Имя библиотеки **pandas** образовано от английского словосочетания «**panel data**» (известного в другом варианте как «longitudinal data» – «данные многомерного временно́го ряда при долговременном наблюдении»), которое обозначает наборы данных нескольких переменных, наблюдаемых в течение нескольких (многих) интервалов времени для одного объекта.

https://dfedorov.spb.ru

Series

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

>>> pd.__version__
```

Можем обратиться по метке индекса

Объект **Series** библиотеки Pandas — одномерный массив индексированных данных.
Его можно создать из списка или массива следующим образом:

```
>>> data = pd.Series([0.25, 0.5, 0.75, 1.0])
>>> data
0    0.25
1    0.50
2    0.75
3    1.00
dtype: float64
```

Список

data.values

Обычный массив NumPy

array([0.25, 0.5 , 0.75, 1.])

data.index

RangeIndex(start=0, stop=4, step=1)

Явное описание индекса расширяет возможности объекта Series . Такой индекс не должен быть целым числом, а может состоять из значений любого нужного типа. Например, при желании мы можем использовать в качестве индекса строковые значения:

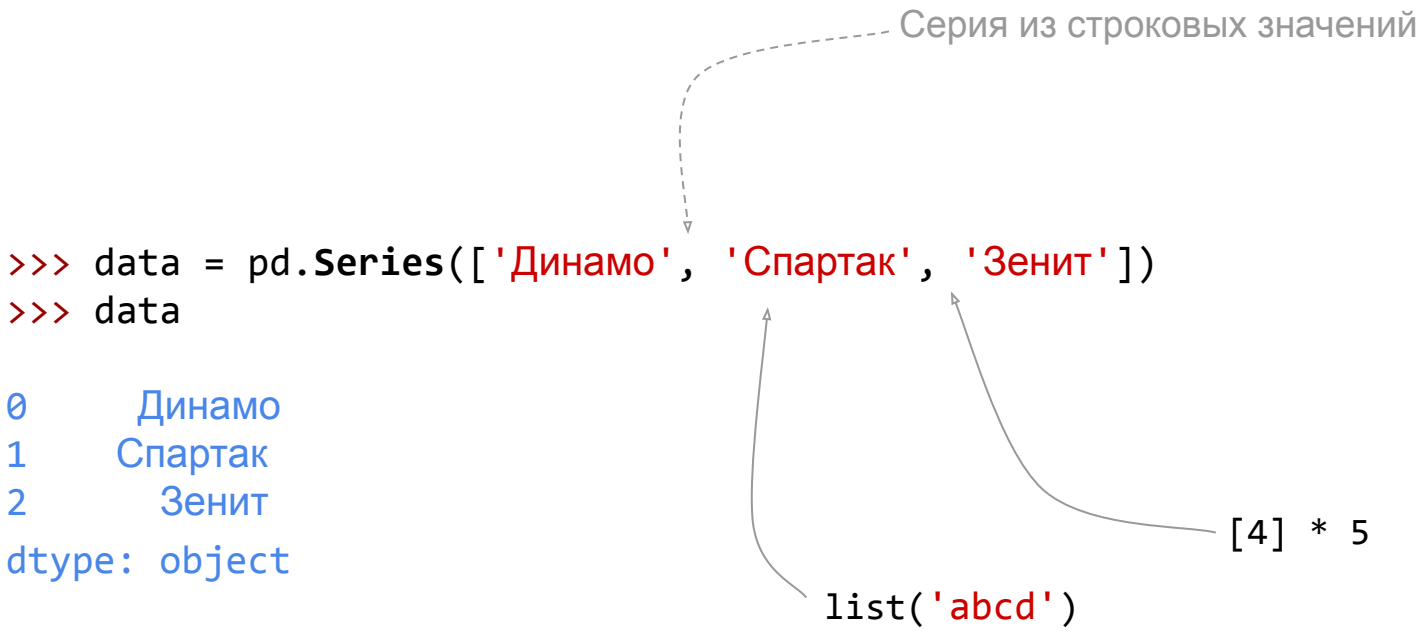
```
>>> data = pd.Series([0.25, 0.5, 0.75, 1.0], index=['a', 'b', 'c', 'd'])
>>> data

a    0.25
b    0.50
c    0.75
d    1.00
dtype: float64

>>> data['b']
0.5

>>> data.index
```

Метки строк (индексные метки)

можно так:
index=[2, 5, 3, 7]

Сюда можно передать массив NumPy или отдельное число

```
>>> len(data)
>>> data.size
>>> data.shape
```


Серия из строковых значений
>>> data = pd.Series(['Динамо', 'Спартак', 'Зенит'])
>>> data
0 Динамо
1 Спартак
2 Зенит
dtype: object
>>> data['Сатурн-1991']
>>> del data['Сатурн-1991']
list('abcd')
[4] * 5

```
>>> import numpy as np

>>> s1 = pd.Series(np.arange(4, 9))

>>> s2 = pd.Series(np.random.normal(size=5))

>>> s1.head() # n = 5
>>> s2.tail()
>>> s2.take([1, 3, 2])
```

Вспоминаем все, что говорили
про создание массивов NumPy

Можно сделать аналогию «объект Series — словарь» еще более наглядной, сконструировав объект Series непосредственно из словаря Python:

```
>>> population_dict = {'California': 38332521,
                       'Texas': 26448193,
                       'New York': 19651127,
                       'Florida': 19552860,
                       'Illinois': 12882135}
```

Обычный словарь чистого Python

```
>>> population = pd.Series(population_dict)
>>> population
```

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

```
>>> population['California']
```

Так обычный словарь не умеет:

```
>>> population['California': 'Florida']
```

Индексация и выборка данных

Объект Series во многом ведет себя подобно одномерному массиву библиотеки NumPy и стандартному словарю языка Python.

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

Ключи

Значения

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

```
>>> data['b']
0.5
```

```
>>> 'a' in data
True

>>> data.keys()
Index(['a', 'b', 'c', 'd'], dtype='object')

>>> list(data.items())
[('a', 0.25), ('b', 0.5), ('c', 0.75), ('d', 1.0)]
```

Объекты **Series** можно модифицировать с помощью синтаксиса, подобного синтаксису для словарей. Аналогично расширению словаря путем присваивания значения для нового ключа можно расширить объект **Series**, присвоив значение для нового значения индекса:

```
>>> data['e'] = 1.25
>>> data

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

>>> data[['e', 'c']]
```

```
>>> data['a':'c'] # срез через явный индекс, включительно

a    0.25
b    0.50
c    0.75
dtype: float64

>>> data[0:2]  # срез через целочисленный индекс, не включительно

a    0.25
b    0.50
dtype: float64

>>> data[(data > 0.3) & (data < 0.8)] # маскирование

b    0.50
c    0.75
dtype: float64

>>> data[['a', 'e']] # прихотливая индексация

a    0.25
e    1.25
dtype: float64
```

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

```
>>> (data > 0.3).all()
>>> (data > 0.3).any()
```

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

1    a
3    b
5    c
dtype: object

>>> data[1]

'a'

>>> data[1:3]

3    b
5    c
dtype: object
```

Подобные обозначения для срезов и индексации могут привести к путанице.
Например, при наличии у объекта Series явного целочисленного индекса операция индексации data[1] будет использовать явные индексы, а операция среза data[1:3] — неявный индекс в стиле языка Python.

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

```
>>> data.loc[1]

'a'

>>> data.loc[1:3]

1    a
3    b
dtype: object

>>> data.iloc[1]

'b'

>>> data.iloc[1:3]

3    b
5    c
dtype: object
```

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

Один из руководящих принципов написания кода на языке Python — «лучше явно, чем неявно». То, что атрибуты loc и iloc по своей природе явные, делает их очень удобными для обеспечения «чистоты» и удобочитаемости кода.

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

a     0.25
b     0.50
c     0.75
d     1.00
dtype: float64

>>> data.loc[['a', 'b']]
>>> data.loc[['a', 'e']]
>>> data.iloc[[1, 3, 2]]
```

Выравнивание индексов в объектах Series

Допустим, мы объединили два различных источника данных, чтобы найти три штата США с наибольшей площадью и три штата США с наибольшим количеством населения:

```
>>> area = pd.Series({'Alaska': 1723337,
                      'Texas': 695662,
                      'California': 423967}, name='area')
>>> population = pd.Series({'California': 38332521,
                            'Texas': 26448193,
                            'New York': 19651127}, name='population')
```

Посмотрим, что получится, если разделить второй результат на первый для вычисления плотности населения:

```
>>> population / area
```

```
Alaska              NaN
California    90.413926
New York            NaN
Texas         38.018740
dtype: float64
```

Аналогичным образом реализовано сопоставление индексов для всех встроенных арифметических выражений языка Python: все отсутствующие значения заполняются по умолчанию значением NaN.

Получившийся в итоге массив содержит объединение индексов двух исходных массивов, которое можно определить посредством стандартной арифметики множеств языка Python для этих индексов:

```
>>> area.index | population.index
Index(['Alaska', 'California', 'New York', 'Texas'], dtype='object')
```

Оператор языка Python	Метод (-ы) библиотеки Pandas
+	add()
–	sub(), subtract()
*	mul(), multiply()
/	truediv(), div(), divide()
//	floordiv()
%	mod()
**	pow()

```
>>> population.div(area, fill_value=0)  # population / area

    Alaska         0.000000
    California    90.413926
    New York            inf
    Texas         38.018740
    dtype: float64
```

По желанию можно задать явным образом значения заполнителей для любых потенциально отсутствующих элементов

```
>>> import pandas as pd

>>> s1 = pd.Series([1, 2], index=['a', 'b'])

>>> s1
a    1
b    2
dtype: int64

>>> s1 * 2
a    2
b    4
dtype: int64
```

```
>>> import pandas as pd

>>> t = pd.Series(2, index=s1.index)

>>> t
a    2
b    2
dtype: int64

>>> s1 * t
a    2
b    4
dtype: int64
```

Переиндексация в объектах Series

```
>>> data = pd.Series([0.25, 0.5, 0.75, 1.0])
>>> data

0    0.25
1    0.50
2    0.75
3    1.00
dtype: float64


>>> index = data.index
>>> index

RangeIndex(start=0, stop=4, step=1)
```

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

a    0.25
b    0.50
c    0.75
d    1.00
dtype: float64

>>> index = data.index
>>> index

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

Как объект Series содержат явный индекс, обеспечивающий возможность ссылаться на данные и модифицировать их. Объект **Index** можно рассматривать или как **неизменяемый массив** (immutable array), или как **упорядоченное множество** (ordered set) (формально мультимножество, так как объекты Index могут содержать повторяющиеся значения).

```
>>> ind = pd.Index([2, 3, 5, 7, 11])
>>> ind
```

```
Int64Index([2, 3, 5, 7, 11], dtype='int64')
```

```
>>> ind[1]
>>> ind[::2]
>>> print(ind.size, ind.shape, ind.ndim, ind.dtype)
```

Неизменяемость делает безопаснее совместное использование индексов несколькими объектами DataFrame и массивами, исключая возможность побочных эффектов в виде случайной модификации индекса по неосторожности.

Объекты библиотеки Pandas спроектированы с прицелом на упрощение таких операций, как соединения наборов данных, зависящие от многих аспектов арифметики множеств. Объект **Index** следует большинству соглашений, используемых встроенной структурой данных **set** языка Python, так что объединения, пересечения, разности и другие операции над множествами можно выполнять привычным образом:

```
>>> indA = pd.Index([1, 3, 5, 7, 9])
>>> indB = pd.Index([2, 3, 5, 7, 11])

>>> indA & indB            # intersection

Int64Index([3, 5, 7], dtype='int64')

>>> indA | indB             # union

Int64Index([1, 2, 3, 5, 7, 9, 11], dtype='int64')

>>> indA.get_loc(3) # получаем позиции меток в индексе
1
```

Методы и свойства объекта Index

Метод	Описание
`append`	Конкатенирует с дополнительными индексными объектами, порождая новый объект Index
`diff`	Вычисляет теоретико-множественную разность, представляя ее в виде индексного объекта
`intersection`	Вычисляет теоретико-множественное пересечение
`union`	Вычисляет теоретико-множественное объединение
`isin`	Вычисляет булев массив, показывающий, содержится ли каждое значение индекса в переданной коллекции
`delete`	Вычисляет новый индексный объект, получающийся после удаления элемента с индексом `i`
`drop`	Вычисляет новый индексный объект, получающийся после удаления переданных значений
`insert`	Вычисляет новый индексный объект, получающийся после вставки элемента в позицию с индексом `i`
`is_monotonic`	Возвращает True, если каждый элемент больше предыдущего или равен ему
`is_unique`	Возвращает True, если в индексе нет повторяющихся значений
`unique`	Вычисляет массив уникальных значений в индексе

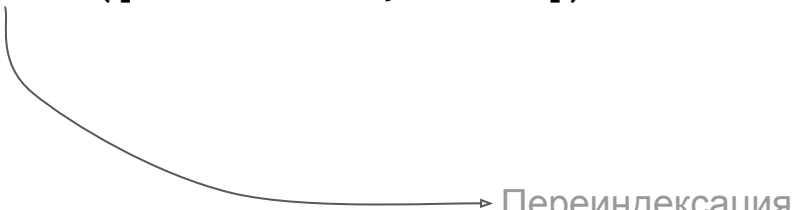
```
>>> sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}

>>> states = ['California', 'Ohio', 'Oregon', 'Texas']

>>> obj = pd.Series(sdata, index=states)
>>> obj
```

```
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
dtype: float64
```

```
>>> sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}

>>> states = ['California', 'Ohio', 'Oregon', 'Texas']

>>> obj = pd.Series(sdata, index=states)
>>> obj
```

```
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
dtype: float64
```

```
>>> obj.reindex(['California', 'Ohio'])
```




```
>>> obj.reindex(['California', 'Ohio', 'Zoo'], fill_value=0)
California        NaN
Ohio          35000.0
Zoo               0.0
dtype: float64
```

```
>>> s3 = pd.Series(['red', 'green', 'blue'], index=[0, 3, 5])
>>> s3
0      red
3    green
5     blue
dtype: object

# пример прямого заполнения
>>> s3.reindex(np.arange(0, 7), method='ffill')
0      red
1      red
2      red
3    green
4    green
5     blue
6     blue
dtype: object

# пример обратного заполнения
>>> s3.reindex(np.arange(0, 7), method='bfill')
0      red
1    green
2    green
3    green
4     blue
5     blue
6      NaN
dtype: object
```

Использование разных типов для одних и тех же меток порождает большие проблемы:

```
>>> s1 = pd.Series([0, 1, 2], index=[0, 1, 2])
>>> s2 = pd.Series([3, 4, 5], index=['0', '1', '2'])
>>> s1 + s2
0   NaN
1   NaN
2   NaN
0   NaN
1   NaN
2   NaN
dtype: float64
```

```
>>> s2.index = s2.index.values.astype(int)
>>> s1 + s2

0    3
1    5
2    7
dtype: int64
```

```
>>> s2.index
Index(['0', '1', '2'], dtype='object')

>>> s2.index.values
array(['0', '1', '2'], dtype=object)

>>> s2.index.values.astype(int)
array([0, 1, 2])
```