Машинное обучение

Преобразование датафреймов

Материалы > Анализ и обработка данных

Продолжим работу по изучению библиотеки Pandas. Сегодня мы поговорим про возможности изменения и соединения датафреймов, а также про способы группировки данных.

> Хотя сами по себе эти навыки являются скорее вспомогательными, овладеть ими совершенно необходимо, если вы хотите быстро и эффективно строить модели ML.

Откроем блокнот к этому занятию

Изменение датафрейма

Вернемся к тому датафрейму, с которым мы работали на прошлом занятии.

```
# создадим несколько списков и массивов Numpy с информацией о семи странах мира
country = np.array(['China', 'Vietnam', 'United Kingdom', 'Russia', 'Argentina', 'Boli
capital = ['Beijing', 'Hanoi', 'London', 'Moscow', 'Buenos Aires', 'Sucre', 'Pretoria'
population = [1400, 97, 67, 144, 45, 12, 59] # млн. человек
area = [9.6, 0.3, 0.2, 17.1, 2.8, 1.1, 1.2] # млн. кв. км.
sea = [1] * 5 + [0, 1] # выход к морю (в этом списке его нет только у Боливии)

# кроме того создадим список кодов стран, которые станут индексом датафрейма
custom_index = ['CN', 'VN', 'GB', 'RU', 'AR', 'BO', 'ZA']

# создадим пустой словарь
countries_dict = {}

# превратим эти списки в значения словаря,
# одновременно снабдив необходимыми ключами
countries_dict['country'] = country
countries_dict['capital'] = capital
countries_dict['population'] = population
countries_dict['area'] = area
countries_dict['sea'] = sea

# создадим датафрейм
countries = pd.DataFrame(countries_dict, index = custom_index)
countries
```

	country	capital	population	area	sea
CN	China	Beijing	1400	9.6	1
VN	Vietnam	Hanoi	97	0.3	1
GB	United Kingdom	London	67	0.2	1

RU	Russia	Moscow	144	17.1	1
AR	Argentina	Buenos Aires	45	2.8	1
BO	Bolivia	Sucre	12	1.1	0
ZA	South Africa	Pretoria	59	1.2	1

Посмотрим, как мы можем преобразовать этот датафрейм.

Копирование датафрейма

Метод .copy()

В первую очередь поговорим про важную особенность при копировании датафрейма. Вначале создадим копию датафрейма с помощью простого присвоения этого объекта новой переменной.

```
1 countries_new = countries
```

Теперь удалим строку с данными про Аргентину, а после этого выведем исходный датафрейм.

```
1 countries_new.drop(labels = 'AR', axis = 0, inplace = True)
2 countries
```

	country	capital	population	area	sea
CN	China	Beijing	1400	9.6	1
VN	Vietnam	Hanoi	97	0.3	1
GB	United Kingdom	London	67	0.2	1
RU	Russia	Moscow	144	17.1	1
BO	Bolivia	Sucre	12	1.1	0
ZA	South Africa	Pretoria	59	1.2	1

Как вы видите, изменения коснулись и его. По этой причине для создания полноценной копии лучше использовать **метод .copy()**.

```
1  # в первую очередь вернем Аргентину в исходный датафрейм countries
2  countries = pd.DataFrame(countries_dict, index = custom_index)
3
4  # создадим копию, на этот раз с помощью метода .copy()
5  countries_new = countries.copy()
6
7  # вновь удалим запись про Аргентину
8  countries_new.drop(labels = 'AR', axis = 0, inplace = True)
9
10 # выведем исходный датафрейм
11 countries
```

	country	capital	population	area	sea
CN	China	Beijing	1400	9.6	1
VN	Vietnam	Hanoi	97	0.3	1
GB	United Kingdom	London	67	0.2	1

RU	Russia	Moscow	144	17.1	1
AR	Argentina	Buenos Aires	45	2.8	1
BO	Bolivia	Sucre	12	1.1	0
ZA	South Africa	Pretoria	59	1.2	1

Такой тип копирования (без изменения исходного датафрейма) может быть полезен, например, когда мы хотим взять датафрейм, протестировать какие-то гипотезы, построить одну или несколько моделей, но при этом не трогать исходные данные.

Про параметр inplace

Создадим несложный датафрейм из вложенных списков.

```
df = pd.DataFrame([[1, 1, 1],
                   [2, 2, 2],
                   [3, 3, 3]],
                   columns = ['A', 'B', 'C'])

df
```

	A	B	C
0	1	1	1
1	2	2	2
2	3	3	3

Как понять, сохраняется ли изменение после применение определенного метода или нет? Если метод выдает датафрейм, изменение не сохраняется.

```
# попробуем удалить столбец A
df.drop(labels = ['A'], axis = 1)
```

	B	C
0	1	1
1	2	2
2	3	3

```
# проверим, сохранилось ли изменение
df
```

	A	B	C
0	1	1	1
1	2	2	2
2	3	3	3

При этом если метод выдает None, изменение постоянно.

```
# изменим параметр inplace на True
print(df.drop(labels = ['A'], axis = 1, inplace = True))
```

```
None
```

```
# проверим
df
```

	B	C
0	1	1
1	2	2
2	3	3

По этой причине нельзя использовать inplace = True и записывать результат в переменную одновременно.

```
df = df.drop(labels = ['B'], axis = 1, inplace = True)
print(df)
```

```
None
```

В этом случае мы записываем None в переменную df.

Столбцы датафрейма

Именование столбцов при создании датафрейма

Создадим список с названиями столбцов на кириллице и транспонированный массив Numpy с данными о странах.

```
custom_columns = ['страна', 'столица', 'население', 'площадь', 'море']
arr = np.array([country, capital, population, area, sea]).T
arr
```

```
array([['China', 'Beijing', '1400', '9.6', '1'],
       ['Vietnam', 'Hanoi', '97', '0.3', '1'],
       ['United Kingdom', 'London', '67', '0.2', '1'],
       ['Russia', 'Moscow', '144', '17.1', '1'],
       ['Argentina', 'Buenos Aires', '45', '2.8', '1'],
       ['Bolivia', 'Sucre', '12', '1.1', '0'],
       ['South Africa', 'Pretoria', '59', '1.2', '1']], dtype='<U32')
```

После этого создадим датафрейм с помощью функции **pd.DataFrame()** с параметром columns, в который передадим названия столбцов.

```
countries = pd.DataFrame(data = arr,
                         index = custom_index,
                         columns = custom_columns)

countries
```

	страна	столица	население	площадь	море
CN	China	Beijing	1400	9.6	1
VN	Vietnam	Hanoi	97	0.3	1
GB	United Kingdom	London	67	0.2	1
RU	Russia	Moscow	144	17.1	1
AR	Argentina	Buenos Aires	45	2.8	1

BO	Bolivia	Sucre	12	1.1	0
ZA	South Africa	Pretoria	59	1.2	1

Вернем прежние названия столбцов.

```
countries.columns = ['country', 'capital', 'population', 'area', 'sea']
```

Переименование столбцов

Для того чтобы переименовать отдельные столбцы, можно воспользоваться **методом .rename()**. Внутри этого метода в параметр columns мы передаем словарь, где ключами будут текущие названия столбцов, а значениями соответствующие им новые названия.

```
# переименуем столбец capital на city
countries.rename(columns = {'capital': 'city'}, inplace = True)
countries
```

	country	city	population	area	sea
CN	China	Beijing	1400	9.6	1
VN	Vietnam	Hanoi	97	0.3	1
GB	United Kingdom	London	67	0.2	1
RU	Russia	Moscow	144	17.1	1
AR	Argentina	Buenos Aires	45	2.8	1
BO	Bolivia	Sucre	12	1.1	0
ZA	South Africa	Pretoria	59	1.2	1

Тип данных в столбце

Все значения одного столбца датафрейма всегда имеют один и тот же тип данных. Посмотреть тип данных каждого из столбцов можно с помощью **атрубута .dtypes**.

```
countries.dtypes
```

```
country       object
city          object
population    object
area          object
sea           object
dtype: object
```

Изменение типа данных

Преобразовать тип данных столбца можно с помощью **метода .astype()**. Этот метод можно применить к конкретному столбцу.

```
# преобразуем тип данных столбца population в int
countries.population = countries.population.astype('int')
```

Кроме того, мы можем пробразовать тип данных сразу в нескольких столбцах. Для этого применим метод **.astype()** ко всему датафрейму. Самому методу мы передадим словарь, где ключами будут названия столбцов, а значениями — соответсвующий им желаемый тип данных.

```
# изменим тип данных в столбцах area и sea
countries = countries.astype({'area': 'float', 'sea' : 'category'})
```

Посмотрим на результат.

```
countries.dtypes
```

```
country         object
city            object
population       int64
area           float64
sea           category
dtype: object
```

Тип данных category

Обратите внимание на новый для нас тип данных category. Во многом он похож на факторную переменную в R.

```
# в category содержится информация об имеющихся в столбце категориях
countries.sea
```

```
CN    1
VN    1
GB    1
RU    1
AR    1
BO    0
ZA    1
Name: sea, dtype: category
Categories (2, object): ['0', '1']
```

Тип category мы рассмотрим более подробно на занятии по кодированию категориальных переменных.

Помимо упомянутых типов данных, нам также знаком объект datetime, который используется для работы с временными рядами. Мы снова обратимся к нему на занятии по очистке данных.

Фильтр столбцов по типу данных

Выбрать столбцы в зависимости от типа содержащихся в них данных можно с помощью **метода .select_dtypes()**. Включить определенные типы данных можно с помощью параметра *include*.

```
# выберем только типы данных int и float
countries.select_dtypes(include = ['int64', 'float64'])
```

	population	area
CN	1400	9.6
VN	97	0.3

GB	67	0.2
RU	144	17.1
AR	45	2.8
BO	12	1.1
ZA	59	1.2

Исключить определенные типы данных можно через *exclude*.

```
# выберем все типы данных, кроме object и category
countries.select_dtypes(exclude = ['object', 'category'])
```

	population	area
CN	1400	9.6
VN	97	0.3
GB	67	0.2
RU	144	17.1
AR	45	2.8
BO	12	1.1
ZA	59	1.2

Добавление строк и столбцов

Добавление строк

Метод ._append() + словарь

Для добавления строк в первую очередь используется **метод .append()**. С его помощью строку можно добавить из питоновского словаря.

```
# создадим словарь с данными Канады и добавим его в датафрейм
dict_ = {'country': 'Canada', 'city': 'Ottawa', 'population': 38, 'area': 10, 'sea' : '

# словарь можно добавлять только если ignore_index = True
countries = countries._append(dict_, ignore_index = True)
countries
```

	country	city	population	area	sea
0	China	Beijing	1400	9.6	1
1	Vietnam	Hanoi	97	0.3	1
2	United Kingdom	London	67	0.2	1
3	Russia	Moscow	144	17.1	1
4	Argentina	Buenos Aires	45	2.8	1
5	Bolivia	Sucre	12	1.1	0
6	South Africa	Pretoria	59	1.2	1
7	Canada	Ottawa	38	10.0	1

Метод ._append() + Series

Мы также можем добавить строки в виде объекта Series.

```
# причем, если передать список из Series, можно добавить сразу несколько строк
list_of_series = [pd.Series(['Spain', 'Madrid', 47, 0.5, 1], index = countries.columns)
                  pd.Series(['Netherlands', 'Amsterdam', 17, 0.04, 1], index = countrie

# нам по-прежнему необходим параметр ignore_index = True
countries._append(list_of_series, ignore_index = True)
countries
```

	country	city	population	area	sea
0	China	Beijing	1400	9.60	1
1	Vietnam	Hanoi	97	0.30	1
2	United Kingdom	London	67	0.20	1
3	Russia	Moscow	144	17.10	1
4	Argentina	Buenos Aires	45	2.80	1
5	Bolivia	Sucre	12	1.10	0
6	South Africa	Pretoria	59	1.20	1
7	Canada	Ottawa	38	10.00	1
8	Spain	Madrid	47	0.50	1
9	Netherlands	Amsterdam	17	0.04	1

Метод ._append() + другой датафрейм

Новая строка может также содержаться в другом датафрейме.

```
# новая строка может также содержаться в другом датафрейме
# обратите внимание, что числовые значения мы помещаем в списки
peru = pd.DataFrame({'country' : 'Peru',
                     'city' : 'Lima',
                     'population': [33],
                     'area' : [1.3],
                     'sea' : [1]})
peru
```

	country	city	population	area	sea
0	Peru	Lima	33	1.3	1

```
# перед добавлением выберем первую строку с помощью метода .iloc[]
countries._append(peru.iloc[0], ignore_index = True)
```

	country	city	population	area	sea
0	China	Beijing	1400	9.6	1
1	Vietnam	Hanoi	97	0.3	1
2	United Kingdom	London	67	0.2	1
3	Russia	Moscow	144	17.1	1
4	Argentina	Buenos Aires	45	2.8	1

5	Bolivia	Sucre	12	1.1	0
6	South Africa	Pretoria	59	1.2	1
7	Canada	Ottawa	38	10.0	1
8	Peru	Lima	33	1.3	1

Использование .iloc[]

Если вновь вывести наш датафрейм countries, мы увидим, что данные об Испании, Нидерландах и Перу не сохранились.

```
# для этого нам надо было либо перезаписать результат метода ._append() в переменную cc
# либо использовать параметр inplace = True.
countries
```

	country	city	population	area	sea
0	China	Beijing	1400	9.6	1
1	Vietnam	Hanoi	97	0.3	1
2	United Kingdom	London	67	0.2	1
3	Russia	Moscow	144	17.1	1
4	Argentina	Buenos Aires	45	2.8	1
5	Bolivia	Sucre	12	1.1	0
6	South Africa	Pretoria	59	1.2	1
7	Canada	Ottawa	38	10.0	1

Добавим данные об этих странах на постоянной основе с помощью **метода .iloc[]** и посмотрим на результат.

```
countries.iloc[[5, 6, 7]] = [['Spain', 'Madrid', 47, 0.5, 1],
                             ['Netherlands', 'Amsterdam', 17, 0.04, 1],
                             ['Peru', 'Lima', 33, 1.3, 1]]

countries
```

	country	city	population	area	sea
0	China	Beijing	1400	9.60	1
1	Vietnam	Hanoi	97	0.30	1
2	United Kingdom	London	67	0.20	1
3	Russia	Moscow	144	17.10	1
4	Argentina	Buenos Aires	45	2.80	1
5	Spain	Madrid	47	0.50	1
6	Netherlands	Amsterdam	17	0.04	1
7	Peru	Lima	33	1.30	1

Обратите внимание, что строки добавились строго на те индексы, которые были указаны в

методе **.iloc[]** (т.е. 5, 6 и 7) и заменили данные, ранее находившиеся на этих индексах (Боливия, Южная Африка и Канада).

В версии Pandas 2.0.0, которая была опубликована 3 апреля 2023 года, метод **.append()** был удален, и применение нижнего подчеркивания, хотя и позволяет выполнить присоединение строк, не является удачным решением.

Разработчики рекомендуют использовать **метод .concat()**. О нем мы поговорим ниже.

Добавление столбцов

Объявление нового столбца

Новый столбец датафрейма можно просто объявить и сразу добавить в него необходимые данные.

```
# например, добавим данные о плотности населения
countries['pop_density'] = [153, 49, 281, 9, 17, 94, 508, 26]
countries
```

	country	city	population	area	sea	pop_density
0	China	Beijing	1400	9.60	1	153
1	Vietnam	Hanoi	97	0.30	1	49
2	United Kingdom	London	67	0.20	1	281
3	Russia	Moscow	144	17.10	1	9
4	Argentina	Buenos Aires	45	2.80	1	17
5	Spain	Madrid	47	0.50	1	94
6	Netherlands	Amsterdam	17	0.04	1	508
7	Peru	Lima	33	1.30	1	26

Метод .insert()

Добавим столбец с кодами стран с помощью **метода .insert()**.

```
countries.insert(loc = 1, # это будет второй по счету столбец
                 column = 'code', # название столбца
                 value = ['CN', 'VN', 'GB', 'RU', 'AR', 'ES', 'NL', 'PE']) # значения с
```

Обратите внимание, метод **.insert()** по умолчанию (без перезаписи в переменную или параметра inplace = True) сохраняет результат.

```
# посмотрим на результат
countries
```

	country	code	city	population	area	sea	pop_density
0	China	CN	Beijing	1400	9.60	1	153
1	Vietnam	VN	Hanoi	97	0.30	1	49
2	United Kingdom	GB	London	67	0.20	1	281
3	Russia	RU	Moscow	144	17.10	1	9
4	Argentina	AR	Buenos Aires	45	2.80	1	17
5	Spain	ES	Madrid	47	0.50	1	94
6	Netherlands	NL	Amsterdam	17	0.04	1	508
7	Peru	PE	Lima	33	1.30	1	26

Теперь рассмотрим несколько способов добавления столбца с рассчитанным значением.

Метод .assign()

Создадим столбец area_miles, в который поместим площадь в милях. Вначале используем **метод .assign()**.

```
countries = countries.assign(area_miles = countries.area / 2.59).round(2)
countries
```

	country	code	city	population	area	sea	pop_density	area_miles
0	China	CN	Beijing	1400	9.60	1	153	3.71
1	Vietnam	VN	Hanoi	97	0.30	1	49	0.12
2	United Kingdom	GB	London	67	0.20	1	281	0.08
3	Russia	RU	Moscow	144	17.10	1	9	6.60
4	Argentina	AR	Buenos Aires	45	2.80	1	17	1.08
5	Spain	ES	Madrid	47	0.50	1	94	0.19
6	Netherlands	NL	Amsterdam	17	0.04	1	508	0.02
7	Peru	PE	Lima	33	1.30	1	26	0.50

Удалим этот столбец, чтобы рассмотреть другие методы.

```
countries.drop(labels = 'area_miles', axis = 1, inplace = True)
```

Можно сложнее

Мы можем усложнить код и добиться такого же результата, применив **методы .iterrows()** и **.iloc[]**.

```
# выведем индекс и содержание строк
for index, row in countries.iterrows():
  # запишем для каждой строки (index) в новый столбец area_miles
  # округленное значение площади row.area в милях
  countries.loc[index, 'area_miles'] = np.round(row.area / 2.59, 2)

# посмотрим на результат
countries
```

	country	code	city	population	area	sea	pop_density	area_miles
0	China	CN	Beijing	1400	9.60	1	153	3.71
1	Vietnam	VN	Hanoi	97	0.30	1	49	0.12
2	United Kingdom	GB	London	67	0.20	1	281	0.08
3	Russia	RU	Moscow	144	17.10	1	9	6.60
4	Argentina	AR	Buenos Aires	45	2.80	1	17	1.08
5	Spain	ES	Madrid	47	0.50	1	94	0.19
6	Netherlands	NL	Amsterdam	17	0.04	1	508	0.02
7	Peru	PE	Lima	33	1.30	1	26	0.50

```
# снова удалим этот столбец
countries.drop(labels = 'area_miles', axis = 1, inplace = True)
```

Можно проще

При этом конечно есть гораздо более простой способ добавления нового столбца.

```
# мы можем объявить столбец и присвоить ему нужно нам значение
countries['area_miles'] = (countries.area / 2.59).round(2)
countries
```

	country	code	city	population	area	sea	pop_density	area_miles
0	China	CN	Beijing	1400	9.60	1	153	3.71
1	Vietnam	VN	Hanoi	97	0.30	1	49	0.12
2	United Kingdom	GB	London	67	0.20	1	281	0.08
3	Russia	RU	Moscow	144	17.10	1	9	6.60
4	Argentina	AR	Buenos Aires	45	2.80	1	17	1.08
5	Spain	ES	Madrid	47	0.50	1	94	0.19
6	Netherlands	NL	Amsterdam	17	0.04	1	508	0.02
7	Peru	PE	Lima	33	1.30	1	26	0.50

Удаление строк и столбцов

Удаление строк

Для удаления строк можно использовать **метод .drop()** с параметрами *label*s (индекс удаляемых строк) и *axis = 0*.

```
# удалим строки с индексом 0 и 1
countries.drop(labels = [0, 1], axis = 0)
```

	country	code	city	population	area	sea	pop_density	area_miles
2	United Kingdom	GB	London	67	0.20	1	281	0.08
3	Russia	RU	Moscow	144	17.10	1	9	6.60
4	Argentina	AR	Buenos Aires	45	2.80	1	17	1.08

5	Spain	ES	Madrid	47	0.50	1	94	0.19
6	Netherlands	NL	Amsterdam	17	0.04	1	508	0.02
7	Peru	PE	Lima	33	1.30	1	26	0.50

Кроме того, можно использовать метод **.drop()** с единственным параметром *index*.

```
# удалим строки с индексом 5 и 7
countries.drop(index = [5, 7])
```

	country	code	city	population	area	sea	pop_density	area_miles
0	China	CN	Beijing	1400	9.60	1	153	3.71
1	Vietnam	VN	Hanoi	97	0.30	1	49	0.12
2	United Kingdom	GB	London	67	0.20	1	281	0.08
3	Russia	RU	Moscow	144	17.10	1	9	6.60
4	Argentina	AR	Buenos Aires	45	2.80	1	17	1.08
6	Netherlands	NL	Amsterdam	17	0.04	1	508	0.02

Мы также можем в параметр *index* передать индекс датафрейма через атрибут **index**.

```
# удалим четвертую строку
countries.drop(index = countries.index[4])
```

	country	code	city	population	area	sea	pop_density	area_miles
0	China	CN	Beijing	1400	9.60	1	153	3.71
1	Vietnam	VN	Hanoi	97	0.30	1	49	0.12
2	United Kingdom	GB	London	67	0.20	1	281	0.08
3	Russia	RU	Moscow	144	17.10	1	9	6.60
5	Spain	ES	Madrid	47	0.50	1	94	0.19
6	Netherlands	NL	Amsterdam	17	0.04	1	508	0.02
7	Peru	PE	Lima	33	1.30	1	26	0.50

С атрибутом датафрейма **index** мы можем делать срезы.

```
# удалим каждую вторую строку, начиная с четвертой с конца
countries.drop(index = countries.index[-4::2])
```

	country	code	city	population	area	sea	pop_density	area_miles
0	China	CN	Beijing	1400	9.6	1	153	3.71
1	Vietnam	VN	Hanoi	97	0.3	1	49	0.12
2	United Kingdom	GB	London	67	0.2	1	281	0.08
3	Russia	RU	Moscow	144	17.1	1	9	6.60
5	Spain	ES	Madrid	47	0.5	1	94	0.19
7	Peru	PE	Lima	33	1.3	1	26	0.50

Удаление столбцов

Параметры *labels* (номера столбцов) и *axis = 1* метода **.drop()** позволяют удалять столбцы.

```
# удалим столбцы area_miles и code
countries.drop(labels = ['area_miles', 'code'], axis = 1)
```

	country	city	population	area	sea	pop_density
0	China	Beijing	1400	9.60	1	153
1	Vietnam	Hanoi	97	0.30	1	49
2	United Kingdom	London	67	0.20	1	281
3	Russia	Moscow	144	17.10	1	9
4	Argentina	Buenos Aires	45	2.80	1	17
5	Spain	Madrid	47	0.50	1	94
6	Netherlands	Amsterdam	17	0.04	1	508
7	Peru	Lima	33	1.30	1	26

Такой же результат можно получить, передав список удаляемых столбцов в *параметр columns*.

```
# снова удалим столбцы area_miles и code
countries.drop(columns = ['area_miles', 'code'])
```

	country	city	population	area	sea	pop_density
0	China	Beijing	1400	9.60	1	153
1	Vietnam	Hanoi	97	0.30	1	49
2	United Kingdom	London	67	0.20	1	281
3	Russia	Moscow	144	17.10	1	9
4	Argentina	Buenos Aires	45	2.80	1	17
5	Spain	Madrid	47	0.50	1	94
6	Netherlands	Amsterdam	17	0.04	1	508
7	Peru	Lima	33	1.30	1	26

Через атрибут датафрейма **columns** мы можем передать номера удаляемых столбцов.

```
# удалим последний столбец (area_miles)
countries.drop(columns = countries.columns[-1])
```

	country	code	city	population	area	sea	pop_density
0	China	CN	Beijing	1400	9.60	1	153
1	Vietnam	VN	Hanoi	97	0.30	1	49
2	United Kingdom	GB	London	67	0.20	1	281
3	Russia	RU	Moscow	144	17.10	1	9
4	Argentina	AR	Buenos Aires	45	2.80	1	17

5	Spain	ES	Madrid	47	0.50	1	94
6	Netherlands	NL	Amsterdam	17	0.04	1	508
7	Peru	PE	Lima	33	1.30	1	26

Наконец удалим пятую строку и несколько столбцов разобранными выше способами и сохраним изменения.

```
countries.drop(index = 4, inplace = True)
countries.drop(columns = ['code', 'pop_density', 'area_miles'], inplace = True)
countries
```

	country	city	population	area	sea
0	China	Beijing	1400	9.60	1
1	Vietnam	Hanoi	97	0.30	1
2	United Kingdom	London	67	0.20	1
3	Russia	Moscow	144	17.10	1
5	Spain	Madrid	47	0.50	1
6	Netherlands	Amsterdam	17	0.04	1
7	Peru	Lima	33	1.30	1

Удаление по многоуровнему индексу

Давайте посмотрим, как удалять строки и столбцы в датафрейме с многоуровневым (иерархическим) индексом. Вначале вновь создадим соответствующий датафрейм.

```
# подготовим данные для многоуровневого индекса строк
rows = [('Asia', 'CN'),
        ('Asia', 'VN'),
        ('Europe', 'GB'),
        ('Europe', 'RU'),
        ('Europe', 'ES'),
        ('Europe', 'NL'),
        ('S. America', 'PE')]

# и столбцов
cols = [('names', 'country'),
        ('names', 'city'),
        ('data', 'population'),
        ('data', 'area'),
        ('data', 'sea')]

# создадим многоуровневый (иерархический) индекс
# для индекса строк добавим названия столбцов индекса через параметр names
custom_multindex = pd.MultiIndex.from_tuples(rows, names = ['region', 'code'])
custom_multicols = pd.MultiIndex.from_tuples(cols)

# поместим индексы в атрибуты index и columns датафрейма
countries.index = custom_multindex
countries.columns = custom_multicols

# посмотрим на результат
countries
```

	names		data		

		country	city	population	area	sea
region	code					
Asia	CN	China	Beijing	1400	9.60	1
	VN	Vietnam	Hanoi	97	0.30	1
Europe	GB	United Kingdom	London	67	0.20	1
	RU	Russia	Moscow	144	17.10	1
	ES	Spain	Madrid	47	0.50	1
	NL	Netherlands	Amsterdam	17	0.04	1
S. America	PE	Peru	Lima	33	1.30	1

Удаление строк

Вначале обратимся к строкам и удалим азиатский регион. Для этого воспользуемся методом **.drop()**, которому передадим параметр *labels = 'Asia'*. Кроме того, укажем, что удаляем именно строки (*axis = 0*) и что Азия находится в индексе под названием region (т.е. *level = 0*).

```
1 countries.drop(labels = 'Asia', axis = 0, level = 0)
```

		names		data		
		country	city	population	area	sea
region	code					
Europe	GB	United Kingdom	London	67	0.20	1
	RU	Russia	Moscow	144	17.10	1
	ES	Spain	Madrid	47	0.50	1
	NL	Netherlands	Amsterdam	17	0.04	1
S. America	PE	Peru	Lima	33	1.30	1

Кроме того, строки можно удалять с помощью параметра *index* и указанием через *level*, по какому столбцу индекса мы будем искать удаляемую строку.

```
1 # удалим запись о России по ее индексу в столбце code (т.е. level = 1)
2 countries.drop(index = 'RU', level = 1)
```

		names		data		
		country	city	population	area	sea
region	code					
Asia	CN	China	Beijing	1400	9.60	1
	VN	Vietnam	Hanoi	97	0.30	1
Europe	GB	United Kingdom	London	67	0.20	1
	ES	Spain	Madrid	47	0.50	1
	NL	Netherlands	Amsterdam	17	0.04	1
S. America	PE	Peru	Lima	33	1.30	1

Удаление столбцов

Удаление столбцов датафрейма с многоуровневым индексом происходит аналогично строкам. Передадим методу **.drop()** параметры *labels*, *level* и *axis = 1* для удаления столбца по его наименованию (*labels*) на нужном нам уровне (*level*) индекса.

```
# удалим все столбцы в разделе names на нулевом уровне индекса столбцов
countries.drop(labels = 'names', level = 0, axis = 1)
```

		data		
		population	area	sea
region	code			
Asia	CN	1400	9.60	1
	VN	97	0.30	1
Europe	GB	67	0.20	1
	RU	144	17.10	1
	ES	47	0.50	1
	NL	17	0.04	1
S. America	PE	33	1.30	1

Для удаления столбцов можно использовать параметр *columns* с указанием соответствующего уровня индекса (*level*) столбцов.

```
# например, удалим столбцы city и area на втором уровне (level = 1) индекса
countries.drop(columns = ['city', 'area'], level = 1)
```

		names	data	
		country	population	sea
region	code			
Asia	CN	China	1400	1
	VN	Vietnam	97	1
Europe	GB	United Kingdom	67	1
	RU	Russia	144	1
	ES	Spain	47	1
	NL	Netherlands	17	1
S. America	PE	Peru	33	1

Обратите внимание, что удалению столбцов не помешал тот факт, что они находятся в разных индексах первого (*level = 0*) уровня, а именно city находится в names, а population — в data.

Применение функций

Библиотека Pandas позволяет использовать функции для изменения данных в датафрейме.

```
# создадим новый датафрейм с данными нескольких человек
people = pd.DataFrame({'name'        : ['Алексей', 'Иван', 'Анна', 'Ольга', 'Николай'],
                       'gender'      : [1, 1, 0, 2, 1],
```

```
                        'age'         : [35, 20, 13, 28, 16],
                        'height'      : [180.46, 182.26, 165.12, 168.04, 178.68],
                        'weight'      : [73.61, 75.34, 50.22, 52.14, 69.72]
                        })

people
```

	name	gender	age	height	weight
0	Алексей	1	35	180.46	73.61
1	Иван	1	20	182.26	75.34
2	Анна	0	13	165.12	50.22
3	Ольга	2	28	168.04	52.14
4	Николай	1	16	178.68	69.72

Метод .map()

Предположим, что мы хотим явно прописать мужской (male) и женский (female) пол в наших данных. В этому случае можно воспользоваться **методом .map()**. Вначале создадим карту (map) того, как преобразовать существующие значения в новые.

```
# такая карта представляет собой питоновский словарь,
# где ключи - это старые данные, а значения - новые
gender_map = {0: 'female', 1 : 'male'}
```

Применим эту карту к нужному нам столбцу датафрейма.

```
people['gender'] = people['gender'].map(gender_map)
people
```

	name	gender	age	height	weight
0	Алексей	male	35	180.46	73.61
1	Иван	male	20	182.26	75.34
2	Анна	female	13	165.12	50.22
3	Ольга	NaN	28	168.04	52.14
4	Николай	male	16	178.68	69.72

Те значения, которые в карте не указаны (в четвертой строке была ошибка, пол был помечен цифрой два), превращаются в NaN (not a number), пропущенное значение.

> Обратите внимание, что словарь, в данном случае, по сути является функцией, которую мы применяем к значениям определенного столбца.

Кроме того замечу, что в большинстве случаев мы будем проводить обратное преобразование, превращая строковые категориальные значения в числовые.

В метод **.map()** мы можем передать и lambda-функцию.

```
# например, для того, чтобы выявить совершеннолетних и несовершеннолетних людей
people['age_group'] = people['age'].map(lambda x: 'adult' if x >= 18 else 'minor')
```

```
3 people
```

	name	gender	age	height	weight	age_group
0	Алексей	male	35	180.46	73.61	adult
1	Иван	male	20	182.26	75.34	adult
2	Анна	female	13	165.12	50.22	minor
3	Ольга	NaN	28	168.04	52.14	adult
4	Николай	male	16	178.68	69.72	minor

Удалим только что созданный столбец age_group.

```
1 people.drop(labels = 'age_group', axis = 1, inplace = True)
```

Вместо lambda-функции, например, для более сложных преобразований, можно испольвовать обычную собственную функцию.

```
1  # обратите внимание, такая функция не допускает дополнительных параметров,
2  # только те данные, которые нужно преобразовать (age)
3  def get_age_group(age):
4
5      # например, мы не можем сделать threshold произвольным параметром
6      threshold = 18
7
8      if age >= threshold:
9          age_group = 'adult'
10
11     else:
12         age_group = 'minor'
13
14     return age_group
```

Применим эту функцию к столбцу age.

```
1 people['age_group'] = people['age'].map(get_age_group)
2 people
```

	name	gender	age	height	weight	age_group
0	Алексей	male	35	180.46	73.61	adult
1	Иван	male	20	182.26	75.34	adult
2	Анна	female	13	165.12	50.22	minor
3	Ольга	NaN	28	168.04	52.14	adult
4	Николай	male	16	178.68	69.72	minor

```
1 # снова удалим созданный столбец
2 people.drop(labels = 'age_group', axis = 1, inplace = True)
```

Функция np.where()

Точно такой же результат мы можем получить, применив функцию **np.where()** библиотеки Numpy к нужному нам столбцу.

```
1 # внутри функции np.where() три параметра: (1) условие,
```

```
# (2) значение, если условие выдает True, (3) и значение, если условие выдает False
people['age_group'] = np.where(people['age'] >= 18, 'adult', 'minor')
people
```

	name	gender	age	height	weight	age_group
0	Алексей	male	35	180.46	73.61	adult
1	Иван	male	20	182.26	75.34	adult
2	Анна	female	13	165.12	50.22	minor
3	Ольга	NaN	28	168.04	52.14	adult
4	Николай	male	16	178.68	69.72	minor

Замечу, что такой способ превращения количественных данных в категориальные называется **binning** или **bucketing**. О нем мы подробно поговорим на занятии по преобразованию данных.

```
# удалим созданный столбец
people.drop(labels = 'age_group', axis = 1, inplace = True)
```

Метод .where()

Метод .where() библиотеки Pandas действует немного иначе. В нем мы прописываем условие, которое хотим применить к отдельному столбцу или всему датафрейму:

- если условие выполняется (т.е. оценивается как True), мы сохраняем текущее значение датафрейма;
- если условие не выполняется (False), то значение заменяется на новое, указанное в методе **.where()**.

Рассмотрим применение этого метода на примерах.

```
# заменим возраст тех, кому меньше 18, на NaN
people.age.where(people.age >= 18, other = np.nan)
```

```
0    35.0
1    20.0
2     NaN
3    28.0
4     NaN
Name: age, dtype: float64
```

В примере выше возраст тех, кто не моложе 18 (True), остался без изменений. Для остальных (False) значение изменилось на пропущенное (параметр *other*). Обратите внимание, что тип данных этого столбца превратился во float. Это связано с тем, что в столбце появились пропущенные значения.

```
# создадим матрицу из вложенных списков
nums_matrix = [[-13, 7, 1],
               [4, -2, 25],
               [45, -3, 8]]

# преобразуем в датафрейм
# (матрица не обязательно должна быть массивом Numpy (!))
nums = pd.DataFrame(nums_matrix)
nums
```

	0	1	2
0	-13	7	1
1	4	-2	25
2	45	-3	8

```
# если число положительное (nums < 0 == True), оставим его без изменений
# если отрицательное (False), заменим на обратное (т.е. сделаем положительным)
nums.where(nums > 0, other = -nums)
```

	0	1	2
0	13	7	1
1	4	2	25
2	45	3	8

Здесь можно отметить два интересных момента:

- мы применили метод **.where()** ко всему датафрейму;
- в качестве значения, на которое нужно заменить текущее при False, мы использовали сами значения датафрейма, но со знаком минус (`-nums`).

Перейдем к методу **.apply()**.

Метод .apply()

Применение функции с аргументами

В отличие от **.map()**, **метод .apply()** позволяет передавать именованные аргументы в применяемую функцию.

```
# объявим функцию, которой можно передать не только значение возраста, но и порог,
# при котором мы будем считать человека совершеннолетним
def get_age_group(age, threshold):

    if age >= int(threshold):
        age_group = 'adult'
    else:
        age_group = 'minor'

    return age_group
```

```
# применим эту функцию к столбцу age, выбрав в качестве порогового значения 21 год
people['age_group'] = people['age'].apply(get_age_group, threshold = 21)

# посмотрим на результат
people
```

	name	gender	age	height	weight	age_group
0	Алексей	male	35	180.46	73.61	adult
1	Иван	male	20	182.26	75.34	minor
2	Анна	female	13	165.12	50.22	minor
3	Ольга	NaN	28	168.04	52.14	adult

4	Николай	male	16	178.68	69.72	minor

Применение к столбцам

В метод **.apply()** можно передать уже имеющуюся в Питоне функцию, например, из библиотеки Numpy.

```
# заменим значения в столбцах height и weight на медиану по столбцам
people[['height', 'weight']] = people[['height', 'weight']].apply(np.median,
                                                                  axis = 0)
people
```

	name	gender	age	height	weight	age_group
0	Алексей	male	35	178.68	69.72	adult
1	Иван	male	20	178.68	69.72	minor
2	Анна	female	13	178.68	69.72	minor
3	Ольга	NaN	28	178.68	69.72	adult
4	Николай	male	16	178.68	69.72	minor

Применение к строкам

Теперь поработаем со строками. Создадим функцию, которая считает индекс массы тела (body mass index, BMI) на основе веса и роста человека.

```
# внутри функции разделим вес на квадрат роста
def get_bmi(x):
    bmi = x['weight'] / (x['height'] / 100) ** 2
    return bmi
```

Теперь применим эту функцию к каждой строке и сохраним результат в новом столбце.

```
# для применения функции к строке используется параметр axis = 1
people['bmi'] = people.apply(get_bmi, axis = 1).round(2)
people
```

	name	gender	age	height	weight	age_group	bmi
0	Алексей	male	35	180.0	74.0	adult	22.84
1	Иван	male	20	182.0	75.0	minor	22.64
2	Анна	female	13	165.0	50.0	minor	18.37
3	Ольга	NaN	28	168.0	52.0	adult	18.42
4	Николай	male	16	179.0	70.0	minor	21.85

Метод .applymap()

Метод .applymap() позволяет применять функции с именованными аргументами *ко всему датафрейму* (метод **.apply()** применяется только к строкам или столбцам). Рассмотрим несложный пример.

```
# создадим датафрейм из чисел
nums_matrix = [[13, 7, 1],
               [4, 2, 25],
               [45, 3, 8]]

nums = pd.DataFrame(nums_matrix)
nums
```

	0	1	2
0	13	7	1
1	4	2	25
2	45	3	8

```
# объявим функцию, которая на входе принимает число x и
# прибавляет к нему другое число, указанное в параметре number
def add_number(x, number):
  return x + number
```

```
# передадим методу .applymap() функцию add_number и
# прибавим единицу к каждому элементу датафрейма
nums.applymap(add_number, number = 1)
```

	0	1	2
0	14	8	2
1	5	3	26
2	46	4	9

Метод .pipe()

Метод .pipe(), как следует из его названия, позволяет создать pipeline и последовательно применить несколько функций к датафрейму. Вновь создадим исходный датафрейм с параметрами нескольких людей.

```
people = pd.DataFrame({'name'        : ['Алексей', 'Иван', 'Анна', 'Ольга', 'Николай'],
                       'gender'      : [1, 1, 0, 2, 1],
                       'age'         : [35, 20, 13, 28, 16],
                       'height'      : [180.46, 182.26, 165.12, 168.04, 178.68],
                       'weight'      : [73.61, 75.34, 50.22, 52.14, 69.72]
                       })

people
```

	name	gender	age	height	weight
0	Алексей	1	35	180.46	73.61
1	Иван	1	20	182.26	75.34
2	Анна	0	13	165.12	50.22
3	Ольга	2	28	168.04	52.14
4	Николай	1	16	178.68	69.72

Объявим несколько функций, которые мы могли бы применить к датафрейму.

```
# в первую очередь скопируем датафрейм
def copy_df(df):
   return df.copy()

# заменим значения столбца на новые с помощью метода .map()
def map_column(df, column, label1, label2):
  labels_map = {0: label1, 1 : label2}
  df[column] = df[column].map(labels_map)
  return df

# кроме этого, создадим функцию для превращения количественной переменной
# в бинарную категориальную
def to_categorical(df, newcol, condcol, thres, cat1, cat2):
  df[newcol] = np.where(df[condcol] >= thres, cat1, cat2)
  return df
```

Последовательно применим эти функции с помощью нескольких методов **.pipe()**.

```
people_processed = (people.
                    pipe(copy_df). # copy_df() применится ко всему датафрейму
                    pipe(map_column, 'gender', 'female', 'male'). # map_column() к стол
                    pipe(to_categorical, 'age_group', 'age', 18, 'adult', 'minor')) # t
```

Посмотрим на результат и кроме того убедимся, что исходный датафрейм не изменился.

```
people_processed
```

	name	gender	age	height	weight	age_group
0	Алексей	male	35	180.46	73.61	adult
1	Иван	male	20	182.26	75.34	adult
2	Анна	female	13	165.12	50.22	minor
3	Ольга	NaN	28	168.04	52.14	adult
4	Николай	male	16	178.68	69.72	minor

```
people
```

	name	gender	age	height	weight
0	Алексей	1	35	180.46	73.61
1	Иван	1	20	182.26	75.34
2	Анна	0	13	165.12	50.22
3	Ольга	2	28	168.04	52.14
4	Николай	1	16	178.68	69.72

Перейдем к следующему разделу, который посвящен соединению датафреймов.

Соединение датафреймов

Рассмотрим, как мы можем соединить два датафрейма с помощью функций/методов **pd.concat()**, **pd.merge()** и **.join()**. Начнем с функции **pd.concat()**.

pd.concat()

В качестве примера возьмем информацию о стоимости канцелярских товаров в двух магазинах.

```
s1 = pd.DataFrame({
  'item': ['карандаш', 'ручка', 'папка', 'степлер'],
  'price': [220, 340, 200, 500]
})

s2 = pd.DataFrame({
  'item': ['клей', 'корректор', 'скрепка', 'бумага'],
  'price': [200, 240, 100, 300]
})
```

Выведем результат.

```
s1
```

	item	price
0	карандаш	220
1	ручка	340
2	папка	200
3	степлер	500

```
s2
```

	item	price
0	клей	200
1	корректор	240
2	скрепка	100
3	бумага	300

Соединение «один на другой»

В первую очередь мы можем совместить два датафрейма, поставив их «один на другой».

```
# передадим в функцию pd.concat() список из соединяемых датафреймов,
# укажем параметр axis = 0 (значение по умолчанию)
pd.concat([s1, s2], axis = 0)
```

	item	price
0	карандаш	220
1	ручка	340
2	папка	200
3	степлер	500
0	клей	200
1	корректор	240
2	скрепка	100
3	бумага	300

Как вы видите, индекс не обновился.

```
# обновим индекс через параметр ignore_index = True
pd.concat([s1, s2], axis = 0, ignore_index = True)
```

	item	price
0	карандаш	220
1	ручка	340
2	папка	200
3	степлер	500
4	клей	200
5	корректор	240
6	скрепка	100
7	бумага	300

Именно таким способом предлагается добавлять новые строки вместо удаленного из библиотеки метода **.append()**. При этом здесь есть нюанс, в функцию **pd.concat()** нельзя передать словарь, только объекты Series и DataFrame.

При соединении датафреймов мы можем создать многоуровневый (иерархический) индекс. Например, создадим отдельную группу для товаров первого магазина (s1) и товаров второго (s2).

```
# передадим в параметр keys названия групп индекса,
# параметр names получим названия уровней индекса
by_shop = pd.concat([s1, s2], axis = 0, keys = ['s1', 's2'], names = ['s', 'id'])
by_shop
```

		item	price
s	id		
s1	0	карандаш	220
	1	ручка	340
	2	папка	200
	3	степлер	500
s2	0	клей	200
	1	корректор	240
	2	скрепка	100
	3	бумага	300

Посмотрим на созданный индекс.

```
by_shop.index
```

```
MultiIndex([('s1', 0),
            ('s1', 1),
            ('s1', 2),
            ('s1', 3),
            ('s2', 0),
            ('s2', 1),
            ('s2', 2),
```

```
            ('s2', 3)],
           names=['s', 'id'])
```

Выведем первую запись в первой группе.

```
by_shop.loc[('s1', 0)]
```

```
item     карандаш
price          220
Name: (s1, 0), dtype: object
```

Соединение «рядом друг с другом»

Датафреймы можно расположить «рядом друг с другом».

```
# для этого сразу используем параметр axis = 1
# одновременно сразу создадим группы для многоуровневого индекса столбцов
pd.concat([s1, s2], axis = 1, keys = ['s1', 's2'])
```

	s1		s2	
	item	price	item	price
0	карандаш	220	клей	200
1	ручка	340	корректор	240
2	папка	200	скрепка	100
3	степлер	500	бумага	300

Выберем вторую группу (второй магазин) с помощью метода **.iloc[]**.

```
pd.concat([s1, s2], axis = 1, keys = ['s1', 's2']).loc[:,'s2']
```

	item	price
0	клей	200
1	корректор	240
2	скрепка	100
3	бумага	300

Полученный через соединение
результат и в целом любой датафрейм можно транспонировать.

```
# для транспонирования датафрейма используется метод .T или .transpose()
pd.concat([s1, s2], axis = 1, keys = ['s1', 's2'
```

		0	1	2	3
s1	item	карандаш	ручка	папка	степлер
	price	220	340	200	500
s2	item	клей	корректор	скрепка	бумага
	price	200	240	100	300

Итак, **pd.concat()** выполняет простое «склеивание» датафреймов по вертикали или по горизонтали. Теперь посмотрим, что делать, если датафреймы нужно соединить по