1. pandas概述

- Numpy主要处理结构化数据，数据量比较小，规则
- 对于大量数据， 需要清理的数据，则需要pandas
- 一般使用方法是 `import pandas as pd`

2. pandas简单使用

本章主要介绍pandas三大件：

- Series
- DataFrame
- Index

2.1 pandas.Series对象

是一个带索引数据构成的一维数组。

Series给数组和一组索引绑定在一起。

如果想获取绑定的内容，分别可以使用values属性和index属性。

2.1.1 Series的创建

岁Series对象的创建主要分为：

- 使用数组直接创建,此时索引为默认索引
- 使用数组创建，显式指定索引
- 使用字典创建

```
# Series对象的创建
# 直接使用数组进行创建

import pandas as pd

data = pd.Series([10, 11, 12, 13,14])

print("完整Series内容是：\n", data)

print("\nSeries的值：  \n\n", data.values)

print("\nSeries的索引是：  \n", data.index)
```

```
完整Series内容是：
```

```
 0    10
1    11
2    12
3    13
4    14
dtype: int64

Series的值:

 [10 11 12 13 14]

Series的索引是:
 RangeIndex(start=0, stop=5, step=1)
```

```
# 显式制定索引

data_index = pd.Series([10,11,12,13,14], index=[1,2,3,4,5])

print("data_index: \n", data_index)
```

```
data_index:
 1    10
2    11
3    12
4    13
5    14
dtype: int64
```

```
#使用字典进行创建

d = {"one":1, "two":2, "three":3, "four":4, "five":5}
d_s = pd.Series(d)
print(d_s)
```

```
one      1
two      2
three    3
four     4
five     5
dtype: int64
```

```
# 对于字典，可以通过index来显式筛选内容

d = {4:"four", 5:"five", 1:"one", 3:"trhee", 2:"two"}
d_s = pd.Series(d,  index=[2,1,3])
print(d_s)
```

```
2      two
1      one
3    trhee
dtype: object
```

2.1.2 Series对象的简单使用

Series和Numpy的数组很像，他们的区别主要体现在索引上：

- Numpy通过隐士索引来对数据进行操作
- Series通过显式索引来将索引和数值关联

显式索引具有更强大的功能，你可以对其进行修改操作：

Series的values和indexs可以直接使用，查看。
请注意这俩的类型。

```
import pandas as pd

data = pd.Series([10,11,12,13,14], index=[1,2,3,9,5])

print("data.values=", data.values)
print(type(data.values))

print()
print("data.indexs=", data.index)
print(type(data.index))
```

```
data.values= [10 11 12 13 14]
<class 'numpy.ndarray'>

data.indexs= Int64Index([1, 2, 3, 9, 5], dtype='int64')
<class 'pandas.core.indexes.numeric.Int64Index'>
```

```
### 对于series的对象的访问，完全可以像数组那样使用

# 数据选择使用显式索引
print("第一个值是：", data[1])

# 隐式索引
print("\n 前三个值是：\n ", data[:9])
print("\n 前三个值是：\n ", data[:4])
```

```
第一个值是：  10

 前三个值是：
  1    10
2    11
3    12
9    13
5    14
dtype: int64

 前三个值是：
  1    10
2    11
3    12
9    13
dtype: int64
```

```
# 对于不按顺序排列的index同样可以
# 请注意下例data[:2]是到index=2的那个位置结束，并不是从0开始的下标直到2结束

data = pd.Series([10, 11, 12, 13, 14], index=[54, 32, 2, 1, 9])
print("不连续数字作为索引也是可以的：\n", data)

print("\n data3  =", data[32])

print("\n data[:1] = \n", data[:1])
```

不连续数字作为索引也是可以的：

```
 54     10
32     11
2      12
1      13
9      14
dtype: int64

 data3  = 11

 data[:1] =
 54     10
dtype: int64
```

```
# 对于自定义index同样可以使用切片
# ！！！注意此时切片的结束位置，包含结束位置!!!

d = {"one":1, "two":2, "three":3, "four":4, "five":5}
d_s = pd.Series(d)

print('d_s["four"] = ', d_s["four"])
print('d_s[":four"] = \n', d_s[:"four"])
```

```
d_s["four"] =  4
d_s[":four"] =
 one      1
two      2
three    3
four     4
dtype: int64
```

对数据的选取，Series处理的并不是特别好，在对数据访问的时候，显式索引和隐士索引容易造成混淆。

为了应对这种混乱，Python为我们提供了三个索引器，用以清晰访问数据：

- loc: 只是用显式索引, label based indexing
- iloc：只使用隐士索引, positional indexing
- ix：前两种索引的混合模式，主要用在DataFrame中, 不推荐

首先我们观察一下对Series的访问的显式索引和隐士索引的使用。

```
# Series对显式索引和隐士索引的使用

s = pd.Series([11,12,13,14,15], index=[3,5,7,9,1])
```

```
print("s = \n", s)

# Series对于单个内容的访问，采用的是显式索引
# s[1]表示的是显式索引 ”1“ 的内容
print("\n s[1]=", s[1])


# Series切片采用的是隐士所含，即默认是从下标0开始的升序索引
# s[1:3] 选中的内容是s[1] 和 s[2]
print("\n s[1:3] = \n", s[1:3])


s = pd.Series([1,2,3,4,5], index=["one", "two", "three", "four", "five"])
print("\n s=\n", s)


# 对于显式索引的切片， 是包含最后一位内容的
# 这一点跟隐士索引有很大区别
print('\n s["two": "four"]=', s["two":"four"])
```

```
s =
 3    11
5    12
7    13
9    14
1    15
dtype: int64

 s[1]= 15

 s[1:3] =
 5    12
7    13
dtype: int64

 s=
 one      1
two      2
three    3
four     4
five     5
dtype: int64

 s["two": "four"]= two      2
three    3
four     4
dtype: int64
```

```
# loc索引器表示切片和取值都是显式的，不使用隐士索引。

s = pd.Series([11,12,13,14,15], index=[3,5,7,9,1])
print("s = \n", s)

print("\n s.loc[3] = ", s.loc[3])
print("\n s.loc[3:9] = \n", s.loc[3:9])
```

```
s =
 3    11
5    12
7    13
9    14
1    15
dtype: int64

 s.loc[3] =  11

 s.loc[3:9] =
 3    11
5    12
7    13
9    14
dtype: int64
```

```
# iloc索引器表示切片和取值都是隐士的，不使用显式索引
s = pd.Series([11,12,13,14,15], index=[3,5,7,9,1])
print("s = \n", s)

print("\n s.iloc[3] = ", s.iloc[3])
print("\n s.iloc[3:9] = \n", s.iloc[3:9])
```

```
s =
 3    11
5    12
7    13
9    14
1    15
dtype: int64

 s.iloc[3] =  14

 s.iloc[3:9] =
```

```
 9    14
1    15
dtype: int64
```

2.2 DataFrame对象

- DataFrame可以看做是通用的NumPy数组，也可以看做特殊的字典
- DataFrame最常见的结构可以想象成一个Excel内容，每一行都有行号，每一列都有列名的二维结构

创建DataFrame的方式比较多，常见的有：

- 通过单个Series创建
- 通过字典列表创建
- 通过Series对象字典创建
- 通过NumPy二维数组创建
- 通过Numpy结构化数组创建

```
# 通过单个Series对象创建

import pandas as pd

s = pd.Series([1,2,3,4,5])
print("S=\n", s)

df = pd.DataFrame(s, columns=['digits'])
print("df=\n", df)
```

```
S=
 0    1
1    2
2    3
3    4
4    5
dtype: int64
df=
    digits
0        1
1        2
2        3
3        4
4        5
```

```
# 通过字典列表创建

dl = [{"个": i, "十":i*10, "百":i*100} for i in range(1,5)]
```

```
print("dl = ", dl)

df = pd.DataFrame(dl)
print("df = \n", df)


# 在通过字典创建的时候，如果有的值并不存在，则自动用NaN填充
# 参看下面例子
dl = [{"a":1, "b":1}, {"b":2, "c":2}, {"c":3, "d":3}]
df = pd.DataFrame(dl)
print("df = \n", df)
```

```
dl =  [{'个': 1, '十': 10, '百': 100}, {'个': 2, '十': 20, '百': 200}, {'个': 3,
'十': 30, '百': 300}, {'个': 4, '十': 40, '百': 400}]
df =
    个   十    百
0   1  10  100
1   2  20  200
2   3  30  300
3   4  40  400
df =
      a    b    c    d
0   1.0  1.0  NaN  NaN
1   NaN  2.0  2.0  NaN
2   NaN  NaN  3.0  3.0
```

```
# 通过Series对象字典创建

s1 = pd.Series([i for i in range(1,6)], index=[1,2,3,4,5])
s2 = pd.Series([i*10 for i in range(1,6)], index=[3,4,5,6,7])

df = pd.DataFrame({"个": s1, "十":s2})
print("df = \n", df)
```

```
df =
      个     十
1   1.0   NaN
2   2.0   NaN
3   3.0  10.0
4   4.0  20.0
5   5.0  30.0
6   NaN  40.0
7   NaN  50.0
```

```
# 通过Numpy二维数组创建
import numpy as np

df = pd.DataFrame(np.zeros([5,3]),
                  columns=["A", "B", "C"],
                 index=["one", "two", "three", "four", "five"])
print("df=\n",df)
```

```
df=
           A    B    C
one      0.0  0.0  0.0
two      0.0  0.0  0.0
three    0.0  0.0  0.0
four     0.0  0.0  0.0
five     0.0  0.0  0.0
```

```
# 通过Numpy结构化数组

d = np.zeros(3, dtype=[("A", "i8"), ("B", "f8")])
print("d = \n", d)

df = pd.DataFrame(d)
print("df=\n", df)
```

```
d =
 [(0, 0.) (0, 0.) (0, 0.)]
df=
    A    B
0   0  0.0
1   0  0.0
2   0  0.0
```

对于DataFrame数据的使用方式，我们以前说过，可以把DataFrame看做是具有行列号和首部标题行的Excel表格，而去除掉列好和首部标题行
DataFrame就可以看做是一个二维数组。

对于DataFrame的数据选择，可以采用字典形式的访问，此时访问的是一列值，也可以采用切片等，下面详细进行介绍：

```
dl = [{"个": i, "十":i*10, "百":i*100} for i in range(1,5)]


df = pd.DataFrame(dl)
print("df = \n", df)
```

```
df =
    个   十    百
0   1   10   100
1   2   20   200
2   3   30   300
3   4   40   400
```

```
# 使用字典的方式访问

print("df['百'] =\n",  df['百'] )
```

```
df['百'] =
 0    100
1    200
2    300
3    400
Name: 百, dtype: int64
```

如果选取的键的名字跟上例中的df的属性或者函数不冲突，可以直接采用圆点符号进行访问：

```
# 上面访问方式等价于下面访问方式
# 但下面访问方式非通用和方法，可能会引起冲突
print("df.百=\n", df.百)
```

```
df.百=
 0    100
1    200
2    300
3    400
Name: 百, dtype: int64
```

DataFrame可以看做是一个增强版的二维数组，此时他的全部值可以用DataFrame.values来表示：

```
# values属性的使用
print("df.values = \n", df.values)
```

```
df.values =
 [[  1  10 100]
 [  2  20 200]
 [  3  30 300]
 [  4  40 400]]
```

对于DataFrame的访问，推荐时候用loc，iloc或者ix三个所引器进行访问，避免引起混淆：

```
# 使用loc进行显示访问
print('df.loc[:2, :"十"]=\n', df.loc[:2, :"十"])
```

```
df.loc[:2, :"十"]=
    个   十
0   1   10
1   2   20
2   3   30
```

```
# 时候iloc进行隐士访问
print('df.iloc[:2, :2] = \n', df.iloc[:2, :2])
```

```
df.iloc[:2, :2] =
    个   十
0   1   10
1   2   20
```

```
# 或者使用ix进行混合访问
# indexing的初衷是避免访问混淆，但ix显然并没有达到这一点，所以，ix索引器不推荐使用


print(df.ix[:2, :"十"])
```

```
   个   十
0   1   10
1   2   20
2   3   30
```

```
/Users/augs/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:5:
DeprecationWarning:
.ix is deprecated. Please use
.loc for label based indexing or
.iloc for positional indexing

See the documentation here:
http://pandas.pydata.org/pandas-docs/stable/indexing.html#ix-indexer-is-
deprecated
  """
```

2.3 pandas.Index对象

Pandas的Index对象是一个独立的对象，用来表示数据的索引，可以把它看做不可变的数组(tuple)或者有序的集合。

当作为不可变数组的时候，除了数组的一些读操作外，还具有一些NumPy数组的属性。

Index作为有序集合操作主要是为了进行一些基于集合的操作，比如集合的交差并补操作。

```
# Idnex作为不可变数组

idx = pd.Index([2,4,6,8,10])
print("idx = ", idx)
print("idx[1:4] = ", idx[1:4])
print("idx.size=", idx.size)
print("idx.shape=", idx.shape)
print("idx.ndim", idx.ndim)
print("idx.dtype=", idx.dtype)
```

```
idx =  Int64Index([2, 4, 6, 8, 10], dtype='int64')
idx[1:4] =  Int64Index([4, 6, 8], dtype='int64')
idx.size= 5
idx.shape= (5,)
idx.ndim 1
idx.dtype= int64
```

```
# Index作为有序集合

idx_1 = pd.Index([1,3,5,6,7,])
idx_2 = pd.Index([2,4,6,7,8,9])

print("交集：  idx_1 & idx_2 = ", idx_1 & idx_2)
print("并集：  idx_1 | idx_2 = ", idx_1 | idx_2)
# print("差集：  idx_1 - idx_2 = ", idx_1.intersection( idx_2))
print("异或：  idx_1 ^ idx_2 = ", idx_1 ^ idx_2)
```

```
交集：  idx_1 & idx_2 =  Int64Index([6, 7], dtype='int64')
并集：  idx_1 | idx_2 =  Int64Index([1, 2, 3, 4, 5, 6, 7, 8, 9], dtype='int64')
异或：  idx_1 ^ idx_2 =  Int64Index([1, 2, 3, 4, 5, 8, 9], dtype='int64')
```

3. Pandas的运算方法

Pandas基于Numpy，相应的运算也是基于Numpy的运算，只不过多了一些Pandas的内容，比如运算结果保留索引和列标签，传递通用函数的时候回自动对齐索引等。

3.1 对通用函数保留索引和列标签

```
import numpy as np
import pandas as pd

df = pd.DataFrame(np.random.randint(100, size=(3,5)), columns=["A", "B", "C",
"D", "E"])

print("df=\n", df)

#如果对df使用通用函数，成成的结果是保留索引和列标签的
df2 = np.exp(df)
print("df2 \n", df2)
```

```
df=
     A   B   C   D   E
0  77  86  86  53  98
1  56  92  27   0  83
2   6  66   7  54  44
df2
                A             B             C             D             E
0  2.758513e+33  2.235247e+37  2.235247e+37  1.041376e+23  3.637971e+42
1  2.091659e+24  9.017628e+39  5.320482e+11  1.000000e+00  1.112864e+36
2  4.034288e+02  4.607187e+28  1.096633e+03  2.830753e+23  1.285160e+19
```

3.2 自动对齐索引

在对Series或者DataFrame进行二元运算的时候， Pandas会在计算过程中对齐两边索引，这对于不完整数据的处理极其重要。

在运算工程中，对于缺失值的处理采用默认缺失值处理方法，对于我们一般是添加NaN。如果想指定缺失值的填充内容，需要：

- 采用Pandas的运算方法，而不是使用运算符
- fill_value参数代表填充的内容

在指定fill_value的时候，需要注意点是，此时是先对参与运算的数据进行缺省值处理，然后才运算，这样很多因为一方是NaN而最终结果也是NaN的运算因为换了缺省值而能够正常运算。而不是先运算，得到缺省值后再处理。

最终结果的索引内容是两个运算索引的并集。

```
# Series的索引自动对齐
# 此处对于缺失值的处理采用默认方法，即对于二元运算方法
# 只要由一方没有数据，则用NaN填充，任何数据与NaN运算结果都是NaN

s1 = pd.Series({"A": 1, "B":2, "C":3, "D":4, "E":5}, name="ONE")
s2 = pd.Series({ "D":4, "E":5, "F":6, "G":7}, name="TWO")

# 采用运算符，此时缺失值只能使用默认的值
print("s1 + s2 =\n", s1 + s2)

# 想更换缺失值的处理内容，需要用到pandas的运算方法和fill_value参数
print("\n s1 + s2 =\n", s1.add(s2, fill_value=100))
```

```
s1 + s2 =
 A     NaN
B     NaN
C     NaN
D     8.0
E    10.0
F     NaN
G     NaN
dtype: float64

 s1 + s2 =
 A    101.0
B    102.0
C    103.0
D      8.0
E     10.0
F    106.0
G    107.0
dtype: float64
```

```python
# 缺失值和索引对其的DataFramne案例

df1 = pd.DataFrame(np.random.randint(10, size=(3,3)), index=list("ABC"),
columns=['I', 'II', 'III'])
print("df1 = \n", df1)


df2 = pd.DataFrame(np.random.randint(100, 200, size=(3,3)), index=list("CDE"),
columns=['II', 'III', "IV"])
print("df1 = \n", df1)

# 使用操作符，index和colums保留， 缺失值采用默认方法处理
df3 = df1 + df2
print("\n df3 = \n", df3)


# 使用pandas方法，可以指定缺省值
df4 = df1.add(df2, fill_value=0)
print("\n df4 = \n", df4)
```

```
df1 =
    I  II  III
A  9   6    7
B  1   9    2
C  3   1    3
df1 =
    I  II  III
A  9   6    7
B  1   9    2
C  3   1    3

 df3 =
     I     II    III  IV
A NaN    NaN    NaN NaN
B NaN    NaN    NaN NaN
C NaN  180.0  138.0 NaN
D NaN    NaN    NaN NaN
E NaN    NaN    NaN NaN

 df4 =
      I     II    III     IV
A  9.0    6.0    7.0    NaN
B  1.0    9.0    2.0    NaN
C  3.0  180.0  138.0  160.0
D  NaN  108.0  108.0  155.0
E  NaN  182.0  145.0  114.0
```

3.3 DataFrame和Series的运算

DataFrame和Series运算默认采用的是行运算，即一行一行的运算，如果想要按列运算，需要使用axis参数。

```
# 默认是按行来进行计算
A1 = np.random.randint(10, size=(3,5))
print("A1 = \n", A1)

print("\n A1 - A1[1] = \n", A1 - A1[1])
```

```
A1 =
 [[7 6 4 8 2]
 [9 0 1 2 7]
 [5 9 1 5 0]]

 A1 - A1[1] =
 [[-2  6  3  6 -5]
 [ 0  0  0  0  0]
 [-4  9  0  3 -7]]
```

```
# 如果想按列运算，可以使用axis参数

df1 = pd.DataFrame(A1, columns=list("ABCDE"))
print("df1 = \n", df1)

# 默认按行计算
df2 = df1 - df1.iloc[1]
print("\n df2 = \n", df2)

# 按列运算需要使用axis参数
df3 = df1.subtract(df["B"], axis=0)
print("\n df3 = \n", df3)
```

```
df1 =
    A  B  C  D  E
0  7  6  4  8  2
1  9  0  1  2  7
2  5  9  1  5  0

 df2 =
    A  B  C  D  E
0 -2  6  3  6 -5
1  0  0  0  0  0
```

```
2 -4  9  0  3 -7

 df3 =
     A   B   C   D   E
0 -79 -80 -82 -78 -84
1 -83 -92 -91 -90 -85
2 -61 -57 -65 -61 -66
```

4. 缺失值的处理

面对大量数据经常会出现残缺不全的情况，面对这样的数据我们需要对缺失值进行预先处理。

处理缺失值一般采用两种方法：

- None： 对象类型内容的默认缺失值
- NaN： 数值类型的默认缺失值，用numpy.nan表示

需要注意的是：

- 不同语言/工具可能对缺失值的默认处理方式不同，所以，如果拿到的是别的处理过的文件，可能需要对已经替换过的缺失值再进行替换一次。
- NaN的使用具有传染性，就是任何数据跟NaN进行操作，最终的结果都是NaN， 性质很像乘法中的0，所以在有的操作函数中，numpy提供了另外一套操作，对NaN缺失值自动过滤掉，而不是让他传染给所有结果
- None和NaN可以互换，或者可以认为这两个值在Pandas中等价

4.1 对NaN不敏感的操作函数

有时候NaN的传染性并不是让人喜欢，他会导致我们的结果出现重大偏差甚至错误，所以在处理的时候我们还提供了另外一套对NaN不敏感的函数：

- np.nansum
- np.nanmin
- np.nanmax

```
# 创建带有NaN的数组
a = np.array([3, 2, 3, np.nan, 4])
print("a = \n", a)

print()
# 对于普通的函数操作，NaN具有传染性
print("np.sum = ", np.sum(a))
print("np.min = ", np.min(a))
print("np.max = ", np.max(a))b'

# 如果想避免NaN传染性带来的问题，需要使用相应NaN非敏感函数
print("np.nansum = ", np.nansum(a))
```

```
print("np.nanmin = ", np.nanmin(a))
print("np.nanmax = ", np.nanmax(a))
```

```
a =
 [ 3.  2.  3. nan  4.]

np.sum =  nan
np.min =  nan
np.max =  nan
np.nansum =  12.0
np.nanmin =  2.0
np.nanmax =  4.0
```

```
/Users/augs/anaconda3/lib/python3.7/site-
packages/numpy/core/fromnumeric.py:83: RuntimeWarning: invalid value
encountered in reduce
  return ufunc.reduce(obj, axis, dtype, out, **passkwargs)
```

4.2 缺失值的发现

发现缺失值可以使用两个函数：

- isnull
- notnull

每个函数都返回布尔类型的掩码数据。

```
# 准备的实验数据
data = pd.Series([23, np.nan, "hahaha", "two", 23, None])
print("data = \n ", data)
```

```
data =
  0        23
1       NaN
2    hahaha
3       two
4        23
5      None
dtype: object
```

```
# isnull用来判断是否是空

# isnull返回的是布尔值
print("\n data.isnull = \n", data.isnull())

# notnull返回的是布尔值
print("\n data.notnull = \n", data.notnull())

# 可以使用这个作为掩码操作
print("\n data[data.notnull()] = \n", data[data.notnull()])
```

```
 data.isnull =
 0    False
1     True
2    False
3    False
4    False
5     True
dtype: bool

 data.notnull =
 0     True
1    False
2     True
3     True
4     True
5    False
dtype: bool

 data[data.notnull()] =
 0        23
2    hahaha
3       two
4        23
dtype: object
```

4.3 剔除缺失值

对于缺失值的处理一般是剔除或者用一个特殊的值来进行填充：

- dropna: 剔除缺失值,常用的两个参数：
  - axis： 控制剔除行或者列
  - thresh：低于这个数量的数据的行或者列剔除
- fillna: 缺失值用别的值进行填充

```
# 剔除缺失值
# 剔除后只留下原来的真正的值
print("\n data.dropna = \n", data.dropna())
```

```
 data.dropna =
 0        23
2    hahaha
3       two
4        23
dtype: object
```

```
# 需要主要的是，如果是二维数据，剔除后需要把整行都剔除，不仅仅是把NaN的一个值剔除
df = pd.DataFrame([[1, np.nan, 3],
                   [2, 4, 9],
                   [np.nan, np.nan, 9]])
print("\n df.dropna = \n", df.dropna())
```

```
 df.dropna =
      0    1  2
1  2.0  4.0  9
```

```
# 如果改变剔除的默认行为，比如剔除包含NaN的行，则需要axis参数
print("\n df.dropna(axis='columns') = \n", df.dropna(axis='columns'))
```

```
 df.dropna(axis='columns') =
    2
0  3
1  9
2  9
```

```
# thresh参数
# 数据列中，低于thresh=2的列被剔除
print("\n df.dropna(axis='columns') = \n", df.dropna(axis='columns',
thresh=2))
```

```
 df.dropna(axis='columns') =
      0  2
0  1.0  3
1  2.0  9
2  NaN  9
```

4.4 填充缺失值

对于缺失值一般使用换一个特定的值进行填充就可以。填充的时候可以使用一些特定的值，比如0，-1等，也可以使用一些处理后的值，比如填充(imputation)或者转换(interpolation)之后的数据。

填充函数是fillna，常用参数为：

- axis: 坐标轴，行或者列
- method: 填充方式
  - ffill: forward-fill, 从前向后填充
  - bfill: backward-fill, 从后向前填充

```
# 准备数据
df = pd.DataFrame([[1, np.nan, 3],
                   [2, 4, 9],
                   [np.nan, np.nan, 9]])
print("\n df = \n", df)
```

```
 df =
      0    1  2
0  1.0  NaN  3
1  2.0  4.0  9
2  NaN  NaN  9
```

```
# 使用某一个值进行填充
print("df.fillna = \n", df.fillna(-1))
```

```
df.fillna =
      0    1  2
0  1.0 -1.0  3
1  2.0  4.0  9
2 -1.0 -1.0  9
```

```
# forward-fill
print("df.fillna(ffill) = \n", df.fillna( method="ffill"))
```

```
df.fillna(ffill) =
      0    1  2
0  1.0  NaN  3
1  2.0  4.0  9
2  2.0  4.0  9
```

```
# backward-fill
print("df.fillna(bfill) = \n", df.fillna(axis=1,  method="bfill"))
```

```
df.fillna(bfill) =
      0    1    2
0  1.0  3.0  3.0
1  2.0  4.0  9.0
2  9.0  9.0  9.0
```

5 层级索引

处理多维数据的时候，虽然Pandas提供了Panel和Panel4D， 但更直观的是使用层级索引(Hierarchical Indeing,也叫多级索引 multi-indexing), 通过层级索引，可以将高维度数据转换成类似以为Series或者二维DataFrame对象的形式。

5.1 层级索引的创建

i. 直接创建

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

```
# 通过输入层级索引，直接创建Series

# 我们创建和一个翻译数组吧，分别包含1-5的汉语，英语，德语的叫法

# 用元组信息作为index
idx = [("yi", "en"),  ("yi", "de"),
       ("er", "en"),    ("er", "de"),
         ("san", "en"),   ("san", "de"),
         ("si", "en"),   ("si", "de"),
         ("wu", "en"),   ("wu", "de")]
# 内容是元组对应的每一个英语或者德育的翻译
digits = ["ONE", "EINS", "TWO", "ZWEI", "THREE", "DREI", "FOUR", "VIER",
```

```
          "FIVE", "FUNF"]

# 创建一维的数组
trans = pd.Series(digits, index=idx)

# 查看打印的结果
print("trans = \n", trans)

# 这个一维数组可以支持常常的比如取值，切片操作
print("\n trans[('san', 'en')] = ", trans[('san', 'en')] )
print("\n trans[('san', 'en'):('wu','en')] = \n",  trans[('san', 'en'):
('wu','en')] )
```

```
trans =
 (yi, en)       ONE
(yi, de)      EINS
(er, en)       TWO
(er, de)      ZWEI
(san, en)    THREE
(san, de)     DREI
(si, en)      FOUR
(si, de)      VIER
(wu, en)      FIVE
(wu, de)      FUNF
dtype: object

 trans[('san', 'en')] =  THREE

 trans[('san', 'en'):('wu','en')] =
 (san, en)    THREE
(san, de)     DREI
(si, en)      FOUR
(si, de)      VIER
(wu, en)      FIVE
dtype: object
```

ii. 使用多级索引创建

可以先创建多级MultiIndex，然后利用多级索引来创建内容。

MultiIndex是一个包含levels的结构，levels表示的是索引的层级，每个层级分别有哪些内容等等。

```
# 创建多级索引
# 使用上面的数据idx


midx = pd.MultiIndex.from_tuples(idx)
print( "多级索引midx=\n", midx)
```

```
多级索引midx=
 MultiIndex(levels=[['er', 'san', 'si', 'wu', 'yi'], ['de', 'en']],
           labels=[[4, 4, 0, 0, 1, 1, 2, 2, 3, 3], [1, 0, 1, 0, 1, 0, 1, 0, 1,
0]])
```

```
# 然后利用创建的多级索引创建数据
trans = pd.Series(digits, index=midx)
print("\n trans = \n", trans)

# 然后可以使用切片等功能获取相应内容
# 注意：因为索引不是按照字典排序，所以不能使用切片 ！！！
print("\n trans['san', 'de'] = ", trans['san', 'de'])
```

```
 trans =
 yi   en      ONE
     de     EINS
er   en      TWO
     de     ZWEI
san  en    THREE
     de     DREI
si   en     FOUR
     de     VIER
wu   en     FIVE
     de     FUNF
dtype: object

 trans['san', 'de'] =  DREI
```

iii. 利用Series创建二维DataFrame

pandas给我们提供了一对函数，stack和unstack，可以让Series和DataFrame相互转换。

```
#数据直接使用一维的trans
trans_df = trans.unstack()
print("折叠后的二维数据是：")
print("trans_df = \n", trans_df)
```

```
折叠后的二维数据是：
trans_df =
        de     en
er    ZWEI    TWO
san   DREI  THREE
si    VIER   FOUR
wu    FUNF   FIVE
yi    EINS    ONE
```

IV. 创建数据的时候直接时候用二维索引

```
trans_df = pd.DataFrame(digits, index=[["YI", "YI", "ER", "ER", "SAN",
"SAN","SI", "SI", "WU", "WU"],
                                       ["EN", "DE","EN", "DE","EN", "DE","EN",
"DE","EN", "DE"]],
                        columns=["TRANS"])
print(trans_df)
```

```
         TRANS
YI  EN     ONE
    DE    EINS
ER  EN     TWO
    DE    ZWEI
SAN EN   THREE
    DE    DREI
SI  EN    FOUR
    DE    VIER
WU  EN    FIVE
    DE    FUNF
```

5.2 多级索引

i. 显式创建多级索引

```
# 通过数组创建
midx = pd.MultiIndex.from_arrays([list("AAABBBCCC"), [1,2,3,1,2,3,1,2,3]])
print("通过数组创建：midx = \n", midx)
```

```
通过数组创建：midx =
 MultiIndex(levels=[['A', 'B', 'C'], [1, 2, 3]],
           labels=[[0, 0, 0, 1, 1, 1, 2, 2, 2], [0, 1, 2, 0, 1, 2, 0, 1, 2]])
```

```
#  通过tuple创建
midx = pd.MultiIndex.from_tuples([("A", 1),("A", 2), ("A", 3),
                                  ("B", 1),("B", 2), ("B", 3),
                                  ("C", 1),("C", 2), ("C", 3),])
print("通过元组创建：midx = \n", midx)
```

```
通过元组创建：midx =
 MultiIndex(levels=[['A', 'B', 'C'], [1, 2, 3]],
           labels=[[0, 0, 0, 1, 1, 1, 2, 2, 2], [0, 1, 2, 0, 1, 2, 0, 1, 2]])
```

```
# 通过笛卡尔积创建(CartesianProduct)
midx = pd.MultiIndex.from_product([list("ABC"), [1,2,3]])
print("通过笛卡尔积创建：midx = \n", midx)
```

```
通过笛卡尔积创建：midx =
 MultiIndex(levels=[['A', 'B', 'C'], [1, 2, 3]],
           labels=[[0, 0, 0, 1, 1, 1, 2, 2, 2], [0, 1, 2, 0, 1, 2, 0, 1, 2]])
```

ii. 多级索引的等级名称

对多级索引的每个等级进行命名可以方便以后的操作，一般可以直接通过对所含的属性names进行赋值得到。

```
midx.names =["NAME", "LEVEL"]
print("带名字的多级索引 midx = \n", midx)
```

```
带名字的多级索引 midx =
 MultiIndex(levels=[['A', 'B', 'C'], [1, 2, 3]],
           labels=[[0, 0, 0, 1, 1, 1, 2, 2, 2], [0, 1, 2, 0, 1, 2, 0, 1, 2]],
           names=['NAME', 'LEVEL'])
```

iii. 多级列索引

列索引也可以包含多级，通过多级行索引，列所含能实现多维数据的创建和操作。

```
# 行多级索引
midx_row = pd.MultiIndex.from_product([list("ABC"), [1,2,3]])

# 列多级索引
midx_col = pd.MultiIndex.from_product([["I", "II", "III"], [1,2]])

#准备数据
data = np.arange(54).reshape(9,6)

df = pd.DataFrame(data, index=midx_row, columns=midx_col)
print(df)
```

```
      I       II      III
      1   2   1   2   1   2
A 1   0   1   2   3   4   5
  2   6   7   8   9  10  11
  3  12  13  14  15  16  17
B 1  18  19  20  21  22  23
  2  24  25  26  27  28  29
  3  30  31  32  33  34  35
C 1  36  37  38  39  40  41
  2  42  43  44  45  46  47
  3  48  49  50  51  52  53
```

5.3 多级索引的的取值和切片

多级索引的取值和切片和和简单的数组的取值和切片很类似，我们这里先介绍Series的多级索引的取值和切片，在介绍DataFrame的取值和切片用法。

对多级索引使用切片的前提是：索引为有序索引！！！

i. Series的多级索引取值操作

```
# 创建数据

trans = pd.Series(digits, index=[["YI", "YI", "ER", "ER", "SAN", "SAN","SI",
"SI", "WU", "WU"],
                                 ["EN", "DE","EN", "DE","EN", "DE","EN",
"DE","EN", "DE"]],
                  )
print("Series的数据:  \n", trans_df)
```

```
Series的数据:
 YI    EN       ONE
      DE      EINS
ER    EN       TWO
      DE      ZWEI
SAN   EN     THREE
      DE      DREI
SI    EN      FOUR
      DE      VIER
WU    EN      FIVE
      DE      FUNF
dtype: object
```

```
# 对trans的取值操作

# 提取单个一级索引的值
print("trans['YI', 'DE'] = \n", trans['YI'])

# 使用二级索引提取单个的值
print("\n trans['YI', 'DE'] = ", trans['YI', 'DE'])
```

```
trans['YI', 'DE'] =
 EN     ONE
DE    EINS
dtype: object

 trans['YI', 'DE'] =  EINS
```

```
# 切片的前提是index必须有序
# 准备有序的index

data = np.arange(100, 700, 100)

midx = pd.MultiIndex.from_product([list("ABC"), [1,2]])
df = pd.Series(data, index=midx)

print("df = \n", df)
```

```
df =
 A  1    100
   2    200
B  1    300
   2    400
C  1    500
   2    600
dtype: int64
```

```
# 对一级索引进行切片

print("一级索引进行切片 df[:'B'] = \n", df[:'B'])
```

```
一级索引进行切片 df[:'B'] =
 A  1    100
   2    200
B  1    300
   2    400
dtype: int64
```

```
# 对二级索引选取
print("\n 二级索引进行切片 df.loc[:'B', :2] =\n", df.loc[:'B', :2])
```

```
 二级索引进行切片 df.loc[:'B', :2] =
 A  1    100
   2    200
B  1    300
   2    400
dtype: int64
```

```
# 掩码选取

print("y掩码选取数据 df[df < 400] = \n", df[df < 400])
```

```
y掩码选取数据 df[df < 400] =
 A  1    100
   2    200
B  1    300
dtype: int64
```

```
# 花哨索引

print(df[['A','C']].loc[:, 1])
```

```
A    100
C    500
dtype: int64
```

ii. DataFrame取值操作

DataFrame的切片操作比较麻烦，对元组的切片还容易引起错误，pandas给提供了专门的IndexSlice来专门帮助处理切片。

```
## 准备数据

# 行多级索引
midx_row = pd.MultiIndex.from_product([list("ABC"), [1,2,3]])

# 列多级索引
midx_col = pd.MultiIndex.from_product([["I", "II", "III"], [1,2]])

#准备数据
data = np.arange(54).reshape(9,6)

df = pd.DataFrame(data, index=midx_row, columns=midx_col)
print(df)
```

```
      I       II      III
      1   2   1   2   1   2
A 1   0   1   2   3   4   5
  2   6   7   8   9  10  11
  3  12  13  14  15  16  17
B 1  18  19  20  21  22  23
  2  24  25  26  27  28  29
  3  30  31  32  33  34  35
C 1  36  37  38  39  40  41
  2  42  43  44  45  46  47
  3  48  49  50  51  52  53
```

```
# 使用列索引获取数据
print("获取DF的一列 df['I',2] = \n", df['I', 2])
```

```
获取DF的一列 df['I',2] =
 A  1     1
    2     7
    3    13
B   1    19
    2    25
    3    31
C   1    37
    2    43
    3    49
Name: (I, 2), dtype: int64
```

```
# 最好使用索引器
print("获取DF的一列 df['I',2] = \n", df.loc['A'])
```

```
获取DF的一列 df['I',2] =
      I       II      III
      1   2   1   2   1   2
1     0   1   2   3   4   5
2     6   7   8   9  10  11
3    12  13  14  15  16  17
```

```
# 使用IndexSlice
idx = pd.IndexSlice
df.loc[idx[:, 2], idx[:'II', :]]
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead tr th {
    text-align: left;
}
```

		I		II	
		1	2	1	2
A	2	6	7	8	9
B	2	24	25	26	27
C	2	42	43	44	45

5.4 多级索引的行列转换

Pandas提供了很多方法，可以让数据在内容保持不变的情况下，按照需要进行行列转换。

i. 有序和无序索引

MultiIndex如果不是有序索引，则大多数切片可能失败。

为了让Index有序，我们一般使用sort_index方法对齐进行处理，使之有序后在进行操作。

```
# 准备无序MultiIndex的数据

trans = pd.Series(digits, index=[["YI", "YI", "ER", "ER", "SAN", "SAN","SI", "SI",
"WU", "WU"],
                                 ["EN", "DE","EN", "DE","EN", "DE","EN",
"DE","EN", "DE"]],
                  )
print("Series的数据: \n", trans)
```

```
Series的数据:
 YI   EN       ONE
      DE      EINS
ER    EN       TWO
      DE      ZWEI
SAN   EN     THREE
      DE      DREI
SI    EN      FOUR
      DE      VIER
WU    EN      FIVE
      DE      FUNF
dtype: object
```

```
# 上面trans是无序index，在对其进行切片操作的时候，会报错
# 使用sort_index进行排序后，进行切片操作就正常

trans2 = trans.sort_index()
print("排序后的 trans2[:'WU'] = \n", trans2[:'WU'] )
```

```
排序后的 trans2[:'WU'] =
 ER    DE      ZWEI
      EN       TWO
SAN   DE      DREI
      EN     THREE
SI    DE       VIER
      EN       FOUR
WU    DE       FUNF
      EN       FIVE
dtype: object
```

ii. stack和unstack

这两个互为逆操作。

```
# unstack把低维度数据扩展成高维度数据

trans4 = trans.unstack()
print("二维数据是： trans.unstack = \n", trans4)
```

```
二维数据是： trans.unstack =
         DE      EN
ER    ZWEI     TWO
SAN   DREI   THREE
SI    VIER    FOUR
WU    FUNF    FIVE
YI    EINS     ONE
```

```
# unstack可以按照指定级别进行扩展
trans4 = trans.unstack(level=0)
print("二维数据是： trans.unstack = \n", trans4)
```

```
二维数据是： trans.unstack =
        ER     SAN     SI     WU     YI
DE   ZWEI    DREI   VIER   FUNF   EINS
EN    TWO   THREE   FOUR   FIVE    ONE
```

iii. 索引的设置和重置

set_index和reset_index可以通过重新设置index来改变数据的表现形式。

```
# 吧索引当做数据插入到二维数组中

trans6 = trans.reset_index()
print("重置索引后数据 trans.reset_index = \n", trans6)
print(type(trans6))
```

```
重置索引后数据 trans.reset_index =
   level_0 level_1      0
0       YI       EN    ONE
1       YI       DE   EINS
2       ER       EN    TWO
3       ER       DE   ZWEI
4      SAN       EN  THREE
5      SAN       DE   DREI
6       SI       EN   FOUR
7       SI       DE   VIER
8       WU       EN   FIVE
9       WU       DE   FUNF
<class 'pandas.core.frame.DataFrame'>
```

```
# 可以使用drop参数来表明是否丢弃index
# 默认drop=False
trans7 = trans.reset_index(drop=True)
print("重置索引并丢弃后 trans.reset_index(drop=True) = \n", trans7)
```

```
重置索引并丢弃后 trans.reset_index(drop=True) =
 0      ONE
1     EINS
2      TWO
3     ZWEI
4    THREE
5     DREI
6     FOUR
7     VIER
8     FIVE
9     FUNF
dtype: object
```

```
# set_index是reset_index的逆操作
# 参数为必填项，必须明确指明哪个columns呗当做index
trans8 = trans6.set_index(['level_0'])
print("重置索引后的数据 trans6.set_index('level_0') = \n", trans8)
```

```
重置索引后的数据 trans6.set_index('level_0') =
         level_1      0
level_0
YI            EN    ONE
YI            DE   EINS
ER            EN    TWO
ER            DE   ZWEI
SAN           EN  THREE
SAN           DE   DREI
SI            EN   FOUR
SI            DE   VIER
WU            EN   FIVE
WU            DE   FUNF
```

5.5 多级索引的数据累计方法

对于多级索引，我们可能需要对某一个索引进行数据处理，比如求mean，sum等操作，此时需要用到参数level，通过对参数level的设置，pandas会选择相应的数据进行处理。

```
import numpy as np
import pandas as pd

## 准备数据

# 行多级索引
midx_row = pd.MultiIndex.from_product([list("ABC"), [1,2,3]], names=["Upper",
"row_digit"])

# 列多级索引
midx_col = pd.MultiIndex.from_product([["I", "II", "III"], [1,2]], names=["ROM",
"col_digit"])

#准备数据
data = np.arange(54).reshape(9,6)

df = pd.DataFrame(data, index=midx_row, columns=midx_col)
print(df)
```

```
ROM                I       II      III
col_digit          1   2   1   2   1   2
Upper row_digit
A     1            0   1   2   3   4   5
      2            6   7   8   9  10  11
      3           12  13  14  15  16  17
B     1           18  19  20  21  22  23
      2           24  25  26  27  28  29
      3           30  31  32  33  34  35
C     1           36  37  38  39  40  41
      2           42  43  44  45  46  47
      3           48  49  50  51  52  53
```

i. level的使用

level参数可以对某一索引进行聚合操作

```
## 上面例子，对A，B，C进行求和

data_sum = df.sum(level="Upper")
print(data_sum)
```

```
ROM           I        II       III
col_digit     1    2    1    2    1    2
Upper
A            18   21   24   27   30   33
B            72   75   78   81   84   87
C           126  129  132  135  138  141
```

ii. level配合axis

level和axis可以配合一起使用，这样就可以死对任意维度进行操作，还可以连续操作，最终得到想要的结果。

```
# 配合axis使用

data_sum = data_sum.sum(axis=1, level="ROM")
print(data_sum)
```

```
ROM      I   II  III
Upper
A       39   51   63
B      147  159  171
C      255  267  279
```

6. 合并数据

对数据集合的合是基本操作之一，也是我们处理大量数据的核心操作，本章主要研究数据的合并操作。

```
# 准备数据

import numpy as np
import pandas as pd

def make_df(cols, ind):
    '''生成一个简单的DataFrame数据'''
    data = {c:[str(c) + str(i) for i in ind] for c in cols}

    return pd.DataFrame(data, ind)

# 测试函数
df = make_df("ABC", range(5))
print(df)
```

```
    A   B   C
0  A0  B0  C0
1  A1  B1  C1
2  A2  B2  C2
3  A3  B3  C3
4  A4  B4  C4
```

6.1 pd.concat

通过pandas的concat能实现简单的数据合并。

i. pd.concat实现简单的Series和DataFrame的合并

```
# 简单的合并

s1 = pd.Series(list("ABC"), index =[1,2,3])
s2 = pd.Series(list("DEF"), index =[4,5,6])

s = pd.concat([s1, s2])

print("合并后: \n", s)
```

```
合并后:
 1    A
2    B
3    C
4    D
5    E
6    F
dtype: object
```

```
# DF合并
df1 = make_df("ABC", [1,2,3])
df2 = make_df("DEF", [4,5,6])

print("df1 = \n", df1)
print("\n df2 = \n", df2)
```

```
df1 =
     A   B   C
1  A1  B1  C1
2  A2  B2  C2
3  A3  B3  C3

 df2 =
     D   E   F
4  D4  E4  F4
5  D5  E5  F5
6  D6  E6  F6
```

```
# 对两个df进行合并
df3 = pd.concat([df1, df2])

print("\n df3 = \n", df3)
```

```
 df3 =
      A    B    C    D    E    F
1   A1   B1   C1  NaN  NaN  NaN
2   A2   B2   C2  NaN  NaN  NaN
3   A3   B3   C3  NaN  NaN  NaN
4  NaN  NaN  NaN   D4   E4   F4
5  NaN  NaN  NaN   D5   E5   F5
6  NaN  NaN  NaN   D6   E6   F6
```

```
/Users/augs/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:2:
FutureWarning: Sorting because non-concatenation axis is not aligned. A future version
of pandas will change to not sort by default.

To accept the future behavior, pass 'sort=False'.

To retain the current behavior and silence the warning, pass 'sort=True'.
```

```
# 合并的时候同样可以指定axis
df4 = pd.concat([df1, df2], axis=1)

print("\n df4 = \n", df4)
```

```
 df4 =
      A    B    C    D    E    F
1   A1   B1   C1  NaN  NaN  NaN
2   A2   B2   C2  NaN  NaN  NaN
3   A3   B3   C3  NaN  NaN  NaN
4  NaN  NaN  NaN   D4   E4   F4
5  NaN  NaN  NaN   D5   E5   F5
6  NaN  NaN  NaN   D6   E6   F6
```

ii. 重复索引的处理

重复索引在合并过程中是默认保留的，DataFrame也允许重复索引值的出现，但会熬成混淆，我们对重复索引的处理主要包括:

- 捕捉异常, 需要设置参数 verify_integrity
- 或略，需要设置参数 ignore_index
- 增加多级索引， 需要设置参数 keys

```
# 准备数据
df1 = make_df("AB", [0,1])
df2 = make_df("AB", [3,4])

df2.index = df1.index

print("df1 = \n", df1)
print("\n df2 = \n", df2)
```

```
df1 =
     A   B
0  A0  B0
1  A1  B1

 df2 =
     A   B
0  A3  B3
1  A4  B4
```

```
## 合并df1， df2
## 合并后后出现重复索引，DataFrame允许出现重复索引
df3 = pd.concat([df1, df2])
print("\n df3 = \n", df3)
```

```
 df3 =
     A   B
0  A0  B0
1  A1  B1
0  A3  B3
1  A4  B4
```

```
## 如果我们不允许出现重复索引，则需要设置verify_integrity

try:
    pd.concat([df1, df2], verify_integrity=True)
except ValueError as e:
    print("ValueError: ", e)
```

```
ValueError:  Indexes have overlapping values: Int64Index([0, 1], dtype='int64')
```

```
## 还可以选择忽略，此时自动生成心的索引，需要设置 ignore_index
df5 = pd.concat([df1, df2], ignore_index=True)
print("df5 = \n", df5)
```

```
df5 =
     A   B
0  A0  B0
1  A1  B1
2  A3  B3
3  A4  B4
```

```
## 还可以通过设置key增加多级索引

df6 = pd.concat([df1, df2], keys=["A", "B"])
print("df6 = \n", df6)
```

```
df6 =
       A   B
A 0  A0  B0
  1  A1  B1
B 0  A3  B3
  1  A4  B4
```

ii. 类似join的合并

如果合并的数据没有相同的列名，即需要合并的数据往外没有相同的列名，此时如何处理合并后的结果需要我们特殊处理。

pd.concat给我们提供了一些选项来解决这个问题。

通过设置参数，我们可以实现：

- 交集合并，join='inner'
- 并集合并，join='outer'
- 自定义列名， join_axis

```
## 准备数据
df1 = make_df("ABC", [1,2])
df2 = make_df("BCD", [3,4])

print("df1 = \n", df1)
print("df2 = \n", df2)
```

```
df1 =
     A   B   C
1  A1  B1  C1
2  A2  B2  C2
df2 =
     B   C   D
3  B3  C3  D3
4  B4  C4  D4
```

```
# 默认采用的是并集合并，即join=outer

df3 = pd.concat([df1, df2])
print("df3 =\n", df3)
```

```
df3 =
      A   B   C    D
1   A1  B1  C1  NaN
2   A2  B2  C2  NaN
3  NaN  B3  C3   D3
4  NaN  B4  C4   D4
```

```
/Users/augs/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:3:
FutureWarning: Sorting because non-concatenation axis is not aligned. A future version
of pandas will change to not sort by default.

To accept the future behavior, pass 'sort=False'.

To retain the current behavior and silence the warning, pass 'sort=True'.

  This is separate from the ipykernel package so we can avoid doing imports until
```

```
# 可以采用交集合并
# 合并后只出现两个列索引的交集部分

df4 = pd.concat([df1, df2], join='inner')
print("df4 =\n", df4)
```

```
df4 =
     B   C
1   B1   C1
2   B2   C2
3   B3   C3
4   B4   C4
```

```
##自定义列名
df5 = pd.concat([df1, df2], join_axes=[df1.columns])
print("df5 = \n", df5)
```

```
df5 =
      A    B    C
1    A1   B1   C1
2    A2   B2   C2
3   NaN   B3   C3
4   NaN   B4   C4
```

iii. append

append方式使用简单，但是它并不是直接更新原有对象的值，而是合并后创建一个新对象，每次合并都要重新创建索引和数据缓存，相对效率比较低，如果大数据量
操作，推荐使用concat。

```
## append

df3 = df1.append(df2)
print("df3 = \n", df3)
```

```
df3 =
      A    B    C     D
1    A1   B1   C1   NaN
2    A2   B2   C2   NaN
3   NaN   B3   C3    D3
4   NaN   B4   C4    D4
```

```
/Users/augs/anaconda3/lib/python3.7/site-packages/pandas/core/frame.py:6211:
FutureWarning: Sorting because non-concatenation axis is not aligned. A future version
of pandas will change to not sort by default.

To accept the future behavior, pass 'sort=False'.

To retain the current behavior and silence the warning, pass 'sort=True'.

  sort=sort)
```

6.2 pd.merge合并

本节内容类似于关系代数中的连接和合并，有关系代数或者SQL基础的同学相对会简单些。

常见的关系连接包括一对一(1:1), 多对一(N:1)和多对多(N:N), 我们使用pandas.merge可以完成相应的操作。

```
## 准备数据
df1 = pd.DataFrame({"name":list("ABCD"), "group":["I", "II", "III", "II"]})
df2 = pd.DataFrame({"name":list("ABCD"), "score":[61,78,74, 98]})
df3 = pd.DataFrame({"group":["I", "II", "III"], "leader":["Alice", "Bob", "Cindy"]})

# 此数据结构意味着每个组需要掌握的专业技能，一组需要有多个技能用重复值表示
df4 = pd.DataFrame({"group":["I", "I", "II","II", "II", "III", "III"],
                    "skills":["Linux", "Python", "Java", "Math", "English", "C++",
"PHP"]})

print("df1 = \n", df1)
print("\n df2 = \n", df2)
print("\n df3 = \n", df3)
print("\n df4 = \n", df4)
```

```
df1 =
   name group
0    A     I
1    B    II
2    C   III
3    D    II

 df2 =
   name  score
0    A     61
1    B     78
2    C     74
3    D     98

 df3 =
   group leader
```

```
0     I  Alice
1    II    Bob
2   III  Cindy

 df4 =
   group   skills
0      I    Linux
1      I   Python
2     II     Java
3     II     Math
4     II  English
5    III      C++
6    III      PHP
```

i. 一对一的连接

我们连接df1和df2的时候，每个都有索引name，通过name可以完成连接，并会自动进行合并。此时如果列的顺序不一致，并不会影响结果，pandas会自动处理。

```
## 一对一连接
df5 = pd.merge(df1, df2)
print("df5 = \n", df5)
```

```
df5 =
   name group  score
0    A     I     61
1    B    II     78
2    C   III     74
3    D    II     98
```

ii. 多对一的连接

此时获得的结果会自动保留重复值。

```
df6 = pd.merge(df1, df3)
print("df6 = \n", df6)
```

```
df6 =
   name group leader
0    A     I  Alice
1    B    II    Bob
2    D    II    Bob
3    C   III  Cindy
```

iii. 多对多的连接

此时因为后重复值，会自动按照最多的可能性对进行扩充，重复值都会得到保留。

```
df7 = pd.merge(df1, df4)
print("df7 = \n", df7)
```

```
df7 =
   name group   skills
0    A     I    Linux
1    A     I   Python
2    B    II     Java
3    B    II     Math
4    B    II  English
5    D    II     Java
6    D    II     Math
7    D    II  English
8    C   III      C++
9    C   III      PHP
```

iv. on参数的使用

两个数据集合并，以上案例都是使用默认的共同有的列作为合并的依据，我们还可以指定在那一列上进行合并，这就是参数on的作用。

```
# 使用on参数指定合并的列

df8 = pd.merge(df1, df2, on="name")
print("df8 = \n", df8)
```

```
df8 =
   name group  score
0    A     I     61
1    B    II     78
2    C   III     74
3    D    II     98
```

v. left_on 和 right_on 参数的使用

如果需要合并的数据集的列不是一个名字，就需要使用left_on和right_on来指定需要合并的两个列的名字。

例如以下两个数据集，一个列是name，一个列是my_names，我们需要这两个列进行合并，则需要进行如下面案例的操作。 在这种
情况下得到的数据会有一列冗余数据，此时可以使用drop方法将不需要的列扔掉。

```
## 准备数据
df1 = pd.DataFrame({"name":list("ABCD"), "group":["I", "II", "III", "II"]})
df2 = pd.DataFrame({"my_names":list("ABCD"), "score":[61,78,74, 98]})

df3 = pd.merge(df1, df2, left_on="name", right_on="my_names")
print("df3 = \n", df3)

print("\n df3.drop = \n", df3.drop("my_names", axis=1))
```

```
df3 =
   name group my_names  score
0    A     I         A     61
1    B    II         B     78
2    C   III         C     74
3    D    II         D     98

 df3.drop =
   name group  score
0    A     I     61
1    B    II     78
2    C   III     74
3    D    II     98
```

vi. left_index 和 right_index

两个数据的合并还可以用键来实现合并，此时通过设置left/right_index来确定是否使用键作为合并的回一句。

```
# 使用准备好的数据
print("df1 = \n", df1)
print("\n df2 = \n", df2)

df4 = pd.merge(df1, df2, left_index=True, right_index=True)
print("\n df4 = \n", df4)
```

```
df1 =
   name group
0    A     I
1    B    II
2    C   III
3    D    II

 df2 =
   my_names  score
0         A     61
1         B     78
2         C     74
3         D     98
```

```
 df4 =
   name group my_names  score
0    A     I        A     61
1    B    II        B     78
2    C   III        C     74
3    D    II        D     98
```

vii. how参数的使用

当两个数据进行合并的时候，如果把两个数据当做集合看待，则合并的方式按照关系数学，有：

- 内连接：how='inner', 此时结果只保留交集
- 外连接: how='outer', 此时结果保留的是两个数据集的并集
- 左连接: how='left', 此时结果保留左侧全部内容，有连接的右侧内容也会保留
- 右连接: how='right', 此时结果保留右侧全部内容，有链接的左侧内容也会保留

```
## 准备数据

d1 = {"name":list("ABCDE"), "score":[65,45,56,78,85]}
d2 = {"name":list("CDEFG"), "height":[176,156,187,191,173]}

df1 = pd.DataFrame(d1)
df2 = pd.DataFrame(d2)

print("df1 = \n", df1)
print("\n df2 = \n", df2)
```

```
df1 =
   name  score
0    A     65
1    B     45
2    C     56
3    D     78
4    E     85

 df2 =
   name  height
0    C     176
1    D     156
2    E     187
3    F     191
4    G     173
```

```python
# 外连接

df3 = pd.merge(df1, df2, how="outer")
print("merge.outer = \n", df3)
```

```
merge.outer =
   name  score  height
0    A   65.0     NaN
1    B   45.0     NaN
2    C   56.0   176.0
3    D   78.0   156.0
4    E   85.0   187.0
5    F    NaN   191.0
6    G    NaN   173.0
```

```python
# 内连接

df3 = pd.merge(df1, df2, how="inner")
print("merge.inner = \n", df3)
```

```
merge.inner =
   name  score  height
0    C     56     176
1    D     78     156
2    E     85     187
```

```python
# 左连接

df3 = pd.merge(df1, df2, how="left")
print("merge.left = \n", df3)
```

```
merge.left =
   name  score  height
0    A     65     NaN
1    B     45     NaN
2    C     56   176.0
3    D     78   156.0
4    E     85   187.0
```

```
# 右连接

df3 = pd.merge(df1, df2, how="right")
print("merge.right = \n", df3)
```

```
merge.right =
   name  score  height
0    C   56.0     176
1    D   78.0     156
2    E   85.0     187
3    F    NaN     191
4    G    NaN     173
```

viii. suffixes参数

当两个数据源有重复的列名的时候，重复列名字又不作为连接操作的数据，merge函数会自动为重复列明添加*x*和*y*作为区分，但是，我们
可以通过suffixes参数设置我们需要的后缀用以区分重复。

```
# 准备数据

d1 = {"name":list("ABCDE"), "score":[65,45,56,78,85]}
d2 = {"name":list("CDEFG"), "score":[176,156,187,191,173]}


df1 = pd.DataFrame(d1)
df2 = pd.DataFrame(d2)

print("df1 = \n", df1)
print("\n df2 = \n", df2)
```

```
df1 =
   name  score
0    A     65
1    B     45
2    C     56
3    D     78
4    E     85

 df2 =
   name  score
0    C    176
1    D    156
2    E    187
3    F    191
4    G    173
```

```
# 默认重复会自动加上后缀加以区分

df3 = pd.merge(df1, df2, on="name")
print("默认后缀：df3 = \n", df3)
```

```
默认后缀：df3 =
   name  score_x  score_y
0    C       56      176
1    D       78      156
2    E       85      187
```

```
# 通过suffixes指定后缀

df3 = pd.merge(df1, df2, on="name", suffixes=['01', '02'])
print("指定后缀：df3 = \n", df3)
```

```
指定后缀：df3 =
   name  score01  score02
0    C       56      176
1    D       78      156
2    E       85      187
```

7. 累计和分组

对大数据进行分析的时候，一项基本工作就是数据累计（summarization），通常包括：

- sum： 求和
- mean：平均数
- median：中位数
- min：最小值
- max：最大值
- count：计数
- first/last: 第一项和最后一项
- std: 标准差
- var：方差
- mad：均值绝对方差
- prod：所有项乘积

```
# 准备数据
# 我们这次准备数据时候用seaborn提供的行星数据，包括天文学家观测到的围绕恒星运行的行星数据
# 下载地址：
#  https://github.com/mwaskom/seaborn-data

import seaborn as sns

planets = sns.load_dataset('planets')
print(planets.shape)

# 显示数据头部
print(planets.head())
```

```
(1035, 6)
            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
```

7.1 describe函数

describe函数计算每一列的常用统计值，给出一个比较笼统的时数值。

通过下面describe计算的数据，我们可以对数据总体做出一个大概的判断，比如各个列的中级，最大值，均值等等。

在统计之前，我们删除掉缺失值。

```
d = planets.dropna().describe()
print(d)
```

```
          number  orbital_period        mass    distance         year
count  498.00000      498.000000  498.000000  498.000000   498.000000
mean     1.73494      835.778671    2.509320   52.068213  2007.377510
std      1.17572     1469.128259    3.636274   46.596041     4.167284
min      1.00000        1.328300    0.003600    1.350000  1989.000000
25%      1.00000       38.272250    0.212500   24.497500  2005.000000
50%      1.00000      357.000000    1.245000   39.940000  2009.000000
75%      2.00000      999.600000    2.867500   59.332500  2011.000000
max      6.00000    17337.500000   25.000000  354.000000  2014.000000
```

7.2 GroupBy

GroupBy借用的是SQL的命令， 其核心思想是：

- split:分割
- applay：应用
- combine：组合

通过运用GroupBy命令和不同的累计函数进行组合使用，对某些标签或索引进行累计分析。

```
# 小案例
import pandas as pd

df = pd.DataFrame({"name":list("ABCABC"), "data":range(100,106)}, columns=["name",
"data"])

print("df = \n", df)

# 按name列进行分组，然后求魅族的平均数
a = df.groupby("name").mean()
print("\n a = \n", a)
```

```
df =
   name  data
0    A   100
1    B   101
2    C   102
3    A   103
4    B   104
5    C   105

 a =
        data
name
A      101.5
B      102.5
C      103.5
```

7.3 GroupBy对象

GroupBy返回的结果是一个抽象类型，可以看组是一个DataFrame的集合。

i. 按列取值返回的结果

```
# 按method列进行组合

gb = planets.groupby("method")

# gb是一个抽象数据
print(gb)

# 对GroupBy结果还可以再次抽取数
print(gb["orbital_period"])
```

```
<pandas.core.groupby.groupby.DataFrameGroupBy object at 0x7f755623f5f8>
<pandas.core.groupby.groupby.SeriesGroupBy object at 0x7f755623fa90>
```

```
# 下面句子相当于按找method进行组合，然后选取orbital_period列，选取后求每个组的中位数

a = planets.groupby("method")['orbital_period'].median()
print("组合后统计结果：  \n", a)
```

```
组合后统计结果：
 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
```

ii. 按组迭代

GroupBy对象支持进行迭代，返回每一组都是Series或者DataFrame数据。

```
# 迭代回来的数据大概是魅族以method作为名称，几行六列的一个DataFrame

for (method, group) in planets.groupby("method"):
    print(method, group.shape)
```

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

iii. 调用方法

借助于Python强大的类方法(@classmethod)， 可以直接对GroupBy的每一组对象添加功能，无论是DataFrame或者Series都可以使用。

```
a = planets.groupby("method")['year'].describe().unstack()
print(a)
```

```
       method
count  Astrometry                         2.000000
       Eclipse Timing Variations          9.000000
       Imaging                           38.000000
       Microlensing                      23.000000
       Orbital Brightness Modulation      3.000000
       Pulsar Timing                      5.000000
       Pulsation Timing Variations        1.000000
       Radial Velocity                  553.000000
       Transit                          397.000000
       Transit Timing Variations          4.000000
mean   Astrometry                      2011.500000
       Eclipse Timing Variations       2010.000000
       Imaging                         2009.131579
       Microlensing                    2009.782609
       Orbital Brightness Modulation   2011.666667
       Pulsar Timing                   1998.400000
       Pulsation Timing Variations     2007.000000
       Radial Velocity                 2007.518987
       Transit                         2011.236776
       Transit Timing Variations       2012.500000
std    Astrometry                         2.121320
       Eclipse Timing Variations          1.414214
       Imaging                            2.781901
       Microlensing                       2.859697
       Orbital Brightness Modulation      1.154701
       Pulsar Timing                      8.384510
       Pulsation Timing Variations             NaN
```

```
       Radial Velocity                     4.249052
       Transit                             2.077867
       Transit Timing Variations           1.290994
                                           ...
50%    Astrometry                       2011.500000
       Eclipse Timing Variations        2010.000000
       Imaging                          2009.000000
       Microlensing                     2010.000000
       Orbital Brightness Modulation    2011.000000
       Pulsar Timing                    1994.000000
       Pulsation Timing Variations      2007.000000
       Radial Velocity                  2009.000000
       Transit                          2012.000000
       Transit Timing Variations        2012.500000
75%    Astrometry                       2012.250000
       Eclipse Timing Variations        2011.000000
       Imaging                          2011.000000
       Microlensing                     2012.000000
       Orbital Brightness Modulation    2012.000000
       Pulsar Timing                    2003.000000
       Pulsation Timing Variations      2007.000000
       Radial Velocity                  2011.000000
       Transit                          2013.000000
       Transit Timing Variations        2013.250000
max    Astrometry                       2013.000000
       Eclipse Timing Variations        2012.000000
       Imaging                          2013.000000
       Microlensing                     2013.000000
       Orbital Brightness Modulation    2013.000000
       Pulsar Timing                    2011.000000
       Pulsation Timing Variations      2007.000000
       Radial Velocity                  2014.000000
       Transit                          2014.000000
       Transit Timing Variations        2014.000000
Length: 80, dtype: float64
```

7.4 累计，过滤和转换

GroupBy基本功能是分组，但分组之后相应的操作，也为数据分析提供了很多高效的方法。
此类方法大概分为：

- aggregate：累计
- filter：过滤
- transform：变换
- apply：应用

```
## 数据准备
import numpy as np
import pandas as pd

rng = np.random.RandomState(0)
df = pd.DataFrame({'key':list("ABCABC"),
                   'data_1':range(100,106),
                   'data_2': rng.randint(0,10, 6)},
                   columns=['key', 'data_1', 'data_2'])
print("df = \n", df)
```

```
df =
   key  data_1  data_2
0   A     100       5
1   B     101       0
2   C     102       3
3   A     103       3
4   B     104       7
5   C     105       9
```

i. 累计

相比较于sum和median之类的功能，累计（aggregate）能实现比较复杂的操作，比如字符串，函数或者函数列表，并且能一次性计算所有累计值。

```
# 同事统计min, median, max
rst = df.groupby('key').aggregate(['min', np.median, np.max])
print(rst)
```

```
    data_1             data_2
       min median amax    min median amax
key
A      100  101.5  103      3    4.0    5
B      101  102.5  104      0    3.5    7
C      102  103.5  105      3    6.0    9
```

```
# 或者还有其他的方法，对每列时候用不同的统计函数

rst = df.groupby('key').aggregate({'data_1':'min',
                                   'data_2':'max'})

print(rst)
```

```
     data_1  data_2
key
A       100       5
B       101       7
C       102       9
```

ii. 过滤

通过过了功能，可以保留我们需要的值，把不需要的去掉。

```
# 过滤函数

def my_filter(x):
    return x['data_2'].std() > 1.5

rst = df.groupby('key').std()
print('df.std = \n', rst)

# 使用过滤函数，把不符合要求的过滤掉
rst = df.groupby('key').filter(my_filter)
print("\n rst.filter_func = \n", rst)
```

```
df.std =
       data_1    data_2
key
A    2.12132  1.414214
B    2.12132  4.949747
C    2.12132  4.242641

 rst.filter_func =
   key  data_1  data_2
1   B     101       0
2   C     102       3
4   B     104       7
5   C     105       9
```

iii. 转换

累计操作把数据集合进行了裁剪和选择，而转换操作是把全量数据进行加工，得到的数据格式与输入数据一致，常见的操作是对数据减去均值，实现数据标准化。

```
# 对数据进行标准化

rst = df.groupby('key').transform(lambda x: x - x.mean())

print(rst)
```

```
   data_1  data_2
0    -1.5     1.0
1    -1.5    -3.5
2    -1.5    -3.0
3     1.5    -1.0
4     1.5     3.5
5     1.5     3.0
```

iv. 应用

apply可以让人在每个数据上应用任意方法，这个函数让输入一个DataFrane，返回的结果可以是pandas对象或者标量。

```
# 求每一项的百分比
def  norm_data_1(x):
    # 求百分比
    x['data_1'] = x['data_1'] / x['data_1'].sum()

    return x

rst = df.groupby('key').apply(norm_data_1)
print(rst)
```

```
  key    data_1  data_2
0   A  0.492611       5
1   B  0.492683       0
2   C  0.492754       3
3   A  0.507389       3
4   B  0.507317       7
5   C  0.507246       9
```

7.5 设置分割的键

对DataFrame的分割可以根据列来，还可以有其他的方法，本节主要介绍对DataFrame的各种分割方法。

i. 将列表，数组，Series或者索引作为分组键

此时分组键可以是与DaytaFrame匹配的任意Series或者列表。

```
# 普通数组作为分组键

L = [0,1,1,0,2,1]
a = df.groupby(L).sum()
print(a)
```

```
   data_1  data_2
0     203       8
1     308      12
2     104       7
```

```
# 直接用键值也可以，比较啰嗦
a = df.groupby(df['key']).sum()
print(a)
```

```
     data_1  data_2
key
A       203       8
B       205       7
C       207      12
```

ii. 用字典或者Series将索引映射到分组的名称

提供字典，按照字典的键值进行分组，最后结果使用字典键值映射的值。

要求索引必须跟字典的键值匹配。

```
# 利用字典分组
D_mapping = {'A':"One", 'B':"Two", 'C':"Three"}

df = df.set_index('key')
a = df.groupby(D_mapping).sum()
print(a)
```

```
       data_1  data_2
One       203       8
Three     207      12
Two       205       7
```

iii. 使用任意Python函数

我们还可以吧Python函数传入groupby，与前面的内容类似，然后得到新的分组。

```
# 传入任意分组

a = df.groupby(str.lower).mean()
print(a)
```

```
   data_1  data_2
a   101.5     4.0
b   102.5     3.5
c   103.5     6.0
```

iii. 多个有效的键组成的列表

有效的键值可以组合起来，从而返回一个多级索引的分组结果。

```
# 利用上面定义的字典，我们可以组合成多级索引

a = df.groupby([str.lower, D_mapping]).mean()
print(a)
```

```
         data_1  data_2
a One     101.5     4.0
b Two     102.5     3.5
c Three   103.5     6.0
```

8. 数据透视表

我们目前所用的累计操作都是按照一个维度进行，数据透视表可以看做是按照二维进行累计的操作功能。

```
# 以泰坦尼克号数据为例子进行展示

import numpy as np
import pandas as pd
import seaborn as sns

titanic = sns.load_dataset("titanic")
print(titanic.shape)
```

```
(891, 15)
```

i. 数据透视表的初步使用

```
# 先进行粗分类
a = titanic.groupby("sex")[['survived']].mean()

# 可以看出，女性获救比例大概是男性的4倍多
print(a)
```

```
        survived
sex
female  0.742038
male    0.188908
```

```
# 尝试按sex，class分组，然后统计逃生人数，求mean后使用层级索引
# 可以清晰的展示出，逃生人数受sex，class 的影响

a = titanic.groupby(['sex', 'class'])['survived'].aggregate('mean').unstack()
print(a)
```

```
class       First    Second     Third
sex
female  0.968085  0.921053  0.500000
male    0.368852  0.157407  0.135447
```

ii. pivot_table

pivot_table实现的效果等同于上一届的管道命令，是一个简写。

```
# 尝试按sex，class分组，然后统计逃生人数，求mean后使用层级索引
a = titanic.pivot_table('survived', index='sex', columns='class')
print(a)
```

```
class       First    Second     Third
sex
female  0.968085  0.921053  0.500000
male    0.368852  0.157407  0.135447
```

iii. 创建多级索引形状的DataFrame结果

```
# 按照sex，age，class统计，age分三个年龄段,0-18-80

# 使用cut函数对年龄进行分段
age = pd.cut(titanic['age'], [0,18,80])

a = titanic.pivot_table('survived', ['sex', age], 'class')
print(a)
```

```
class               First    Second     Third
sex    age
female (0, 18]   0.909091  1.000000  0.511628
       (18, 80]  0.972973  0.900000  0.423729
male   (0, 18]   0.800000  0.600000  0.215686
       (18, 80]  0.375000  0.071429  0.133663
```

```
# 对列也可以使用类似策略
# 使用qcut对票价进行划分成两部分，每一部分人数相等

fare = pd.qcut(titanic['fare'], 2)

a = titanic.pivot_table('survived', ['sex', age], [fare, 'class'])
print(a)
```

```
fare            (-0.001, 14.454]                    (14.454, 512.329]  \
class                      First    Second     Third             First
sex    age
female (0, 18]               NaN  1.000000  0.714286          0.909091
       (18, 80]              NaN  0.880000  0.444444          0.972973
male   (0, 18]               NaN  0.000000  0.260870          0.800000
       (18, 80]              0.0  0.098039  0.125000          0.391304

fare
class              Second     Third
sex    age
female (0, 18]   1.000000  0.318182
       (18, 80]  0.914286  0.391304
male   (0, 18]   0.818182  0.178571
       (18, 80]  0.030303  0.192308
```

9. 字符串的向量化操作

数据科学的处理离不开自付出的处理，相应的pandas也提供了字符串向量化操作的功能，我们一般利用这个来对采集来的信息进行清理。

我们可以使用python的方式来处理字符串，但是一旦字符串中包含缺省值，此时，使用pandas的字符串向量化功能就能避免出现崩溃的情况。

```
#
import pandas as pd

strs = ['One', "Two", "Three", None, "Four"]
strs = pd.Series(strs)
print(strs)
```

```
0      One
1      Two
2    Three
3     None
4     Four
dtype: object
```

i. 字符串向量化应用初步

```
#对字符串进行大写转换
s = strs.str.upper()
print(s)

# 对字符串进行小写转换
ss = strs.str.lower()
print("\n\n", ss)
```

```
0      ONE
1      TWO
2    THREE
3     None
4     FOUR
dtype: object
```

```
 0      one
1      two
2    three
3     None
4     four
dtype: object
```

ii. 其他pandas字符串方法

python中字符串的方法基本上可以直接应用在pandas中，需要注意的是返回值的不同，需要相应做出调整，例如判断类的返回的是一个bool值的数据结构，len之类的返回的是一个值。

相应字符串方法不在一一举例，需要的时候可以查看手册。

iii. 正则表达式

正则是字符串的大杀器，在pandas中同样也实现了正则的一些接口，如以下API：

- match: 调用re.match， 返回bool类型内容
- extract: 调用re.match, 返回匹配的字符串组groups
- findall： 调用re.findall
- replace: 正则的替换模式
- contains： re.search，返回bool内容
- count： 利用正则模式统计数量
- split：等价于 str.split， 支持正则
- rsplit： 等价于str.rsplit，支持正则

```
print("strs = \n", strs)

print()
a = strs.str.extract('([O, o, n, e, N, T ]+)')
print(a)
```

```
strs =
 0       One
1       Two
2     Three
3      None
4      Four
dtype: object

     0
0  One
1    T
2    T
3  NaN
4    o
```

iv. 其他字符串的使用方法

pandas还提供了一些其他的方法来实现字符串的操作。

- get: 获取雅安苏索引位置上的值，start=0
- slice： 对元素进行切片
- slice_replace: 对元素进行切片替换
- cat： 连接字符串
- repeat：重复元素
- normalize： 将字符串转换为Unicode规范形式
- pad： 在字符串的左边，右边或者两边增加空格
- wrap：将字符串按照指定宽度换行

- join： 用分隔符链接Series的每个元素
- get_dummies: 按照分隔符提取每个元素的dummy变量，转换为one-hot编码的DataFrame

```
# 对slice的使用和函数的直接切片一个效果
# df.str.slice(2,5) 等于 df.str[2:5]
print("strs = \n", strs)

print()

a = strs.str.slice(1,4)
print(a)

print()
# 等价于
b = strs.str[1:4]
print(b)
```

```
strs =
 0      One
1      Two
2    Three
3     None
4     Four
dtype: object

0      ne
1      wo
2     hre
3    None
4     our
dtype: object

0      ne
1      wo
2     hre
3    None
4     our
dtype: object
```

```
# df.str.get(i) 和 df.str[i]功能类似

# 以下案例用字母o切分字符串后选择后面的一组
a = strs.str.split('o').str.get(-1)
print(a)
```

```
0      One
1
2    Three
3     None
4       ur
dtype: object
```

iii. v. get_dummies

如果数据中包含已经倍编码的指标（coded indicator）， 可以使用get_dummies快速的把编码的信息分解。

```
# 假定： A=游泳， B=爬山， C=跑步， D=篮球

df = pd.DataFrame({'english':strs,
                   'hobbies':["B|C", "A|C|D", "B|D", "A|B|C", "A|B|C|D"]})
print(df)

print()

a = df['hobbies'].str.get_dummies('|')
print(a)
```

```
  english  hobbies
0     One      B|C
1     Two    A|C|D
2   Three      B|D
3    None    A|B|C
4    Four  A|B|C|D

   A  B  C  D
0  0  1  1  0
1  1  0  1  1
2  0  1  0  1
3  1  1  1  0
4  1  1  1  1
```

10. 处理时间序列

Pyhton处理时间序列常用的包有datetime，dateutil，但同样也存在性能弱的问题，pandas为了处理的大量时间相关数据，把时间相关数据作为datetime64类型进行处理，相对来讲，这种数据类型节省内存，处理起来速度快。

在pandas中，增加了Timestamp对象，所有日期与时间的处理方法都是通过Timestamp实现。

numpy/pandas利用Timestamp和datetime64的数据类型，将python的日期处理包datetime和dateutil有机结合起来，可以实现对日期数据的高效灵活处理。

i. datetime64 数据类型

datetime64是numpy处理时间相关内容的数据类型，可以对时间类型数据做灵活处理，同时还可以支持各种时间单位的操作。

常见的时间单位是：

- Y: Year
- M: Month
- W: Week
- D: Day
- h: Hour
- m: Minute
- s: Second
- ms: millisecond
- us: micorosecond
- ns: nanosecond
- ps: picosecond
- fs: femtosecond
- as: attosecond

```
import numpy as np

# datetime64数据类型
date = np.array("20180-03-12", dtype=np.datetime64)
print(date)
```

```
20180-03-12
```

```
# 向量化操作
d = date + np.arange(5)
print(d)
```

```
['20180-03-12' '20180-03-13' '20180-03-14' '20180-03-15' '20180-03-16']
```

```
# 添加时间单位，此处采用的是ns
a = np.datetime64("2019-01-13 12:45:32.30", "ns")
print(a)
```

```
2019-01-13T12:45:32.300000000
```

ii. Timestamp

```
import pandas as pd

# 利用pd.to_datetime可以将多种不同的格式时间进行处理
date = pd.to_datetime("5th of June, 2019")
print(date)
# date可以使用时间格式化功能
print(date.strftime("%A"))
```

```
2019-06-05 00:00:00
Wednesday
```

```
# 支持向量化操作

# 按天计算，  进行向量化
d = date + pd.to_timedelta(np.arange(10), "D")
print(d)
```

```
DatetimeIndex(['2019-06-05', '2019-06-06', '2019-06-07', '2019-06-08',
               '2019-06-09', '2019-06-10', '2019-06-11', '2019-06-12',
               '2019-06-13', '2019-06-14'],
              dtype='datetime64[ns]', freq=None)
```

iii. 时间做索引

```
idx = pd.DatetimeIndex(['2019-01-01', '2019-02-01','2019-03-01',
                        '2019-04-01','2019-05-01','2019-06-01'])

date = pd.Series(range(6), index=idx)
print(date)
```

```
2019-01-01    0
2019-02-01    1
2019-03-01    2
2019-04-01    3
2019-05-01    4
2019-06-01    5
dtype: int64
```

```
#既然是索引，就可以使用来进行数据的提取
# 切片包含结束位置
print(date[ '2019-02-01':'2019-06-01'])
```

```
2019-02-01    1
2019-03-01    2
2019-04-01    3
2019-05-01    4
2019-06-01    5
dtype: int64
```

```
# 一些特殊的时间操作
# 可以通过年费切片获取概念的全部数据
print(date["2019"])
```

```
2019-01-01    0
2019-02-01    1
2019-03-01    2
2019-04-01    3
2019-05-01    4
2019-06-01    5
dtype: int64
```

iv. pandas时间序列的数据结构

pandas对时间序列准备了几个特殊的数据结构：

- pd.DatetimeIndex: 针对时间戳数据
- pd.PeriodIndex: 针对时间周期数据
- pd.TimedeltaIndex: 针对时间增量或持续时间

```
# pd.to_datetime传输一个时间日期会返回一个Timestamp类型数据
# 传递时间序列会返回DatetimeIndex类型数据
from datetime import datetime

dates = pd.to_datetime([datetime(2019,4,3), '5th of June, 2018', '2017-Jul-9',
                        "09-02-2018", '20190105'])
# 对一个时间序列，会返回DatetimeIndex类型数据
print(dates)
```

```
DatetimeIndex(['2019-04-03', '2018-06-05', '2017-07-09', '2018-09-02',
               '2019-01-05'],
              dtype='datetime64[ns]', freq=None)
```

```
# DatetimeIndex 类型通过 pd.to_period和一个频率代码可以转换成PeriodIndex类型
d = dates.to_period('D')
print(d)
```

```
PeriodIndex(['2019-04-03', '2018-06-05', '2017-07-09', '2018-09-02',
             '2019-01-05'],
            dtype='period[D]', freq='D')
```

```
# 当用一个日期减去一个日期，返回的是TimedeltaIndex类型

d = dates - dates[0]
print(d)
```

```
TimedeltaIndex(['0 days', '-302 days', '-633 days', '-213 days', '-88 days'],
dtype='timedelta64[ns]', freq=None)
```

v. xxx_range类函数

pandas提供了可以规律产生时间序列的函数, 此类函数的使用和range类似，pandas提供了三个函数：

- pd.date_range： 可以处理时间戳,
- pd.period_range：可以处理周期
- pd.timedelta_range：可以处理时间间隔

```
d = pd.date_range("2018-03-03", "2018-12-02", periods=5)
print(d)
```

```
DatetimeIndex(['2018-03-03 00:00:00', '2018-05-10 12:00:00',
               '2018-07-18 00:00:00', '2018-09-24 12:00:00',
               '2018-12-02 00:00:00'],
              dtype='datetime64[ns]', freq=None)
```

```
d = pd.date_range("2018-03-03", "2018-12-02", freq="M")
print(d)
```

```
DatetimeIndex(['2018-03-31', '2018-04-30', '2018-05-31', '2018-06-30',
               '2018-07-31', '2018-08-31', '2018-09-30', '2018-10-31',
               '2018-11-30'],
              dtype='datetime64[ns]', freq='M')
```

```
d = pd.period_range("2018-01-01", periods=5, freq="M")
print(d)
```

```
PeriodIndex(['2018-01', '2018-02', '2018-03', '2018-04', '2018-05'], dtype='period[M]',
freq='M')
```

```
d = pd.timedelta_range(0, periods=5, freq="H")
print(d)
```

```
TimedeltaIndex(['00:00:00', '01:00:00', '02:00:00', '03:00:00', '04:00:00'],
dtype='timedelta64[ns]', freq='H')
```