Введение в PANDAS

Камзолов Никита, МОЭВМ, 2023

Для чего нужен анализ данных

- Аналитика данных: продуктовая, маркетинговая и другая.
- Data science и работа с большими данными.
- Статистика.

Почему именно Pandas

- Простые методы обработки данных
- Отличная интеграция с другими библиотеками
- Высокая эффективность
- Активное и поддерживающее сообщество
- Прекрасная документация

Начало работы

- Установка

```
pip install pandas #перед загрузкой не забудьте обновить pip
pip install numpy #рекомендуется к использованию в паре с pandas
```

- Импорт

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

- Jupyter Notebook

```
pip install jupyter #альтернативно можно установить Anaconda
```

Series

Индексированный список

Создание Series

- Из списка:

```
pd.Series([1, 2, 3, 4], index="one, two, three, four".split(', '))
```

```
one      1
two      2
three    3
four     4
dtype: int64
```

- Из словаря с заданием имени:

```
pd.Series({1: 'one', 2: 'two', 3: 'three', 4: 'four'}, name='my_series')
```

```
1      one
2      two
3    three
4     four
Name: my_series, dtype: object
```

- Из значения

```
pd.Series(10, index=['a'])
```

```
a    10
dtype: int64
```

- Из numpy массива

```
arr = np.arange(0, 1, 0.1)
pd.Series(arr)
```

```
0    0.0
1    0.1
2    0.2
3    0.3
4    0.4
5    0.5
6    0.6
7    0.7
8    0.8
9    0.9
dtype: float64
```

Основные свойства Series

- .index (возвращает индекс структуры)

```
series = pd.Series([1, 2, 3, 4], index="one, two, three, four".split(', '))
series.index
```

```
Index(['one', 'two', 'three', 'four'], dtype='object')
```

- .values (возвращает массив NumPy со всеми значениями структуры)

```
series = pd.Series([1, 2, 3, 4], index="one, two, three, four".split(', '))
series.values
```

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

- Размер и форма

```
print(len(series))
print(series.size)
print(series.shape) #будет иметь смысл для DataFrame,для series всегда одно значение
```

```
4
4
(4,)
```

- Тип данных

```
series = pd.Series([1, 2, 3, 4, 5, 6, 7, 8, 9], index=list('abcdefghi'))
series.dtype
```

```
dtype('int64')
```

Ограничение размера

- .head()

```
series = pd.Series([1, 2, 3, 4, 5, 6, 7, 8, 9], index=list('abcdefghi'))
series.head(4) # Можно вызвать без значения, по умолчанию n = 10
```

```
a    1
b    2
c    3
d    4
dtype: int64
```

- .tail()

```
series = pd.Series([1, 2, 3, 4, 5, 6, 7, 8, 9], index=list('abcdefghi'))
series.tail(4)
```

```
f    6
g    7
h    8
i    9
dtype: int64
```

- .take()

```
series = pd.Series([0, 1, 2, 3, 4, 5, 6, 7, 8], index=list('abcdefghi'))
series.take((1, 4, 5)) # Порядковые имена элементов серии
```

```
b    1
e    4
f    5
dtype: int64
```

Работа с индексами

- MultiIndex
  - Из кортежа

```
tuples = [('one', 'foo'),
          ('one', 'bar'),
          ('two', 'foo'),
          ('two', 'bar')]
index = pd.MultiIndex.from_tuples(tuples)
pd.Series([1, 2, 3, 4], index=index)
```

```
one  foo    1
     bar    2
two  foo    3
     bar    4
dtype: int64
```

  - Из двух Iterable'ов

```
iterables = [['one', 'two'], ['foo', 'bar']]
#сопоставление всех элементов первого списка, всем элементам второго
index = pd.MultiIndex.from_product(iterables)
pd.Series([1, 2, 3, 4], index=index)
```

```
one  foo    1
     bar    2
two  foo    3
     bar    4
dtype: int64
```

- Изменение индекса

```
series = pd.Series([1, 2, 3, 4, 5])
series.index = list('abcde')
series #изменение in-place
```

```
a    1
b    2
c    3
d    4
e    5
dtype: int64
```

```
series = pd.Series([1, 2, 3, 4, 5], index=list('abcde'))
#возвращает новый объект, меняя порядок индексов
#если нового индекса нет в старом списке, подставится значение NaN
series.reindex(list('edcbA'))
```

```
e    5.0
d    4.0
c    3.0
b    2.0
A    NaN
dtype: float64
```

```
series = pd.Series([1, 2, 3, 4, 5], index=list('abcde'))
#можно выбрать значение, которым заполнить NaN
series.reindex(list('edcbA'), fill_value=0)
```

```
e    5
d    4
c    3
b    2
A    0
dtype: int64
```

Полезные методы Series

- .describe()

```
series = pd.Series([0, 1, 2, 3, 4, 5, 6, 7, 8], index=list('abcdefghi'))
series.describe()
```

```
count    9.000000
mean     4.000000
std      2.738613
min      0.000000
25%      2.000000
50%      4.000000
75%      6.000000
max      8.000000
dtype: float64
```

- .value_counts()

```
series = pd.Series(['a', 'b', 'b', 'c', 'c', 'c', 'd', 'd', 'd', 'd'])
series.value_counts() #количество каждого элемента в серии
```

```
d    4
c    3
b    2
a    1
dtype: int64
```

- .abs()

```
series = pd.Series([-3, -2, -1, 0, 1, 2, 3])
series.abs()
```

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

- .sum(), .mean(), .min(), .max()

```
series = pd.Series([-3, -2, -1, 0, 1, 2, 3])
print(series.sum())#сумма значений
print(series.mean())#среднее значение
print(series.min())#минимальное значение
print(series.max())#максимальное значение
```

```
0
0.0
-3
3
```

- .any(), .all()

```
series = pd.Series([-3, -2, -1, 0, 1, 2, 3])
# .any() возвращает True, если хотя бы один элемент True
# .all() # возвращает True, если все элементы True
print((series > 0).any())
print((series > 0).all())
```

```
True
False
```

- quantile()

```
series = pd.Series(np.arange(0, 10, 0.01))
print(series.quantile(0.5))
print(series.quantile(0.95))
```

```
4.995
9.490499999999999
```

- unique(), nunique()

```
series = pd.Series([1, 2, 2, 3, 3, 4, 4, 5, 5])
print(series.unique())#все уникальные значения
print(series.nunique())#количество уникальных значений
```

```
[1 2 3 4 5]
5
```

.astype()

```
series = pd.Series(list('123'))
series.astype('int') #string -> int
```

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

- .isin()

```
series = pd.Series([3, 2, 1, 0, 1, 2, 3])
series.isin([1, 2, 3]) #проверить каждое значение на принадлежность набору данных
```

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

- .prod

```
series = pd.Series([1, 2, 3, 4])
series.prod() #произведение элементов выборки
```

```
24
```

- .to_numpy()

```
series = pd.Series([3, 2, 1, 0, 1, 2, 3])
series.to_numpy() #преобразование в пипру массив
```

```
array([3, 2, 1, 0, 1, 2, 3])
```

- idxmax(), idxmin()

```
series = pd.Series([3, 2, 1, 0, 1, 2, 3], index=list('abcdefg'))
series.idxmin()#индекс минимального значения
series.idxmax()#индекс максимального значения
```

```
'a'
```

Получение данных из Series

- .iloc (получение данных по позиции)

```
series = pd.Series([3, 2, 1, 0, 1, 2, 3], index=list('abcdefg'))
series.iloc[0]
```

```
3
```

Можно доставать сразу несколько

```
series = pd.Series([3, 2, 1, 0, 1, 2, 3], index=list('abcdefg'))
series.iloc[[0, 1, 2, 3]]
```

```
a    3
b    2
c    1
d    0
dtype: int64
```

- .loc (получение данных по индексу)

```
series = pd.Series([3, 2, 1, 0, 1, 2, 3], index=list('abcdabc'))
#выводятся все индексы, которые подходят под запрос
series.loc[['a', 'b', 'c', 'd']]
```

```
a    3
a    1
b    2
b    2
c    1
c    3
d    0
dtype: int64
```

- Оператор []
  - Нецелочисленный индекс

```
series = pd.Series([1, 2, 3, 4, 5], index=list('abcde'))
series
```

```
a    1
b    2
c    3
d    4
e    5
dtype: int64
```

Может быть использовано, как поиск по позиции и, как поиск по индексу

```
series[1]
```

```
2
```

```
series['b']
```

```
2
```

- Оператор []
  - Целочисленный индекс

```
series = pd.Series([1, 2, 3, 4, 5], index=[1, 2, 3, 4, 5])
series
```

```
1    1
2    2
3    3
4    4
5    5
dtype: int64
```

Может быть использовано только, как поиск по индексу.

```
series[1]
```

```
1
```

При отсутствии индекса в серии – ошибка

```
series[0]
```

```
--------------------
KeyError
```

Изменение данных Series

Для изменения данных Series используются методы аналогичные получению данных, но в них через оператор `=` записывается новое значение.

```
series = pd.Series([1, 2, 3, 4, 5, 6, 7], index=list('abcdeab'))
series.loc['a'] = 15
series
```

```
a    15
b     2
c     3
d     4
e     5
a    15
b     7
dtype: int64
```

```
series = pd.Series([1, 2, 3, 4, 5], index=[1, 2, 3, 4, 5])
series.iloc[1] = 10
series
```

```
1     1
2    10
3     3
4     4
5     5
dtype: int64
```

```
series = pd.Series([1, 2, 3, 4, 5, 6, 7], index=list('abcdeab'))
series['a'] = 10
series
```

```
a    10
b     2
c     3
d     4
e     5
a    10
b     7
dtype: int64
```

Срезы Series

- Срезы по позиции (аналогично спискам в Python)

```
series = pd.Series([1, 2, 3, 4, 5], index=[1, 2, 3, 4, 5])
series.iloc[-1]
```

```
5
```

```
series = pd.Series([1, 2, 3, 4, 5], index=[1, 2, 3, 4, 5])
series.iloc[1::2] #каждое второе начиная с 1-ой позиции (четные позиции)
```

```
2    2
4    4
dtype: int64
```

```
series = pd.Series([1, 2, 3, 4, 5], index=[1, 2, 3, 4, 5])
series.iloc[1:3:] #с 1 по 3 позицию (правая позиция не включается)
```

```
2    2
3    3
dtype: int64
```

- Срезы по индексам

```
series = pd.Series([1, 2, 3, 4, 5], index=list('abcde'))
series.loc['a':'d'] #с 'a' по 'd' (правая позиция включается)
```

```
a    1
b    2
c    3
d    4
dtype: int64
```

```
series = pd.Series([1, 2, 3, 4, 5], index=list('abcde'))
series.loc['a':'d':2] #также можно задавать шаг
```

```
a    1
c    3
dtype: int64
```

Фильтрация значений

Можно использовать логические выражения для фильтрации значений серии.

```
series = pd.Series([1, 2, 3, 4, 5], index=list('abcde'))
series[series < 3]
```

```
a    1
b    2
dtype: int64
```

Можно фильтровать по нескольким условиям

```
series = pd.Series([1, 2, 3, 4, 5], index=list('abcde'))
series[(series > 1) & (series <= series.mean())] #скобки обязательны из-за приоритета операторов
```

```
b    2
c    3
dtype: int64
```

Математические операции над Series

- Операции с числами

Series поддерживает все математические операции Python, если его тип данных их поддерживает

```
series = pd.Series([1, 2, 3, 4, 5])
series / 2
```

```
0    0.5
1    1.0
2    1.5
3    2.0
4    2.5
dtype: float64
```

```
series + 10
```

```
0    11
1    12
2    13
3    14
4    15
dtype: int64
```

```
series ** 2
```

```
0     1
1     4
2     9
3    16
4    25
dtype: int64
```

```
series = pd.Series(list('abcde'))
series * 4
```

```
0    aaaa
1    bbbb
2    cccc
3    dddd
4    eeee
dtype: object
```

- # Математические операции с другими Series

Для выполнения математических операций с другими Series необходимо, чтобы у обеих Series совпадали индексы (порядок индексов не важен). При отсутствии индекса в одном из Series, в значение будет записано NaN.

```
series1 = pd.Series([1, 2, 3, 4, 5], index=list('abcde'))
series2 = pd.Series([1, 2, 3, 4, 5], index=list('ecdba'))
series1 + series2
```

```
a    6
b    6
c    5
d    7
e    6
dtype: int64
```

```
series1 = pd.Series([1, 2, 3, 4, 5], index=list('abcde'))
series2 = pd.Series([1, 2, 3, 4, 5], index=list('bcdef'))
series1 * series2
```

```
a     NaN
b     2.0
c     6.0
d    12.0
e    20.0
f     NaN
dtype: float64
```

```
series1 = pd.Series(list('abcde'), index=list('abcde'))
series2 = pd.Series(list('abcde'), index=list('abcde'))
series1 + series2 #работает со строками
```

```
a    aa
b    bb
c    cc
d    dd
e    ee
dtype: object
```

Помимо обычных математических операций, Series поддерживает их аналоговые математические методы, у которых есть дополнительный параметр fill_value – значение, которым заполнить возникающие при слиянии NaN значения (если в обоих структурах значение по индексу – NaN, то в результирующую Series также войдет NaN)

Математическая операция Python	Метод-аналог Series
+	.add
-	.sub
/	.div
*	.mul
//	.floordiv
%	.mod
**	pow

```
series1 = pd.Series([1, 2, 3, 4, 5], index=list('abcde'))
series2 = pd.Series([1, 2, 3, 4, 5], index=list('abcdf'))
series1.add(series2, fill_value=10)
```

```
a     2.0
b     4.0
c     6.0
d     8.0
e    15.0
f    15.0
dtype: float64
```

Сортировка Series

- .sort_values (сортировка значений)
  - ascending – если True, сортирует в возрастающем порядке, в противном случае – в убывающем (по умолчанию True)
  - inplace – если True, то сортировка не создает новую структуру Series, а проводится ‘in-place’ (по умолчанию False)
  - na_position – если ‘first’, помещает NaN значения в начало, если ‘last’, то в конец (по умолчанию last)
  - ignore_index – если True, выставит серии после сортировки индекс по умолчанию (0…n) (по умолчанию False)
  - kind : {‘quicksort’, ‘mergesort’, ‘heapsort’, ‘stable’} – выбор метода сортировки (по умолчанию ‘quicksort’. Здесь стоит обратить внимание на тип сортировки ‘stable’. Стабильная сортировка сохраняет элементы с одним и тем же ключом в одном и том же относительном порядке. Это часто может быть полезно, если требуется провести сортировку не только по значениям, но и по индексам. Нестабильная сортировка, сломает порядок элементов первого этапа сортировки.
  - key – применяет переданную функцию перед началом сортировки, при этом не изменяя значения итоговой структуры Series.

- Сортировка значений по умолчанию

```
series = pd.Series(['B', 'a', 'A', 'C', 'D', 'd'])
series.sort_values()
```

```
2    A
0    B
3    C
4    D
1    a
5    d
dtype: object
```

- Применяем доступные параметры метода:

```
series.sort_values(
    ascending = False, #сортируем по убыванию
    ignore_index = True, #сбросим индекс
    key = lambda col: col.str.lower() #приводим строку к нижнему регистру,
                                      #чтобы сделать сортировку нерегистрозависимой
)
```

```
0    D
1    d
2    C
3    B
4    a
5    A
dtype: object
```

- .sort_index (сортировка индексов)
  - level – выполнить сортировку только по указанному уровню индекса (по умолчанию None)
  - sort_remaining – если True, сортировка будет производиться по всем индексам поочередно, после сортировки по указанному level (по умолчанию True)
  - Остальные параметры аналогичны .sort_values()

```
series = pd.Series([1, 2, 3, 4, 5, 6], index=['B', 'a', 'A', 'C', 'D', 'd'])
series.sort_index()
```

```
A    3
B    1
C    4
D    5
a    2
d    6
dtype: int64
```

Применение параметра kind

```
series = pd.Series([i for i in range(0, 20)], [np.random.choice(["a", "b", "c"]) for i in range(0, 20)])
```

```
series.sort_values(inplace=True, ascending=False)
series.sort_index(inplace=True)
series #сортировка по индексам успешна, но в рамках одного индекса, значения не отсортированы
```

```
a     0
a    15
a    13
a     2
a     4
a     5
a     7
a     6
b    18
b    17
b    16
b    14
b     9
c     3
c    19
c     1
c    11
c    12
c     8
c    10
dtype: int64
```

```
series.sort_values(inplace=True, ascending=False)
series.sort_index(inplace=True, kind='stable')
series #применяем стабильную сортировку
```

```
a    15
a    13
a     7
a     6
a     5
a     4
a     2
a     0
b    18
b    17
b    16
b    14
b     9
c    19
c    12
c    11
c    10
c     8
c     3
c     1
dtype: int64
```

Методы apply и map

- .map() – используется для замены каждого значения в серии другим значением, которое может быть получено из функции, словаря или Series.
  - .map() со словарем

```
series = pd.Series(list('aaabbbc'))
series.map({'a': 'A', 'b': 'B'}) #нет соответствия для c => NaN
```

```
0      A
1      A
2      A
3      B
4      B
5      B
6    NaN
dtype: object
```

- .map() с функцией

```
def _map(value):
    return value * 2
series = pd.Series(list('aaabbbc'))
series.map(_map)
```

```
0    aa
1    aa
2    aa
3    bb
4    bb
5    bb
6    cc
dtype: object
```

```
series = pd.Series(list('abc'))
series.map("Hello {}".format)
```

```
0    Hello a
1    Hello b
2    Hello c
dtype: object
```

- .map() с Series (аналогичен словарю)

```
series = pd.Series(list('abc'))
series_for_map = pd.Series(list('ABC'), index=list('abc'))
series.map(series_for_map)
```

```
0    A
1    B
2    C
dtype: object
```

```
series = pd.Series(list('abc'))
series_for_map = pd.Series(list('ABC'), index=list('abd'))
series.map(series_for_map) #нет соответствия для c => NaN
```

```
0      A
1      B
2    NaN
dtype: object
```

- .apply() – применяет ко всем значениям Series, переданную функцию. Может также принимать на вход параметры для этой функции.

```
def add_some_value(x, value):
    return x + value

series = pd.Series([1, 2, 3, 4, 5, 6])
series.apply(add_some_value, args=(5,)) #необходимо передавать кортеж из-за особенности метода
```

```
0     6
1     7
2     8
3     9
4    10
5    11
```

```
def add_some_value(x, **kwargs):
    x += kwargs['a']
    x -= kwargs['b']
    x *= kwargs['c']
    return x

series = pd.Series([1, 2, 3, 4, 5, 6])
series.apply(add_some_value, a=1, b=2, c=3) #альтернативный способ передачи аргументов
```

```
0     0
1     3
2     6
3     9
4    12
5    15
dtype: int64
```

Также .apply() может принимать на вход numpy u-функции.

- .apply(np.negative) – применить отрицание
- .apply(np.log) – логарифмировать
- .apply(np.sqrt) – взять корень
- .apply(np.square) – возвести в квадрат
- с полным списком можно ознакомиться здесь - тык