой *Series*:

```
>>> df = pd.DataFrame({'price':[1, 2, 3, 5, 6], 'count': [10, 20, 30, 40,
50], 'percent': [24, 51, 71, 25, 42]})
```

Если значение элемента исходной структуры находится в списке [1, 3, 25, 30, 10], то значение элемента в новой структуре равно `True`, иначе – `False`:

```
>>> df.isin([1, 3, 25, 30, 10])
   price  count  percent
0   True   True    False
1  False  False    False
2   True   True    False
3  False  False     True
4  False  False    False
```

Глава 3. Типы данных в *pandas*

При изучении любого языка программирования одной из важнейших тем, которую нужно освоить, является система типов. Например, при работе с языком *Python* вы сталкивались с такими типами как `int` и `float` для работы с числами, `str` - для работы со строками; есть типы более сложные, такие как списки, словари, множества. Библиотека *pandas* содержит свой набор типов данных для эффективного хранения и манипулирования данными. В предыдущих главах вы уже сталкивались с таким понятием как `dtype`: например, при выводе содержимого *Series* с помощью функции `print`, когда тип хранящихся значений отображается в последней строке:

```
>>> s = pd.Series([1,2,3])
>>> print(s)
0 1
1 2
2 3
dtype: int64
```

В приведенном примере тип хранимых в `s` значений: `int64`.

Или при выводе подробной информации о структуре *DataFrame* с помощью метода `info()`: помимо названий столбцов, информации о количестве элементов, занимаемой памяти и т.п., выводится тип хранимых данных по каждому столбцу:

```
>>> d = [{'name': 'pen', 'price': 3.9, 'count': 8}, {'name': 'book',
'price': 4.5, 'count': 11}]
>>> df = pd.DataFrame(d)
```

```
>>> df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2 entries, 0 to 1
Data columns (total 3 columns):
count 2 non-null int64
name  2 non-null object
price 2 non-null float64
dtypes: float64(1), int64(1), object(1)
memory usage: 128.0+ bytes
```

Из приведенной информации видно, что в поле `count` хранятся элементы с типом `int64`, в поле `name` - с типом `object`, в поле `price` - с типом `float64`. В этой главе мы более подробно изучим типы библиотеки *pandas* и научимся с ними работать.

3.1 Типы данных

Основные типы данных, которые используются в *pandas*, приведены в таблице 3.1.

Таблица 3.1 - Типы данных в *pandas*

Тип данных	Описание
`int64`	64 разрядное целочисленное значение, не зависит от платформы
`float64`	64 разрядное число с плавающей точкой, не зависит от платформы
`object`	Текст или любое другое значение
`bool`	Булевое значение: `True` / `False`
`category`	Конечное множество текстовых элементов
`datetime64[ns]`	Дата / Время
`timedelta64[ns]`	Разница между двумя `datetime` элементами

Как правило, вам не нужно явно задавать тип значений, *pandas* может самостоятельно вывести тип, но если необходимо его указать или сделать преобразование типа, то стоит воспользоваться возможностями, которые библиотека предоставляет.

Информацию об используемых в структуре типах можно получить через атрибут `dtypes`. Для структуры `df`, из примера выше, мы получим следующий результат:

```
>>> df.dtypes
count   int64
name    object
price   float64
dtype: object
```

Атрибут `dtype` есть у структур *DataFrame* и *Series*.

В рамках *pandas* также можно создавать и использовать типы расширений. В комплекте с библиотекой идут типы расширений, представленные в таблице 3.2.

Таблица 3.2 - Типы расширений *pandas*

Тип	**Описание**
`DatetimeTZDtype`	`datetime` с поддержкой часового пояса
`CategoricalDtype`	Тип для категориальных данных (конечное множество текстовых элементов)
`PeriodDtype`	Тип для работы с периодическими данными
`SparseDtype`	Тип для работы с разреженными данными
`IntervalDtype`	Тип для работы с интервальными данными

3.2 Инструменты для работы с типами

Pandas предоставляет ряд инструментов для работы с типами, под “работой с типами” мы будет понимать различные операции приведения типов.

3.2.1 astype()

Первый инструмент, который мы рассмотрим — это метод `astype()`. Он доступен как для структур *Series*, так и для *DataFrame*:

```
astype(self, dtype, copy=True, errors='raise', **kwargs)
```

- `dtype`: тип данных или словарь с именами столбцов
  - В качестве типа данных для приведения могут быть использованы `numpy.dtype`, тип *pandas*, либо словарь в формате `{col: dtype, …}`, где `col` - имя столбца, `dtype` - желаемый тип данных.
- `copy`: `bool`; значение по умолчанию: `True`
  - `astype` будет возвращать копию структуры, если параметр равен `True`, в ином случае будет модифицироваться текущая структура.
- `errors: {'raise', 'ignore'}`; значение по умолчанию: `'raise'`
  - Управляет процессом выброса исключений:
    - `raise` : разрешает выброс исключений;
    - `ignore` : игнорирует исключения.
- `**kwargs`
  - Аргументы для передачи конструктору.

Преобразование типов для структуры *Series*

Создадим структуру с целыми числами:

```
>>> s = pd.Series([1,2,3])
>>> s.dtype
dtype('int64')
```

Приведем эти данные к типу `float64`:

```
>>> s.astype('float64')
0   1.0
1   2.0
2   3.0
dtype: float64
```

Преобразование типов структуры *DataFrame*

Создадим структуру:

```
>>> d = [{'name': 'pen', 'price': 3.9, 'count': 8}, {'name': 'book',
'price': 4.5, 'count': 11}]
>>> df = pd.DataFrame(d)
>>> df.dtypes
count    int64
name    object
price    float64
dtype: object
```

Приведем тип поля `'count'` к `int32`:

```
>>> df['count'] = df['count'].astype('int32')
>>> df.dtypes
count    int32
name    object
price    float64
dtype: object
```

Вернем прежний тип полю `'count'`, при этом используем другой способ вызова функции `astype()`:

```
>>> df = df.astype({'count':'int64'})
>>> df.dtypes
count    int64
name    object
price    float64
dtype: object
```

3.2.2 Функции подготовки данных

Часто возникает задача предварительной подготовки данных перед выполнением операции преобразования типа. Набор данных для экспериментов представлен в таблице 3.3.

Таблица 3.3 — Набор данных для экспериментов

Температура	Давление	Осадки	Дата
−8 °C	96292 Па	Да	2019-11-20
−10.3 °C	97292 Па	Да	2019-11-21
−9.1 °C	96325 Па	Нет	2019-11-22

Загрузим эту таблицу в *DataFrame*:

```
>>> df = pd.read_csv('c:/data.csv', sep=',')
>>> df
    Температура Давление    Осадки  Дата
0   -8 °C       96292 Па    Да      2019-11-20
1   -10.3 °C    97292 Па    Да      2019-11-21
2   -9.1 °C     96325 Па    Нет     2019-11-22
```

Посмотрим на типы:

```
>>> df.dtypes
Температура object
Давление    object
Осадки      object
Дата        object
dtype: object
```

Как вы можете видеть типы всех полей - `object`, а нам нужно, чтобы в данные в первом столбце (Температура) были в формате `float64`, во втором - `int64`, в третьем - `bool`. Если мы попытаемся напрямую выполнить приведение с помощью функции `astype()`, то эта операция завершиться неудачно, так как, например, для температуры, вместе с

интересующим нас численным значением находится единица измерения. Для решения этой задачи напишем несколько функций преобразования, которые помогут привести данные к нужному виду:

```
>>> temper_convertor = lambda x: x.replace('°C', '').strip()
>>> pressure_convertor = lambda x: x.replace('Па', '').strip()
>>> prec_convertor = lambda x: True if x == 'Да' else False
```

Применим созданные функции к элементам структуры:

```
>>> df['Температура'] =
df['Температура'].apply(temper_convertor).astype('float64')
>>> df['Давление'] =
df['Давление'].apply(pressure_convertor).astype('int64')
>>> df['Осадки'] = df['Осадки'].apply(prec_convertor)
>>> df.dtypes
Температура    float64
Давление       int64
Осадки         bool
Дата           object
dtype: object
```

3.2.3 Вспомогательные функции

Pandas предоставляет вспомогательные функции для преобразования данных, которые избавляют от необходимости разрабатывать соответствующие решения самим, таких функций три:

`to_numeric()` - преобразует данные в числовой тип;

`to_datetime()` - преобразует данные в тип `datetime`;

`to_timedelta()` - преобразует данные в тип `timedelta`.

Загрузим заново подготовленный набор данных:

```
>>> df = pd.read_csv('c:/data.csv', sep=',')
>>> df.dtypes
Температура    object
Давление       object
Осадки         object
Дата           object
dtype: object
```

Воспользуемся приведенными выше функциями для преобразования данных:

```
>>> df['Температура'] = pd.to_numeric(df['Температура'],
errors='coerce')
>>> df['Давление'] = pd.to_numeric(df['Давление'],
errors='coerce')
>>> df['Дата'] = pd.to_datetime(df['Дата'], errors='coerce')

>>> df.dtypes
Температура    float64
Давление       int64
Осадки         object
Дата           datetime64[ns]
dtype:    object
```

3.2.4 Выборка данных по типу

Для выборки данных по типу используется функция `select_dtypes()`, она возвращает *DataFrame*, построенный из исходного *DataFrame'*а, в который будут входить столбцы с типами, указанными в аргументе `include`, и не будут входит столбцы, типы которых, перечислены в `exclude` аргументе:

```
>>> df.select_dtypes(include=['float64', 'int64'])
     Температура    Давление
0    -8.0           96292
1    -10.3          97292
2    -9.1           96325
```

```
>>> df.select_dtypes(exclude='datetime64[ns]')
     Температура    Давление  Осадки
0    -8.0           96292     Да
1    -10.3          97292     Да
2    -9.1           96325     Нет
```

3.3 Категориальные типы

Категориальные типы данных в *pandas* похожи по своей сути на качественные признаки в статистике. Они задаются в виде конечного набора строковых переменных. В качестве примеров можно привести следующие:

- цвет: *Red*, *Green*, *Blue*;
- начертание шрифта: *Normal*, *Bold*, *Italic*;
- выравнивание текста: *Right*, *Center*, *Left*.

3.3.1 Создание структуры с набором категориальных данных

Рассмотрим варианты создания структуры с элементами категориального типа.

Работа со структурой *Series*

Если мы просто создадим структуру *Series*, без указания типа, то получим набор элементов с типом `object`:

```
>>> s = pd.Series(['r', 'r', 'g', 'b'])
>>> s
0    r
1    r
2    g
3    b
dtype: object
```

Если необходимо явно указать, что элементы относятся к категориальному типу, то это нужно сделать через аргумент dtype:

```
>>> s = pd.Series(['r', 'r', 'g', 'b'], dtype='category')
>>> s
0    r
1    r
2    g
3    b
dtype: category
Categories (3, object): [b, g, r]
```

Работа с типом *Categorical*

Если заранее известна структура категории: набор ее элементов и порядок, то можно создать объект класса *Categorical*:

```
pandas.Categorical(values, categories=None, ordered=None, dtype=None,
fastpath=False)
```

- `values` : список
  - Элементы данных. Если дополнительно указывается категория через параметр `categories`, то значения не из категории заменяются на `NaN`.
- `categories` : набор уникальных элементов, `None`; значение по умолчанию: `None`
  - Задает набор значений, которые может принимать элемент категории. Если равен `None`, то категория строится по набору уникальных элементов из параметра `values`.
- `ordered` : `bool`; значение по умолчанию: `False`
  - Определяет, является категория порядковой или нет. Если значение равно `True`, то категория является порядковой.
- `dtype` : `CategoricalDtype`
  - Тип `CategoricalDtype`, который используется для категории.

Создадим категорию:

```
>>> colors = pd.Categorical(['r', 'g', 'g', 'b', 'r'])
>>> colors
[r, g, g, b, r]
Categories (3, object): [b, g, r]
```

В этом примере категория была построена по элементам переданных данных, также как и в примере с *Series*. Укажем явно категорию:

```
>>> colors = pd.Categorical(['r', 'g', 'g', 'b', 'r'], categories=['r',
'g', 'b'])
>>> colors
[r, g, g, b, r]
Categories (3, object): [r, g, b]
```

Если в наборе данных будут присутствовать элементы, которые не входят в категорию, им будут присвоены значения NaN:

```
>>> colors = pd.Categorical(['r', 'g', 'g', 'b', 'r', 'y', 'o'],
categories=['r', 'g', 'b'])
>>> colors
[r, g, g, b, r, NaN, NaN]
Categories (3, object): [r, g, b]
```

Из объекта *Categorical* можно построить структуру *Series*:

```
>>> colors_s = pd.Series(colors)
>>> colors_s
0    r
1    g
2    g
3    b
4    r
5    NaN
6    NaN
dtype: category
Categories (3, object): [r, g, b]
```

Для очистки данных используется функция dropna():

```
>>> colors_s = pd.Series(colors).dropna()
>>> colors_s
0    r
1    g
2    g
3    b
4    r
dtype: category
Categories (3, object): [r, g, b]
```

Работа со структурой *DataFrame*

По аналогии с *Series* можно создать *DataFrame* с категориальными данными:

```
>>> df = pd.DataFrame({'C1':list('rrg'), 'C2':list('rgb')},
dtype='category')
>>> df
     C1   C2
0    r    r
1    r    g
2    g    b

>>> df.dtypes
C1    category
C2    category
dtype: object

>>> df['C1']
0    r
1    r
2    g
Name: C1, dtype: category
Categories (2, object): [g, r]
```

3.3.2 Порядковые категории

Категориальные данные, о которых шла речь выше, не включают в себя отношение порядка, то есть для переменных, принимающих значения из таких категорий невозможно выполнить сравнение “больше-меньше”. Существуют категории, для которых отношение порядка задается, это может быть роль в фильме, образование и т.п.

Построим набор данных, с определением порядка:

```
>>> level = pd.Categorical(['h', 'h', 'm', 'l'], categories=['l',
'm', 'h'], ordered=True)
>>> level
[h, h, m, l]
Categories (3, object): [l < m < h]
```

Обратите внимание на последнюю строчку, в ней указано как соотносятся между собой значения элементов категории в отношении “больше-меньше”. Если мы не укажем параметр `ordered=True`, то для такого набора данных нельзя будет выполнить поиск минимального и максимального элемента:

```
>>> c_var = pd.Series(pd.Categorical(['r', 'g', 'g', 'b', 'r'],
categories=['r', 'g', 'b'], ordered=False))
>>> c_var
0    r
1    g
2    g
3    b
4    r
dtype: category
Categories (3, object): [r, g, b]
```

```
>>> c_var.min()
Traceback (most recent call last): ...
'Categorical to an ordered one\n'.format(op=op))
TypeError: Categorical is not ordered for operation min
you can use .as_ordered() to change the Categorical to an ordered
one
```

Для созданного набора `level`, такую операцию сделать можно:

```
>>> lev_var = pd.Series(level)
>>> print('min: {}, max: {}'.format(lev_var.min(), lev_var.max()))
'min: l, max: h'
```

Для непорядковых категорий запрещены сравнения на уровне объектов, например:

```
>>> c1 = pd.Series(pd.Categorical(['r', 'g', 'b', 'r'], categories=['r',
'g', 'b'], ordered=False))
>>> c2 = pd.Series(pd.Categorical(['b', 'g', 'g', 'r'], categories=['r',
'g', 'b'], ordered=False))
>>> c1 > c2
Traceback (most recent call last):
...
     raise TypeError('Unordered Categoricals can only compare '
TypeError: Unordered Categoricals can only compare equality or not
```

Если объекты будут содержать данные с элементами типа *порядковая* категория, то все будет выполнено корректно:

```
>>> v1 = pd.Series(pd.Categorical(['h', 'm', 'l', 'h'], categories=['l',
'm', 'h'], ordered=True))
>>> v2 = pd.Series(pd.Categorical(['m', 'm', 'h', 'l'], categories=['l',
'm', 'h'], ordered=True))
>>> v1 > v2
0     True
1    False
2    False
3     True
dtype: bool
```

Глава 4. Работа с пропусками в данных

Часто, в больших объемах данных, которые подготавливаются для анализа, имеются пропуски. Для того, чтобы можно было использовать алгоритмы машинного обучения, строящие модели по этим данным, необходимо эти пропуски заполнить. На вопрос “чем заполнять?” мы не будем отвечать в рамках данной книги, т.к. он относится больше к теме машинного обучения и анализа данных. А вот на вопрос “как заполнять?” мы ответим, и решать эту задачу будем средствами библиотеки *pandas*, которая предоставляет инструменты, позволяющие это сделать.

4.1 *Pandas* и отсутствующие данные

Для наших экспериментов создадим структуру *DataFrame*, которая будет содержать пропуски. Для этого импортируем необходимые нам библиотеки:

```
>>> import pandas as pd
>>> from io import StringIO
```

После этого создадим объект в формате *csv*. *CSV* - это один из наиболее простых и распространенных форматов хранения данных, в котором элементы отделяются друг от друга запятыми:

```
>>> data = 'price,count,percent\n1,10,\n2,20,51\n3,30,'
>>> df = pd.read_csv(StringIO(data))
```

Полученный объект `df` - это *DataFrame* с пропусками:

```
>>> df
   price  count  percent
0      1     10      NaN
1      2     20     51.0
2      3     30      NaN
```

В нашем примере, у объектов с индексами 0 и 2 отсутствуют данные в поле `'percent'`. Отсутствующие данные помечаются как `NaN`.

Добавим к существующей структуре еще один объект (запись), у которого будет отсутствовать значение в поле `'count'`:

```
>>> df.loc[3] = {'price':4, 'count':None, 'percent':26.3}
>>> df
   price  count  percent
0    1.0   10.0      NaN
1    2.0   20.0     51.0
2    3.0   30.0      NaN
3    4.0    NaN     26.3
```

Для начала обратимся к методам из библиотеки *pandas*, которые позволяют быстро определить наличие элементов `NaN` в структурах.

Если таблица небольшая, то можно использовать библиотечный метод `isnull()`:

```
>>> pd.isnull(df)
   price  count  percent
0  False  False     True
1  False  False    False
2  False  False     True
3  False   True    False
```

В результате мы получаем таблицу того же размера, но на месте реальных данных, в ней находятся элементы типа `bool`, которые равны `False`, если значение поля не-`NaN`, либо `True` в противном случае.

В дополнение к этому, можно посмотреть подробную информацию об объекте, с помощью метода `info()`:

```
>>> df.info()
<class 'pandas.core.frame.DataFrame'>
Int64Index: 4 entries, 0 to 3
Data columns (total 3 columns):
price      4 non-null float64
count      3 non-null float64
percent    2 non-null float64
dtypes: float64(3)
memory usage: 128.0 bytes
```

Из примера видно, что объект `df` имеет три столбца (`count`, `percent` и `price`), при этом в столбце `price` все объекты значимы - не `NaN`, в столбце `count` - один `NaN` объект, в поле `percent` - два `NaN` объекта.

Можно воспользоваться следующим подходом для определения количества `NaN` элементов в записях:

```
>>> df.isnull().sum()
price      0
count      1
percent    2
dtype: int64
```

4.2 Замена отсутствующих данных

Отсутствующие данные объектов можно заменить на конкретные числовые значения с помощью метода `fillna()`.

Для экспериментов будем использовать структуру df, созданную в предыдущем разделе:

```
>>> df
   price  count  percent
0    1.0   10.0      NaN
1    2.0   20.0     51.0
2    3.0   30.0      NaN
3    4.0    NaN     26.3

>>> df.fillna(0)
   price  count  percent
0    1.0   10.0      0.0
1    2.0   20.0     51.0
2    3.0   30.0      0.0
3    4.0    0.0     26.3
```

Этот метод не изменяет текущую структуру, он возвращает структуру *DataFrame*, созданную на базе существующей, с заменой NaN значений на те, что переданы в метод в качестве аргумента. Данные можно заполнить средним значением по столбцу:

```
>>> df.fillna(df.mean())
   price  count  percent
0    1.0   10.0    38.65
1    2.0   20.0    51.00
2    3.0   30.0    38.65
3    4.0   20.0    26.30
```

В зависимости от задачи, используется тот или иной метод заполнения отсутствующих элементов, это может быть нулевое значение, математическое ожидание, медиана и т.п.

Для замены NaN элементов на конкретные значения, можно использовать интерполяцию, которая реализована в методе interpolate(), алгоритм интерполяции задается через аргумент метода.

4.3 Удаление объектов/столбцов с отсутствующими данными

Довольно часто используемый подход при работе с отсутствующими данными - это удаление записей (строк) или полей (столбцов), в которых встречаются пропуски.

Для того, чтобы удалить все объекты, которые содержат значения NaN воспользуйтесь методом dropna() без аргументов:

```
>>> df.dropna()
   price  count  percent
1    2.0   20.0     51.0
```

Вместо записей, можно удалить поля, для этого нужно вызвать метод dropna() с аргументом axis=1:

```
>>> df.dropna(axis=1)
   price
0    1.0
1    2.0
2    3.0
3    4.0
```

Pandas позволяет задать порог на количество не-NaN элементов. В приведенном ниже примере будут удалены все столбцы в которых количество не-NaN элементов меньше трех:

```
>>> df.dropna(axis = 1, thresh=3)
   price  count
0    1.0   10.0
1    2.0   20.0
2    3.0   30.0
3    4.0    NaN
```

Глава 5. Работа со структурами данных в *pandas*: удаление, объединение, расширение, группировка

5.1 Добавление элементов в структуру *pandas*

Начнем наш обзор с операции добавления элементов в структуру. Все действия будут рассмотрены для двух структур: *Series* и *DataFrame*. Для начала создадим структуры, с которыми мы будем работать.

Структура *Series*:

```
>>> s = pd.Series([1, 2, 3, 4, 5], ['A', 'B', 'C', 'D', 'E'])
```

Словарь для *DataFrame*:

```
>>> d = {'color':['red', 'green', 'blue'], 'speed': [56, 24, 65],
'volume': [80, 65, 50]}
```

Структура *DataFrame*:

```
>>> df = pd.DataFrame(d)
>>> df
   color  speed  volume
0    red     56      80
1  green     24      65
2   blue     65      50
```

5.1.1 Добавление в *Series*

Добавить новый элемент в структуру *Series* очень просто, достаточно указать новый индекс для объекта и задать значение элементу.

Добавим в структуру `s` новый элемент с индексом `'F'` и значением 6:

```
>>> s['F']=6
>>> s
A    1
B    2
C    3
D    4
E    5
F    6
dtype: int64
```

5.1.2 Добавление в *DataFrame*

При работе с *DataFrame* в таблицу можно добавить как дополнительный столбец, так и целую запись. Начнем со столбца, данная операция очень похожа на добавление элемента в *Series*: мы указываем в квадратных скобках имя столбца и присваиваем ему список значений для каждой записи:

```
>>> df
   color  speed  volume
0    red     56      80
1  green     24      65
2   blue     65      50
```

Добавим новый столбец 'type', указав для каждой записи (строки) значение данного поля:

```
>>> df['type']=['circle', 'square', 'triangle']
>>> df
   color  speed  volume      type
0    red     56      80    circle
1  green     24      65    square
2   blue     65      50  triangle
```

Можно добавить столбец с константным значением:

```
>>> df['value'] = 7
>>> df
   color  speed  volume      type  value
0    red     56      80    circle      7
1  green     24      65    square      7
2   blue     65      50  triangle      7
```

В *DataFrame* можно добавить объект *Series* как строку. Это действие относится больше к теме “объединение данных”, но в данном контексте она уместна:

```
>>> new_row = pd.Series(['yellow', 34, 10, 'rectangle', 7], ['color',
'speed', 'volume', 'type', 'value'])
>>> df.append(new_row, ignore_index=True)
    color  speed  volume       type  value
0     red     56      80     circle      7
1   green     24      65     square      7
2    blue     65      50   triangle      7
3  yellow     34      10  rectangle      7
```

5.2 Удаление элементов из структуры в *pandas*

5.2.1 Удаление из *Series*

Для удаления элементов из структуры *Series* используется метод `drop()`, которому, в качестве аргумента, передается список меток для удаления. При этом, необходимо помнить, что при использовании данного метода, по умолчанию, текущая структура не изменяется, а возвращается объект *Series*, в котором будут отсутствовать выбранные метки.

Исходное состояние структуры `s`:

```
>>> s
A    1
B    2
C    3
D    4
E    5
F    6
dtype: int64
>>> s_new = s.drop(['A', 'B'])
```

После операции `drop()`, структура `s` осталась прежней:

```
>>> s
A    1
B    2
C    3
D    4
E    5
F    6
dtype: int64
```

Структура `s_new` содержит все элементы из `s` за исключением тех, индексы которых были переданы методу `drop()`:

```
>>> s_new
C    3
D    4
E    5
F    6
dtype: int64
```

Как видно из примера, структура `s` не изменилась. Вызов метода `drop()` привел к тому, что была создана еще одна структура с именем `s_new` без указанных элементов. Если нужно изменить непосредственно саму структуру, то, дополнительно, необходимо аргументу `inplace` метода `drop()` присвоить значение `True`:

```
>>> s.drop(['A', 'B'], inplace=True)
>>> s
C    3
D    4
E    5
F    6
dtype: int64
```

5.2.2 Удаление из *DataFrame*

Для удаления элементов из структуры *DataFrame* также применяется метод `drop()`. Для демонстрации будем использовать объект `df`, созданный в предыдущем разделе:

```
>>> df
   color  speed  volume      type  value
0    red     56      80    circle      7
1  green     24      65    square      7
2   blue     65      50  triangle      7

>>> df_new = df.drop([0])
>>> df_new
   color  speed  volume      type  value
1  green     24      65    square      7
2   blue     65      50  triangle      7
```

Если необходимо модифицировать саму структуру `df`, то укажите дополнительно параметр `inplace=True`:

```
>>> df.drop([0], inplace=True)
>>> df
   color  speed  volume      type  value
1  green     24      65    square      7
2   blue     65      50  triangle      7
```

DataFrame - это двумерная таблица, из которой можно удалять не только строки, но и столбцы. Для этого необходимо указать ось, с которой мы будем работать, она задается через параметр `axis`, по умолчанию `axis=0`, что означает работу со строками. Если указать `axis=1`, то это позволит удалить ненужные столбцы:

```
>>> df
   color  speed  volume      type  value
1  green     24      65    square      7
2   blue     65      50  triangle      7
>>> df.drop(['color', 'value'], axis=1, inplace=True)
```

```
>>> df
   speed  volume      type
1     24      65    square
2     65      50  triangle
```

5.3 Объединение данных

Pandas предоставляет набор инструментов для решения задачи объединения данных. В нашем распоряжении имеется возможность просто объединять структуры в одну без дополнительной обработки, либо сделать этот процесс более интеллектуальным, задействовав представляемый *pandas* функционал.

Мы не будем рассматривать этот функционал отдельно для *Series* и *DataFrame*, все возможности будут показаны только для *DataFrame*.

5.3.1 Использование метода *concat*

Задачу объединения структур в *pandas* решает метод *concat:*

```
pandas.concat(objs, axis=0, join='outer', join_axes=None,
ignore_index=False, keys=None, levels=None, names=None,
verify_integrity=False,  copy=True)
```

Рассмотрим наиболее важные аргументы:

- `objs`: массив или словарь структур *Series*, *DataFrame* или *Panel*.
  - Структуры для объединения.
- `axis`: 0 - строки, 1 - столбцы; значение по умолчанию: 0
  - Ось, вдоль которой будет производиться объединение.
- `join`: `{'inner', 'outer'}`; значение по умолчанию: `'outer'`
  - Тип операции объединения, `'outer'` - итоговая структура будет результатом объединения (логическое ИЛИ) переданных

структур, `'inner'` - итоговая структура будет результатом пересечения (логическое И) переданных структур.

- `ignore_index`: `bool`; значение по умолчанию: `False`
  - `True` – не используется значение индекса в процессе объединения, `False` – используется.

Для начала создадим несколько структур *DataFrame:*

```
dfr1 = pd.DataFrame({'a_type':['a1', 'a2', 'a3'], 'b_type':['b1', 'b2',
'b3'], 'c_type':['c1', 'c2', 'c3']}, index=[0, 1, 2])
>>> dfr1
  a_type b_type c_type
0     a1     b1     c1
1     a2     b2     c2
2     a3     b3     c3
>>> dfr2 = pd.DataFrame({'a_type':['a4', 'a5', 'a6'], 'b_type':['b4',
'b5', 'b6'], 'c_type':['c4', 'c5', 'c6']}, index=[3, 4, 5])
>>> dfr2
  a_type b_type c_type
3     a4     b4     c4
4     a5     b5     c5
5     a6     b6     c6
```

Теперь объединим эти две структуры в одну:

```
>>> df1 = pd.concat([dfr1, dfr2])
>>> df1
  a_type b_type c_type
0     a1     b1     c1
1     a2     b2     c2
2     a3     b3     c3
3     a4     b4     c4
4     a5     b5     c5
5     a6     b6     c6
```

Создадим ещё одну структуру и объединим ее с первой изменив ось:

```
>>> dfr3 = pd.DataFrame({'d_type':['d1', 'd2', 'd3'], 'e_type':['e1',
'e2', 'e3']})
>>> dfr3
  d_type e_type
0     d1     e1
1     d2     e2
2     d3     e3

>>> df2 = pd.concat([dfr1, dfr3], axis=1)
>>> df2
  a_type b_type c_type d_type e_type
0     a1     b1     c1     d1     e1
1     a2     b2     c2     d2     e2
2     a3     b3     c3     d3     e3
```

Для выделения в итоговой структуре составляющие компоненты, используйте параметр `keys` при объединении:

```
>>> df3 = pd.concat([dfr1, dfr2], keys=['dfr1', 'dfr2'])
>>> df3
       a_type b_type c_type
dfr1 0     a1     b1     c1
     1     a2     b2     c2
     2     a3     b3     c3
dfr2 3     a4     b4     c4
     4     a5     b5     c5
     5     a6     b6     c6
```

```
>>> df3.loc['dfr2']
  a_type b_type c_type
3     a4     b4     c4
4     a5     b5     c5
5     a6     b6     c6
```

Если итоговая структура должна являться результатом объединения (логическое ИЛИ), то параметру `join` необходимо присвоить значение `'outer'`.

```
>>> dfr4 = pd.DataFrame({'d_type':['d2', 'd3', 'd4'], 'e_type':['e2',
'e3', 'e4']}, index=[1, 2, 3])
>>> dfr4
  d_type e_type
1     d2     e2
2     d3     e3
3     d4     e4

>>> df4 = pd.concat([dfr1, dfr4], axis=1, join='outer')
>>> df4
  a_type b_type c_type d_type e_type
0     a1     b1     c1    NaN    NaN
1     a2     b2     c2     d2     e2
2     a3     b3     c3     d3     e3
3    NaN    NaN    NaN     d4     e4
```

Если итоговая структура должна являться результатом пересечения (логическое И), то параметру `join` необходимо присвоить значение `'inner'`:

```
>>> df5 = pd.concat([dfr1, dfr4], axis=1, join='inner')
>>> df5
  a_type b_type c_type d_type e_type
1     a2     b2     c2     d2     e2
2     a3     b3     c3     d3     e3
```

5.3.2 Использование *Database-style* подхода

Суть данного подхода в том, что используется очень быстрый способ объединения структур данных, который идеологически похож на операции с реляционными базами данных. В состав *pandas* входит функция `merge`, которая представляет данный функционал:

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

Разберем аргументы данной функции (в представленном ниже списке указаны наиболее часто используемые аргументы):

- `left`: *DataFrame*
  - “левая” *DataFrame* структура.
- `right`: *DataFrame*
  - “правая” *DataFrame* структура.
- `how`: `{'left', 'right', 'outer', 'inner'}`; значение по умолчанию: `'inner'`
  - Один из методов объединения: `'left'`, `'right'`, `'outer'`, `'inner'`:

        - `'left'` - это аналог *SQL* операции `'LEFT OUTER JOIN'` при этом будут использоваться ключи только из левого *DataFrame.*
        - `'right'` - аналог *SQL* операции `'RIGHT OUTER JOIN'` - используются ключи из правого *DataFrame*.
        - *'*`outer'` - аналог *SQL* операции `'FULL OUTER JOIN'` - используется объединение ключей из правого и левого *DataFrame*.
        - `'inner'` - аналог *SQL* операции `'INNER JOIN'` - используется пересечение ключей из правого и левого *DataFrame*.
- on: список
    - Список имен столбцов для объединения, столбцы должны входить как в левый, так и в правый *DataFrame*.
- `left_on`: список
    - Список столбцов левого *DataFrame*, которые будут использоваться как ключи.
- `right_on`: список
    - Список столбцов из правого DataFrame, которые будут использоваться как ключи.
- `left_index`: `bool`; значение по умолчанию: `False`
    - Если параметр равен `True`, то будет использован индекс (метки строк) из левого *DataFrame* в качестве ключа(ей) для объединения.
- `right_index`: `bool`; значение по умолчанию: `False`
    - Если параметр равен `True`, то будет использован индекс (метки строк) из правого *DataFrame* в качестве ключа(ей) для объединения.

- `sort`: `bool`; значение по умолчанию: `False`
  - Если параметр равен `True`, то данные в полученном *DataFrame* будут отсортированы в лексикографическом порядке.

Рассмотрим несколько примеров того, как можно использовать функцию `merge()`.

Создадим три *DataFrame'a*:

```
dfr1 = pd.DataFrame({'k':['k1', 'k2', 'k3'], 'a_type':['a1', 'a2', 'a3'],
'b_type':['b1', 'b2', 'b3']})
>>> dfr1
    k a_type b_type
0  k1     a1     b1
1  k2     a2     b2
2  k3     a3     b3

>>> dfr2 = pd.DataFrame({'k':['k1', 'k2', 'k3'], 'c_type':['c1', 'c2',
'c3']})
>>> dfr2
    k c_type
0  k1     c1
1  k2     c2
2  k3     c3

>>> dfr3 = pd.DataFrame({'k':['k0', 'k1', 'k2'], 'c_type':['c1', 'c2',
'c3']})
>>> dfr3
    k c_type
0  k0     c1
1  k1     c2
2  k2     c3
```

Объединим их через `merge()`, в качестве ключа будем использовать столбец k:

```
>>> dfm1 = pd.merge(dfr1, dfr2, on='k')
>>> dfm1
    k a_type b_type c_type
0  k1     a1     b1     c1
1  k2     a2     b2     c2
2  k3     a3     b3     c3
```

Пример использования `how='left'`:

```
>>> dfm2 = pd.merge(dfr1, dfr3, how='left', on='k')
>>> dfm2
    k a_type b_type c_type
0  k1     a1     b1     c2
1  k2     a2     b2     c3
2  k3     a3     b3    NaN
```

Пример использования `how='right'`:

```
>>> dfm3 = pd.merge(dfr1, dfr3, how='right', on='k')
>>> dfm3
    k a_type b_type c_type
0  k1     a1     b1     c2
1  k2     a2     b2     c3
2  k0    NaN    NaN     c1
```

Пример использования `how='outer'`:

```
>>> dfm4 = pd.merge(dfr1, dfr3, how='outer', on='k')
>>> dfm4
    k a_type b_type c_type
0  k1     a1     b1     c2
1  k2     a2     b2     c3
2  k3     a3     b3    NaN
3  k0    NaN    NaN     c1
```

Пример использования how='inner':

```
>>> dfm5 = pd.merge(dfr1, dfr3, how='inner', on='k')
>>> dfm5
    k a_type b_type c_type
0  k1     a1     b1     c2
1  k2     a2     b2     c3
```

Глава 6. Работа с внешними источниками данных

6.1 Работа с данными в формате *CSV*

CSV (*Comma-Separated Values* - значения, разделённые запятыми) является одним из наиболее популярных форматов для хранения табличных данных, представляет собой текстовый документ, в котором элементы данных разделены запятыми, а строки файла являются записями в таблице.

6.1.1 Чтение данных

Для загрузки данных из *CSV* файлов в *pandas* используется метод `read_csv()`. Количество аргументов в нем достаточно велико, из наиболее часто используемых можно выделить следующие:

- `filepath_or_buffer`: `str`[1].
  - Путь до файла или буфера, который содержит данные в формате *CSV*.
- `sep`: `str`; значение по умолчанию: `','`
  - Разделитель, по умолчанию он равен `','`, т.к. в *CSV* данные разделены запятыми. Довольно часто встречается вариант, когда данные разделяются табуляцией, так называемые *TSV*-файлы, если вы используете такой формат, то необходимо параметру `sep` присвоить значение `'\t'`.
- `header`: `int` или список `int`'ов; значение по умолчанию: 0
  - Номер строки, которая содержит имена столбцов загружаемой таблицы. По умолчанию `header=0`. Если `header=None`, то имена столбцов можно передать в параметре `names`.

1 Также возможны: *pathlib.Path*, *py._path.local.LocalPath* или любой объект с методом *read()*

- `names`: массив; значение по умолчанию: `None`
    - Список имен столбцов таблицы, используется, если в файле нет строки с именами столбцов и параметр `header` равен `None`.

Данные в формате *CSV* можно как непосредственно создавать в *Python* программе, так и загрузить из файла. Начнем с первого варианта. Для начала загрузим необходимые нам библиотеки:

```
>>> import pandas as pd
>>> from io import StringIO
```

Теперь создадим строку, содержащую данные в формате *CSV* и построим на их базе *DataFrame*:

```
>>> csv_d1 = 'col_A, col_B, col_C\na1, b1, c1\na2, b2, c2'
>>> df = pd.read_csv(StringIO(csv_d1))
>>> df
  col_A  col_B  col_C
0    a1     b1     c1
1    a2     b2     c2
```

В приведенном выше примере в первой строке набора данных содержатся заголовки колонок таблицы. Но не всегда бывает так. Ниже приведен случай, когда в таблице с данными нет информации об именах столбцов, эту ситуацию можно исправить, если параметру `header` присвоить `None` и передать нужные значения через параметр `names`:

```
>>> csv_d2 = 'a1, b1, c1\na2, b2, c2\na3, b3, c3'
>>> df1 = pd.read_csv(StringIO(csv_d2), header=None, names=['type_a',
'type_b', 'type_c'])
>>> df1
  type_a type_b type_c
0     a1     b1     c1
1     a2     b2     c2
2     a3     b3     c3
```

Если файл с данными в формате *CSV* находится на диске, то для его загрузки достаточно передать имя файла в качестве первого параметра метода `read_csv()`:

```
>>> df3 = pd.read_csv('c:/test.csv')
>>> df3
   Name  Age         City
0  Anna   29       Moscow
1  John   21     New-York
2  Ivan   18        Tomsk
3  Mike   32  Los-Angeles
```

В случае, когда размер файла очень большой и его невозможно загрузить за одни раз в память (*DataFrame*), воспользуйтесь загрузкой по частям. Для этого нужно в функцию `read_csv()` дополнительно передать параметр `chunksize`, через который указывается количество строк, которые нужно считать в рамках одной порции. Такой подход позволяет работать с *DataFrame*'ами как с итераторами:

```
>>> df_chunks = pd.read_csv('c:/test.csv', chunksize=2)
>>> for i, chunk in enumerate(df_chunks):
      print(f'Chunk #{i}')
      print(chunk)
Chunk #0
   Name  Age      City
0  Anna   29    Moscow
1  John   21  New-York
Chunk #1
   Name  Age         City
2  Ivan   18        Tomsk
3  Mike   32  Los-Angeles
```

```
>>> df_chunks = pd.read_csv('c:/test.csv', chunksize=2)
>>> print(df_chunks.get_chunk())
   Name  Age      City
0  Anna   29    Moscow
1  John   21  New-York

>>> print(df_chunks.get_chunk())
   Name  Age         City
2  Ivan   18        Tomsk
3  Mike   32  Los-Angeles
```

6.1.2 Запись данных

Для записи данных в формате *CSV* используется метод `to_csv()`. Рассмотрим некоторые из аргументов, которые могут быть полезны:

- `path_or_buf`: `str` или `handle` файла; значение по умолчанию: `None`
  - Имя файла или буфера, в котором будут сохранены данные в формате *CSV*.
- `sep`: `str`; значение по умолчанию: `','`
  - Разделитель элементов данных, по умолчанию `sep=','`.
- `header`: `bool` или список строк; значение по умолчанию: `True`
  - Если в качестве значения передается `True`, то в файл первой строкой запишутся имена столбцов, взятые из структуры данных. Если будет передан список строк, то имена столбцов будут взяты из него.
- `encoding`: `str`
  - Кодировка. Если вы используете *Python* 3, то по умолчанию она равна `'utf-8'`.

Подготовим данные для записи:

```
>>> csv_data = 'col_A, col_B, col_C\na1, b1, c1\na2, b2, c2'
>>> df = pd.DataFrame(StringIO(csv_data))
```

Теперь запишем содержимое структуры `df` в файл с именем `tmp.csv`:

```
>>> df.to_csv('tmp.csv')
```

6.2 Работа с данными в формате *JSON*

С развитием и широким распространением языка программирования *JavaScript* возросла популярность формата данных, который органически совместим с этим языком - *JSON (JavaScript Object Notation)*. Подробно об этом формате можно прочитать в википедии (*https://ru.wikipedia.org/wiki/JSON*).

6.2.1 Чтение данных

Для чтения данных в формате *JSON* используется метод `read_json()`. Рассмотрим наиболее часто используемые аргументы данного метода:

- `path_or_buf`: *JSON* строка или файл; значение по умолчанию: `None`
  - Путь (это может быть как файл на диске, так и *URL*) до *JSON* файла или строка, содержимое которой - корректный *JSON*.
- `orient`: `str`; значение по умолчанию: `None`
  - Ориентация, для того, чтобы загружаемый *JSON* мог быть преобразован в структуру данных *pandas* он должен иметь определенный вид. Далее представлены возможные значения `orient` и соответствующий им *JSON:*
    - `'split'` : словарь со структурой `{index -> [index], columns -> [columns], data -> [values]}`

        - 'records': список со структурой [{column -> value}, ... , {column -> value}]
        - 'index': словарь со структурой {index -> {column -> value}}
        - 'columns': словарь со структурой {column -> {index -> value}}
        - 'values': массив значений.
- typ: тип объекта для записи; значение по умолчанию: 'frame'
    - Тип структуры *pandas*, 'series' - это *Series*, 'frame' - *DataFrame*. В зависимости от значения typ, можно использовать определенные значения orient. Если typ='series', то orient может быть 'split', 'records' или 'index', если typ='frame', то orient нужно выбрать из следующего списка: 'split', 'records', 'index', 'columns', 'values'.

Сейчас немного потренируемся с чтением *JSON* файлов и буферов. В зависимости от того, каким образом отформатированы данные в *JSON* файле или буфере, используется то или иное значение параметра orient метода read_json().

orient='split'

Для orient='split' формат данных *JSON* должен выглядеть следующим образом:

```
{
     'columns': ['col_A', 'col_B', 'col_C'],
     'index': [0, 1],
     'data': [['a1', 'b1', 'c1'], ['a2', 'b2', 'c2']]
}
```

Код для чтения *JSON*:

```
>>> json_buf='{'columns':['col_A', 'col_B', 'col_C'],'index':
[0,1],'data':[['a1','b1','c1'],['a2','b2','c2']]}'
>>> df1=pd.read_json(json_buf, orient='split')
>>> df1
  col_A col_B col_C
0    a1    b1    c1
1    a2    b2    c2
```

orient='records'

JSON данные:

```
[
    {
        'col_A': 'a1',
        'col_B': 'b1',
        'col_C':'c1'
    },
    {
        'col_A': 'a2',
        'col_B': 'b2',
        'col_C': 'c2'
    }
]
```

Код для чтения *JSON*:

```
>>> json_buf2='[{'col_A': 'a1', 'col_B': 'b1', 'col_C':'c1'}, {'col_A':
'a2', 'col_B': 'b2', 'col_C': 'c2'}]'
>>> df2 = pd.read_json(json_buf2, orient='record')
>>> df2
  col_A col_B col_C
0    a1    b1    c1
1    a2    b2    c2
```

orient='index'

JSON данные:

```
{
    0: {'col_A': 'a1', 'col_B': 'b1', 'col_C':'c1' }
    1: {'col_A': 'a2', 'col_B': 'b2', 'col_C':'c2' }
}
```

Код для чтения *JSON*:

```
>>> json_buf3='{'0': {'col_A': 'a1', 'col_B': 'b1', 'col_C':'c1' }, '1':
{'col_A': 'a2', 'col_B': 'b2', 'col_C':'c2' } }'
>>> df3 = pd.read_json(json_buf3, orient='index')
>>> df3
  col_A col_B col_C
0    a1    b1    c1
1    a2    b2    c2
```

orient='columns'

JSON данные:

```
{
     'col_A': {'0': 'a1', '1': 'a2'},
     'col_B': {'0': 'b1', '1': 'b2'},
     'col_C': {'0': 'c1', '1': 'c2'}
}
```

Код для чтения *JSON*:

```
>>> json_buf4='{'col_A': {'0': 'a1', '1': 'a2'},'col_B': {'0': 'b1', '1':
'b2'},'col_C': {'0': 'c1', '1': 'c2'}}'
>>> df4 = pd.read_json(json_buf4, orient='columns')
>>> df4
  col_A col_B col_C
0    a1    b1    c1
1    a2    b2    c2
```

`orient='values'`

JSON данные:

```
[
    ['a1', 'b1', 'c1'],
    ['a2', 'b2', 'c2']
]
```

Код для чтения *JSON*:

```
>>> json_buf5='[['a1', 'b1', 'c1'], ['a2', 'b2', 'c2']]'
>>> df5 = pd.read_json(json_buf5, orient='values')
>>> df5
    0   1   2
0  a1  b1  c1
1  a2  b2  c2
```

6.2.2 Запись данных

При работе с *JSON* довольно часто приходится преобразовывать уже готовые структуры данных в этот формат. Для этого используется функция `to_json()`.

Два самых важных аргумента данного метода - это `path_or_buf` и `orient`, их назначение тоже, что и в методе `read_json()`, только сейчас речь идет о записи данных, т.е. мы указываем файл или буфер, в который будут помещены данные.

Возьмем структуру *DataFrame* из предыдущей части:

```
>>> d = {'color':['red', 'green', 'blue'], 'speed': [56, 24, 65],
'volume': [80, 65, 50]}
>>> df = pd.DataFrame(d)
```

```
>>> df
   color  speed  volume
0    red     56      80
1  green     24      65
2   blue     65      50
```

В зависимости от того, какого вида *JSON* файл (буфер) мы хотим получить, необходимо параметру `orient` присвоить соответствующее значение, рассмотрим различные варианты.

`orient='split':`

```
>>> json_split = df.to_json(orient='split')
>>> json_split
'{'columns':['color','speed','volume'],'index':[0,1,2],'data':
[['red',56,80],['green',24,65],['blue',65,50]]}'
```

`orient='records':`

```
>>> json_records = df.to_json(orient='records')
>>> json_records
'[{'color':'red','speed':56,'volume':80},
{'color':'green','speed':24,'volume':65},
{'color':'blue','speed':65,'volume':50}]'
```

`orient='index':`

```
>>> json_index = df.to_json(orient='index')
>>> json_index
'{'0':{'color':'red','speed':56,'volume':80},'1':
{'color':'green','speed':24,'volume':65},'2':
{'color':'blue','speed':65,'volume':50}}'
```

orient='columns':

```
>>> json_columns = df.to_json(orient='columns')
>>> json_columns
'{'color':{'0':'red','1':'green','2':'blue'},'speed':
{'0':56,'1':24,'2':65},'volume':{'0':80,'1':65,'2':50}}'
```

orient='values':

```
>>> json_values = df.to_json(orient='values')
>>> json_values
'[['red',56,80],['green',24,65],['blue',65,50]]'
```

6.3 Работа с *Excel* файлами

Рассмотрим базовые возможности, которые предоставляет *pandas* для работы с *Excel*-файлами.

6.3.1 Чтение данных

Для чтения данных используется метод `read_excel()`. Он может работать как с файлами в формате *Excel* 2003 (расширение *.xls*), так и с файлами в формате *Excel* 2007 (расширение *.xlsx*). Для экспериментов создадим *Excel*-файл в формате *Excel* 2003 с именем *“test.xls”*, добавим в него два листа *“Sheet1”*, *“Sheet2”* (см. таблицы 6.1 и 6.2).

Таблица 6.1 - Содержание листа *Sheet1* файла *test.xls*

red	17	1
blue	35	2
white	42	3
yellow	63	4
green	53	5

Таблица 6.2 - Содержание листа *Sheet2* файла *test.xls*

3	*a*	*a1*
5	*h*	*b1*
1	*d*	*c1*
3	*s*	*d1*
6	*a*	*e1*

Прочитаем лист *“Sheet1”* и поместим полученные данные в *DataFrame*:

```
>>> df_xls_sheet1 = pd.read_excel('c:\\test.xls', sheetname='Sheet1',
header=None)
>>> df_xls_sheet1
        0   1  2
0     red  17  1
1    blue  35  2
2   white  42  3
3  yellow  63  4
4   green  53  5
```

Обратите внимание на параметр `header`. По умолчанию он равен нулю - это означает, что заголовки столбцов лежат в строке с номером 0. В нашем случае у таблиц нет заголовков, поэтому нужно параметру `header` присвоить значение `None`.

Прочитаем содержимое листа *“Sheet2”:*

```
>>> df_xls_sheet2 = pd.read_excel('c:\\test.xls', sheetname='Sheet2',
header=None)
>>> df_xls_sheet2
   0  1   2
0  3  a  a1
1  5  h  b1
2  1  d  c1
3  3  s  d1
4  6  a  e1
```

Для работы с *Excel*-файлом можно использовать класс `ExcelFile`, объекты которого связываются с определенным файлом на диске. Объекты этого класса являются контекстными менеджерами, что делает возможным работу с конструкцией `with`:

```
>>> with pd.ExcelFile('c:\\test.xls') as excel:
      df1 = pd.read_excel(excel, sheetname='Sheet1', header=None)
      print(df1)

    0   1  2
0     red  17  1
1    blue  35  2
2   white  42  3
3  yellow  63  4
4   green  53  5
```

В самом простом случае, работа с `ExcelFile` может выглядеть вот так:

```
>>> excel = pd.ExcelFile('c:\\test.xls')
>>> df2 = pd.read_excel(excel, sheetname='Sheet2', header=None)
>>> df2
  0  1   2
0  3  a  a1
1  5  h  b1
2  1  d  c1
3  3  s  d1
4  6  a  e1
```

6.3.2 Запись данных

Для записи данных в *Excel*-файл предварительно подготовим соответствующий *DataFrame*. Воспользуемся примером из раздела, посвященного *JSON*:

```
>>> json_buf='[['a1', 'b1', 'c1'], ['a2', 'b2', 'c2']]'
>>> df = pd.read_json(json_buf, orient='values')
>>> df
    0   1   2
0  a1  b1  c1
1  a2  b2  c2
```

Запись осуществляется с помощью метода `to_excel()`:

```
>>> df.to_excel('test_excel.xlsx', sheet_name='Sheet1')
```

В результате мы получим *Excel* файл с именем *test_excel.xlsx*, у которого будет одна страница, называющаяся *Sheet1*, содержание которой представлено в таблице 6.3.

Таблица 6.3 - Содержание листа *Sheet1* файла *test_excel.xlsx*

	0	1	2
0	*a1*	*b1*	*c1*
1	*a2*	*b2*	*c2*

Если параметру `header` присвоить значение `None`, то в таблице будет отсутствовать заголовки колонок:

```
>>> df.to_excel('test_excel1.xlsx', sheet_name='Sheet1', header=None)
```

Содержимое файла *test_excel1.xlsx* представлено в таблице 6.4.

Таблица 6.4 - Содержание листа *Sheet1* файла *test_excel1.xlsx*

0	*a1*	*b1*	*c1*
1	*a2*	*b2*	*c2*

Для того, чтобы убрать индексы (имена) строк нужно дополнительно добавить параметр `index=False`:

```
>>> df.to_excel('test_excel2.xlsx', sheet_name='Sheet1', header=None,
index=False)
```

В результате получим файл *test_excel2.xlsx* с данными из таблицы 6.5.

Таблица 6.5 - Содержание листа *Sheet1* файла *test_excel2.xlsx*

a1	*b1*	*c1*
a2	*b2*	*c2*

Глава 7. Операции над данными

7.1 Арифметические операции

Структуры данных *pandas* можно складывать, вычитать, умножать и делить (поэлементно).

Для начала создадим две структуры *DataFrame:*

```
>>> json_buf1='[['10', '20', '30'], ['40', '50', '60']]'
>>> df1 = pd.read_json(json_buf1, orient='values')
>>> json_buf2='[['12', '24', '14'], ['16', '54', '25']]'
>>> df2 = pd.read_json(json_buf2, orient='values')
>>> df1
    0   1   2
0  10  20  30
1  40  50  60
>>> df2
    0   1   2
0  12  24  14
1  16  54  25
```

Рассмотренные ниже методы не модифицируют саму структуру, они возвращают новую структуру, которую можно сохранить в отдельной переменной.

Для сложения структур используется метод `add()`:

```
>>> df1.add(df2)
    0    1   2
0  22   44  44
1  56  104  85
```

К элементам структуры можно добавить константу:

```
>>> df1.add(5)
    0   1   2
0  15  25  35
1  45  55  65
```

Вычитание осуществляется с помощью метода `sub()`:

```
>>> df1.sub(df2)
    0  1   2
0  -2 -4  16
1  24 -4  35

>>> df1.sub(7)
    0   1   2
0   3  13  23
1  33  43  53
```

Для умножения структур применяется метод `mul()`:

```
>>> df1.mul(df2)
     0     1     2
0  120   480   420
1  640  2700  1500

>>> df1.mul(2)
    0    1    2
0  20   40   60
1  80  100  120
```

Для деления используется метод `div()`:

```
>>> df1.div(df2)
          0         1         2
0  0.833333  0.833333  2.142857
1  2.500000  0.925926  2.400000
>>> df1.div(2)
      0     1     2
0   5.0  10.0  15.0
1  20.0  25.0  30.0
```

7.2 Логические операции

По имеющимся структурам можно строить новые, элементами которых будут логические переменных, значения которых определяются тем, удовлетворяет ли элемент исходной структуры определенному условию или нет.

Возьмем структуру `df2` из предыдущего раздела:

```
>>> json_buf2='[['12', '24', '14'], ['16', '54', '25']]'
>>> df2 = pd.read_json(json_buf2, orient='values')
>>> df2
    0   1   2
0  12  24  14
1  16  54  25
```

Определим элементы структуры `df2`, значения которые больше 20:

```
>>> df2 > 20
       0     1      2
0  False  True  False
1  False  True   True
```

Pandas предоставляет инструменты свертки структур данных для получения сводной информации. Для выполнения операции “логическое ИЛИ” по строкам или столбцам используется метод `any()`. Выбор направления определяется параметром `axis`.

Свертка по столбцам:

```
>>> (df2 > 20).any()
0    False
1     True
2     True
dtype: bool
```

Свертка по строкам:

```
>>> (df2 > 20).any(axis=1)
0    True
1    True
dtype: bool
```

Для выполнения операции “логическое И” по строкам или столбцам используется метод `all()`. Выбор направления также определяется параметром `axis`.

Свертка по столбцам:

```
>>> (df2 > 20).all()
0    False
1     True
2    False
dtype: bool
```

Свертка по строкам:

```
>>> (df2 > 20).all(axis=1)
0    False
1    False
dtype: bool
```

Для сравнения элементов структур на равенство можно использовать метод `equals()` или оператор проверки на равенства (==) из языка *Python*. Создадим ещё одну структуру *DataFrame*:

```
>>> json_buf3='[['12', '17', '18'], ['16', '54', '68']]'
>>> df3 = pd.read_json(json_buf3, orient='values')
>>> df3
    0   1   2
0  12  17  18
1  16  54  68
```

Вспомним как выглядит структура `df2`:

```
>>> df2
    0   1   2
0  12  24  14
1  16  54  25
```

Если сравнить эти структуры с помощью оператора == , то в результате получим *DataFrame*, элементами которого будут логические переменные. Если значения элементов на данной позиции в обоих *DataFrame'*ах совпадают, то переменная будет равна `True`, в противном случае `False`:

```
>>> df2 == df3
      0      1      2
0  True  False  False
1  True   True  False
```

Можно сравнивать каждый элемент структуры с некоторым константным значением:

```
>>> df2 == 12
       0      1      2
0   True  False  False
1  False  False  False
```

Для быстрой проверки равенства двух структур используется метод `equals()`. Если структуры равны (на одних и тех же позициях стоят одни и те же значения), то результат будет `True`, в противном случае `False`:

```
>>> df2.equals(df3)
False
>>> df2.equals(df2)
True
```

7.3 Статистики

Pandas предоставляет методы для расчета различных статистик (список приведен в таблице 7.1). Если вы работаете со структурами *DataFrame*, то можно указать ось, по которой будет производиться расчет: `axis=0` для столбцов, `axis=1` для строк, по умолчанию `axis=0`.

Таблица 7.1 - Методы для расчета статистик

Метод	Описание
`count`	Количество *не-NA* объектов
`sum`	Сумма
`mean`	Среднее значение
`mad`	Среднее абсолютное отклонение
`median`	Медиана
`min`	Минимум
`max`	Максимум
`mode`	Мода
`abs`	Абсолютное значение
`prod`	Произведение

std	Стандартное отклонение
var	Несмещенная дисперсия
sem	Стандартная ошибка среднего
skew	Скошенность (момент 3-го порядка)
kurt	Эксцесс (момент 4-го порядка)
quantile	Квантиль (%)
cumsum	Кумулятивная сумма
cumprod	Кумулятивное произведение
cummax	Кумулятивный максимум
cummin	Кумулятивный минимум

Теперь рассмотрим несколько примеров того, как можно использовать данные методы. Создадим *DataFrame* размера три на четыре:

```
>>> json_buf='[['12', '24', '14', '17'], ['16', '54', '25', '83'], ['65', '35', '12', '72']]'
>>> df = pd.read_json(json_buf, orient='value')
>>> df
    0   1   2   3
0  12  24  14  17
1  16  54  25  83
2  65  35  12  72
```

Ниже представлены примеры работы функций расчета статистик.

Сумма по столбцам:

```
>>> df.sum()
0     93
1    113
2     51
3    172
dtype: int64
```

Сумма по строкам:

```
>>> df.sum(axis=1)
0     67
1    178
2    184
dtype: int64
```

Среднее значение по столбцам:

```
>>> df.mean()
0    31.000000
1    37.666667
2    17.000000
3    57.333333
dtype: float64
```

Медиана по столбцам:

```
>>> df.median()
0    16.0
1    35.0
2    14.0
3    72.0
dtype: float64
```

Кумулятивная сумма по столбцам:

```
>>> df.cumsum()
     0    1   2    3
0   12   24  14   17
1   28   78  39  100
2   93  113  51  172
```

Для получения сводной информации по статистикам можно воспользоваться методом `describe()`:

```
>>> df.describe()
              0          1     2          3
count   3.000000   3.000000   3.0   3.000000
mean   31.000000  37.666667  17.0  57.333333
std    29.512709  15.176737   7.0  35.360053
min    12.000000  24.000000  12.0  17.000000
25%    14.000000  29.500000  13.0  44.500000
50%    16.000000  35.000000  14.0  72.000000
75%    40.500000  44.500000  19.5  77.500000
max    65.000000  54.000000  25.0  83.000000
```

Для структур *DataFrame*, по умолчанию, статистики рассчитываются для столбцов.

Завершим обзор возможностей *pandas* для расчета статистик функцией `value_counts()`. Для начала создадим список из тридцати значений, каждое из которых является случайным числом в диапазоне от 0 до 7:

```
>>> import random
>>> random.seed(123)
>>> rnd_list = [random.randint(0, 7) for i in range(30)]
```

Для получения информации о количестве конкретных чисел в получившемся списке воспользуемся функцией `value_counts()`:

```
>>> s = pd.Series(rnd_list)
>>> s.value_counts()
0    9
2    7
4    4
3    4
1    4
5    2
dtype: int64
```

Как видно, больше всего в нашем массиве нулей.

7.4 Функциональное расширение

7.4.1 Потоковая обработка данных

Потоковая обработка данных - это подход, который позволяет в удобном виде задавать обработку данных таким образом, что структура *pandas* является аргументом некоторой функции, результат которой передается в следующую функцию и т.д. Это похоже на конвейерную обработку. Для использования данного подхода применяется метод `pipe()`. Рассмотрим это на примере. Для начала реализуем такую обработку без использования функции `pipe()`.

Построим *DataFrame* из *JSON*-строки:

```
>>> json_buf='[['12', '24', '14', '17'], ['16', '54', '25', '83'], ['65', '35', '12', '72']]'
>>> df = pd.read_json(json_buf, orient='value')
```

Создадим три функции: возведение в квадрат, корень третьей степени и функция, вычитающая число 10:

```
>>> sqr = lambda x: x**2
>>> root3 = lambda x: x**(1.0/3.0)
>>> minus10 = lambda x: x - 10
```

Если мы хотим сначала возвести каждый элемент структуры `df` в квадрат, потом вычесть из полученного значения 10, а следом взять корень третьей степени, то самый простой способ реализации такой задачи выглядит так:

```
>>> root3(minus10(sqr(df)))
           0          1         2          3
0   5.117230   8.271904  5.708267   6.534335
1   6.265827  14.270259  8.504035  19.018449
2  16.153471  10.670680  5.117230  17.295859
```

С такой записью есть несколько проблем: во-первых: она не очень наглядная (хотя не все с этим согласятся), во-вторых: у данных функций могут быть дополнительные аргументы, что будет затруднять чтение и понимание смысла строки кода; в-третьих: модификация такой записи, при большом количестве используемых функций, вызовет трудности.

Обойти эти затруднения можно с помощью функции `pipe()`, аргументом которой является обрабатывающая функция:

```
>>> (df.pipe(sqr)
     .pipe(minus10)
     .pipe(root3))
           0          1         2          3
0   5.117230   8.271904  5.708267   6.534335
1   6.265827  14.270259  8.504035  19.018449
2  16.153471  10.670680  5.117230  17.295859
```

7.4.2 Применение функции к элементам строки или столбца

В разделе, посвященном расчету статистик, мы использовали специальные функции (`mean()`, `std()` и т.д.), вычисляющие нужные нам численные показатели по элементам строки или столбца. *Pandas*

предоставляет возможность использовать свои собственные функции применяя аналогичный подход. Для этого используется метод `apply()`.

Эксперименты будем проводить со структурой `df` (см. потоковая обработка данных):

```
>>> df.apply(lambda x: sum(x)/len(x))
0    31.000000
1    37.666667
2    17.000000
3    57.333333
dtype: float64
```

В приведенном выше примере, аргументами функции `lambda x: sum(x)/len(x)` являются списки, состоящие из столбцов исходной структуры.

Вот вариант функции, которая вычисляет квадратный корень из суммы по столбцам:

```
>>> df.apply(lambda x: sum(x)**(0.5))
0     9.643651
1    10.630146
2     7.141428
3    13.114877
dtype: float64
```

Такую же операцию можно сделать построчно:

```
>>> df.apply(lambda x: sum(x)**(0.5), axis=1)
0     8.185353
1    13.341664
2    13.564660
dtype: float64
```

7.4.3 Агрегация (*API*)

Ещё одним функциональным расширением, которое предоставляет библиотека *pandas* является агрегация. Суть в том, что можно использовать несколько функций в том виде, как это мы делали в предыдущем разделе, когда разбирали функцию `apply()`, только в этом случае нам поможет функция `agg()`.

Работать будем с уже известной нам структурой `df` (см. потоковая обработка данных):

```
>>> df
    0   1   2   3
0  12  24  14  17
1  16  54  25  83
2  65  35  12  72
```

Для начала найдем сумму элементов по столбцам, вы уже должны знать как минимум 2-3 способа как это можно сделать, вот ещё одни в копилку:

```
>>> df.agg('sum')
0     93
1    113
2     51
3    172
dtype: int64
```

Но что если мы хотим сразу посчитать сумму, среднее значение и стандартное отклонение? Используя агрегацию это очень просто сделать:

```
>>> df.agg(['sum', 'mean', 'std'])
              0           1     2           3
sum   93.000000  113.000000  51.0  172.000000
mean  31.000000   37.666667  17.0   57.333333
std   29.512709   15.176737   7.0   35.360053
```

Не забудем и про свои собственные функции:

```
>>> strange = lambda x: sum(x)**(0.5)
>>> min_div_5 = lambda x: min(x) / 5.0
>>> df.agg([max, strange, min_div_5])
                  0          1          2          3
max       65.000000  54.000000  25.000000  83.000000
<lambda>   9.643651  10.630146   7.141428  13.114877
<lambda>   2.400000   4.800000   2.400000   3.400000
```

Как вы заметили, в данном примере, на месте имен функций `strange` и `min_div_5` стоит надпись `<lambda>`, чтобы это убрать перепишем эти функции как обычные *python*-функции:

```
>>> def strange(x):
      return sum(x)**(0.5)

>>> def min_div_5(x):
      return min(x) / 5.0

>>> df.agg([max, strange, min_div_5])
                   0          1          2          3
max        65.000000  54.000000  25.000000  83.000000
strange     9.643651  10.630146   7.141428  13.114877
min_div_5   2.400000   4.800000   2.400000   3.400000
```

Теперь все в порядке! Если вернуться к предыдущему варианту с *lambda*-функциями, можно добиться нужного нам результата присвоив функциям соответствующие имена (в качестве имени может выступать любая строка):

```
>>> strange = lambda x: sum(x)**(0.5)
>>> min_div_5 = lambda x: min(x) / 5.0
>>> strange.__name__ = 'strange fun'
>>> min_div_5.__name__ = 'min / 5'
>>> df.agg([max, strange, min_div_5])
                     0          1          2          3
max          65.000000  54.000000  25.000000  83.000000
strange fun   9.643651  10.630146   7.141428  13.114877
min / 5       2.400000   4.800000   2.400000   3.400000
```

7.4.4 Трансформирование данных

Формат вызова трансформирующей функции похож на тот, что используется при агрегации. В отличии от последней, трансформация - это применение функции к каждому элементу структуры, в результате будет возвращена структура того же размера (или больше), но с измененными элементами.

Создадим две функции:

```
>>> mul2 = lambda x: x * 2
>>> mul2.__name__ = 'mul2'
>>> div2 = lambda x: x / 2
>>> div2.__name__ = 'div2'
```

Используем трансформирующую функцию для модификации элементов структуры df (см. потоковая обработка данных):

```
>>> df.transform([mul2])
     0    1    2    3
  mul2 mul2 mul2 mul2
0   24   48   28   34
1   32  108   50  166
2  130   70   24  144
```

Вариант с двумя функциями для модификации:

```
>>> df.transform([mul2, div2])
     0           1           2           3
  mul2  div2 mul2  div2 mul2  div2 mul2  div2
0   24   6.0   48  12.0   28   7.0   34   8.5
1   32   8.0  108  27.0   50  12.5  166  41.5
2  130  32.5   70  17.5   24   6.0  144  36.0
```

Можно работать с отдельными столбцами:

```
>>> df[0].transform([mul2, div2])
   mul2  div2
0    24   6.0
1    32   8.0
2   130  32.5
```

7.5 Использование методов типа *str* для работы с текстовыми данными

Часто структуры данных *pandas* используются для хранения текстовых данных - строк и символов. Текстовые данные также могут быть частью самой структуры, например заголовки столбцов и т.п. *Pandas* позволяет использовать функционал типа `str` языка *Python* для работы с такими данными.

Рассмотрим пару примеров на практике. Для начала создадим *Series*, содержащий текстовые данные:

```
>>> s = pd.Series([' hellO', ' abcABC ', ' one', 'TWO ', ' tHRee'])
>>> s
0       hellO
1     abcABC
2         one
3        TWO
4       tHRee
dtype: object
```

Через атрибут `str` мы можем получить доступ ко всем методам типа данных `str` из языка *Python* применительно к элементам структуры *Series*.

Приведем все буквы элементов структуры `s` к нижнему регистру:

```
>>> s.str.lower()
0       hello
1     abcabc
2         one
3        two
4       three
dtype: object
```

Удалим все лишние пробелы и сделаем заглавными первые буквы слов:

```
>>> s.str.lower().str.strip().str.title()
0     Hello
1    Abcabc
2       One
3       Two
4     Three
dtype: object
```

Весь список методов, доступных для объекта типа `str`, можно найти в документации по языку *Python*.

Глава 8. Настройка *pandas*

8.1 *API* для работы с настройками *pandas*

Библиотека *pandas* предоставляет *API* для работы с настройками, в него входят функции, перечисленные в таблице 8.1.

Таблица 8.1 - Функции для работы с настройками *pandas*

Функция	**Описание**
`get_option()`	Получение значения параметра
`set_option()`	Установка нового значения параметра
`reset_option()`	Сброс параметра на значение “по умолчанию”
`describe_option()`	Вывод текстового описания параметра
`option_context()`	Присвоение параметрам новых значений в рамках определенного блока кода. Используется с оператором `with`

Также возможна работа с настройками как с атрибутами (через точку).

Рассмотрим первый вариант: изменение настроек через вызов функции. Для получения значения параметра используется функция `get_option()`, в качестве аргумента ей передается настраевыемый параметр в текстовом виде.

Максимальное количество выводимых строк:

```
>>> pd.get_option('display.max_rows')
60
```

Используемая кодировка:

```
>>> pd.get_option('display.encoding')
'cp1251'
```

Максимальное количество выводимых столбцов с данными:

```
>>> pd.get_option('display.max_columns')
0
```

Установка нового значения осуществляется с помощью функции `set_option()`: ее первым аргументом является название параметра в текстовом виде, вторым - новое значение параметра.

Создадим структуру *Series* для демонстрации:

```
>>> s = pd.Series([10, 3, 46, 1, 312, 344, 193, 42, 39, 77, 3])
>>> s
0      10
1       3
2      46
3       1
4     312
5     344
6     193
7      42
8      39
9      77
10      3
dtype: int64
```

Текущее значение параметра `display.max_rows` (максимальное количество выводимых строк):

```
>>> pd.get_option('display.max_rows')
60
```

Изменим это значение на 5:

```
>>> pd.set_option('display.max_rows', 5)
>>> pd.get_option('display.max_rows')
5
```

Вывод содержимого структуры `s` с новыми настройками:

```
>>> s
0     10
1      3
      ..
9     77
10     3
Length: 11, dtype: int64
```

Установим прежнее значение:

```
>>> pd.set_option('display.max_rows', 60)
>>> pd.get_option('display.max_rows')
60


>>> s
0      10
1       3
2      46
3       1
4     312
5     344
6     193
7      42
8      39
9      77
10      3
dtype: int64
```

Значение параметра можно сбросить на значение “по умолчанию”, для этого используется функция `reset_option()`.

Исходное значение display.max_rows:

```
>>> pd.get_option('display.max_rows')
60
```

Присвоим новое значение параметру display.max_rows=10:

```
>>> pd.set_option('display.max_rows', 10)
>>> pd.get_option('display.max_rows')
10
```

Сбрасим display.max_rows на значение “по умолчанию”:

```
>>> pd.reset_option('display.max_rows')
>>> pd.get_option('display.max_rows')
60
```

Описание параметра можно получить с помощью функции describe_option():

```
>>> pd.describe_option('display.max_rows')
display.max_rows : int
    If max_rows is exceeded, switch to truncate view. Depending on
    `large_repr`, objects are either centrally truncated or printed as
    a summary view. 'None' value means unlimited.

    In case python/IPython is running in a terminal and `large_repr`
    equals 'truncate' this can be set to 0 and pandas will auto-detect
    the height of the terminal and print a truncated object which fits
    the screen height. The IPython notebook, IPython qtconsole, or
    IDLE do not run in a terminal and hence it is not possible to do
    correct auto-detection.
    [default: 60] [currently: 60]
```

Если необходимо выполнить блок кода, в рамках которого нужно временно присвоить ряду настроенных параметров определенные значения, то в этом случае рекомендуется использовать функцию `option_context()`:

```
>>> with pd.option_context('display.max_rows', 25):
    print(pd.get_option('display.max_rows'))
25
>>> pd.get_option('display.max_rows')
60
```

Второй вариант работы с настройками — это использование свойства `options`. Выведем значения уже знакомых нам настроек:

```
>>> pd.options.display.max_rows
60
>>> pd.options.display.max_columns
0
```

Присвоим параметру новое значение:

```
>>> pd.options.display.max_columns = 3
>>> pd.options.display.max_columns
3
>>> pd.options.display.max_columns = 20
>>> pd.options.display.max_columns
20
```

8.2 Настройки библиотеки *pandas*

Ниже, в таблице 8.2, представлены настраиваемые параметры, которые доступны при работе с *pandas*.

Таблица 8.2 - Настраиваемые параметры *pandas*

Настройка	Значение по умолчанию	Описание
`display.chop_threshold`	`None`	Если *float*-значение ниже этого порога, то оно будет отображаться равным 0
`display.colheader_justify`	`right`	Выравнивание заголовков столбцов. Используется *DataFrameFormatter'ом.*
`display.date_dayfirst`	`False`	Если параметр равен *True*, то при отображении и парсинге дат будет использоваться следующий порядок: день/месяц/год.
`display.date_yearfirst`	`False`	Если параметр равен *True*, то при отображении и парсинге дат будет использоваться следующий порядок: год/месяц/день.
`display.encoding`	`UTF-8`	Определяет кодировку для строк, возвращаемых через метод *to_string*, а также для отображения строк в консоли.
`display.max_columns`	20	Если *Python/IPython* запущен в терминале, то данный параметр может быть установлен в 0, в этом случае *pandas* будет автоматически определять ширину терминала и использовать другой формат представления, если все столбцы не поместились вертикально. Значение *None* определяет неограниченное количество символов.
`display.max_colwidth`	50	Максимальная ширина столбца (в символах) для представления структур данных *pandas*. Когда происходит переполнение столбца по символам, то в вывод будут добавлены символы "...".
`display.max_rows`	60	Параметр устанавливает максимальное количество строк, которые будут выведены при отображении структур *pandas*.

		Для снятия ограничения на количество строк, присвойте этому параметру значение *None*.
`display.memory_usage`	`True`	Указывает, должен ли отображаться объем занимаемой памяти для *DataFrame* при вызове метода *df.info()*.
`display.notebook_repr_html`	`True`	Если параметр равен *True*, то *IPython notebook* будет использовать *html*-представление для объектов *pandas*.
`display.precision`	6	Определяет количество знаков после запятой при выводе чисел с плавающей точкой.
`display.width`	80	Ширина дисплея в символах. Если *python/IPython* запущен в терминале, то параметр может быть установлен в *None*, в этом случае *pandas* автоматически определит ширину.
`display.html.border`	1	Определяет значение для атрибута *border=value*, который будет вставлен в тег <*table*> для *HTML*-представления структуры *DataFrame*.
`io.excel.xls.writer`	`xlwt`	Используемый по умолчанию движок для работы с *Excel*-файлами в формате “*xls*”.
`io.excel.xlsm.writer`	`openpyxl`	Используемый по умолчанию движок для работы с *Excel*-файлами в формате “*xlsm*”.
`io.excel.xlsx.writer`	`openpyxl`	Используемый по умолчанию движок для работы с *Excel*-файлами в формате “*xlsx*”.
`mode.sim_interactive`	`False`	Моделирование интерактивного режима, используется при тестировании.

Глава 9. Инструменты для работы с данными

9.1 Скользящее окно. Статистики

Первый набор инструментов, который мы рассмотрим, относится к группе, которую можно назвать скользящие статистики (или оконные функции). Суть их заключается в том, что различные статистики, такие как математическое ожидание, медиана, ковариация, стандартное отклонение и т.п. расчитываются не для всех объектов структуры, а только для группы подряд идущих значений, размер этой группы предварительно задается задавать вручную (этот параметр носит название - размер окна).

Рисунок 9.1 — Скользящее окно

Расчет статистик (например среднего значения) будет выглядеть так:

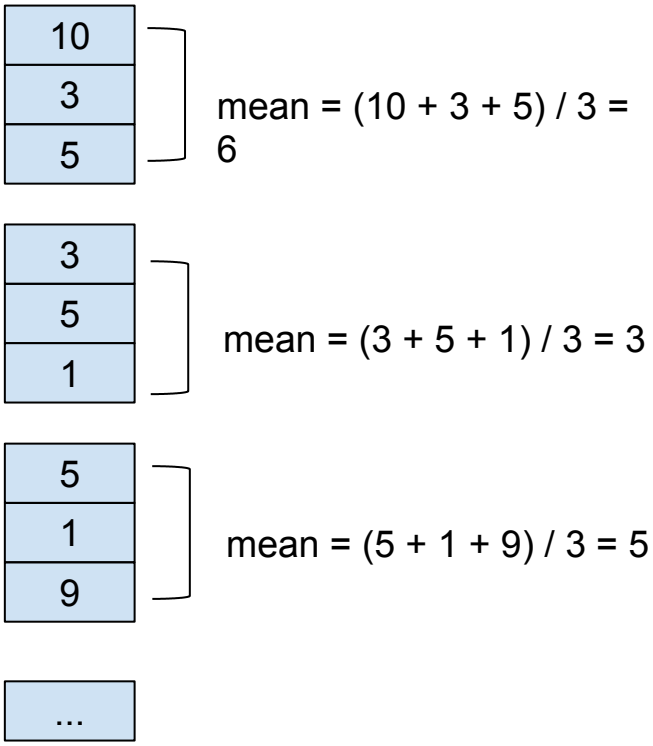


Рисунок 9.2 — Расчет статистик (среднего значения)

Для проведения экспериментов создадим структуру *Series*:

```
>>> import pandas as pd
>>> import random
>>> arr = [random.randint(0, 50) for i in range(500)]
>>> s = pd.Series(arr)
>>> s.shape
(500,)
>>> s[0:5]
0    28
1    48
2     8
3     4
4    21
dtype: int64
```

Для расчета различных статистик с заданным окном, для начала, создадим объект класса `Rolling`. Для этого воспользуемся функцией `pandas.DataFrame.rolling()`:

```
DataFrame.rolling(window, min_periods=None, freq=None, center=False, win_type=None, on=None, axis=0, closed=None)
```

Данная функция имеет следующие параметры:

- `window`: `int`
  - Размер окна.
- `min_periods`: `int`; значение по умолчанию: `None`
  - Минимальное количество элементов в окне для получения значения статистики.
- `freq`: `str` или объект `DateOffset`; значение по умолчанию: `None`
  - В настоящий момент данный параметр имеет статус `Deprecated` и не рекомендован к использованию.
- `center`: `bool`; значение по умолчанию: `False`
  - Установка метки в центр окна.
- `win_type`: `str`; значение по умолчанию: `None`
  - Тип окна (об этом параметре более подробно будет рассказано позже).
- `on`: `str`; значение по умолчанию: `None`
  - Задает столбец, по которому будут производиться вычисления.
- `closed`: `str`; значение по умолчанию: `None`
  - Определяет конечные точки закрытого интервала (`'right'`, `'left'`, `'both'`, `'neither'`).
- `axis`: `int` или `str`; значение по умолчанию: 0
  - Ось, вдоль которой будут производиться вычисления (0 - вычисления по строкам, 1 - по столбцам).

Создадим объект `Rolling` с настройками по умолчанию и зададим размер окна равным 10:

```
>>> roll = s.rolling(window=10)
>>> roll
Rolling [window=10,center=False,axis=0]
```

Данный объект предоставляет методы для расчета скользящих статистик. Список методов практически аналогичен тому, что был приведен в главе 7 (раздел “статистики”). Ниже, в таблице 9.1, приведены наиболее часто используемые функции.

Таблица 9.1 - Методы, используемые для расчета скользящих статистик

Метод	**Описание**
`apply()`	Функция общего назначения
`corr()`	Корреляция
`cov()`	Ковариация
`count()`	Количество *не-NA* объектов
`kurt()`	Эксцесс (момент 4-го порядка)
`max()`	Максимум
`mean()`	Среднее значение
`median()`	Медиана
`min()`	Минимум
`quantile()`	Квантиль (%)
`skew()`	Скошенность (момент 3-го порядка)
`std()`	Стандартное отклонение
`sum()`	Сумма
`var()`	Несмещенная дисперсия

Теперь можно использовать функции расчета статистик применительно к созданному набору данных с заданным окном:

```
>>> pd.options.display.max_rows=20
>>> roll.median()
0       NaN
1       NaN
2       NaN
3       NaN
4       NaN
5       NaN
6       NaN
7       NaN
8       NaN
9      24.5
       ...
490    29.5
491    24.0
492    18.5
493    21.0
494    23.0
495    23.0
496    23.0
497    23.0
498    23.0
499    23.0
Length: 500, dtype: float64
```

Для ограничения выводимой информации мы воспользовались настройкой `max_rows`, о том, что это такое и как использовать данный параметр написано в главе 8. Как вы можете видеть: до десятого элемента (в нашем случае, это элемент с индексом 9 т.к. счет начинается с нуля) элементы структуры имеют значение `NaN`. Это связно

с тем, что размер окна равен 10, и статистика считается по десяти предыдущим элементам, такой момент наступает, когда мы доходим до индекса 9.

Вычисление стандартного отклонения для окна размера 10:

```
>>> roll.std()
0            NaN
1            NaN
2            NaN
3            NaN
4            NaN
5            NaN
6            NaN
7            NaN
8            NaN
9      16.445533
         ...
490    15.307587
491    16.308144
492    17.049275
493    17.034605
494    15.486195
495    16.642983
496    16.578433
497    17.678927
498    17.142864
499    16.892141
Length: 500, dtype: float64
```

Определение элемента с максимальным численным значением среди элементов окна:

```
>>> roll.max()
0       NaN
1       NaN
2       NaN
3       NaN
4       NaN
5       NaN
6       NaN
7       NaN
8       NaN
9      48.0
       ...
490    48.0
491    48.0
492    48.0
493    48.0
494    48.0
495    48.0
496    48.0
497    48.0
498    48.0
499    48.0
Length: 500, dtype: float64
```

Можно использовать свои функции для расчета статистик, для этого их необходимо предварительно создать, а потом передать в качестве параметра методу *apply()*.

Pandas позволяет задать тип окна, это делается с помощью параметра *win_type* функции *rolling*. Ниже приведена таблица возможных значений данного параметра.

Таблица 9.2 - Параметры, определяющие тип окна для расчета статистик

Значение параметра	Описание
boxcar	Прямоугольное окно (окно Дирихле)
triang	Треугольное окно
blackman	Окно Блэкмана
hamming	Окно Хемминга
bartlett	Окно Барлетта
parzen	Окно Парзена
bohman	Окно Бохмана
blackmanharris	Окно Блэкмана-Харриса
nuttall	Окно Наттела
barthann	Окно Барлетта-Ханна
kaiser	Окно Кайзера
gaussian	Окно Гаусса
general_gaussian	Обобщенное Гауссовое окно
slepian	Окно Слепиана

Описание особенностей работы с данными параметрами выходит за рамки книги.

9.2 Расширяющееся окно. Статистики

В предыдущем разделе мы познакомились со скользящими окнами и расчетом статистик по ним. Этот раздел посвящен расширяющимся окнами, размер которых, в отличии от скользящих, изменяется, расширяясь, начиная с первого элемента.

Принцип работы расширяющегося окна, представлен на рисунках ниже.

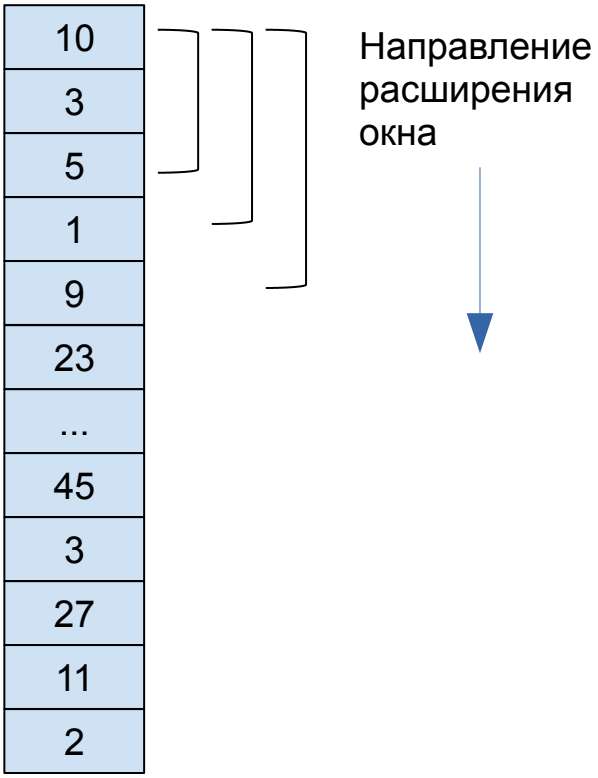


Рисунок 9.3 — Расширяющееся окно

Расчет среднего значения для такого окна будет производиться следующим образом:

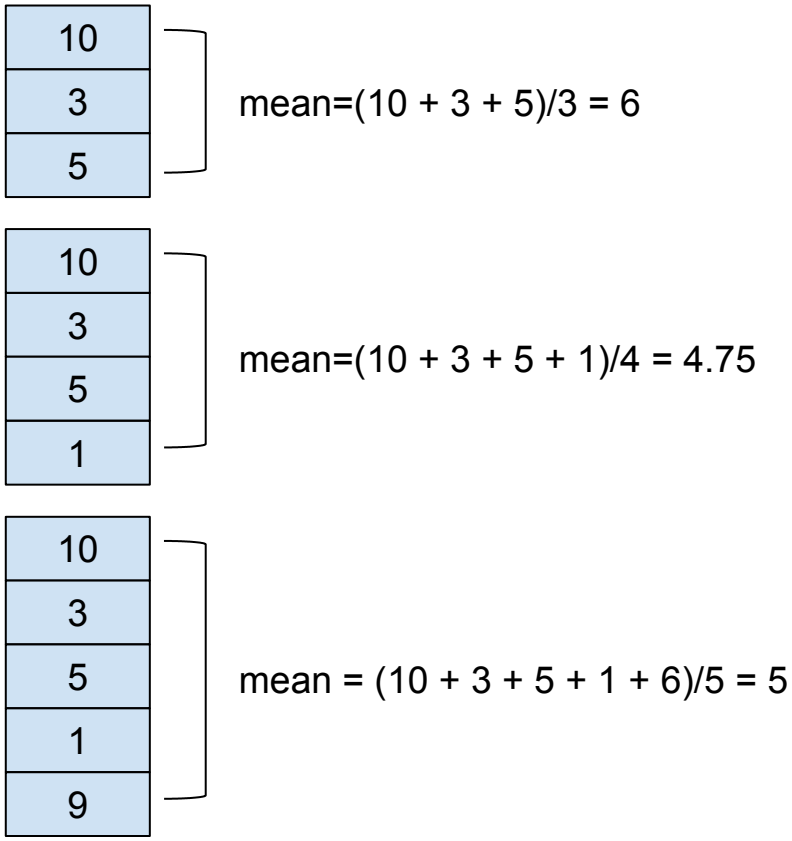


Рисунок 9.4 — Расчет среднего значения для расширяющего окна

Для работы с расширяющимся окном используется объект класса `Expanding`, который можно получить через метод `expanding()` структуры *pandas*.

Воспользуемся структурой *s*, созданной в предыдущем разделе:

```
>>> s[0:5]
0    28
1    48
2     8
3     4
4    21
dtype: int64
```

Создадим объект Expanding:

```
>>> ex = s.expanding()
>>> ex
Expanding [min_periods=1,center=False,axis=0]
```

Для расчета статистик используются функции, аналогичные тем, что применяются при работе с объектами Rolling (см. таблицу 1):

```
>>> pd.options.display.max_rows=20
```

Вычислим среднее значение для расширяющегося окна:

```
>>> ex.mean()
0      28.000000
1      38.000000
2      28.000000
3      22.000000
4      21.800000
5      21.333333
6      24.285714
7      26.750000
8      24.777778
9      26.300000
         ...
490    26.348269
491    26.298780
492    26.255578
493    26.248988
494    26.242424
495    26.197581
496    26.207243
497    26.238956
498    26.220441
499    26.252000
Length: 500, dtype: float64
```

Вычисление стандартного отклонения для расширяющегося окна:

```
>>> ex.std()
0           NaN
1     14.142136
2     20.000000
3     20.264912
4     17.555626
5     15.743782
6     16.357611
7     16.671190
8     16.679162
9     16.445533
        ...
490   14.573350
491   14.599827
492   14.616493
493   14.602396
494   14.588340
495   14.607777
496   14.594634
497   14.597109
498   14.588310
499   14.590760
Length: 500, dtype: float64
```

Подход к работе с расширяющимся окном аналогичен тому, что используется для скользящего окна.

9.3 Время-ориентированное скольжение

Суть время-ориентированного скольжения в том, что в качестве окна используется временной интервал.

Создадим *DataFrame*, в котором индексом будет временная метка:

```
>>> import pandas as pd
>>> df = pd.DataFrame([1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
index=pd.date_range('20180101 00:00:00', periods=10, freq='s'))
>>> df
                     0
2018-01-01 00:00:00   1
2018-01-01 00:00:01   2
2018-01-01 00:00:02   3
2018-01-01 00:00:03   4
2018-01-01 00:00:04   5
2018-01-01 00:00:05   6
2018-01-01 00:00:06   7
2018-01-01 00:00:07   8
2018-01-01 00:00:08   9
2018-01-01 00:00:09  10
```

Создадим объект `Rolling` с окном в 2 секунды:

```
>>> rt = df.rolling('2s')
>>> rt.sum()
                       0
2018-01-01 00:00:00   1.0
2018-01-01 00:00:01   3.0
2018-01-01 00:00:02   5.0
2018-01-01 00:00:03   7.0
2018-01-01 00:00:04   9.0
2018-01-01 00:00:05  11.0
2018-01-01 00:00:06  13.0
2018-01-01 00:00:07  15.0
2018-01-01 00:00:08  17.0
2018-01-01 00:00:09  19.0
```

Вот такой результат будет для окна в 5 секунд:

```
>>> rt = df.rolling('5s')
>>> rt.sum()
                        0
2018-01-01 00:00:00   1.0
2018-01-01 00:00:01   3.0
2018-01-01 00:00:02   6.0
2018-01-01 00:00:03  10.0
2018-01-01 00:00:04  15.0
2018-01-01 00:00:05  20.0
2018-01-01 00:00:06  25.0
2018-01-01 00:00:07  30.0
2018-01-01 00:00:08  35.0
2018-01-01 00:00:09  40.0
```

9.4 Агрегация данных

Функция агрегации позволяет одновременно производить расчет различных статистик для заданного набора данных. Для экспериментов возьмем структуру `df` из предыдущего параграфа:

```
>>> df
                     0
2018-01-01 00:00:00   1
2018-01-01 00:00:01   2
2018-01-01 00:00:02   3
2018-01-01 00:00:03   4
2018-01-01 00:00:04   5
2018-01-01 00:00:05   6
2018-01-01 00:00:06   7
2018-01-01 00:00:07   8
2018-01-01 00:00:08   9
2018-01-01 00:00:09  10
```

Создадим для него объект скольжение:

```
>>> rt = df.rolling(window='5s')
>>> rt
Rolling
[window=5000000000,min_periods=1,center=False,win_type=freq,axis=0]
```

Теперь посчитаем одновременно сумму, среднее значение и стандартное отклонение. Функции для расчета статистик возьмем из библиотеки numpy:

```
>>> import numpy as np
>>> rt.agg([np.sum, np.mean, np.std])
                        0
                      sum mean       std
2018-01-01 00:00:00   1.0  1.0       NaN
2018-01-01 00:00:01   3.0  1.5  0.707107
2018-01-01 00:00:02   6.0  2.0  1.000000
2018-01-01 00:00:03  10.0  2.5  1.290994
2018-01-01 00:00:04  15.0  3.0  1.581139
2018-01-01 00:00:05  20.0  4.0  1.581139
2018-01-01 00:00:06  25.0  5.0  1.581139
2018-01-01 00:00:07  30.0  6.0  1.581139
2018-01-01 00:00:08  35.0  7.0  1.581139
2018-01-01 00:00:09  40.0  8.0  1.581139
```

При необходимости, можно самостоятельно реализовать функции, рассчитывающие нужные статистики.

Глава 10. Временные ряды

Популярным направлением в области работы с данными является анализ временных рядов. Библиотека *pandas* содержит набор инструментов, который позволяет производить манипуляции с данными, представленными в виде временных рядов.

Для работы с временными рядами в *pandas* используются классы, представленные в таблице 10.1.

Таблица 10.1 - Классы pandas, используемые для работы с временными рядами

Класс	Создание	Описание
`Timestamp`	`to_datetime`, `Timestamp`	Единичная временная метка
`DatetimeIndex`	`to_datetime`, `date_range`, `bdate_range`, `DatetimeIndex`	Набор объектов `Timestamp`
`Period`	`Period`	Единичный временной интервал
`PeriodIndex`	`period_range`, `PeriodIndex`	Набор объектов `Period`

В *pandas* различают временные метки и временные интервалы. Временная метка - это конкретное значение даты/времени, например: 2018-01-01 01:20:35. Временной интервал предполагает наличие неполной временной метки и маркера, который определяет период, например метка 2018-01 будет иметь маркер `'M'`, что означает - месяц. `DatetimeIndex` - это класс, позволяющий хранить массив временных меток, которые могут быть использованы в качестве индексов при

построение структур данных *pandas*. `PeriodIndex` - класс, хранящий массив временных интервалов, который также может быть использован в качестве индекса.

10.1 Работа с временными метками

10.1.1 Создание временной метки

Для создания временной метки (объекта класса `Timestamp`) можно воспользоваться конструктором `Timestamp()`, либо методом `to_datetime()`.

Начнем со знакомства с конструктором `Timestamp`:

```
pandas.Timestamp(ts_input, freq, tz, unit, offset)
```

Ниже представлено описание аргументов.

- `ts_input`: `datetime`, `str`, `int`, `float`
  - Значение, которое будет преобразовано в объект класса `Timestamp`.
- `freq`: `str`, `DateOffset`
  - Величина сдвига.
- `tz`: `str`, `pytz.timezone`, `dateutil.tz.tzfile` или `None`; значение по умолчанию: `None`
  - Временная зона.
- `unit`: `str`; значение по умолчанию: `None`
  - Единица измерения, параметр используется если значение `ts_input` имеет тип `int` или `float`.
- `offset`: `str`, `DateOffset`: значение по умолчанию: `None`
  - Не поддерживается, используйте `freq`.

Рассмотрим варианты создания объекта `Timestamp`.

Создание объекта из строки:

```
>>> ts = pd.Timestamp('2018-10-5')
>>> ts
Timestamp('2018-10-05 00:00:00')
>>> ts = pd.Timestamp('2018-10-5 01:15:33')
>>> ts
Timestamp('2018-10-05 01:15:33')
```

Использование объекта `datetime`:

```
>>> from datetime import datetime
>>> dt = datetime.now()
>>> dt
datetime.datetime(2019, 12, 2, 21, 33, 48, 621911)
>>> ts = pd.Timestamp(dt)
>>> ts
Timestamp('2019-12-02 21:33:48.621911')
```

Использование `int` и `float` значений:

```
>>> ts = pd.Timestamp(1517246359, unit='s')
>>> ts
Timestamp('2018-01-29 17:19:19')
>>> ts = pd.Timestamp(1517246359.732405, unit='s')
>>> ts
Timestamp('2018-01-29 17:19:19.732404947')
```

Для создания объектов `Timestamp` можно использовать метод `to_datetime()`. Это мощный инструмент, который обладает широким функционалом, мы не будем рассматривать все его возможности, наша задача - просто познакомиться с ним:

```
>>> ts = pd.to_datetime('2018-01-01 00:01:02')
```

```
>>> ts
Timestamp('2018-01-01 00:01:02')

>>> ts = pd.to_datetime('20180101023215', format='%Y%m%d%H%M%S')
>>> ts
Timestamp('2018-01-01 02:32:15')

>>> ts = pd.to_datetime(1517246359, unit='s')
>>> ts
Timestamp('2018-01-29 17:19:19')
```

Обратите внимание, что при использовании `Timestamp` можно оперировать с временными данным в следующем диапазоне:

```
>>> pd.Timestamp.min
Timestamp('1677-09-21 00:12:43.145225')
>>> pd.Timestamp.max
Timestamp('2262-04-11 23:47:16.854775807')
```

10.1.2 Создание ряда временных меток

Ряд временных меток - это объект класса `DatetimeIndex`, который может выступать в качестве индекса при создании структур *Series* и *DataFrame*. Объект такого класса можно создать с помощью уже известного нам метода `to_datetime()`, либо `date_range()`.

Для начала рассмотрим вариант работы с `to_datetime()`. Если в качестве аргумента передать данному методу список строк, чисел (`int` или `float`) или объектов `Timestamp`, то в результате будет создан соответствующий объект `DatetimeIndex`:

```
>>> dti = pd.to_datetime(['2018-01-01', '2018-01-02'])
>>> dti
DatetimeIndex(['2018-01-01', '2018-01-02'], dtype='datetime64[ns]',
freq=None)
```

Можно использовать список объектов Timestamp:

```
>>> dti = pd.to_datetime([pd.Timestamp('2018-01-01'), pd.Timestamp('2018-01-02')])
>>> dti
DatetimeIndex(['2018-01-01', '2018-01-02'], dtype='datetime64[ns]', freq=None)

>>> dti = pd.to_datetime([1517246359, 1517246360, 1517246361], unit='s')
>>> dti
DatetimeIndex(['2018-01-29 17:19:19', '2018-01-29 17:19:20',
               '2018-01-29 17:19:21'],
              dtype='datetime64[ns]', freq=None)
```

Для создания временного ряда из заданного диапазона применяется метод date_range():

```
pandas.date_range(start=None, end=None, periods=None, freq='D', tz=None, normalize=False, name=None, closed=None, **kwargs)
```

Среди аргументов данного метода выделим следующие:

- start: str; значение по умолчанию: None
  - Левая граница генерируемых данных.
- end: str; значение по умолчанию: None
  - Правая граница генерируемых данных.
- period: integer; значение по умолчанию: None
  - Количество элементов в создаваемом массиве.
- freq: str(или DateOffset); значение по умолчанию: 'D'
  - Шаг, с которым будут генерироваться данные.
- tz: str; значение по умолчанию: None
  - Временная зона, например: “Europe/Brussels”.

Метод `date_range()` возвращает объект класса `DatetimeIndex`, который можно использовать в качестве индекса при построении структур данных *pandas*.

Ниже, в таблице 10.2, приведены некоторые из возможных значений параметра `freq`.

Таблица 10.2 - Возможные значения параметра `freq`, определяющего шаг при создании диапазона методом `date_range()`

Значение параметра `freq`	**Описание**
`D`	День
`W`	Неделя
`M`	Месяц
`SM`	Половина месяца (15 дней)
`Q`	Квартал
`A, Y`	Год
`H`	Час
`T, min`	Минута
`S`	Секунда
`L, ms`	Миллисекунда
`U, us`	Микросекунда
`N`	Наносекунда

Создадим массив временных меток с шагом в один час, который охватывает интервал в пять дней:

```
>>> dt_h = pd.date_range(start='2017-02-01', freq='H', periods=120)
>>> dt_h[:30]
DatetimeIndex(['2017-02-01 00:00:00', '2017-02-01 01:00:00',
               '2017-02-01 02:00:00', '2017-02-01 03:00:00',
               '2017-02-01 04:00:00', '2017-02-01 05:00:00',
               '2017-02-01 06:00:00', '2017-02-01 07:00:00',
               '2017-02-01 08:00:00', '2017-02-01 09:00:00',
               '2017-02-01 10:00:00', '2017-02-01 11:00:00',
               '2017-02-01 12:00:00', '2017-02-01 13:00:00',
               '2017-02-01 14:00:00', '2017-02-01 15:00:00',
               '2017-02-01 16:00:00', '2017-02-01 17:00:00',
               '2017-02-01 18:00:00', '2017-02-01 19:00:00',
               '2017-02-01 20:00:00', '2017-02-01 21:00:00',
               '2017-02-01 22:00:00', '2017-02-01 23:00:00',
               '2017-02-02 00:00:00', '2017-02-02 01:00:00',
               '2017-02-02 02:00:00', '2017-02-02 03:00:00',
               '2017-02-02 04:00:00', '2017-02-02 05:00:00'],
              dtype='datetime64[ns]', freq='H')
```

Таким же образом можно создать DatetimeIndex, содержащий временные метки с шагом в одну минуту и общим интервалом в один час:

```
>>> dt_m = pd.date_range(start='2017-02-01', freq='min', periods=60)
>>> dt_m[:30]
DatetimeIndex(['2017-02-01 00:00:00', '2017-02-01 00:01:00',
               '2017-02-01 00:02:00', '2017-02-01 00:03:00',
               '2017-02-01 00:04:00', '2017-02-01 00:05:00',
               '2017-02-01 00:06:00', '2017-02-01 00:07:00',
               '2017-02-01 00:08:00', '2017-02-01 00:09:00',
               '2017-02-01 00:10:00', '2017-02-01 00:11:00',
               '2017-02-01 00:12:00', '2017-02-01 00:13:00',
```

```
               '2017-02-01 00:14:00', '2017-02-01 00:15:00',
               '2017-02-01 00:16:00', '2017-02-01 00:17:00',
               '2017-02-01 00:18:00', '2017-02-01 00:19:00',
               '2017-02-01 00:20:00', '2017-02-01 00:21:00',
               '2017-02-01 00:22:00', '2017-02-01 00:23:00',
               '2017-02-01 00:24:00', '2017-02-01 00:25:00',
               '2017-02-01 00:26:00', '2017-02-01 00:27:00',
               '2017-02-01 00:28:00', '2017-02-01 00:29:00'],
              dtype='datetime64[ns]', freq='T')

>>> len(dt_m)
60
>>> len(dt_h)
120
```

Следующим нашим шагом будет создание структуры *Series*, в которой, в качестве индексов, будет использоваться первый, созданный нами DatetimeIndex:

```
>>> import random
>>> rnd = [random.randint(-5, 5) for i in range(len(dt_h))]
>>> s_dt = pd.Series(rnd, index=dt_h)
>>> pd.options.display.max_rows = 20
>>> s_dt
2017-02-01 00:00:00   -3
2017-02-01 01:00:00    4
2017-02-01 02:00:00    1
2017-02-01 03:00:00    2
2017-02-01 04:00:00    1
2017-02-01 05:00:00   -2
2017-02-01 06:00:00   -1
2017-02-01 07:00:00    0
2017-02-01 08:00:00    5
2017-02-01 09:00:00    0
```

```
                     ..
2017-02-05 14:00:00    1
2017-02-05 15:00:00   -4
2017-02-05 16:00:00   -3
2017-02-05 17:00:00    3
2017-02-05 18:00:00   -4
2017-02-05 19:00:00   -3
2017-02-05 20:00:00    2
2017-02-05 21:00:00    2
2017-02-05 22:00:00    2
2017-02-05 23:00:00   -3
Freq: H, Length: 120, dtype: int64
```

10.2 Работа с временными интервалами

10.2.1 Создание временного интервала

Единичный временной интервал - это объект класса `Period`. Его можно создать, используя одноименный конструктор, основными параметрами которого являются `value` и `freq`:

- `value`: `str`; значение по умолчанию: `None`
  - Временной период.
- `freq`: `str`; значение по умолчанию: `None`
  - Строка с меткой, определяющей временной интервал.

Ниже приведены примеры того, как можно конструировать объекты *Period*:

```
>>> pd.Period('2018')
Period('2018', 'A-DEC')
>>> pd.Period('2018-01')
Period('2018-01', 'M')
>>> pd.Period('2018-01-01')
Period('2018-01-01', 'D')
```

```
>>> pd.Period('2018', freq='M')
Period('2018-01', 'M')
```

После того, как создан объект класса `Period` с ним можно производить различные арифметические действия. Например: создадим объект с интервалом в день:

```
>>> prd = pd.Period('2018', freq='D')
>>> prd
Period('2018-01-01', 'D')
```

Теперь прибавим к нему число 7, что будет означать прибавление семи дней к текущей дате:

```
>>> prd + 7
Period('2018-01-08', 'D')
```

Если прибавить число большее 31, то увидим, что изменился месяц:

```
>>> prd + 53
Period('2018-02-23', 'D')
```

Точно также, если мы создадим объект, в котором интервалом будет месяц, то арифметические операции будут производится над месяцами относительно указанной даты:

```
>>> prd_m = pd.Period('2018', freq='M')
>>> prd_m
Period('2018-01', 'M')
>>> prd_m + 5
Period('2018-06', 'M')
```

10.2.2 Создание ряда временных интервалов

Массивы временных интервалов могут использоваться при построении структур данных *pandas*. Для работы с такими массивами существует специальный класс - `PeriodIndex`.

Создание объекта такого класса осуществляется посредством конструктора `PeriodIndex()` или метода `period_range()`.

Работа с конструктором `PeriodIndex` аналогична работе с конструктором `Period`, только в качестве `value` мы должны передать список временных меток:

```
>>> pd.PeriodIndex(['2018', '2017', '2016'], freq='M')
PeriodIndex(['2018-01', '2017-01', '2016-01'], dtype='period[M]',
freq='M')

>>> pd.PeriodIndex(['2018', '2017', '2016'], freq='D')
PeriodIndex(['2018-01-01', '2017-01-01', '2016-01-01'],
dtype='period[D]', freq='D')
```

Более удобным инструментом для создания рядов временных интервалов является метод `period_range()`, который, по принципу работы с ним, похож на `date_range()` из раздела “Создание ряда временных меток”:

```
pandas.period_range(start=None, end=None, periods=None, freq='D',
name=None):
```

- `start`: `str`; значение по умолчанию: `None`
  - Левая граница генерируемых данных.
- `end`: `str`; значение по умолчанию: `None`
  - Правая граница генерируемых данных.

- `period`: `integer`; значение по умолчанию: `None`
  - Количество элементов в массиве.
- `freq`: `str` (или `DateOffset`); значение по умолчанию: `'D'`
  - Шаг, с которым будут генерироваться данные.
- `name`: `str`; значение по умолчанию: `None`
  - Имя объекта `PeriodIndex`.

Рассмотрим несколько примеров использования данного метода. Ряд временных интервалов в диапазоне между 2018 и 2019-м годом с шагом в один месяц:

```
>>> pd.period_range('2018', '2019', freq='M')
PeriodIndex(['2018-01', '2018-02', '2018-03', '2018-04', '2018-05',
'2018-06', '2018-07', '2018-08', '2018-09', '2018-10', '2018-11', '2018-
12', '2019-01'], dtype='period[M]', freq='M')
```

Ряд временных интервалов в диапазоне между 2018 и 2019-м годом с шагом в один день:

```
>>> pd.period_range('2018', '2019', freq='D')
PeriodIndex(['2018-01-01', '2018-01-02', '2018-01-03', '2018-01-04',
             '2018-01-05', '2018-01-06', '2018-01-07', '2018-01-08',
             '2018-01-09', '2018-01-10',
             ...
             '2018-12-23', '2018-12-24', '2018-12-25', '2018-12-26',
             '2018-12-27', '2018-12-28', '2018-12-29', '2018-12-30',
             '2018-12-31', '2019-01-01'],
            dtype='period[D]', length=366, freq='D')
```

Начиная с 2018.01.01 построить десять временных интервалов с шагом в неделю:

```
>>> pd.period_range('2018-01-01', periods=10, freq='W')
PeriodIndex(['2018-01-01/2018-01-07', '2018-01-08/2018-01-14',
             '2018-01-15/2018-01-21', '2018-01-22/2018-01-28',
             '2018-01-29/2018-02-04', '2018-02-05/2018-02-11',
             '2018-02-12/2018-02-18', '2018-02-19/2018-02-25',
             '2018-02-26/2018-03-04', '2018-03-05/2018-03-11'],
            dtype='period[W-SUN]', freq='W-SUN')
```

10.3 Использование временных рядов в качестве индексов

Объекты классов `Timestamp`, `DatetimeIndex`, `Period`, `PeriodIndex` могут использоваться в качестве индексов структур данных *pandas*.

Создадим `DatetimeIndex` - ряд временных меток с отсечкой в один день:

```
>>> dt_d = pd.date_range(start='2017-02-01', freq='D', periods=50)
>>> dt_d
DatetimeIndex(['2017-02-01', '2017-02-02', '2017-02-03', '2017-02-04',
               '2017-02-05', '2017-02-06', '2017-02-07', '2017-02-08',
               '2017-02-09', '2017-02-10', '2017-02-11', '2017-02-12',
               '2017-02-13', '2017-02-14', '2017-02-15', '2017-02-16',
               '2017-02-17', '2017-02-18', '2017-02-19', '2017-02-20',
               '2017-02-21', '2017-02-22', '2017-02-23', '2017-02-24',
               '2017-02-25', '2017-02-26', '2017-02-27', '2017-02-28',
               '2017-03-01', '2017-03-02', '2017-03-03', '2017-03-04',
               '2017-03-05', '2017-03-06', '2017-03-07', '2017-03-08',
               '2017-03-09', '2017-03-10', '2017-03-11', '2017-03-12',
               '2017-03-13', '2017-03-14', '2017-03-15', '2017-03-16',
               '2017-03-17', '2017-03-18', '2017-03-19', '2017-03-20',
               '2017-03-21', '2017-03-22'],
              dtype='datetime64[ns]', freq='D')
```

На базе полученного объекта создадим структуру *Series*:

```
>>> import random
>>> dr = [random.randint(-10, 10) for i in range(len(dt_d))]
>>> s = pd.Series(dr, index=dt_d)
```

Для получения доступа к структуре можно использовать числовые индексы:

```
>>> s[:5]
2017-02-01    10
2017-02-02    -7
2017-02-03    -5
2017-02-04    -4
2017-02-05     5
Freq: D, dtype: int64
```

Либо метку времени в текстовом виде:

```
>>> s['2017-02-01']
10
```

Для получения данных из заданного временного диапазона допускается использование срезов:

```
>>> s['2017-02-01':'2017-02-10']
2017-02-01    10
2017-02-02    -7
2017-02-03    -5
2017-02-04    -4
2017-02-05     5
2017-02-06     5
2017-02-07    -6
2017-02-08     8
2017-02-09    -8
2017-02-10     3
Freq: D, dtype: int64
```

Использование временных рядов в качестве индексов, предоставляет дополнительные возможности по выборке данных. Получим данные за февраль 2017 года из созданной выше структуры s:

```
>>> pd.options.display.max_rows=10
>>> s['2017-02']
2017-02-01    10
2017-02-02    -7
2017-02-03    -5
2017-02-04    -4
2017-02-05     5
              ..
2017-02-24    -1
2017-02-25   -10
2017-02-26     5
2017-02-27    -4
2017-02-28     4
Freq: D, Length: 28, dtype: int64
```

Вместо строкового представления даты допускается применять объекты datetime:

```
>>> from datetime import datetime
>>> s[datetime(2017,2,1):datetime(2017,2,8)]
2017-02-01    10
2017-02-02    -7
2017-02-03    -5
2017-02-04    -4
2017-02-05     5
2017-02-06     5
2017-02-07    -6
2017-02-08     8
Freq: D, dtype: int64
```

Глава 11. Визуализация данных

11.1 Построение графиков

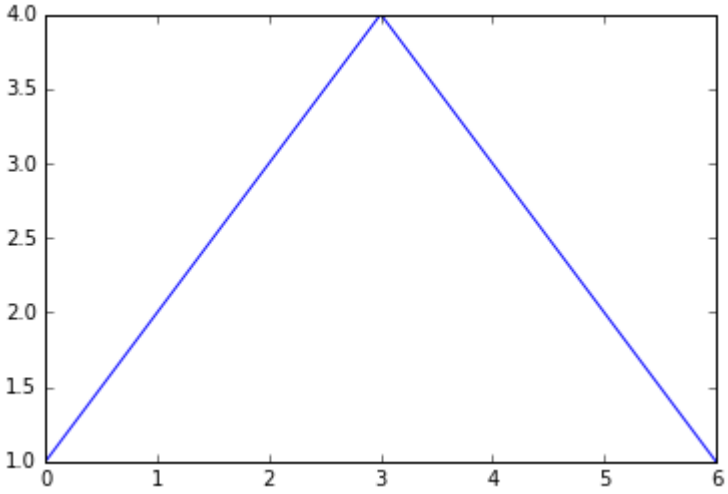
Основным инструментом для визуализации данных в библиотеке *pandas* является метод `plot()`, который может быть вызван у объекта структуры *Series* или *DataFrame*. Если использовать этот метод без параметров, то будет построен линейный график. За вид графика отвечает аргумент `kind`. В зависимости от его значения, будет меняться форма графического представления данных. Возможные значения данного параметра представлены в таблице 11.1.

Таблица 11.1 - Описание значений параметра `kind`

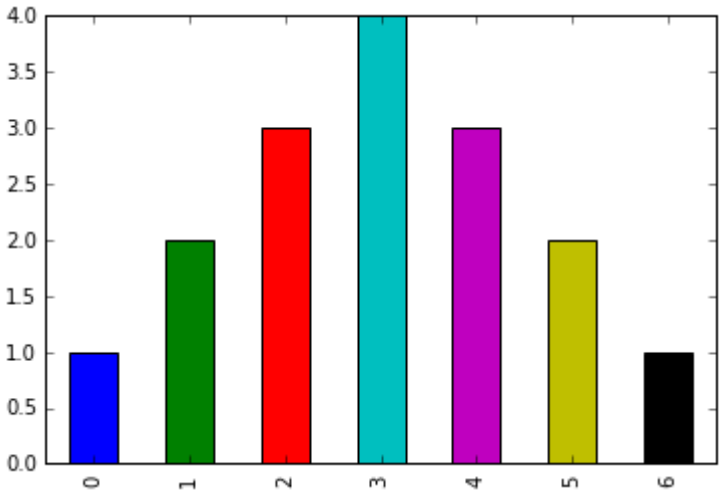
Значение параметра `kind`	**Описание**
`'bar'` или `'barh'`	Построение столбцовой диаграммы
`'hist'`	Построение гистограмм
`'box'`	Коробчатая диаграмма (ящик с усами)
`'kde'`	Построение графика плотности
`'area'`	Диаграмма с областями
`'scatter'`	Точечный график
`'hexbin'`	Визуализация данных с использованием шестиугольников
`'pie'`	Круговая диаграмма

Покажем на примере работу описанного выше подхода:

```
>>> import pandas as pd
>>> s = pd.Series([1, 2, 3, 4, 3, 2, 1])
>>> s.plot()
```



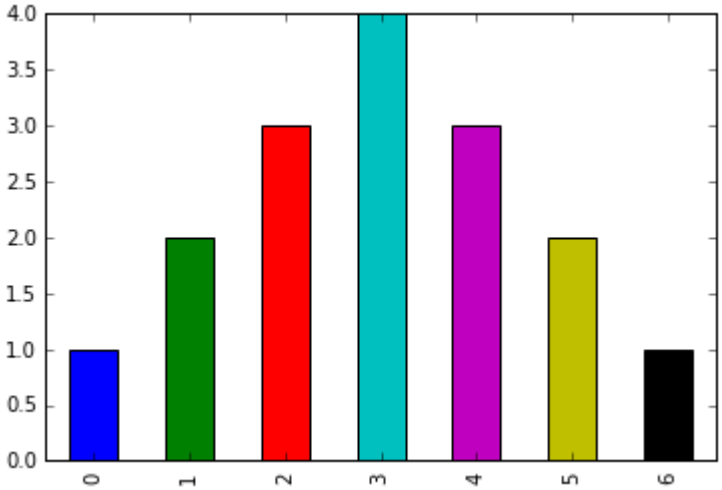

```
>>> s.plot(kind='bar')
```



Pandas предоставляет еще один способ построения диаграмм. Суть его заключается в том, что мы не используем аргумент `kind` метода `plot()`, а вызываем специальный метод для построения нужной нам диаграммы в формате `<Структура pandas>.plot.<метод построения диаграммы>`.

Для столбчатой диаграммы это будет выглядеть так:

```
>>> s.plot.bar()
```



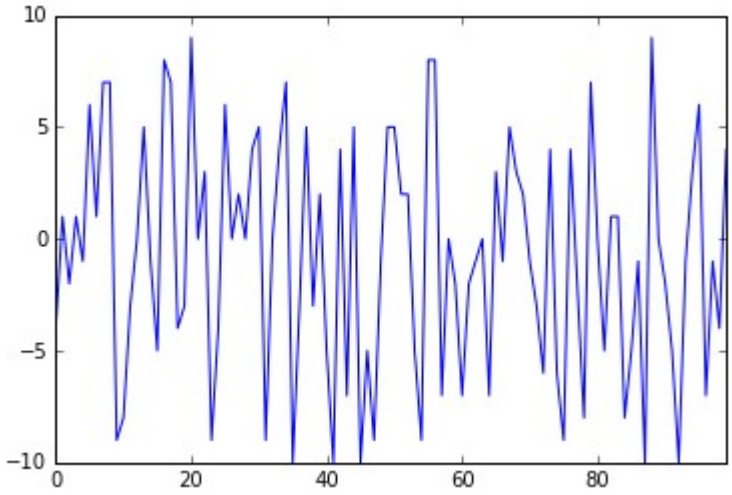
11.1.1 Линейные графики

Познакомимся поближе с построением линейных графиков. Для этого подготовим набор данных:

```
>>> import random
>>> rnd = [random.randrange(-10, 10) for i in range(100)]
>>> s = pd.Series(rnd)
```

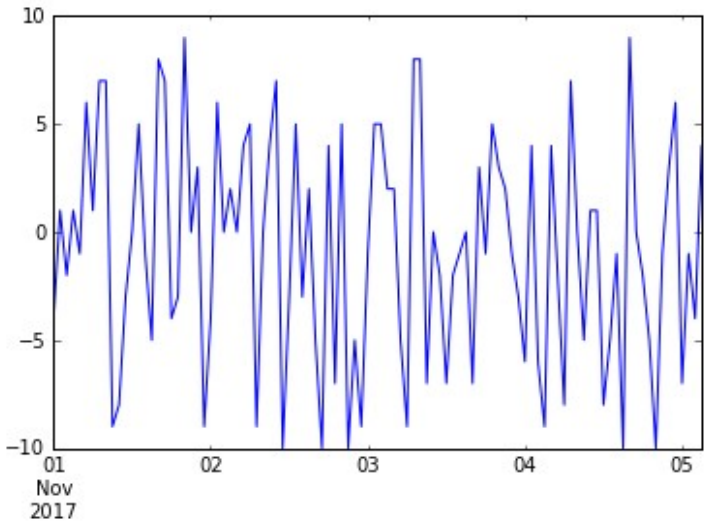
Построим график:

```
>>> s.plot()
```



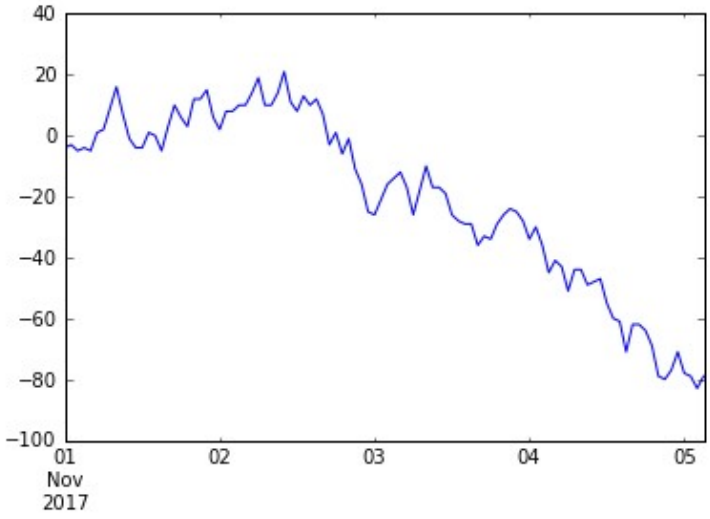
Если в качестве индекса использовать DatetimeIndex, то по оси абсцисс будут выставлены соответствующие временные метки:

```
>>> datetime_index= pd.date_range('2017.11.01', freq='H',
periods=len(rnd))
>>> s_dt = pd.Series(rnd, index=datetime_index)
>>> s_dt.plot()
```



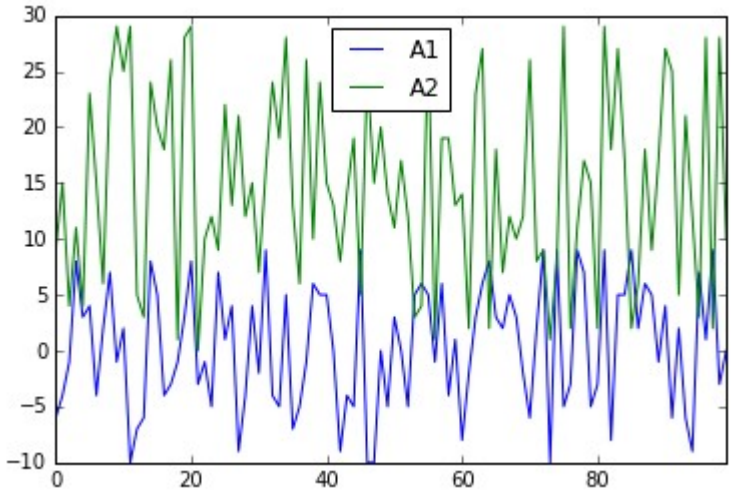
Удобно строить графики для вычисляемых статистик, в качестве примера возьмем сумму расширяющегося окна:

```
>>> ex = s_dt.expanding()
>>> ex.sum().plot()
```



Если вы работаете с *DataFrame*, то на одном поле можно отобразить сразу несколько графиков, каждый из которых будет соответствовать столбцу структуры:

```
>>> d_arr = [[random.randrange(-10, 10), random.randrange(0, 30)] for i
in range(100)]
>>> df = pd.DataFrame(d_arr, columns=['A1', 'A2'])
>>> df.plot()
```

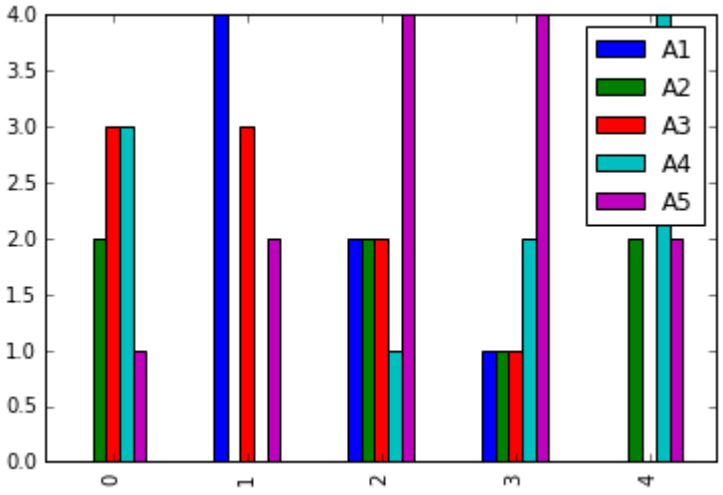


11.1.2 Столбчатые диаграммы

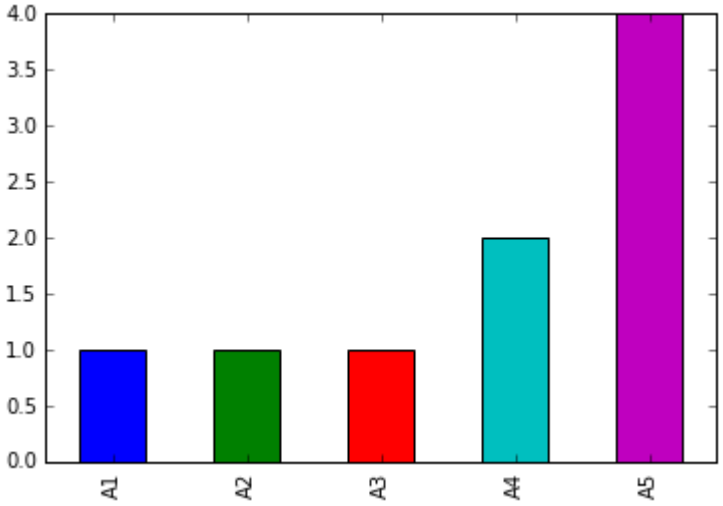
Для сравнения данных хорошо подходят столбчатые диаграммы (бары), при условии, что они (данные) подготовлены соответствующим образом. В качестве примера приведем визуализацию `DataFrame`*'а*. Для того, чтобы создать `DataFrame` размера 10x3 воспользуемся методом из библиотеки `numpy`, которую предварительного нужно импортировать (и установить, если ее еще нет на вашем компьютере):

```
>>> import numpy as np
>>> df1 = pd.DataFrame(np.random.randint(5, size=(5,5)), columns=['A1',
'A2', 'A3', 'A4', 'A5'])
```

```
>>> df1
   A1  A2  A3  A4  A5
0   1   4   0   2   1
1   4   1   1   0   4
2   2   3   4   3   3
3   2   0   1   3   0
4   4   2   4   3   4
>>> df1.plot(kind='bar')
```

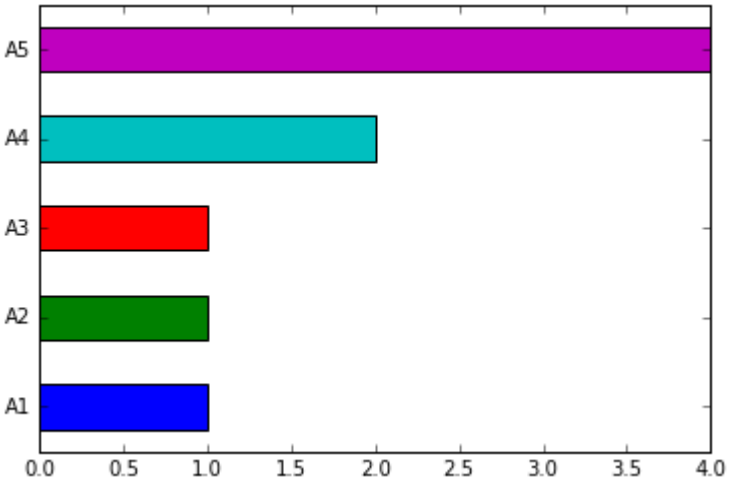



```
>>> df1.loc[3].plot(kind='bar')
```

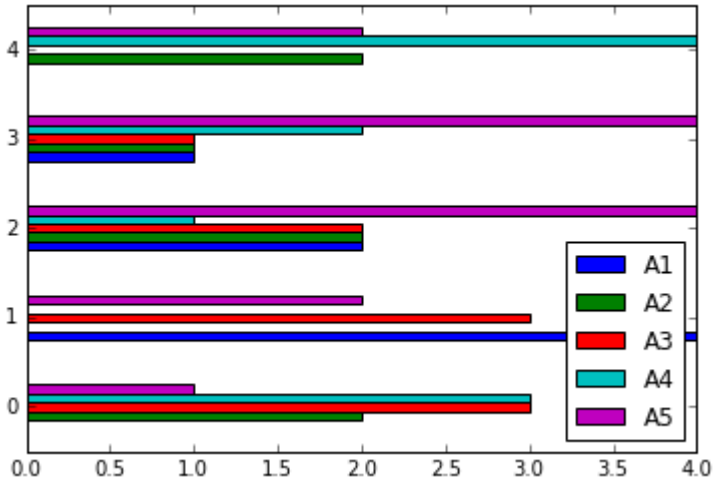


Для построения горизонтальной столбчатой диаграммы аргументу `kind` нужно присвоить значение `'barh'`:

```
>>> df1.loc[3].plot(kind='barh')
```

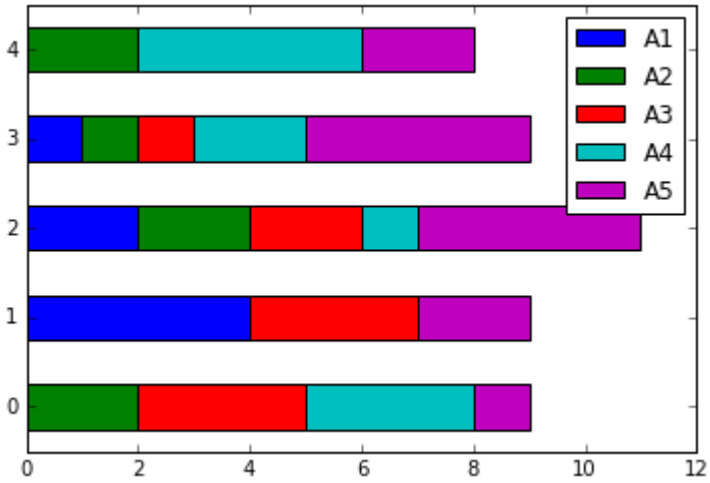



```
>>> df1.plot(kind='barh')
```



Для стекового представления данных, используйте параметр `stacked=True` метода `plot()`:

```
>>> df1.plot(kind='barh', stacked=True)
```



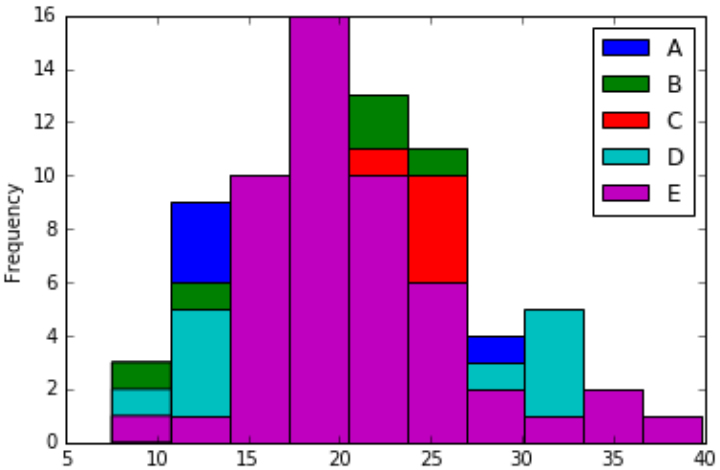
11.1.3 Гистограммы

Для представления возможностей *pandas* по работе с гистограммами создадим `DataFrame` размера 50x5, содержащий числа от 0 до 20 с распределением хи-квадрат:

```
>>> df2 = pd.DataFrame(np.random.chisquare(20, size=(50,5)),
columns=['A', 'B', 'C', 'D', 'E'])
```

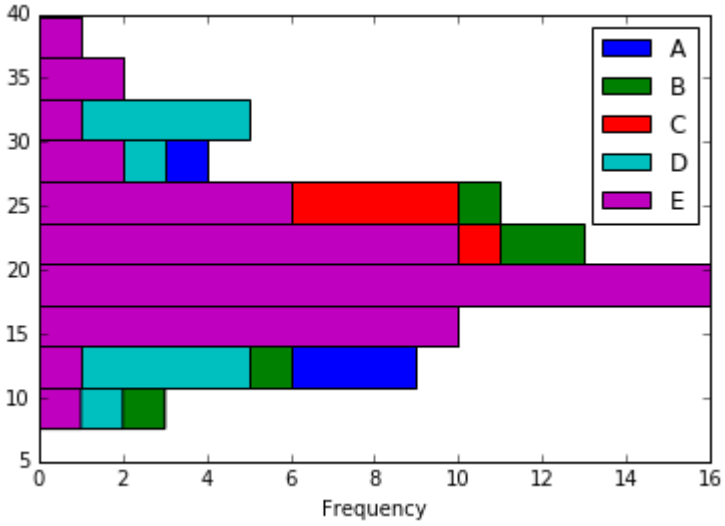
Построим гистограмму, используя второй из рассмотренных нами в начале главы подходов:

```
>>> df2.plot.hist()
```



Для смены ориентации диаграммы необходимо в метод `hist()` передать параметр `orientation='horizontal'`:

```
>>> df2.plot.hist(orientation='horizontal')
```

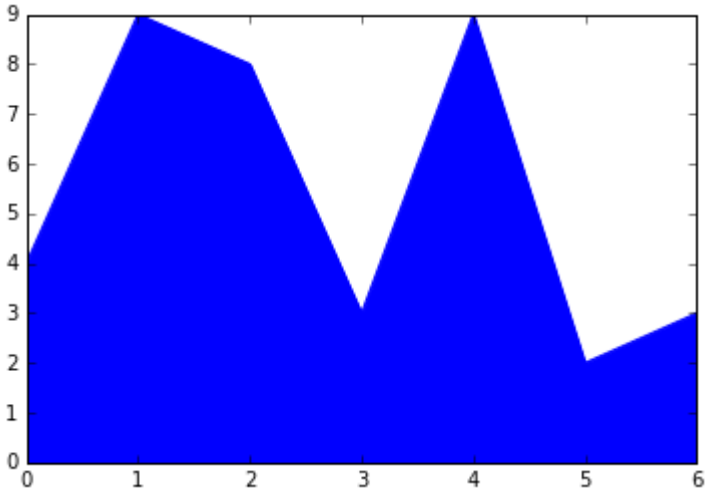


11.1.4 График с заливкой

График с заливкой - это вольный перевод термина *area plot*, выглядит он как обычный график, у которого залито пространство под кривой.

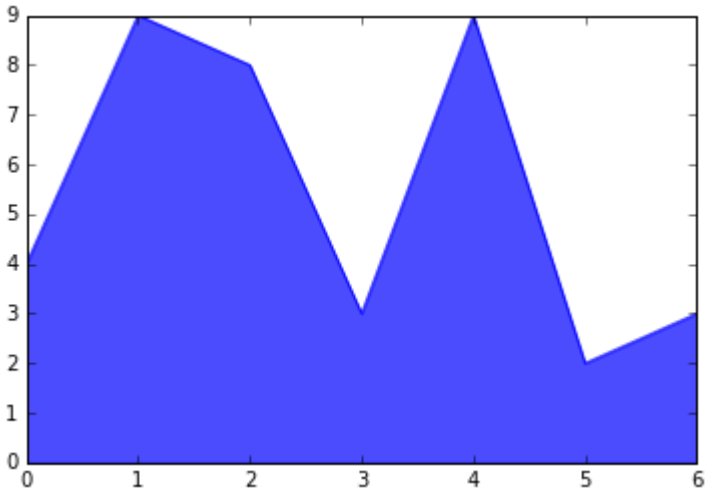
Создадим *Series* со случайными данными и построим график с заливкой:

```
>>> s = pd.Series([random.randrange(2, 10) for i in range(7)])
>>> s.plot.area()
```



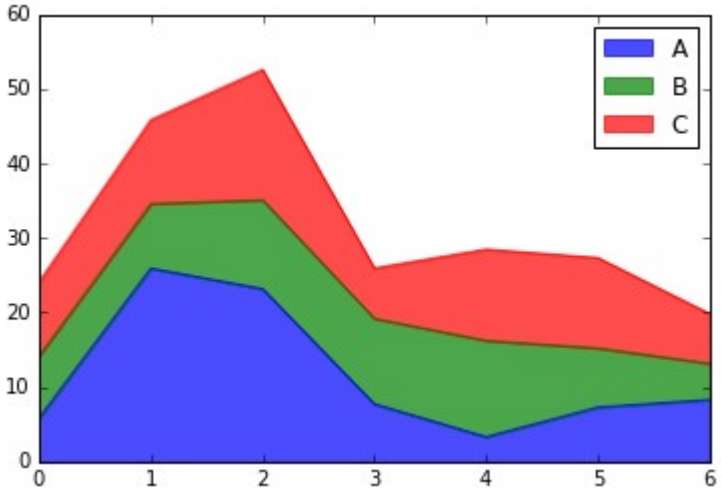
Можно изменить прозрачность заливки через параметр `alpha`:

```
>>> s.plot.area(alpha=0.7)
```

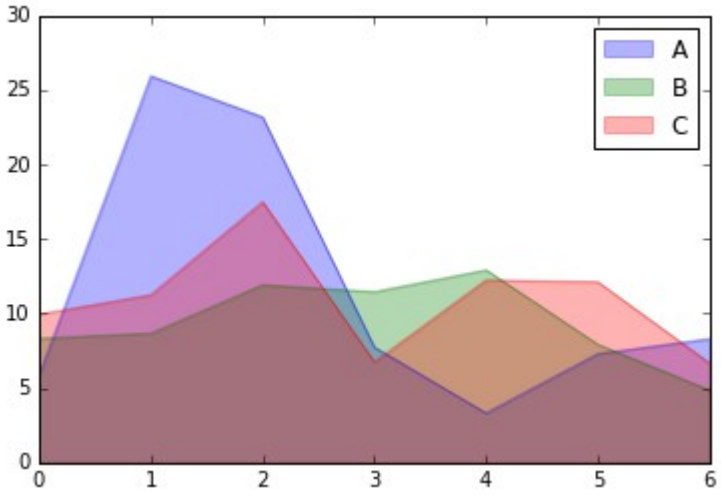


Посмотрим на работу данного инструмента с *DataFrame*:

```
>>> df3 = pd.DataFrame(np.random.chisquare(10, size=(7,3)), columns=['A',
'B', 'C'])
>>> df3.plot.area(alpha=0.7)
```



```
>>> df3.plot.area(stacked=False, alpha=0.3)
```

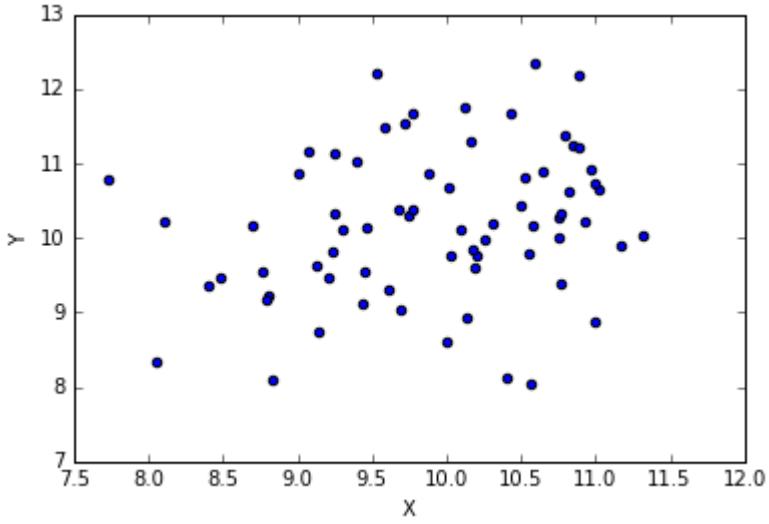


11.1.5 Точечный график

Точечный график можно получить, присвоив аргументу `kind` значение `'scatter'`, либо воспользоваться методом `scatter()`.

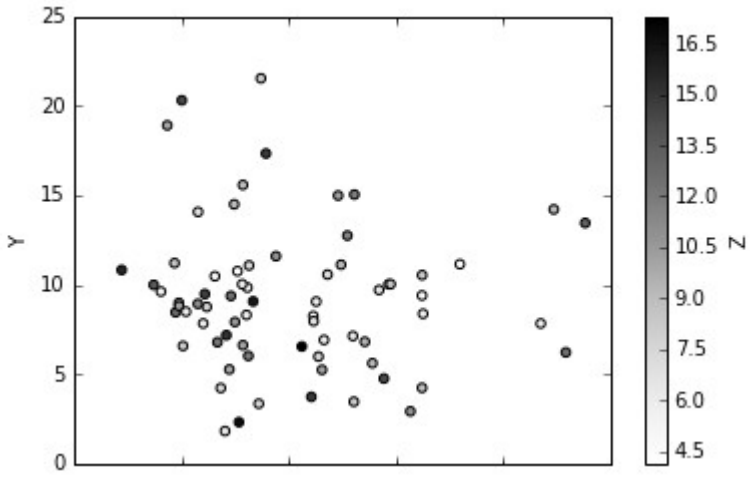
Для демонстрации модифицируем структуру *DataFrame* из предыдущего примера:

```
>>> df4 = pd.DataFrame(np.random.normal(10, size=(70,2)), columns=['X', 'Y'])
>>> df4.plot.scatter(x='X', y='Y')
```



Добавим еще один столбец, который будет определять цвет (интенсивность) точки:

```
>>> df5 = pd.DataFrame(np.random.chisquare(10, size=(70,3)),
columns=['X', 'Y', 'Z'])
>>> df5.plot.scatter(x='X', y='Y', c='Z')
```



11.1.6 Круговая диаграмма

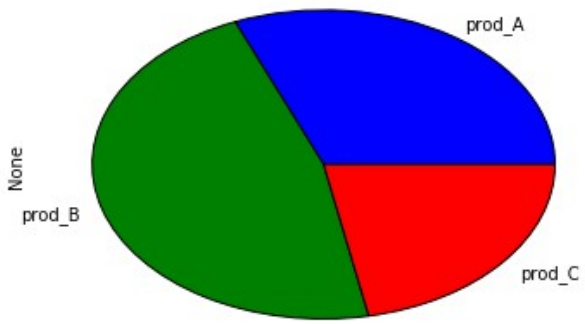
Круговая диаграмма позволяет наглядно представить количественные соотношения между элементами конкретного набора данных.

Создадим структуру, элементами которой будет количество продукции:

```
>>> s = pd.Series([10, 15, 7], index=['prod_A', 'prod_B', 'prod_C'])
```

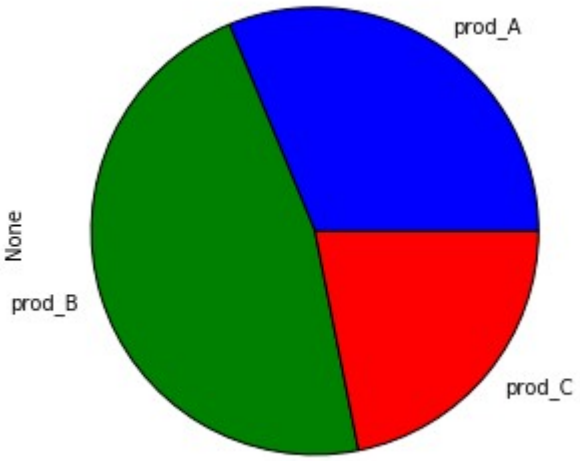
Построим круговую диаграмму для данного набора:

```
>>> s.plot.pie()
```



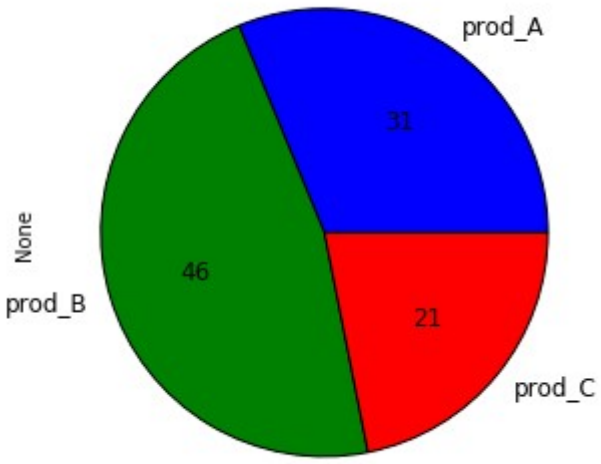
Размер диаграммы можно задавать через параметр `figsize`:

```
>>> s.plot.pie(figsize=(5,5))
```



В секторах диаграммы можно дополнительно выводить численные значения, соответствующие размеру сектора:

```
>>> s.plot.pie(figsize=(5,5), autopct='%d', fontsize=12)
```



Для вывода имени диаграммы его (имя) нужно предварительно присвоить соответствующей *pandas* структуре. Это можно сделать при создании структуры, задав нужное значение аргументу:

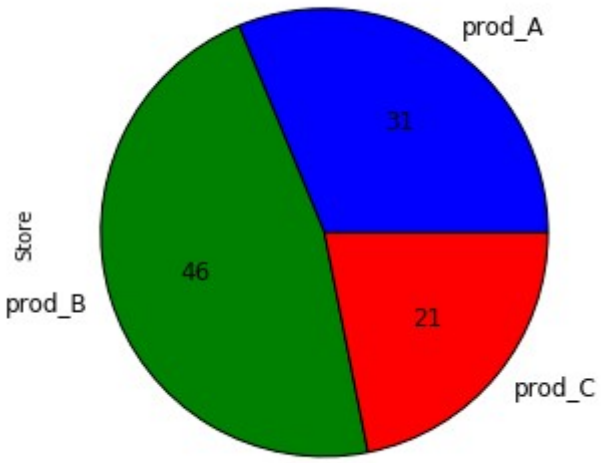
```
>>> s = pd.Series([10, 15, 7], index=['prod_A', 'prod_B', 'prod_C'],
name='Store')
```

Либо присвоить его уже созданной структуре:

```
>>> s.name = 'Store'
```

Данное имя будет указано рядом с построенной диаграммой:

```
>>> s.plot.pie(figsize=(5,5), autopct='%d', fontsize=12)
```



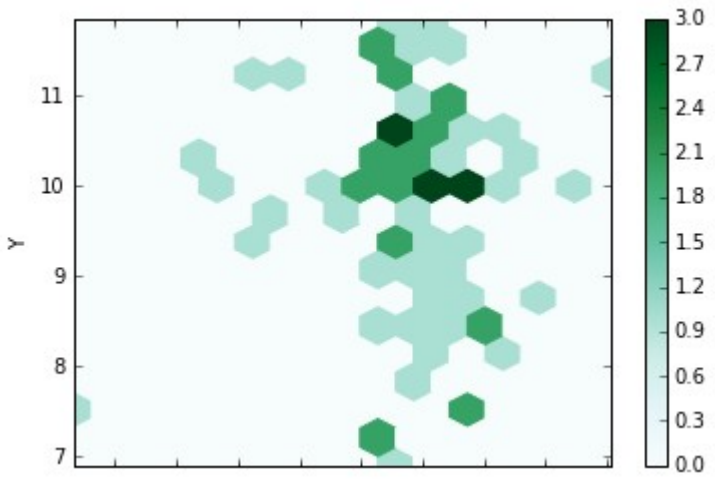
11.1.7 Диаграмма из шестиугольников

Довольно интересной является диаграмма из шестиугольников. Для демонстрации создадим дополнительный *DataFrame*:

```
>>> df6 = pd.DataFrame(np.random.laplace(10, size=(70,3)), columns=['X',
'Y', 'Z'])
```

Построим соответствующую диаграмму:

```
>>> df6.plot.hexbin(x='X', y='Y', gridsize=15)
```



Размер шестиугольника задается через параметр `gridsize`.

11.2 Настройка внешнего вида диаграммы

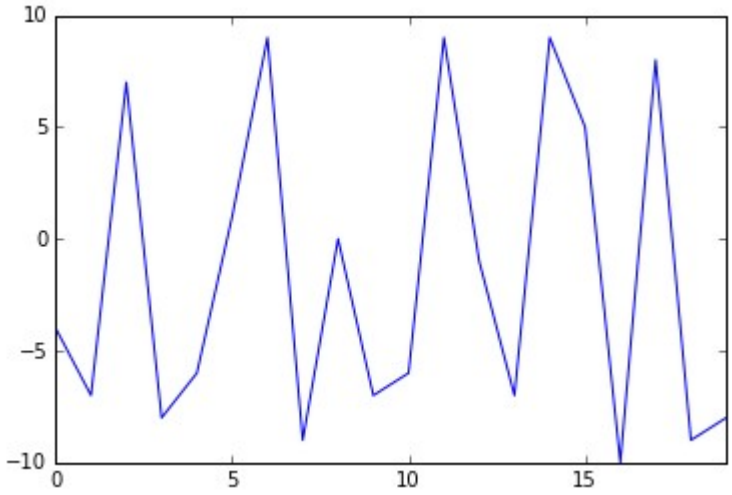
При работе с диаграммами в *pandas* в нашем распоряжении оказывается довольно большое количество инструментов для настройки внешнего вида непосредственно самой диаграммы и дополнительных выводимых элементов (таблицы, легенда и т.п.).

В предыдущем разделе был дан обзор различных видов диаграмм, которые можно построить. Для того, чтобы изменить внешний вид диаграммы - цвет, тип линии и т.п., необходимо передать соответствующее значение нужному свойству через аргумент.

11.2.1 Настройка внешнего вида линейного графика

Если оставить у параметров значения по умолчанию, то линейный график будет выглядеть следующим образом:

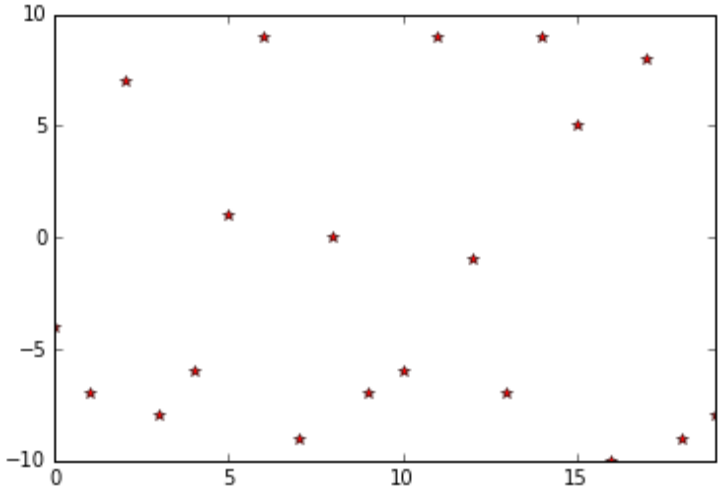
```
>>> rnd = [random.randrange(-10, 10) for i in range(20)]
>>> s = pd.Series(rnd)
>>> s.plot()
```



Для того, чтобы изменить внешний вид графика необходимо аргументу `style` метода `plot()` присвоить соответствующее значение.

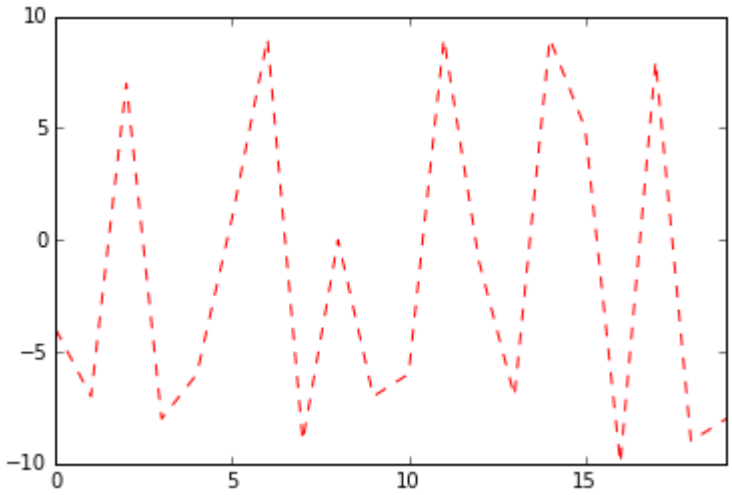
Например, отобразим данные из структуры `s` в виде звезд без соединительных линий:

```
>>> s.plot(style='r*')
```



Если нужно отобразить график в виде штриховой линией, то для этого аргументу style нужно присвоить значение *'r--'*:

```
>>> s.plot(style='r--')
```



В таблицах ниже представлены символы для оформления внешнего вида графика.

Таблица 11.2 - Символы, определяющие цвет

Символ	**Цвет**
`'b'`	синий
`'g'`	зеленый
`'r'`	красный
`'c'`	голубой
`'m'`	пурпурный
`'y'`	желтый
`'k'`	черный
`'w'`	белый

Таблица 11.3 - Символы, определяющие тип линии/точек

Символ	Описание
'-'	сплошная линия
'--'	штриховая линия
'-.'	линия “точка-тире”
':'	пунктирная линия
'.'	маркер: точка
','	маркер: пиксель
'o'	маркер: круг
'v'	маркер: треугольник с направленной вниз вершиной
'^'	маркер: треугольник с направленной вверх вершиной
'<'	маркер: треугольник с направленной влево вершиной
'>'	маркер: треугольник с направленной вправо вершиной
'1'	маркер: трехлучевая звезда, направленная лучом вниз
'2'	маркер: трехлучевая звезда, направленная лучом вверх
'3'	маркер: трехлучевая звезда, направленная лучом влево
'4'	маркер: трехлучевая звезда, направленная лучом вправо
's'	маркер: квадрат
'p'	маркер: пятиугольник
'*'	маркер: звезда
'h'	маркер: шестиугольник (тип 1)
'H'	маркер: шестиугольник (тип 2)
'+'	маркер: плюс
'x'	маркер: x
'D'	diamond marker
'd'	thin_diamond marker
'|'	vline marker
'_'	hline marker

11.2.2 Вывод графиков на разных плоскостях

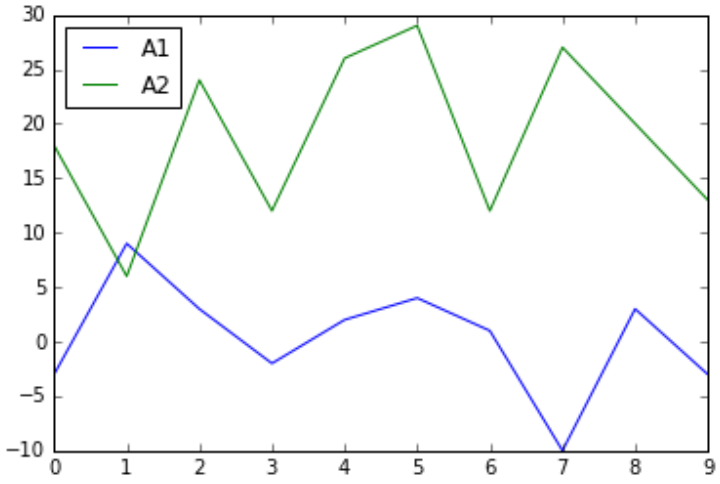
Для того чтобы вывести графики на разных полях, нужно аргументу `subplots` присвоить значение `True`, и, если это необходимо, задать размер выводимых диаграмм.

Создадим новый *DataFrame*:

```
>>> d_arr = [[random.randrange(-10, 10), random.randrange(0, 30)] for i
in range(10)]
>>> df = pd.DataFrame(d_arr, columns=['A1', 'A2'])
```

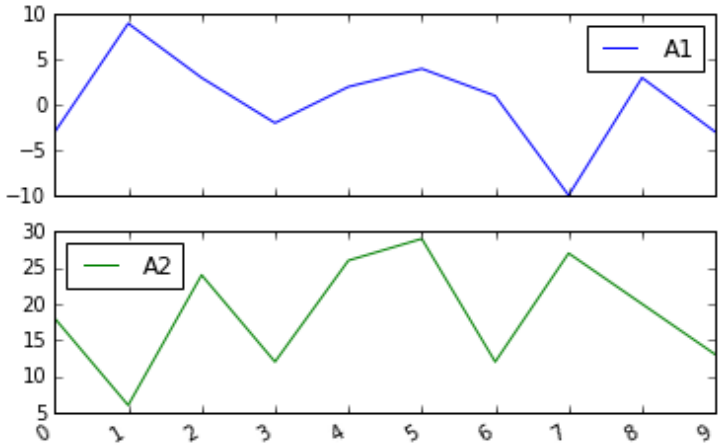
Выведем графики на одном поле:

```
>>> df.plot()
```



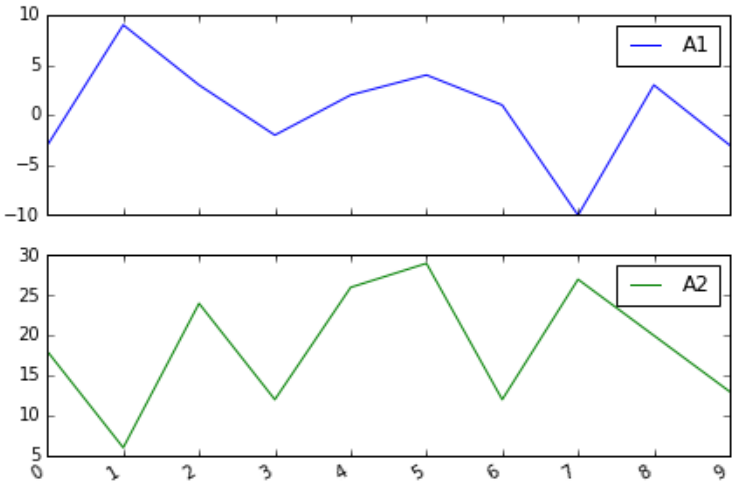
Теперь разнесем их по разным полям:

```
>>> df.plot(subplots=True)
```



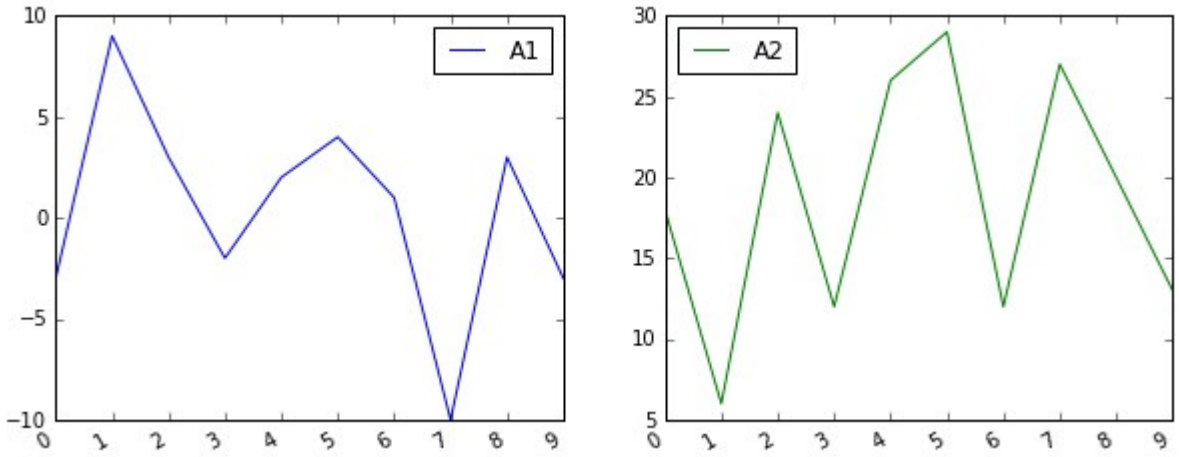
Для того, чтобы было удобно их просматривать - зададим нужный размер:

```
>>> df.plot(subplots=True, figsize=(7,5))
```



Можно изменить компоновку графиков, это делается через параметр `layout`, которому в качестве значения присваивается кортеж, первый аргумент которого количество строк, второй - количество столбцов, создаваемой компоновки:

```
>>> df.plot(subplots=True, layout=(1,2), figsize=(10,4))
```



Глава 12. Настройка внешнего вида таблиц

Если вы работает в *Jupyter Notebook*, то, наверное уже успели оценить, внешний вид таблиц, которые *pandas* формирует при их отображении.

Сравните то, что выводится, если использовать *REPL*:

```
>>> d = [{"name": "pen", "price": 3.9, "count": 8},
    {"name": "book", "price": 4.5, "count": 11},
    {"name": "folder", "price": 10.3, "count": 7}]
>>> df = pd.DataFrame(d)
>>> df
    count   name    price
0   8       pen     3.9
1   11      book    4.5
2   7       folder  10.3
```

и результат выполнения этого же кода в *Jupyter Notebook*:

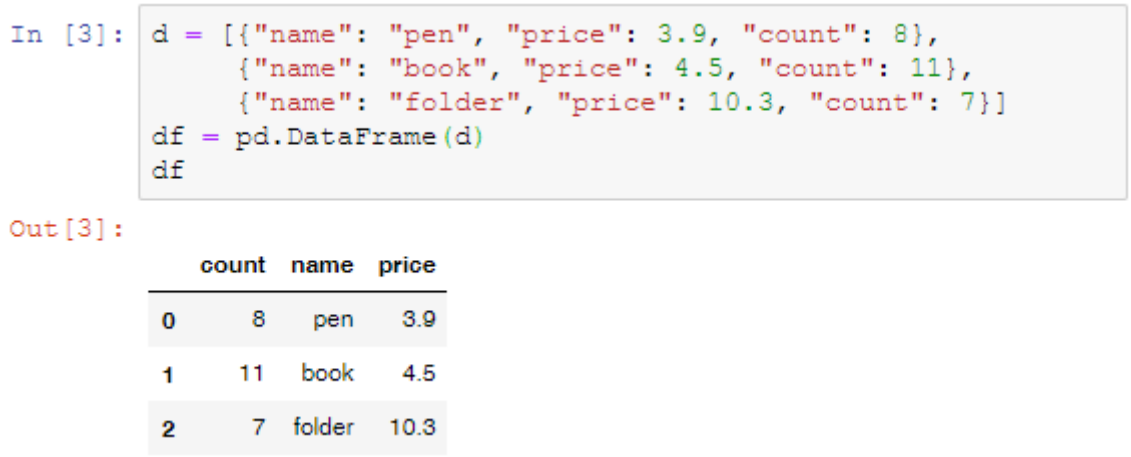


Рисунок 12.1 — Вывод *DataFrame* таблицы в *Jupyter Notebook*

Но это самый простой вариант, который можно получить. *Pandas* предоставляет инструменты, которые позволяют провести серьезную работу над визуальной составляющей отображаемых данных, что значительно повышает их читаемость и выразительность.

Для начала создадим набор данных, с которым будем работать:

```
import numpy as np
np.random.seed(123)
df = pd.DataFrame(np.random.randn(7, 5), columns=['a', 'b', 'c', 'd',
'e'])
```

Получим следующую таблицу:

	a	b	c	d	e
0	-1.085631	0.997345	0.282978	-1.506295	-0.578600
1	1.651437	-2.426679	-0.428913	1.265936	-0.866740
2	-0.678886	-0.094709	1.491390	-0.638902	-0.443982
3	-0.434351	2.205930	2.186786	1.004054	0.386186
4	0.737369	1.490732	-0.935834	1.175829	-1.253881
5	-0.637752	0.907105	-1.428681	-0.140069	-0.861755
6	-0.255619	-2.798589	-1.771533	-0.699877	0.927462

Рисунок 12.2 — Таблица *DataFrame* без дополнительного форматирования

Для настройки стиля можно использовать следующие методы:

- `Styler.applymap()`: обеспечивает поэлементную работу с данными. Метод принимает функцию, которая применяется к каждому элементу структуры *pandas*. Передаваемая функция должна принимать скалярную величину и возвращать *CSS* элемент в формате “`атрибут: значение`”.
- `Styler.apply()`: единицей обработки является столбец, строка или вся таблица. Она принимает в качестве аргумента функцию, аргументом которой является структура *Series* или *DataFrame*, а возвращать она должна структуру того же размера, элементы которой — это строки, являющиеся *CSS* элементами в формате “`атрибут: значение`”.

Помимо изменения стилевого представления данных, можно производить изменение непосредственно самих данных, например: добавить знак процента или валюты, ограничить количество десятичных знаков и т.д., делается это с помощью функции `Styler.format()`.

12.1 Изменение формата представления данных

Для изменения формата представления данных используется функция `Styler.format()`, которой в качестве аргумента передается `formatter` - строка, построенная по правилам *Format specifications* https://docs.python.org/3/library/string.html#format-specification-mini-language, словарь или функция, принимающая элемент данных и возвращающая его отформатированное представление в виде строки.

Для демонстрации работы `format()` возьмем исходный набор данных, округлим значения до третьего знака после запятой и добавим символы, обозначающие температуру по шкале Цельсия:

```
>>> df.style.format("{:.3} °C")
```

	a	b	c	d	e
0	-1.09 °C	0.997 °C	0.283 °C	-1.51 °C	-0.579 °C
1	1.65 °C	-2.43 °C	-0.429 °C	1.27 °C	-0.867 °C
2	-0.679 °C	-0.0947 °C	1.49 °C	-0.639 °C	-0.444 °C
3	-0.434 °C	2.21 °C	2.19 °C	1.0 °C	0.386 °C
4	0.737 °C	1.49 °C	-0.936 °C	1.18 °C	-1.25 °C
5	-0.638 °C	0.907 °C	-1.43 °C	-0.14 °C	-0.862 °C
6	-0.256 °C	-2.8 °C	-1.77 °C	-0.7 °C	0.927 °C

Рисунок 12.3 — Таблица с модификацией данных исходной структуры

Можно предварительно сформировать словарь с описанием форматов столбцов, а потом передать его в качестве `formatter`'а:

```
f_dict = {'a':'{:.3} °C', 'b': '{0:.1}', 'c': '{:.2} %', 'd': '$ {:.2}'}
df.style.format(f_dict)
```

	a	b	c	d	e
0	-1.09 °C	1e+00	0.28 %	$ -1.5	-0.5786
1	1.65 °C	-2e+00	-0.43 %	$ 1.3	-0.86674
2	-0.679 °C	-0.09	1.5 %	$ -0.64	-0.443982
3	-0.434 °C	2e+00	2.2 %	$ 1.0	0.386186
4	0.737 °C	1e+00	-0.94 %	$ 1.2	-1.25388
5	-0.638 °C	0.9	-1.4 %	$ -0.14	-0.861755
6	-0.256 °C	-3e+00	-1.8 %	$ -0.7	0.927462

Рисунок 12.4 — Таблица с модификацией данных исходной структуры

Если форматирование требует более сложной логики, то можно воспользоваться вариантом, когда в качестве `formatter`'а передается функция:

```
df.style.format(lambda x: None if x < 0 else "{:.2f}".format(x))
```

	a	b	c	d	e
0	None	1.00	0.28	None	None
1	1.65	None	None	1.27	None
2	None	None	1.49	None	None
3	None	2.21	2.19	1.00	0.39
4	0.74	1.49	None	1.18	None
5	None	0.91	None	None	None
6	None	None	None	None	0.93

Рисунок 12.5 — Таблица с модификацией данных исходной структуры через *lambda*-функцию

12.2 Создание собственных стилей

Рассмотрим различные варианты того, как можно создавать собственные стили для модификации внешнего вида табличных данных. Для этого нам нужно создать функцию, которая будет возвращать стилизованное значение элемента данных (строки/столбца/всей таблицы), и передать ее в `Styler.applymap()` либо в `Styler.apply()`.

12.2.1 Задание цвета надписи для элементов данных

Для задания цвета надписи определим функцию, которая будет задавать красный цвет для значений меньше нуля и зеленый для значений больше двух:

```
def font_color_mod(val):
    color_val = "black"
    if val < 0:
        color_val = "red"
    elif val >= 2:
        color_val = "green"

    return f"color: {color_val}"

df_font_mod = df.style.applymap(font_color_mod)
df_font_mod
```

Получим следующим образом стилизованную таблицу:

	a	b	c	d	e
0	-1.08563	0.997345	0.282978	-1.50629	-0.5786
1	1.65144	-2.42668	-0.428913	1.26594	-0.86674
2	-0.678886	-0.094709	1.49139	-0.638902	-0.443982
3	-0.434351	2.20593	2.18679	1.00405	0.386186
4	0.737369	1.49073	-0.935834	1.17583	-1.25388
5	-0.637752	0.907105	-1.42868	-0.140069	-0.861755
6	-0.255619	-2.79859	-1.77153	-0.699877	0.927462

Рисунок 12.6 — Таблица с измененным цветом надписей элементов

12.2.2 Задание цвета ячейки таблицы

Для работы со столбцами/строками используется метод `apply()`, которому в качестве аргумента передается функция, при этом, по умолчанию, в нее будут передаваться столбцы в виде объектов *Series*, если необходимо вести обработку по строкам, то нужно задать направление через параметр `axis`:

- `axis=0` - задает обход по столбцам;
- `axis=1` - задает обход по строкам.

Напишем функцию, которая будет выделять в строке наибольшее и наименьшее значение:

```
def highlight_min_max_in_row(row):
    min_map = row == row.min()
    max_map = row == row.max()
    style_table = []
    for m in zip(min_map, max_map):
        tmp = 'background-color: skyblue' if m[0] else ''
        tmp = 'background-color: orange' if m[1] else tmp if len(tmp)>0
else ''
        style_table.append(tmp)
    return style_table

df.style.apply(highlight_min_max_in_row, axis=1)
```

	a	b	c	d	e
0	-1.08563	0.997345	0.282978	-1.50629	-0.5786
1	1.65144	-2.42668	-0.428913	1.26594	-0.86674
2	-0.678886	-0.094709	1.49139	-0.638902	-0.443982
3	-0.434351	2.20593	2.18679	1.00405	0.386186
4	0.737369	1.49073	-0.935834	1.17583	-1.25388
5	-0.637752	0.907105	-1.42868	-0.140069	-0.861755
6	-0.255619	-2.79859	-1.77153	-0.699877	0.927462

Рисунок 12.7 — Таблица с измененным цветом ячеек

12.2.3 Задание цвета строки таблицы

Можно задать цвет столбцу или строке таблицы, для этого нужно выставить значение `background-color` для всех ячеек столбца / строки. Выделим все строки таблицы, в которых встречается значение по модулю больше двух:

```
def highlight_row(row):
    style_table = ['background-color: coral']*len(row) if abs(row).max()
> 2 else ['']*len(row)
    return style_table

df.style.apply(highlight_row, axis=1)
```

	a	b	c	d	e
0	-1.08563	0.997345	0.282978	-1.50629	-0.5786
1	1.65144	-2.42668	-0.428913	1.26594	-0.86674
2	-0.678886	-0.094709	1.49139	-0.638902	-0.443982
3	-0.434351	2.20593	2.18679	1.00405	0.386186
4	0.737369	1.49073	-0.935834	1.17583	-1.25388
5	-0.637752	0.907105	-1.42868	-0.140069	-0.861755
6	-0.255619	-2.79859	-1.77153	-0.699877	0.927462

Рисунок 12.8 — Таблица с измененным цветом строк

12.3 Встроенные инструменты задания стилей

Библиотека *pandas* содержит большое количество различных встроенных инструментов для задания внешнего вида таблицы. Рассмотрим некоторые из них.

12.3.1 Подсветка минимального и максимального значений

В одном из предыдущих примеров мы сами определяли минимальный / максимальный элемент и задавали цвет ячейки через *CSS* атрибут. *Pandas* представляет встроенные методы для решения данной задачи:

```
df.style.highlight_max(axis=1)
```

	a	b	c	d	e
0	-1.08563	0.997345	0.282978	-1.50629	-0.5786
1	1.65144	-2.42668	-0.428913	1.26594	-0.86674
2	-0.678886	-0.094709	1.49139	-0.638902	-0.443982
3	-0.434351	2.20593	2.18679	1.00405	0.386186
4	0.737369	1.49073	-0.935834	1.17583	-1.25388
5	-0.637752	0.907105	-1.42868	-0.140069	-0.861755
6	-0.255619	-2.79859	-1.77153	-0.699877	0.927462

Рисунок 12.9 — Таблица с подсветкой максимального значения

```
df.style.highlight_min(axis=1)
```

	a	b	c	d	e
0	-1.08563	0.997345	0.282978	-1.50629	-0.5786
1	1.65144	-2.42668	-0.428913	1.26594	-0.86674
2	-0.678886	-0.094709	1.49139	-0.638902	-0.443982
3	-0.434351	2.20593	2.18679	1.00405	0.386186
4	0.737369	1.49073	-0.935834	1.17583	-1.25388
5	-0.637752	0.907105	-1.42868	-0.140069	-0.861755
6	-0.255619	-2.79859	-1.77153	-0.699877	0.927462

Рисунок 12.10 — Таблица с подсветкой минимального значения

12.3.2 Подсветка null-элементов

Среди доступных инструментов отметим подсветку null-элементов. Зададим некоторым элементам структуры null-значение:

```
df.iloc[3,4] = None
df.iloc[4,1] = None
```

	a	b	c	d	e
0	-1.085631	0.997345	0.282978	-1.506295	-0.578600
1	1.651437	-2.426679	-0.428913	1.265936	-0.866740
2	-0.678886	-0.094709	1.491390	-0.638902	-0.443982
3	-0.434351	2.205930	2.186786	1.004054	NaN
4	0.737369	NaN	-0.935834	1.175829	-1.253881
6	-0.637752	0.907105	-1.428681	-0.140069	-0.861755
6	-0.255619	-2.798589	-1.771533	-0.699877	0.927462

Рисунок 12.11— Таблица с исходными данными

Теперь воспользуемся функцией highlight_null, для подсветки null-элементов:

```
df.style.highlight_null(null_color='crimson')
```

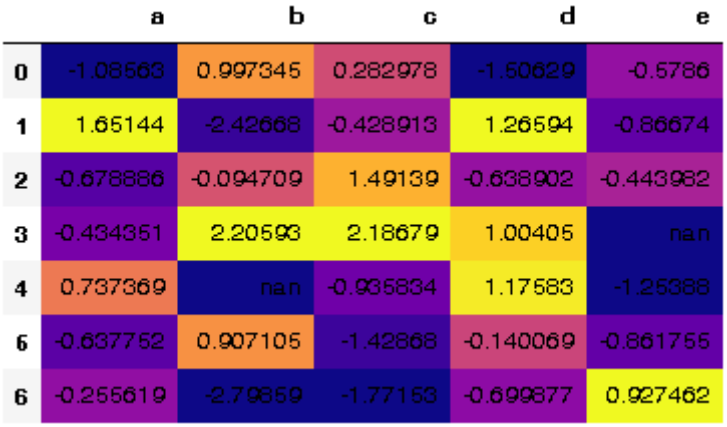
	a	b	c	d	e
0	-1.08563	0.997345	0.282978	-1.50629	-0.5786
1	1.65144	-2.42668	-0.428913	1.26594	-0.86674
2	-0.678886	-0.094709	1.49139	-0.638902	-0.443982
3	-0.434351	2.20593	2.18679	1.00405	nan
4	0.737369	nan	-0.935834	1.17583	-1.25388
6	-0.637752	0.907105	-1.42868	-0.140069	-0.861755
6	-0.255619	-2.79859	-1.77153	-0.699877	0.927462

Рисунок 12.12— Таблица с подсветкой null-значений

12.3.3 Задание тепловой карты

Для раскраски таблицы палитрой, воспользуйтесь функцией `background_gradient()`:

```
df.style.background_gradient(cmap='plasma')
```



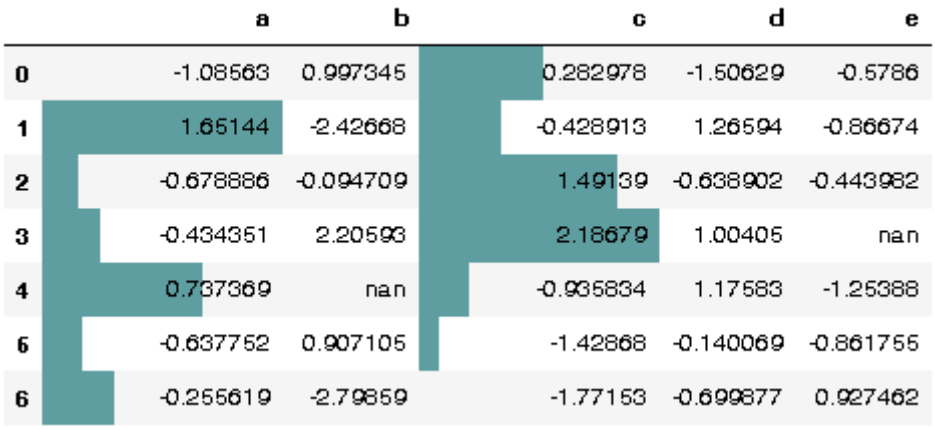
	a	b	c	d	e
0	-1.08563	0.997345	0.282978	-1.50629	-0.5786
1	1.65144	-2.42668	-0.428913	1.26594	-0.86674
2	-0.678886	-0.094709	1.49139	-0.638902	-0.443982
3	-0.434351	2.20593	2.18679	1.00405	nan
4	0.737369	nan	-0.935834	1.17583	-1.25388
5	-0.637752	0.907105	-1.42868	-0.140069	-0.861755
6	-0.255619	-2.79859	-1.77153	-0.699877	0.927462

Рисунок 12.13 — Таблица с тепловой картой

12.3.4 Наложение столбчатой диаграммы

Ещё один интересный функционал, который предоставляет *pandas* - это наложение на таблицу столбчатой диаграммы, по которой можно оценить распределение численных данных:

```
df.style.bar(subset=['a', 'c'], color='cadetblue')
```



	a	b	c	d	e
0	-1.08563	0.997345	0.282978	-1.50629	-0.5786
1	1.65144	-2.42668	-0.428913	1.26594	-0.86674
2	-0.678886	-0.094709	1.49139	-0.638902	-0.443982
3	-0.434351	2.20593	2.18679	1.00405	nan
4	0.737369	nan	-0.935834	1.17583	-1.25388
5	-0.637752	0.907105	-1.42868	-0.140069	-0.861755
6	-0.255619	-2.79859	-1.77153	-0.699877	0.927462

Рисунок 12.14 — Таблица со столбчатой диаграммой

12.3.5 Цепочки вычислений (*Method Chaining*) для настройки внешнего вида таблицы

Если необходимо за один раз настроить внешний вид таблицы, то это можно сделать с помощью построения цепочки вычислений, которая позволяет вызвать методы применения стилей друг за другом:

```
(df.style
.applymap(font_color_mod)
.bar(subset=['a', 'c'], color='cadetblue')
.highlight_null(null_color='crimson'))
```

	a	b	c	d	e
0	-1.08563	0.997345	0.282978	-1.50629	-0.5786
1	1.65144	-2.42668	-0.428913	1.26594	-0.86674
2	-0.678886	-0.094709	1.49139	-0.638902	-0.443982
3	-0.434351	2.20593	2.18679	1.00405	nan
4	0.737369	nan	-0.935834	1.17583	-1.25388
5	-0.637752	0.907105	-1.42868	-0.140069	-0.861755
6	-0.255619	-2.79859	-1.77153	-0.699877	0.927462

Рисунок 12.15 — Таблица с применением ряда последовательных модификаций

Заключение

На этом позволим себе закончить обзор возможностей библиотеки *pandas*, мы постарались остановиться на важных, по нашему мнению, аспектах этого инструмента. Надеемся, что информация, которую вы встретили на страницах книги, оказалась полезной. Если у вас есть замечания или пожелания по содержанию, то пишите нам на *devpractice.mail@gmail.com*, мы будем очень рады обратной связи.