



РАБОТА С БИБЛИОТЕКОЙ PANDAS ДЛЯ ОБРАБОТКИ И АНАЛИЗА ДАННЫХ:

- ❖ работа с таблицами,
- ❖ фильтрация,
- ❖ сортировка,
- ❖ группировка,
- ❖ объединение данных

ПОЛЕЗНЫЕ ССЫЛКИ

- <https://habr.com/ru/companies/ruvds/articles/494720/>
- https://pandas.pydata.org/docs/user_guide/index.html (включить переводчик)
- Уэс Маккинни Python и анализ данных: Первичная обработка данных с применением pandas, NumPy и Jupiter / пер. с англ. А. А. Слинкина. 3-е изд. — М.: МК Пресс, 2023. — 536 с.: ил.
- Бейдер Дэн, Эймос Дэвид, Яблонски Джоанна, Хейслер Флетчер Знакомство с Python. — СПб.: Питер, 2023. — 512 с.: ил. — (Серия «Библиотека программиста»).

PANDAS

- Pandas — это одна из самых популярных библиотек в экосистеме Python, предназначенная для обработки, анализа и подготовки данных.

Она предоставляет мощные структуры данных, такие как **Series** и **DataFrame**, которые позволяют легко работать с табличной информацией.

Pandas активно используется в:

- анализе и визуализации данных;
- машинном обучении;
- финансовой аналитике;
- подготовке отчетов;
- обработке больших объемов CSV- или Excel-файлов.

С помощью Pandas можно легко загружать, фильтровать, трансформировать и визуализировать данные — всё это с минимальным количеством кода.

PANDAS

Установка и подключение Pandas

- Чтобы начать работу с Pandas, сначала установим его:

```
pip install pandas
```

А затем импортируем в ваш Python-скрипт:

```
import pandas as pd
```

СТРУКТУРЫ ДАННЫХ PANDAS

Две основные структуры данных: **Series** и **DataFrame**

Series – одномерный похожий на массив объект, содержащий последовательность данных и ассоциированный с ним массив меток, который называется индексом.

Т.е. это массив с метками (индексами), который может хранить любые типы данных

```
s = pd.Series([10, 20, 30], index=["a", "b", "c"])
```

Простейший объект Series состоит только из массива данных:

```
In [14]: obj = pd.Series([4, 7, -5, 3])

In [15]: obj
Out[15]:
0    4
1    7
2   -5
3    3
dtype: int64
```

Series - объект, который записывается

в интерактивном режиме: индекс находится слева, а значения – справа

ВНИМАНИЕ! Если не задали индекс для данных, то по умолчанию создается индекс, состоящий из целых чисел от 0 до N – 1 (где N – длина массива данных)

Для объекта **Series** получить представление самого массива и его индекса можно с помощью атрибутов

values и **index** соответственно:

```
In [16]: obj.array
Out[16]:
<PandasArray>
[4, 7, -5, 3]
Length: 4, dtype: int64
```

```
In [17]: obj.index
Out[17]: RangeIndex(start=0, stop=4, step=1))
```

Результатом применения атрибута **.array** является объект **PandasArray**

ОБЪЕКТ SERIES

Создать объект **Series** с индексом, идентифицирующим каждый элемент данных:

```
In [18]: obj2 = pd.Series([4, 7, -5, 3], index=["d", "b", "a", "c"])

In [19]: obj2
Out[19]:
d    4
b    7
a   -5
c    3
dtype: int64

In [20]: obj2.index
Out[20]: Index(['d', 'b', 'a', 'c'], dtype='object')
```

Для выделения одного или нескольких значений можно использовать метки в индексе:

```
In [21]: obj2["a"]
Out[21]: -5

In [22]: obj2["d"] = 6

In [23]: obj2[["c", "a", "d"]]
Out[23]:
c    3
a   -5
d    6
dtype: int64
```

['c', 'a', 'd'] интерпретируется как список индексов, хотя он содержит не целые числа, а строки

ОБЪЕКТ SERIES

Фильтрация с помощью булева массива, скалярное умножение или применение математических функций, сохраняют связь между индексом и значением (и др.функции):

```
In [24]: obj2[obj2 > 0]
```

```
Out[24]:
```

```
d    6
```

```
b    7
```

```
c    3
```

```
dtype: int64
```

```
In [25]: obj2 * 2
```

```
Out[25]:
```

```
d    12
```

```
b    14
```

```
a   -10
```

```
c     6
```

```
dtype: int64
```

```
In [26]: import numpy as np
```

```
In [27]: np.exp(obj2)
```

```
Out[27]:
```

```
d    403.428793
```

```
b    1096.633158
```

```
a     0.006738
```

```
c    20.085537
```

```
dtype: float64
```

Объект **Series** можно также представлять себе как упорядоченный словарь фиксированной длины, поскольку он отображает индекс на данные. Его можно передавать многим функциям, ожидающим получить словарь:

```
In [28]: "b" in obj2
```

```
Out[28]: True
```

```
In [29]: "e" in obj2
```

```
Out[29]: False
```

ОБЪЕКТ SERIES

Если имеется словарь Python, содержащий данные, то из него можно создать объект **Series**:

```
In [30]: sdata = {"Ohio": 35000, "Texas": 71000, "Oregon": 16000, "Utah": 5000}

In [31]: obj3 = pd.Series(sdata)

In [32]: obj3
Out[32]:
Ohio      35000
Texas     71000
Oregon    16000
Utah       5000
dtype: int64
```

Объект Series можно преобразовать обратно в словарь методом **to_dict**:

```
In [33]: obj3.to_dict()
Out[33]: {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}
```


ОБЪЕКТ SERIES

Если передается только словарь, то в индексе получившегося объекта **Series** ключи будут храниться в порядке, который определяется методом словаря **keys** и зависит от того, в каком порядке ключи вставлялись. Этот порядок можно переопределить, передав индекс, содержащий ключи словаря в том порядке, в каком они должны находиться в результирующем объекте

Series:

```
In [34]: states = ["California", "Ohio", "Oregon", "Texas"]
```

```
In [35]: obj4 = pd.Series(sdata, index=states)
```

```
In [36]: obj4
```

```
Out[36]:
```

California	NaN
Ohio	35000.0
Oregon	16000.0
Texas	71000.0

dtype: float64

В данном случае три значения, найденных в **sdata**, помещены в соответствующие им позиции, а для метки 'California' никакого значения не нашлось, по этому ей соответствует признак **NaN** (не число), которым в pandas обозначаются отсутствующие значения. Поскольку строки '**Utah**' не было в списке **states**, то ее нет и в результирующем объекте.

ОБЪЕКТ SERIES

Отсутствующие данные - «NA» и «null» как синонимы. Для распознавания отсутствующих данных в pandas следует использовать функции **isna** и **notna**:

```
In [37]: pd.isna(obj4)
Out[37]:
California    True
Ohio          False
Oregon        False
Texas         False
dtype: bool
```

```
In [38]: pd.notna(obj4)
Out[38]:
California    False
Ohio          True
Oregon        True
Texas         True
dtype: bool
```

у объекта **Series** есть также методы экземпляра:

```
In [39]: obj4.isna()
Out[39]:
California    True
Ohio          False
Oregon        False
Texas         False
dtype: bool
```

При выполнении арифметических операций объект **Series** автоматически выравнивает данные по индексам:

```
In [40]: obj3
Out[40]:
Ohio          35000
Texas         71000
Oregon        16000
Utah           5000
dtype: int64
```

```
In [41]: obj4
Out[41]:
California    NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
dtype: float64
```

```
In [42]: obj3 + obj4
Out[42]:
California    NaN
Ohio          70000.0
Oregon        32000.0
Texas        142000.0
Utah          NaN
dtype: float64
```

ОБЪЕКТ SERIES

У самого объекта **Series**, и у его индекса имеется атрибут **name**, тесно связанный с другими частями функциональности pandas:

```
In [43]: obj4.name = "population"
```

```
In [44]: obj4.index.name = "state"
```

```
In [45]: obj4
```

```
Out[45]:
```

```
state
```

```
California      NaN
```

```
Ohio            35000.0
```

```
Oregon          16000.0
```

```
Texas           71000.0
```

```
Name: population, dtype: float64
```

Индекс объекта **Series** можно изменить на месте с помощью присваивания:

```
In [46]: obj
```

```
Out[46]:
```

```
0      4
```

```
1      7
```

```
2     -5
```

```
3      3
```

```
dtype: int64
```

```
In [47]: obj.index = ["Bob", "Steve", "Jeff", "Ryan"]
```

```
In [48]: obj
```

```
Out[48]:
```

```
Bob      4
```

```
Steve    7
```

```
Jeff    -5
```

```
Ryan     3
```

```
dtype: int64
```

ОБЪЕКТ DATAFRAME

Объект **DataFrame** представляет табличную структуру данных, состоящую из упорядоченной коллекции столбцов, причем типы значений (числовой, строковый, булев и т. д.) в разных столбцах могут различаться. Т.е это двумерная таблица, где строки и столбцы имеют метки.

В объекте **DataFrame** хранятся два индекса: по строкам и по столбцам. Можно считать, что это словарь объектов **Series**, имеющих общий индекс.

Один из самых распространенных способов сконструировать объект **DataFrame** – на основе словаря списков одинаковой длины или массивов NumPy (рассмотрим в следующей лекции)

```
data = {"state": ["Ohio", "Ohio", "Ohio", "Nevada", "Nevada", "Nevada"],
        "year": [2000, 2001, 2002, 2001, 2002, 2003],
        "pop": [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]}
frame = pd.DataFrame(data)
```

Получившемуся **DataFrame** автоматически будет назначен индекс, как и в случае **Series**, и столбцы расположатся в порядке, определяемом порядком ключей в **data** (который зависит от того, в каком порядке ключи вставлялись в словарь):

```
In [50]: frame
Out[50]:
```

	state	year	pop
0	Ohio	2000	1.5
1	Ohio	2001	1.7
2	Ohio	2002	3.6
3	Nevada	2001	2.4
4	Nevada	2002	2.9
5	Nevada	2003	3.2

```
In [19]: frame
```

```
Out[19]:
```

	state	year	pop
0	Ohio	2000	1.5
1	Ohio	2001	1.7
2	Ohio	2002	3.6
3	Nevada	2001	2.4
4	Nevada	2002	2.9
5	Nevada	2003	3.2

ОБЪЕКТ DATAFRAME

Для больших объектов DataFrame метод **head** отбирает только первые пять строк:

```
In [51]: frame.head()
Out[51]:
```

	state	year	pop
0	Ohio	2000	1.5
1	Ohio	2001	1.7
2	Ohio	2002	3.6
3	Nevada	2001	2.4
4	Nevada	2002	2.9

Если запросить столбец, которого нет в **data**, то он будет заполнен значениями **NaN**:

tail возвращает последние пять строк:

```
In [52]: frame.tail()
Out[52]:
```

	state	year	pop
1	Ohio	2001	1.7
2	Ohio	2002	3.6
3	Nevada	2001	2.4
4	Nevada	2002	2.9
5	Nevada	2003	3.2

Если задать последовательность столбцов, то столбцы DataFrame расположатся строго в указанном порядке:

```
In [53]: pd.DataFrame(data, columns=["year", "state", "pop"])
Out[53]:
```

	year	state	pop
0	2000	Ohio	1.5
1	2001	Ohio	1.7
2	2002	Ohio	3.6
3	2001	Nevada	2.4
4	2002	Nevada	2.9
5	2003	Nevada	3.2

```
In [54]: frame2 = pd.DataFrame(data, columns=["year", "state", "pop", "debt"])
```

```
In [55]: frame2
Out[55]:
```

	year	state	pop	debt
0	2000	Ohio	1.5	NaN
1	2001	Ohio	1.7	NaN
2	2002	Ohio	3.6	NaN
3	2001	Nevada	2.4	NaN
4	2002	Nevada	2.9	NaN
5	2003	Nevada	3.2	NaN

```
In [56]: frame2.columns
Out[56]: Index(['year', 'state', 'pop', 'debt'], dtype='object')
```


ОБЪЕКТ DATAFRAME

Столбец DataFrame

можно извлечь как **объект Series**, воспользовавшись нотацией словарей, или с помощью атрибута:

```
In [57]: frame2["state"]
Out[57]:
0    Ohio
1    Ohio
2    Ohio
3  Nevada
4  Nevada
5  Nevada
Name: state, dtype: object
```

```
In [58]: frame2.year
Out[58]:
0    2000
1    2001
2    2002
3    2001
4    2002
5    2003
Name: year, dtype: int64
```

Строки также можно извлечь по позиции или по имени с помощью специальных атрибутов **iloc** и **loc** :

```
In [59]: frame2.loc[1]
Out[59]:
year    2001
state   Ohio
pop      1.7
debt    NaN
Name: 1, dtype: object
```

```
In [60]: frame2.iloc[2]
Out[60]:
year    2002
state   Ohio
pop      3.6
debt    NaN
Name: 2, dtype: object
```

Столбцы можно модифицировать путем присваивания. Например, пустому столбцу **debt** можно было бы присвоить скалярное значение или массив значений:

```
In [61]: frame2["debt"] = 16.5
```

```
In [62]: frame2
Out[62]:
```

	year	state	pop	debt
0	2000	Ohio	1.5	16.5
1	2001	Ohio	1.7	16.5
2	2002	Ohio	3.6	16.5
3	2001	Nevada	2.4	16.5
4	2002	Nevada	2.9	16.5
5	2003	Nevada	3.2	16.5

```
In [63]: frame2["debt"] = np.arange(6.)
```

```
In [64]: frame2
Out[64]:
```

	year	state	pop	debt
0	2000	Ohio	1.5	0.0
1	2001	Ohio	1.7	1.0
2	2002	Ohio	3.6	2.0
3	2001	Nevada	2.4	3.0
4	2002	Nevada	2.9	4.0
5	2003	Nevada	3.2	5.0

ОБЪЕКТ DATAFRAME

Когда столбцу присваивается список или массив, длина значения должна совпадать с длиной DataFrame. Если же присваивается объект Series, то его метки будут точно выровнены с индексом DataFrame, а в «дырки» будут вставлены значения NA:

```
In [65]: val = pd.Series([-1.2, -1.5, -1.7], index=["two", "four", "five"])
```

```
In [66]: frame2["debt"] = val
```

```
In [67]: frame2
```

```
Out[67]:
```

	year	state	pop	debt
0	2000	Ohio	1.5	NaN
1	2001	Ohio	1.7	NaN
2	2002	Ohio	3.6	NaN
3	2001	Nevada	2.4	NaN
4	2002	Nevada	2.9	NaN
5	2003	Nevada	3.2	NaN

Затем для удаления этого столбца используем метод **del**:

```
In [70]: del frame2["eastern"]
```

```
In [71]: frame2.columns
```

```
Out[71]: Index(['year', 'state', 'pop', 'debt'], dtype='object')
```

Присваивание несуществующему столбцу приводит к созданию нового столбца.

Для удаления столбцов служит ключевое слово **del**, как и в обычном словаре. Для демонстрации работы **del** сначала добавим новый столбец булевых значений, показывающих, находится ли в столбце **state** значение "Ohio".

```
In [68]: frame2["eastern"] = frame2["state"] == "Ohio"
```

```
In [69]: frame2
```

```
Out[69]:
```

	year	state	pop	debt	eastern
0	2000	Ohio	1.5	NaN	True
1	2001	Ohio	1.7	NaN	True
2	2002	Ohio	3.6	NaN	True
3	2001	Nevada	2.4	NaN	False
4	2002	Nevada	2.9	NaN	False
5	2003	Nevada	3.2	NaN	False

ОБЪЕКТ DATAFRAME

Еще одна распространенная форма данных – словарь словарей:

```
In [72]: populations = {"Ohio": {2000: 1.5, 2001: 1.7, 2002: 3.6},  
.....:                 "Nevada": {2001: 2.4, 2002: 2.9}}
```

Если передать вложенный словарь объекту DataFrame, то pandas интерпретирует ключи внешнего словаря как столбцы, а ключи внутреннего словаря – как индексы строк:

```
In [73]: frame3 = pd.DataFrame(populations)
```

```
In [74]: frame3
```

```
Out[74]:
```

	Ohio	Nevada
2000	1.5	NaN
2001	1.7	2.4
2002	3.6	2.9

Ключи внутренних словарей объединяются для образования индекса результата. Однако этого не происходит, если индекс задан явно:

Объект DataFrame можно транспонировать (переставить местами строки и столбцы)

```
In [75]: frame3.T
```

```
Out[75]:
```

	2000	2001	2002
Ohio	1.5	1.7	3.6
Nevada	NaN	2.4	2.9

```
In [76]: pd.DataFrame(populations, index=[2001, 2002, 2003])
```

```
Out[76]:
```

	Ohio	Nevada
2001	1.7	2.4
2002	3.6	2.9
2003	NaN	NaN

ОБЪЕКТ DATAFRAME - АРГУМЕНТЫ КОНСТРУКТОРА

DATAFRAME

Тип	Примечания
Двумерный ndarray	Матрица данных, дополнительно можно передать метки строк и столбцов
Словарь массивов, списков или кортежей	Каждая последовательность становится столбцом объекта DataFrame. Все последовательности должны быть одинаковой длины
Структурированный массив NumPy	Интерпретируется так же, как «словарь массивов»
Словарь объектов Series	Каждое значение становится столбцом. Если индекс явно не задан, то индексы объектов Series объединяются и образуют индекс строк результата
Словарь словарей	Каждый внутренний словарь становится столбцом. Ключи объединяются и образуют индекс строк, как в случае «словаря объектов Series»
Список словарей или объектов Series	Каждый элемент списка становится строкой объекта DataFrame. Объединение ключей словаря или индексов объектов Series становится множеством меток столбцов DataFrame
Список списков или кортежей	Интерпретируется так же, как «двумерный ndarray»
Другой объект DataFrame	Используются индексы DataFrame, если явно не заданы другие индексы
Объект NumPy MaskedArray	Как «двумерный ndarray», с тем отличием, что замаскированные значения становятся отсутствующими в результирующем объекте DataFrame

ЗАГРУЗКА И СОХРАНЕНИЕ ДАННЫХ

Загрузка из CSV

```
df_csv = pd.read_csv("data.csv")
```

Загрузка из Excel

```
df_excel = pd.read_excel("data.xlsx")
```

Загрузка из JSON

```
df_json = pd.read_json("data.json")
```

После обработки загруженных данных требуется их сохранить в файл.

Сохранение данных в файл

Сохранение в CSV

```
df.to_csv("output.csv", index=False)
```

Сохранение в Excel

```
df.to_excel("output.xlsx")
```

ПРИМЕР ЗАГРУЗКИ ДАННЫХ

Импорт библиотеки Pandas;

Загрузка и вывод информации из датасета о Титанике в формате файла CSV

Получение данных из файла формата Excel

(df = pd.read_excel)

```
import pandas as pd

# URL файла с GitHub
url = 'https://raw.githubusercontent.com/datasciencedojo/datasets/master/titanic.csv'

# Загрузка данных из CSV
df = pd.read_csv(url)

# Просмотр первых 5 строк
print(df.head())

# Основная информация о данных
print(df.info())

# Статистика по числовым столбцам
print(df.describe())
```

```
PassengerId  Survived  Pclass  \
0            1         0       3
1            2         1       1
2            3         1       3
3            4         1       1
4            5         0       3

Name      Sex  Age  SibSp  \
0  Braund, Mr. Owen Harris    male  22.0      1
1  Cumings, Mrs. John Bradley (Florence Briggs Th...  female  38.0      1
2  Heikkinen, Miss. Laina    female  26.0      0
3  Futrelle, Mrs. Jacques Heath (Lily May Peel)  female  35.0      1
4  Allen, Mr. William Henry    male  35.0      0

Parch  Ticket  Fare  Cabin  Embarked
0      0   A/5 21171   7.2500   NaN      S
1      0   PC 17599  71.2833   C85      C
2      0  STON/O2. 3101282   7.9250   NaN      S
3      0  113803   53.1000  C123      S
4      0  373450   8.0500   NaN      S

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 12 columns):
#   Column             Non-Null Count  Dtype
---  -
0   PassengerId        891 non-null    int64
1   Survived           891 non-null    int64
2   Pclass              891 non-null    int64
3   Name                891 non-null    object
4   Sex                 891 non-null    object
5   Age                 714 non-null    float64
6   SibSp               891 non-null    int64
7   Parch              891 non-null    int64
8   Ticket              891 non-null    object
9   Fare                891 non-null    float64
10  Cabin               204 non-null    object
11  Embarked            889 non-null    object
dtypes: float64(2), int64(5), object(5)
memory usage: 83.7+ KB

None

PassengerId  Survived  Pclass  Age  SibSp  \
count  891.000000  891.000000  891.000000  714.000000  891.000000
mean    446.000000    0.383838    2.308642   29.699118    0.523008
std    257.353842    0.486592    0.836071   14.526497    1.102743
min      1.000000    0.000000    1.000000    0.420000    0.000000
25%    223.500000    0.000000    2.000000   20.125000    0.000000
50%    446.000000    0.000000    3.000000   28.000000    0.000000
75%    668.500000    1.000000    3.000000   38.000000    1.000000
max    891.000000    1.000000    3.000000   80.000000    8.000000

Parch  Fare
count  891.000000  891.000000
mean     0.381594   32.204208
std     0.806057   49.693429
min     0.000000    0.000000
25%     0.000000    7.910400
50%     0.000000   14.454200
75%     0.000000   31.000000
```


ИНДЕКСАЦИЯ, ФИЛЬТРАЦИЯ И ВЫБОР ДАННЫХ

Логическая фильтрация строк –
можно фильтровать данные
в соответствии с
определенными
условиями

```
df[df["Возраст"] > 25]
```

```
# Выбор продуктов с продажами выше среднего  
high_sales_products = df[df['Sales'] > mean_sales]
```

Выбор по метке и индексу (методы loc и iloc)

loc - выбор по
метке

```
df.loc[0, "Имя"]
```

iloc - Выбор по индексу

```
df.iloc[0, 1]
```


ФИЛЬТРАЦИЯ ДАННЫХ

```
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eve'],
    'Age': [25, 30, 35, 40, 25],
    'Salary': [50000, 60000, 70000, 80000, 45000],
    'Department': ['HR', 'IT', 'IT', 'Finance', 'HR']
}
df = pd.DataFrame(data)

# Фильтр по возрасту > 30
filtered = df[df['Age'] > 30]

# Фильтр по нескольким условиям
it_employees = df[(df['Department'] == 'IT') & (df['Salary'] > 65000)]

# Фильтр с isin()
hr_finance = df[df['Department'].isin(['HR', 'Finance'])]

print(filtered)
print("")
print(it_employees)
```

```
[<]>
   Name  Age  Salary Department
2  Charlie  35   70000         IT
3   David  40   80000      Finance
```

```
   Name  Age  Salary Department
2  Charlie  35   70000         IT
```

ИНДЕКСАЦИЯ, ФИЛЬТРАЦИЯ И ВЫБОР ДАННЫХ

Работа с пропущенными значениями

Отсутствующие данные – типичное явление в большинстве аналитических приложений.

Например, при вычислении всех описательных статистик для объектов `pandas` отсутствующие данные не учитываются.

Для представления отсутствующих данных типа `float64` в `pandas` используется значение с плавающей точкой `NaN` (не число), которое называют маркером.

Метод `isna` дает булев объект `Series`, в котором элементы `True` соответствуют

```
OT In [16]: float_data.isna()
ЗН Out[16]:
0      False
1      False
2       True
3      False
dtype: bool
```

Методы обработки отсутствующих данных

Метод	Описание
<code>dropna</code>	Фильтрует метки оси в зависимости от того, существуют ли для метки отсутствующие данные, причем есть возможность указать различные пороги, определяющие, какое количество отсутствующих данных считать допустимым
<code>fillna</code>	Восполняет отсутствующие данные указанным значением или использует какой-нибудь метод интерполяции, например <code>"ffill"</code> или <code>"bfill"</code>
<code>isna</code>	Возвращает объект, содержащий булевы значения, которые показывают, какие значения отсутствуют
<code>notna</code>	Логическое отрицание <code>isna</code> ; возвращает <code>True</code> для присутствующих и <code>False</code> для отсутствующих значений

ИНДЕКСАЦИЯ, ФИЛЬТРАЦИЯ И ВЫБОР ДАННЫХ

Работа с пропущенными значениями - фильтрация отсутствующих данных

Поиск пропущенных (пустых) значений массива Удаление пропущенных (пустых) значений массива

`df.isnull()` # найти NaN

`df.dropna()` # удалить строки с NaN

Для объектов Series

```
In [26]: data = pd.DataFrame([[1., 6.5, 3.], [1., np.nan, np.nan],  
.....:                      [np.nan, np.nan, np.nan], [np.nan, 6.5, 3.]])
```

```
In [27]: data
```

```
Out[27]:
```

	0	1	2
0	1.0	6.5	3.0
1	1.0	NaN	NaN
2	NaN	NaN	NaN
3	NaN	6.5	3.0

```
In [28]: data.dropna()
```

```
Out[28]:
```

0	1	2	
0	1.0	6.5	3.0

Для объектов DataFrame

```
In [23]: data = pd.Series([1, np.nan, 3.5, np.nan, 7])
```

```
In [24]: data.dropna()
```

```
Out[24]:
```

0	1.0
2	3.5
4	7.0

dtype: float64

ИНДЕКСАЦИЯ, ФИЛЬТРАЦИЯ И ВЫБОР ДАННЫХ

Работа с пропущенными значениями - фильтрация отсутствующих данных

Заполнение NaN – установка значений в пустые элементы

```
df.fillna(0) # заменить
```

```
NaN на 0
```

```
In [39]: df.fillna(0)
Out[39]:
```

	0	1	2
0	-0.204708	0.000000	0.000000
1	-0.555730	0.000000	0.000000
2	0.092908	0.000000	0.769023
3	1.246435	0.000000	-1.296221
4	0.274992	0.228913	1.352917
5	0.886429	-2.001637	-0.371843
6	1.669025	-0.438570	-0.539741

fillna умеет, например восполнять данные, подставляя среднее или медиану:

```
In [47]: data = pd.Series([1., np.nan, 3.5, np.nan, 7])
```

```
In [48]: data.fillna(data.mean())
```

```
Out[48]:
```

```
0    1.000000
1    3.833333
2    3.500000
3    3.833333
4    7.000000
dtype: float64
```

Аргументы метода **fillna**

Аргумент	Описание
value	Скалярное значение или похожий на словарь объект для восполнения отсутствующих значений
method	Метод интерполяции: "bfill" (обратное восполнение) или "ffill" (прямое восполнение). По умолчанию None
axis	Ось, по которой производится восполнение ("index" или "columns"); по умолчанию axis="index"
limit	Для прямого и обратного восполнений максимальное количество непрерывно следующих друг за другом промежутков, подлежащих восполнению

ОБРАБОТКА ПРОПУЩЕННЫХ ЗНАЧЕНИЙ

```
import numpy as np
df_missing = pd.DataFrame({
    'A': [1, 2, np.nan, 4],
    'B': [5, np.nan, np.nan, 8],
    'C': [10, 20, 30, 40]
})

# Удаление строк с пропусками
df_dropped = df_missing.dropna()

# Замена пропусков
df_filled = df_missing.fillna({
    'A': df_missing['A'].mean(),
    'B': 0
})

# Интерполяция
df_interpolated = df_missing.interpolate()
```

```
print(df_missing)
print("")
print(df_dropped)
print("")
print(df_interpolated)
```

⇒

	A	B	C
0	1.0	5.0	10
1	2.0	NaN	20
2	NaN	NaN	30
3	4.0	8.0	40

	A	B	C
0	1.0	5.0	10
3	4.0	8.0	40

	A	B	C
0	1.0	5.0	10
1	2.0	6.0	20
2	3.0	7.0	30
3	4.0	8.0	40

ГРУППИРОВКА И АГРЕГАЦИЯ: МЫ МОЖЕМ ГРУППИРОВАТЬ ДАННЫЕ ПО РАЗЛИЧНЫМ КРИТЕРИЯМ, НАПРИМЕР, ПО КАТЕГОРИИ ПРОДУКТОВ, И АГРЕГИРОВАТЬ ИНФОРМАЦИЮ.

Группировка данных с groupby()

```
df.groupby("Город")["Возраст"].mean()
```

```
# Группировка по категории и вычисление средних продаж  
category_sales = df.groupby('Category')['Sales'].mean()
```

Применение агрегатных функций

```
df.agg({"Возраст": ["mean", "max"], "Зарплата": "sum"})
```


ГРУППИРОВКА ДАННЫХ

```
# Средняя зарплата по отделам
grouped = df.groupby('Department')['Salary'].mean()
print(grouped)
```

Department	
Finance	80000.0
HR	47500.0
IT	65000.0

```
# Несколько статистик
stats = df.groupby('Department').agg({
    'Salary': ['mean', 'max', 'min'],
    'Age': 'median'
})
print(stats)
```

	Salary			Age
	mean	max	min	median
Department				
Finance	80000.0	80000	80000	40.0
HR	47500.0	50000	45000	25.0
IT	65000.0	70000	60000	32.5

```
# Отделы, где средняя зарплата > 60000
high_paid_depts = df.groupby('Department').filter(lambda x: x['Salary'].mean() > 60000)
print(high_paid_depts)
```

	Name	Age	Salary	Department	Salary_normalized	Age_z	Name_length	\
1	Bob	30.0	60000	IT	0.428571	-0.153393	3	
2	Charlie	35.0	70000	IT	0.714286	0.613572	7	
3	David	40.0	80000	Finance	1.000000	1.380537	5	

Агрегация данных в Pandas

```
import pandas as pd

data = {
    'Product': ['A', 'B', 'A', 'B', 'A'],
    'Revenue': [100, 150, 200, 250, 300],
    'Quantity': [10, 15, 20, 25, 30]
}

df = pd.DataFrame(data)

agg_result = df.groupby('Product').agg({'Revenue': 'sum', 'Quantity': 'mean'})

print(agg_result)
```

#	Revenue	Quantity
# Product		
# A	600	20.0
# B	400	20.0

СЛИЯНИЕ, ОБЪЕДИНЕНИЕ И СОЕДИНЕНИЕ ТАБЛИЦ

Объединение по ключам `merge()` :

```
pd.merge(df1, df2, on="ID")
```

Объединение по индексам `join()` :

```
df1.join(df2, how="left")
```

Вертикальное или горизонтальное объединение `concat()` :

```
pd.concat([df1, df2])
```

Объединение данных из разных датафреймов:

```
combined = pd.concat([df1, df2], ignore_index=True)
```

Объединение по индексу:

```
joined = df1.join(df2, how='left')
```

ОБЪЕДИНЕНИЕ ДАННЫХ

Конкатенация и объединение таблиц в Pandas

```
import pandas as pd

data1 = {'A': [1, 2], 'B': [3, 4]}
data2 = {'A': [5, 6], 'B': [7, 8]}
```

```
df1 = pd.DataFrame(data1)
df2 = pd.DataFrame(data2)

result = pd.concat([df1, df2])
print(result)
```

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

ПРЕОБРАЗОВАНИЕ ДАННЫХ -УСТРАНЕНИЕ ДУБЛИКАТОВ

Строки-дубликаты могут появиться в объекте **DataFrame** по разным причинам.

Метод **uplicated** объекта `DataFrame` возвращает булев объект `Series`, который для каждой строки показывает, является она дубликатом или нет (все значения в столбцах в точности равны соответственным значениям в ранее встретившейся строке).

```
In [51]: data.duplicated()
Out[51]:
0      False
1      False
2      False
3      False
4      False
5      False
6       True
dtype: bool
```

А метод **drop_duplicates** возвращает `DataFrame`, содержащий только строки, которым в массиве, возвращенном методом `uplicated`, соответствует значение `False`:

```
In [52]: data.drop_duplicates()
Out[52]:
   k1 k2
0  one  1
1  two  1
2  one  2
3  two  3
4  one  3
5  two  4
```

ПРЕОБРАЗОВАНИЕ ДАННЫХ -УСТРАНЕНИЕ ДУБЛИКАТОВ

Можно указать произвольное подмножество столбцов, которые необходимо исследовать на наличие дубликатов. Допустим, есть еще один столбец значений, и мы хотим отфильтровать строки, которые содержат повторяющиеся значения только в столбце

```
In [53]: data["v1"] = range(7)
```

```
In [54]: data
```

```
Out[54]:
```

	k1	k2	v1
0	one	1	0
1	two	1	1
2	one	2	2
3	two	3	3
4	one	3	4
5	two	4	5
6	two	4	6

```
In [55]: data.drop_duplicates(subset=["k1"])
```

```
Out[55]:
```

	k1	k2	v1
0	one	1	0
1	two	1	1

По умолчанию методы `uplicated` и `drop_duplicates` оставляют первую встретившуюся строку с данной комбинацией значений. Но если задать параметр `keep="last"`, то будет оставлена последняя строка:

```
In [56]: data.drop_duplicates(["k1", "k2"], keep="last")
```

```
Out[56]:
```

	k1	k2	v1
0	one	1	0
1	two	1	1
2	one	2	2
3	two	3	3
4	one	3	4
6	two	4	6

УДАЛЕНИЕ ДУБЛИКАТОВ

```
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eve'],
    'Age': [25, 30, 35, 40, 25],
    'Salary': [50000, 60000, 70000, 80000, 45000],
    'Department': ['HR', 'IT', 'IT', 'Finance', 'HR']
}
df = pd.DataFrame(data)

# Удаление полных дубликатов
df_no_duplicates = df.drop_duplicates()

# Удаление дубликатов по конкретному столбцу
df_unique_names = df.drop_duplicates(subset=['Age'])

# Сохранение последнего дубликата
df_last = df.drop_duplicates(subset=['Age'], keep='last')

print(df_unique_names)
print("")
print(df_last)
```

```
→
```

	Name	Age	Salary	Department
0	Alice	25	50000	HR
1	Bob	30	60000	IT
2	Charlie	35	70000	IT
3	David	40	80000	Finance

	Name	Age	Salary	Department
1	Bob	30	60000	IT
2	Charlie	35	70000	IT
3	David	40	80000	Finance
4	Eve	25	45000	HR

ОЧИСТКА И ПОДГОТОВКА ДАННЫХ

Часто бывает необходимо произвести преобразование набора данных, исходя из значений в некотором массиве, объекте Series или столбце объекта DataFrame.

ПРИМЕР: Допустим, что требуется добавить столбец, в котором указано соответствующее сорту мяса животное. Создадим отображение сортов мяса на виды животных:

Метод **map** объекта Series принимает функцию или похожий на словарь объект, содержащий отображения, которое реализует преобразование значений:

```
meat_to_animal = {  
    "bacon": "pig",  
    "pulled pork": "pig",  
    "pastrami": "cow",  
    "corned beef": "cow",  
    "honey ham": "pig",  
    "nova lox": "salmon"  
}
```

Можно было бы также передать функцию, выполняющую преобразование:

```
In [60]: data["animal"] = data["food"].map(meat_to_animal)
```

```
In [61]: data
```

```
Out[61]:
```

	food	ounces	animal
0	bacon	4.0	pig
1	pulled pork	3.0	pig
2	bacon	12.0	pig
3	pastrami	6.0	cow
4	corned beef	7.5	cow
5	bacon	8.0	pig
6	pastrami	3.0	cow
7	honey ham	5.0	pig
8	nova lox	6.0	salmon

```
In [62]: def get_animal(x):  
    ....: return meat_to_animal[x]
```

```
In [63]: data["food"].map(get_animal)
```

```
Out[63]:
```

0	pig
1	pig
2	pig
3	cow
4	cow
5	pig
6	cow
7	pig
8	salmon

Name: food, dtype: object

ОЧИСТКА И ПОДГОТОВКА ДАННЫХ - ЗАМЕНА ЗНАЧЕНИЙ

Метод `replace` предлагает для этого более простой и гибкий способ модифицировать подмножество значений, хранящихся в объекте `Series`. Рассмотрим такой

объект `Series`:
Рассмотрим такой объект `Series`:

```
In [64]: data = pd.Series([1., -999., 2., -999., -1000., 3.])
```

```
In [65]: data
```

```
Out[65]:
```

```
0      1.0
```

```
1    -999.0
```

```
2      2.0
```

```
3    -999.0
```

```
4   -1000.0
```

```
5      3.0
```

```
dtype: float64
```

Значение `-999` здесь выбран как маркер отсутствия данных.

Заменим значения `-999` теми, которые понимает `pandas`, воспользуемся методом `replace`, порождающим новый объект `Series`:

```
In [66]: data.replace(-999, np.nan)
```

```
Out[66]:
```

```
0      1.0
```

```
1      NaN
```

```
2      2.0
```

```
3      NaN
```

```
4   -1000.0
```

```
5      3.0
```

```
dtype: float64
```

ОЧИСТКА И ПОДГОТОВКА ДАННЫХ - ЗАМЕНА ЗНАЧЕНИЙ

Чтобы заменить сразу несколько значений, нужно передать их список и заменяющее значение:

```
In [67]: data.replace([-999, -1000], np.nan)
Out[67]:
0    1.0
1    NaN
2    2.0
3    NaN
4    NaN
5    3.0
dtype: float64
```

Если для каждого заменяемого значения нужно свое заменяющее, передай те список замен:

```
In [66]: data.replace(-999, np.nan)
Out[66]:
0    1.0
1    NaN
2    2.0
3    NaN
4   -1000.0
5    3.0
dtype: float64
```

В аргументе можно передавать также словарь

```
In [69]: data.replace({-999: np.nan, -1000: 0})
Out[69]:
0    1.0
1    NaN
2    2.0
3    NaN
4    0.0
5    3.0
dtype: float64
```

Методы сортировки датасетов в Pandas

```
import pandas as pd

data = {
    'Name': ['Anna', 'Mike', 'Lucy', 'John', 'Jack'],
    'Age': [25, 21, 23, 29, 20],
    'Salary': [50000, 55000, 52000, 60000, 49000]
}

df = pd.DataFrame(data)

sorted_df = df.sort_values(by='Age')
```

СОРТИРОВКА ДАННЫХ



```
# Сортировка по одному столбцу
df_sorted = df.sort_values('Salary', ascending=False) # По убыванию

# Сортировка по нескольким столбцам
df_sorted2 = df.sort_values(['Department', 'Salary'], ascending=[True, False])

print(df_sorted2)
```



	Name	Age	Salary	Department	Salary_normalized	Age_z	Name_length
3	David	40.0	80000	Finance	1.000000	1.380537	5
0	Alice	25.0	50000	HR	0.142857	-0.920358	5
4	Eve	25.0	45000	HR	0.000000	-0.920358	3
2	Charlie	35.0	70000	IT	0.714286	0.613572	7
1	Bob	30.0	60000	IT	0.428571	-0.153393	3

ВЫЧИСЛЕНИЯ С ОБЪЕКТАМИ CATEGORICAL

В pandas имеется расширенный тип **Categorical**, специально предназначенный для хранения данных, представленных целочисленными категориями, т. е. в некоторой кодировке. Это популярная техника сжатия, применяемая, когда данные содержат много одинаковых значений. Она может дать значительный прирост производительности и одновременно снизить потребление памяти, особенно

в случае строковых данных.

Рассмотрим уже встречавшийся ранее объект Series:

```
In [208]: fruits = ['apple', 'orange', 'apple', 'apple'] * 2
In [209]: N = len(fruits)
In [210]: rng = np.random.default_rng(seed=12345)
In [211]: df = pd.DataFrame({'fruit': fruits,
.....:                     'basket_id': np.arange(N),
.....:                     'count': rng.integers(3, 15, size=N),
.....:                     'weight': rng.uniform(0, 4, size=N)},
.....:                     columns=['basket_id', 'fruit', 'count', 'weight'])
```

Теперь значением `fruit_cat` является экземпляр типа `pandas.Categorical`, который можно получить с помощью атрибута `.array`

Здесь `df['fruit']` – массив строковых объектов. Его можно следующим образом преобразовать в категориальную форму:

```
In [213]: fruit_cat = df['fruit'].astype('category')
In [214]: fruit_cat
Out[214]:
0    apple
1    orange
2    apple
3    apple
4    apple
5    orange
6    apple
7    apple
Name: fruit, dtype: category
Categories (2, object): ['apple', 'orange']
```

```
In [215]: c = fruit_cat.array
In [216]: type(c)
Out[216]: pandas.core.arrays.categorical.Categorical
Объект Categorical имеет атрибуты categories и codes:
In [217]: c.categories
Out[217]: Index(['apple', 'orange'], dtype='object')
In [218]: c.codes
Out[218]: array([0, 1, 0, 0, 0, 1, 0, 0], dtype=int8)
```

ВЫЧИСЛЕНИЯ С ОБЪЕКТАМИ CATEGORICAL

Возьмем случайные числовые данные и воспользуемся функцией распределения по интервалам `pandas.qcut`.

Она возвращает объект **pandas.Categorical**:
Вычислим квартильное распределение этих данных по интервалам и получим некоторые статистики

Хотя точные выборочные квартили и полезны, но не так, как отчет, содержащий имена квартилей. Для получения такого отчета можно передать функции

qcut аргумент **labels**

```
In [231]: rng = np.random.default_rng(seed=12345)
```

```
In [232]: draws = rng.standard_normal(1000)
```

```
In [233]: draws[:5]
```

```
Out[233]: array([-1.4238, 1.2637, -0.8707, -0.2592, -0.0753])
```

```
In [234]: bins = pd.qcut(draws, 4)
```

```
In [235]: bins
```

```
Out[235]:
```

```
[(-3.121, -0.675], (0.687, 3.211], (-3.121, -0.675], (-0.675, 0.0134], (-0.675, 0.0134], ..., (0.0134, 0.687], (0.0134, 0.687], (-0.675, 0.0134], (0.0134, 0.687], (-0.675, 0.0134]]
```

```
Length: 1000
```

```
Categories (4, interval[float64, right]): [(-3.121, -0.675] < (-0.675, 0.0134] < (0.0134, 0.687] < (0.687, 3.211]]
```

```
In [236]: bins = pd.qcut(draws, 4, labels=['Q1', 'Q2', 'Q3', 'Q4'])
```

```
In [237]: bins
```

```
Out[237]:
```

```
['Q1', 'Q4', 'Q1', 'Q2', 'Q2', ..., 'Q3', 'Q3', 'Q2', 'Q3', 'Q2']
```

```
Length: 1000
```

```
Categories (4, object): ['Q1' < 'Q2' < 'Q3' < 'Q4']
```

```
In [238]: bins.codes[:10]
```

```
Out[238]: array([0, 3, 0, 1, 1, 0, 0, 2, 2, 0], dtype=int8)
```

ВЫЧИСЛЕНИЯ С ОБЪЕКТАМИ CATEGORICAL

В примере выше категориальный объект bins с метками не содержит информацию о границах интервалов, поэтому для получения сводных статистик воспользуемся

```
In [239]: bins = pd.Series(bins, name='quartile')
```

```
In [240]: results = (pd.Series(draws)
.....:               .groupby(bins)
.....:               .agg(['count', 'min', 'max']))
.....:               .reset_index())
```

```
In [241]: results
```

```
Out[241]:
```

	quartile	count	min	max
0	Q1	250	-3.119609	-0.678494
1	Q2	250	-0.673305	0.008009
2	Q3	250	0.018753	0.686183
3	Q4	250	0.688282	3.211418

В столбце результата 'quartile' сохранена исходная категориальная информация из bins, включая упорядочение:

```
In [242]: results['quartile']
```

```
Out[242]:
```

```
0    Q1
1    Q2
2    Q3
3    Q4
```

```
Name: quartile, dtype: category
```

```
Categories (4, object): ['Q1' < 'Q2' < 'Q3' < 'Q4']
```

ВЫЧИСЛЕНИЯ С ОБЪЕКТАМИ CATEGORICAL

Категориальные методы класса Series в pandas

Метод	Описание
<code>add_categories</code>	Добавить новые (невстречающиеся) категории в конец списка существующих категорий
<code>as_ordered</code>	Упорядочить категории
<code>as_unordered</code>	Не упорядочивать категории
<code>remove_categories</code>	Удалить категории, подставив вместо всех удаленных значений null
<code>remove_unused_categories</code>	Удалить категории, не встречающиеся в данных
<code>rename_categories</code>	Заменить имена категорий; количество категорий при этом должно остаться тем же
<code>reorder_categories</code>	Ведет себя так же, как <code>rename_categories</code> , но может и изменить результат, чтобы упорядочить категории
<code>set_categories</code>	Заменить старое множество категорий новым; при этом категории можно добавлять или удалять

ПРЕОБРАЗОВАНИЕ ДАННЫХ

```
# Конвертация в другой тип
df['Age'] = df['Age'].astype('float32')

# Категориальные данные
df['Department'] = df['Department'].astype('category')

# Минимакс-нормализация
df['Salary_normalized'] = (df['Salary'] - df['Salary'].min()) / (df['Salary'].max() - df['Salary'].min())

# Z-нормализация
df['Age_z'] = (df['Age'] - df['Age'].mean()) / df['Age'].std()

# Лямбда-функция
df['Name_length'] = df['Name'].apply(lambda x: len(x))

# Пользовательская функция
def salary_category(salary):
    if salary < 50000: return 'Low'
    elif salary < 70000: return 'Medium'
    else: return 'High'

df['Salary_Category'] = df['Salary'].apply(salary_category)

print(df)
```

```

   Name  Age  Salary Department  Salary_normalized  Age_z  Name_length  \
0  Alice  25.0   50000         HR          0.142857 -0.920358           5
1   Bob   30.0   60000         IT          0.428571 -0.153393           3
2 Charlie  35.0   70000         IT          0.714286  0.613572           7
3  David  40.0   80000    Finance          1.000000  1.380537           5
4   Eve   25.0   45000         HR          0.000000 -0.920358           3

   Salary_Category
0          Medium
1          Medium
2             High
3             High
4             Low
```


СПИСОК ОСНОВНЫХ МЕТОДОВ PANDAS

- `pd.DataFrame(data, columns)`: Создает новый DataFrame из данных. `data` может быть списком, массивом, словарем и другими структурами данных, а `columns` - список с именами столбцов.
- `df.head(n)`: Возвращает первые `n` строк DataFrame.
- `df.tail(n)`: Возвращает последние `n` строк DataFrame.
- `df.info()`: Выводит информацию о DataFrame, включая количество непропущенных значений, типы данных и использование памяти.
- `df.describe()`: Возвращает статистический анализ числовых столбцов, включая среднее, стандартное отклонение, минимум и максимум.
- `df.shape`: Возвращает кортеж с количеством строк и столбцов в DataFrame.
- `df.columns`: Возвращает список с названиями столбцов.
- `df.index`: Возвращает индекс DataFrame.
- `df.values`: Возвращает данные DataFrame в виде массива NumPy.
- `df.set_index(keys)`: Устанавливает индекс DataFrame, где `keys` - столбец или список столбцов, которые будут индексами.
- `df.reset_index()`: Сбрасывает индекс, восстанавливая его в числовой.
- `df.sort_values(by)`: Сортирует DataFrame по указанному столбцу или столбцам.
- `df.groupby(by)`: Группирует данные по указанному столбцу или столбцам для проведения агрегации.
- `df.apply(func)`: Применяет функцию `func` к каждому столбцу или строке в DataFrame.

СПИСОК ОСНОВНЫХ МЕТОДОВ PANDAS

- `df.isna()`: Возвращает DataFrame с булевыми значениями, указывающими на пропущенные значения.
- `df.dropna()`: Удаляет строки с пропущенными значениями.
- `df.fillna(value)`: Заменяет пропущенные значения значением `value`.
- `df.pivot_table(values, index, columns, aggfunc)`: Создает сводную таблицу, где `values` - агрегируемые значения, `index` - строки, `columns` - столбцы, и `aggfunc` - функция агрегации.
- `df.merge(other, on)`: Объединяет два DataFrame по общему столбцу или столбцам.
- `df.plot(kind)`: Создает график из данных, где `kind` - тип графика, такой как 'line', 'bar', 'hist', 'scatter' и другие.
- `df.to_csv(file_path)`: Сохраняет DataFrame в CSV-файл.
- `df.to_excel(file_path)`: Сохраняет DataFrame в Excel-файл.
- `df.to_sql(table_name, connection)`: Записывает DataFrame в SQL-базу данных.
- `df.loc[row, column]`: Индексация DataFrame по меткам, позволяет выбирать конкретные строки и столбцы.
- `df.iloc[row, column]`: Индексация DataFrame по целочисленным индексам, позволяет выбирать конкретные строки и столбцы.
- `df.rename(columns)`: Переименовывает столбцы DataFrame в соответствии с указанными именами в словаре `columns`.
- `df.drop(columns)`: Удаляет указанные столбцы из DataFrame.
- `df.duplicated()`: Возвращает булевы значения, указывающие на дубликаты строк.
- `df.drop_duplicates()`: Удаляет дубликаты строк из DataFrame.

СПИСОК ОСНОВНЫХ МЕТОДОВ PANDAS

- `df.corr()`: Возвращает корреляционную матрицу для числовых столбцов.
- `df.merge()`: Объединяет два DataFrame по заданным столбцам и ключам.
- `df.sample(n)`: Возвращает случайные `n` строк из DataFrame.
- `df.nlargest(n, column)`: Возвращает `n` строк с наибольшими значениями в указанном столбце.
- `df.nsmallest(n, column)`: Возвращает `n` строк с наименьшими значениями в указанном столбце.
- `df.drop(columns)`: Удаляет указанные столбцы из DataFrame.
- `df.astype(types)`: Изменяет типы данных столбцов DataFrame в соответствии с указанными типами в словаре `types`.
- `df.melt(id_vars)`: Переводит DataFrame в "расплавленный" формат, где столбцы преобразуются в два столбца: "переменная" и "значение".
- `df.replace(to_replace, value)`: Заменяет значения в DataFrame на указанные значения.
- `df.str.replace(old, new)`: Производит замену текстовых значений в столбцах с использованием регулярных выражений.
- `df.to_datetime(column)`: Преобразует столбец в формат даты и времени.
- `df.resample(rule)`: Применяет ресемплинг временных данных с указанным правилом (например, 'D' для дней, 'M' для месяцев).
- `df.diff(periods)`: Вычисляет разницу между текущим и предыдущим значением в столбце.
- `df.rolling(window)`: Выполняет скользящее окно для вычисления статистик на скользящих интервалах времени.
- `df.nunique()`: Возвращает количество уникальных значений в каждом столбце.
- `df.value_counts()`: Подсчитывает количество вхождений каждого уникального значения в столбце.
- `df.to_dict()`: Преобразует DataFrame в словарь, где ключами могут быть столбцы.
- `df.to_numpy()`: Преобразует DataFrame в массив NumPy.

СПИСОК ОСНОВНЫХ МЕТОДОВ PANDAS

- `df.dtypes`: Возвращает типы данных для каждого столбца в DataFrame.
- `df.duplicated()`: Возвращает булевы значения, указывающие на дубликаты строк.
- `df.drop_duplicates()`: Удаляет дубликаты строк из DataFrame.
- `df.isin(values)`: Создает маску, указывающую на вхождение значений из `values` в DataFrame.
- `df.where(condition, other)`: Возвращает DataFrame, заменяя значения, которые не удовлетворяют условию, на `other`.
- `df.droplevel(level)`: Удаляет один или несколько уровней иерархического индекса.
- `df.unstack(level)`: Поворачивает иерархический индекс, преобразуя уровни в столбцы.
- `df.stack()`: Обратная операция unstack, преобразует столбцы в иерархический индекс.
- `df.resample(rule)`: Применяет ресемплинг временных данных с указанным правилом (например, 'D' для дней, 'M' для месяцев).
- `df.diff(periods)`: Вычисляет разницу между текущим и предыдущим значением в столбце.
- `df.rolling(window)`: Выполняет скользящее окно для вычисления статистик на скользящих интервалах времени.
- `df.nunique()`: Возвращает количество уникальных значений в каждом столбце.
- `df.value_counts()`: Подсчитывает количество вхождений каждого уникального значения в столбце.
- `df.to_string()`: Преобразует DataFrame в строку, удобную для отображения.
- `df.to_markdown()`: Генерирует таблицу Markdown из DataFrame.
- `df.style`: Позволяет настраивать стили отображения DataFrame, включая форматирование, цветовую подсветку и многое другое.
- `df.to_clipboard()`: Копирует данные DataFrame в буфер обмена.
- `df.to_html()`: Преобразует DataFrame в HTML-таблицу.