Pandas

Библиотека pandas

Пакет для статистической обработки данных, по функциональности близкий к SQL и R.
Включает в себя функциональность работы с базами данных и таблицами Excel.

In [1]:

```
import numpy as np
import pandas as pd
import scipy.stats as sps

import warnings
warnings.simplefilter("ignore", FutureWarning)
```

started 10:12:09 2020-04-12, finished in 620ms

Тип данных Series

Одномерный набор данных. Отсутствующий данные записываются как `np.nan` (например, в этот день термометр сломался или метеоролог был пьян). При вычислении среднего и других операций соответствующие функции не учитывают отсутствующие значения.

In [2]:

```
1  l = [1, 3, 5, np.nan, 6, 8]
2  s = pd.Series(l)
3  s
```

started 10:12:10 2020-04-12, finished in 11ms

Out[2]:

```
0    1.0
1    3.0
2    5.0
3    NaN
4    6.0
5    8.0
dtype: float64
```

Полезно знать: Для поиска пропусков есть специальный метод `.isna()` . Он эквивалентен конструкции `s != s`

In [3]:

```
1  s.isna()
```

started 10:12:10 2020-04-12, finished in 13ms

Out[3]:

```
0    False
1    False
2    False
3     True
4    False
5    False
dtype: bool
```

Основная информация о наборе данных: количество записей, среднее, стандартное отклонение, минимум, нижний квартиль, медиана, верхний квартиль, максимум, а так же тип данных.

In [4]:

```
1  s.describe()
```

started 10:12:10 2020-04-12, finished in 9ms

Out[4]:

```
count    5.000000
mean     4.600000
std      2.701851
min      1.000000
25%      3.000000
50%      5.000000
75%      6.000000
max      8.000000
dtype: float64
```

В данном примере обычная индексация.

In [5]:

```
1  s[2]
```

started 10:12:10 2020-04-12, finished in 5ms

Out[5]:

```
5.0
```

In [6]:

```
1  s[2] = 7
2  s
```

started 10:12:10 2020-04-12, finished in 5ms

Out[6]:

```
0    1.0
1    3.0
2    7.0
3    NaN
4    6.0
5    8.0
dtype: float64
```

In [7]:

```
1  s[2:5]
```

started 10:12:10 2020-04-12, finished in 5ms

Out[7]:

```
2    7.0
3    NaN
4    6.0
dtype: float64
```

In [8]:

```
1  s1 = s[1:]
2  s1
```

started 10:12:10 2020-04-12, finished in 4ms

Out[8]:

```
1    3.0
2    7.0
3    NaN
4    6.0
5    8.0
dtype: float64
```

In [9]:

```
1  s2 = s[:-1]
2  s2
```

started 10:12:10 2020-04-12, finished in 5ms

Out[9]:

```
0    1.0
1    3.0
2    7.0
3    NaN
4    6.0
dtype: float64
```

В сумме `s1+s2` складываются данные с одинаковыми индексами. Поскольку в `s1` нет данного и индексом 0, а в `s2` - с индексом 5, в `s1+s2` в соответствующих позициях будет `NaN`.

In [10]:

```
1  s1 + s2
```

started 10:12:10 2020-04-12, finished in 6ms

Out[10]:

```
0     NaN
1     6.0
2    14.0
3     NaN
4    12.0
5     NaN
dtype: float64
```

К наборам данных можно применять функции из `numpy`.

In [11]:

```
1  np.exp(s)
```

started 10:12:10 2020-04-12, finished in 3ms

Out[11]:

```
0       2.718282
1      20.085537
2    1096.633158
3            NaN
4     403.428793
5    2980.957987
dtype: float64
```

При создании набора данных `s` мы не указали, что будет играть роль индекса. По умолчанию это последовательность неотрицательных целых чисел 0, 1, 2, ...

In [12]:

```
1  s.index
```

started 10:12:10 2020-04-12, finished in 2ms

Out[12]:

```
RangeIndex(start=0, stop=6, step=1)
```

Но можно создавать наборы данных с индексом, заданным списком.

In [13]:

```
1  i = list('abcdef')
2  i
```

started 10:12:10 2020-04-12, finished in 2ms

Out[13]:

```
['a', 'b', 'c', 'd', 'e', 'f']
```

In [14]:

```
1  s = pd.Series(l, index=i)
2  s
```

started 10:12:10 2020-04-12, finished in 4ms

Out[14]:

```
a    1.0
b    3.0
c    5.0
d    NaN
e    6.0
f    8.0
dtype: float64
```

In [15]:

```
1  s['c']
```

started 10:12:10 2020-04-12, finished in 4ms

Out[15]:

```
5.0
```

Если индекс - строка, то вместо `s['c']` можно писать `s.c` .

In [16]:

```
1  s.c
```

started 10:12:10 2020-04-12, finished in 3ms

Out[16]:

```
5.0
```

Набор данных можно создать из словаря.

In [17]:

```
1  s = pd.Series({'a':1, 'b':2, 'c':0})
2  s
```

started 10:12:10 2020-04-12, finished in 4ms

Out[17]:

```
a    1
b    2
c    0
dtype: int64
```

Можно отсортировать набор данных.

In [18]:

```
1  s.sort_values()
```

started 10:12:10 2020-04-12, finished in 7ms

Out[18]:

```
c    0
a    1
b    2
dtype: int64
```

Роль индекса может играть, скажем, последовательность дат (или времён измерения и т.д.).

In [19]:

```
1  d = pd.date_range('20160101', periods=10)
2  d
```

started 10:12:10 2020-04-12, finished in 9ms

Out[19]:

```
DatetimeIndex(['2016-01-01', '2016-01-02', '2016-01-03', '2016-01-04',
               '2016-01-05', '2016-01-06', '2016-01-07', '2016-01-08',
               '2016-01-09', '2016-01-10'],
              dtype='datetime64[ns]', freq='D')
```

In [20]:

```
1  s = pd.Series(sps.norm.rvs(size=10), index=d)
2  s
```

started 10:12:10 2020-04-12, finished in 7ms

Out[20]:

```
2016-01-01   -0.563028
2016-01-02    2.015834
2016-01-03   -2.008353
2016-01-04    1.203577
2016-01-05    1.216309
2016-01-06    2.288571
2016-01-07   -1.443840
2016-01-08    1.250447
2016-01-09   -0.209155
2016-01-10   -0.551679
Freq: D, dtype: float64
```

Операции сравнения возвращают наборы булевых данных.

In [21]:

```
1  s > 0
```

started 10:12:10 2020-04-12, finished in 7ms

Out[21]:

```
2016-01-01    False
2016-01-02     True
2016-01-03    False
2016-01-04     True
2016-01-05     True
2016-01-06     True
2016-01-07    False
2016-01-08     True
2016-01-09    False
2016-01-10    False
Freq: D, dtype: bool
```

Если такой булев набор использовать для индексации, получится поднабор только из тех данных, для которых условие есть `True` .

In [22]:

```
1  s[s > 0]
```

started 10:12:10 2020-04-12, finished in 5ms

Out[22]:

```
2016-01-02    2.015834
2016-01-04    1.203577
2016-01-05    1.216309
2016-01-06    2.288571
2016-01-08    1.250447
dtype: float64
```

Кумулятивные максимумы - от первого элемента до текущего.

In [23]:

```
1  s.cummax()
```

started 10:12:10 2020-04-12, finished in 5ms

Out[23]:

```
2016-01-01   -0.563028
2016-01-02    2.015834
2016-01-03    2.015834
2016-01-04    2.015834
2016-01-05    2.015834
2016-01-06    2.288571
2016-01-07    2.288571
2016-01-08    2.288571
2016-01-09    2.288571
2016-01-10    2.288571
Freq: D, dtype: float64
```

Кумулятивные суммы.

In [24]:

```
1  s.cumsum()
```

started 10:12:10 2020-04-12, finished in 5ms

Out[24]:

```
2016-01-01   -0.563028
2016-01-02    1.452807
2016-01-03   -0.555546
2016-01-04    0.648032
2016-01-05    1.864341
2016-01-06    4.152912
2016-01-07    2.709072
2016-01-08    3.959519
2016-01-09    3.750364
2016-01-10    3.198685
Freq: D, dtype: float64
```

Произвольные функции на префиксах можно считать с помощью конструкции `expanding` .
Например, так можно посчитать медианы на префиксах. Будет не быстрее, чем вручную, но аккуратнее.

In [25]:

```
1  s.expanding().apply(np.median, raw=True)
```

started 10:12:10 2020-04-12, finished in 7ms

Out[25]:

```
2016-01-01   -0.563028
2016-01-02    0.726403
2016-01-03   -0.563028
2016-01-04    0.320275
2016-01-05    1.203577
2016-01-06    1.209943
2016-01-07    1.203577
2016-01-08    1.209943
2016-01-09    1.203577
2016-01-10    0.497211
Freq: D, dtype: float64
```

Если вы хотите посчитать разности соседних элементов, воспользуйтесь методом `diff` .
Ключевое слово `periods` отвечает за то, с каким шагом будут считаться разности.

In [26]:

```
1  s.diff()
```

started 10:12:10 2020-04-12, finished in 5ms

Out[26]:

```
2016-01-01         NaN
2016-01-02    2.578862
2016-01-03   -4.024187
2016-01-04    3.211930
2016-01-05    0.012732
2016-01-06    1.072262
2016-01-07   -3.732411
2016-01-08    2.694287
2016-01-09   -1.459602
2016-01-10   -0.342524
Freq: D, dtype: float64
```

Результат будет иметь тот же размер, но в начале появятся пропущенные значения.
От них можно избавиться при помощи метода `dropna`

In [27]:

```
1  s.diff().dropna()
```

started 10:12:10 2020-04-12, finished in 6ms

Out[27]:

```
2016-01-02    2.578862
2016-01-03   -4.024187
2016-01-04    3.211930
2016-01-05    0.012732
2016-01-06    1.072262
2016-01-07   -3.732411
2016-01-08    2.694287
2016-01-09   -1.459602
2016-01-10   -0.342524
Freq: D, dtype: float64
```

Упражнение

Посчитайте на префиксах усреднённую сумму квадратов разностей соседних элементов `s`

In [28]:

```
1 ▾ # ВАШ КОД
2   s.diff().dropna().expanding().apply(lambda x: np.mean(x**2))
```

started 10:12:10 2020-04-12, finished in 6ms

Out[28]:

```
2016-01-02     6.650530
2016-01-03    11.422306
2016-01-04    11.053702
2016-01-05     8.290317
2016-01-06     6.862203
2016-01-07     8.040317
2016-01-08     7.928727
2016-01-09     7.203941
2016-01-10     6.416539
Freq: D, dtype: float64
```

Наконец, построим график.

In [29]:

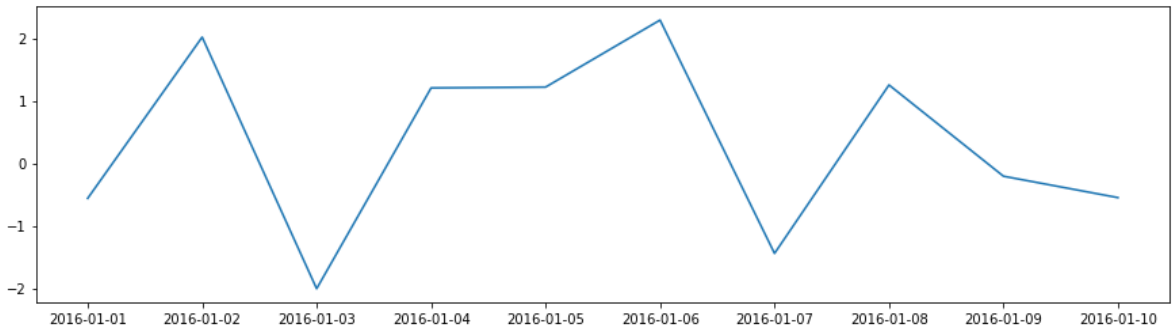
```
1  import matplotlib.pyplot as plt
2  %matplotlib inline
3
4  # Нужно для новых версий библиотек для преобразования дат
5  from pandas.plotting import register_matplotlib_converters
6  register_matplotlib_converters()
```

started 10:12:10 2020-04-12, finished in 101ms

In [30]:

```
plt.figure(figsize=(15, 4))
plt.plot(s)
plt.show()
```

started 10:12:10 2020-04-12, finished in 339ms



Более подробно ознакомиться с методами можно [в официальной документации (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.html)](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.html).

Тип данных DataFrame

Двумерная таблица данных. Имеет индекс и набор столбцов (возможно, имеющих разные типы). Таблицу можно построить, например, из словаря, значениями в котором являются одномерные наборы данных.

In [31]:

```
d = {'one': pd.Series(range(6), index=list('abcdef')),
     'two': pd.Series(range(7), index=list('abcdefg')),
     'three': pd.Series(sps.norm.rvs(size=7), index=list('abcdefg'))}
df = pd.DataFrame(d)
df
```

started 10:12:11 2020-04-12, finished in 14ms

Out[31]:

	one	two	three
a	0.0	0	-1.496178
b	1.0	1	0.562452
c	2.0	2	0.520759
d	3.0	3	-0.461771
e	4.0	4	0.426069
f	5.0	5	-0.534984
g	NaN	6	-0.364495

Таблица с несколькими разными типами данных

In [32]:

```
df2 = pd.DataFrame({ 'A': 1.,
                     'B': pd.Timestamp('20130102'),
                     'C': pd.Series(1,index=list(range(4)),
                                    dtype='float32'),
                     'D': np.array([3] * 4,
                                   dtype='int32'),
                     'E': pd.Categorical(["test", "train",
                                          "test", "train"]),
                     'F': 'foo' })
df2
```

started 10:12:11 2020-04-12, finished in 18ms

Out[32]:

	A	B	C	D	E	F
0	1.0	2013-01-02	1.0	3	test	foo
1	1.0	2013-01-02	1.0	3	train	foo
2	1.0	2013-01-02	1.0	3	test	foo
3	1.0	2013-01-02	1.0	3	train	foo

In [33]:

```
df2.dtypes
```

started 10:12:11 2020-04-12, finished in 13ms

Out[33]:

```
A           float64
B    datetime64[ns]
C           float32
D             int32
E          category
F            object
dtype: object
```

Данные

Вернемся к первой таблице и посмотрим на ее начало и конец

In [34]:

```
df.head()
```

started 10:12:11 2020-04-12, finished in 14ms

Out[34]:

	one	two	three
a	0.0	0	-1.496178
b	1.0	1	0.562452
c	2.0	2	0.520759
d	3.0	3	-0.461771
e	4.0	4	0.426069

In [35]:

```
1  df.tail(3)
```

started 10:12:11 2020-04-12, finished in 9ms

Out[35]:

	one	two	three
e	4.0	4	0.426069
f	5.0	5	-0.534984
g	NaN	6	-0.364495

Индексы

In [36]:

```
1  df.index
```

started 10:12:11 2020-04-12, finished in 9ms

Out[36]:

```
Index(['a', 'b', 'c', 'd', 'e', 'f', 'g'], dtype='object')
```

Названия колонок

In [37]:

```
1  df.columns
```

started 10:12:11 2020-04-12, finished in 5ms

Out[37]:

```
Index(['one', 'two', 'three'], dtype='object')
```

Получение обычной матрицы данных

In [38]:

```
1  df.values
```

started 10:12:11 2020-04-12, finished in 5ms

Out[38]:

```
array([[ 0.        ,  0.        , -1.4961779 ],
       [ 1.        ,  1.        ,  0.56245248],
       [ 2.        ,  2.        ,  0.52075873],
       [ 3.        ,  3.        , -0.46177052],
       [ 4.        ,  4.        ,  0.4260685 ],
       [ 5.        ,  5.        , -0.53498361],
       [        nan,  6.        , -0.36449528]])
```

Описательные статистики

In [39]:

```
1  df.describe()
```

started 10:12:11 2020-04-12, finished in 44ms

Out[39]:

	one	two	three
count	6.000000	7.000000	7.000000
mean	2.500000	3.000000	-0.192593
std	1.870829	2.160247	0.750586
min	0.000000	0.000000	-1.496178
25%	1.250000	1.500000	-0.498377
50%	2.500000	3.000000	-0.364495
75%	3.750000	4.500000	0.473414
max	5.000000	6.000000	0.562452

Транспонирование данных

In [40]:

```
1  df.T
```

started 10:12:11 2020-04-12, finished in 20ms

Out[40]:

	a	b	c	d	e	f	g
one	0.000000	1.000000	2.000000	3.000000	4.000000	5.000000	NaN
two	0.000000	1.000000	2.000000	3.000000	4.000000	5.000000	6.000000
three	-1.496178	0.562452	0.520759	-0.461771	0.426069	-0.534984	-0.364495

Сортировка по столбцу

In [41]:

```
1  df.sort_values(by='three', ascending=False)
```

started 10:12:11 2020-04-12, finished in 20ms

Out[41]:

	one	two	three
b	1.0	1	0.562452
c	2.0	2	0.520759
e	4.0	4	0.426069
g	NaN	6	-0.364495
d	3.0	3	-0.461771
f	5.0	5	-0.534984
a	0.0	0	-1.496178

Упражнение: Сгенерируйте массив точек в 3D, создайте по нему датафрейм и отсортируйте строки лексикографически

In [42]:

```
# ВАШ КОД
pd.DataFrame(
    sps.norm().rvs((100, 3)),
    columns=['x', 'y', 'z']
).sort_values(by=['x', 'y', 'z'])
```

started 10:12:11 2020-04-12, finished in 26ms

Out[42]:

	x	y	z
11	-2.453554	0.070865	1.270085
0	-1.645365	0.680996	0.755448
17	-1.620365	1.704682	-0.238129
60	-1.523821	-0.625632	-0.641644
32	-1.517200	-0.756461	0.003390
80	-1.398230	-0.060793	-1.175577
75	-1.324837	-1.609736	-1.506753
62	-1.313305	-1.755254	0.164605
72	-1.302290	1.187174	-1.943044
70	-1.247956	0.183947	1.391994
78	-1.135145	-1.885870	0.963771
23	-1.090302	0.138006	-1.988282
25	-1.024776	-0.670445	2.099342
22	-1.013627	-0.676556	-0.195575
14	-0.981960	-0.056889	0.335150
63	-0.938201	-0.279104	-0.302362
20	-0.937994	-1.968142	-1.255503
35	-0.911286	-1.088244	1.087172
53	-0.883287	0.113176	-1.088797
47	-0.865473	2.025099	1.124176
65	-0.864217	0.312016	1.537469
58	-0.854635	0.079393	-0.787223
27	-0.851288	1.317678	-0.037738
16	-0.788237	0.021794	-0.100366
18	-0.783662	0.671847	1.025736
1	-0.744607	0.458559	2.413493
12	-0.685703	-1.303589	0.146562
82	-0.655297	0.050457	0.842924
79	-0.631054	0.936222	0.523676
57	-0.629291	0.872425	1.142377
...	...	...	...
2	0.651573	-0.221656	-1.396755

	x	y	z
24	0.684697	-0.469834	-1.055235
90	0.701518	0.871349	1.309766
66	0.712471	-1.323101	-0.719519
4	0.745422	-1.296470	0.752359
26	0.766701	-0.997251	0.310131
34	0.813565	-0.110211	0.305231
19	0.878318	0.505328	1.205696
64	0.878949	-0.533732	-0.671754
29	0.886485	1.685864	-0.703312
9	0.915882	-0.931523	-1.096249
54	0.940493	1.595040	0.121120
56	0.950962	0.435919	-0.687967
76	1.022134	-0.331206	0.315905
86	1.051156	-0.304207	-0.283757
6	1.064211	0.224760	-1.017922
41	1.087217	1.446344	0.155127
46	1.104473	1.483516	0.222925
10	1.128220	1.176469	-0.182456
28	1.129427	-1.066361	-1.730470
15	1.144520	1.610451	0.963700
74	1.179138	0.444020	0.641559
59	1.384312	-0.423372	-0.853989
99	1.456734	1.345555	0.800017
97	1.518513	-0.236113	1.678095
45	1.520634	-1.263686	0.761155
38	1.592373	-1.157783	-0.578329
50	1.657409	-1.065516	0.007514
51	2.075467	0.194988	-0.474311
33	2.301687	0.583367	-1.070727

100 rows × 3 columns

Индексация

В отличии от обычной системы индексации в Python и Numpy, в Pandas принята иная система индексации, которая является несколько нелогичной, однако, на практике часто оказывается удобной при обработке сильно неоднородных данных. Для написания продуктивного кода при обработке большого объема данных стоит использовать атрибуты `.at`, `.iat`, `.loc`, `.iloc`, `.ix`.

Если в качестве индекса указать имя столбца, получится одномерный набор данных типа Series.

In [43]:

```
1  df['one']
```

started 10:12:11 2020-04-12, finished in 5ms

Out[43]:

```
a    0.0
b    1.0
c    2.0
d    3.0
e    4.0
f    5.0
g    NaN
Name: one, dtype: float64
```

К столбцу можно обращаться как к полю объекта, если имя столбца позволяет это сделать

In [44]:

```
1  df.one
```

started 10:12:11 2020-04-12, finished in 3ms

Out[44]:

```
a    0.0
b    1.0
c    2.0
d    3.0
e    4.0
f    5.0
g    NaN
Name: one, dtype: float64
```

Индексы полученного одномерного набора данных

In [45]:

```
1  df['one'].index
```

started 10:12:11 2020-04-12, finished in 3ms

Out[45]:

```
Index(['a', 'b', 'c', 'd', 'e', 'f', 'g'], dtype='object')
```

У данного столбца есть имя, его можно получить так

In [46]:

```
1  df['one'].name
```

started 10:12:11 2020-04-12, finished in 3ms

Out[46]:

```
'one'
```

Получение элемента массива

In [47]:

```
1  df['one']['c']
```

started 10:12:11 2020-04-12, finished in 5ms

Out[47]:

```
2.0
```

Правила индексации в pandas несколько отличаются от общепринятых. Если указать диапазон индексов, то это означает диапазон строк. Причём последняя строка включается в таблицу.

In [48]:

```
1  df['b':'d']
```

started 10:12:11 2020-04-12, finished in 7ms

Out[48]:

	one	two	three
b	1.0	1	0.562452
c	2.0	2	0.520759
d	3.0	3	-0.461771

Диапазон целых чисел даёт диапазон строк с такими номерами, не включая последнюю строку (как обычно при индексировании списков). Всё это кажется довольно нелогичным.

In [49]:

```
1  df[1:3]
```

started 10:12:11 2020-04-12, finished in 19ms

Out[49]:

	one	two	three
b	1.0	1	0.562452
c	2.0	2	0.520759

Логичнее работает атрибут `loc` : первая позиция -- всегда индекс строки, а вторая -- столбца.

In [50]:

```
1  df.loc['b']
```

started 10:12:11 2020-04-12, finished in 9ms

Out[50]:

```
one      1.000000
two      1.000000
three    0.562452
Name: b, dtype: float64
```

In [51]:

```
1  df.loc['b', 'one']
```

started 10:12:11 2020-04-12, finished in 4ms

Out[51]:

```
1.0
```

In [52]:

```
1  df.loc['a':'b', 'one']
```

started 10:12:11 2020-04-12, finished in 7ms

Out[52]:

```
a    0.0
b    1.0
Name: one, dtype: float64
```

In [53]:

```
1  df.loc['a':'b', :]
```

started 10:12:11 2020-04-12, finished in 8ms

Out[53]:

	one	two	three
a	0.0	0	-1.496178
b	1.0	1	0.562452

In [54]:

```
1  df.loc[:, 'one']
```

started 10:12:11 2020-04-12, finished in 4ms

Out[54]:

```
a    0.0
b    1.0
c    2.0
d    3.0
e    4.0
f    5.0
g    NaN
Name: one, dtype: float64
```

Атрибут `iloc` подобен `loc` : первый индекс - номер строки, второй - номер столбца. Это целые числа, конец диапазона не включается (как обычно в питоне).

In [55]:

```
df.iloc[2]
```

started 10:12:11 2020-04-12, finished in 4ms

Out[55]:

```
one      2.000000
two      2.000000
three    0.520759
Name: c, dtype: float64
```

In [56]:

```
df.iloc[1:3]
```

started 10:12:11 2020-04-12, finished in 6ms

Out[56]:

	one	two	three
b	1.0	1	0.562452
c	2.0	2	0.520759

In [57]:

```
df.iloc[1:3, 0:2]
```

started 10:12:11 2020-04-12, finished in 11ms

Out[57]:

	one	two
b	1.0	1
c	2.0	2

Булевская индексация -- выбор строк с заданным условием

In [58]:

```
df[df.three > 0]
```

started 10:12:11 2020-04-12, finished in 7ms

Out[58]:

	one	two	three
b	1.0	1	0.562452
c	2.0	2	0.520759
e	4.0	4	0.426069

Упражнение

Сгенерируйте случайную целочисленную матрицу $n \times m$ ($n = 20, m = 10$), создайте из неё датафрейм, пронумеруйте столбцы случайной перестановкой чисел из $\{1, \dots, m\}$. Выберите столбцы с чётными номерами и строки, в которых чётных элементов больше, чем нечётных.

In [59]:

```
# ВАШ КОД
n, m = 20, 10
data = sps.randint.rvs(-100, 100, size=(n, m))
cols = np.arange(1, m + 1)
np.random.shuffle(cols)
task_df = pd.DataFrame(data, columns=cols)
col_mask = (cols % 2) == 0
row_mask = np.sum(data % 2, axis=1) < (m / 2)
task_df.loc[row_mask, col_mask]
```

started 10:12:11 2020-04-12, finished in 13ms

Out[59]:

	2	6	8	4	10
1	64	18	-30	94	94
4	34	73	-74	-20	-48
5	-67	39	-35	19	6
6	-34	17	-87	-80	96
8	-81	68	60	-17	74
10	-87	-97	-29	46	12
11	-74	-14	36	18	54
14	-40	94	-34	56	2
17	-90	96	56	-68	-83
19	64	14	94	16	-81

Изменение таблиц

К таблице можно добавлять новые столбцы.

In [60]:

```
df['4th'] = df['one'] * df['two']
df['flag'] = df['two'] > 2
df
```

started 10:12:11 2020-04-12, finished in 9ms

Out[60]:

	one	two	three	4th	flag
a	0.0	0	-1.496178	0.0	False
b	1.0	1	0.562452	1.0	False
c	2.0	2	0.520759	4.0	False
d	3.0	3	-0.461771	9.0	True
e	4.0	4	0.426069	16.0	True
f	5.0	5	-0.534984	25.0	True
g	NaN	6	-0.364495	NaN	True

И удалять имеющиеся.

In [61]:

```
del df['two']
df['foo'] = 0
df
```

started 10:12:11 2020-04-12, finished in 16ms

Out[61]:

	one	three	4th	flag	foo
a	0.0	-1.496178	0.0	False	0
b	1.0	0.562452	1.0	False	0
c	2.0	0.520759	4.0	False	0
d	3.0	-0.461771	9.0	True	0
e	4.0	0.426069	16.0	True	0
f	5.0	-0.534984	25.0	True	0
g	NaN	-0.364495	NaN	True	0

Изменение элемента

In [62]:

```
df.iat[1, 0] = -1

# Эквивалентные формы:
# df['one']['b'] = -1  <--  SettingWithCopyWarning
# df.at['b', 'one'] = -1

df
```

started 10:12:11 2020-04-12, finished in 18ms

Out[62]:

	one	three	4th	flag	foo
a	0.0	-1.496178	0.0	False	0
b	-1.0	0.562452	1.0	False	0
c	2.0	0.520759	4.0	False	0
d	3.0	-0.461771	9.0	True	0
e	4.0	0.426069	16.0	True	0
f	5.0	-0.534984	25.0	True	0
g	NaN	-0.364495	NaN	True	0

Добавим копию столбца `one`, в которую входят только строки до третьей.

In [63]:

```
1  df['one_tr'] = df['one'][:3]
2  df
```

started 10:12:11 2020-04-12, finished in 13ms

Out[63]:

	one	three	4th	flag	foo	one_tr
a	0.0	-1.496178	0.0	False	0	0.0
b	-1.0	0.562452	1.0	False	0	-1.0
c	2.0	0.520759	4.0	False	0	2.0
d	3.0	-0.461771	9.0	True	0	NaN
e	4.0	0.426069	16.0	True	0	NaN
f	5.0	-0.534984	25.0	True	0	NaN
g	NaN	-0.364495	NaN	True	0	NaN

Пропуски

Удаление всех строк с пропусками

In [64]:

```
1  df.dropna(how='any')
```

started 10:12:11 2020-04-12, finished in 11ms

Out[64]:

	one	three	4th	flag	foo	one_tr
a	0.0	-1.496178	0.0	False	0	0.0
b	-1.0	0.562452	1.0	False	0	-1.0
c	2.0	0.520759	4.0	False	0	2.0

Замена всех пропусков на значение

In [65]:

```
1  df.fillna(value=666)
```

started 10:12:11 2020-04-12, finished in 9ms

Out[65]:

	one	three	4th	flag	foo	one_tr
a	0.0	-1.496178	0.0	False	0	0.0
b	-1.0	0.562452	1.0	False	0	-1.0
c	2.0	0.520759	4.0	False	0	2.0
d	3.0	-0.461771	9.0	True	0	666.0
e	4.0	0.426069	16.0	True	0	666.0
f	5.0	-0.534984	25.0	True	0	666.0
g	666.0	-0.364495	666.0	True	0	666.0

Замена всех пропусков на среднее по столбцу

In [66]:

```
1  df.fillna(value=df.mean())
```

started 10:12:11 2020-04-12, finished in 9ms

Out[66]:

	one	three	4th	flag	foo	one_tr
a	0.000000	-1.496178	0.000000	False	0	0.000000
b	-1.000000	0.562452	1.000000	False	0	-1.000000
c	2.000000	0.520759	4.000000	False	0	2.000000
d	3.000000	-0.461771	9.000000	True	0	0.333333
e	4.000000	0.426069	16.000000	True	0	0.333333
f	5.000000	-0.534984	25.000000	True	0	0.333333
g	2.166667	-0.364495	9.166667	True	0	0.333333

Булевская маска пропущенных значений

In [67]:

```
1  df.isnull()
```

started 10:12:11 2020-04-12, finished in 11ms

Out[67]:

	one	three	4th	flag	foo	one_tr
a	False	False	False	False	False	False
b	False	False	False	False	False	False
c	False	False	False	False	False	False
d	False	False	False	False	False	True
e	False	False	False	False	False	True
f	False	False	False	False	False	True
g	True	False	True	False	False	True

Простые операции

Создадим таблицу из массива случайных чисел.

In [68]:

```
1 ▾ df1 = pd.DataFrame(sps.uniform.rvs(size=(10, 4)),
2                      columns=['A', 'B', 'C', 'D'])
3   df1
```

started 10:12:11 2020-04-12, finished in 9ms

Out[68]:

	A	B	C	D
0	0.250877	0.754734	0.276125	0.720824
1	0.575403	0.900451	0.877726	0.269396
2	0.049269	0.333501	0.266401	0.631570
3	0.070402	0.018452	0.765624	0.105073
4	0.277331	0.247629	0.819190	0.020735
5	0.950293	0.004797	0.389922	0.409014
6	0.646476	0.772134	0.681104	0.856197
7	0.230166	0.766094	0.232642	0.384684
8	0.370390	0.967563	0.482347	0.979930
9	0.740447	0.546133	0.475937	0.277485

In [69]:

```
df2 = pd.DataFrame(sps.uniform.rvs(size=(7, 3)),
                   columns=['A', 'B', 'C'])
df2
```

started 10:12:11 2020-04-12, finished in 21ms

Out[69]:

	A	B	C
0	0.788811	0.886728	0.754926
1	0.649572	0.911963	0.183802
2	0.662806	0.555376	0.101021
3	0.282103	0.026328	0.829280
4	0.476916	0.063974	0.021611
5	0.751839	0.757885	0.367958
6	0.158158	0.729036	0.280558

In [70]:

```
df1 + df2
```

started 10:12:11 2020-04-12, finished in 19ms

Out[70]:

	A	B	C	D
0	1.039689	1.641463	1.031051	NaN
1	1.224976	1.812414	1.061528	NaN
2	0.712075	0.888877	0.367423	NaN
3	0.352506	0.044780	1.594904	NaN
4	0.754246	0.311603	0.840801	NaN
5	1.702132	0.762682	0.757880	NaN
6	0.804634	1.501170	0.961662	NaN
7	NaN	NaN	NaN	NaN
8	NaN	NaN	NaN	NaN
9	NaN	NaN	NaN	NaN

In [71]:

```
1  2 * df1 + 3
```

started 10:12:11 2020-04-12, finished in 11ms

Out[71]:

	A	B	C	D
0	3.501754	4.509468	3.552250	4.441647
1	4.150807	4.800901	4.755452	3.538793
2	3.098538	3.667002	3.532802	4.263140
3	3.140805	3.036905	4.531249	3.210146
4	3.554661	3.495259	4.638380	3.041469
5	4.900586	3.009594	3.779844	3.818029
6	4.292952	4.544267	4.362207	4.712395
7	3.460333	4.532188	3.465284	3.769367
8	3.740779	4.935125	3.964694	4.959860
9	4.480894	4.092267	3.951873	3.554971

In [72]:

```
1  np.sin(df1)
```

started 10:12:11 2020-04-12, finished in 11ms

Out[72]:

	A	B	C	D
0	0.248254	0.685095	0.272629	0.660004
1	0.544173	0.783607	0.769288	0.266150
2	0.049249	0.327353	0.263261	0.590412
3	0.070344	0.018451	0.692987	0.104880
4	0.273789	0.245106	0.730593	0.020733
5	0.813586	0.004797	0.380116	0.397705
6	0.602377	0.697665	0.629651	0.755356
7	0.228139	0.693326	0.230549	0.375266
8	0.361979	0.823505	0.463860	0.830458
9	0.674618	0.519387	0.458171	0.273938

Построим графики кумулятивных сумм

In [73]:

```
cs = df1.cumsum()
cs
```

started 10:12:11 2020-04-12, finished in 16ms

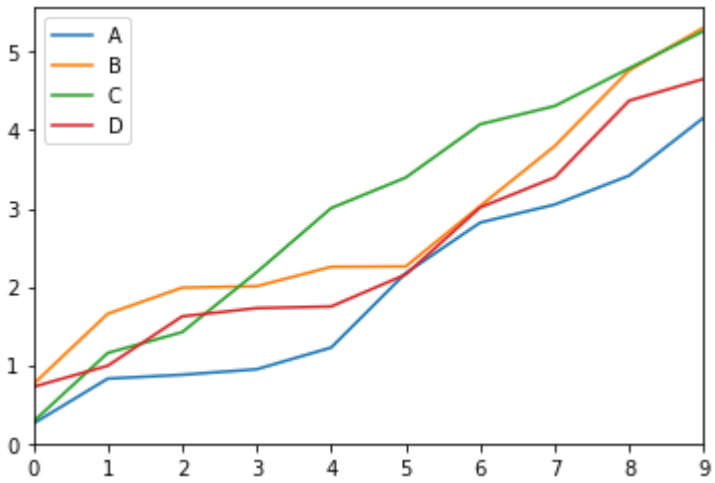
Out[73]:

	A	B	C	D
0	0.250877	0.754734	0.276125	0.720824
1	0.826281	1.655185	1.153851	0.990220
2	0.875550	1.988686	1.420252	1.621790
3	0.945952	2.007138	2.185876	1.726863
4	1.223283	2.254768	3.005066	1.747598
5	2.173576	2.259565	3.394988	2.156612
6	2.820052	3.031698	4.076092	3.012809
7	3.050218	3.797792	4.308734	3.397493
8	3.420608	4.765355	4.791081	4.377423
9	4.161055	5.311488	5.267017	4.654908

In [74]:

```
cs.plot()
plt.show()
```

started 10:12:11 2020-04-12, finished in 281ms



Упражнение

Сгенерируйте случайную выборку $X_1, \dots, X_n$ $(n = 100)$ из стандартного нормального распределения, соберите из неё `pd.DataFrame`, замените случайные 10% элементов на пропуски (`np.nan`), а затем добавьте по столбцу для оценок первых 4 моментов на префиксах —

$$\frac{1}{m}\sum_{i=1}^{m} X_i^k,\ i \in \overline{1,m},\ m \in \overline{1,n},\ k \in \overline{1,4}$$

Ваша функция должна корректно обрабатывать пропуски. В конце постройте график.

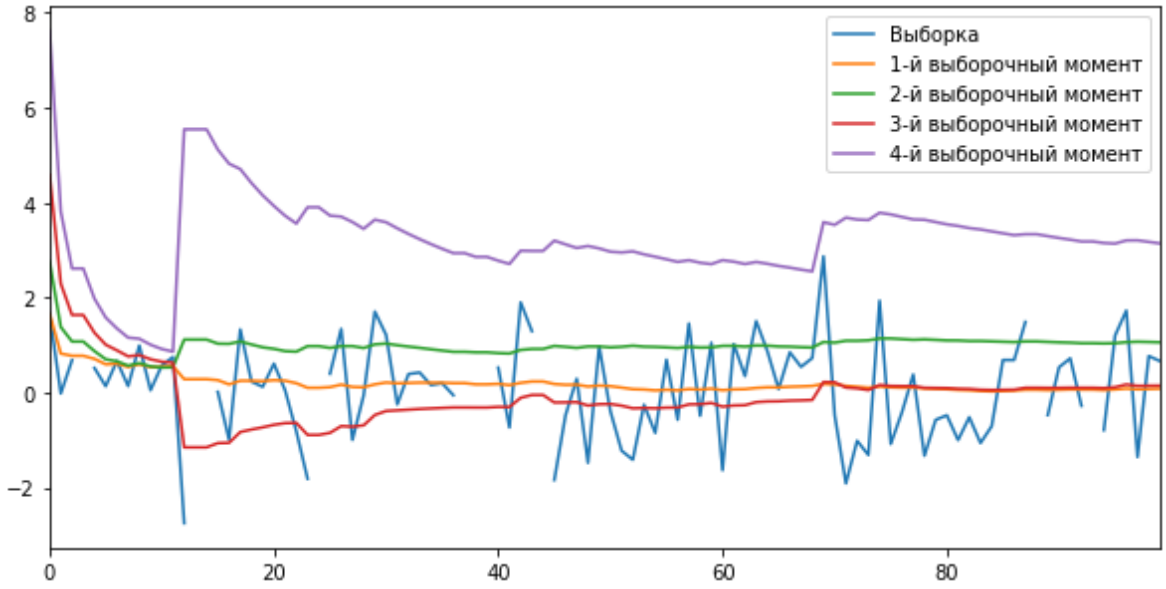
In [75]:

```
# YOUR CODE
n = 100
sample = sps.norm.rvs(size=n)
index = np.random.choice(np.arange(n), int(0.1 * n), replace=True)
sample[index] = np.nan
sample_df = pd.DataFrame(sample, columns=['Выборка'])

for k in range(1, 5):
    sample_df['{}-й выборочный момент'.format(k)] = (
        sample_df['Выборка'] ** k
    ).expanding().apply(np.nanmean)

sample_df.plot(figsize=(10, 5));
```

started 10:12:11 2020-04-12, finished in 379ms



Чтение данных

Загрузка текстовых файлов табличного вида производится с помощью функции `pd.read_csv` .
Основные аргументы следующие:

- `filepath_or_buffer` --- пусть к файлу;
- `sep` --- разделитель колонок в строке (запятая, табуляция и т.д.);
- `header` --- номер строки или список номеров строк, используемых в качестве имен колонок;
- `names` --- список имен, которые будут использованы в качестве имен колонок;
- `index_col` --- колонка, используемая в качестве индекса;
- `usecols` --- список имен колонок, которые будут загружены;
- `nrows` --- сколько строк прочитать;
- `skiprows` --- номера строк с начала, которые нужно пропустить;
- `skipfooter` --- сколько строк в конце пропустить;
- `na_values` --- список значений, которые распознавать как пропуски;
- `parse_dates` --- распознавать ли даты, можно передать номера строк;
- `date_parser` --- парсер дат;
- `dayfirst` --- день записывается перед месяцем или после;
- `thousands` --- разделитель тысяч;
- `decimal` --- разделитель целой и дробной частей;
- `comment` --- символ начала комментария.

Полный список параметров:

```
pd.read_csv(filepath_or_buffer, sep=',', delimiter=None, header='infer',
names=None, index_col=None, usecols=None, squeeze=False, prefix=None,
mangle_dupe_cols=True, dtype=None, engine=None, converters=None, true_values=None,
false_values=None, skipinitialspace=False, skiprows=None, nrows=None,
na_values=None, keep_default_na=True, na_filter=True, verbose=False,
skip_blank_lines=True, parse_dates=False, infer_datetime_format=False,
keep_date_col=False, date_parser=None, dayfirst=False, iterator=False,
chunksize=None, compression='infer', thousands=None, decimal=b'.',
lineterminator=None, quotechar='"', quoting=0, escapechar=None, comment=None,
encoding=None, dialect=None, tupleize_cols=False, error_bad_lines=True,
warn_bad_lines=True, skipfooter=0, skip_footer=0, doublequote=True,
delim_whitespace=False, as_recarray=False, compact_ints=False, use_unsigned=False,
low_memory=True, buffer_lines=None, memory_map=False, float_precision=None)
```

Загрузка таблиц формата Excel

Загрузка таблиц формата Excel производится с помощью функции `pd.read_excel` . Основные аргументы следующие:

- `io` --- пусть к файлу;
- `sheetname` --- какие листы таблицы загрузить;
- Остальные параметры аналогично.

```
pd.read_excel(io, sheetname=0, header=0, skiprows=None, skip_footer=0,
index_col=None, names=None, parse_cols=None, parse_dates=False, date_parser=None,
na_values=None, thousands=None, convert_float=True, has_index_names=None,
converters=None, dtype=None, true_values=None, false_values=None, engine=None,
squeeze=False, **kwds)
```

Запись данных

Запись таблицы в текстовый файл производится с помощью функции `df.to_csv` . Основные аргументы следующие:

- `df` --- DataFrame, который нужно записать;
- `path_or_buf` --- путь, куда записать;
- `sep` --- разделитель колонок в строке (запятая, табуляция и т.д.);
- `na_rep` --- как записать пропуски;
- `float_format` --- формат записи дробных чисел;
- `columns` --- какие колонки записать;
- `header` --- как назвать колонки при записи;
- `index` --- записывать ли индексы в файл;
- `index_label` --- имена индексов, которые записать в файл.

Полный список параметров:

```
df.to_csv(path_or_buf=None, sep=',', na_rep='', float_format=None, columns=None,
header=True, index=True, index_label=None, mode='w', encoding=None,
compression=None, quoting=None, quotechar='"', line_terminator='\n',
chunksize=None, tupleize_cols=False, date_format=None, doublequote=True,
escapechar=None, decimal='.')
```

Запись таблицы в формат Excel производится с помощью функции `df.to_excel` . Основные аргументы аналогичные. Полный список параметров:

```
df.to_excel(excel_writer, sheet_name='Sheet1', na_rep='', float_format=None,
columns=None, header=True, index=True, index_label=None, startrow=0, startcol=0,
engine=None, merge_cells=True, encoding=None, inf_rep='inf', verbose=True,
freeze_panes=None)
```

Примеры чтения данных и работы с датами

Прочитаем файл, который содержит два столбца -- дата и число. Столбцы разделяются табуляцией.

In [76]:

```
df = pd.read_csv('./example.csv', sep='\t', parse_dates=[0])
df.head()
```

started 10:12:12 2020-04-12, finished in 44ms

Out[76]:

	Time	Value
0	2019-01-09	66
1	2019-02-09	34
2	2019-03-09	18
3	2019-04-09	32
4	2019-05-09	84

В информации о таблице видим, что дата определилась, т.к. формат колонки `Time` обозначен как `datetime64[ns]` .

In [77]:

```
df.info()
```

started 10:12:12 2020-04-12, finished in 4ms

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 18 entries, 0 to 17
Data columns (total 2 columns):
Time     18 non-null datetime64[ns]
Value    18 non-null int64
dtypes: datetime64[ns](1), int64(1)
memory usage: 416.0 bytes
```

Но при печати понимаем, что часть дат распозналась неправильно. Если число месяца меньше 13, то pandas путает день и месяц. В одном и том же столбце. Кошмар...

In [78]:

```
1  df['Time']
```

started 10:12:12 2020-04-12, finished in 4ms

Out[78]:

```
0    2019-01-09
1    2019-02-09
2    2019-03-09
3    2019-04-09
4    2019-05-09
5    2019-06-09
6    2019-07-09
7    2019-08-09
8    2019-09-09
9    2019-10-09
10   2019-11-09
11   2019-12-09
12   2019-09-13
13   2019-09-14
14   2019-09-15
15   2019-09-16
16   2019-09-17
17   2019-09-18
Name: Time, dtype: datetime64[ns]
```

Укажем, что день всегда следует первым. Теперь все правильно

In [79]:

```
1  df = pd.read_csv('./example.csv', sep='\t', parse_dates=[0],
2                   dayfirst=True)
3  df['Time']
```

started 10:12:12 2020-04-12, finished in 8ms

Out[79]:

```
0    2019-09-01
1    2019-09-02
2    2019-09-03
3    2019-09-04
4    2019-09-05
5    2019-09-06
6    2019-09-07
7    2019-09-08
8    2019-09-09
9    2019-09-10
10   2019-09-11
11   2019-09-12
12   2019-09-13
13   2019-09-14
14   2019-09-15
15   2019-09-16
16   2019-09-17
17   2019-09-18
Name: Time, dtype: datetime64[ns]
```

Панды довольно ленивые, и если не попросить pandas распознать дату, то ничего делать не будет -- оставит ее как `object` .

In [80]:

```
df = pd.read_csv('./example.csv', sep='\t')
df.info()
```

started 10:12:12 2020-04-12, finished in 7ms

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 18 entries, 0 to 17
Data columns (total 2 columns):
Time     18 non-null object
Value    18 non-null int64
dtypes: int64(1), object(1)
memory usage: 416.0+ bytes
```

Тогда можно воспользоваться функцией `pd.to_datetime`

In [81]:

```
df['Time'] = pd.to_datetime(df['Time'], dayfirst=True)
df['Time']
```

started 10:12:12 2020-04-12, finished in 5ms

Out[81]:

```
0    2019-09-01
1    2019-09-02
2    2019-09-03
3    2019-09-04
4    2019-09-05
5    2019-09-06
6    2019-09-07
7    2019-09-08
8    2019-09-09
9    2019-09-10
10   2019-09-11
11   2019-09-12
12   2019-09-13
13   2019-09-14
14   2019-09-15
15   2019-09-16
16   2019-09-17
17   2019-09-18
Name: Time, dtype: datetime64[ns]
```

Установим дату как индекс, получив временной ряд. Это будет изучено в следующем году.

In [82]:

```
df = df.set_index('Time')
df
```

started 10:12:12 2020-04-12, finished in 9ms

Out[82]:

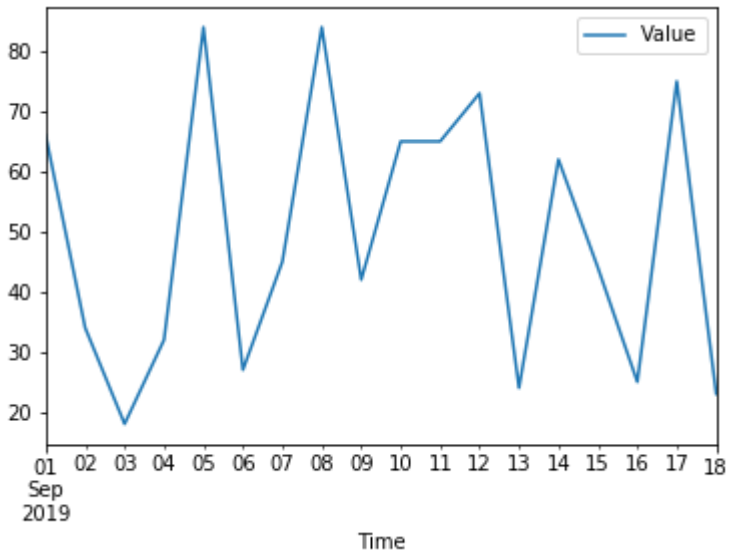
Time	Value
2019-09-01	66
2019-09-02	34
2019-09-03	18
2019-09-04	32
2019-09-05	84
2019-09-06	27
2019-09-07	45
2019-09-08	84
2019-09-09	42
2019-09-10	65
2019-09-11	65
2019-09-12	73
2019-09-13	24
2019-09-14	62
2019-09-15	44
2019-09-16	25
2019-09-17	75
2019-09-18	23

Его можно нарисовать

In [83]:

```
1  df.plot();
```

started 10:12:12 2020-04-12, finished in 287ms



Усреднение данных по трем дням

In [84]:

```
1  df.resample('3D').mean()
```

started 10:12:12 2020-04-12, finished in 18ms

Out[84]:

Time	Value
2019-09-01	39.333333
2019-09-04	47.666667
2019-09-07	57.000000
2019-09-10	67.666667
2019-09-13	43.333333
2019-09-16	41.000000

Интервалы времени

Интервал времени задается объектом `pd.Timedelta`

Возможные обозначения интервалов: `'Y', 'M', 'W', 'D', 'days', 'day', 'hours', hour', 'hr', 'h', 'm', 'minute', 'min', 'minutes', 'T', 'S', 'seconds', 'sec', 'second', 'ms', 'milliseconds', 'millisecond', 'milli', 'millis', 'L', 'us', 'microseconds', 'microsecond', 'micro', 'micros', 'U', 'ns', 'nanoseconds', 'nano', 'nanos', 'nanosecond', 'N'`

Например, интервал времени в 5 недель 6 дней 5 часов 37 минут 23 секунды 12 милисекунд:

In [85]:

```
1  pd.Timedelta('5W 6 days 5hr 37min 23sec 12ms')
```

started 10:12:12 2020-04-12, finished in 6ms

Out[85]:

```
Timedelta('41 days 05:37:23.012000')
```

Поробуем понять что такое `Y` и `M`.

In [86]:

```
1  pd.Timedelta('1Y'), pd.Timedelta('1M')
```

started 10:12:12 2020-04-12, finished in 5ms

Out[86]:

```
(Timedelta('365 days 05:49:12'), Timedelta('0 days 00:01:00'))
```

`Y` обозначает год. Он сделан таким из-за високосных годов. Поскольку месяцы разной длины, то их вообще нельзя здесь задать. Поэтому `M` обозначает минуты.

Интервал можно добавить к какой-нибудь дате, или вычесть из нее.

In [87]:

```
1 ▾ pd.to_datetime('2019.09.18 18:30') \
2       + pd.Timedelta('8hr 37min 23sec 12ms')
```

started 10:12:12 2020-04-12, finished in 17ms

Out[87]:

```
Timestamp('2019-09-19 03:07:23.012000')
```

In [88]:

```
1 ▾ pd.to_datetime('2019.09.18 18:30') \
2       - pd.Timedelta('20hr 50min 23sec 12ms')
```

started 10:12:12 2020-04-12, finished in 8ms

Out[88]:

```
Timestamp('2019-09-17 21:39:36.988000')
```

Сделать регулярный список дат позволяет функция `pd.timedelta_range`, которая реализует функционал `range` для дат. Ей нужно передать **ровно три аргумента** из следующих четырех:

- `start` --- интервал начала отчета;
- `end` --- интервал окончания отчета;
- `periods` --- количество интервалов;
- `freq` --- частота отсчета.

Пример

Врач на протяжении дня измеряет пациенту температуру каждые 3 часа в течение 2 недель. Также пациенту необходимо спать с 11 вечера до 7 утра. Каждый день измерения температуры начинаются в 8 часов. Первое измерение 22 марта 2020 года. Определите моменты времени, когда нужно измерить пациенту температуру.

In [89]:

```
# Периоды измерения температуры днем
periods = pd.timedelta_range(start='8H', freq='3H', end='23H')
periods
```

started 10:12:12 2020-04-12, finished in 8ms

Out[89]:

```
TimedeltaIndex(['08:00:00', '11:00:00', '14:00:00', '17:00:00', '20:0
0:00',
                '23:00:00'],
               dtype='timedelta64[ns]', freq='3H')
```

In [90]:

```
# Даты измерний температуры
dates = pd.to_datetime('2020.03.22') \
    + pd.timedelta_range(start=0, freq='1D', end='2W')
dates
```

started 10:12:12 2020-04-12, finished in 8ms

Out[90]:

```
DatetimeIndex(['2020-03-22', '2020-03-23', '2020-03-24', '2020-03-25',
               '2020-03-26', '2020-03-27', '2020-03-28', '2020-03-29',
               '2020-03-30', '2020-03-31', '2020-04-01', '2020-04-02',
               '2020-04-03', '2020-04-04', '2020-04-05'],
              dtype='datetime64[ns]', freq='D')
```

In [91]:

```
# Время измерния температуры
n, m = len(dates), len(periods)
dates_new = dates.repeat(m)
periods_new = pd.to_timedelta(np.tile(periods, n))
time = dates_new + periods_new
time
```

started 10:12:12 2020-04-12, finished in 8ms

Out[91]:

```
DatetimeIndex(['2020-03-22 08:00:00', '2020-03-22 11:00:00',
               '2020-03-22 14:00:00', '2020-03-22 17:00:00',
               '2020-03-22 20:00:00', '2020-03-22 23:00:00',
               '2020-03-23 08:00:00', '2020-03-23 11:00:00',
               '2020-03-23 14:00:00', '2020-03-23 17:00:00',
               '2020-03-23 20:00:00', '2020-03-23 23:00:00',
               '2020-03-24 08:00:00', '2020-03-24 11:00:00',
               '2020-03-24 14:00:00', '2020-03-24 17:00:00',
               '2020-03-24 20:00:00', '2020-03-24 23:00:00',
               '2020-03-25 08:00:00', '2020-03-25 11:00:00',
               '2020-03-25 14:00:00', '2020-03-25 17:00:00',
               '2020-03-25 20:00:00', '2020-03-25 23:00:00',
               '2020-03-26 08:00:00', '2020-03-26 11:00:00',
               '2020-03-26 14:00:00', '2020-03-26 17:00:00',
               '2020-03-26 20:00:00', '2020-03-26 23:00:00',
               '2020-03-27 08:00:00', '2020-03-27 11:00:00',
               '2020-03-27 14:00:00', '2020-03-27 17:00:00',
               '2020-03-27 20:00:00', '2020-03-27 23:00:00',
               '2020-03-28 08:00:00', '2020-03-28 11:00:00',
               '2020-03-28 14:00:00', '2020-03-28 17:00:00',
               '2020-03-28 20:00:00', '2020-03-28 23:00:00',
               '2020-03-29 08:00:00', '2020-03-29 11:00:00',
               '2020-03-29 14:00:00', '2020-03-29 17:00:00',
               '2020-03-29 20:00:00', '2020-03-29 23:00:00',
               '2020-03-30 08:00:00', '2020-03-30 11:00:00',
               '2020-03-30 14:00:00', '2020-03-30 17:00:00',
               '2020-03-30 20:00:00', '2020-03-30 23:00:00',
               '2020-03-31 08:00:00', '2020-03-31 11:00:00',
               '2020-03-31 14:00:00', '2020-03-31 17:00:00',
               '2020-03-31 20:00:00', '2020-03-31 23:00:00',
               '2020-04-01 08:00:00', '2020-04-01 11:00:00',
               '2020-04-01 14:00:00', '2020-04-01 17:00:00',
               '2020-04-01 20:00:00', '2020-04-01 23:00:00',
               '2020-04-02 08:00:00', '2020-04-02 11:00:00',
               '2020-04-02 14:00:00', '2020-04-02 17:00:00',
               '2020-04-02 20:00:00', '2020-04-02 23:00:00',
               '2020-04-03 08:00:00', '2020-04-03 11:00:00',
               '2020-04-03 14:00:00', '2020-04-03 17:00:00',
               '2020-04-03 20:00:00', '2020-04-03 23:00:00',
               '2020-04-04 08:00:00', '2020-04-04 11:00:00',
               '2020-04-04 14:00:00', '2020-04-04 17:00:00',
               '2020-04-04 20:00:00', '2020-04-04 23:00:00',
               '2020-04-05 08:00:00', '2020-04-05 11:00:00',
               '2020-04-05 14:00:00', '2020-04-05 17:00:00',
               '2020-04-05 20:00:00', '2020-04-05 23:00:00'],
              dtype='datetime64[ns]', freq=None)
```

Введение в анализ данных, 2020

Команда Физтех-статистики

https://mipt-stats.gitlab.io/ (https://mipt-stats.gitlab.io/)

Частично использованы материалы http://www.inp.nsk.su/~grozin/python/pandas.html (http://www.inp.nsk.su/~grozin/python/pandas.html) и http://pandas.pydata.org/pandas-docs/stable/10min.html (http://pandas.pydata.org/pandas-docs/stable/10min.html)