Приведение данных в порядок

Structuring data

Wide form data

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1		Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec	Grand Total
2	Abatz	534	997	207	291	223	370	789	71	563	701	806	200	5,752
3	Blognation	262	656	577	968	169	851	933	427	214	653	106	578	6,394
4	Centizu	378	53	518	339	761	223	809	513	414	180	302	740	5,230
5	Chatterbridge	534	646	285	802	395	368	643	824	207	833	234	430	6,201
26	Yabox	672	884	728	342	548	345	868	236	648	787	879	486	7,423
27	Zoovu	811	809	96	466	459	141	689	485	158	380	847	221	5,562
28	Zooxo	429	626	910	527	79	969	125	791	523	910	54	350	6,293
29	Zoozzy	54	80	342	610	251	482	594	389	546	254	73	400	4,075

Tidy (or narrow) data

	A	B	C	D	E	F	G
1	invoice	company	purchase_date	product	quantity	price	extended amount
2	ZN-870-29	Realcube	3/5/2019	shirt	19	17	$ 323.00
3	JQ-501-63	Zooxo	7/9/2019	book	30	14	$ 420.00
4	FI-165-58	Dabtype	8/12/2019	poster	7	23	$ 161.00
5	XP-005-55	Skipfire	11/18/2019	pen	7	29	$ 203.00
6	NB-917-18	Bluezoom	4/18/2019	poster	36	19	$ 684.00
7	MI-696-11	Zooveo	10/17/2019	pen	-1	30	$ (30.00)
8	MQ-907-02	Babbleset	10/27/2019	poster	30	21	$ 630.00
9	NX-102-26	Fliptune	10/16/2019	book	40	28	$ 1,120.00
10	LE-516-00	Buzzbean	6/17/2019	poster	-3	16	$ (48.00)
11	VD-518-20	Dabshots	3/12/2019	shirt	19	28	$ 532.00
12	OS-688-56	Fiveclub	1/25/2019	book	39	22	$ 858.00
13	KI-908-67	Zoonoodle	3/10/2019	pen	23	28	$ 644.00

Unstructured data

	A	B	C	D
1	Total Sales	$ 510,270.00		Top Selling Products
2				shirt
3	Average Sales	$ 510.27		book
4				poster
5				pen
6	Top Sales Person			
7	Ben Franklin			
8	Andrew Jackson			
9	Harry Truman			

Реальные данные редко бывают очищенными и однородными. В частности, во многих интересных наборах данных некоторое количество данных отсутствует. Еще более затрудняет работу то, что в различных источниках данных отсутствующие данные могут быть помечены различным образом.

Способ обработки отсутствующих данных библиотекой Pandas определяется тем, что она основана на пакете NumPy, в котором отсутствует встроенное понятие NA-значений для всех типов данных, кроме данных с плавающей точкой.

NaN: отсутствующие числовые данные

NaN , представляет собой специальное значение *с плавающей точкой*, распознаваемое всеми системами, использующими стандартное IEEE-представление чисел с плавающей точкой:

```
>>> vals2 = np.array([1, np.nan, 3, 4])
>>> vals2.dtype
dtype('float64')
```

Вы должны отдавать себе отчет, что значение NaN в чем-то подобно «вирусу данных»: оно «заражает» любой объект, к которому «прикасается». Вне зависимости от операции результат арифметического действия с участием NaN будет равен NaN :

```
>>> 1 + np.nan
nan
>>> 0 * np.nan
nan
>>> vals2.sum(), vals2.min(), vals2.max()
(nan, nan, nan)
>>> np.nansum(vals2), np.nanmin(vals2), np.nanmax(vals2)
```

```
>>> pd.Series([1, np.nan, 2, None])
```

None превратился в np.nan

```
0    1.0
1    NaN
2    2.0
3    NaN
dtype: float64
```

Увидел np.nan - перевел все во float64

```
>>> x = pd.Series(range(2), dtype=int)
>>> x

0    0
1    1
dtype: int64

>>> x[0] = None
>>> x

0    NaN
1    1.0
dtype: float64
```

Увидел np.nan - перевел все во float64

Операции над пустыми значениями

isnull() — генерирует булеву маску для отсутствующих значений или isna()

notnull() — противоположность метода isnull().

dropna() — возвращает отфильтрованный вариант данных.

fillna() — возвращает копию данных, в которой пропущенные значения заполнены или восстановлены.

Выявление пустых значений

У структур данных библиотеки Pandas имеются два удобных метода для выявления пустых значений: **isnull**() и **notnull**() . Каждый из них возвращает булеву маску для данных.

```
>>> data = pd.Series([1, np.nan, 'hello', None])
>>> data
0        1
1      NaN
2    hello
3     None
dtype: object

>>> data.isnull()
0    False
1     True
2    False
3     True
dtype: bool

>>> data[data.notnull()]
0        1
2    hello
dtype: object
```

pd.**isnull**(data)

Аналогичные булевы результаты дает использование методов isnull() и notnull() для объектов DataFrame .

Удаление пустых значений

Помимо продемонстрированного выше маскирования, существуют удобные методы: **dropna()** (отбрасывающий NA-значения) и **fillna()** (заполняющий NA-значения).

```
>>> data.dropna()

0      1
2  hello
dtype: object
```

```
>>> df = pd.DataFrame([[1, np.nan, 2],
                       [2, 3, 5],
                       [np.nan, 4, 6]])
>>> df
```

	0	1	2
0	1.0	NaN	2
1	2.0	3.0	5
2	NaN	4.0	6

```
>>> df.dropna()
```

	0	1	2
1	2.0	3.0	5

Нельзя выбросить из DataFrame отдельные значения, только целые строки или столбцы.

По умолчанию dropna() отбрасывает все строки, в которых присутствует хотя бы одно пустое значение.

```
>>> df.dropna(axis='columns')
```

	2
0	2
1	5
2	6

Задание параметра axis=1 отбрасывает все столбцы, содержащие хотя бы одно пустое значение.

Возможно, вам захочется отбросить строки или столбцы, все значения (или большинство) в которых представляют собой NA. Такое поведение можно задать с помощью параметров **how** и **thresh** , обеспечивающих точный контроль допустимого количества пустых значений.

```
>>> df[3] = np.nan
>>> df
```

	0	1	2	3
0	1.0	NaN	2	NaN
1	2.0	3.0	5	NaN
2	NaN	4.0	6	NaN

```
>>> df.dropna(axis='rows', thresh=3)
```

	0	1	2	3
1	2.0	3.0	5	NaN

Для более точного контроля можно задать с помощью параметра **thresh минимальное количество непустых значений для строки/столбца, при котором он не отбрасывается**.

```
>>> df.dropna(axis='columns', how='all')
```

	0	1	2
0	1.0	NaN	2
1	2.0	3.0	5
2	NaN	4.0	6

По умолчанию **how='any'**, то есть отбрасываются все строки или столбцы (в зависимости от ключевого слова axis), содержащие **хоть одно пустое значение**.

Можно также указать значение **how='all'**, при нем будут отбрасываться только строки / столбцы, **все** значения в которых пустые.

Иногда предпочтительнее вместо отбрасывания пустых значений заполнить их каким-то допустимым значением. Это значение может быть фиксированным, например нулем, или интерполированным или восстановленным на основе «хороших» данных значением. Это настолько распространенная операция, что библиотека Pandas предоставляет метод **fillna**, возвращающий копию массива с замененными пустыми значениями.

```
>>> data = pd.Series([1, np.nan, 2, None, 3],
                     index=list('abcde'))
>>> data

a    1.0
b    NaN
c    2.0
d    NaN
e    3.0
dtype: float64
```

```
>>> data.fillna(0)

a    1.0
b    0.0
c    2.0
d    0.0
e    3.0
dtype: float64
```

Можно заполнить NA-элементы одним фиксированным значением, например, нулями

```
>>> data.fillna(method='ffill')

a    1.0
b    1.0
c    2.0
d    2.0
e    3.0
dtype: float64
```

Можно задать параметр заполнения по направлению «вперед», копируя предыдущее значение в следующую ячейку

```
>>> data.fillna(method='bfill')

a    1.0
b    2.0
c    2.0
d    3.0
e    3.0
dtype: float64
```

Можно задать параметр заполнения по направлению «назад», копируя следующее значение в предыдущую ячейку

```
>>> df
```

	0	1	2	3
0	1.0	NaN	2	NaN
1	2.0	3.0	5	NaN
2	NaN	4.0	6	NaN

Для объектов DataFrame опции аналогичны, но в дополнение можно задать ось, по которой будет выполняться заполнение.

```
>>> df.fillna(method='ffill', axis=1)
```

	0	1	2	3
0	1.0	1.0	2.0	2.0
1	2.0	3.0	5.0	5.0
2	NaN	4.0	6.0	6.0

Если при заполнении по направлению «вперед» предыдущего значения нет, то NA-значение остается незаполненным.

Еще одна часто встречающаяся операция – применение функции, определенной для одномерных массивов, к каждому столбцу или строке. Именно это и делает метод **apply** объекта DataFrame:

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

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

```
>>> f = lambda x: x.max() - x.min()
>>> df.apply(f)
```

Функция f, вычисляющая разность между максимальным и минимальным значениями Series, вызывается один раз **для каждого столбца** df. В результате получается объект Series, для которого индексом являются столбцы df.

```
A    1
B    7
C    3
D    3
dtype: int64
```

Если передать методу apply аргумент axis='columns', то функция будет вызываться по одному разу для каждой строки:

```
>>> df.apply(f, axis='columns')
```

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

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

Функция, передаваемая методу **apply**, не обязана возвращать скалярное значение, она может вернуть и объект **Series**, содержащий несколько значений:

```
>>> def f(x):
     return pd.Series([x.min(), x.max()], index=['min', 'max'])

>>> df.apply(f)
```

	A	B	C	D
min	6	2	2	4
max	7	9	5	7

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

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

Допустим, требуется вычислить форматированную строку для каждого элемента df с плавающей точкой. Это позволяет сделать метод **applymap**:

```
>>> format = lambda x: f'{x:.2f}'
>>> df.applymap(format)
```

	A	B	C	D
0	6.00	9.00	2.00	6.00
1	7.00	4.00	3.00	7.00
2	7.00	2.00	5.00	4.00

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

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

В классе **Series** есть метод **map** для применения функции к каждому элементу:

```
>>> df['C'].map(format)

0    2.00
1    3.00
2    5.00
Name: C, dtype: object
```

apply для DataFrame?

```
# создаем датафрейм с 5 строками и 3 столбцами
>>> df = pd.DataFrame(np.arange(0, 15).reshape(5, 3),
                      index=['a', 'b', 'c', 'd', 'e'],
                      columns=['c1', 'c2', 'c3'])
>>> df
```

	c1	c2	c3
a	0	1	2
b	3	4	5
c	6	7	8
d	9	10	11
e	12	13	14

```
# добавляем несколько столбцов и строк в датафрейм
# столбец c4 со значениями NaN
>>> df['c4'] = np.nan

# строка 'f' со значениями от 15 до 18
>>> df.loc['f'] = np.arange(15, 19)

# строка 'g', состоящая из значений NaN
>>> df.loc['g'] = np.nan

# столбец 'C5', состоящий из значений NaN
>>> df['c5'] = np.nan

# меняем значение в столбце 'c4' строки 'a'
>>> df['c4']['a'] = 20
>>> df
```

	c1	c2	c3	c4	c5
a	0.0	1.0	2.0	20.0	NaN
b	3.0	4.0	5.0	NaN	NaN
c	6.0	7.0	8.0	NaN	NaN
d	9.0	10.0	11.0	NaN	NaN
e	12.0	13.0	14.0	NaN	NaN
f	15.0	16.0	17.0	18.0	NaN
g	NaN	NaN	NaN	NaN	NaN

```
# создаем массив NumPy с одним значением NaN
>>> a = np.array([1, 2, np.nan, 3])
# создаем объект Series из массива
>>> s = pd.Series(a)
# средние значения массива и серии отличаются
>>> a.mean(), s.mean()


# показываем, как методы .sum(), .mean() и .cumsum()
# обрабатывают значения NaN
# на примере столбца c4 датафрейма df
>>> s = df.c4
>>> s.sum(), # значения NaN обработаны как 0

>>> s.mean() # NaN также обработаны как 0

# в методе .cumsum() значения NaN тоже обрабатываются как 0,
# но в итоговом объекте Series значения NaN сохраняются
>>> s.cumsum()

# при выполнении арифметических операций значение NaN
# будет перенесено в результат
>>> df.c4 + 1
```

	c1	c2	c3	c4	c5
a	0.0	1.0	2.0	20.0	NaN
b	3.0	4.0	5.0	NaN	NaN
c	6.0	7.0	8.0	NaN	NaN
d	9.0	10.0	11.0	NaN	NaN
e	12.0	13.0	14.0	NaN	NaN
f	15.0	16.0	17.0	18.0	NaN
g	NaN	NaN	NaN	NaN	NaN

```
# возвращаем новый датафрейм, в котором
# значения NaN заполнены нулями
>>> filled = df.fillna(0)
>>> filled
```

```
# значения NaN не учитываются при вычислении
# средних значений
>>> df.mean()
```

```
# после замены значений NaN на 0 получим
# другие средние значения
>>> filled.mean()
```

	c1	c2	c3	c4	c5
a	0.0	1.0	2.0	20.0	NaN
b	3.0	4.0	5.0	NaN	NaN
c	6.0	7.0	8.0	NaN	NaN
d	9.0	10.0	11.0	NaN	NaN
e	12.0	13.0	14.0	NaN	NaN
f	15.0	16.0	17.0	18.0	NaN
g	NaN	NaN	NaN	NaN	NaN

```
>>> df.c4

a    20.0
b     NaN
c     NaN
d     NaN
e     NaN
f    18.0
g     NaN
Name: c4, dtype: float64

# заполняем пропуски в столбце c4 датафрейма df
# в прямом порядке
>>> df.c4.fillna(method="ffill")

a    20.0
b    20.0
c    20.0
d    20.0
e    20.0
f    18.0
g    18.0
Name: c4, dtype: float64
```

```
# выполняем обратное заполнение
>>> df.c4.fillna(method="bfill")

a    20.0
b    18.0
c    18.0
d    18.0
e    18.0
f    18.0
g     NaN
Name: c4, dtype: float64
```

```
# создаем новую серию значений, которую используем
# для заполнения значений NaN там,
# где метки индекса будут совпадать
>>> fill_values = pd.Series([100, 101, 102], index=['a', 'e', 'g'])
>>> fill_values
a    100
e    101
g    102
dtype: int64

# заполняем пропуски в столбце c4 с помощью fill_values
# a, e и g будут заполнены, поскольку метки совпали,
# однако значение a не изменится, потому что оно
# не является пропуском
>>> df.c4.fillna(fill_values)
a     20.0
b      NaN
c      NaN
d      NaN
e    101.0
f     18.0
g    102.0
Name: c4, dtype: float64
```

```
>>> df.c4

a    20.0
b     NaN
c     NaN
d     NaN
e     NaN
f    18.0
g     NaN
Name: c4, dtype: float64
```

```
>>> df
```

	c1	c2	c3	c4	c5
a	0.0	1.0	2.0	20.0	NaN
b	3.0	4.0	5.0	NaN	NaN
c	6.0	7.0	8.0	NaN	NaN
d	9.0	10.0	11.0	NaN	NaN
e	12.0	13.0	14.0	NaN	NaN
f	15.0	16.0	17.0	18.0	NaN
g	NaN	NaN	NaN	NaN	NaN

```
# заполняем значения NaN в каждом столбце
# средним значением этого столбца
>>> df.fillna(df.mean())
```

	c1	c2	c3	c4	c5
a	0.0	1.0	2.0	20.0	NaN
b	3.0	4.0	5.0	19.0	NaN
c	6.0	7.0	8.0	19.0	NaN
d	9.0	10.0	11.0	19.0	NaN
e	12.0	13.0	14.0	19.0	NaN
f	15.0	16.0	17.0	18.0	NaN
g	7.5	8.5	9.5	19.0	NaN

Интерполяция пропущенных значений

```
# выполняем линейную интерполяцию
# значений NaN с 1 по 2
>>> s = pd.Series([1, np.nan, np.nan, np.nan, 2])
>>> s.interpolate()
0    1.00
1    1.25
2    1.50
3    1.75
4    2.00
dtype: float64

>>> import datetime
>>> from datetime import datetime, date

# создаем временной ряд, но при этом значение
# по одной дате будет пропущено
>>> sts = pd.Series([1, np.nan, 2],
                    index=[datetime(2014, 1, 1),
                           datetime(2014, 2, 1),
                           datetime(2014, 4, 1)])
>>> sts
2014-01-01    1.0
2014-02-01    NaN
2014-04-01    2.0
dtype: float64
```

```
# выполняем линейную интерполяцию на основе
# количества элементов в данной серии
>>> ts.interpolate()
2014-01-01    1.0
2014-02-01    1.5
2014-04-01    2.0
dtype: float64

# этот программный код учитывает тот факт,
# что у нас отсутствует запись для 2014-03-01
>>> ts.interpolate(method="time")
2014-01-01    1.000000
2014-02-01    1.344444
2014-04-01    2.000000
dtype: float64
```

```
# создаем объект Series, чтобы продемонстрировать интерполяцию,
# основанную на индексных метках
>>> s = pd.Series([0, np.nan, 100], index=[0, 1, 10])
>>> s
0       0.0
1       NaN
10    100.0
dtype: float64

# выполняем линейную интерполяцию
>>> s.interpolate()
0       0.0
1      50.0
10    100.0
dtype: float64

# выполняем интерполяцию на основе значений индекса
>>> s.interpolate(method="values")
0       0.0
1      10.0
10    100.0
dtype: float64
```

Обработка дублирующихся данных

```
# создаем датафрейм с дублирующимися строками
>>> data = pd.DataFrame({'a': ['x'] * 3 + ['y'] * 4,
                         'b': [1, 1, 2, 3, 3, 4, 4]})
>>> data
```

	a	b
0	x	1
1	x	1
2	x	2
3	y	3
4	y	3
5	y	4
6	y	4

```
# определяем, какие строки являются дублирующимися,
# то есть какие строки уже ранее встречались в датафрейме
>>> data.duplicated()
```

```
# удаляем дублирующиеся строки, каждый раз оставляя
# первое из дублирующихся наблюдений
>>> data.drop_duplicates()
```

	a	b
0	x	1
2	x	2
3	y	3
5	y	4

```
# удаляем дублирующиеся строки, каждый раз оставляя
# последнее из дублирующихся наблюдений
>>> data.drop_duplicates(keep='last')
```

	a	b
1	x	1
2	x	2
4	y	3
6	y	4

```
# добавляем столбец c со значениями от 0 до 6
# метод .duplicated() сообщает об отсутствии
# дублирующихся строк
>>> data['c'] = range(7)
>>> data.duplicated()
```

```
# но если мы укажем, что нужно удалить дублирующиеся строки
# с учетом значений в столбцах a и b,
# результаты будут выглядеть так
>>> data.drop_duplicates(['a', 'b'])
```

	a	b	c
0	x	1	0
2	x	2	2
3	y	3	3
5	y	4	5

Сопоставление значений другим значениям

```
# создаем два объекта Series для иллюстрации
# процесса сопоставления значений
>>> x = pd.Series({"one": 1, "two": 2, "three": 3})
>>> x
one      1
two      2
three    3
dtype: int64

>>> y = pd.Series({1: "a", 2: "b", 3: "c"})
>>> y
1    a
2    b
3    c
dtype: object

# сопоставляем значения серии x значениям серии y
>>> x.map(y)
one      a
two      b
three    c
dtype: object
```

```
# если между значением серии y и индексной меткой серии x
# не будет найдено соответствие, будет выдано значение NaN
>>> x = pd.Series({"one": 1, "two": 2, "three": 3})
>>> y = pd.Series({1: "a", 2: "b"})
>>> x.map(y)

one        a
two        b
three    NaN
dtype: object
```

Замена значений

```
# создаем объект Series, чтобы проиллюстрировать
# метод .replace()
>>> s = pd.Series([0., 1., 2., 3., 2., 4.])
>>> s


# заменяем значение, соответствующее
# индексной метке 2, на значение 5
>>> s.replace(2, 5)


# заменяем все элементы новыми значениями
>>> s.replace([0, 1, 2, 3, 4], [4, 3, 2, 1, 0])


# заменяем элементы, используя словарь
>>> s.replace({0: 10, 1: 100})
```