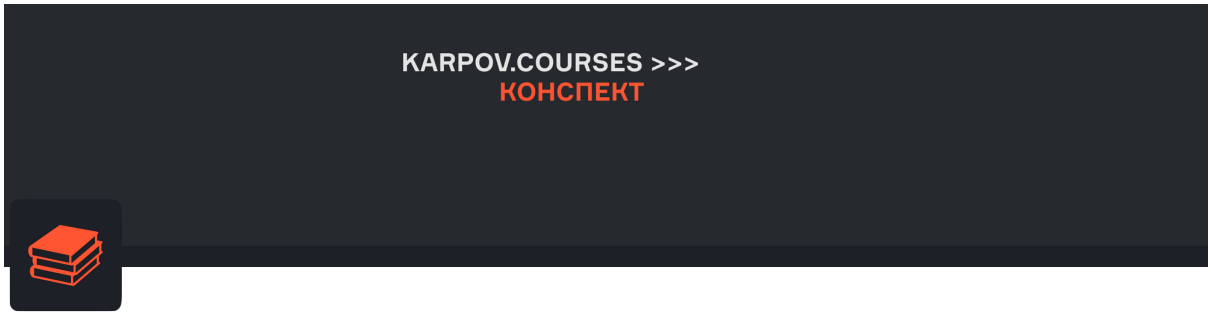


> Конспект > 5 урок > pandas

> Оглавление

> Чтение файлов и обзор данных

`pandas` умеет читать данные в самых разных формах хранения:

1. CSV (comma separated values, простой формат хранения таблиц)
2. Excel таблицы
3. Напрямую из баз данных

4. Веб-форматы: `json`, `xml`
5. Специальные форматы для данных: `parquet`, `feather`, `orc`

Мы сегодня научимся читать CSV. Это очень простой формат, и из-за этого очень популярный: почти во всех соревнованиях по data science входные данные дают именно в CSV.

Не переживайте, `pandas` всегда будет приводить все читаемые данные к одному формату – поэтому вы без труда сможете открыть любой другой источник, если будете уметь работать с CSV.

CSV

Формат CSV очень простой: на каждую запись выделяется отдельная строка, а колонки отделяются специальным символом (по умолчанию запятой `,`).

Таким образом, таблица

Имя	Возраст	Город
Миша	28	Москва
Саша	12	Казань
Илья	54	Мурманск

будет выглядеть в файле как

```
Имя,Возраст,Город
Миша,28,Москва
Саша,12,Казань
Илья,54,Мурманск
```

CSV-файл является валидным, только если каждая строка содержит необходимое количество разделителей (количество колонок - 1). Разделитель может быть любым, но он не должен встречаться в содержимом ячеек таблицы. По этой причине вместо запятой часто приходится использовать другой символ, например, точку с запятой `;`.

В `pandas` чтение CSV-файлов реализуется функцией `read_csv()`, которая принимает в качестве первого аргумента путь до файла (полный или относительный) и возвращает внутренний объект типа `DataFrame`.

```
df = pd.read_csv('train.csv')
```

`DataFrame` объект хранит в себе всю информацию о считанных данных, включая сами данные, колонки, в которые они организованы и их тип, количество строк и колонок и многое другое. Кроме того, `DataFrame` имеет различные методы для работы с данными, которые в нём содержатся.

Беглый взгляд на данные

- `.head()` – метод, который показывает первые n строк таблицы. По умолчанию показывает первые 5 строк, но можно передать аргумент, указав таким образом n.

```
# Полезно, чтобы взглянуть на данные
df.head()  # покажет первые 5 строк
df.head(3)  # покажет первые 3 строки
```

- `.describe()` – получить основные статистики таблицы. Возвращает таблицу, где для каждой колонки указаны количество заполненных записей (count), а также среднее (mean), стандартное отклонение (std), квантили (25%, 50%, 75%), минимум и максимум (min и max соответственно) для каждой колонки с числами.

```
# Полезно, чтобы посмотреть на масштаб величин и их разброс
df.describe()
```

- `.columns` – получить список колонок в виде объекта типа `Index`. Из него можно получать элементы по индексу и брать срезы.
- `.dtypes` – получить типы данных колонок в таблице. Стоит отметить, что в `pandas` используются свои типы данных, не всегда такие как в базовом Python. В частности, строки не выделяются в отдельный тип, а записываются как произвольные объекты (тип `object`). Кроме того, `pandas` не всегда правильно определяет тип данных в колонке при чтении, поэтому иногда его нужно указывать явным образом. Это можно сделать непосредственно при чтении через функцию `read_csv()`:

```
df = pd.read_csv('train.csv', dtype={'season': int})
```

Функция `read_csv()` имеет множество аргументов, полезных в определенных ситуациях. Здесь мы их разбирать не будем, но вы можете почитать о них в документации.

> Фильтрация данных

В `pandas` можно быстро и просто фильтровать данные по значениям. Фильтров может быть сколько угодно, и они могут быть весьма комплексными.

```
# Можно фильтровать данные через квадратные скобки
df[df['workingday'] == 0]  # получить все поездки за выходные

# Этот запрос можно читать как "взять df, где у df поле workingday равно 0"
```

Логическое "И"

Можно потребовать, чтобы выполнилось несколько условий одновременно. Для этого используется оператор `&`:

```
# Переносы строк внутри [] не играют роли - мы их делаем только для читаемости
df[
    (df['workingday'] == 1) & (df['season'] == 1)
]

# Получим весенние поездки в рабочие дни
```

Сразу отметим две вещи:

- **Только** `&` в pandas означает логическое "И" – требование, чтобы оба условия выполнились. Обратите внимание, в Python для логического "И" в булевых выраженях можно также использовать оператор `and`. Например, можно написать:

```
if var1 is None and len(arr) > 0:
    print('var1 is None and array "arr" has length > 0')
```

В pandas так сделать нельзя. Это из-за того, что выражения `df['workingday'] == 1` не являются булевыми типами (об этом немного позже).

- Оба условия обернуты в скобки. В `pandas` так стоит делать всегда, так как без них оператор `&` будет выполняться в неправильном порядке, и код не запустится.

Логическое "ИЛИ"

Логическое "ИЛИ" делается через символ `|`. Оно требует выполнения хотя бы одного условия из всех. Заметьте, нельзя использовать питоновский `or`, как и в случае с логическим "И" (оператор `&`).

```
# Либо влажность < 10%, либо температура в Цельсиях строго больше 30
df[
    (df['humidity'] < 10) | (df['temp'] > 30.)
]
```

Логическое "НЕ"

Логическое "НЕ" делается через символ `~`. Точно так же, как в прошлых операторах, встроенный в Python `not` не подойдет.

```
# Не забудьте про скобки, иначе ~ применится в неправильном порядке,
# и мы получим некорректный результат
df[
    ~(df['workingday'] == 0)
]

# Получим все поездки не в выходные
```

При комбинировании операторов стоит быть особенно внимательным со скобками:

```
# Когда объединяете "НЕ" с другими условиями, окружите его тоже скобками
df[
    (~(df['workingday'] == 0)) & (df['temp'] >= 10) & (df['temp'] < 30)
]
# Не рабочий день и температура от 10 включительно до 30 не включительно
# можно было использовать df['workingday'] != 0
```

> Функции-фильтры

В `pandas` есть много функций для сложных проверок. Пройдемся по главным.

- `.isna()` – фильтрует по записям, в которых пропущено значение. Обратите внимание, в `pandas` нельзя использовать конструкцию `is None`.

```
# Взять df, где у df поле windspeed не заполнено
# Этот код не будет работать:
# df[df['windspeed'] is None]

# А этот будет:
df[df['windspeed'].isna()]
```

`.isna()` можно применить над целым `DataFrame`, в данном случае вернётся новый `DataFrame`, в ячейках которого будет лежать `True` там, где в оригинальной таблице пропущено значение, и `False` – в противном случае.

```
# Однако смотреть на это может быть не очень удобно
df.isna()
```

Результат:

	datetime	season	holiday	workingday	weather	temp	atemp	humidity	windspeed	casual	registered	count
0	False	False	False	False	False	False	False	False	False	False	False	False
1	False	False	False	False	False	False	False	False	False	False	False	False
2	False	False	False	False	False	False	False	False	False	False	False	False
3	False	False	False	False	False	False	False	False	False	False	False	False
4	False	False	False	False	False	False	False	False	False	False	False	False
...	...	...	...	...	...	...	...	...	...	...	...	...
10881	False	False	False	False	False	False	False	False	False	False	False	False
10882	False	False	False	False	False	False	False	False	False	False	False	False
10883	False	False	False	False	False	False	False	False	False	False	False	False
10884	False	False	False	False	False	False	False	False	False	False	False	False
10885	False	False	False	False	False	False	False	False	False	False	False	False

10886 rows × 12 columns

Если мы хотим проверить, что нигде в таблице действительно нет пропусков, можно воспользоваться вспомогательной функцией `.any()`. Она проверяет, состоит ли срез данных целиком из значений `False`. Функция берёт все данные и смотрит их булевый тип. Возвращает `True`, если хотя бы одна из записей имеет значение `True`, иначе возвращает `False`.

```
# Первый any сворачивает до уровня колонок, второй сворачивает в одно значение
if df.isna().any().any():
    print('есть пропуски')
else:
    print('пропусков нет')
```

- `.isin()` – проверяет, находится ли значение в разрешенном списке. Как и в `.isna()`, мы не можем использовать питоновскую конструкцию `in`:

```
# Этот код не будет работать:
# df[df['season'] in [1, 3, 4]]

# А этот будет:
df[df['season'].isin([1, 3, 4])]
```

Помимо этого, в `pandas` есть ещё множество функций для работы со строками и датами. Часть из них мы подробнее рассмотрим в домашней работе.

> Series и Index

Примеры выше могли оставить несколько вопросов.

1. Почему не работают встроенные `in`, `not`?
2. Откуда такой странный синтаксис с квадратными скобками?
3. Что будет, если вызвать просто `df['season'] == 1`? Это `True`/`False` или что-то другое?

Series

Давайте начнем с 3-его вопроса:

```
df['season'] == 1
```

Результат:

```
0          True
1          True
2          True
3          True
4          True
          ...
10881     False
10882     False
10883     False
10884     False
10885     False
Name: season, Length: 10886, dtype: bool
```

Это объект типа `Series`. Его можно воспринимать как `DataFrame` с одной колонкой. У `Series` есть *индекс* (англ. *index*) – числа слева – и значения, в данном случае, булевые (что можно увидеть как `dtype: bool`).

Во всех примерах фильтры на самом деле возвращали объект `Series`, который мы передавали в `df[...]`. Далее `pandas` искал в `Series` значения `True`, запоминал их индекс, а затем в оригинальном датафрейме `df` забирал значения по этому индексу.

Логические операторы в фильтрах также работали с `Series`. Например, `~(df['season'] == 1)` вернёт `Series`, аналогичный приведённому выше с точностью до наоборот – вместо `True` будет `False`, а вместо `False` – `True`.

Такое использование `Series` для логических проверок привело к тому, что встроенные в Python `in` и `not` не работают. Просто запомните, что вместо `df["col1"] in [1, 2, 3]` надо использовать `df["col1"].isin([1, 2, 3])`, а вместо `not` использовать символ `~`.

Index

Индекс в `pandas` похож на индекс в списках и на ключи в словарях. По элементу индекса можно отбирать элементы, причём несколькими способами:

- Для «прямого» обращения по индексу используется `.loc`:

```
df.loc[4]
```

Данный запрос вернёт объект `Series` с колонками, причём строковые названия колонок будут составлять индекс `Series`. К ним можно обращаться как к ключам в словаре:

```
df.loc[4]['temp']
```

Обратите внимание, здесь интерфейс `DataFrame` и `Series` различается. Вызов напрямую `df[4]` приведёт к ошибке.

В качестве индекса может быть что угодно. С помощью функции `.set_index()` можно поставить любую колонку таблицы вместо существующего индекса:

```
# Сначала создаём копию DataFrame, чтобы иметь возможность
# продолжать работать с оригиналом
df_dt = df.copy().set_index('datetime')

# Полученный индекс имеет помимо прочего название (name) - datetime
```

- Обратиться по порядку, а не непосредственно по индексу можно с помощью `.iloc`:

```
df.iloc[4]
```

.loc и .iloc

`.loc` и `.iloc` обладают большим количеством возможностей, так что остановимся на них поподробнее.

- `.loc` можно указать сразу и индекс, и колонку — через запятую:

```
df_dt.loc['2011-01-01 01:00:00', 'weather']
```

Для `.iloc` такое не работает, приходится писать через две скобки:

```
df_dt.iloc[4]['weather']
```

- Для `.loc` работает всё, что мы знаем из срезов, но при указании диапазона берутся все данные, которые лежат между ними в текущей сортировке:

```
df_dt.loc['2011-01-01 01:00:00':'2011-01-01 03:00:00']

# Также можно указывать шаг
# df_dt.loc['2011-01-01 01:00:00':'2011-01-01 03:00:00':2]
```

- Для `.iloc` работает всё, что мы знаем из срезов, правая граница не включается:

```
df_dt.iloc[0:3]

# Также можно указывать шаг
# df_dt.iloc[0:3:2]
```

- С помощью `.loc` также можно перезаписывать значения в `DataFrame`:

```
df_dt.loc['2011-01-01 04:00:00', 'weather'] = 234

# Через .iloc это сделать нельзя, pandas выдаст предупреждение,
# и ничего не произойдёт
# df_dt.iloc[4]['weather'] = -2
```

Наконец отметим, что индекс `DataFrame` можно сбросить обратно к числовым значениям с помощью функции `.reset_index()`:

```
# Возвращает копию, НЕ редактирует исходный DataFrame
df_dt.reset_index()

# Модифицирует исходный DataFrame
df_dt.reset_index(inplace=True)
```

> Группировка данных

Хорошо, теперь мы умеем фильтровать данные. Тем не менее, часто поступают запросы на подсчет какой-то агрегированной величины среди определенной группы данных.

Скажем, может поступать запрос «подсчитай суммарную выручку в разбивке по кварталам» или «подсчитай среднее число пользователей за день для каждого

дня с начала года». Особенность таких запросов в том, что их обработка строится в несколько действий:

1. Отбираем подмножество по определенному критерию.
2. Над подмножеством применяем функцию, которая вернет одно число (т.е. превратит **целое подмножество** в одно **число**).
3. Возвращаемся в п.1, отбираем другое подмножество, повторяем алгоритм.
4. Возвращаем все полученные агрегации.

Такие задачи решаются через **группировку** данных с последующей **агрегацией**. В `pandas` есть все инструменты для их решения.

```
"""
Допустим, нас попросили посчитатать среднюю температуру в разбивке по сезонам.
Для этого надо сгруппировать по сезонам, т.е.
сделать 4 подмножества данных - по одному на каждый сезон,
затем каждую группу "схлопнуть" до одного числа - среднего по подмножеству.
Получим 4 числа - по одному среднему на каждую группу.
"""
# Рекомендуется использовать такой способ
df.groupby('season').agg({'temp': 'mean'})
# сначала группируем по значению season,
# затем просим из каждой группы взять колонку temp и подсчитать mean - среднее
# mean - это специальная строка, принимает только определенные значения

# доступные функции можно найти здесь
# https://stackoverflow.com/questions/53943319/what-are-all-python-pandas-agg-functions?rq=1
```

Результат:

	temp
season	
1	12.530491
2	22.823483
3	28.789111
4	16.649239

```
# группировать можно сразу по нескольким параметрам
# так мы подсчитаем среднее для всех колонок
df.groupby(['season', 'workingday']).mean()
```

Результат:

season	workingday	holiday	weather	temp	atemp	humidity	windspeed	casual	registered	count
1	0	0.082751	1.446387	12.344918	14.907646	56.988345	15.776540	26.615385	79.111888	105.727273
	1	0.000000	1.414114	12.617593	15.379768	55.973742	14.102107	10.267505	111.058534	121.326039
2	0	0.057143	1.388095	22.742310	26.536792	61.114286	11.556256	79.867857	142.825000	222.692857
	1	0.000000	1.438457	22.859503	26.696046	60.736926	14.226238	33.060222	178.889065	211.949287
3	0	0.108108	1.369369	28.675068	32.735743	65.875000	11.862931	82.248874	150.271396	232.520270
	1	0.000000	1.365312	28.844000	32.446949	63.280759	11.338448	37.767480	197.562602	235.330081
4	0	0.108108	1.346847	15.696351	19.106227	64.557432	11.411330	48.507883	143.629505	192.137387
	1	0.000000	1.514085	17.107616	20.518667	66.951246	11.806496	18.995125	183.288732	202.283857

```
# а так - только для humidity
df.groupby(['season', 'workingday'])['humidity'].mean()
# равнозначный код
# df.groupby(['season', 'workingday']).agg({'humidity': 'mean'})
```

Результат:

```
season  workingday
1       0             56.988345
        1             55.973742
2       0             61.114286
        1             60.736926
3       0             65.875000
        1             63.280759
4       0             64.557432
        1             66.951246
Name: humidity, dtype: float64
```

В последних двух примерах в полученных объектах `DataFrame` и `Series` индекс представлен объектом `MultiIndex`. Работать с ним нетривиально, но его всегда можно сбросить через функцию `.reset_index()` или передав в функцию `.groupby()` ключ `as_index=False`.

Функция `.groupby()` возвращает данные в промежуточном, нечитаемом состоянии. Для того, чтобы с ними работать, к ним сначала нужно применить агрегирующую функцию.

> Работа с датами и временем

Время и даты в `pandas` имеют свои особенности, поэтому про них поговорим отдельно.

Мы уже знаем `datetime.datetime` и `datetime.date` и использовали их на практике. Но `pandas` имеет свои типы для работы со временем! Они были введены, потому что библиотека `datetime` не дает той гибкости в работе, которую хотел реализовать `pandas`. Придется разбираться с этим новым типом. Знакомьтесь, `pd.datetime`.

```
# Любой тип (не только datetime.datetime, но и строки, и числа)
# можно преобразовать к pd.datetime
```

```
pd.to_datetime(df['datetime']) # после преобразования
```

Результат:

```
0        2011-01-01 00:00:00
1        2011-01-01 01:00:00
2        2011-01-01 02:00:00
3        2011-01-01 03:00:00
4        2011-01-01 04:00:00
                 ...
10881    2012-12-19 19:00:00
10882    2012-12-19 20:00:00
10883    2012-12-19 21:00:00
10884    2012-12-19 22:00:00
10885    2012-12-19 23:00:00
Name: datetime, Length: 10886, dtype: datetime64[ns]
```

На первый вгляд кажется, что ничего не поменялось. Но посмотрите на тип данных – до преобразования он был `object`, а после преобразования стал `datetime64[ns]`. Второй тип умеет более точно работать с временем.

Давайте посмотрим, что же такого хорошего нам дал `pd.datetime`:

```
# Для начала сохраним результат: создадим копию и в ней превратим колонку в pd.datetime
df_1 = df.copy()
df_1['datetime'] = pd.to_datetime(df_1['datetime'])
```

```
# Теперь можно вытягивать куски из даты и использовать в фильтрациях и аггрегациях
# Используйте .dt, чтобы вытащить не саму дату, а какую-то ее часть
df_1[df_1['datetime'].dt.month == 5]  # через .dt взяли месяц
# получили данные за пятый месяц
```

Объект, к которому мы обращаемся через `.dt`, содержит в себе представление этой колонки, разделенное на компоненты, такие как дни, месяцы и т.д.

Посмотрим, как можно делать группировки с использованием `pd.datetime`:

```
# Можно использовать .dt.month, чтобы сгруппировать по месяцам
df_1.groupby(
    df_1['datetime'].dt.month  # для каждого месяца
).agg({
```

```
    'temp': 'mean',  # узнаем среднюю температуру
    "humidity": 'min'  # и минимальную влажность
})  # в разбивке по месяцам
```

Результат:

datetime	temp	humidity
1	9.840000	25
2	11.798535	8
3	15.902175	0
4	18.718372	16
5	22.674079	21
6	27.064496	20
7	30.841711	17
8	29.736689	25
9	25.779032	28
10	20.933853	29
11	15.185752	16
12	13.831206	26

```
rents_by_week = df_1.groupby(
    df_1['datetime'].dt.weekofyear  # для каждой недели в году
```

```
).agg({
    'temp': 'count'
})

# sample(10) - это взять 10 случайных записей
# Чтобы была воспроизводимость, фиксируем состояние генератора случайности
rents_by_week.sample(10, random_state=42)
# количество поездок в разбивке по неделям
```

Результат:

	temp
datetime	
46	312
29	144
31	288
51	96
41	336
48	144
6	330
15	334
11	308
5	261

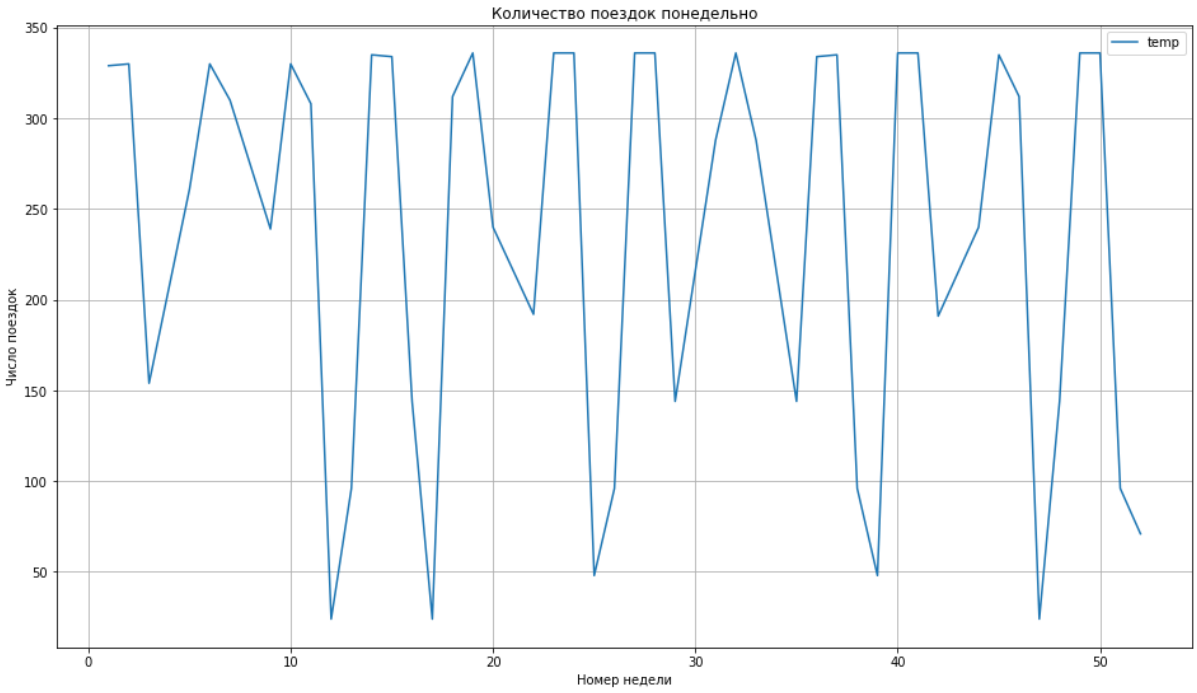
Больше узнать о том, какие компоненты даты и времени доступны, можно узнать из документации.

> Визуализация

В примере выше мы получили разбивку числа поездок по неделям. Получилась большая таблица, и она весьма трудна для чтения. Давайте ее визуализируем.

```
# самые простой способ - попросить pandas нарисовать
# но контроля над картинками будет мало
rents_by_week.plot(
    # названия агрументов приходят из matplotlib.pyplot, см. ниже
    xlabel='Номер недели',
    ylabel='Число поездок',
    title='Количество поездок понедельно',
    grid=True,
    figsize=(16, 9)
)
```

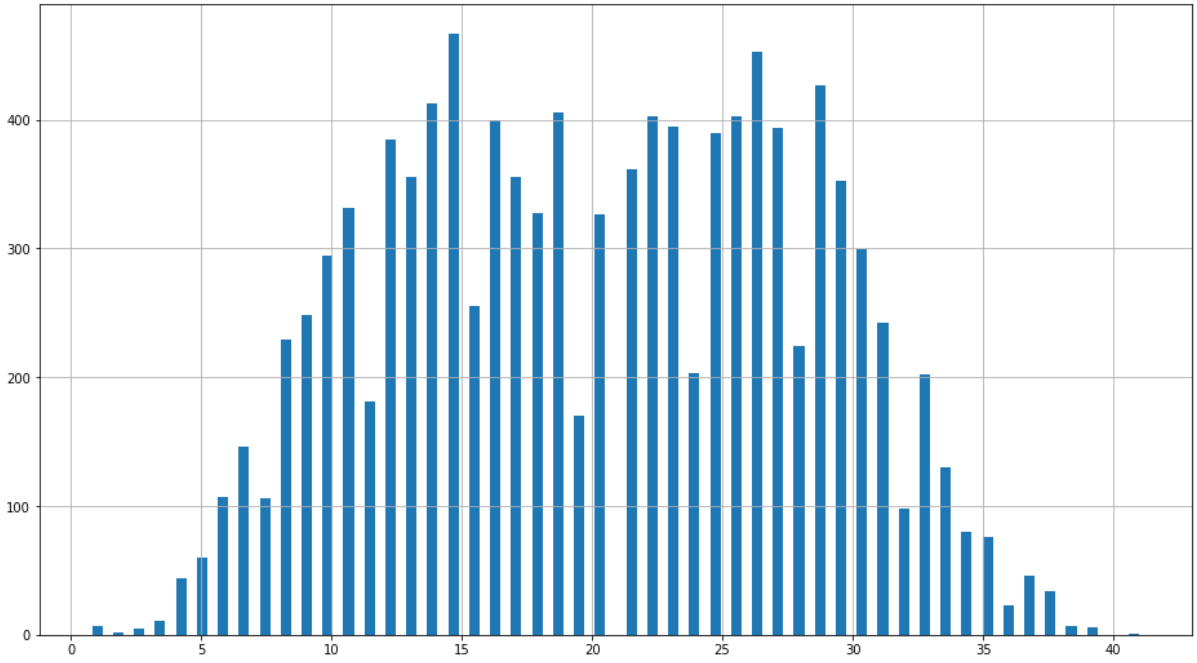
Результат:



Гистограммы

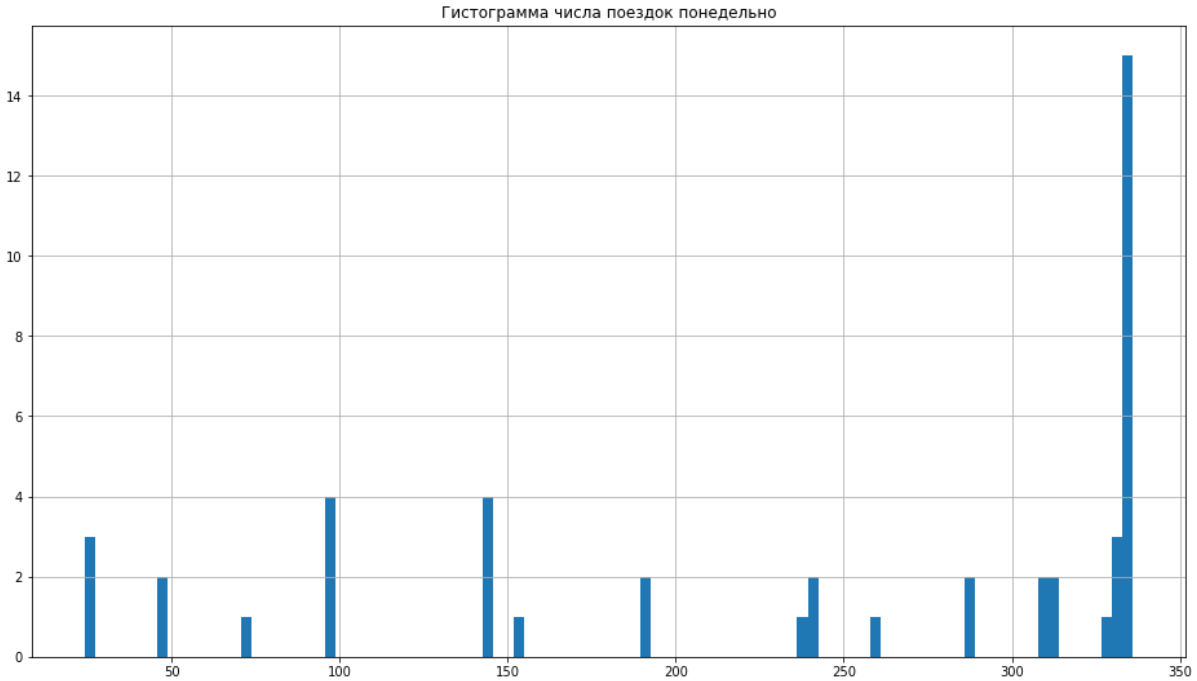
```
# Также в pandas есть встроенная функция для постройки гистограм
df_1['temp'].hist(bins=100, figsize=(16, 9))
# рисуем распределение температур
```

Результат:




```
ax = rents_by_week.hist(bins=100, figsize=(16, 9))
# объект графика возвращается функцией .hist - его можно положить в переменную
# затем добавлять все, что хотим. Добавим title
ax[0, 0].set_title('Гистограмма числа поездок понедельно')
```

Результат:



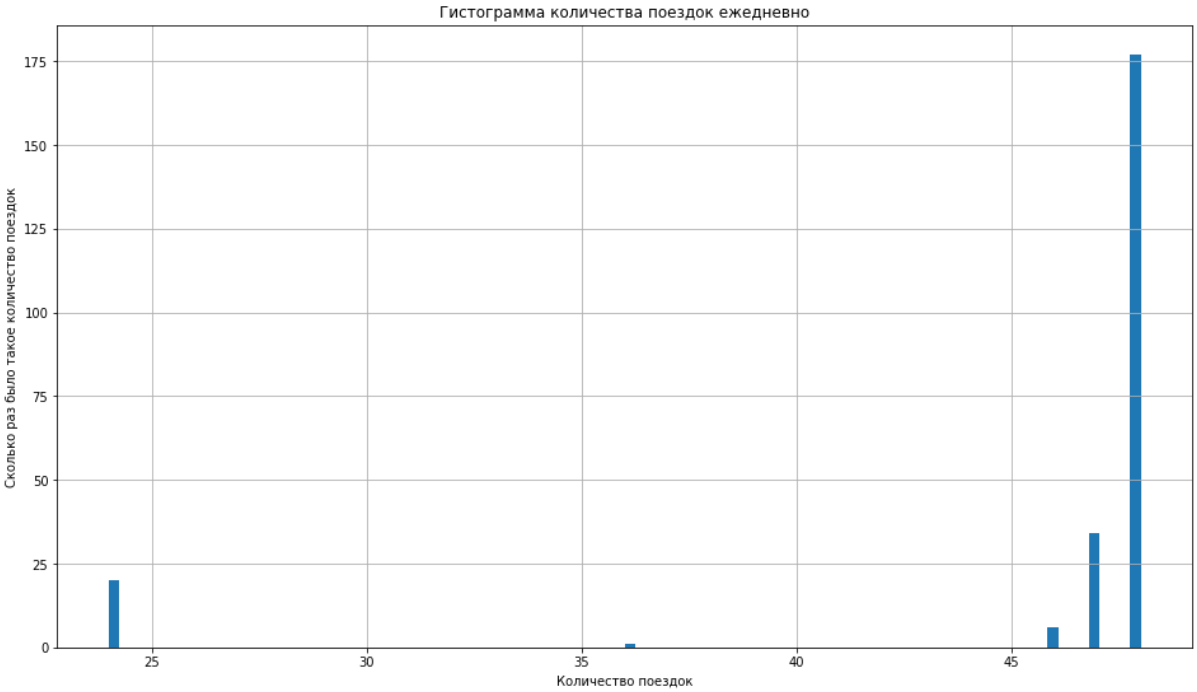
Во втором случае мы сохранили результат функции `.hist()` – массив с объектами типа `Axes` в переменную `ax`, и теперь можем им манипулировать (в данном случае – добавить название графика).

Видим, что в какие-то недели было очень мало поездок по сравнению с другими.

Кажется, что более показательным будет анализ по дням. Давайте сделаем это.

```
ax = df_1.groupby(
    df_1['datetime'].dt.dayofyear  # группируем по дням
).agg({
    'datetime': 'count'  # считаем количество записей в группе
}).hist(  # строим гистограмму
    bins=100,
    figsize=(16, 9)
)
ax = ax[0, 0]  # matplotlib возвращает двумерный массив, распакуем его
ax.set_title('Гистограмма количества поездок ежедневно')
ax.set_xlabel('Количество поездок')
ax.set_ylabel('Сколько раз было такое количество поездок')
```

Результат:



Интересный график. Похоже, в большинстве случаев число поездок не сильно разбрасывалось и лежало в диапазоне 45-50, но бывали и дни, когда число поездок падало в два раза.

P.S.: если явно не указывать левый и правый конец графика, то они подбираются так, чтобы график вместил все данные. То есть, по графику выше можно сделать

вывод, что число поездок в день меньше 50 для всех дней.

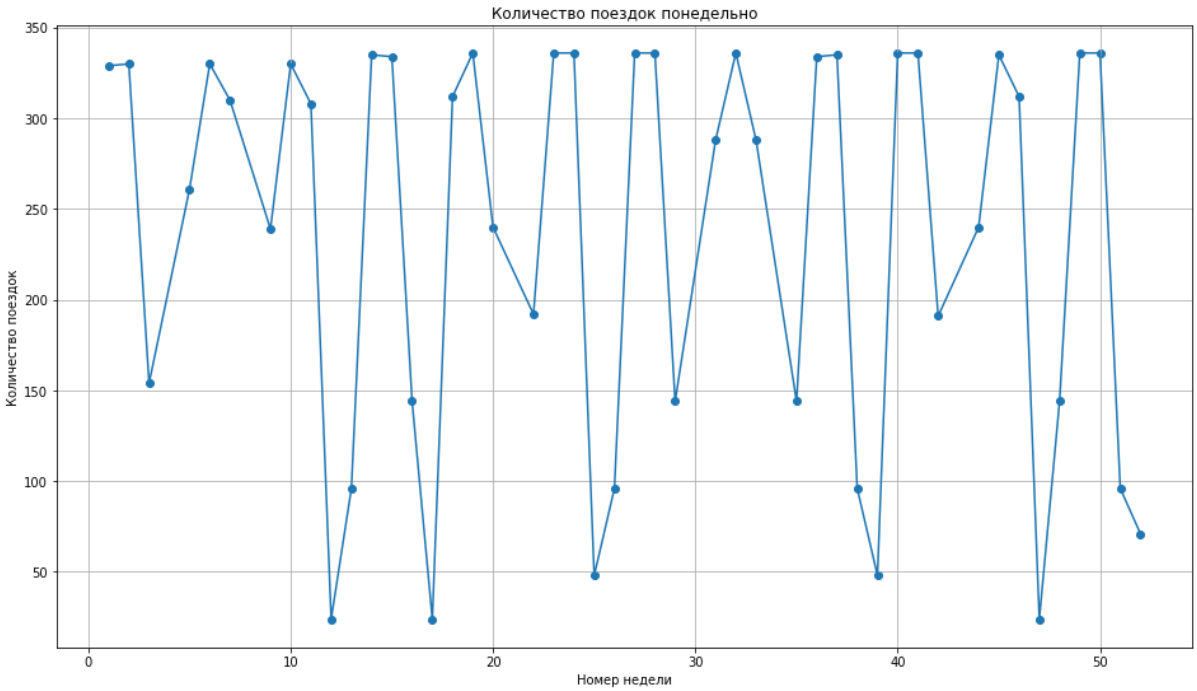
Точная настройка через matplotlib

На самом деле, все функции выше использовали `matplotlib` для визуализации.

Мы также можем самостоятельно отрисовать те же графики через `matplotlib`, что называется, вручную.

```
# Через matplotlib.pyplot можно контролировать все аспекты
import matplotlib.pyplot as plt
fig, ax = plt.subplots(1, 1, figsize=(16, 9))
ax.plot(rents_by_week)
# добавим еще точки на тот же график - этого не делали в pandas
ax.scatter(
    # по оси X - индекс. Наша неделя ушла в индекс после группировки
    rents_by_week.index,
    # по оси Y - значение колонки 0. Она называется temp, можно было по имени обратиться
    rents_by_week.iloc[:, 0]
)
ax.set_xlabel('Номер недели')
ax.set_ylabel('Количество поездок')
ax.set_title('Количество поездок понедельно')
ax.grid(True)
fig.show()
```

Результат:



Как видно, рисовать через `matplotlib` напрямую не совсем удобно - надо писать много кода и знать названия методов. Рекомендуем

пользоваться `df.hist()` и `df.plot()` для рисования графиков, либо же сторонними библиотеками наподобие `seaborn`.

К визуализации данных еще вернемся в блоке "Машинное обучение".

> Сохранение данных

Любой `DataFrame` можно выгрузить в файл, причём в разных форматах. Мы будем учиться выгружать в CSV и Excel.

CSV

Про CSV уже говорилось в начале занятия – и в него выгружать очень просто:

```
rents_by_week.to_csv('rents_by_week.csv', sep=';')
# sep - это разделитель. По умолчанию запятая,
# но ";" лучше читается, если открывать файл в русском Excel
```

Excel

Тут немного сложнее. Для записи в Excel есть несколько библиотек, и они не поставляются вместе с `pandas` – надо ставить отдельно.

Мы будем использовать `openpyxl`. Установим ее:

```
!pip install openpyxl
```

```
# Теперь можем писать в excel
# Обратите внимание на engine='openpyxl' - эту библиотеку мы только что установили
rents_by_week.to_excel('rents_by_week.xlsx', engine='openpyxl')
```

```
# Полученный файл можно открыть в MS Excel, а можно и в pandas
pd.read_excel('rents_by_week.xlsx', engine='openpyxl').head(5)
```

Аргумент `engine='openpyxl'` является необязательным.

Если его пропустить, то `pandas` будет пытаться определить самостоятельно движок для обработки файла. На практике это может вылиться в то, что он попросит установить дополнительные библиотеки, поэтому мы советуем явно установить один движок и везде указывать его в `pandas`. В этом случае вы будете контролировать зависимости проекта и четко знать, кого винить в случае проблем

с файлами Excel. Более подробно про работу `pd.read_excel()` можно прочитать в документации.