Задание на 08.11.2022

Работа с Pandas

Для справки

Структуры данных: серии и датафреймы

Серии — одномерные массивы данных. Они очень похожи на списки, но отличаются по поведению — например, операции применяются к списку целиком, а в сериях — поэлементно.

То есть, если список умножить на 2, получите тот же список, повторенный 2 раза.

```
> vector = [1, 2, 3]
> vector * 2
[1, 2, 3, 1, 2, 3]
```

А если умножить серию, ее длина не изменится, а вот элементы удвоятся.

```
> import pandas as pd
> series = pd.Series([1, 2, 3])
> series * 2
0    2
1    4
2    6
dtype: int64
```

Обратите внимание на первый столбик вывода. Это индекс, в котором хранятся адреса каждого элемента серии. Каждый элемент потом можно получать, обратившись по нужному адресу.

```
> series = pd.Series(['foo', 'bar'])
> series[0]
'foo'
```

Еще одно отличие серий от списков — в качестве индексов можно использовать произвольные значения, это делает данные нагляднее. Представим, что мы анализируем помесячные продажи. Используем в качестве индексов названия месяцев, значениями будет выручка:

```
> months = ['jan', 'feb', 'mar', 'apr']
> sales = [100, 200, 300, 400]
> data = pd.Series(data=sales, index=months)
> data
jan    100
feb    200
mar    300
apr    400
dtype: int64
```

Теперь можем получать значения каждого месяца:

```
> data['feb']
200
```

Так как серии — одномерный массив данных, в них удобно хранить измерения по одному. На практике удобнее группировать данные вместе. Например, если мы анализируем помесячные продажи, полезно видеть не только выручку, но и количество проданных товаров, количество новых клиентов и средний чек. Для этого отлично подходят датафреймы.

Датафреймы — это таблицы. У их есть строки, колонки и ячейки.

Технически, колонки датафреймов — это серии. Поскольку в колонках обычно описывают одни и те же объекты, то все колонки делят один и тот же индекс:

```
> months = ['jan', 'feb', 'mar', 'apr']
> sales = {
...    'revenue':     [100, 200, 300, 400],
...    'items_sold':  [23, 43, 55, 65],
...    'new_clients': [10, 20, 30, 40]
...}
> sales_df = pd.DataFrame(data=sales, index=months)
> sales_df
     revenue  items_sold  new_clients
jan      100          23           10
feb      200          43           20
mar      300          55           30
apr      400          65           40
```

Создаем датафреймы и загружаем данные

Бывает, что мы не знаем, что собой представляют данные, и не можем задать структуру заранее. Тогда удобно создать пустой датафрейм и позже наполнить его данными.

```
> df = pd.DataFrame()
```

А иногда данные уже есть, но хранятся в переменной из стандартного Python, например, в словаре. Чтобы получить датафрейм, эту переменную передаем в ту же команду:

```
> df = pd.DataFrame(data=sales, index=months))
```

Случается, что в некоторых записях не хватает данных. Например, посмотрите на список goods_sold — в нём продажи, разбитые по товарным категориям. За первый месяц мы продали машины, компьютеры и программное обеспечение. Во втором машин нет, зато появились велосипеды, а в третьем снова появились машины, но велосипеды исчезли:

```
> goods_sold = [
...     {'computers': 10, 'cars': 1, 'soft': 3},
...     {'computers': 4, 'soft': 5, 'bicycles': 1},
```

```
...     {'computers': 6, 'cars': 2, 'soft': 3}
... ]
```

Если загрузить данные в датафрейм, Pandas создаст колонки для всех товарных категорий и, где это возможно, заполнит их данными:

```
> pd.DataFrame(goods_sold)
   bicycles  cars  computers  soft
0       NaN   1.0         10     3
1       1.0   NaN          4     5
2       NaN   2.0          6     3
```

Обратите внимание, продажи велосипедов в первом и третьем месяце равны NaN — расшифровывается как Not a Number. Так Pandas помечает отсутствующие значения.

Теперь разберем, как загружать данные из файлов. Чаще всего данные хранятся в экселевских таблицах или csv-, tsv- файлах.

Экселевские таблицы читаются с помощью команды pd.read_excel(). Параметрами нужно передать адрес файла на компьютере и название листа, который нужно прочитать. Команда работает как с xls, так и с xlsx:

```
> pd.read_excel('file.xlsx', sheet_name='Sheet1')
```

Файлы формата csv и tsv — это текстовые файлы, в которых данные отделены друг от друга запятыми или табуляцией:

```
# CSV
month,customers,sales
feb,10,200

# TSV
month\tcustomers\tsales
feb\t10\t200
```

Оба читаются с помощью команды .read_csv(), символ табуляции передается параметром sep (от англ. *separator* — разделитель):

```
> pd.read_csv('file.csv')
> pd.read_csv('file.tsv', sep='\t')
```

При загрузке можно назначить столбец, который будет индексом. Представьте, что мы загружаем таблицу с заказами. У каждого заказа есть свой уникальный номер, Если назначим этот номер индексом, сможем выгружать данные командой df[order_id]. Иначе придется писать фильтр df[df[‘id’] == order_id].

Чтобы назначить колонку индексом,
добавим команду read_csv() параметр index_col, равный названию нужной колонки:

```
> pd.read_csv('file.csv', index_col='id')
```

Исследуем загруженные данные

Представим, что мы анализируем продажи американского интернет-магазина. У нас есть данные о заказах и клиентах (orders.csv и customers.csv). Загрузим файл с продажами интернет-магазина в переменную orders. Раз загружаем заказы, укажем, что колонка id пойдет в индекс:

```
> orders = pd.read_csv('orders.csv', index_col='id')
```

У любого датафрейма есть четыре атрибута: .shape, .columns, .index и .dtypes.

.shape показывает, сколько в датафрейме строк и колонок. Он возвращает пару значений (n_rows, n_columns). Сначала идут строки, потом колонки.

```
> orders.shape
(5009, 5)
```

В датафрейме 5009 строк и 5 колонок.

Масштаб оценили. Теперь посмотрим, какая информация содержится в каждой колонке. С помощью .columns узнаем названия колонок:

```
> orders.columns
Index(['order_date', 'ship_mode', 'customer_id', 'sales'], dtype='object')
```

Теперь видим, что в таблице есть дата заказа, метод доставки, номер клиента и выручка.

С помощью .dtypes узнаем типы данных, находящихся в каждой колонке и поймем, надо ли их обрабатывать. Бывает, что числа загружаются в виде текста. Если мы попробуем сложить две текстовых значения '1' + '1', то получим не число 2, а строку '11':

```
> orders.dtypes
order_date      object
ship_mode       object
customer_id     object
sales          float64
dtype: object
```

Тип object — это текст, float64 — это дробное число типа 3,14.

С помощью атрибута .index посмотрим, как называются строки:

```
> orders.index
Int64Index([100006, 100090, 100293, 100328, 100363, 100391, 100678, 100706,
            100762, 100860,
            ...
            167570, 167920, 168116, 168613, 168690, 168802, 169320, 169488,
            169502, 169551],
```

```
        dtype='int64', name='id', length=5009)
```

Ожидаемо, в индексе датафрейма номера заказов: 100762, 100860 и так далее.

В колонке sales хранится стоимость каждого проданного товара. Чтобы узнать разброс значений, среднюю стоимость и медиану, используем метод .describe():

```
> orders.describe()
        sales
count   5009.0
mean     458.6
std      954.7
min        0.6
25%       37.6
50%      152.0
75%      512.1
max    23661.2
```

Наконец, чтобы посмотреть на несколько примеров записей датафрейма, используем команды .head() и .sample(). Первая возвращает 6 записей из начала датафрейма. Вторая — 6 случайных записей:

```
> orders.head()
       order_date s    hip_mode    customer_id    sales
id
100006  2014-09-07  Standard    DK-13375  377.970
100090  2014-07-08  Standard    EB-13705  699.192
100293  2014-03-14  Standard    NF-18475   91.056
100328  2014-01-28  Standard    JC-15340    3.928
100363  2014-04-08  Standard    JM-15655   21.376
```

Получив первое представление о датафреймах, теперь обсудим, как доставать из него данные.

Получаем данные из датафреймов

Данные из датафреймов можно получать по-разному: указав номера колонок и строк, использовав условные операторы или язык запросов.

Указываем нужные строки и колонки

Продолжаем анализировать продажи интернет-магазина, которые загрузили в предыдущем разделе. Допустим, я хочу вывести столбец sales. Для этого название столбца нужно заключить в квадратные скобки и поставить после них названия датафрейма: orders['sales']:

```
> orders['sales']
id
100006    377.970
100090    699.192
```

```
100293     91.056
100328      3.928
100363     21.376
100391     14.620
100678    697.074
100706    129.440
...
```

Обратите внимание, результат команды — новый датафрейм с таким же индексом.

Если нужно вывести несколько столбцов, в квадратные скобки нужно вставить список с их названиями: orders[['customer_id', 'sales']]. Будьте внимательны: квадратные скобки стали двойными. Первые — от датафрейма, вторые — от списка:

```
> orders[['customer_id', 'sales']]
           customer_id     sales
id
100006    DK-13375   377.970
100090    EB-13705   699.192
100293    NF-18475    91.056
100328    JC-15340     3.928
100363    JM-15655    21.376
100391    BW-11065    14.620
100363    KM-16720   697.074
100706    LE-16810   129.440
...
```

Перейдем к строкам. Их можно фильтровать по индексу и по порядку. Например, мы хотим вывести только заказы 100363, 100391 и 100706, для этого есть команда .loc[]:

```
> show_these_orders = ['100363', '100363', '100706']
> orders.loc[show_these_orders]
           order_date   ship_mode  customer_id    sales
id
100363  2014-04-08  Standard    JM-15655   21.376
100363  2014-04-08  Standard    JM-15655   21.376
100706  2014-12-16    Second    LE-16810  129.440
```

А в другой раз бывает нужно достать просто заказы с 1 по 3 по порядку, вне зависимости от их номеров в таблице. Тогда используют команду .iloc[]:

```
> show_these_orders = [1, 2, 3]
> orders.iloc[show_these_orders]
           order_date   ship_mode   customer_id    sales
id
100090  2014-04-08  Standard    JM-15655   21.376
100293  2014-04-08  Standard    JM-15655   21.376
100328  2014-12-16    Second    LE-16810  129.440
```

Можно фильтровать датафреймы по колонкам и столбцам одновременно:

```
> columns = ['customer_id', 'sales']
> rows = ['100363', '100363', '100706']
```

```
> orders.loc[rows][columns]
            customer_id    sales
id
100363    JM-15655   21.376
100363    JM-15655   21.376
100706    LE-16810  129.440
...
```

Часто вы не знаете заранее номеров заказов, которые вам нужны. Например, если задача — получить заказы, стоимостью более 1000 рублей. Эту задачу удобно решать с помощью условных операторов.

Если — то. Условные операторы

Задача: нужно узнать, откуда приходят самые большие заказы. Начнем с того, что достанем все покупки стоимостью более 1000 долларов:

```
> filter_large = orders['sales'] > 1000
> orders.loc[filter_slarge]
            order_date   ship_mode   customer_id     sales
id
101931  2014-10-28     First    TS-21370  1252.602
102673  2014-11-01  Standard    KH-16630  1044.440
102988  2014-04-05    Second    GM-14695  4251.920
103100  2014-12-20     First    AB-10105  1107.660
103310  2014-05-10  Standard    GM-14680  1769.784
...
```

В сериях все операции применяются по-элементно. Так вот, операция `orders['sales'] > 1000` идет по каждому элементу серии и, если условие выполняется, возвращает `True`. Если не выполняется — `False`. Получившуюся серию мы сохраняем в переменную `filter_large`.

Вторая команда фильтрует строки датафрейма с помощью серии. Если элемент `filter_large` равен `True`, заказ отобразится, если `False` — нет. Результат — датафрейм с заказами, стоимостью более 1000 долларов.

Интересно, сколько дорогих заказов было доставлено первым классом? Добавим в фильтр ещё одно условие:

```
> filter_large = df['sales'] > 1000
> filter_first_class = orders['ship_mode'] == 'First'
> orders.loc[filter_large & filter_first_class]
          order_date    ship_mode customer_id     sales
id
101931  2014-10-28     First    TS-21370  1252.602
103100  2014-12-20     First    AB-10105  1107.660
106726  2014-12-06     First    RS-19765  1261.330
112158  2014-12-02     First    DP-13165  1050.600
116666  2014-05-08     First    KT-16480  1799.970
...
```

Логика не изменилась. В переменную filter_large сохранили серию, удовлетворяющую условию orders['sales'] > 1000. В filter_first_class — серию, удовлетворяющую orders['ship_mode'] == 'First'.

Затем объединили обе серии с помощью логического ‘И’: filter_first_class & filter_first_class. Получили новую серию той же длины, в элементах которой True только у заказов, стоимостью больше 1000, доставленных первым классом. Таких условий может быть сколько угодно.

Язык запросов

Еще один способ решить предыдущую задачу — использовать язык запросов. Все условия пишем одной строкой 'sales > 1000 & ship_mode == 'First' и передаем ее в метод .query(). Запрос получается компактнее.

```
> orders.query('sales > 1000 & ship_mode == First')
          order_date    ship_mode customer_id     sales
id
101931  2014-10-28     First    TS-21370  1252.602
103100  2014-12-20     First    AB-10105  1107.660
106726  2014-12-06     First    RS-19765  1261.330
112158  2014-12-02     First    DP-13165  1050.600
116666  2014-05-08     First    KT-16480  1799.970
...
```

Значения для фильтров можно сохранить в переменной, а в запросе сослаться на нее с помощью символа @: sales > @sales_filter.

```
> sales_filter = 1000
> ship_mode_filter = 'First'
> orders.query('sales > @sales_filter & ship_mode > @ship_mode_filter')


            order_date ship_mode customer_id     sales
id
101931  2014-10-28     First    TS-21370  1252.602
103100  2014-12-20     First    AB-10105  1107.660
106726  2014-12-06     First    RS-19765  1261.330
112158  2014-12-02     First    DP-13165  1050.600
116666  2014-05-08     First    KT-16480  1799.970
...
```

Разобравшись, как получать куски данных из датафрейма, перейдем к тому, как считать агрегированные метрики: количество заказов, суммарную выручку, средний чек, конверсию.

Считаем производные метрики

Задача: посчитаем, сколько денег магазин заработал с помощью каждого класса доставки. Начнем с простого — просуммируем выручку со всех заказов. Для этого используем метод .sum():

```
> orders['sales'].sum()
2297200.8603000003
```

Добавим класс доставки. Перед суммированием сгруппируем данные с помощью метода .groupby():

```
> orders.groupby('ship_mode')['sales'].sum()

ship_mode
First      3.514284e+05
Same Day   1.283631e+05
Second     4.591936e+05
Standard   1.358216e+06
```

3.514284e+05 — научный формат вывода чисел. Означает $3.51 * 10^5$. Нам такая точность не нужна, поэтому можем сказать Pandas, чтобы округлял значения до сотых:

```
> pd.options.display.float_format = '{:,.1f}'.format
> orders.groupby('ship_mode')['sales'].sum()
ship_mode
First       351,428.4
Same Day    128,363.1
Second      459,193.6
Standard  1,358,215.7
```

Теперь видим сумму выручки по каждому классу доставки. По суммарной выручке неясно, становится лучше или хуже. Добавим разбивку по датам заказа:

```
> orders.groupby(['ship_mode', 'order_date'])['sales'].sum()
ship_mode order_date
First     2014-01-06    12.8
          2014-01-11     9.9
          2014-01-14    62.0
          2014-01-15   149.9
          2014-01-19   378.6
          2014-01-26   152.6
...
```

Видно, что выручка прыгает ото дня ко дню: иногда 10 долларов, а иногда 378. Интересно, это меняется количество заказов или средний чек? Добавим к выборке количество заказов. Для этого вместо .sum() используем метод .agg(), в который передадим список с названиями нужных функций.

```
> orders.groupby(['ship_mode', 'order_date'])['sales'].agg(['sum', 'count'])
                          sum  count
ship_mode order_date
First     2014-01-06  12.8      1
          2014-01-11   9.9      1
          2014-01-14  62.0      1
          2014-01-15 149.9      1
          2014-01-19 378.6      1
          2014-01-26 152.6      1
...
```

Получается, что это так прыгает средний чек. Интересно, а какой был самый удачный день? Чтобы узнать, отсортируем получившийся датафрейм: выведем 10 самых денежных дней по выручке:

```
> orders.groupby(['ship_mode', 'order_date'])['sales'].agg(['sum']).sort_values(by='sum',
ascending=False).head(10)
                              sum
ship_mode order_date
Standard  2014-03-18 26,908.4
          2016-10-02 18,398.2
First     2017-03-23 14,299.1
Standard  2014-09-08 14,060.4
First     2017-10-22 13,716.5
Standard  2016-12-17 12,185.1
          2017-11-17 12,112.5
          2015-09-17 11,467.6
          2016-05-23 10,561.0
          2014-09-23 10,478.6
```

Команда разрослась, и её теперь неудобно читать. Чтобы упростить, можно разбить её на несколько строк. В конце каждой строки ставим обратный слеш \:

```
> orders \
... .groupby(['ship_mode', 'order_date'])['sales'] \
... .agg(['sum']) \
... .sort_values(by='sum', ascending=False) \
... .head(10)
                        sum
ship_mode order_date
Standard  2014-03-18 26,908.4
          2016-10-02 18,398.2
First     2017-03-23 14,299.1
Standard  2014-09-08 14,060.4
First     2017-10-22 13,716.5
Standard  2016-12-17 12,185.1
          2017-11-17 12,112.5
          2015-09-17 11,467.6
          2016-05-23 10,561.0
          2014-09-23 10,478.6
```

В самый удачный день — 18 марта 2014 года — магазин заработал 27 тысяч долларов с помощью стандартного класса доставки. Интересно, откуда были клиенты, сделавшие эти заказы? Чтобы узнать, надо объединить данные о заказах с данными о клиентах.

Объединяем несколько датафреймов

До сих пор мы смотрели только на таблицу с заказами. Но ведь у нас есть еще данные о клиентах интернет-магазина. Загрузим их в переменную customers и посмотрим, что они собой представляют:

```
> customers = pd.read_csv('customers.csv', index='id')
> customers.head()
                  name          segment         state          city
id
CG-12520      Claire Gute   Consumer        Kentucky       Henderson
```

```
DV-13045  Darrin Van Huff  Corporate      California      Los Angeles
SO-20335   Sean O'Donnell   Consumer        Florida  Fort Lauderdale
BH-11710  Brosina Hoffman   Consumer      California      Los Angeles
AA-10480     Andrew Allen   Consumer  North Carolina         Concord
```

Мы знаем тип клиента, место его проживания, его имя и имя контактного лица. У каждого клиента есть уникальный номер id. Этот же номер лежит в колонке customer_id таблицы orders. Значит мы можем найти, какие заказы сделал каждый клиент. Например, посмотрим, заказы пользователя CG-12520:

```
> cust_filter = 'CG-12520'
> orders.query('customer_id == @cust_filter')
             order_date ship_mode customer_id   sales
id
CA-2016-152156  2016-11-08    Second    CG-12520  993.90
CA-2017-164098  2017-01-26     First    CG-12520   18.16
US-2015-123918  2015-10-15  Same Day    CG-12520  136.72
```

Вернемся к задаче из предыдущего раздела: узнать, что за клиенты, которые сделали 18 марта заказы со стандартной доставкой. Для этого объединим таблицы с клиентами и заказами. Датафреймы объединяют с помощью методов .concat(), .merge() и .join(). Все они делают одно и то же, но отличаются синтаксисом — на практике достаточно уметь пользоваться одним из них.

Покажем на примере .merge():

```
> new_df = pd.merge(orders, customers, how='inner', left_on='customer_id', right_index=True)
> new_df.columns
Index(['order_date', 'ship_mode', 'customer_id', 'sales', 'name', 'segment',
       'state', 'city'],
      dtype='object')
```

В .merge() я сначала указали названия датафреймов, которые хотим объединить. Затем уточнил, как именно их объединить и какие колонки использовать в качестве ключа.

Ключ — это колонка, связывающая оба датафрейма. В нашем случае — номер клиента. В таблице с заказами он в колонке customer_id, а таблице с клиентами — в индексе. Поэтому в команде мы пишем: left_on='customer_id', right_index=True.

Решаем задачу

Закрепим полученный материал, решив задачу. Найдем 5 городов, принесших самую большую выручку в 2016 году.

Для начала отфильтруем заказы из 2016 года:

```
> orders_2016 = orders.query("order_date >= '2016-01-01' & order_date <= '2016-12-31'")
> orders_2016.head()
       order_date ship_mode customer_id   sales
id
100041 2016-11-20  Standard    BF-10975   328.5
100083 2016-11-24  Standard    CD-11980    24.8
```

```
100153 2016-12-13  Standard    KH-16630    63.9
100244 2016-09-20  Standard    GM-14695   475.7
100300 2016-06-24    Second    MJ-17740 4,823.1
```

Город — это атрибут пользователей, а не заказов. Добавим информацию о пользователях:

```
> with_customers_2016 = pd.merge(customers, orders_2016, how='inner', left_index=True, right_on='customer_id')
```

Сруппируем получившийся датафрейм по городам и посчитаем выручку:

```
> grouped_2016 = with_customers_2016.groupby('city')['sales'].sum()
> grouped_2016.head()
city
Akron              1,763.0
Albuquerque          692.9
Amarillo             197.2
Arlington          5,672.1
Arlington Heights     14.1
Name: sales, dtype: float64
```

Отсортируем по убыванию продаж и оставим топ-5:

```
> top5 = grouped_2016.sort_values(ascending=False).head(5)
> print(top5)
city
New York City   53,094.1
Philadelphia    39,895.5
Seattle         33,955.5
Los Angeles     33,611.1
San Francisco   27,990.0
Name: sales, dtype: float64
```

Готово!

Задание

Возьмите данные о orders.csv и customers.csv и посчитайте:

1. Сколько заказов, отправлено первым классом за последние 3 лет?
2. Сколько в базе клиентов из Калифорнии?
3. Сколько заказов они сделали?
4. Постройте сводную таблицу средних чеков по всем штатам за каждый год.