Лекция 2: Библиотека Pandas

Автор: Сергей Вячеславович Макрушин, 2022 г.

e-mail: s-makrushin@yandex.ru (mailto:s-makrushin@yandex.ru)

При подготовке лекции использованы материалы:

- Уэс Маккинли Python и анализ данных / Пер. с англ. Слипкин А.А. - М.: ДМК Пресс, 2015

V 0.5 05.09.2022

In [1]:

```
# загружаем стиль для оформления презентации
from IPython.display import HTML
from urllib.request import urlopen
html = urlopen("file:./lec_v2.css")
HTML(html.read().decode('utf-8'))
```

Out[1]:

Оглавление

-

- к оглавлению

Знакомство с Pandas

- к оглавлению

Pandas - библиотекая построенная на основе функционала библиотеки NumPy, обеспечивающая удобную инфраструкутуру для обработки панельных (табличных) данных (Pandas - от panel data sets).

Основным классом Pandas является **DataFrame**, объекты DataFrame - многомерные (прежде всего, двухмерные, т.е. табличные) массивы с метками для строк и столбцов. DataFrame позволяет хранить:

- разнородные данные в различных столбцах
- корректно работать с пропущенными данными.

Кроме операций, поддерживаемых NumPy, библиотека Pandas реализует множество операций для работы с данными, характерных для работы с электронными таблицами и базами данных.

Pandas может эффективно использоваться для следующих задач:

- **Изменнии формы** таблиц (reshape).
- Работа с **пропусками** (отсутствующими значениями).
- Работа со **строковыми заначениями** в таблицах.
- **Фильтрация** табличных наборов данных.
- Работа с различными видами **индексов таблиц** (в том числе иерархическими индексами).
- Построение **срезов** и **прихотливая индексаация** для различных типов индексов.
- **Группировка** значений в табличных данных.
- **Выравнивание** (alignment) и **объединение** (join) таблиц.
- Построение **сводных таблиц** (pivoting).
- Чтение и запись данных в различных форматах.

Сравнение Pandas с альтернативными инструментами

NumPy

Инструмент для численных расчетов.

Ключевые возможности:

- Высокопроизводительные (по памяти и скорости) операции с многомерными числовыми массивами.
- Удобство описания манипуляций с многомерными однородными массивами данных.

Использование:

- Данные из NumPy могут напрямую поставляться в библиотеки машнниого обучения и глубокго обучения.

Excel

Инструмент для:

- **ввода и редактирования структурированных данных**
- **манипулирования и подготовки данных**
- **проведения расчетов**
- **моделирования**
- **хранения структурированной информации**
- **презентации данных**
  - **в табличном виде**
  - **в графическом виде**

Pandas

Инструмент для манипулирования данными их анализа и визуализации

Ключевые возможности:

- Удобная и высокопроизводительная работа с табличными данными: двухмерным массивом с резнородными (в т.ч. по типу данных) столбцами и возможностью нетривиальной индексации строк таблиц.

Использование:

- Данные из Pandas обычно **НЕ могут** напрямую поставляться в библиотеки машнниого обучения и глубокго обучения, но Pandas очень часто **используется для предварительной подготовки данных** для машинного обучения.

В сравнении с Excel:

- (-) **ввода, редактирования и просмотра структурированных данных**
- (+) **манипулирования и подготовки данных**
- (+) **проведения расчетов**
- (+/-) **моделирования**
- (+/-) **хранения структурированной информации**
- **презентации данных**
  - (-) **в табличном виде**
  - (-/+(с помощью други инструментов))**в графическом виде**

Серии (Series) - одномерные массивы в Pandas

- к оглавлению

In [120]:

```
import numpy as np
import pandas as pd
```

Фундаментальные структуры данных Pandas - классы **Series**, **DataFrame** и **Index**.

Объект **Series** - одномерный массив индексированных данных.

In [121]:

```
# создание Series на основе списка Python:
sr1 = pd.Series([5, 6, 2, 9, 12])
sr1
```

Out[121]:

```
0     5
1     6
2     2
3     9
4    12
dtype: int64
```

In [15]:

```
sr1.values # атрибут values - это массив NumPy со значениями
```

Out[15]:

```
array([ 5,  6,  2,  9, 12], dtype=int64)
```

In [122]:

```
type(sr1.values)
```

Out[122]:

```
numpy.ndarray
```

In [16]:

```
sr1.index # index - массивоподобный объект типа pd.Index
```

Out[16]:

```
RangeIndex(start=0, stop=5, step=1)
```

In [123]:

```
# Обращение к элементу серии по индексу:
sr1[2]
```

Out[123]:

```
2
```

In [124]:

```
# Срез серии по индексу:
sr1[:3]
```

Out[124]:

```
0    5
1    6
2    2
dtype: int64
```

Основное различие между одномерным массивом библиотеки NumPy и Series - **наличие у Series явного индекса, определяющего доступ к данным массива**.

Индекс массива NumPy:

- всегда целочисленный
- представлен последовательно идущими целыми числами начиная с 0
- описывается неявно (т.е. не подразумевается явное определение индекса т.к. не допускаются альтернативные варианты индексации)

Индекс объекта Series:

- может состоять из значений типа, выбранного пользователем (например, строк)

- индекс может описываться явно (вариант по умолчанию совпадает со способом индексации в NumPy) и связывается со значениями

In [126]:

```
# Создание серии с явным определением индекса:
sr2 = pd.Series([5, 6, 2, 9, 12],
                index=['Cochise County', 'Pima County', 'Santa Cruz County',
                                         'Maricopa County', 'Yuma County'])
sr2
```

Out[126]:

```
Cochise County        5
Pima County           6
Santa Cruz County     2
Maricopa County       9
Yuma County          12
dtype: int64
```

In [127]:

```
# Обращение к элементу серии по нецелочисленному индексу:
sr2['Pima County']
```

Out[127]:

```
6
```

In [128]:

```
sr2['Pima County':]
```

Out[128]:

```
Pima County           6
Santa Cruz County     2
Maricopa County       9
Yuma County          12
dtype: int64
```

Объект Series можно рассматривать как специализированный вариант словаря.

- Словарь - структура, задающая соответствие произвольных ключей набору произвольных значений
- Объект Series:
  - структура, задающая соответствие **типизированных ключей** набору **типизированных значений**
  - кроме того, для ключей (значений индекса) задана **последовательность их следования**

In [129]:

```
# объект Series можно создавать непосредственно из словаря Python:
# (т.к. словарь не определяет порядок обхода, то такая форма задания может привести
# к созданию серии с иной последовательностью индекс-значение)
sr3 = pd.Series({'California': 38332521,
                 'Texas': 26448193,
                 'New York': 19651127,
                 'Florida': 19552860,
                 'Illinois': 12882135})
sr3
```

Out[129]:

```
California    38332521
Texas         26448193
New York      19651127
Florida       19552860
Illinois      12882135
dtype: int64
```

In [12]:

```
# изменение индекса:
sr3.index = ["Cochice", "Pima", "Santa Cruz", "Maricopa", "Yuma"]
sr3
```

Out[12]:

```
Cochice       38332521
Pima          26448193
Santa Cruz    19651127
Maricopa      19552860
Yuma          12882135
dtype: int64
```

Датафрэйм (DataFrame) - двумерные массивы в Pandas

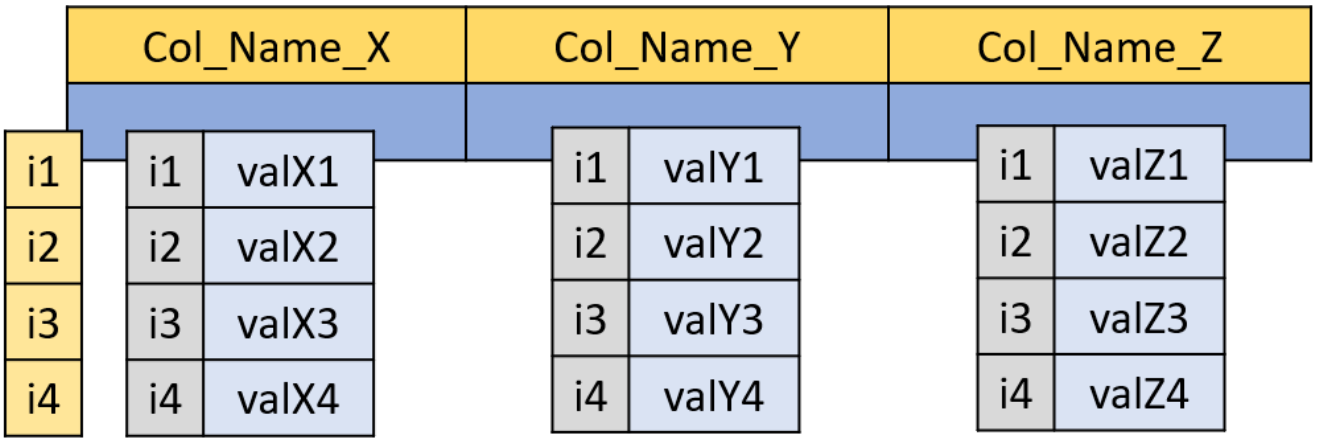
- к оглавлению

Введение

- к оглавлению

DataFrame - аналог двухмерного массива с гибкими индексами строк и гибкими именами столбцов.

Аналогично тому, что двумерный массив можно рассматривать как упорядоченную последовательность выровненных столбцов, объект DataFrame можно рассматривать как упорядоченную последовательность выровненных объектов Series. Под «выравниванием» понимается то, что они используют один и тот же индекс.



Логическое представление организации DataFrame

In [130]:

```
# создание DataFrame на основе двух Series:
s_population = pd.Series({'California': 38332521,
                  'Texas': 26448193,
                  'New York': 19651127,
                  'Florida': 19552860,
                  'Illinois': 12882135})
s_area = pd.Series({'California': 423967, 'Texas': 695662, 'New York': 141297,
             'Florida': 170312, 'Illinois': 149995})
states = pd.DataFrame({'population': s_population,
                                  'area': s_area})
states # jupyter умеет красиво выводить таблицы Pandas DataFrame
```

Out[130]:

	population	area
California	38332521	423967
Texas	26448193	695662
New York	19651127	141297
Florida	19552860	170312
Illinois	12882135	149995

In [131]:

```
# для всех столбцов DataFrame имеется единый индекс:
states.index
```

Out[131]:

```
Index(['California', 'Texas', 'New York', 'Florida', 'Illinois'], dtype='obje
ct')
```

In [132]:

```
# у объекта DataFrame есть атрибут columns, содержащий метки столбцов, - объект типа Index
states.columns
```

Out[132]:

```
Index(['population', 'area'], dtype='object')
```

In [133]:

```
# DataFrame можно рассматривать как специализированный словарь столбцов.
# DataFrame задает соответствие имени столбца объекту Series:
states['area']
```

Out[133]:

```
California    423967
Texas         695662
New York      141297
Florida       170312
Illinois      149995
Name: area, dtype: int64
```

NB! Важно понимать, что:

- в NumPy элементы по оси 0 принято рассматривать как **строки** (т.е. считается, что `np1[1]` - вернет строку с индексом 1)
- в Pandas аналогичная конструкция (`pd1[1]`) возвращает **столбец** типа Series.

In [134]:

```
np1 = np.array([[1, 2, 3], [4, 5, 6]])
np1, np1.shape
```

Out[134]:

```
(array([[1, 2, 3],
        [4, 5, 6]]),
 (2, 3))
```

In [29]:

```
np1[1] # строка с индексом 1
```

Out[29]:

```
array([4, 5, 6])
```

In [135]:

```
#  в Pandas первое измерение (axis=0) рассматривается как размерность серий (столбцов),
pd1 = pd.DataFrame(data=np1)
pd1
```

Out[135]:

	0	1	2
0	1	2	3
1	4	5	6

In [136]:

```
pd1[1] # обращение к столбцу с именем (индексом) 1
```

Out[136]:

```
0    2
1    5
Name: 1, dtype: int32
```

Т.е. индексация DataFrame (т.е. операция вида: `pd1[...]`) ориентирована на манипулирование столбцами.

*DataFrame можно рассматривать как **серию серий*** :

In [137]:

```
type(pd1[1])
```

Out[137]:

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

In [138]:

```
# из этого понятно, почему:
pd1[1][0]
```

Out[138]:

```
2
```

In [34]:

```
# тогда как:
np1[1][0]
```

Out[34]:

```
4
```

In [139]:

```
# создание DataFrame на базе массива NumPy с заданием индекса и имен столбцов
pd2 = pd.DataFrame(data=np1, index=['la', 'lb'],
                   columns=['cl1', 'cl2', 'cl3'] )
pd2
```

Out[139]:

	cl1	cl2	cl3
la	1	2	3
lb	4	5	6

In [140]:

```
# использование заданных индексов:
pd2['cl2']
```

Out[140]:

```
la    2
lb    5
Name: cl2, dtype: int32
```

In [141]:

```
pd2['cl2']['la']
```

Out[141]:

```
2
```

In [142]:

```
# создание DataFrame из списка словарей (ключи - имена столбцов):
pd3 = pd.DataFrame([{'a': 1, 'b': 2, 'c':'Alpha'},
                    {'a':0, 'b': 3, 'c': 'Beta'}])
pd3
```

Out[142]:

	a	b	c
0	1	2	Alpha
1	0	3	Beta

In [143]:

```
# явное задание индекса:
pd3 = pd.DataFrame([{'a': 1, 'b': 2, 'c':'Alpha'},
                    {'a':0, 'b': 3, 'c': 'Beta'}],
                   index=['first', 'second'])
pd3
```

Out[143]:

	a	b	c
first	1	2	Alpha
second	0	3	Beta

In [144]:

```
# в Pandas допускаются пропуски данных
# (и явная индексация упрощает задание данных с пропусками):
pd3 = pd.DataFrame([{'a': 1, 'c':'Alpha'},
                    {'a':0, 'b': 3, 'c': 'Beta'}],
                   index=['first', 'second'])
pd3
```

Out[144]:

	a	c	b
first	1	Alpha	NaN
second	0	Beta	3.0

In [145]:

```
# создание DataFrame из словаря списков (ключи - имена столбцов):
data = {'county': ['Cochice', 'Pima', 'Santa Cruz', 'Maricopa', 'Yuma'],
        'year': [2012, 2012, 2013, 2014, 2014],
        'reports': [4, 24, 31, 2, 3]}
pd4 = pd.DataFrame(data)
pd4
```

Out[145]:

	county	year	reports
0	Cochice	2012	4
1	Pima	2012	24
2	Santa Cruz	2013	31
3	Maricopa	2014	2
4	Yuma	2014	3

In [146]:

```
# явное определение порядка и состава столбцов и индекса:
pd4 = pd.DataFrame(data, columns=['reports', 'county'],
                   index=[chr(ord('a') + i) for i in range(5)])
pd4
```

Out[146]:

	reports	county
a	4	Cochice
b	24	Pima
c	31	Santa Cruz
d	2	Maricopa
e	3	Yuma

Индексация

- к оглавлению

Индексация для серий

In [147]:

```
sr4 = pd.Series([0.25, 0.5, 0.75, 1.0],
                index=['a', 'b', 'c', 'd'])
sr4
```

Out[147]:

```
a    0.25
b    0.50
c    0.75
d    1.00
dtype: float64
```

Серии поддерживают интерфейс, близкий к словарям Python

In [148]:

```
# извлечение элемента серии по аналогии с использованием словаря:
sr4['b']
```

Out[148]:

```
0.5
```

In [149]:

```
# аналогично словарям поддерживается проверка вхождения элемента в индекс серии:
'a' in sr4
```

Out[149]:

```
True
```

In [150]:

```
sr4.keys()
```

Out[150]:

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

In [151]:

```
# в отличие от словарей keys() нужно указывать явно:
for i in sr4.keys():
    print(f'{i} -> {sr4[i]}')
```

```
a -> 0.25
b -> 0.5
c -> 0.75
d -> 1.0
```

In [152]:

```
# итерация по значениям, а не по ключам!
for i in sr4:
    print(f'{i}')
```

```
0.25
0.5
0.75
1.0
```

In [153]:

```
list(sr4.items())
```

Out[153]:

```
[('a', 0.25), ('b', 0.5), ('c', 0.75), ('d', 1.0)]
```

In [155]:

```
for i, v in sr4.items():
    print(f'{i} -> {v}')
```

```
a -> 0.25
b -> 0.5
c -> 0.75
d -> 1.0
```

In [156]:

```
# модификация (добавление) элемента серии:
sr4['e'] = 1.25
sr4
```

Out[156]:

```
a    0.25
b    0.50
c    0.75
d    1.00
e    1.25
dtype: float64
```

In [157]:

```
sr4['e'] = 1.75
sr4
```

Out[157]:

```
a    0.25
b    0.50
c    0.75
d    1.00
e    1.75
dtype: float64
```

Серии поддерживают механизмы индексации, аналогичные массивам NumPy: срезы, маскирование и прихотливое индексирование.

In [158]:

```
# срез с использованием явных индексов (в срезах с явными использованием индексов правая гр
sr4['a':'c']
```

Out[158]:

```
a    0.25
b    0.50
c    0.75
dtype: float64
```

NB! При использовании **явных индексов** правая граница средза **включается в результат**.

In [53]:

```
# прихотливое индексирование с использованием явных индексов:
sr4[['b','a','c']]
```

Out[53]:

```
b    0.50
a    0.25
c    0.75
dtype: float64
```

In [159]:

```
# срез с использованием НЕявных (целочисленных) индексов:
sr4[0:2]
```

Out[159]:

```
a    0.25
b    0.50
dtype: float64
```

In [160]:

```
# прихотливое индексирование с использованием НЕявных индексов:
sr4[[1, 0, 2]]
```

Out[160]:

```
b    0.50
a    0.25
c    0.75
dtype: float64
```

NB! В случае использования **НЕявного целочисленного индекса** использование срезов может выглядеть неоднозначно и **приводить к ошибкам**.

In [57]:

```
sr5 = pd.Series(['a', 'b', 'c'], index=[1, 3, 5])
```

In [161]:

```
# при обычном индексировании используется явный индекс
sr5[1]
```

Out[161]:

```
'a'
```

In [162]:

```
# при использовании среза используется НЕявный индекс:
sr5[1:3] # этот результат может противоречить ожидаемому
```

Out[162]:

```
3    b
5    c
dtype: object
```

Из-за этой потенциальной путаницы в случае целочисленных индексов в библиотеке Pandas предусмотрены специальные атрибуты-индексаторы, позволяющие явным образом применять определенные схемы индексации:

- атрибут **loc** позволяет выполнить индексацию и срезы с использованием явного индекса
- атрибут **iloc** дает возможность выполнить индексацию и срезы, применяя неявный индекс в стиле языка Python

In [60]:

```
sr5
```

Out[60]:

```
1    a
3    b
5    c
dtype: object
```

In [163]:

```
sr5.loc[1] # явный индекс
```

Out[163]:

```
'a'
```

In [165]:

```
sr5.iloc[1] # неявный индекс
```

Out[165]:

```
'b'
```

In [166]:

```
sr5.loc[1:3] # в срезах с явными использованием индексов правая граница включается!
```

Out[166]:

```
1    a
3    b
dtype: object
```

In [167]:

```
sr5.iloc[1:3]
```

Out[167]:

```
3    b
5    c
dtype: object
```

Маскирование для DataFrame

In [170]:

```
# Применение маскирования для серий аналогично NumPy:
sr4[(sr4 > 0.3) & (sr4 < 0.8)]
```

Out[170]:

```
b    0.50
c    0.75
dtype: float64
```

Что происходит внутри:

In [168]:

```
sr4 > 0.3
```

Out[168]:

```
a    False
b     True
c     True
d     True
e     True
dtype: bool
```

In [169]:

```
(sr4 > 0.3) & (sr4 < 0.8)
```

Out[169]:

```
a    False
b     True
c     True
d    False
e    False
dtype: bool
```

Индексация для DataFrame

In [171]:

```
states
```

Out[171]:

	population	area
California	38332521	423967
Texas	26448193	695662
New York	19651127	141297
Florida	19552860	170312
Illinois	12882135	149995

DataFrame может рассматриваться как словарь (серия) серий:

In [174]:

```
states['area']
```

Out[174]:

```
California    423967
Texas         695662
New York      141297
Florida       170312
Illinois      149995
Name: area, dtype: int64
```

In [175]:

```
# для имен столбцов, не конфликтующих с методами DataFrame и синтаксисом Python, допустим т
states.area
```

Out[175]:

```
California    423967
Texas         695662
New York      141297
Florida       170312
Illinois      149995
Name: area, dtype: int64
```

In [176]:

```
# синтаксис словаря допустим и для присвоения (создания новой серии-столбца):
states['density'] = states['population'] / states['area']
states
```

Out[176]:

	population	area	density
California	38332521	423967	90.413926
Texas	26448193	695662	38.018740
New York	19651127	141297	139.076746
Florida	19552860	170312	114.806121
Illinois	12882135	149995	85.883763

NB! Операции среза и маскирования **относятся к строкам (!)**, а не столбцам (это не очень логично, но удобно на практике):

In [177]:

```
states['New York'] # прия явном использовании индекса правая граница включается!
```

Out[177]:

	population	area	density
California	38332521	423967	90.413926
Texas	26448193	695662	38.018740
New York	19651127	141297	139.076746

In [178]:

```
states[:3] # при использовании НЕявного индекса граница не включается
```

Out[178]:

	population	area	density
California	38332521	423967	90.413926
Texas	26448193	695662	38.018740
New York	19651127	141297	139.076746

In [179]:

```
# маскирование работает по строкам:
states[states.density > 100]
```

Out[179]:

	population	area	density
New York	19651127	141297	139.076746
Florida	19552860	170312	114.806121

DataFrame поддерживает двухмерный вариант loc, iloc

In [180]:

```
states.loc[states.density > 100, ['population', 'density']]
```

Out[180]:

	population	density
New York	19651127	139.076746
Florida	19552860	114.806121

In [78]:

```
states.iloc[0, 2] = 90
states
```

Out[78]:

	population	area	density
California	38332521	423967	90.000000
Texas	26448193	695662	38.018740
New York	19651127	141297	139.076746
Florida	19552860	170312	114.806121
Illinois	12882135	149995	85.883763

Обработка данных в библиотеке Pandas

- к оглавлению

Универсальные функции и выравнивание

- к оглавлению

Все универсальные функции библиотеки NumPy работают с объектами Series и DataFrame библиотеки Pandas.

In [97]:

```
import numpy as np
```

In [98]:

```
rs = np.random.RandomState(42)
sr6 = pd.Series(rs.randint(0, 10, 4))
sr6
```

Out[98]:

```
0    6
1    3
2    7
3    4
dtype: int32
```

Результатом применения универсальной функции NumPy к объектам Pandas будет новый объект с сохранением индексов

In [99]:

```
sr7 = np.exp(sr6)
sr7, type(sr7)
```

Out[99]:

```
(0     403.428793
 1      20.085537
 2    1096.633158
 3      54.598150
 dtype: float64,
 pandas.core.series.Series)
```

In [100]:

```
sr7
```

Out[100]:

```
0     403.428793
1      20.085537
2    1096.633158
3      54.598150
dtype: float64
```

In [94]:

```
sr6 # исходная серия осталась неизменной
```

Out[94]:

```
0    6
1    3
2    7
3    4
dtype: int32
```

In [101]:

```
pd5 = pd.DataFrame(rs.randint(0, 10, (3, 4)),
                   columns=['A', 'B', 'C', 'D'])
pd5
```

Out[101]:

	A	B	C	D
0	6	9	2	6
1	7	4	3	7
2	7	2	5	4

In [102]:

```
np.sin(pd5 * np.pi / 4)
```

Out[102]:

	A	B	C	D
0	-1.000000	7.071068e-01	1.000000	-1.000000e+00
1	-0.707107	1.224647e-16	0.707107	-7.071068e-01
2	-0.707107	1.000000e+00	-0.707107	1.224647e-16

При бинарных операциях над двумя объектами Series или DataFrame библиотека Pandas будет выравнивать индексы в процессе выполнения операции. Получившийся в итоге массив содержит объединение индексов двух исходных массивов. Недостающие значения будут отмечены как NaN («нечисловое значение»), с помощью которого библиотека Pandas отмечает пропущенные данные.

In [103]:

```
pd5
```

Out[103]:

	A	B	C	D
0	6	9	2	6
1	7	4	3	7
2	7	2	5	4

In [104]:

```
pd6 = pd.DataFrame(rs.randint(0, 10, (4, 4)), index=list(range(1,5)),
                   columns=['B', 'C', 'D', 'F'])
pd6
```

Out[104]:

	B	C	D	F
1	1	7	5	1
2	4	0	9	5
3	8	0	9	2
4	6	3	8	2

In [105]:

```
sr8 = pd5['A'] + pd6['B'] # выполняется выравнивание по индексам (участвуют две серии)
sr8
```

Out[105]:

```
0     NaN
1     8.0
2    11.0
3     NaN
4     NaN
dtype: float64
```

In [108]:

```
pd7 = pd5 + pd6 # выполняется выравнивание по столбцам и по индексам
pd7
```

Out[108]:

	A	B	C	D	F
0	NaN	NaN	NaN	NaN	NaN
1	NaN	5.0	10.0	12.0	NaN
2	NaN	6.0	5.0	13.0	NaN
3	NaN	NaN	NaN	NaN	NaN
4	NaN	NaN	NaN	NaN	NaN

Работа с пустыми значениями

- к оглавлению

В Pandas в качестве пустых значений рассматривается значение `NaN` ("Not a Number"), поддерживаемое форматом чисел с плавающей точкой (`np.nan` в NumPy) и значением `None` для объектов Python.

In [428]:

```
sr8
```

Out[428]:

```
0     NaN
1     8.0
2    11.0
3     NaN
4     NaN
dtype: float64
```

In [106]:

```
# получение маски пустых значений
sr8.isna()
```

Out[106]:

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

In [109]:

```
pd7.isna()
```

Out[109]:

	A	B	C	D	F
0	True	True	True	True	True
1	True	False	False	False	True
2	True	False	False	False	True
3	True	True	True	True	True
4	True	True	True	True	True

In [431]:

```
# очистка от пустых значений:
sr8.dropna()
```

Out[431]:

```
1     8.0
2    11.0
dtype: float64
```

In [110]:

```
pd7.dropna() # default how='any'
```

Out[110]:

	A	B	C	D	F

In [111]:

```
# default axis=0, удаляем строки, в которых все значения NaN:
pd7.dropna(how='all')
```

Out[111]:

	A	B	C	D	F
1	NaN	5.0	10.0	12.0	NaN
2	NaN	6.0	5.0	13.0	NaN

In [112]:

```
# последовательное применение dropna:
# сначала для строк (т.к. default axis=0),
# потом для столбцов dropna(axis=1), помним: (default how='any'):
pd7.dropna(how='all').dropna(axis=1)
```

Out[112]:

	B	C	D
1	5.0	10.0	12.0
2	6.0	5.0	13.0

In [113]:

```
pd7.fillna(0.0) # заполнение NaN заданными значениями
```

Out[113]:

	A	B	C	D	F
0	0.0	0.0	0.0	0.0	0.0
1	0.0	5.0	10.0	12.0	0.0
2	0.0	6.0	5.0	13.0	0.0
3	0.0	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0	0.0

Агрегирование и группировка

- к оглавлению

In [114]:

```
pd75 = pd.DataFrame({'A': rs.rand(5), 'B': rs.rand(5)})
pd75
```

Out[114]:

	A	B
0	0.859940	0.563288
1	0.680308	0.385417
2	0.450499	0.015966
3	0.013265	0.230894
4	0.942202	0.241025

In [115]:

```
pd75.values
```

Out[115]:

```
array([[0.85994041, 0.56328822],
       [0.68030754, 0.3854165 ],
       [0.45049925, 0.01596625],
       [0.01326496, 0.23089383],
       [0.94220176, 0.24102547]])
```

In [214]:

```
pd75.values.mean(axis=0)
```

Out[214]:

```
array([0.58924278, 0.28731805])
```

In [116]:

```
pd75.values.mean()
```

Out[116]:

```
0.43828041783921934
```

In [439]:

```
# default axis=0, m.e. агрегируем значения вдоль оси 0
# (m.e. при агрегировании меняем индекс элементов вдоль этой оси):
pd75.mean()
```

Out[439]:

```
A    0.407849
B    0.392064
dtype: float64
```

In [117]:

```
pd75.mean(axis=1)
```

Out[117]:

```
0    0.711614
1    0.532862
2    0.233233
3    0.122079
4    0.591614
dtype: float64
```

In [331]:

```
# атрибут values:
pd75.values, type(pd75.values)
```

Out[331]:

```
(array([[0.85994041, 0.56328822],
        [0.68030754, 0.3854165 ],
        [0.45049925, 0.01596625],
        [0.01326496, 0.23089383],
        [0.94220176, 0.24102547]]),
 numpy.ndarray)
```

In [118]:

```
pd.DataFrame({'sum':pd75.sum(), 'prod':pd75.prod(),
              'mean':pd75.mean(), 'median':pd75.median(), 'std':pd75.std(), 'var':pd75.var(
              'min':pd75.min(), 'max':pd75.max()})
```

Out[118]:

	sum	prod	mean	median	std	var	min	max
A	2.946214	0.003294	0.589243	0.680308	0.373212	0.139288	0.013265	0.942202
B	1.436590	0.000193	0.287318	0.241025	0.202942	0.041185	0.015966	0.563288

In [215]:

```
pd75.describe()
```

Out[215]:

	A	B
count	5.000000	5.000000
mean	0.589243	0.287318
std	0.373212	0.202942
min	0.013265	0.015966
25%	0.450499	0.230894
50%	0.680308	0.241025
75%	0.859940	0.385417
max	0.942202	0.563288

In [119]:

```
# квантиль:
pd75.quantile(0.5)
```

Out[119]:

```
A    0.680308
B    0.241025
Name: 0.5, dtype: float64
```

In [443]:

```
pd75.quantile(np.arange(0.0, 1.1, 0.1))
```

Out[443]:

	A	B
0.0	0.139494	0.046450
0.1	0.200554	0.107740
0.2	0.261614	0.169029
0.3	0.306988	0.262586
0.4	0.336675	0.388410
0.5	0.366362	0.514234
0.6	0.402245	0.545506
0.7	0.438128	0.576779
0.8	0.521891	0.595441
0.9	0.653534	0.601493
1.0	0.785176	0.607545

Обработка нескольких наборов данных

Объединение наборов данных

- к оглавлению

In [181]:

```
pd8 = pd.DataFrame([[1, 2], [3, 4]], columns=list('AB'))
pd8
```

Out[181]:

	A	B
0	1	2
1	3	4

In [182]:

```
pd9 = pd.DataFrame([[5, 6], [7, 8]], columns=list('AB'))
pd9
```

Out[182]:

	A	B
0	5	6
1	7	8

In [184]:

```
# append создает новый объект DataFrame:
pd89 = pd8.append(pd9) # при конкатенации может происходить дублирование индекса
pd89
```

Out[184]:

	A	B
0	1	2
1	3	4
0	5	6
1	7	8

In [185]:

```
pd89.index
```

Out[185]:

```
Int64Index([0, 1, 0, 1], dtype='int64')
```

In [186]:

```
# автоматически создается новый индекс:
pd8.append(pd9, ignore_index=True)
```

Out[186]:

	A	B
0	1	2
1	3	4
2	5	6
3	7	8

In [187]:

```
pd8.index
```

Out[187]:

```
RangeIndex(start=0, stop=2, step=1)
```

Функция `pd.merge()` реализует множество типов соединений: «один-к-одному», «многие-к-одному» и «многие-ко-многим». Все эти три типа соединений доступны через один и тот же вызов `pd.merge()`, тип выполняемого соединения зависит от формы входных данных.

In [188]:

```
pd10 = pd.DataFrame({'employee': ['Bob', 'Jake', 'Lisa', 'Sue'],
                     'group': ['Accounting', 'Engineering', 'Engineering', 'HR']})
pd11 = pd.DataFrame({'employee': ['Lisa', 'Bob', 'Jake', 'Sue'],
                     'hire_date': [2004, 2008, 2012, 2014]})
```

In [189]:

```
pd10
```

Out[189]:

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

In [190]:

```
pd11
```

Out[190]:

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

In [191]:

```
# Функция pd.merge() распознает, что в обоих объектах DataFrame имеется столбец
# employee, и автоматически выполняет соединение один-к-одному, используя этот столбец в ка
pd.merge(pd10, pd11)
```

Out[191]:

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

In [192]:

```
pd12 = pd.DataFrame({'group': ['Accounting', 'Engineering', 'HR'],
                     'supervisor': ['Carly', 'Guido', 'Steve']})
pd12
```

Out[192]:

	group	supervisor
0	Accounting	Carly
1	Engineering	Guido
2	HR	Steve

In [193]:

```
# соединение многие-к-одному по столбцу group:
pd.merge(pd10, pd12)
```

Out[193]:

	employee	group	supervisor
0	Bob	Accounting	Carly
1	Jake	Engineering	Guido
2	Lisa	Engineering	Guido
3	Sue	HR	Steve

In [194]:

```
pd13 = pd.DataFrame({'group': ['Accounting', 'Accounting', 'Engineering', 'Engineering', 'H
                               'skills': ['math', 'spreadsheets', 'coding', 'linux', 'spreadshe
pd13
```

Out[194]:

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

In [454]:

```
# соединение многие-ко-многим по столбцу group:
pd.merge(pd10, pd13)
```

Out[454]:

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

Метод `pd.merge()` по умолчанию выполняет поиск в двух входных объектах соответствующих названий столбцов и использует найденное в качестве ключа. Однако, зачастую имена столбцов не совпадают, для этого случая в методе pd.merge() имеются специальные параметры.

- on для явного указания имени (имен) столбцов;
- left_on и right_on для явного указания имен столбцов, в случае, если у первого и второго DataFrame они не совпадают;
- left_index и right_index для указания индекса в качестве ключа слияния.

In [195]:

```
# пнример:
pd14 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
                     'salary': [70000, 80000, 120000, 90000]})
pd14
```

Out[195]:

	name	salary
0	Bob	70000
1	Jake	80000
2	Lisa	120000
3	Sue	90000

In [196]:

```
pd15 = pd.merge(pd10, pd14, left_on='employee', right_on='name')
pd15
```

Out[196]:

	employee	group	name	salary
0	Bob	Accounting	Bob	70000
1	Jake	Engineering	Jake	80000
2	Lisa	Engineering	Lisa	120000
3	Sue	HR	Sue	90000

In [197]:

```
# лишний столбец можно удалить:
pd15.drop('name', axis=1, inplace=True) # inplace=True - не создается новый DataFrame
pd15
```

Out[197]:

	employee	group	salary
0	Bob	Accounting	70000
1	Jake	Engineering	80000
2	Lisa	Engineering	120000
3	Sue	HR	90000

GroupBy: разбиение, применение, объединение

- к оглавлению

Операцию GroupBy удобно представить в виде последовательного применения операций: разбиение, применение и объединение (**split, apply, combine**):

- **split** (шаг разбиения): включает разделение на части и группировку объекта DataFrame на основе значений заданного ключа.
- **apply** (шаг применения): включает вычисление какой-либо функции, обычно агрегирующей, преобразование или фильтрацию в пределах отдельных групп.
- **combine** (шаг объединения): во время шага выполняется слияние результатов предыдущих операций в выходной массив.

Для DataFrame операцию "разбить, применить, объединить" можно реализовать с помощью метода groupby(), передав в него имя желаемого ключевого столбца. Функция groupby() возвращает не набор объектов DataFrame, а объект DataFrameGroupBy, который можно рассматривать как специальное представление объекта DataFrame, готовое к группировке, но не выполняющее никаких фактических вычислений до этапа применения агрегирования (используется принцип отложенных вычислений).

Для получения результата нужно вызвать один из агрегирующих методов объекта DataFrameGroupBy, что приведет к выполнению соответствующих шагов применения/объединения.

In [198]:

```
pd16 = pd.DataFrame({'key': ['A', 'B', 'C', 'A', 'B', 'C'],
                     'data': range(1, 7)}, columns=['key', 'data'])
```

In [458]:

```
pd16
```

Out[458]:

	key	data
0	A	1
1	B	2
2	C	3
3	A	4
4	B	5
5	C	6

In [199]:

```
pd16.groupby('key')
```

Out[199]:

```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x0000021722143308>
```

In [200]:

```
pd16.groupby('key').sum()
```

Out[200]:

key	data
A	5
B	7
C	9

In [201]:

```
# загружаем набор данных об открытии экзопланет:
import seaborn as sns
planets = sns.load_dataset('planets')
planets.shape
```

Out[201]:

```
(1035, 6)
```

In [202]:

```
# заголовок таблицы
planets.head()
```

Out[202]:

	method	number	orbital_period	mass	distance	year
0	Radial Velocity	1	269.300	7.10	77.40	2006
1	Radial Velocity	1	874.774	2.21	56.95	2008
2	Radial Velocity	1	763.000	2.60	19.84	2011
3	Radial Velocity	1	326.030	19.40	110.62	2007
4	Radial Velocity	1	516.220	10.50	119.47	2009

In [203]:

```
# подсчитываем количество не NaN значений в каждой группе:
planets.groupby('year').count()
```

Out[203]:

year	method	number	orbital_period	mass	distance
1989	1	1	1	1	1
1992	2	2	2	0	0
1994	1	1	1	0	0
1995	1	1	1	1	1
1996	6	6	6	4	6
1997	1	1	1	1	1
1998	5	5	5	5	5
1999	15	15	15	14	15
2000	16	16	16	14	16
2001	12	12	12	11	12
2002	32	32	32	31	31
2003	25	25	25	22	24
2004	26	26	22	15	23
2005	39	39	38	34	37
2006	31	31	28	20	28
2007	53	53	52	32	45
2008	74	74	69	43	64
2009	98	98	96	74	83
2010	102	102	96	41	93
2011	185	185	184	91	155
2012	140	140	132	24	91
2013	118	118	107	30	71
2014	52	52	51	5	6

In [204]:

```
# группировка экзопланет по методу их идентификации:
planets.groupby('method').count()
```

Out[204]:

method	number	orbital_period	mass	distance	year
Astrometry	2	2	0	2	2
Eclipse Timing Variations	9	9	2	4	9
Imaging	38	12	0	32	38
Microlensing	23	7	0	10	23
Orbital Brightness Modulation	3	3	0	2	3
Pulsar Timing	5	5	0	1	5
Pulsation Timing Variations	1	1	0	0	1
Radial Velocity	553	553	510	530	553
Transit	397	397	1	224	397
Transit Timing Variations	4	3	0	3	4

In [205]:

```
# сколько орбитальных периодов было обнаружено каждым из методов:
planets.groupby('method')['orbital_period'].count()
```

Out[205]:

```
method
Astrometry                        2
Eclipse Timing Variations         9
Imaging                          12
Microlensing                      7
Orbital Brightness Modulation     3
Pulsar Timing                     5
Pulsation Timing Variations       1
Radial Velocity                 553
Transit                         397
Transit Timing Variations         3
Name: orbital_period, dtype: int64
```

In [206]:

```
# медианное значение орбитальных периодов (в днях), выявленных каждым из методов:
planets.groupby('method')['orbital_period'].median()
```

Out[206]:

```
method
Astrometry                         631.180000
Eclipse Timing Variations         4343.500000
Imaging                          27500.000000
Microlensing                      3300.000000
Orbital Brightness Modulation        0.342887
Pulsar Timing                       66.541900
Pulsation Timing Variations       1170.000000
Radial Velocity                    360.200000
Transit                              5.714932
Transit Timing Variations           57.011000
Name: orbital_period, dtype: float64
```

In [207]:

```
# по группам, выделенным с помощью groupby, можно итерироваться:
for (method, group) in planets.groupby('method'): # тип group - DataFrame
    print(f"{method} shape={group.shape}")
```

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

На этапе применения у объектов GroupBy кроме обычных агрегирующих методов, таких как sum(), median() и т. п., имеются методы aggregate(), filter(), transform() и apply(), эффективно выполняющие множество полезных операций до объединения сгруппированных данных.

Метод aggregate() может принимать на входе строку, функцию или список и вычислять все сводные показатели сразу.

In [208]:

```
planets.groupby('method')['orbital_period'].aggregate(['min', np.median, max])
```

Out[208]:

method	min	median	max
Astrometry	246.360000	631.180000	1016.000000
Eclipse Timing Variations	1916.250000	4343.500000	10220.000000
Imaging	4639.150000	27500.000000	730000.000000
Microlensing	1825.000000	3300.000000	5100.000000
Orbital Brightness Modulation	0.240104	0.342887	1.544929
Pulsar Timing	0.090706	66.541900	36525.000000
Pulsation Timing Variations	1170.000000	1170.000000	1170.000000
Radial Velocity	0.736540	360.200000	17337.500000
Transit	0.355000	5.714932	331.600590
Transit Timing Variations	22.339500	57.011000	160.000000

Операция фильтрации `filter` дает возможность опускать данные в зависимости от свойств группы. Например, нам может понадобиться оставить в результате все группы

In [218]:

```
def filter_func(x):
    return x['orbital_period'].max()/x['orbital_period'].min() > 10000
```

In [219]:

```
gr1 = planets.groupby('method').filter(filter_func)
gr1.shape
```

Out[219]:

```
(558, 6)
```

В то время как агрегирующая функция должна возвращать сокращенную версию данных, преобразование `transform` может вернуть версию полного набора данных, преобразованную ради дальнейшей их перекомпоновки. При подобном преобразовании форма выходных данных совпадает с формой входных. Распространенный пример - центрирование данных путем вычитания среднего значения по группам.

In [222]:

```
planets['cntr_orbital_period'] = \
planets.groupby('method')['orbital_period'].transform(lambda x: x - x.mean())
planets[:15]
```

Out[222]:

	method	number	orbital_period	mass	distance	year	cntr_orbital_period
0	Radial Velocity	1	269.300	7.10	77.40	2006	-554.05468
1	Radial Velocity	1	874.774	2.21	56.95	2008	51.41932
2	Radial Velocity	1	763.000	2.60	19.84	2011	-60.35468
3	Radial Velocity	1	326.030	19.40	110.62	2007	-497.32468
4	Radial Velocity	1	516.220	10.50	119.47	2009	-307.13468
5	Radial Velocity	1	185.840	4.80	76.39	2008	-637.51468
6	Radial Velocity	1	1773.400	4.64	18.15	2002	950.04532
7	Radial Velocity	1	798.500	NaN	21.41	1996	-24.85468
8	Radial Velocity	1	993.300	10.30	73.10	2008	169.94532
9	Radial Velocity	2	452.800	1.99	74.79	2010	-370.55468
10	Radial Velocity	2	883.000	0.86	74.79	2010	59.64532
11	Radial Velocity	1	335.100	9.88	39.43	2009	-488.25468
12	Radial Velocity	1	479.100	3.88	97.28	2008	-344.25468
13	Radial Velocity	3	1078.000	2.53	14.08	1996	254.64532
14	Radial Velocity	3	2391.000	0.54	14.08	2001	1567.64532

Метод `apply()` позволяет применять произвольную функцию к результатам группировки. В качестве параметра эта функция должна получать объект DataFrame, а возвращать или объект библиотеки Pandas (например, DataFrame, Series), или скалярное значение, в зависимости от возвращаемого значения будет вызвана соответствующая операция объединения.

In [224]:

```
def norm_by_min_in_year(x):
    # x - объект DataFrame сгруппированных значений
    x['orbital_period_normalized'] = x['orbital_period']/x['orbital_period'].min()
    return x
```

In [225]:

```
planets.groupby('year').apply(norm_by_min_in_year)
```

Out[225]:

	method	number	orbital_period	mass	distance	year	cntr_orbital_period	orbital_period
0	Radial Velocity	1	269.300000	7.10	77.40	2006	-554.054680	
1	Radial Velocity	1	874.774000	2.21	56.95	2008	51.419320	
2	Radial Velocity	1	763.000000	2.60	19.84	2011	-60.354680	
3	Radial Velocity	1	326.030000	19.40	110.62	2007	-497.324680	
4	Radial Velocity	1	516.220000	10.50	119.47	2009	-307.134680	
...	...	...	...	...	...	...	...	
1030	Transit	1	3.941507	NaN	172.00	2006	-17.160566	
1031	Transit	1	2.615864	NaN	148.00	2007	-18.486209	
1032	Transit	1	3.191524	NaN	174.00	2007	-17.910549	
1033	Transit	1	4.125083	NaN	293.00	2008	-16.976990	
1034	Transit	1	4.187757	NaN	260.00	2008	-16.914316	

1035 rows × 8 columns

Спасибо за внимание!

Технический раздел

next qs line
next an line
next df line
next ex line
next pl line
next mn line
next plmn line
next hn line

- Home

- News **red** and **green** and **blue** and __selected__
- A ⇒ b ⇒ c ‣ Contact
- ≂ ~ ≈ ± About

- **Def:** Определение
- **Ex:** пример (кейс)
- **Q:** вопрос (проблема)
- **A:** ответ
- Алгоритм:
  - **S1:** Шаг 1
  - **S2:** Шаг 2
- Свойства:
  - **P1:** Свойство 1
  - **P2:** Свойство 2
- Утверждение
  - ⇒ следствие
- Свойства:
  - **+** положительные
  - **-** отрицательные
  - **±** смешанные