Манипуляции с Pandas для анализа данных

Авторство Пицик Харитон 211гр.

Введение: что такое Pandas & почему Pandas?

Pandas — библиотека, являющаяся надстройкой над NumPy и предназначенная для обработки табличных данных и временных рядов. (Интересный факт: название библиотеки происходит от слова «панельные данные». Панельные данные - это многомерные данные, полученные в результате серий наблюдений для одних и тех же людей / компаний (термин из эконометрики)).

Pandas обеспечивает работу с данными в рамках среды Python, что позволяет не переключаться на изначально более специализированные инструменты, такие как R.

FunFact: как Pandas отрисовывает таблицы?

В Pandas отрисовка красивых таблиц (обычно это то, что вы видите) в интерактивных средах, таких как Jupyter Notebook (дальнейшее описание подразумевает, что мы работаем именно в нём), реализована с помощью HTML-рендеринга. В Pandas за это отвечает система repr-методов и форматтеров. Как это работает: у объекта DataFrame (например) при вызове функции вывода (head, tail, etc.) срабатывает внутренний метод _repr_html_(). Этот метод возвращает HTML-таблицу с заголовками, ну в общем вы понимаете о чём я. repr довольно уникальная штука, вот поддерживаемые методы:

1. __repr__() — то, что выводится в консоль;
2. _repr_html_() — HTML представление;
3. _repr_latex_() — LaTeXпредставление (!!! если поддерживается !!!);
4. и т.д.

В модуле pandas.io.formatter определены методы для декорирования таблиц. В рамках этой лекции я лишь упомяну **styles.Styler**, который предоставляет возможность кастомизировать визуал с помощью CSS шаблонов. Упомнял я его чтобы уточнить: даже без Styler, вывод DataFrame даёт «приятный» визуал, т.к. при выводе в любом случае вызывается вышеупомянутый _repr_html_().

Считываем данные

Считывание .csv (comma separated values) таблицы — pd.read_csv(). На этом можно было бы закончить лекцию, но данные бывают большими: под большими я подразумеваю просто гигантские, которые по размеру будут серьёзными для нашего компьютера. В pandas мы можем загружать данные чанками (возможно, у кого-то в голове сейчас откликнутся итераторы из Python).

В общем суть такая: нам нужно считывать данные чанками. Сделать это можно с помощью ключевого слова *chunksize* для команды *pd.read_csv()*.

```
for chunk in pd.read_csv('data.csv', chunksize=10000):
    chunk.to_csv('chunked_file.csv', mode = 'a', index = False, header = False)
```

Обратим также внимание на сам объект (как вы могли догадаться, он итерируемый):

```
chunk = pd.read_csv('data.csv', chunksize = 10000)
type(chunk) #pandas.io.parsers.readers.TextFileReader
```

readers.TextFileReader получается в результате read_csv или read_table и позволяет читать огромные CSV-файлы по частям, не загружая всё сразу в память. chunk — *итерируемый*, т.е. мы можем запихнуть это в цикл for, а можем получать по одному с помощью метода get_chunk().

```
#read_table vs read_csv
#помогает для считывание tsv (tab-separated values)
pd.read_table("file.txt")  =  pd.read_csv("file.txt", sep="\t")
```

Оптимизация типов данных

Pandas по умолчанию загружает данные с типами, которые не всегда эффективны. Например, столбцы с числами могут загружаться как float64, даже если достаточно float32, а категориальные данные хранятся как object, вместо category.

Проверить текущие типы данных можно с метода df.dtypes. Самый очевидный (и удобный) способ приведения типов — функции Pandas:

```
df['col'] = df['col'].astype(int)

df[['col1', 'col2']].astype(int)

df.astype({'col1': int, 'col2': int, 'col3': float})

#еще есть метод to_numeric (unsigned int)
df['col'] = pd.to_numeric(df['col'], errors = 'coerce')

#Если в столбце много повторяющихся значений, category может сэкономить память
df['category_column'] = df['category_column'].astype('category')

#Если явно не указано явно, Pandas преобразует даты в object.
df['date'] = pd.to_datetime(df['date'], errors='coerce')


#Если много разреженных типов, можем смело использовать SparseArray
df['sparse_column'] = pd.arrays.SparseArray(df['dense_column'])
```

Также для оптимизации типов поддерживается pyarrow:

```
df['col'] = df['col'].astype('string[pyarrow]')
```

так Pandas использует тип Arrow-backed string dtype, которые эффективные (за счёт поддержки ранее обсуждаемой SIMD) и хранятся в бинарном формате.

Есть метод convert_dtypes, который пытается сам преобразовать типы в наиболее подходящие согласно иерархии типов.

Векторизация для операций Pandas (numba, cython)

Раз уж мы начали обсуждать что-то, связанное с оптимизацией, давайте рассмотрим, как применется векторизация в Pandas. (Вернее, ускорение операций)

Numba

Numba ускоряет непосредственно сам Python. Он является JIT-компилятором (just-in-time, динамическая компиляция — технология, в которой байт-код преобразовывается в машинный непосредственно во время работы программы), который «очень любит» циклы и работу с NumPy. Давайте рассмотрим примитивный кейс использования numba. Перед этим добавлю, что Numba переводит подмножество NumPy и Python операций в машинный код оптимизированным способом через LLVM (low level virtual machine) с помощью пакета llvmlite для Python.

```
import pandas as pd
import numba
#100.000 строк, 4 колонки
df = pd.DataFrame(np.random.randint(0,100,size=(100000, 4)),columns=['a', 'b', 'c', 'd'])
```

```
def add_new_col(x):
    return x * 5

@numba.vectorize
def add_new_col_numba(x):
    return x * 5
```

Результаты следующие:

```
# наша функция
In [1]: %timeit df['new_col'] = df['a'].apply(multiply)
23.9 ms ± 1.93 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

# встроенная имплементация Pandas
In [2]: %timeit df['new_col'] = df['a'] * 5
545 µs ± 21.4 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)

# наша функция с numba
# мы отдаем весь вектор значений, чтобы numba сам провел оптимизацию цикла
In [3]: %timeit df['new_col'] = multiply_numba(df['a'].to_numpy())
329 µs ± 2.37 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Таким образом, «оптимизированная» версия быстрее в 70 раз! Однако в случаее абсолютных величин, реализация в Pandas не сильно отстала от numb'ы. Давайте рассмотрим более сложный кейс:

```
# возводим значения строки в квадрат и берем их среднее
def square_mean(row):
    row = np.power(row, 2)
    return np.mean(row)
# применение:
# df['new_col'] = df.apply(square_mean, axis=1)

# numba не умеет работать с примитивами pandas (Dataframe, Series и тд.)
# поэтому мы даем ей двумерный массив numpy
@numba.njit
def square_mean_numba(arr):
    res = np.empty(arr.shape[0])
    arr = np.power(arr, 2)
    for i in range(arr.shape[0]):
        res[i] = np.mean(arr[i])
    return res
# применение:
# df['new_col'] = square_mean_numba(df.to_numpy())
```

По этому графику мы видим, что результаты уже разительно отличаются:

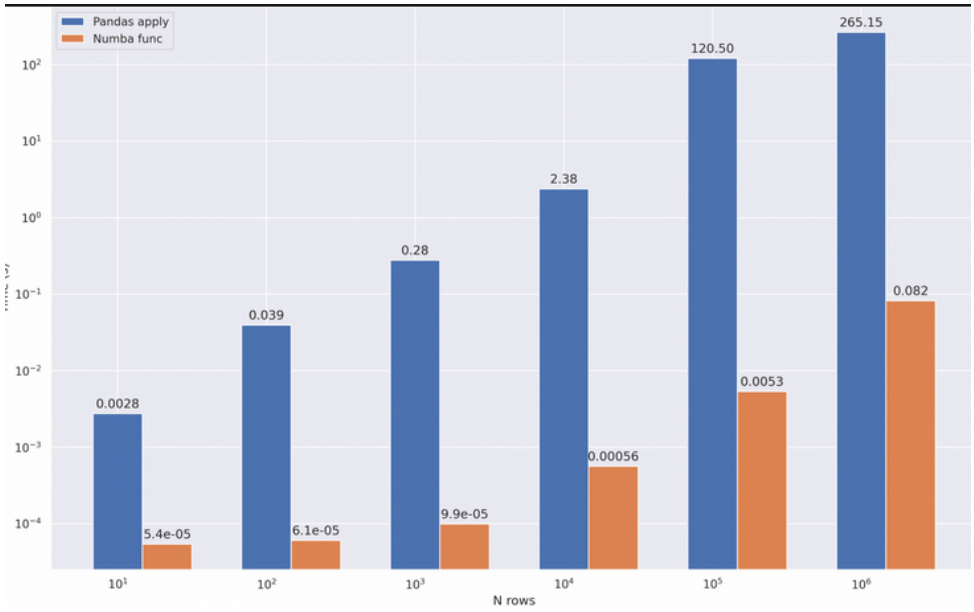


Рис. 1: Numba vs. Pandas

Cython

В большинстве случаев, конечно, Python и NumPy хватает для большинства вычислительных процессов, однако если они становятся слишком вычислительно сложными или просто требуется их ускорить, применяется модуль **Cython**.

Рассмотрим следующий блок кода, в котором мы реализуем функцию для дальнейшего **.apply**:

```
In [3]: def f(x):
   ...:     return x * (x - 1)
   ...:

In [4]: def integrate_f(a, b, N):
   ...:     s = 0
   ...:     dx = (b - a) / N
   ...:     for i in range(N):
   ...:         s += f(a + i * dx)
   ...:     return s * dx
   ...:
```

Получим следующее время:

```
In [5]: %timeit df.apply(lambda x: integrate_f(x["a"], x["b"], x["N"]), axis=1)
84 ms +- 1.01 ms per loop (mean +- std. dev. of 7 runs, 10 loops each)
```

С помощью функции IPython **prun** можно убедиться, что большая часть времени теряется именно в этих функциях.

Теперь давайте скопируем нашу функцию, но под Cython:

```
In [8]: %%cython
   ...: def f_plain(x):
   ...:     return x * (x - 1)
   ...: def integrate_f_plain(a, b, N):
   ...:     s = 0
   ...:     dx = (b - a) / N
   ...:     for i in range(N):
   ...:         s += f_plain(a + i * dx)
   ...:     return s * dx
   ...:
```

Результат:

```
In [9]: %timeit df.apply(lambda x: integrate_f_plain(x["a"], x["b"], x["N"]), axis=1)
47.2 ms +- 366 us per loop (mean +- std. dev. of 7 runs, 10 loops each)
```

Индексация и отличия от NumPy

Pandas предоставляет несколько вариантов индексации, почему бы нам об этом не поговорить.

LOC, ILOC

Индексация по локальному и фактическому индексам, позволяет выбрать набор строк / столбцов по заданным *лейблам*.

ILOC позволяет индексировать, по факту, как мы привыкли индексировать массивы (начиная с 0 и т.д.). Это ещё называется *неявной индексацией*. Давайте быстренько рассмотрим пример:

```
>>> mydict = [{'a': 1, 'b': 2, 'c': 3, 'd': 4},
...           {'a': 100, 'b': 200, 'c': 300, 'd': 400},
...           {'a': 1000, 'b': 2000, 'c': 3000, 'd': 4000}]
>>> df = pd.DataFrame(mydict)
>>> df
      a     b     c     d
0     1     2     3     4
1   100   200   300   400
2  1000  2000  3000  4000


>>> df.iloc[[0]]
   a  b  c  d
0  1  2  3  4
>>> type(df.iloc[[0]])
<class 'pandas.core.frame.DataFrame'>
```

LOC же, в свою очередь, является *явной индексацией*. Грубо говоря это означает, что мы можем использовать индексы, которые видим в таблице. Опять же, рассмотрим пример

```
>>> df = pd.DataFrame([[1, 2], [4, 5], [7, 8]],
...                   index=['cobra', 'viper', 'sidewinder'],
...                   columns=['max_speed', 'shield'])
>>> df
            max_speed  shield
cobra               1       2
viper               4       5
sidewinder          7       8

>>> df.loc['viper']
max_speed    4
shield       5
Name: viper, dtype: int64
```

FunFact: в более ранних версиях Pandas есть метод .x, который позволяет объединять .loc и .iloc в одну индексацию.

AT, IAT

Метод at позволяет получить **один** элемент, находящийся на пересечении выбранных строки и столбца и делает это *явно*, по аналогии с loc:

```
df = pd.DataFrame([[0, 2, 3], [0, 4, 1], [10, 20, 30]],
                  index=[4, 5, 6], columns=['A', 'B', 'C'])
```

```
df
>>  A   B   C
4   0   2   3
5   0   4   1
6  10  20  30

df.at[4, 'B']
>> 2

#тоже самое
df.loc[4].at['B']
>> 2
```

iat, как вы могли догадаться, позволяет делать то же самое но по *неявному* индексу:

```
df = pd.DataFrame([[0, 2, 3], [0, 4, 1], [10, 20, 30]],
                  columns=['A', 'B', 'C'])

df
>>  A   B   C
0   0   2   3
1   0   4   1
2  10  20  30

df.iat[1, 2]
>> 1
```

+

MultiIndex

Объект MultiIndex позволяет воспроизвести более сложные конструкции, например:

```
index = [('California', 2000), ('California', 2010),
         ('New York', 2000), ('New York', 2010),
         ('Texas', 2000), ('Texas', 2010)]

populations = [33871648, 37253956,
               18976457, 19378102,
               20851820, 25145561]


data = pd.Series(populations, index=index)
index = pd.MultiIndex.from_tuples(index)
data = data.reindex(index)
data

>> California  2000    33871648
               2010    37253956
   New York    2000    18976457
               2010    19378102
   Texas       2000    20851820
               2010    25145561
   dtype: int64
```

Можно думать, что MultiIndex — это некое новое измерение наших данных. Вы могли бы заметить, что вообще-то, мы могли бы представить наши данные и без подобного «оверхеда», но иногда представление мультииндексом бывает полезным:

```
data_df = data.unstack()
data_df
```

	2000	2010
California	33871648	37253956
New York	18976457	19378102
Texas	20851820	25145561

Рис. 2: unstack (прошу прощения за качество)

Обработка NaN значений

Как в целом это устроено

Зачастую приходиться обрабатывать так называемые Na значения в данных, но что вы вообще понимаете под Na? Одинаковые ли они все? Как Pandas отличает Na от реально существующих данных?

Давайте рассмотрим 2 схемы, по которым ЯП определяют отсутствие данных: использование маски, которая глобально идентифицирует отсутствующие данные, или какое-то заданное значение для каждого типа (*сигнальный* метод).

В случае маскирования, сама маска представляет собой булевый массив, или же может присваивать бит в представлении данных там, «где их нет» (для локального указания нулевого статуса значения).

В сигнальном методе, как я уже и говорил, применяется специальное значение для обозначения отсутствующей даты (например, -9999 для int или любой другой, более сложный паттерн). Например, это также может быть NaN для типов с плавающей точкой, по соглашению института инженеров электротехники и электроники (IEEE).

Возможно вы догадываетесь, но здесь нет во всём хорошего решения. В случае с масками, необходимо аллоцировать память под дополнительный булевый массив, размерность которого совпадает с датасетом, что понятное дело добавляет оверхеда для итак непростой задачи. Сигналы же ограничивают область значений типа, что может вовлечь за собой дополнительную (часто неоптимизированную) логику вычислений в CPU (и GPU). Возвращаясь к первому предложению (нет хорошего во всём решения), дополню, что разные языки используют разные соглашения. Не ходя далеко от темы, язык R использует зарезервированные битовые шаблоны в каждом типе данных как сигнальные значения, указывающие на отсутствие данных.

Наконец-то возвращаемся к пандам

Способ, которым Pandas обрабатывает отсутствующие значения, ограничен его зависимостью от пакета NumPy, который не имеет встроенного понятия значений NA для типов данных, не являющихся точками с плавающей запятой. Pandas решил использовать sentinels для недостающих данных и далее решил использовать два уже существующих в Python нулевых значения: специальное значение NaN с плавающей точкой и объект Python None.

Всё что пока-что нам нужно знать про None из Python, что это если вы решите аггрегировать / проитерироваться на объекте с None внутри — вы вылетите в ошибку. np.nan же в свою очередь работает как вирус: любой метод, возвращающий что-либо (кроме специализированных) и хоть как-то взаимодействующий с np.nan вернёт np.nan:

```
arr = np.array([np.nan, 2, 3, 4])
arr.min(), arr.max(), arr.sum()
```

Встречаем специализированные методы:

```
arr = np.array([np.nan, 2, 3, 4])
arr.nanmin(), arr.nanmax(), arr.nansum()
```

Обратите внимание на следующий каст в Python:

```
x = pd.Series(range(2), dtype=int)
x
>>0    0
  1    1
  dtype: int64

x[0] = None
x
>> 0    NaN
   1    1.0
   dtype: float64
```

Команды для обработки пропущенных значений

```
data = pd.Series([1, np.nan, 'hello', None])
data.isnull()
>> 0    False
   1     True
   2    False
   3     True
   dtype: bool

data[data.notnull()]
>>  0        1
    2    hello
    dtype: object

data.dropna()
>> 0        1
   2    hello
   dtype: object
```

Здесь затронуты примитивные команды, но на dropna и fillna (см. ниже) хочу остановиться подробнее.

Для dropna хочу акцентировать внимание на ключевых словах **thresh** и **how**. Thresh — принимает int и означает минимальное количество непустых значений, при котором строка / столбец *не удаляется*. How (по умолчанию 'any') означает «стратегию», по которой будут удаляться строки / столбцы. При значении 'all' удалятся только те строки / столбцы, которые везде содержат пустоту.

The default is `how='any'`, such that any row or column (depending on the `axis` keyword) containing a null value will be dropped. You can also specify `how='all'`, which will only drop rows/columns that are *all* null values:

```
In [20]: df[3] = np.nan
         df
```

Out[20]:

	0	1	2	3
0	1.0	NaN	2	NaN
1	2.0	3.0	5	NaN
2	NaN	4.0	6	NaN

```
In [21]: df.dropna(axis='columns', how='all')
```

Out[21]:

	0	1	2
0	1.0	NaN	2
1	2.0	3.0	5
2	NaN	4.0	6

For finer-grained control, the `thresh` parameter lets you specify a minimum number of non-null values for the row/column to be kept:

```
In [22]: df.dropna(axis='rows', thresh=3)
```

Out[22]:

	0	1	2	3
1	2.0	3.0	5	NaN

Here the first and last row have been dropped, because they contain only two non-null values.

Рис. 3: dropna

Также есть метод fillna который просто заполняет пустые значения указанным значением. Однако стоит уточнить про его ключевое слово method, которое позволяет выбрать стратегию как именно заполнять. например, method= 'ffil' (forward-fill) позволяет «толкать вперёд» предыдущее значение (угадайте что делает bfill) (Для этих методов можно указывать ось, для fillna с method это особенно имеет смысл писать явно).

Слияние и конкатенация

Давайте рассмотрим методы и стратегии конкатенации, слияния данных.

Concat

pd.concat, как можно было догадаться, просто конкатенирует данные относительно заданной оси

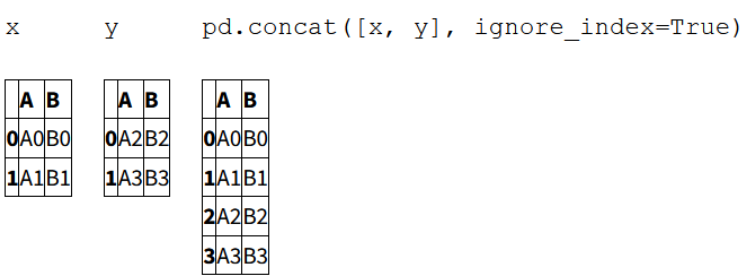
```
ser1 = pd.Series(['A', 'B', 'C'], index=[1, 2, 3])
ser2 = pd.Series(['D', 'E', 'F'], index=[4, 5, 6])
pd.concat([ser1, ser2])
```

```
>>  1    A
    2    B
    3    C
    4    D
    5    E
```

```
6    F
dtype: object
```

Удивительно, но факт: это также работает с DataFrame. Однако здесь стоит обратить внимание: индексация не сохраняется, и её приходится перезаписывать... могли бы подумать вы, однако можно (если это будет удобно) прописать ключевое слово ignore_index:

```
pd.concat([df_x, df_y], ignore_index = True)
```



	A	B
0	A0	B0
1	A1	B1

	A	B
0	A2	B2
1	A3	B3

	A	B
0	A0	B0
1	A1	B1
2	A2	B2
3	A3	B3

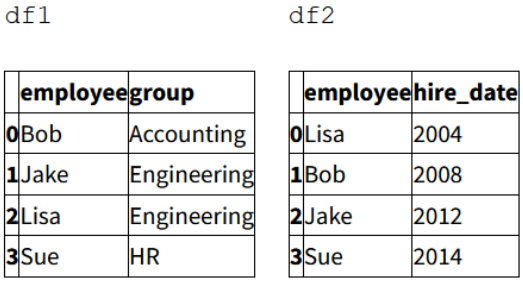
Рис. 4: Конкатенация без конфликтов по индексу

Можно указать метод join, который позволяет указать метод присоединения, например `join='inner'` означает, что будут объединены только пересекающиеся колонки. Также просто упомяну метод `join_axes`, в котором можно указать, по каким строкам / столбцам будет оформлено присоединение.

Merge

Для объединения данных есть ещё продвинутые методы, один из таких — merge. С ним немного посложнее, так как в зависимости от ситуации нужно понимать, как именно он соединит данные. Рассмотрим основные ситуации (надеюсь никто ещё не расстраивается, что я беру готовые мини-датасеты):

Один-к-одному



	employee	group
0	Bob	Accounting
1	Jake	Engineering
2	Lisa	Engineering
3	Sue	HR

	employee	hire_date
0	Lisa	2004
1	Bob	2008
2	Jake	2012
3	Sue	2014

Рис. 5: Один к одному

	employee	group	hire_date
0	Bob	Accounting	2008
1	Jake	Engineering	2012
2	Lisa	Engineering	2004
3	Sue	HR	2014

Рис. 6: Результат

В целом это самый простой случай объединения. Всё можно понять по картинкам.

Многие-к-одному

```
df3
```

	employee	group	hire_date
0	Bob	Accounting	2008
1	Jake	Engineering	2012
2	Lisa	Engineering	2004
3	Sue	HR	2014

```
df4
```

	group	supervisor
0	Accounting	Carly
1	Engineering	Guido
2	HR	Steve

```
pd.merge(df3, df4)
```

	employee	group	hire_date	supervisor
0	Bob	Accounting	2008	Carly
1	Jake	Engineering	2012	Guido
2	Lisa	Engineering	2004	Guido
3	Sue	HR	2014	Steve

Рис. 7: Многие к одному

Многие к одному — случай, когда один из датасетов содержит дубликаты в столбце-ключе (общий столбец, по которому объединяются данные. В тривиальных случаях Pandas определяет их самостоятельно, но всё равно лучше прописывать явно с помощью ключевого слова on). В этом случае возвращается DataFrame с сохранёнными дубликатами.

Многие-к-многим

Ситуация, когда обе ключевые колонки объединяемых датасетов содержат дубликаты:

```
df1
```

	employee	group
0	Bob	Accounting
1	Jake	Engineering
2	Lisa	Engineering
3	Sue	HR

```
df5
```

	group	skills
0	Accounting	math
1	Accounting	spreadsheets
2	Engineering	coding
3	Engineering	linux
4	HR	spreadsheets
5	HR	organization

```
pd.merge(df1, df5)
```

	employee	group	skills
0	Bob	Accounting	math
1	Bob	Accounting	spreadsheets
2	Jake	Engineering	coding
3	Jake	Engineering	linux
4	Lisa	Engineering	coding
5	Lisa	Engineering	linux
6	Sue	HR	spreadsheets
7	Sue	HR	organization

Рис. 8: Многие к многим

В качестве некой «модификации» ключевого слова on есть методы `left_on, right_on`. Они пригождаются, когда ключевые колонки в разных датасетах имеют разные названия. Тогда мы указываем для left_on колонку из датасета, **к которому** хотим слить данные, а для right_on колонку из датасета, **который** хотим слить с исходным.

Продвинутая трансформация данных

Объект GroupBy и метод groupby

Тут не такая простая тема, как может показаться на первый взгляд.

Иногда (часто) нам удобно аггрегировать по заданному индексу / лейблу. С этой задачей нам может помочь метод groupby. Подкованные люди могли заметить, что это референс на SQL, где также есть groupby (и join и merge, в этом плане в целом они похожи). Как происходит groupby:

1. разделение по значению выбранного ключа / индекса;
2. применение аггрегирующей функции;
3. слияние разделившихся данных в результат.

Давайте рассмотрим подробнее. В этом нам поможет следующая картинка:

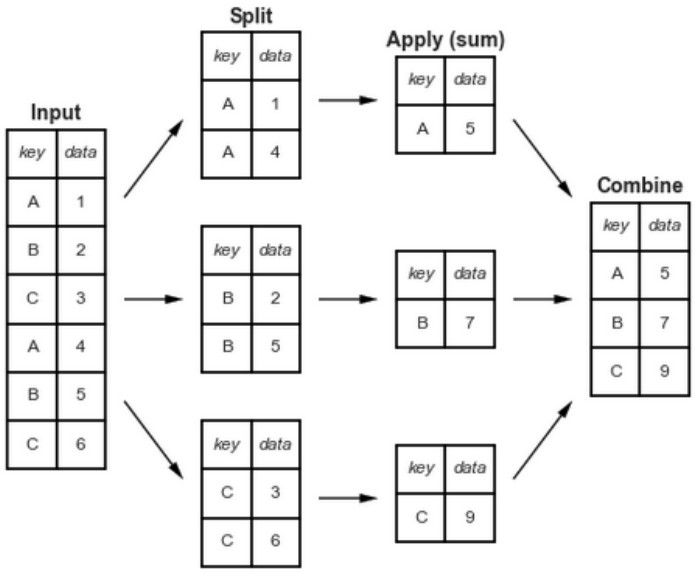


Рис. 9: GroupBy

Важно уточнить, что сам по себе groupby не вернёт ничего, а если быть точнее, то вернётся...

```
<pandas.core.groupby.DataFrameGroupBy object at 0x117272160>
```

...DataFrameGroupBy object. Можно думать, что это некоторое подвешенное состояние процедуры, ожидающее применения какой-то аггрегирующей функции. Если мы решим проиндексировать этот объект, то получим SeriesGroupBy object, суть которого такая же, но разве что возвратом будет Series, а не DataFrame.

```
df = pd.DataFrame({'key': ['A', 'B', 'C', 'A', 'B', 'C'],
                   'data': range(6)}, columns=['key', 'data'])
df.groupby('key').sum()
```

	data
key	
A	3
B	5
C	7

Рис. 10: Результат функции

Также на groupby можно проитерироваться, выглядит это как-то так:

```
for (method, group) in planets.groupby('method'):
    print("{0:30s} shape={1}".format(method, group.shape))
```

```
Astrometry                     shape=(2, 6)
Eclipse Timing Variations      shape=(9, 6)
Imaging                        shape=(38, 6)
Microlensing                   shape=(23, 6)
Orbital Brightness Modulation  shape=(3, 6)
Pulsar Timing                  shape=(5, 6)
Pulsation Timing Variations    shape=(1, 6)
Radial Velocity                shape=(553, 6)
Transit                        shape=(397, 6)
Transit Timing Variations      shape=(4, 6)
```

Рис. 11: Итерация на groupby

Аггрегирование через filter, apply, transform

Apply метод позволяет применить функцию к столбцу (или создать новый столбец на основе уже существующего). Функция может быть как и предопределённая (min, max...), а может быть «кастомной», которую вы предварительно определили:

```
def plus_five(x):
    return x + 5

df['new_col'] = df['exist_col'].apply(plus_five)
```

Это означает, что функция применяется к каждому элементу строки/столбца (в зависимости от аргумента axis), результат функции зависит от определяемой функции (или с помощью ключевого слова result_type).

Filter

filter позволяет «дропнуть» дату опираясь на заданные признаки. Например, давайте профильтруем данные так, чтобы остались строки, в которых есть стандартное отклонение превышающее 4:

df

	key	data1	data2
0	A	0	5
1	B	1	0
2	C	2	3
3	A	3	3
4	B	4	7
5	C	5	9

```
df.groupby('key').std()
```

key	data1	data2
A	2.12132	1.414214
B	2.12132	4.949747
C	2.12132	4.242641

```
df.groupby('key').filter(filter_func)
```

	key	data1	data2
1	B	1	0
2	C	2	3
4	B	4	7
5	C	5	9

Рис. 12: Filter

где filter_func — следующая функция:

```
def filter_func(x):
    return x['data2'].std() > 4
```

Метод filter возвращает булевый массив и применяет маскирование к исходному датасету, что позволяет вывести только подходящие элементы.

Transform

До этих пор мы возвращали некоторую уменьшенную, «отредактированную» версию данных, однако метод transform позволяет модернизировать исходные. Давайте рассмотрим трансформацию, при которой мы из каждого элемента вычитаем среднее по всему датасету:

```
df.groupby('key').transform(lambda x: x - x.mean())
```

	data1	data2
0	-1.5	1.0
1	-1.5	-3.5
2	-1.5	-3.0
3	1.5	-1.0
4	1.5	3.5
5	1.5	3.0

Рис. 13: Transform

Примитивная визуализация

Инструменты Pandas позволяют применять визуализацию к датасету / продвинутому датасету. Понятное дело, что это лучше делать с помощью специализированных библиотек, таких как Matplotlib.pyplot или Seaborn (или вообще Plotly), но в случае если вам нужен какой-то график «на скорую руку», можно воспользоваться и Pandas.

Как я уже и говорил, для визуализации существуют более специализированные инструменты, поэтому тут остановимся прям совсем ненадолго. Самое примитивное, что можно сделать — метод .plot:

```
np.random.seed(123)

ts = pd.Series(np.random.randn(1000), index = pd.date_range("1/1/2000", periods = 1000))

ts.plot()
```

Если же мы применим .plot функцию для DataFrame, то Pandas автоматически раскрасит разные значения в разные цвета. (По факту, .plot функция для Series и DataFrame является некой «обёрткой» для аналогичной функции из Matplotlib). Следовательно, существует масса различных графиков, которые можно явно задавать с помощью метода **kind**. На слайде приведены основные, из названия +- должно быть понятно, какой из них что делает:

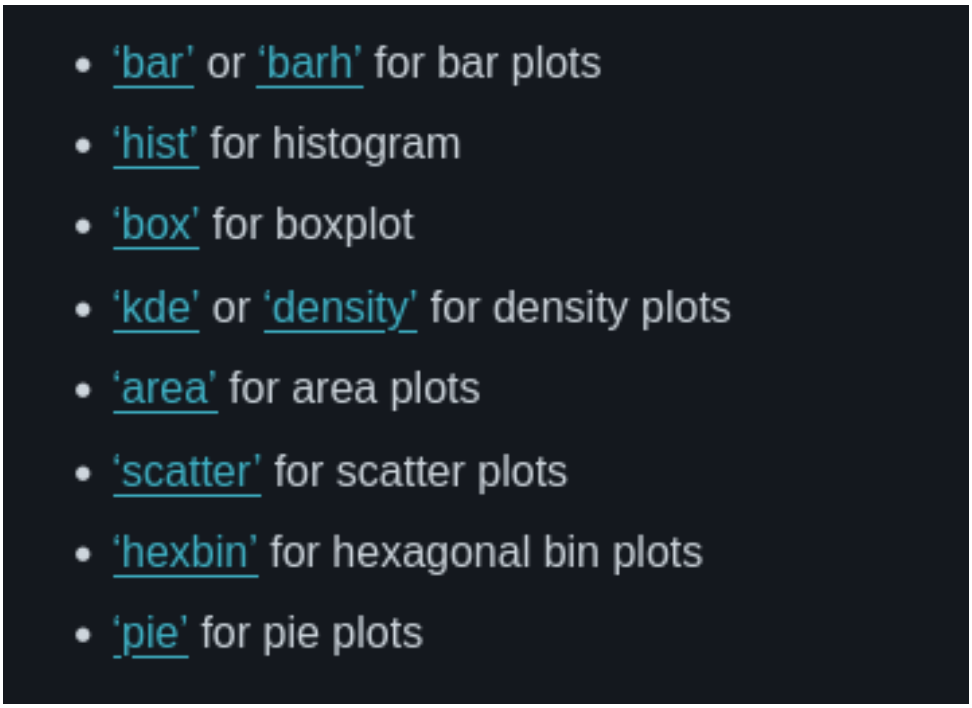


Рис. 14: plot params

Заключение

В принципе на этом всё, что я хотел рассказать про Pandas. Мы успели затронуть что такое в принципе Pandas, как он ускоряет операции и как можно ещё это ускорить, какие основные размышления в нём применются и какую-никакую примитивную визуализацию.