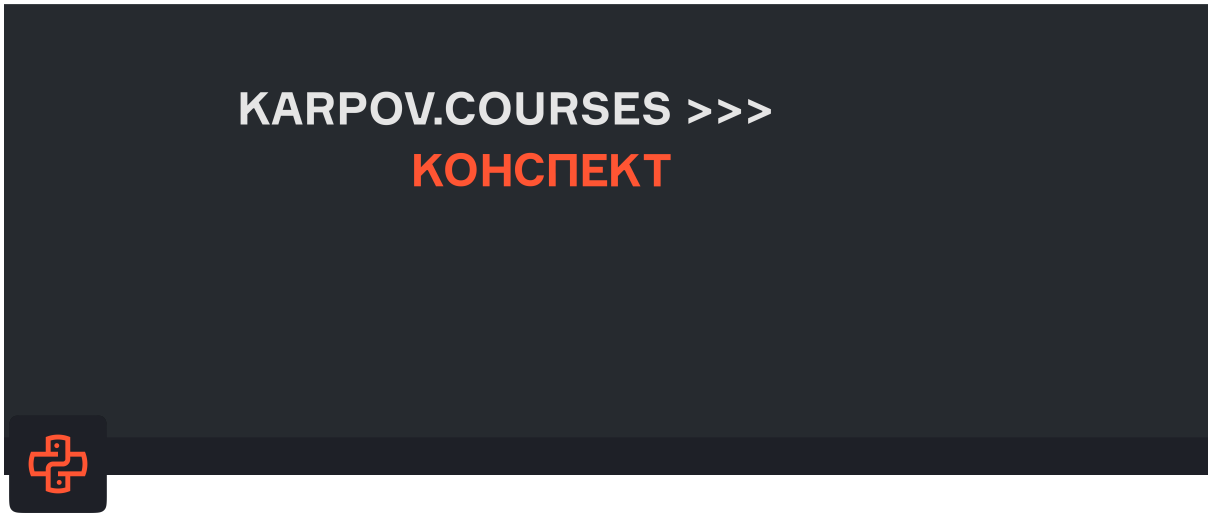


> Конспект > Продвинутые темы > PYTHON

> Ускоряем и оптимизируем панд

Как вы уже могли догадаться по названию, первый урок модуля посвящен оптимизации работы в Pandas!В следующих степах мы посмотрим, какими способами можно итерироваться по датафреймам и применять операции к каждому элементу, а также определим, какие варианты являются наиболее эффективными.

Начнем с самого простого. Логично предположить, что один из самых очевидных вариантов — использовать цикл и проитерироваться по всему датафрейму, применив нужную операцию к каждой строчке.

Первый метод — pd.iterrows(), на каждой итерации возвращает строку датафрейма в виде пар (index, series), где первый элемент является индексом, а содержимое самого ряда представляется в виде Series. Таким образом, можно использовать его в цикле, например:

```
for index, row in df[:1].iterrows():
    print(f"Тип индекса: {type(index)},\nТип содержимого строки: {type(row)}")
```

```
Тип индекса: < class 'int' >,
Тип содержимого строки: < class 'pandas.core.series.Series' >
```

Посмотрим, в каком формате возвращается содержимое строки:

```
for index, row in df[:2].iterrows():
    print("Индекс: {},\nСодержимое строки:\n{}\n".format(index, row))
```

```
Индекс: 0,
Содержимое строки:
key                  2009-06-15 17:26:21.0000001
fare_amount                                  4.5
pickup_datetime        2009-06-15 17:26:21+00:00
pickup_longitude                         -73.8443
pickup_latitude                           40.7213
dropoff_longitude                        -73.8416
dropoff_latitude                          40.7123
passenger_count                                 1
Name: 0, dtype: object

Индекс: 1,
Содержимое строки:
key                  2010-01-05 16:52:16.0000002
fare_amount                                 16.9
pickup_datetime        2010-01-05 16:52:16+00:00
pickup_longitude                          -74.016
pickup_latitude                           40.7113
dropoff_longitude                        -73.9793
dropoff_latitude                           40.782
passenger_count                                 1
Name: 1, dtype: object
```

К конкретным колонкам тоже можно обратиться. Так, выведем индекс и число пассажиров для первых трех наблюдений:

```
for index, row in df[:3].iterrows():
    print("Индекс {}; Число пассажиров: {}".format(index, row['passenger_count']))
```

```
Индекс 0; Число пассажиров: 1
Индекс 1; Число пассажиров: 1
Индекс 2; Число пассажиров: 2
```

Этот метод является одним из наименее эффективных, подробнее о причинах можно почитать вот здесь. Тем не менее, работает быстрее, чем самые обычные циклы.

> Shape of data, или работа с таблицами

В предыдущих уроках вы уже познакомились с базовыми методами работы с таблицами (например, `pivot`) и встретились с иерархическими индексами (мультииндексами). Поначалу они могут выглядеть как что-то страшное и непонятное, от чего хочется поскорее избавиться. Но всё не так плохо! В pandas есть ряд полезных методов, которые упрощают работу с подобными индексами, а также позволяют с легкостью приводить данные к нужному формату.

Начнем с методов `stack` и `unstack`, которые очень похожи на `pivot()`, и предназначены для работы с `MultiIndex`.

stack

`Stack` — помещает уровень столбцов в уровни индекса строк. Результирующий объект – `Series`.

Stack

df2

		A	B
first	second		
bar	one	1	2
	two	3	4
baz	one	5	6
	two	7	8

MultiIndex

stacked = df2.stack()

first	second		
bar	one	A	1
		B	2
	two	A	3
		B	4
baz	one	A	5
		B	6
	two	A	7
		B	8

MultiIndex

Например, создадим датафрейм из трех колонок:

```
np.random.seed(17)
df = pd.DataFrame({'col_1':['item_1', 'item_2', 'item_3', 'item_4', 'item_5'],
                   'col_2': np.random.randint(0, 10, 5),
                   'col_3': np.random.randint(0, 10, 5)})
df
```

```
     col_1  col_2  col_3
0   item_1      1      6
1   item_2      6      4
2   item_3      6      7
3   item_4      9      4
4   item_5      0      7
```

Применяем `.stack()`:

```
df_stacked = df.stack()
df_stacked
```

```
0  col_1    item_1
   col_2         1
   col_3         6
1  col_1    item_2
   col_2         6
   col_3         4
2  col_1    item_3
   col_2         6
   col_3         7
3  col_1    item_4
   col_2         9
```

```
   col_3         4
4  col_1    item_5
   col_2         0
   col_3         7
dtype: object
```

Индексам также можно присвоить названия:

```
df_stacked.index.names = ['id', 'column']
df_stacked
```

```
id  column
0   col_id    item_1
    col_2          7
    col_3          0
1   col_id    item_2
    col_2          4
    col_3          9
2   col_id    item_3
    col_2          4
    col_3          7
3   col_id    item_4
    col_2          5
    col_3          5
4   col_id    item_5
    col_2          8
    col_3          3
dtype: object
```

В качестве **аргументов** `stack` можно передать два параметра.

- `level` – отвечает за уровень, по которому будет проведена стыковка
- `dropna` – нужно ли убрать ряды с пропущенными значениями

Посмотрим как это работает на примере следующего датафрейма о весе и росте котиков:

```
                weight          height
        old_kg  new_kg  old_cm  new_cm
Persik     NaN     3.4     NaN    26.0
Barsik     3.0     4.1    25.0    30.0
```

Сначала применяем метод `stack` без указания аргументов. По умолчанию стыковка происходит по уровню -1, а ряды с пропущенными значениями удаляются. Теперь в датафрейме есть две колонки – `height` и `weight` и два уровня индексов.

```
df.stack()
```

```
                height  weight
Persik  new_cm    26.0     NaN
        new_kg     NaN     3.4
Barsik  new_cm    30.0     NaN
        new_kg     NaN     4.1
```

```
        old_cm    25.0     NaN
        old_kg     NaN     3.0
```

Как бы выглядела табличка с пропущенными значениями? Можно заметить, что в случае `dropna=False`, для Персика появились еще две строки со старыми параметрами роста и веса (`old_cm` и `old_kg`), которые полностью заполнены NaN.

```
df.stack(dropna=False)
```

```
               height  weight
Persik  new_cm    26.0     NaN
        new_kg     NaN     3.4
        old_cm     NaN     NaN
        old_kg     NaN     NaN
Barsik  new_cm    30.0     NaN
        new_kg     NaN     4.1
        old_cm    25.0     NaN
        old_kg     NaN     3.0
```

Теперь проведем стыковку по нулевому уровню. В качестве столбцов выступают `new_cm`, `new_kg`, `old_cm` и `old_kg`, а в индексах остались имена животных и рост/вес.

```
               new_cm  new_kg  old_cm  old_kg
Persik  height    26.0     NaN     NaN     NaN
        weight     NaN     3.4     NaN     NaN
Barsik  height    30.0     NaN    25.0     NaN
        weight     NaN     4.1     NaN     3.0
```

xs

Значения конкретного уровня индексов можно получить используя метод `.xs()`, передав ему интересующее нас значение индекса и уровень. Например, чтобы вывести все значения для `height`, в аргументах нужно указать само значение и название столбца с индексами интересующего нас уровня, в данном случае — `param`.

Применяем `stack` и присваиваем уровням индекса названия:

```
df_stacked_2 = df.stack([0,1])
df_stacked_2.index.names = ['name', 'param', 'param_type']  # присваиваем индексам названия
df_stacked_2
```

```
name    param    param_type
Persik  height   new_cm      26.0
        weight   new_kg       3.4
Barsik  height   new_cm      30.0
                 old_cm      25.0
        weight   new_kg       4.1
                 old_kg       3.0
dtype: float64
```

Достаем все значения роста `height` из уровня индексов `param` с помощью `xs`:

```
df_stacked_2.xs('height', level='param')
```

```
name    param_type
Persik  new_cm        26.0
Barsik  new_cm        30.0
        old_cm        25.0
dtype: float64
```

Попробуем достать значения из колонок исходного датафрейма. Для этого необходимо указать `axis=1`, далее – уровень и ключ, т.е. название интересующего нас уровня. Например, возьмем старый вес Персика и Барсика:

```
df
```

```
                weight          height    # level 0
        old_kg  new_kg  old_cm  new_cm    # level 1
Persik     NaN     3.4     NaN    26.0
Barsik     3.0     4.1    25.0    30.0
```

```
df.xs(axis=1, level=1, key='old_kg')
```

```
        weight
Persik     NaN
Barsik     3.0
```

Старый и новый:

```
df.xs(axis=1, key='weight')  # в данном случае можно не указывать level, т.к. по умолчанию level=0
```

```
        old_kg  new_kg
Persik     NaN     3.4
Barsik     3.0     4.1
```

Note: Чтобы получить значения из исходного датафрейма, можно также передать кортеж из уровней:

```
df[('weight', 'old_kg')]
```

```
Persik    NaN
Barsik    3.0
Name: (weight, old_kg), dtype: float64
```

Документация:

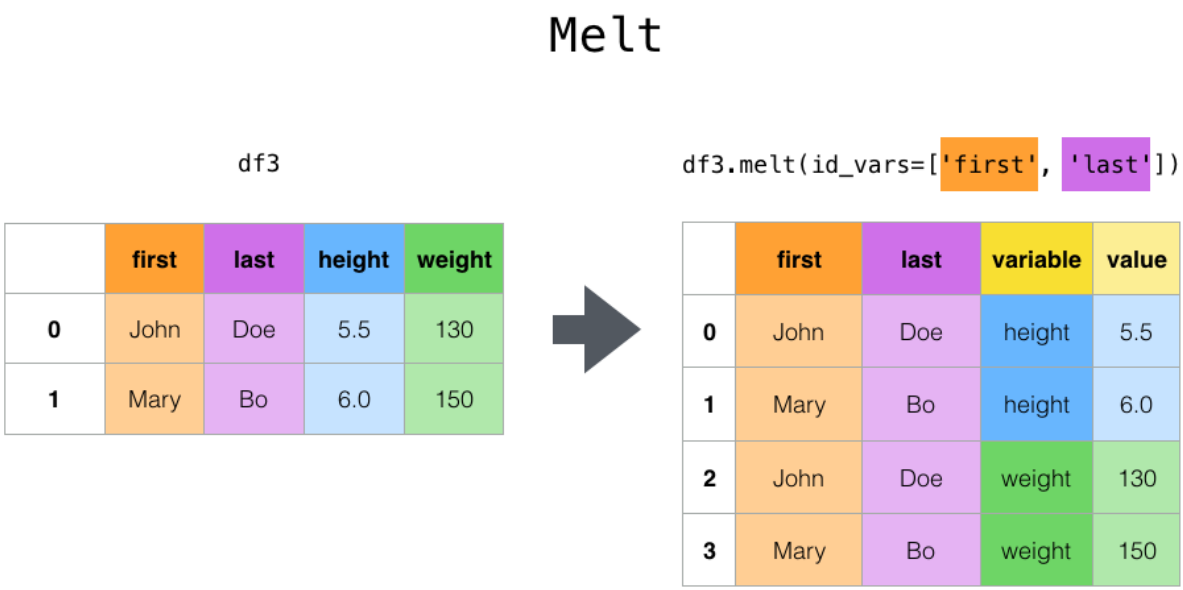
- stack
- xs

Подробнее в видео тут

Широкий и длинный формат

melt

С помощью метода `melt` можно "расплавить" данные и привести их к длинному формату. Так, одна или несколько колонок помещаются в качестве идентификационных переменных, а остальные столбцы считаются измеряемыми переменными. Их названия и значения помещаются в колонки `variable` и `value`.



	first	last	height	weight
0	John	Doe	5.5	130
1	Mary	Bo	6.0	150

	first	last	variable	value
0	John	Doe	height	5.5
1	Mary	Bo	height	6.0
2	John	Doe	weight	130
3	Mary	Bo	weight	150

Для изменения названий полученных столбцов используются параметры `var_name` и `value_name`.

Посмотрим на примере маленького датасета с характеристиками:

```
df3 = pd.DataFrame({'name': ['Persik', 'Brownie'], 'type': ['cat', 'dog'],
                    'color': ['ginger', 'white'], 'height': [17, 30],
                    'weight': [3.4, 4.3]})
df3
```

```
      name type   color  height  weight
0   Persik  cat  ginger      17     3.4
1  Brownie  dog   white      30     4.3
```

Расплавляем! Если не указать колонки, которые нужно использовать в качестве идентификаторов, то названия всех столбцов помещаются в `variable`, а соответствующие в `value`.

```
df3.melt().head()
```

```
  variable    value
0     name   Persik
1     name  Brownie
2     type      cat
3     type      dog
4    color   ginger
```

Используем имена в качестве идентификатора:

```
df3.melt(id_vars='name').head()
```

```
      name variable   value
0   Persik     type     cat
1  Brownie     type     dog
2   Persik    color  ginger
3  Brownie    color   white
4   Persik   height      17
```

Для изменения названий полученных столбцов используются параметры `var_name` и `value_name`. Например, передаем в качестве `id_vars` имена, для значений (`value_vars`) используем только три колонки и изменяем названия новых колонок:

```
df3.melt(id_vars=['name'], value_vars=['type', 'color', 'height'],
         var_name='characteristics', value_name='value')
```

```
      name characteristics   value
0   Persik            type     cat
1  Brownie            type     dog
2   Persik           color  ginger
3  Brownie           color   white
4   Persik          height      17
5  Brownie          height      30
```

wide_to_long

Еще один вариант для перевода данных из широкого формата в длинный — `pd.wide_to_long()`.

Предположим, мы собрали побольше данных о котике Персике и пёсике Брауни, и добавили данные о весе и росте уже за два года:

```
      name type  AvgHeight_2019  AvgHeight_2020  AvgWeight_2019  AvgWeight_2020   color
0   Persik  cat       17.077963       17.134233             3.4          3.5545  ginger
1  Brownie  dog       30.673324       30.674466             4.3          4.5716   white
```

Посмотрим на аргументы функции более подробно.

- `data` — датафрейм
- `stubnames` — части названий переменных, которые мы хотим преобразовать из широкого формата в длинный
- `i` — переменные, которые не трансформируются, и в результате помещаются в индексы
- `j` — имя новой переменной
- `sep` — разделитель (между параметром и значением)

В данном случае у нас есть две общих характеристики, отвечающих за рост и вес в конкретный год. Названия соответствующих переменных состоят из `AvgHeight` / `AvgWeight` и года, поэтому в `stubnames` мы передаем список параметров (вес и рост), а оставшаяся часть названия (2018, 2019) будет использована в качестве значений новой переменной `year`. Столбцы `type` и `name` помещаем в индексы, а параметр `color` оставляем обычной колонкой.

```
lng = pd.wide_to_long(df4, ['AvgHeight', 'AvgWeight'], i=['type', 'name'], j='year', sep='_')
lng
```

```
                   color  AvgHeight  AvgWeight
type name    year
cat  Persik  2019  ginger  17.077963     3.4000
             2020  ginger  17.134233     3.5545
dog  Brownie 2019   white  30.673324     4.3000
             2020   white  30.674466     4.5716
```

А теперь возвращаем всё обратно к **широкому** формату:

```
wd = lng.unstack(level='year')
wd.columns = ['_'.join(map(str, col)) for col in wd.columns]  # соединяем названия
wd.drop('color_2019', inplace=True, axis=1)  # убираем лишнюю колонку
wd = wd.rename(columns={'color_2020': 'color'})  # исправляем название
wd.reset_index() # избавляемся от мультииндекса
```

```
  type     name   color  AvgHeight_2019  AvgHeight_2020  AvgWeight_2019  AvgWeight_2020
0  cat   Persik  ginger       17.077963       17.134233             3.4          3.5545
1  dog  Brownie   white       30.673324       30.674466             4.3          4.5716
```

Подробнее в видео тут

explode

Одно из нововведений в pandas версии 0.25.0 — метод `explode()`. Сначала создадим датафрейм из двух столбцов: колонку `B` заполним единичками, а в `A` запишем следующие элементы:

- в две ячейки – списки, состоящие из нескольких элементов
- пустой список
- 'kitten'

```
df = pd.DataFrame({'A': [[1, 2, 3],
                         'kitten',
                         [],
                         ['kitten', 'puppy']],
                   'B': 1})
df
```

```
               A   B
0      [1, 2, 3]   1
1         kitten   1
2             []   1
3 [kitten, puppy]  1
```

Такой формат данных в ячейках не очень удобен для дальнейшей работы. Например, как нам посчитать, сколько раз встретилось то или иное значение в `A`?

Как раз здесь нам поможет `explode`. Метод преобразовывает каждый элемент списка в отдельный ряд, при этом сами индексы строк дублируются. На вход необходимо передать либо одну колонку, либо их список.

```
df.explode('A')
```

```
       A   B
0      1   1
0      2   1
0      3   1
1 kitten   1
2    NaN   1
3 kitten   1
3  puppy   1
```

Посчитаем, сколько раз встречаются те или иные значения:

```
df.explode('A').A.value_counts().to_frame(name='count') # переименовываем "A" в "count"
```

```
       count
kitten     2
puppy      13          12          11          1
```

Подробнее в видео тут

> Управляем временем

resample

Теперь посмотрим, какие возможности pandas предоставляет для работы с временными рядами! Один из наиболее часто используемых и удобных методов — `.resample()`, позволяющий преобразовать данные и применить к ним другой метод (`sum()`, `size()` и пр.). Таким образом, можно рассчитать показатели, например, за весь день, неделю, месяц и т.п. С полным списком возможных значений можно ознакомиться здесь.

Date Offset	Обозначение	Описание
DateOffset	None	
Week	'W'	одна неделя
MonthEnd	'M'	конец календарного месяца
MonthBegin	'MS'	начало календарного месяца
QuarterEnd	'Q'	конец календарного квартала
YearEnd	'A'	конец календарного года
YearBegin	'AS' or 'BYS'	начало календарного года
Day	'D'	день
Hour	'H'	один час
Minute	'T' or 'min'	одна минута
Second	'S'	одна секунда
Milli	'L' or 'ms'	одна миллисекунда
Micro	'U' or 'us'	одна микросекунда

Например, посчитать сумму показателя по дням, имея данные по часам, можно следующим образом:

```
data.resample(rule='D').sum()
```

где `rule` — параметр, отвечающий за то, по какому периоду нужно агрегировать данные. В данном случае параметр он равен `'D'` (Day).

Для получения данных за каждые 6 часов используем число и обозначение `H`:

```
data.resample(rule='6H').sum()
```

В качестве **индексов** датафрейма **обязательно** нужно использовать колонку формата `DateTime`, отсортированную в правильном порядке. Поэтому предварительно всегда следует проверить правильность типа данных и, если требуется, привести его к правильному с помощью `pd.to_datetime()`.

- документация `.resample()`
- документация `pd.to_datetime()`
- конспект

> Стильный урок

стиль

Помимо методов для работы с данными, `pandas` включает в себя возможности для форматирования таблиц!

Например:

- `df.style.highlight_null()` – подсветить ячейки с пропущенными значениями
- `df.style.highlight_max()` – подсветить ячейки с максимальными значениями по колонкам
- `df.style.highlight_min()` – подсветить ячейки с минимальными значениями по колонкам
- `df.style.applymap(func)` – применить стилевую функцию к каждой ячейке датафрэйма
- `df.style.apply(func, axis, subset)` – применить стилевую функцию к каждой колонке/строке в зависимости от `axis`, `subset` позволяет выбрать часть колонок для оформления
- `render()` – после декорирования возвращает HTML, описывающий табличку

Можно использовать несколько методов одновременно, применяя их друг за другом (method chaining). Давайте посмотрим на `style` подробнее в следующих стэпах, а затем разберемся с тем, как можно отформатировать табличку с retention :)

содержание

1. индексы и подписи
2. как раскрасить ячейки в зависимости от значений
3. форматируем числа в ячейках
4. что такое retention и как его визуализировать с помощью style

Документация

индексы и подписи

Сначала создадим небольшой датафрейм:

```
np.random.seed(77)
df = pd.DataFrame({'A': list(range(5)),
                   'B': np.random.randint(0, 10, 5),
                   'C': np.random.randint(-10, 10, 5),
                   'D': np.random.randint(-10, 100, 5)})
df
```

	A	B	C	D
0	0	7	-10	49
1	1	4	-3	90
2	2	4	2	44
3	3	5	9	26
4	4	8	-10	37

Первый метод – `.hide_index()`, позволяет спрятать индексы:

```
df.style.hide_index()
```

A	B	C	D
0	7	-10	49
1	4	-3	90
2	4	2	44
3	5	9	26
4	8	-10	37

Далее – `.set_caption()`. С его помощью можно добавить подпись к таблице:

```
df.style.hide_index().set_caption('Cool table')
```

Cool table

A	B	C	D
0	7	-10	49
1	4	-3	90
2	4	2	44
3	5	9	26
4	8	-10	37

раскрашиваем ячейки

highlight_min/max

`highlight_max` – подсвечивает (выделяет) цветом **наибольшее** значение. Можно применить либо к каждой строке (`axis=0/'index'`), либо к каждой колонке

(`axis=1/'columns'`).

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

	A	B	C	D
0	0	7	-10	49
1	1	4	-3	90
2	2	4	2	44
3	3	5	9	26
4	4	8	-10	37

```
df.style.highlight_max(axis='index')
```

	A	B	C	D
0	0	7	-10	49
1	1	4	-3	90
2	2	4	2	44
3	3	5	9	26
4	4	8	-10	37

Аналогичная функция для подсветки **минимальных** значений – `highlight_min()`.

```
df.style.highlight_min()
```

	A	B	C	D
0	0	7	-10	49
1	1	4	-3	90
2	2	4	2	44
3	3	5	9	26
4	4	8	-10	37

background_gradient

`background_gradient` – раскрашивает ячейки в зависимости от их значений. В итоге получается что-то похожее на heatmap (тепловую карту). Например:

```
(df.style
 .highlight_min('A', color='red')
 .highlight_max('B', color='orange')
 .background_gradient(subset=['C','D'],cmap='viridis')
)
```

	A	B	C	D
0	0	7	-10	49
1	1	4	-3	90
2	2	4	2	44
3	3	5	9	26
4	4	8	-10	37

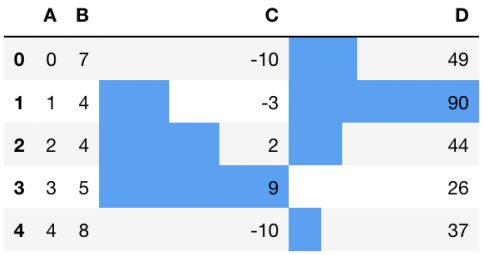
Здесь мы сначала выделяем красным минимальное значение в столбце `A` (`highlight_min`), затем – оранжевым максимальное в колонке `B` (`highlight_max`), и применяем `background_gradient` для `C` и `D`, указав палитру viridis.

style.bar

Визуализировать значения можно прямо в таблице с помощью `.bar()`. Данный метод принимает несколько аргументов:

- `subset` – для каких колонок нужно построить небольшой барплот
- `color` – цвет

```
df.style.bar(subset=['C', 'D'], color='#67A5EB')
```

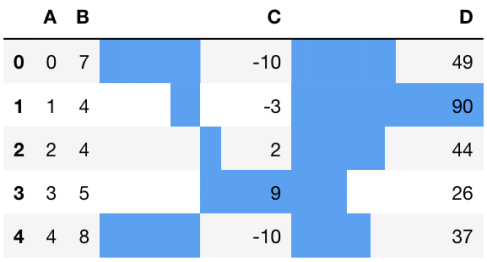


	A	B	C	D
0	0	7	-10	49
1	1	4	-3	90
2	2	4	2	44
3	3	5	9	26
4	4	8	-10	37

- `align` – как выровнять столбики (`mid` – центр ячейки в (max-min)/2; `zero` – ноль находится в центре ячейки; `left` – минимальное значение находится в левой части

ячейки)

```
df.style.bar(subset=['C', 'D'], color='#67A5EB', align='mid')
```



	A	B	C	D
0	0	7	-10	49
1	1	4	-3	90
2	2	4	2	44
3	3	5	9	26
4	4	8	-10	37

Также можно указать сразу несколько цветов. Значения меньше 0 будут окрашены в красный, больше – в зелёный.

```
(df
 .style
 .hide_index()
 .bar(subset=['C'], align='mid',color=['#d65f5f', '#5fba7d'])
)
```

A	B	C	D
0	7	-10	49
1	4	-3	90
2	4	2	44
3	5	9	26
4	8	-10	37

форматирование отображения чисел

Иногда может понадобится различное число знаков после запятой. Для этого подходит метод `.format()`, которому нужно передать строку, указывающую сколько знаков необходимо оставить.

```
# генерируем данные
df = pd.DataFrame({'A': np.linspace(1, 10, 5)})
df = pd.concat([df, pd.DataFrame(np.random.randn(5, 4), columns=list('BCDE'))],axis=1)
df['F'] = np.random.choice(['A', 'B'], size=5)
df.iloc[3, 3] = np.nan
df.iloc[0, 2] = np.nan
df
```

	A	B	C	D	E	F
0	1.00	0.797939	NaN	-1.652119	0.717119	B
1	3.25	0.977228	-1.040849	-0.643520	-0.112520	A
2	5.50	-0.314166	1.627440	-0.361227	-0.173046	B
3	7.75	-1.951309	-0.978210	NaN	-1.178379	A
4	10.00	-0.515551	-0.063015	-0.559371	0.796697	A

Форматируем:

- оставляем только 2 знака после точки
- добавляем знак + для положительных значений
- применяем ко всем колонкам, кроме `F`

```
df.style.format("{:+.2f}", subset=df.columns.drop('F'))
```

	A	B	C	D	E	F
0	+1.00	+0.80	+nan	-1.65	+0.72	B
1	+3.25	+0.98	-1.04	-0.64	-0.11	A
2	+5.50	-0.31	+1.63	-0.36	-0.17	B
3	+7.75	-1.95	-0.98	+nan	-1.18	A
4	+10.00	-0.52	-0.06	-0.56	+0.80	A

Также можем скрыть индексы и добавить название:

```
(df.style
 .format({'B': "{:0<4.0f}", 'D': '{:+.2f}'})
 .hide_index()
 .set_caption('Новая таблица'))
```

Новая таблица

A	B	C	D	E	F
1.000000	1000	nan	-1.65	0.717119	B
3.250000	1000	-1.040849	-0.64	-0.112520	A
5.500000	-000	1.627440	-0.36	-0.173046	B
7.750000	-200	-0.978210	+nan	-1.178379	A
10.000000	-100	-0.063015	-0.56	0.796697	A

И при желании импортировать в Excel (но не всё форматирование переносится):

```
(df.style
 .bar(align='mid', color=['#d65f5f', '#5fba7d'])
 .to_excel('styled.xlsx', engine='openpyxl')
)
```

retention

Retention – показатель удержания пользователей. Иными словами – отражает то, сколько пользователей возвращаются в продукт спустя заданное время.

Обычно день начала использования сервиса называется Day 0 – момент, когда юзер впервые воспользовался продуктом. **N-Day Retention** показывает, сколько процентов пользователей, начавших пользоваться продуктом в день 0, вернулись и продолжили использовать продукт N дней спустя.

Под днем не всегда понимается день – интервалы измерения ретеншена зависят от характеристик самого продукта. Так, некоторые сервисы подразумевают ежедневное использование (напр. социальные сети), а другие – более редкое (бронирование, такси, доставка, рестораны). Согласитесь, вряд ли пользователи бронируют авиабилеты или отели каждый день :) В случае сервиса доставки или ресторана, мы могли бы посмотреть на недельные интервалы использования (week by week). Тогда retention бы показывал, сколько пользователей вернулись в 1-7 день, 8-14 и т.д.

визуализация

Один из вариантов визуализации представлен ниже. Что же здесь происходит?

- `Cohort`, строки – когорта пользователей. Например, 2011-01 означает, что пользователи из этой группы первый раз сделали заказ в онлайн магазине в январе 2011 года, 2011-02 – в феврале, и т.д.
- `CohortPeriod`, столбцы – месяц. 0 – когда пользователи только-только сделали первую покупку. Далее – сколько из них оформили заказ в 1 месяце, 2, ..., 12-м. Часть значений остается пропущенной, поскольку период наблюдений для части пользователей меньше, чем для остальных. Для юзеров, присоединившихся в декабре 2010, имеются данные за весь год, в то время как для ребят из когорты 2011-11 – всего лишь за 0 и 1 месяц.

Загрузить исходную табличку можно отсюда следующим образом: `pd.read_csv("https://stepik.org/media/attachments/lesson/367416/user_retention.csv", index_col=0)`

User retention by cohort

CohortPeriod	0	1	2	3	4	5	6	7	8	9	10	11	12
Cohort													
2010-12	100.00%	38.19%	33.44%	38.71%	35.97%	39.66%	37.97%	35.44%	35.44%	39.45%	37.34%	50.00%	27.43%
2011-01	100.00%	23.99%	28.27%	24.23%	32.78%	29.93%	26.13%	25.65%	31.12%	34.68%	36.82%	14.96%	
2011-02	100.00%	24.74%	19.21%	27.89%	26.84%	24.74%	25.53%	28.16%	25.79%	31.32%	9.21%		
2011-03	100.00%	19.09%	25.45%	21.82%	23.18%	17.73%	26.36%	23.86%	28.86%	8.86%			
2011-04	100.00%	22.74%	22.07%	21.07%	20.74%	23.75%	23.08%	26.09%	8.36%				
2011-05	100.00%	23.66%	17.20%	17.20%	21.51%	24.37%	26.52%	10.39%					
2011-06	100.00%	20.85%	18.72%	27.23%	24.68%	33.62%	10.21%						
2011-07	100.00%	20.94%	20.42%	23.04%	27.23%	11.52%							
2011-08	100.00%	25.15%	25.15%	25.15%	13.77%								
2011-09	100.00%	29.87%	32.55%	12.08%									
2011-10	100.00%	26.42%	13.07%										
2011-11	100.00%	13.44%											
2011-12	100.00%												

Довольно сложно воспринимать подобную информацию без цвета, поэтому применяем рассмотренные ранее методы:

```
ur_style = (user_retention
            .style
            .set_caption('User retention by cohort')  # добавляем подпись
            .background_gradient(cmap='viridis')  # раскрашиваем ячейки по столбцам
            .highlight_null('white')  # делаем белый фон для значений NaN
            .format("{:.2%}", na_rep=""))  # числа форматируем как проценты, NaN заменяем на пустоту
ur_style
```

User retention by cohort

CohortPeriod	0	1	2	3	4	5	6	7	8	9	10	11	12
Cohort													
2010-12	100.00%	38.19%	33.44%	38.71%	35.97%	39.66%	37.97%	35.44%	35.44%	39.45%	37.34%	50.00%	27.43%
2011-01	100.00%	23.99%	28.27%	24.23%	32.78%	29.93%	26.13%	25.65%	31.12%	34.68%	36.82%	14.96%	
2011-02	100.00%	24.74%	19.21%	27.89%	26.84%	24.74%	25.53%	28.16%	25.79%	31.32%	9.21%		
2011-03	100.00%	19.09%	25.45%	21.82%	23.18%	17.73%	26.36%	23.86%	28.86%	8.86%			
2011-04	100.00%	22.74%	22.07%	21.07%	20.74%	23.75%	23.08%	26.09%	8.36%				
2011-05	100.00%	23.66%	17.20%	17.20%	21.51%	24.37%	26.52%	10.39%					
2011-06	100.00%	20.85%	18.72%	27.23%	24.68%	33.62%	10.21%						
2011-07	100.00%	20.94%	20.42%	23.04%	27.23%	11.52%							
2011-08	100.00%	25.15%	25.15%	25.15%	13.77%								
2011-09	100.00%	29.87%	32.55%	12.08%									
2011-10	100.00%	26.42%	13.07%										
2011-11	100.00%	13.44%											
2011-12	100.00%												

Отлично! Теперь довольно легко заметить, что ретеншен в каждый из месяцев был наибольшим среди пользователей из самой первой когорты, 2010-12. Подумайте, что может влиять на подобный показатель (e.g. какие изменения).