Библиотека pandas (часть 2)

Урок 6.1. Применение функций и метод .apply()

В этом модуле мы продолжим работать с библиотекой pandas и поговорим о более продвинутых способах обработки данных с помощью этой библиотеки. Но прежде чем обсуждать различные методы, давайте загрузим небольшой датасет из csv-файла. Формат *csv* расшифровывается как *comma separated values*, данные хранятся в текстовом виде, и столбцы отделяются друг от друга запятыми.

Датасет содержит информацию о том, как люди сбрасывали вес в течение 3 месяцев с применением различных методик, и как при этом менялась их самооценка. Давайте сначала загрузим файл в Jupyter. (Показать, как загрузить файл через *Upload* в *Home* в рабочую папку).

In [1]:

```
import pandas as pd
import numpy as np  # може пригодится
df = pd.read_csv("WeightLoss.csv")
```

In [2]:

```
df.head()
```

Out[2]:

	id	group	w1	w2	w3	se1	se2	se3
0	1	Control	4	3	3.0	14.0	13.0	15.0
1	2	Control	4	4	3.0	13.0	14.0	17.0
2	3	Control	4	3	1.0	17.0	12.0	16.0
3	4	Control	3	2	1.0	11.0	11.0	12.0
4	5	Control	5	3	2.0	16.0	15.0	14.0

Переменные:

- `group` : экспериментальная группа: контрольная (`Control`), диета (`Diet`), диета и упражнения `DietEx` ;
- `w1` : потеря веса после 1-го месяца эксперимента;
- `w2` : потеря веса после 2-го месяца эксперимента;
- `w3` : потеря веса после 3-го месяца эксперимента;
- `se1` : самооценка после 1-го месяца эксперимента;
- `se2` : самооценка после 2-го месяца эксперимента;
- `se3` : самооценка после 3-го месяца эксперимента.

Прежде чем переходить к обсуждению методов, которые позволяют применять функции к столбцам или строкам датафрейма, давайте вспомним, как добавить новые столбцы на основе старых в датафрейм. Создадим столбец `total` с общим числом килограммов, потерянных за три месяца, а потом на его основе создадим столбец с той же информацией, но уже с граммах:

In [3]:

```
df['total'] = df['w1'] + df['w2'] + df['w3']
df['total_gr'] = df['total'] * 1000
df.head()
```

Out[3]:

	id	group	w1	w2	w3	se1	se2	se3	total	total_gr
0	1	Control	4	3	3.0	14.0	13.0	15.0	10.0	10000.0
1	2	Control	4	4	3.0	13.0	14.0	17.0	11.0	11000.0
2	3	Control	4	3	1.0	17.0	12.0	16.0	8.0	8000.0
3	4	Control	3	2	1.0	11.0	11.0	12.0	6.0	6000.0
4	5	Control	5	3	2.0	16.0	15.0	14.0	10.0	10000.0

Если бы у нас было много столбцов (например, данные были бы не за 3 месяца, а за 12 или 24), складывать их «вручную» было бы неудобно. Поэтому у нас, скорее всего, возникло бы желание выбрать столбцы через срез и применить к ним какую-нибудь готовую функцию. В `pandas` это можно сделать с помощью `.apply()`. Давайте с ним познакомимся пока на примере попроще, применительно к одному столбцу. Например, логарифмируем значения в столбце `total_gr`. Для этого нам нужно выбрать сам столбец, действия с которым нас интересуют, дописать к нему метод `.apply()`, от английского «применять» и указать внутри скобок, в качестве аргумента, функцию, котоая будет реализовывать желаемую операцию:

In [4]:

```
df['total_gr'].apply(np.log)  # применяем log из numpy
```

Out[4]:

```
0     9.210340
1     9.305651
2     8.987197
3     8.699515
4     9.210340
5     9.615805
6     9.615805
7     9.210340
8     9.210340
9     8.987197
10    8.987197
11    8.987197
12    9.305651
13    9.210340
14    9.680344
15    9.392662
16    8.699515
17    9.546813
18    8.987197
19    8.853665
20    9.546813
21    9.740969
22    9.104980
23    9.546813
24    9.798127
25         NaN
26    9.852194
27    8.987197
28    9.104980
29    9.472705
30    9.615805
31    9.680344
32    9.903488
33    9.615805
Name: total_gr, dtype: float64
```

Получился новый столбец, новый объект типа `pandas Series`. Можно было бы добавить его в датасет, но такие вещи мы делать уже умеем и так, поэтому давайте перейдем к более интересной задаче: добавим столбец, в котором будет сохранены значения средней потери веса за три месяца по каждому человеку. Выберем в помощью `.loc` и текстового среза нужные столбцы и применим к ним функцию `mean` для среднего, плюс, укажем, что эта функция должна применяться по строкам, то есть среднее значение должно считаться по каждому человеку:

In [5]:

```
df['avloss'] = df.loc[:,'w1':'w3'].apply(np.mean, axis=1)
df.head()
```

Out[5]:

	id	group	w1	w2	w3	se1	se2	se3	total	total_gr	avloss
0	1	Control	4	3	3.0	14.0	13.0	15.0	10.0	10000.0	3.333333
1	2	Control	4	4	3.0	13.0	14.0	17.0	11.0	11000.0	3.666667
2	3	Control	4	3	1.0	17.0	12.0	16.0	8.0	8000.0	2.666667
3	4	Control	3	2	1.0	11.0	11.0	12.0	6.0	6000.0	2.000000
4	5	Control	5	3	2.0	16.0	15.0	14.0	10.0	10000.0	3.333333

Если бы мы написали `axis=0`, то получили бы среднюю потерю веса по всем людям за каждый месяц (три значения):

In [6]:

```
df.loc[:,'w1':'w3'].apply(np.mean, axis=0)
```

Out[6]:

```
w1    5.294118
w2    4.352941
w3    2.212121
dtype: float64
```

Самое интересное и полезное: в `.apply()` можно прописывать свои функции, которые мы заранее определим. Напишем функцию, которая будет считать размах: вычитать из максимального значения минимальное и возвращать результат. Напишем небольшую lambda-функцию в Python и назовём её `f`. Мы не обсуждали отдельно написание собственных функций, но lambda-функции устроены несложно. Сначала мы указываем название функции, потом после знака равенства начинаем её определять. Стартуем с ключевого слова `lambda`, чтобы Python понимал, что это функция. После указываем аргумент – то, с чем функция должна работать, то, что подаётся ей на вход. Назвать его мы можем как угодно, у нас будет `x`. Далее через двоеточие мы прописываем, что с этим `x` нужно сделать, то есть вернуть на выходе, в результате исполнения функции.

In [7]:

```
f = lambda x: x.max() - x.min()
```

Здесь `x` – это какой-то перечень значений, по нему мы считаем минимум и максимум, а потом из одного вычитаем другое. Применим написанную нами функцию к тем же столбцам и скажем, что опять функция должна применяться по строкам – к каждому человеку:

In [8]:

```
df['wrange'] = df.loc[:,'w1':'w3'].apply(f, axis=1)
df.head()
```

Out[8]:

	id	group	w1	w2	w3	se1	se2	se3	total	total_gr	avloss	wrange
0	1	Control	4	3	3.0	14.0	13.0	15.0	10.0	10000.0	3.333333	1.0
1	2	Control	4	4	3.0	13.0	14.0	17.0	11.0	11000.0	3.666667	1.0
2	3	Control	4	3	1.0	17.0	12.0	16.0	8.0	8000.0	2.666667	3.0
3	4	Control	3	2	1.0	11.0	11.0	12.0	6.0	6000.0	2.000000	2.0
4	5	Control	5	3	2.0	16.0	15.0	14.0	10.0	10000.0	3.333333	3.0

Что получилось? Для каждого участника была посчитана максимальная потеря веса за 3 месяца, потом минимальная, посчитан результат и сохранён в отдельный столбец `wrange` .

Урок 6.2. Применение функций: группировка и агрегирование

In [2]:

```
import pandas as pd
import numpy as np
df = pd.read_csv("WeightLoss.csv")
df['total'] = df['w1'] + df['w2'] + df['w3']
df['total_gr'] = df['total'] * 1000
```

Иногда нас интересуют сводные характеристики не по всей таблице, а по группам: например, хочется посмотреть, сколько килограммов в сумме потеряли люди, которые сидели на диете и те, кто помимо диеты выполнял комплекс упражнений. Для группировки в `pandas` используется метод `groupby()`:

In [3]:

```
df.groupby('group')
```

Out[3]:

```
<pandas.core.groupby.groupby.DataFrameGroupBy object at 0x1100a62e8>
```

Результат как таковой от нас скрыт, это особый объект в `pandas`, результат представляет собой список пар «название группы и сам датафрейм». Чтобы увидеть пример результата явно, сконвертируем в список и посмотрим на первый элемент:

In [4]:

```
list(df.groupby('group'))[0]
```

Out[4]:

```
('Control',     id    group  w1  w2   w3   se1   se2   se3  total  total_gr
 0    1  Control   4   3  3.0  14.0  13.0  15.0   10.0   10000.0
 1    2  Control   4   4  3.0  13.0  14.0  17.0   11.0   11000.0
 2    3  Control   4   3  1.0  17.0  12.0  16.0    8.0    8000.0
 3    4  Control   3   2  1.0  11.0  11.0  12.0    6.0    6000.0
 4    5  Control   5   3  2.0  16.0  15.0  14.0   10.0   10000.0
 5    6  Control   6   5  4.0  17.0  18.0  18.0   15.0   15000.0
 6    7  Control   6   5  4.0  17.0  16.0  19.0   15.0   15000.0
 7    8  Control   5   4  1.0   NaN   NaN   NaN   10.0   10000.0
 8    9  Control   5   4  1.0  14.0  14.0  15.0   10.0   10000.0
 9   10  Control   3   3  2.0  14.0  15.0  13.0    8.0    8000.0
 10  11  Control   4   2  2.0  16.0  16.0  11.0    8.0    8000.0
 11  12  Control   5   2  1.0  15.0  13.0  16.0    8.0    8000.0)
```

Теперь попробуем к каждой группе применить функцию, которая будет суммировать значения по каждому показателю в каждой группе. Тут не понадобится `.apply()`, есть специальный метод для агрегирования – `agg()`.

In [5]:

```
df.groupby('group').agg('sum')  # название функции - в кавычках
```

Out[5]:

group	id	w1	w2	w3	se1	se2	se3	total	total_gr
Control	78	54	40	25.0	164.0	157.0	166.0	119.0	119000.0
Diet	222	64	47	27.0	178.0	165.0	194.0	138.0	138000.0
DietEx	295	62	61	21.0	152.0	133.0	159.0	133.0	133000.0

А теперь посчитаем средние показателей по каждой группе:

In [6]:

```
df.groupby('group').agg('mean')
```

Out[6]:

group	id	w1	w2	w3	se1	se2	se3	total	t
Control	6.5	4.500000	3.333333	2.083333	14.909091	14.272727	15.090909	9.916667	9916.
Diet	18.5	5.333333	3.916667	2.250000	14.833333	13.750000	16.166667	11.500000	11500.
DietEx	29.5	6.200000	6.100000	2.333333	15.200000	13.300000	17.666667	14.777778	14777.

Если нужно применить сразу несколько функций, их можно оформить в виде списка. Найдем минимальное, максимальное и среднее значение показателей по каждой группе:

In [7]:

```
df.groupby('group').agg(['min', 'max', 'mean'])
```

Out[7]:

	id			w1			w2	w3	...	se2	
	min	max	mean	min	max	mean	min	max	mean	min	...	mean	min
group													
Control	1	12	6.5	3	6	4.500000	2	5	3.333333	1.0	...	14.272727	11.0
Diet	13	24	18.5	3	7	5.333333	2	6	3.916667	1.0	...	13.750000	11.0
DietEx	25	34	29.5	3	9	6.200000	4	9	6.100000	1.0	...	13.300000	16.0

3 rows × 27 columns

Как и в случае с `.apply()`, внутри `.agg()` можно прописывать свои функции, но тогда уже их

название должно идти без кавычек.

Урок 6.3. Сортировка и упорядочение

In [3]:

```
import pandas as pd
import numpy as np
df = pd.read_csv("WeightLoss.csv")
df['total'] = df['w1'] + df['w2'] + df['w3']
df['total_gr'] = df['total'] * 1000
```

Попробуем отсортировать строки в таблице по значениям в каком-нибудь столбце. Для этого нам пригодится метод `.sort_values()`. Отсортируем строки по показателю `total`:

In [4]:

```
df.sort_values('total')
```

Out[4]:

	id	group	w1	w2	w3	se1	se2	se3	total	total_gr
16	17	Diet	3	2	1.0	16.0	17.0	15.0	6.0	6000.0
3	4	Control	3	2	1.0	11.0	11.0	12.0	6.0	6000.0
19	20	Diet	4	2	1.0	12.0	11.0	11.0	7.0	7000.0
2	3	Control	4	3	1.0	17.0	12.0	16.0	8.0	8000.0
27	28	DietEx	3	4	1.0	16.0	13.0	NaN	8.0	8000.0
18	19	Diet	4	3	1.0	12.0	11.0	14.0	8.0	8000.0
9	10	Control	3	3	2.0	14.0	15.0	13.0	8.0	8000.0
10	11	Control	4	2	2.0	16.0	16.0	11.0	8.0	8000.0
11	12	Control	5	2	1.0	15.0	13.0	16.0	8.0	8000.0
28	29	DietEx	3	5	1.0	13.0	13.0	16.0	9.0	9000.0
22	23	Diet	4	3	2.0	15.0	15.0	15.0	9.0	9000.0
13	14	Diet	5	4	1.0	13.0	14.0	15.0	10.0	10000.0
0	1	Control	4	3	3.0	14.0	13.0	15.0	10.0	10000.0
8	9	Control	5	4	1.0	14.0	14.0	15.0	10.0	10000.0
7	8	Control	5	4	1.0	NaN	NaN	NaN	10.0	10000.0
4	5	Control	5	3	2.0	16.0	15.0	14.0	10.0	10000.0
1	2	Control	4	4	3.0	13.0	14.0	17.0	11.0	11000.0
12	13	Diet	6	3	2.0	12.0	11.0	14.0	11.0	11000.0
15	16	Diet	6	4	2.0	16.0	15.0	18.0	12.0	12000.0
29	30	DietEx	6	5	2.0	15.0	12.0	18.0	13.0	13000.0
17	18	Diet	5	5	4.0	13.0	11.0	18.0	14.0	14000.0
20	21	Diet	6	5	3.0	17.0	16.0	19.0	14.0	14000.0
23	24	Diet	7	4	3.0	16.0	14.0	18.0	14.0	14000.0
30	31	DietEx	6	6	3.0	15.0	13.0	18.0	15.0	15000.0
5	6	Control	6	5	4.0	17.0	18.0	18.0	15.0	15000.0
6	7	Control	6	5	4.0	17.0	16.0	19.0	15.0	15000.0
33	34	DietEx	8	6	1.0	17.0	17.0	17.0	15.0	15000.0
14	15	Diet	7	6	3.0	17.0	11.0	18.0	16.0	16000.0
31	32	DietEx	9	5	2.0	16.0	14.0	17.0	16.0	16000.0
21	22	Diet	7	6	4.0	19.0	19.0	19.0	17.0	17000.0
24	25	DietEx	7	7	4.0	15.0	11.0	19.0	18.0	18000.0
26	27	DietEx	9	7	3.0	13.0	12.0	17.0	19.0	19000.0
32	33	DietEx	7	9	4.0	16.0	16.0	19.0	20.0	20000.0
25	26	DietEx	4	7	NaN	16.0	12.0	18.0	NaN	NaN

Сортировка может происходить по нескольким столбцам сразу. Например, давайте сделаем так, чтобы вначале шли люди, которые меньше всего сбросили килограммов за три месяца и чья самооценка в последний месяц была небольшой:

In [5]:

```
df.sort_values(['total', 'se3'])
```

Out[5]:

	id	group	w1	w2	w3	se1	se2	se3	total	total_gr
3	4	Control	3	2	1.0	11.0	11.0	12.0	6.0	6000.0
16	17	Diet	3	2	1.0	16.0	17.0	15.0	6.0	6000.0
19	20	Diet	4	2	1.0	12.0	11.0	11.0	7.0	7000.0
10	11	Control	4	2	2.0	16.0	16.0	11.0	8.0	8000.0
9	10	Control	3	3	2.0	14.0	15.0	13.0	8.0	8000.0
18	19	Diet	4	3	1.0	12.0	11.0	14.0	8.0	8000.0
2	3	Control	4	3	1.0	17.0	12.0	16.0	8.0	8000.0
11	12	Control	5	2	1.0	15.0	13.0	16.0	8.0	8000.0
27	28	DietEx	3	4	1.0	16.0	13.0	NaN	8.0	8000.0
22	23	Diet	4	3	2.0	15.0	15.0	15.0	9.0	9000.0
28	29	DietEx	3	5	1.0	13.0	13.0	16.0	9.0	9000.0
4	5	Control	5	3	2.0	16.0	15.0	14.0	10.0	10000.0
0	1	Control	4	3	3.0	14.0	13.0	15.0	10.0	10000.0
8	9	Control	5	4	1.0	14.0	14.0	15.0	10.0	10000.0
13	14	Diet	5	4	1.0	13.0	14.0	15.0	10.0	10000.0
7	8	Control	5	4	1.0	NaN	NaN	NaN	10.0	10000.0
12	13	Diet	6	3	2.0	12.0	11.0	14.0	11.0	11000.0
1	2	Control	4	4	3.0	13.0	14.0	17.0	11.0	11000.0
15	16	Diet	6	4	2.0	16.0	15.0	18.0	12.0	12000.0
29	30	DietEx	6	5	2.0	15.0	12.0	18.0	13.0	13000.0
17	18	Diet	5	5	4.0	13.0	11.0	18.0	14.0	14000.0
23	24	Diet	7	4	3.0	16.0	14.0	18.0	14.0	14000.0
20	21	Diet	6	5	3.0	17.0	16.0	19.0	14.0	14000.0
33	34	DietEx	8	6	1.0	17.0	17.0	17.0	15.0	15000.0
5	6	Control	6	5	4.0	17.0	18.0	18.0	15.0	15000.0
30	31	DietEx	6	6	3.0	15.0	13.0	18.0	15.0	15000.0
6	7	Control	6	5	4.0	17.0	16.0	19.0	15.0	15000.0
31	32	DietEx	9	5	2.0	16.0	14.0	17.0	16.0	16000.0
14	15	Diet	7	6	3.0	17.0	11.0	18.0	16.0	16000.0
21	22	Diet	7	6	4.0	19.0	19.0	19.0	17.0	17000.0
24	25	DietEx	7	7	4.0	15.0	11.0	19.0	18.0	18000.0
26	27	DietEx	9	7	3.0	13.0	12.0	17.0	19.0	19000.0
32	33	DietEx	7	9	4.0	16.0	16.0	19.0	20.0	20000.0
25	26	DietEx	4	7	NaN	16.0	12.0	18.0	NaN	NaN

По умолчанию сортировка происходит по возрастанию, но это можно поправить:

In [6]:

```
df.sort_values(['total', 'se3'], ascending = False)
```

Out[6]:

	id	group	w1	w2	w3	se1	se2	se3	total	total_gr
32	33	DietEx	7	9	4.0	16.0	16.0	19.0	20.0	20000.0
26	27	DietEx	9	7	3.0	13.0	12.0	17.0	19.0	19000.0
24	25	DietEx	7	7	4.0	15.0	11.0	19.0	18.0	18000.0
21	22	Diet	7	6	4.0	19.0	19.0	19.0	17.0	17000.0
14	15	Diet	7	6	3.0	17.0	11.0	18.0	16.0	16000.0
31	32	DietEx	9	5	2.0	16.0	14.0	17.0	16.0	16000.0
6	7	Control	6	5	4.0	17.0	16.0	19.0	15.0	15000.0
5	6	Control	6	5	4.0	17.0	18.0	18.0	15.0	15000.0
30	31	DietEx	6	6	3.0	15.0	13.0	18.0	15.0	15000.0
33	34	DietEx	8	6	1.0	17.0	17.0	17.0	15.0	15000.0
20	21	Diet	6	5	3.0	17.0	16.0	19.0	14.0	14000.0
17	18	Diet	5	5	4.0	13.0	11.0	18.0	14.0	14000.0
23	24	Diet	7	4	3.0	16.0	14.0	18.0	14.0	14000.0
29	30	DietEx	6	5	2.0	15.0	12.0	18.0	13.0	13000.0
15	16	Diet	6	4	2.0	16.0	15.0	18.0	12.0	12000.0
1	2	Control	4	4	3.0	13.0	14.0	17.0	11.0	11000.0
12	13	Diet	6	3	2.0	12.0	11.0	14.0	11.0	11000.0
0	1	Control	4	3	3.0	14.0	13.0	15.0	10.0	10000.0
8	9	Control	5	4	1.0	14.0	14.0	15.0	10.0	10000.0
13	14	Diet	5	4	1.0	13.0	14.0	15.0	10.0	10000.0
4	5	Control	5	3	2.0	16.0	15.0	14.0	10.0	10000.0
7	8	Control	5	4	1.0	NaN	NaN	NaN	10.0	10000.0
28	29	DietEx	3	5	1.0	13.0	13.0	16.0	9.0	9000.0
22	23	Diet	4	3	2.0	15.0	15.0	15.0	9.0	9000.0
2	3	Control	4	3	1.0	17.0	12.0	16.0	8.0	8000.0
11	12	Control	5	2	1.0	15.0	13.0	16.0	8.0	8000.0
18	19	Diet	4	3	1.0	12.0	11.0	14.0	8.0	8000.0
9	10	Control	3	3	2.0	14.0	15.0	13.0	8.0	8000.0
10	11	Control	4	2	2.0	16.0	16.0	11.0	8.0	8000.0
27	28	DietEx	3	4	1.0	16.0	13.0	NaN	8.0	8000.0
19	20	Diet	4	2	1.0	12.0	11.0	11.0	7.0	7000.0
16	17	Diet	3	2	1.0	16.0	17.0	15.0	6.0	6000.0
3	4	Control	3	2	1.0	11.0	11.0	12.0	6.0	6000.0
25	26	DietEx	4	7	NaN	16.0	12.0	18.0	NaN	NaN

По умолчаю изменения исходного датафрейма не происходит, но это можно исправить, добавив опцию `inplace=True`. Тогда строки в исходном датасете поменяют своё расположение в соответствии с выбранной сортировкой.

Работа с пропущенными значениями (NaN)

Иногда при работе с данными можно столкнуться с такой проблемой: в данных есть пропущенные значения, но удалять из таблицы строки с пропущенными значениями не хочется, поскольку в этих строках есть также заполненные ячейки с ценной информацией.

Особенно это актуально при работе с большими таблицами, где очень много показателей, но при этом значительная доля этих показателей специфическая, например, рассчитывается не для каждого года и страны, что может привести к ситуации, когда в каждой строке таблицы есть хотя бы одна ячейка с пропущенным значением. Очевидно, что в таком случае при обычном удалении строк с пропущенными значениями мы останемся с пустой таблицей! Поэтому часто в таких ситуациях пропущенные значения не удаляют, а заполняют.

Если данные не содержат нулевых значений, то пропущенные значения можно заменить нулями. Для этого достаточно применить к датафрейму pandas метод `.fillna()` :

In [3]:
```
import pandas as pd
import numpy as np

# создадим маленький датафрейм
ages = pd.DataFrame({'age':[24, 25, np.nan, 29],
                     'income':[20000, np.nan, 26000, 30000],
                     'children':[2, 1, 3, np.nan]})
ages
```

Out[3]:

	age	income	children
0	24.0	20000.0	2.0
1	25.0	NaN	1.0
2	NaN	26000.0	3.0
3	29.0	30000.0	NaN

In [4]:
```
# заполним всё нулями
ages.fillna(0)
```

Out[4]:

	age	income	children
0	24.0	20000.0	2.0
1	25.0	0.0	1.0
2	0.0	26000.0	3.0
3	29.0	30000.0	0.0

При работе с опросами часто пропущенные значения заменяют специальными кодами. Часто эти коды сильно отличаются от «обычных» значений. Например, ответы на вопрос о некотором утверждении имеют метки:

1 – абсолютно не согласен с утверждением, 2 – не согласен с утверждением, 3 – затрудняюсь ответить, 4 – согласен с утверждением, 5 – абсолютно согласен с утверждением.

Тогда в случае отсутствия ответа на вопрос можно поставить значение 98 или 99, потому что встретить такой ответ при обычном сценарии развития событий не получится, сам вопрос не позволит.

Однако иногда недостаточно заполнить пустые ячейки одним значением. Ведь когда мы заполняем их одним значением, мы все равно теряем информацию – нулевые значения или закодированные особым образом мы потом отфильтруем и не будем использовать в дальнейшем анализе. Если не хочется действовать так радикально, пропущенные значения можно заполнить средним или медианным значением по столбцу. При добавлении значения, равного среднему или медиане, распределение данных, а также среднее и медианное значения практически не меняются. Если разница между средним и медианой небольшая, распределение не является сильно скошенным вправо или влево, нет нехарактерных значений – выбросов, то нет большой разницы, каким значением – средним или медианным – заполнять. Если разница довольно большая, то лучше заполнять медианным значением, поскольку в таком случае оно более адекватно отражает середину распределения.

Для заполнения средним значением или медианой тоже подойдет метод `.fillna()` .

In [9]:
```
# вспомним, как считается среднее по столбцам
ages.mean()
```

Out[9]:
```
pandas.core.series.Series
```

In [8]:
```
# подставим .mean()
# могли бы взять .median()
ages.fillna(ages.mean())
```

Out[8]:

	age	income	children
0	24.0	20000.000000	2.0
1	25.0	25333.333333	1.0
2	26.0	26000.000000	3.0
3	29.0	30000.000000	2.0

Если почитать документацию, то можно заметить, что метод `.fillna()` принимает на вход словарь, последовательность pandas *Series* или датафрейм pandas. В примере выше у нас был *Series*, но можно было бы поставить другие значения и оформить их в виде словаря:

In [10]:
```
# заполним столбец age средним
# столбец income – медианой
# столбец children – нулями

ages.fillna({'age': ages['age'].mean(),
             'income': ages['income'].median(),
             'children': 0})
```

Out[10]:

	age	income	children
0	24.0	20000.0	2.0
1	25.0	26000.0	1.0
2	26.0	26000.0	3.0
3	29.0	30000.0	0.0

Если содержательно данные позволяют заполнить пропущенную ячейку значением из того же столбца ячейкой выше (например, нет значения дохода за текущий год, но его можно заполнить значением дохода за прошлый год), то можно воспользоваться методом `ffill` (от *forward fill* – заполнение вперед) и указать специальный аргумент `method` :

In [11]:
```
# до заполнения
ages
```

Out[11]:

	age	income	children
0	24.0	20000.0	2.0
1	25.0	NaN	1.0
2	NaN	26000.0	3.0
3	29.0	30000.0	NaN

In [12]:
```
# вместо NaN в строке 2 стоит значение из строки 1 (age)
# вместо NaN в строке 1 стоит значение из строки 0 (income)
# вместо NaN в строке 3 стоит значение из строки 2 (children)

ages.fillna(method='ffill')
```

Out[12]:

	age	income	children
0	24.0	20000.0	2.0
1	25.0	20000.0	1.0
2	25.0	26000.0	3.0
3	29.0	30000.0	3.0

Аналогичным способом можно заполнить пропущенную ячейку значением из того же столбца ячейкой ниже – метод `bfill` (от *backward fill* – заполнение назад):

In [13]:
```
ages.fillna(method='bfill')
```

Out[13]:

	age	income	children
0	24.0	20000.0	2.0
1	25.0	26000.0	1.0
2	29.0	26000.0	3.0
3	29.0	30000.0	NaN

В примере выше в столбце *children* в последней строке остался `NaN`, так как ниже строки нет, и брать значение для заполнения неоткуда.

Больше информации можно найти в документации (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.fillna.html) по методу `.fillna()`.

Если хочется узнать про более продвинутые способы заполнения пропущенных значений (для понимания требуются знания статистики и моделирования), можно почитать (https://www.theanalysisfactor.com/multiple-imputation-in-a-nutshell/) про множественную импутацию и ее реализацию (https://scikit-learn.org/stable/modules/impute.html) в Python.

Урок 6.4. Работа с NaN-ами

In [1]:

```
import pandas as pd
import numpy as np
df = pd.read_csv("WeightLoss.csv")
df['total'] = df['w1'] + df['w2'] + df['w3']
df['total_gr'] = df['total'] * 1000
```

Вызывая сводную информацию по таблице через `.info()`, мы видели, что в датасете присутствуют пропущенные значения (`NaN` от *Not a Number*):

In [3]:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 34 entries, 0 to 33
Data columns (total 10 columns):
id          34 non-null int64
group       34 non-null object
w1          34 non-null int64
w2          34 non-null int64
w3          33 non-null float64
se1         33 non-null float64
se2         33 non-null float64
se3         32 non-null float64
total       33 non-null float64
total_gr    33 non-null float64
dtypes: float64(6), int64(3), object(1)
memory usage: 2.7+ KB
```

Конечно, считать количество незаполненных ячеек по каждому столбцу самостоятельно, неудобно. Давайте посчитаем это автоматически:

In [4]:

```
df.isnull().sum()
```

Out[4]:

```
id          0
group       0
w1          0
w2          0
w3          1
se1         1
se2         1
se3         2
total       1
total_gr    1
dtype: int64
```

Ещё пропущенные значения можно заполнять – заменить `NaN` на 0 или любое другое значение или на среднее (медианное значение по столбцу).

In [5]:

```
df.fillna(0)
```

Out[5]:

	id	group	w1	w2	w3	se1	se2	se3	total	total_gr
0	1	Control	4	3	3.0	14.0	13.0	15.0	10.0	10000.0
1	2	Control	4	4	3.0	13.0	14.0	17.0	11.0	11000.0
2	3	Control	4	3	1.0	17.0	12.0	16.0	8.0	8000.0
3	4	Control	3	2	1.0	11.0	11.0	12.0	6.0	6000.0
4	5	Control	5	3	2.0	16.0	15.0	14.0	10.0	10000.0
5	6	Control	6	5	4.0	17.0	18.0	18.0	15.0	15000.0
6	7	Control	6	5	4.0	17.0	16.0	19.0	15.0	15000.0
7	8	Control	5	4	1.0	0.0	0.0	0.0	10.0	10000.0
8	9	Control	5	4	1.0	14.0	14.0	15.0	10.0	10000.0
9	10	Control	3	3	2.0	14.0	15.0	13.0	8.0	8000.0
10	11	Control	4	2	2.0	16.0	16.0	11.0	8.0	8000.0
11	12	Control	5	2	1.0	15.0	13.0	16.0	8.0	8000.0
12	13	Diet	6	3	2.0	12.0	11.0	14.0	11.0	11000.0
13	14	Diet	5	4	1.0	13.0	14.0	15.0	10.0	10000.0
14	15	Diet	7	6	3.0	17.0	11.0	18.0	16.0	16000.0
15	16	Diet	6	4	2.0	16.0	15.0	18.0	12.0	12000.0
16	17	Diet	3	2	1.0	16.0	17.0	15.0	6.0	6000.0
17	18	Diet	5	5	4.0	13.0	11.0	18.0	14.0	14000.0
18	19	Diet	4	3	1.0	12.0	11.0	14.0	8.0	8000.0
19	20	Diet	4	2	1.0	12.0	11.0	11.0	7.0	7000.0
20	21	Diet	6	5	3.0	17.0	16.0	19.0	14.0	14000.0
21	22	Diet	7	6	4.0	19.0	19.0	19.0	17.0	17000.0
22	23	Diet	4	3	2.0	15.0	15.0	15.0	9.0	9000.0
23	24	Diet	7	4	3.0	16.0	14.0	18.0	14.0	14000.0
24	25	DietEx	7	7	4.0	15.0	11.0	19.0	18.0	18000.0
25	26	DietEx	4	7	0.0	16.0	12.0	18.0	0.0	0.0
26	27	DietEx	9	7	3.0	13.0	12.0	17.0	19.0	19000.0
27	28	DietEx	3	4	1.0	16.0	13.0	0.0	8.0	8000.0
28	29	DietEx	3	5	1.0	13.0	13.0	16.0	9.0	9000.0
29	30	DietEx	6	5	2.0	15.0	12.0	18.0	13.0	13000.0
30	31	DietEx	6	6	3.0	15.0	13.0	18.0	15.0	15000.0
31	32	DietEx	9	5	2.0	16.0	14.0	17.0	16.0	16000.0
32	33	DietEx	7	9	4.0	16.0	16.0	19.0	20.0	20000.0
33	34	DietEx	8	6	1.0	17.0	17.0	17.0	15.0	15000.0

Если данных у нас много, и при этом потеря строк с пропущенными значениями нас не смущает, такие строки можно просто удалить:

In [6]:

```
df = df.dropna()
```

Урок 6.5. Иерархическое индексирование

Создадим небольшой датасет, который содержит число переводов с английского на русский и с французского на русский для одного переводчика за каждый день:

In [2]:

```
import pandas as pd
df = pd.DataFrame([['2019-03-11', 'en', 3],
                   ['2019-03-11', 'fr', 5],
                   ['2019-03-12', 'en', 6],
                   ['2019-03-13', 'fr', 1],
                   ['2019-03-13', 'en', 2],
                   ['2019-03-16', 'fr', 4],
                   ['2019-03-17', 'en', 3]],
                   columns = ['date', 'lang', 'n'])
```

In [3]:

```
df
```

Out[3]:

	date	lang	n
0	2019-03-11	en	3
1	2019-03-11	fr	5
2	2019-03-12	en	6
3	2019-03-13	fr	1
4	2019-03-13	en	2
5	2019-03-16	fr	4
6	2019-03-17	en	3

Ранее мы обсуждали, что при необходимости строки в датафрейме можно называть по какому-нибудь столбцу. Например, в качестве идентификатора строки вместо её номера использовать имя респондента или название страны. Однако в таком случае мы можем столкнуться с проблемой: идентификатор строки должен быть уникальным, то есть столбец, который мы используем в качестве названий, не должен содержать повторяющихся значений.

Вернемся к нашему датасету: если мы захотим использовать в качестве идентификатора строки столбцы `date` или `lang`, ничего не получится: даты повторяются и языки тоже. Как быть? Использовать мульти-индекс – индекс строки, который будет состоять из пары значений (`date - lang`). Такая пара уже будет уникальной:

In [4]:

```
df.set_index(['date', 'lang'], inplace=True)
```

In [5]:

```
df
```

Out[5]:

		n
date	lang	
2019-03-11	en	3
	fr	5
2019-03-12	en	6
2019-03-13	fr	1
	en	2
2019-03-16	fr	4
2019-03-17	en	3

Посмотрим на то, как теперь выглят объект `Index` :

In [6]:

```
df.index
```

Out[6]:

```
MultiIndex(levels=[['2019-03-11', '2019-03-12', '2019-03-13', '2019-03-16',
'2019-03-17'], ['en', 'fr']],
           labels=[[0, 0, 1, 2, 2, 3, 4], [0, 1, 0, 1, 0, 1, 0]],
           names=['date', 'lang'])
```

Почему такое индексирование называется иерархическим? Потому что теперь, при обращении к строке, нам нужно будет указывать двойной индекс: например, дату и язык, то есть внутри даты будут вложены языки – образуется некоторая иерархия, сложная структура:

In [7]:

```
df.loc['2019-03-11'] # только фиксированная дата
```

Out[7]:

	n
lang	
en	3
fr	5

In [8]:

```
df.loc[('2019-03-11', 'en')]  # фиксированная дата и язык
```

Out[8]:

```
n    3
Name: (2019-03-11, en), dtype: int64
```

По таким индексам также удобно сортировать строки:

In [9]:

```
df.sort_index()  # сортировка по дате и языку (язык - по алфавиту)
```

Out[9]:

		n
date	**lang**	
2019-03-11	**en**	3
	fr	5
2019-03-12	**en**	6
2019-03-13	**en**	2
	fr	1
2019-03-16	**fr**	4
2019-03-17	**en**	3

Работа с пропущенными значениями (NaN)

Иногда при работе с данными можно столкнуться с такой проблемой: в данных есть пропущенные значения, но удалять из таблицы строки с пропущенными значениями не хочется, поскольку в этих строках есть также заполненные ячейки с ценной информацией.

Особенно это актуально при работе с большими таблицами, где очень много показателей, но при этом значительная доля этих показателей специфическая, например, рассчитывается не для каждого года и страны, что может привести к ситуации, когда в каждой строке таблицы есть хотя бы одна ячейка с пропущенным значением. Очевидно, что в таком случае при обычном удалении строк с пропущенными значениями мы останемся с пустой таблицей! Поэтому часто в таких ситуациях пропущенные значения не удаляют, а заполняют.

Если данные не содержат нулевых значений, то пропущенные значения можно заменить нулями. Для этого достаточно применить к датафрейму pandas метод `.fillna()` :

In [3]:

```
import pandas as pd
import numpy as np

# создадим маленький датафрейм
ages = pd.DataFrame({'age':[24, 25, np.nan, 29],
                     'income':[20000, np.nan, 26000, 30000],
                     'children':[2, 1, 3, np.nan]})
ages
```

Out[3]:

	age	income	children
0	24.0	20000.0	2.0
1	25.0	NaN	1.0
2	NaN	26000.0	3.0
3	29.0	30000.0	NaN

In [4]:

```
# заполним всё нулями
ages.fillna(0)
```

Out[4]:

	age	income	children
0	24.0	20000.0	2.0
1	25.0	0.0	1.0
2	0.0	26000.0	3.0
3	29.0	30000.0	0.0

При работе с опросами часто пропущенные значения заменяют специальными кодами. Часто эти коды сильно отличаются от «обычных» значений. Например, ответы на вопрос о некотором утверждении имеют метки:

1 – абсолютно не согласен с утверждением, 2 – не согласен с утверждением, 3 – затрудняюсь ответить, 4 – согласен с утверждением, 5 – абсолютно согласен с утверждением.

Тогда в случае отсутствия ответа на вопрос можно поставить значение 98 или 99, потому что встретить такой ответ при обычном сценарии развития событий не получится, сам вопрос не позволит.

Однако иногда недостаточно заполнить пустые ячейки одним значением. Ведь когда мы заполняем их одним значением, мы все равно теряем информацию – нулевые значения или закодированные особым образом мы потом отфильтруем и не будем использовать в дальнейшем анализе. Если не хочется действовать так радикально, пропущенные значения можно заполнить средним или медианным значением по столбцу. При добавлении значения, равного среднему или медиане, распределение данных, а также среднее и медианное значения практически не меняются. Если разница между средним и медианой небольшая, распределение не является сильно скошенным вправо или влево, нет нехарактерных значений – выбросов, то нет большой разницы, каким значением – средним или медианным – заполнять. Если разница довольно большая, то лучше заполнять медианным значением, поскольку в таком случае оно более адекватно отражает середину распределения.

Для заполнения средним значением или медианой тоже подойдет метод `.fillna()` .

In [9]:

```
# вспомним, как считается среднее по столбцам
ages.mean()
```

Out[9]:

```
pandas.core.series.Series
```

In [8]:

```
# подставим .mean()
# могли бы взять .median()
ages.fillna(ages.mean())
```

Out[8]:

	age	income	children
0	24.0	20000.000000	2.0
1	25.0	25333.333333	1.0
2	26.0	26000.000000	3.0
3	29.0	30000.000000	2.0

Если почитать документацию, то можно заметить, что метод `.fillna()` принимает на вход словарь, последовательность pandas *Series* или датафрейм pandas. В примере выше у нас был *Series*, но можно было бы поставить другие значения и оформить их в виде словаря:

In [10]:

```
# заполним столбец age средним
# столбец income – медианой
# столбец children – нулями

ages.fillna({'age': ages['age'].mean(),
             'income': ages['income'].median(),
             'children': 0})
```

Out[10]:

	age	income	children
0	24.0	20000.0	2.0
1	25.0	26000.0	1.0
2	26.0	26000.0	3.0
3	29.0	30000.0	0.0

Если содержательно данные позволяют заполнить пропущенную ячейку значением из того же столбца ячейкой выше (например, нет значения дохода за текущий год, но его можно заполнить значением дохода за прошлый год), то можно воспользоваться методом `ffill` (от *forward fill* – заполнение вперед) и указать специальный аргумент `method` :

In [11]:

```
# до заполнения
ages
```

Out[11]:

	age	income	children
0	24.0	20000.0	2.0
1	25.0	NaN	1.0
2	NaN	26000.0	3.0
3	29.0	30000.0	NaN

In [12]:

```
# вместо NaN в строке 2 стоит значение из строки 1 (age)
# вместо NaN в строке 1 стоит значение из строки 0 (income)
# вместо NaN в строке 3 стоит значение из строки 2 (children)

ages.fillna(method='ffill')
```

Out[12]:

	age	income	children
0	24.0	20000.0	2.0
1	25.0	20000.0	1.0
2	25.0	26000.0	3.0
3	29.0	30000.0	3.0

Аналогичным способом можно заполнить пропущенную ячейку значением из того же столбца ячейкой ниже – метод `bfill` (от *backward fill* – заполнение назад):

In [13]:

```
ages.fillna(method='bfill')
```

Out[13]:

	age	income	children
0	24.0	20000.0	2.0
1	25.0	26000.0	1.0
2	29.0	26000.0	3.0
3	29.0	30000.0	NaN

В примере выше в столбце *children* в последней строке остался `NaN`, так как ниже строки нет, и брать значение для заполнения неоткуда.

Больше информации можно найти в документации (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.fillna.html) по методу `.fillna()`.

Если хочется узнать про более продвинутые способы заполнения пропущенных значений (для понимания требуются знания статистики и моделирования), можно почитать (https://www.theanalysisfactor.com/multiple-imputation-in-a-nutshell/) про множественную импутацию и ее реализацию (https://scikit-learn.org/stable/modules/impute.html) в Python.