引言

- 数据分析结果的好坏依赖于数据的好坏。很多数据集存在数据缺失，或数据格式不统一（畸形数据），或错误数据的情况。不管是不完善的报表，还是技术处理数据的失当都会不可避免的引起“脏”数据。
- 数据清洗是一项复杂且繁琐的工作，同时也是整个数据分析过程中最为重要的环节。但在实际的工作中一个分析项目70%左右的时间花在清洗数据上面。
- 数据清洗的目的有两个：第一是通过清洗让数据可用；第二是让数据变得更适合进行后续的分析工作。换句话说就是有”脏”数据要洗，干净的数据也要洗。

内容提要

- DataFrame 对象
- 下标存取
- 字符串处理、日期处理
- 缺失数据的处理
- 删除指定数据和重复数据
- 数据清洗基本流程及示例

8.1 DataFrame 对象

- DataFrame(数据框/数据表)对象是Pandas中最常用的数据对象。Pandas提供了将许多数据结构（字典、numpy二维数组等）转换为DataFrame对象的方法，还提供了许多输入输出函数（read_csv()等）来将各种文件格式转换成DataFrame 对象。

8.1 DataFrame 对象

- DataFrame对象是一个二维表格。其中，每列中的元素类型必须一致，而不同的列可以拥有不同的元素类型。
- 每行和每列都有索引，默认是位置索引，但通常会指定标签索引（相当于表格的列名和行名），还可以给列索引和行索引取名。
- 索引通常只有一级，但是也可以建立多级索引（第0级、第1级、…），多级索引相当于数据分类。行索引和列索引都可以是多级的。

	Measures	pH	Dens	Ca	Conduc	Date	Name
Depth	Contour						
0-10	Depression	5.3525	0.9775	10.6850	1.4725	2015-05-26	Lois
	Slope	5.5075	1.0500	12.2475	2.0500	2015-04-30	Roy
	Top	5.3325	1.0025	13.3850	1.3725	2015-05-21	Roy
10-30	Depression	4.8800	1.3575	7.5475	5.4800	2015-03-21	Lois
	Slope	5.2825	1.3475	9.5150	4.9100	2015-02-06	Diana
	Top	4.8600	1.3325	10.2375	3.5825	2015-04-11	Diana

8.1 DataFrame 对象

- 例如：

	A	B	C	D	E	F	G	H
1	Depth	Contour	pH	Dens	Ca	Conduc	Date	Name
2	0-10	Depression	5.3525	0.9775	10.685	1.4725	2015/5/26 0:00	Lois
3	0-10	Slope	5.5075	1.05	12.2475	2.05	2015/4/30 0:00	Roy
4	0-10	Top	5.3325	1.0025	13.385	1.3725	2015/5/21 0:00	Roy
5	10-30	Depression	4.88	1.3575	7.5475	5.48	2015/3/21 0:00	Lois
6	10-30	Slope	5.2825	1.3475	9.515	4.91	2015/2/6 0:00	Diana
7	10-30	Top	4.85	1.3325	10.2375	3.5825	2015/4/11 0:00	Diana

```
df_soil = pd.read_csv("data/Soils-simple.csv", index_col=[0, 1], parse_dates=["Date"])
df_soil.columns.name = "Measures" # 设置列索引名
print(df_soil)
```

通过index_col参数指定第0和第1列为行索引，用 parse_dates参数指定进行日期转换的列，在指定列时可以使用列的序号(是文件中的列序号)或列名，例如这里也可以使用parse_dates=[6]。

```
Measures                pH     Dens       Ca  Conduc       Date   Name
Depth Contour
0-10  Depression   5.3525   0.9775  10.6850  1.4725 2015-05-26   Lois
      Slope        5.5075   1.0500  12.2475  2.0500 2015-04-30    Roy
      Top          5.3325   1.0025  13.3850  1.3725 2015-05-21    Roy
10-30 Depression   4.8800   1.3575   7.5475  5.4800 2015-03-21   Lois
      Slope        5.2825   1.3475   9.5150  4.9100 2015-02-06  Diana
      Top          4.8500   1.3325  10.2375  3.5825 2015-04-11  Diana
```

8.1 DataFrame 对象

- 在本例中，有4列浮点数类型、1列日期类型和1列object类型。object类型的列可以保存任何Python对象，在 Pandas中字符串列使用object类型。
- DataFrame对象的dtypes属性可以获得表示各个列类型的Series对象：

```
print(df_soil.dtypes)
```

Measures		pH	Dens	Ca	Conduc	Date	Name
Depth	Contour						
0-10	Depression	5.3525	0.9775	10.6850	1.4725	2015-05-26	Lois
	Slope	5.5075	1.0500	12.2475	2.0500	2015-04-30	Roy
	Top	5.3325	1.0025	13.3850	1.3725	2015-05-21	Roy
10-30	Depression	4.8800	1.3575	7.5475	5.4800	2015-03-21	Lois
	Slope	5.2825	1.3475	9.5150	4.9100	2015-02-06	Diana
	Top	4.8500	1.3325	10.2375	3.5825	2015-04-11	Diana

```
Measures
pH                  float64
Dens                float64
Ca                  float64
Conduc              float64
Date          datetime64[ns]
Name                  object
dtype: object
```

- 与数组类似，通过shape属性可以得到DataFrame的行数和列数：

```
print(df_soil.shape)
```

```
(6, 6)
```

8.1 DataFrame 对象

- values属性获得DataFrame对象中的数据的Numpy数组表示，Pandas建议使用to_numpy()方法转换数据。由于本例中的列类型不统一，所以数组元素类型为object：

```
print(df_soil.values)
print(df_soil.values.dtype)
```

```
[[5.3525000000001 0.9775 10.685 1.4725 Timestamp('2015-05-26 00:00:00')  'Lois']
 [5.5075 1.05 12.2475 2.05 Timestamp('2015-04-30 00:00:00') 'Roy']
 [5.3325000000005 1.0025 13.385000000002 1.3725  Timestamp('2015-05-21 00:00:00') 'Roy']
 [4.88 1.3575 7.5475 5.48 Timestamp('2015-03-21 00:00:00') 'Lois']
 [5.2825000000015 1.3475 9.515 4.91 Timestamp('2015-02-06 00:00:00')  'Diana']
 [4.85 1.3325 10.2375 3.5825000000005 Timestamp('2015-04-11 00:00:00')  'Diana']]
object
```

Measures		pH	Dens	Ca	Conduc	Date	Name
Depth	Contour						
0-10	Depression	5.3525	0.9775	10.6850	1.4725	2015-05-26	Lois
	Slope	5.5075	1.0500	12.2475	2.0500	2015-04-30	Roy
	Top	5.3325	1.0025	13.3850	1.3725	2015-05-21	Roy
10-30	Depression	4.8800	1.3575	7.5475	5.4800	2015-03-21	Lois
	Slope	5.2825	1.3475	9.5150	4.9100	2015-02-06	Diana
	Top	4.8500	1.3325	10.2375	3.5825	2015-04-11	Diana

8.1 DataFrame 对象

- DataFrame对象拥有行索引和列索引，可以通过索引标签对其中的数据进行存取。index属性保存行索引，而columns属性保存列索引。

```
print(df_soil.columns)
print(df_soil.columns.name)
```

```
Index(['pH', 'Dens', 'Ca', 'Conduc', 'Date', 'Name'], dtype='object', name='Measures')
Measures
```

Measures		pH	Dens	Ca	Conduc	Date	Name
Depth	Contour						
0-10	Depression	5.3525	0.9775	10.6850	1.4725	2015-05-26	Lois
	Slope	5.5075	1.0500	12.2475	2.0500	2015-04-30	Roy
	Top	5.3325	1.0025	13.3850	1.3725	2015-05-21	Roy
10-30	Depression	4.8800	1.3575	7.5475	5.4800	2015-03-21	Lois
	Slope	5.2825	1.3475	9.5150	4.9100	2015-02-06	Diana
	Top	4.8500	1.3325	10.2375	3.5825	2015-04-11	Diana

8.1 DataFrame 对象

- 在本例中，行索引是一个表示多级索引的MultiIndex对象，每级的索引名可以通过names属性存取：

```
print(df_soil.index)
print(df_soil.index.names)
```

```
MultiIndex([( '0-10',    'Depression'),
            ( '0-10',         'Slope'),
            ( '0-10',           'Top'),
            ('10-30',  'Depression'),
            ('10-30',         'Slope'),
            ('10-30',           'Top')],
           names=['Depth', 'Contour'])
['Depth', 'Contour']
```

Measures		pH	Dens	Ca	Conduc	Date	Name
Depth	Contour						
0-10	Depression	5.3525	0.9775	10.6850	1.4725	2015-05-26	Lois
	Slope	5.5075	1.0500	12.2475	2.0500	2015-04-30	Roy
	Top	5.3325	1.0025	13.3850	1.3725	2015-05-21	Roy
10-30	Depression	4.8800	1.3575	7.5475	5.4800	2015-03-21	Lois
	Slope	5.2825	1.3475	9.5150	4.9100	2015-02-06	Diana
	Top	4.8500	1.3325	10.2375	3.5825	2015-04-11	Diana

8.1 DataFrame 对象

- []运算符可以通过列索引标签获取指定的列，当下标是单个标签时，所得到的是Series对象，例如df_soil['pH']; 而当下标是标签列表时，则得到一个新的DataFrame对象，例如df_soil['pH'，'Ca']:

print(df_soil['pH'])

```
Depth   Contour
0-10    Depression    5.3525
        Slope         5.5075
        Top           5.3325
10-30   Depression    4.8800
        Slope         5.2825
        Top           4.8500
Name: pH, dtype: float64
```

print(df_soil[['pH','Ca']])

Measures		pH	Ca
Depth	Contour		
0-10	Depression	5.3525	10.6850
	Slope	5.5075	12.2475
	Top	5.3325	13.3850
10-30	Depression	4.8800	7.5475
	Slope	5.2825	9.5150
	Top	4.8500	10.2375

8.1 DataFrame 对象

- .loc[]可通过行索引标签元组获取指定的行，例如df.loc[("0-10", "Top")]获得Depth为“0-10”,Contour为“Top”的行，而df.loc[("0-10",)]获取Depth为“0-10”的所有行。
- 当结果为一行时，得到的是Series对象；而多行时是DataFrame对象。

```
print(df_soil.loc[('0-10', 'Top')])
```

```
Measures
pH                          5.3325
Dens                        1.0025
Ca                          13.385
Conduc                      1.3725
Date        2015-05-21 00:00:00
Name                         Roy
Name: (0-10, Top), dtype: object
```

```
print(df_soil.loc[('0-10',)])
```

Measures	pH	Dens	Ca	Conduc	Date	Name
Contour						
Depression	5.3525	0.9775	10.6850	1.4725	2015-05-26	Lois
Slope	5.5075	1.0500	12.2475	2.0500	2015-04-30	Roy
Top	5.3325	1.0025	13.3850	1.3725	2015-05-21	Roy

注意这里有两级行索引

Measures		pH	Dens	Ca	Conduc	Date	Name
Depth	Contour						
0-10	Depression	5.3525	0.9775	10.6850	1.4725	2015-05-26	Lois
	Slope	5.5075	1.0500	12.2475	2.0500	2015-04-30	Roy
	Top	5.3325	1.0025	13.3850	1.3725	2015-05-21	Roy
10-30	Depression	4.8800	1.3575	7.5475	5.4800	2015-03-21	Lois
	Slope	5.2825	1.3475	9.5150	4.9100	2015-02-06	Diana
	Top	4.8500	1.3325	10.2375	3.5825	2015-04-11	Diana

8.1 DataFrame 对象

- 调用DataFrame()可以将多种格式的数据转换成DataFrame对象，它的三个参数data、 index和 columns分别为数据、行索引和列索引。 data参数可以是：
  - 二维数组或者能转换为二维数组的嵌套列表。
  - 字典：字典中的每对“键-值”将成为DataFrame对象的一列。值可以是一维数组、列表或 Series对象。
- 当未指定索引标签index和 columns时，采用整数位置索引。

```
A = np.random.randint(0, 10, (4, 2))
df1 = pd.DataFrame(data=A, index=['r1', 'r2', 'r3', 'r4'],  columns=['c1', 'c2'])
df2 = pd.DataFrame(data=A, columns=['c1', 'c2'])
df3 = pd.DataFrame(data=A)
```

print(df1)

	c1	c2
r1	3	0
r2	2	6
r3	1	5
r4	0	8

print(df2)

	c1	c2
0	3	0
1	2	6
2	1	5
3	0	8

print(df3)

	0	1
0	3	0
1	2	6
2	1	5
3	0	8

8.1 DataFrame 对象

- 调用DataFrame()可以将多种格式的数据转换成DataFrame对象，它的三个参数data、 index和 columns分别为数据、行索引和列索引。 data参数可以是：
  - 二维数组或者能转换为二维数组的嵌套列表。
  - 字典：字典中的每对“键-值”将成为DataFrame对象的一列。值可以是一维数组、列表或 Series对象。
- 当未指定索引标签index和 columns时，采用整数位置索引。

```
stud_dict = {'name': ['张三', '李四'], 'sex': ['男', '女'], 'major': ['大数据', '应用数学']}
df4 = pd.DataFrame(stud_dict)
df5 = pd.DataFrame(stud_dict, index=['r1', 'r2'])
```

print(df4)

```
   name  sex  major
0  张三   男   大数据
1  李四   女   应用数学
```

print(df5)

```
    name  sex  major
r1  张三   男   大数据
r2  李四   女   应用数学
```

8.1 DataFrame 对象

- DataFrame对象提供了一系列to_*()方法将数据转换为其他格式，例如to_dict(), to_csv(), to_json(), to_excel(), to_record()， to_numpy(), to_string(), to_sql()。
- 例如：to_dict()方法将DataFrame对象转换为字典，它的orient参数决定字典元素的类型：

print(df4)	# 列表字典
	print(df4.to_dict(orient="list"))
		# 字典列表
		print(df4.to_dict(orient="records"))

name sex major 0 张三 男 大数据 1 李四 女 应用数学	{'name': ['张三', '李四'], 'sex': ['男', '女'], 'major': ['大数据', '应用数学']}	[{'name': '张三', 'sex': '男', 'major': '大数据'}, {'name': '李四', 'sex': '女', 'major': '应用数学'}]

仅有一个字典，列索引为键，列值列表为值

包含多个字典的列表，每行数据对应一个字典

8.2 下标存取

- DataFrame提供了丰富的下标存取方法，除了直接使用[]运算符之外，还可以使用.loc[]、 .iloc[]、query方法等存取器存取其中的元素。

存取方法	说明
[col_label]	以单个标签作为下标，获取单列，返回Series对象
[col_labels]	以标签列表作为下标，获取多列，返回DataFrame对象
[row_slice]	整数切片或标签切片，得到指定范围之内的行
[row_bool_array]	选择布尔数组中True对应的行
.get(col_label, default)	与字典的gel()方法的用法相同
.at[index_label, col_label]	选择行标签和列标签对应的值，返回单个元素
.iat[index, col]	选择行编号和列编号对应的值，返回单个元素
.loc[index_label, col_label]	通过单个标签值、标签列表、标签数姐、布尔数组、标签切片等选择指定行与列上的数据
.iloc[index, col]	通过单个整数值、整数列表、整数数组、布尔数组、整数切片选择指定行与列上的数据
.query()	通过表达式选择满足条件的行
.head() / .tail()	获取头部 / 尾部n行数据
.nlargest(n,columns)	按某些列值降序排列后，获取前n行
.nsmallest(n,columns)	按某些列值升序排列后，获取前n行

8.2 下标存取

通过**[ ]运算符**对DataFrame对象进行存取时，支持以下几种形式的下标对象：

存取列
- 单个**索引标签**： 获取标签对应的列，返回一个Series对象。
- 多个**索引标签**： 获取以列表、数组(注意不能是元组)表示的多个标签对应的列，返回一个DataFrame对象。

存取行
- 整数切片： 以整数下标获取切片对应的行。
- 标签切片： 当使用标签作为切片时**包含终值**。
- 布尔数组： 获取数组中True对应的行。

- 布尔DataFrame: 将 DataFrame对象中False对应的元素设置为**NaN**。

print(df)

	c1	c2	c3
r1	5	0	3
r2	3	7	9
r3	3	5	2
r4	4	7	6
r5	8	8	1

print(df['c2']) # print(df.c2)

```
r1    0
r2    7
r3    5
r4    7
r5    8
Name: c2, dtype: int32
```

print(df[['c1', 'c3']])

	c1	c3
r1	5	3
r2	3	9
r3	3	2
r4	4	6
r5	8	1

8.2 下标存取

通过[ ]**运算符**对DataFrame对象进行存取时，支持以下5 种下标对象：

存取列
- 单个**索引标签**： 获取标签对应的列，返回一个Series对象。
- 多个**索引标签**： 获取以列表、数组(注意不能是元组)表示的多个标签对应的列，返回一个DataFrame对象。

存取行
- 整数切片： 以整数下标获取切片对应的行。
- 标签切片： 当使用标签作为切片时**包含终值**。
- 布尔数组： 获取数组中True对应的行。

- 布尔DataFrame: 将 DataFrame对象中False对应的元素设置为NaN。

print(df)

	c1	c2	c3
r1	5	0	3
r2	3	7	9
r3	3	5	2
r4	4	7	6
r5	8	8	1

print(df[2:4])

	c1	c2	c3
r3	3	5	2
r4	4	7	6

print(df['r2':'r4'])

	c1	c2	c3
r2	3	7	9
r3	3	5	2
r4	4	7	6

print(df.c1>3)

```
r1     True
r2    False
r3    False
r4     True
r5     True
Name: c1,
dtype: bool
```

print(df[df.c1>3])

	c1	c2	c3
r1	5	0	3
r4	4	7	6
r5	8	8	1

df[[True, False, False, True,True]]

8.2 下标存取

通过**[ ]运算符**对DataFrame对象进行存取时，支持以下5 种下标对象：

存取列
- 单个**索引标签**： 获取标签对应的列，返回一个Series对象。
- 多个**索引标签**： 获取以列表、数组(注意不能是元组)表示的多个标签对应的列，返回一个DataFrame对象。

存取行
- 整数切片： 以整数下标获取切片对应的行。
- 标签切片： 当使用标签作为切片时**包含终值**。
- 布尔数组： 获取数组中True对应的行。

- 布尔DataFrame: 将 DataFrame对象中False对应的元素设置为**NaN**。

print(df)

	c1	c2	c3
r1	5	0	3
r2	3	7	9
r3	3	5	2
r4	4	7	6
r5	8	8	1

print(df>2)

	c1	c2	c3
r1	True	False	True
r2	True	True	True
r3	True	True	False
r4	True	True	True
r5	True	True	False

print(df[df>2])

	c1	c2	c3
r1	5	NaN	3.0
r2	3	7.0	9.0
r3	3	5.0	NaN
r4	4	7.0	6.0
r5	8	8.0	NaN

8.2 下标存取

- .loc[]的下标对象是一个元组，其中的两个元素分别与DataFrame的两个轴相对应。若下标不是元组，则该下标对应第0轴（即获取行）。每个轴的下标对象都支持单个标签、标签列表、标签切片以及布尔数组。

print(df)

	c1	c2	c3
r1	5	0	3
r2	3	7	9
r3	3	5	2
r4	4	7	6
r5	8	8	1

 # r2行

```
print(df.loc['r2'])
```

```
c1    3
c2    7
c3    9
Name: r2,
dtype: int32
```

 # r2和r4行

```
print(df.loc[['r2', 'r4']])
```

	c1	c2	c3
r2	3	7	9
r4	4	7	6

 # 切片：r2~r4行

```
print(df.loc['r2':'r4'])
```

	c1	c2	c3
r2	3	7	9
r3	3	5	2
r4	4	7	6

8.2 下标存取

- .loc[]的下标对象是一个元组，其中的两个元素分别与DataFrame的两个轴相对应。若下标不是元组，则该下标对应第0轴（即获取行）。每个轴的下标对象都支持单个标签、标签列表、标签切片以及布尔数组。

df

	c1	c2	c3
r1	5	0	3
r2	3	7	9
r3	3	5	2
r4	4	7	6
r5	8	8	1

#单个元素r2行c3列

```
df.loc['r2', 'c3']
```

```
9
```

 # r2行,c1、c3列

```
df.loc['r2', ['c1', 'c3']]
```

```
c1    3
c3    9
Name: r2, dtype: int32
```

 # r2行,c1~c3列

```
df.loc['r2', 'c1':'c3']
```

```
c1    3
c2    7
c3    9
Name: r2, dtype: int32
```

 # r2~r4行、c3列

```
df.loc['r2':'r4', 'c3']
```

```
r2    9
r3    2
r4    6
Name: c3, dtype: int32
```

 # r2~r4行、c2~c3列

```
df.loc['r2':'r4', 'c2':'c3']
```

	c2	c3
r2	7	9
r3	5	2
r4	7	6

8.2 下标存取

df

	c1	c2	c3
r1	5	0	3
r2	3	7	9
r3	3	5	2
r4	4	7	6
r5	8	8	1

#r2行

```
df.loc['r2', :]
```

```
c1    3
c2    7
c3    9
Name: r2, dtype: int32
```

 # r2~r4行

```
df.loc['r2':'r4', :]
```

	c1	c2	c3
r2	3	7	9
r3	3	5	2
r4	4	7	6

 # c1列

```
print(df.loc[:, 'c1'])
```

```
r1    5
r2    3
r3    3
r4    4
r5    8
Name: c1, dtype: int32
```

 # c1、c3列

```
df.loc[:, ['c1', 'c3']]
```

	c1	c3
r1	5	3
r2	3	9
r3	3	2
r4	4	6
r5	8	1

 # 所有行、所有列

```
df.loc[:, :]
```

	c1	c2	c3
r1	5	0	3
r2	3	7	9
r3	3	5	2
r4	4	7	6
r5	8	8	1

 # c2>5的行

```
df.loc[df.c2>5]
```

	c1	c2	c3
r2	3	7	9
r4	4	7	6
r5	8	8	1

 # c2>5的行，c2、c3列

```
df.loc[df.c2>5,'c2':'c3']
```

	c2	c3
r2	7	9
r4	7	6
r5	8	1

8.2 下标存取

- .iloc[]和.loc[]类似，不过它使用整数下标。

print(df)

	c1	c2	c3
r1	5	0	3
r2	3	7	9
r3	3	5	2
r4	4	7	6
r5	8	8	1

 # 第2行

df.iloc[1]

```
c1    3
c2    7
c3    9
Name: r2,
dtype: int32
```

 # 第2行和第4行

df.iloc[[1, 3]]

	c1	c2	c3
r2	3	7	9
r4	4	7	6

 # 切片：第2~4行

df.iloc[1 : 4]

	c1	c2	c3
r2	3	7	9
r3	3	5	2
r4	4	7	6

 # 第2~4行、第3列

df.iloc[1:4, 2])

```
r2    9
r3    2
r4    6
Name: c3,
dtype: int32
```

 # 第2~4行、第2~3列

df.iloc[1:4, 1:3])

	c2	c3
r2	7	9
r3	5	2
r4	7	6

 # 第2列>5的行

df.iloc[df.c2.values>5]

	c1	c2	c3
r2	3	7	9
r4	4	7	6
r5	8	8	1

 # 第2列>5的行，第2、3列

df.iloc[df.c2.values>5,1:3]

	c2	c3
r2	7	9
r4	7	6
r5	8	1

此时注意：不能直接使用 df.c2>5 作为下标，而是要使用 df.c2.values>5，或 list(df.c2>5)

8.2 下标存取

- .loc[ ] 可以指定多级索引中每级索引上的标签。

指定了第0级和第1级行索引标签

```
df_soil.loc[('10-30','Slope'), ['pH','Ca']]
```

```
Measures
pH    5.2825
Ca     9.515
Name: (10-30, Slope),
dtype: object
```

缺失第1级行索引标签，自动转换为元组（'10-30 ', slice(None)）

```
df_soil.loc['10-30', ['pH','Ca']]
```

Measures		pH	Ca
Contour			
Depression		4.8800	7.5475
Slope		5.2825	9.5150
Top		4.8500	10.2375

缺失第0级行索引标签，使用np.s_对象创建元组（ slice(None) , 'Slope' ）

```
df_soil.loc[np.s_[:,'Slope'], ['pH','Ca']]
```

Measures		pH	Ca
Depth	Contour		
0-10	Slope	5.5075	12.2475
10-30	Slope	5.2825	9.5150

Measures		pH	Dens	Ca	Conduc	Date	Name
Depth	Contour						
0-10	Depression	5.3525	0.9775	10.6850	1.4725	2015-05-26	Lois
	Slope	5.5075	1.0500	12.2475	2.0500	2015-04-30	Roy
	Top	5.3325	1.0025	13.3850	1.3725	2015-05-21	Roy
10-30	Depression	4.8800	1.3575	7.5475	5.4800	2015-03-21	Lois
	Slope	5.2825	1.3475	9.5150	4.9100	2015-02-06	Diana
	Top	4.8500	1.3325	10.2375	3.5825	2015-04-11	Diana

此时列标签不能省略

8.2 下标存取

- 缺失的索引标签也可以直接用slice(None)指定，表示本级索引的所有标签

直接指定第1级行索引标签为 slice(None)

```
df_soil.loc[('10-30',slice(None)),['pH','Ca']]
```

Measures		pH	Ca
Contour			
Depression		4.8800	7.5475
Slope		5.2825	9.5150
Top		4.8500	10.2375

直接指定第0级行索引标签为 slice(None)

```
df_soil.loc[(slice(None),'Slope'),['pH','Ca']]
```

Measures		pH	Ca
Depth	Contour		
0-10	Slope	5.5075	12.2475
10-30	Slope	5.2825	9.5150

Measures		pH	Dens	Ca	Conduc	Date	Name
Depth	Contour						
0-10	Depression	5.3525	0.9775	10.6850	1.4725	2015-05-26	Lois
	Slope	5.5075	1.0500	12.2475	2.0500	2015-04-30	Roy
	Top	5.3325	1.0025	13.3850	1.3725	2015-05-21	Roy
10-30	Depression	4.8800	1.3575	7.5475	5.4800	2015-03-21	Lois
	Slope	5.2825	1.3475	9.5150	4.9100	2015-02-06	Diana
	Top	4.8500	1.3325	10.2375	3.5825	2015-04-11	Diana

8.2 下标存取

- query()方法

当需要根据一定的条件对行进行过滤时，通常可以先创建一个布尔数组，使用该数组获取True对应的行，例如下面的程序获得pH值大于5、 Ca含量小于11%的行。

(df_soil.pH>5) & (df_soil.Ca<11)

```
Depth  Contour
0-10   Depression     True
       Slope         False
       Top           False
10-30  Depression    False
       Slope          True
       Top           False
dtype: bool
```

df_soil[(df_soil.pH>5) & (df_soil.Ca<11)]

Measures		pH	Dens	Ca	Conduc	Date	Name
Depth	Contour						
0-10	Depression	5.3525	0.9775	10.685	1.4725	2015-05-26	Lois
10-30	Slope	5.2825	1.3475	9.515	4.9100	2015-02-06	Diana

Measures		pH	Dens	Ca	Conduc	Date	Name
Depth	Contour						
0-10	Depression	5.3525	0.9775	10.6850	1.4725	2015-05-26	Lois
	Slope	5.5075	1.0500	12.2475	2.0500	2015-04-30	Roy
	Top	5.3325	1.0025	13.3850	1.3725	2015-05-21	Roy
10-30	Depression	4.8800	1.3575	7.5475	5.4800	2015-03-21	Lois
	Slope	5.2825	1.3475	9.5150	4.9100	2015-02-06	Diana
	Top	4.8500	1.3325	10.2375	3.5825	2015-04-11	Diana

8.2 下标存取

- query()方法

当需要根据一定的条件对行进行过滤时，通常可以先创建一个布尔数组，使用该数组获取True对应的行，例如下面的程序获得pH值大于5、 Ca含量小于11%的行。

由于Python中无法自定义not、and和or等关键字的行为，因此需要改用~、& 、 |等位运算符。然而这些运算符的优先级比比较运算符要高，因此需要用括号将比较运算括起来。

(df_soil.pH>5) & (df_soil.Ca<11)

```
Depth   Contour
0-10    Depression     True
        Slope         False
        Top           False
10-30   Depression    False
        Slope          True
        Top           False
dtype: bool
```

df_soil[(df_soil.pH>5) & (df_soil.Ca<11)]

Measures		pH	Dens	Ca	Conduc	Date	Nam
Depth	Contour						
0-10	Depression	5.3525	0.9775	10.685	1.4725	2015-05-26	Loi
10-30	Slope	5.2825	1.3475	9.515	4.9100	2015-02-06	Dian

Measures		pH	Dens	Ca	Conduc	Date	Name
Depth	Contour						
0-10	Depression	5.3525	0.9775	10.6850	1.4725	2015-05-26	Lois
	Slope	5.5075	1.0500	12.2475	2.0500	2015-04-30	Roy
	Top	5.3325	1.0025	13.3850	1.3725	2015-05-21	Roy
10-30	Depression	4.8800	1.3575	7.5475	5.4800	2015-03-21	Lois
	Slope	5.2825	1.3475	9.5150	4.9100	2015-02-06	Diana

8.2 下标存取

- query()方法

使用 query()方法可以简化上述程序。

```
df_soil.query("pH>5 and Ca<11")
```

Measures		pH	Dens	Ca	Conduc	Date	Name
Depth	Contour						
0-10	Depression	5.3525	0.9775	10.685	1.4725	2015-05-26	Lois
10-30	Slope	5.2825	1.3475	9.515	4.9100	2015-02-06	Diana

query()方法的参数是一个运算表达式字符串。其中可以使用not、and和or等关键字进行向量布尔运算，表达式中的变量名表示与其对应的列。如果希望在表达式中使用其他全局或局域变量的值，可以在变量名之前添加@。例如：

```
a=5
b=11
print(df_soil.query("pH>@a and Ca<@b"))
```

Measures		pH	Dens	Ca	Conduc	Date	Name
Depth	Contour						
0-10	Depression	5.3525	0.9775	10.6850	1.4725	2015-05-26	Lois
	Slope	5.5075	1.0500	12.2475	2.0500	2015-04-30	Roy
	Top	5.3325	1.0025	13.3850	1.3725	2015-05-21	Roy
10-30	Depression	4.8800	1.3575	7.5475	5.4800	2015-03-21	Lois
	Slope	5.2825	1.3475	9.5150	4.9100	2015-02-06	Diana
	Top	4.8500	1.3325	10.2375	3.5825	2015-04-11	Diana

8.2 下标存取

- get()方法

get(col_label, default)方法与字典的get方法类似，列索引作为参数，返回参数col_label指定的列。如果该列存在，则返回该列对应的Series对象，否则，返回default参数指定的默认值。

df

	c1	c2	c3
r1	5	0	3
r2	3	7	9
r3	3	5	2
r4	4	7	6
r5	8	8	1

df.get('c2') *# 获取c2列*

```
r1    0
r2    7
r3    5
r4    7
r5    8
Name: c2, dtype: int32
```

df.get('c4',default='无')

无

8.2 下标存取

- head()和tail()方法

head(n)方法和tail(n)方法分别用于获取数据框头部n行数据和末尾n行数据,参数n默认值为5。

df

	c1	c2	c3
r1	5	0	3
r2	3	7	9
r3	3	5	2
r4	4	7	6
r5	8	8	1

df.head(2)

	c1	c2	c3
r1	5	0	3
r2	3	7	9

df.tail(2)

	c1	c2	c3
r4	4	7	6
r5	8	8	1

8.2 下标存取

- nlargest(n, columns) / nsmallest(n, columns)方法

根据指定列降序/升序排列后，返回数据框的前n行。

nlargest(n, columns)方法等同于df.sort_values(columns, ascending=False).head(n)

nsmallest(n, columns)方法等同于df.sort_values(columns, ascending=True).head(n)

```
df = pd.DataFrame({'population': [59000000, 65000000, 434000, 434000, 434000,
                                  337000, 11300, 11300, 11300],
                   'GDP': [1937894, 2583560, 12011, 4520, 12128, 17036, 182, 38, 311],
                   'alpha-2': ['IT', 'FR', 'MT', 'MV', 'BN', 'IS', 'NR', 'TV', 'AI'] },
                  index=['Italy', 'France', 'Malta', 'Maldives', 'Brunei', 'Iceland',
                         'Nauru', 'Tuvalu', 'Anguilla'])
```

	population	GDP	alpha-2
Italy	59000000	1937894	IT
France	65000000	2583560	FR
Malta	434000	12011	MT
Maldives	434000	4520	MV
Brunei	434000	12128	BN
Iceland	337000	17036	IS
Nauru	11300	182	NR
Tuvalu	11300	38	TV
Anguilla	11300	311	AI

8.2 下标存取

- nlargest(n, columns) / nsmallest(n, columns)方法

根据指定列降序/升序排列后，返回数据框的前n行。

nlargest(n, columns)方法等同于df.sort_values(columns, ascending=False).head(n)

nsmallest(n, columns)方法等同于df.sort_values(columns, ascending=True).head(n)

	population	GDP	alpha-2
Italy	59000000	1937894	IT
France	65000000	2583560	FR
Malta	434000	12011	MT
Maldives	434000	4520	MV
Brunei	434000	12128	BN
Iceland	337000	17036	IS
Nauru	11300	182	NR
Tuvalu	11300	38	TV
Anguilla	11300	311	AI

```
# 按population降序排列后的前3行
print(df.nlargest(3, 'population'))
```

	population	GDP	alpha-2
France	65000000	2583560	FR
Italy	59000000	1937894	IT
Malta	434000	12011	MT

8.2 下标存取

- nlargest(n, columns) / nsmallest(n, columns)方法

根据指定列降序/升序排列后，返回数据框的前n行。

nlargest(n, columns)方法等同于df.sort_values(columns, ascending=False).head(n)

nsmallest(n, columns)方法等同于df.sort_values(columns, ascending=True).head(n)

```
# 如果population有相同的行，保留最后一个行
print(df.nlargest(3, 'population', keep='last'))
```

	population	GDP	alpha-2
France	65000000	2583560	FR
Italy	59000000	1937894	IT
Brunei	434000	12128	BN

```
# 如果population有相同的行，保留所有相同行
print(df.nlargest(3, 'population', keep='all'))
```

	population	GDP	alpha-2
France	65000000	2583560	FR
Italy	59000000	1937894	IT
Malta	434000	12011	MT
Maldives	434000	4520	MV
Brunei	434000	12128	BN

	population	GDP	alpha-2
Italy	59000000	1937894	IT
France	65000000	2583560	FR
Malta	434000	12011	MT
Maldives	434000	4520	MV
Brunei	434000	12128	BN
Iceland	337000	17036	IS
Nauru	11300	182	NR
Tuvalu	11300	38	TV
Anguilla	11300	311	AI

8.2 下标存取

- nlargest(n, columns) / nsmallest(n, columns)方法

根据指定列降序/升序排列后，返回数据框的前n行。

nlargest(n, columns)方法等同于df.sort_values(columns, ascending=False).head(n)

nsmallest(n, columns)方法等同于df.sort_values(columns, ascending=True).head(n)

	population	GDP	alpha-2
Italy	59000000	1937894	IT
France	65000000	2583560	FR
Malta	434000	12011	MT
Maldives	434000	4520	MV
Brunei	434000	12128	BN
Iceland	337000	17036	IS
Nauru	11300	182	NR
Tuvalu	11300	38	TV
Anguilla	11300	311	AI

```
# 按population降序、GDP降序排列后的前3行
print(df.nlargest(3, ['population', 'GDP']))
```

	population	GDP	alpha-2
France	65000000	2583560	FR
Italy	59000000	1937894	IT
Brunei	434000	12128	BN

8.3 改变DataFrame的形状

- 与DataFrame结构相关的操作主要包括

函数或运算符	功能	函数	功能
shape属性	获取形状，同Numpy二维数组	set_index()	设置索引，即列转换为行索引
rename()	索引标签重命名	reset_index()	将行索引转换为列
df[‘col_label’]	添加或删除列	stack()	将列索引转换为行索引
assign()	返回添加新列之后的数据	uastack()	将行索引转换为列索引
drop()	删除行或列 del df[‘col_label’], del df.col_label	reorder_levels()	设置索引级别的顺序
append()	添加行	reorder_levels()	设置索引级别的顺序
concat()	拼接多块数据	swaplevel()	交换索引中两个级别的顺序
		sort_values()	按值排序
		sort_index()	按索引排序
		pivot()	数据透视
		melt()	数据融合（透视的逆变换）

8.3.1 修改index、columns名

一般常用的有两个方法：

- 修改index或columns属性
  - DataFrame.**index** = [newName]
  - DataFrame.**columns** = [newName]
- 使用**rename()**方法（推荐）

DataFrame.**rename**(mapper = None ， index = None，columns = None，copy = True， inplace = False，level = None）

参数：

- mapper, index，columns：映射函数，或旧名与新名的映射关系的字典。
- axis：int或str，可以是轴名称(‘index’，‘columns’)或数字(0,1)。默认为’index’。
- copy：默认为True，是否复制基础数据。
- **inplace**：是否原地（就地）操作，默认为False，返回新的DataFrame。否则为True，则忽略copy参数。
- level：在多级索引中，指定要修改的索引的级别。

8.3.1 修改index、columns名

修改方式	修改后结构
df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]}) df.columns = ["AA", "BB"] print(df)	AA BB 0 1 4 1 2 5 2 3 6

# Rename columns using a mapping（字典映射） 	
df = df.rename(columns={"AA": "a", "BB": "c"})
#或者 df.rename(columns={"AA": "a", "BB": "c"} ,inplace=True)
print(df)
	a c
	0 1 4
	1 2 5
	2 3 6

# Rename index using a mapping（字典映射） 	
df = df.rename(index={0: "x", 1: "y", 2: "z"})
print(df)
	a c
	x 1 4
	y 2 5
	z 3 6

# Using mapper and axis-style parameters 	
df = df.rename(str.upper, axis='columns')
print(df)
	A C
	x 1 4
	y 2 5
	z 3 6

8.3.2 添加列

DataFrame可以看作一个Series对象的**字典**，因此通过DataFrame[colname] = values即可添加新列。

```
df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
df["C"] = [7, 8, 9]
print(df)
```

```
   A  B  C
0  1  4  7
1  2  5  8
2  3  6  9
```

有时新添加的列是从已经存在的列计算而来，这时可以使用**eval()方法**计算。

```
df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
df["C"] = df.eval('B*10')
print(df)
```

⇔ df["C"] = df.B*10
⇔ df["C"] = df["B"]*10

```
   A  B  C
0  1  4  40
1  2  5  50
2  3  6  60
```

assign()方法添加由关键字参数指定的列，它返回一个新的DataFrame对象，**原数据的内容保持不变。**

```
df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
print(df.assign(C=df.B+2))
```

```
   A  B  C
0  1  4  6
1  2  5  7
2  3  6  8
```

8.3.3 插入新列

DataFrame的**insert()方法**用于在指定列序号位置插入新列：

df.**insert**(loc, column, value)

loc : 插入列的位置序号， column : 列标签

value: 新列，可以是列表，也可以是序列

```
df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
df.insert(loc=0, column='C', value=[7,8,9])
print(df)
```

	C	A	B
0	7	1	4
1	8	2	5
2	9	3	6

```
df.insert(loc=2, column='D', value=pd.Series([1,0,5]))
print(df)
```

	C	A	D	B
0	7	1	1	4
1	8	2	0	5
2	9	3	5	6

8.3.4 调整列顺序

有两种方法:

（1）读取某列保存到变量s（即用s引用它），然后从df中移除该列，最后s插入df中指定位置。----适合单列调整

（2）指定具有新顺序的列标签列表new_order，然后从df中获取这些列并覆盖df。---适合调整多列

```
   C  A  D  B
0  7  1  1  4
1  8  2  0  5
2  9  3  5  6
```

```
s=df['C']
del df['C']
df.insert(1,'C',s)
print(df)
```

```
   A  C  D  B
0  1  7  1  4
1  2  8  0  5
2  3  9  5  6
```

```
new_order=['D','B','A','C']
df=df[new_order]
print(df)
```

```
   D  B  A  C
0  1  4  1  7
1  0  5  2  8
2  5  6  3  9
```

8.3.5 删除列

使用**drop()方法**可以删除指定的列，参数与rename()方法类似，默认返回一个新的数据框。

```
df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6],
                   "C": [7, 8, 9], "D": [0, 0, 0]})
df = df.drop(columns=["C", "D"])
# 等价于 df.drop(columns=["C", "D"],inplace=True)
print(df)
```

	A	B
0	1	4
1	2	5
2	3	6

也可以使用**del 命令**删除一列。

```
df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6],
                   "C": [7, 8, 9], "D": [0, 0, 0]})
del df["D"]    # 注意，不能del df.D
print(df)
```

	A	B	C
0	1	4	7
1	2	5	8
2	3	6	9

8.3.6 添加行

append()方法用于添加行，它**没有inplace参数**，只能返回一个全新对象。

添加的数据可以源自另一个数据框，也可以来自字典或Series，但此时ignore_index参数必须设置为True，表示忽略行索引，从而自动添加新行的索引。

```
# 从另一个数据框中添加(相当于合并两个结构相同的数据框)
df = pd.DataFrame([[1, 2], [3, 4]], columns=list('AB'))
df2 = pd.DataFrame([[5, 6], [7, 8]], columns=list('AB'))
print(df.append(df2))
```

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

```
# With ignore_index set to True
print(df.append(df2, ignore_index=True))
```

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

```
# 忽略行索引时，还可以从字典添加行
row_dict = {"A": 5, "B": 6}
print(df.append(row_dict, ignore_index=True))
```

```
   A  B
0  1  2
1  3  4
2  5  6
```

```
# 忽略行索引时，还可以从Series添加行
row_series = pd.Series([5, 6], index=['A', 'B'])
print(df.append(row_series, ignore_index=True))
```

```
   A  B
0  1  2
1  3  4
2  5  6
```

8.3.6 添加行

由于每次调用append()都会复制所有的数据，因此在循环中使用append()添加数据会极大地降低程序的运算速度。可以使用一个列表缓存所有的分块数据，然后调用**pd.concat()方法**将所有这些数据沿着指定轴拼贴到一起。

```
# 从另一个数据框中添加(相当于合并两个结构相同的数据框)
df = pd.DataFrame([[1, 2], [3, 4]], columns=list('AB'))
df2 = pd.DataFrame([[5, 6], [7, 8]], columns=list('AB'))

df_list = [df, df2]  # 数据缓冲列表

# 在0轴(即列上)拼接数据，并忽略行索引
df3 = pd.concat(df_list, axis=0, ignore_index=True)
print(df3)
```

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

8.3.7 删除行

drop()方法用于删除指定标签对应的行或列。删除行时指定**labels参数**为要删除的行索引，axis参数默认为0（即对行操作）。

```
df = pd.DataFrame([[1, 2], [3, 4],[5, 6], [7, 8]], columns=list('AB'))
print(df)
```

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

```
print(df.drop(labels=[2,3]))
print(df.drop([2,3]))
print(df.drop([2,3],axis=0))
```

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

删除满足条件的行：先使用query()方法查询满足条件的数据行（返回一个数据框），然后提取该数据框的行索引index对象，最后使用drop()方法删除由这些行索引指定的行。

```
# df中A列<4的行的行索引
ix=df.query('A<4').index
print(df.drop(labels=ix))
```

df.query('A<4')

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

ix

```
Int64Index([0, 1],
dtype='int64')
```

df.drop(labels=ix)

```
   A  B
2  5  6
3  7  8
```

8.3.8 排序

sort_index()方法用于按行索引或列索引排序数据。

- 参数axis = 0（默认值）时，数据行按行索引顺序升序或降序排列；
- axis = 1时，数据列按列索引顺序升序或降序排列。
- 默认升序（ascending = True），默认不修改自身（inplace = False）。

```
df = pd.DataFrame({'Name': ['Tom', 'emily', 'Fred'],
                   'Sex':  ['M', 'F', 'M'],
                   'Age':  [35, 16, 28]})
```

df

	Name	Sex	Age
0	Tom	M	35
1	emily	F	16
2	Fred	M	28

df.sort_index()

	Name	Sex	Age
0	Tom	M	35
1	emily	F	16
2	Fred	M	28

df.sort_index(ascending=False)

	Name	Sex	Age
2	Fred	M	28
1	emily	F	16
0	Tom	M	35

df.sort_index(axis=1)

	Age	Name	Sex
0	35	Tom	M
1	16	emily	F
2	28	Fred	M

8.3.8 排序

sort_values()方法用于按值排序。

参数axis = 0（默认值）时，数据行按一个或多个列升序或降序排列；axis = 1时，数据列按一个或多个行值升序或降序排列。默认升序（ascending = True），默认不修改自身（inplace = False）。

df

	Name	Sex	Age
0	Tom	M	35
1	emily	F	16
2	Fred	M	28

```
df.sort_values(by=['Name'])
# 按姓名升序排列数据行
```

	Name	Sex	Age
2	Fred	M	28
0	Tom	M	35
1	emily	F	16

```
df.sort_values(by=['Name'],
    ascending=False)
# 按姓名降序排列数据行
```

	Name	Sex	Age
1	emily	F	16
0	Tom	M	35
2	Fred	M	28

```
df.sort_values(by=['Sex','Age'])
# 按性别、年龄升序排列数据行
```

	Name	Sex	Age
1	emily	F	16
2	Fred	M	28
0	Tom	M	35

```
df.sort_values( by=['Sex', 'Age'],
    ascending=[True, False])
# 按性别升序、年龄降序排列数据行
```

	Name	Sex	Age
1	emily	F	16
0	Tom	M	35
2	Fred	M	28

8.3.9 列转化为行索引

下面首先从CSV文件读入数据，并使用groupby()计算分组的平均值。

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1		Group	Contour	Depth	Gp	Block	pH	N	Dens	P	Ca	Mg	K	Na	Conduc
2	1	1	Top	0-10	T0	1	5.4	0.188	0.92	215	16.35	7.65	0.72	1.14	1.09
3	2	1	Top	0-10	T0	2	5.65	0.165	1.04	208	12.25	5.15	0.71	0.94	1.35
4	3	1	Top	0-10	T0	3	5.14	0.26	0.95	300	13.02	5.68	0.68	0.6	1.41
5	4	1	Top	0-10	T0	4	5.14	0.169	1.1	248	11.92	7.88	1.09	1.01	1.64
6	5	2	Top	10-30	T1	1	5.14	0.164	1.12	174	14.17	8.12	0.7	2.17	1.85
7	6	2	Top	10-30	T1	2	5.1	0.094	1.22	129	8.55	6.92	0.81	2.67	3.18
8	7	2	Top	10-30	T1	3	4.7	0.1	1.52	117	8.74	8.16	0.39	3.32	4.16
9	8	2	Top	10-30	T1	4	4.46	0.112	1.47	170	9.49	9.16	0.7	3.76	5.14
10	9	3	Top	30-60	T3	1	4.37	0.112	1.07	121	8.85	10.35	0.74	5.74	5.73
11	10	3	Top	30-60	T3	2	4.39	0.058	1.54	115	4.73	6.91	0.77	5.85	6.45
12	11	3	Top	30-60	T3	3	4.17	0.078	1.26	112	6.29	7.95	0.26	5.3	8.37
13	12	3	Top	30-60	T3	4	3.89	0.07	1.42	117	6.61	9.76	0.41	8.3	9.21
14	13	4	Top	60-90	T6	1	3.88	0.077	1.25	127	6.41	10.96	0.56	9.67	10.64
15	14	4	Top	60-90	T6	2	4.07	0.046	1.54	91	3.82	6.61	0.5	7.67	10.07
16	15	4	Top	60-90	T6	3	3.88	0.055	1.53	91	4.98	8	0.23	8.78	11.26
17	16	4	Top	60-90	T6	4	3.74	0.053	1.4	79	5.86	10.14	0.41	11.04	12.15
18	17	5	Slope	0-10	S0	1	5.11	0.247	0.94	261	13.25	7.55	0.61	1.86	2.61
19	18	5	Slope	0-10	S0	2	5.46	0.298	0.96	300	12.3	7.5	0.68	2	1.98
20	19	5	Slope	0-10	S0	3	5.61	0.145	1.1	242	9.66	6.76	0.63	1.01	0.76
21	20	5	Slope	0-10	S0	4	5.85	0.186	1.2	229	13.78	7.12	0.62	3.09	2.85
22	21	6	Slope	10-30	S1	1	4.57	0.102	1.37	156	8.58	9.92	0.63	3.67	3.24
23	22	6	Slope	10-30	S1	2	5.11	0.097	1.3	139	8.58	8.69	0.42	4.7	4.63
24	23	6	Slope	10-30	S1	3	4.78	0.122	1.3	214	8.22	7.75	0.32	3.07	3.67

8.3.9 列转化为行索引

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1		Group	Contour	Depth	Gp	Block	pH	N	Dens	P	Ca	Mg	K	Na	Conduc
26	25	7	Slope	30-60	S3	1	3.96	0.059	1.53	98	4.8	10	0.36	6.52	7.72
27	26	7	Slope	30-60	S3	2	4	0.05	1.5	115	5.06	8.91	0.28	7.91	9.78
28	27	7	Slope	30-60	S3	3	4.12	0.086	1.55	148	6.16	7.58	0.16	6.39	9.07
29	28	7	Slope	30-60	S3	4	4.99	0.048	1.46	97	7.49	9.38	0.4	9.7	9.13
30	29	8	Slope	60-90	S6	1	3.8	0.049	1.48	108	3.82	8.8	0.24	9.57	11.57
31	30	8	Slope	60-90	S6	2	3.96	0.036	1.28	103	4.78	7.29	0.24	9.67	11.42
32	31	8	Slope	60-90	S6	3	3.93	0.048	1.42	109	4.93	7.47	0.14	9.65	13.32
33	32	8	Slope	60-90	S6	4	4.02	0.039	1.51	100	5.66	8.84	0.37	10.54	11.57
34	33	9	Depression	0-10	D0	1	5.24	0.194	1	445	12.27	6.27	0.72	1.02	0.75
35	34	9	Depression	0-10	D0	2	5.2	0.256	0.78	380	11.39	7.55	0.78	1.63	2.2
36	35	9	Depression	0-10	D0	3	5.3	0.136	1	259	9.96	8.08	0.45	1.97	2.27
37	36	9	Depression	0-10	D0	4	5.67	0.127	1.13	248	9.12	7.04	0.55	1.43	0.67
38	37	10	Depression	10-30	D1	1	4.46	0.087	1.24	276	7.24	9.4	0.43	4.17	5.08
39	38	10	Depression	10-30	D1	2	4.91	0.092	1.47	158	7.37	10.57	0.59	5.07	6.37
40	39	10	Depression	10-30	D1	3	4.79	0.047	1.46	121	6.99	9.91	0.3	5.15	6.82
41	40	10	Depression	10-30	D1	4	5.36	0.095	1.26	195	8.59	8.66	0.48	4.17	3.65
42	41	11	Depression	30-60	D3	1	3.94	0.054	1.6	148	4.85	9.62	0.18	7.2	10.14
43	42	11	Depression	30-60	D3	2	4.52	0.051	1.53	115	6.34	9.78	0.34	8.52	9.74
44	43	11	Depression	30-60	D3	3	4.35	0.032	1.55	82	5.99	9.73	0.22	7.02	8.6
45	44	11	Depression	30-60	D3	4	4.64	0.065	1.46	152	4.43	10.54	0.22	7.61	9.09
46	45	12	Depression	60-90	D6	1	3.82	0.038	1.4	105	4.65	9.85	0.18	10.15	12.26
47	46	12	Depression	60-90	D6	2	4.24	0.035	1.47	100	4.56	8.95	0.33	10.51	11.29
48	47	12	Depression	60-90	D6	3	4.22	0.03	1.56	97	5.29	8.37	0.14	8.27	9.51
49	48	12	Depression	60-90	D6	4	4.41	0.058	1.58	130	4.58	9.46	0.14	9.28	12.69

8.3.9 列值转化为行索引

注意，下面的soil_mean对象的行索引是两级索引：

```
df_soils = pd.read_csv("Soils.csv", index_col=0) # csv的第1列作为行索引
soils=df_soils[["Depth", "Contour", "Group", "pH", "N"]] # 取若干列
# 先按Depth分组，组内再按Contour分组，然后各列按分组求平均值
soils_g = soils.groupby(["Depth", "Contour"]).
soils_mean = soils_g.mean()
```

print(soils.head())

	Depth	Contour	Group	pH	N
1	0-10	Top	1	5.40	0.188
2	0-10	Top	1	5.65	0.165
3	0-10	Top	1	5.14	0.260
4	0-10	Top	1	5.14	0.169
5	10-30	Top	2	5.14	0.164

print(soils_mean.head())

		Group	pH	N
Depth	Contour			
0-10	Depression	9	5.3525	0.17825
	Slope	5	5.5075	0.21900
	Top	1	5.3325	0.19550
10-30	Depression	10	4.8800	0.08025
	Slope	6	5.2825	0.10100

分组：4 种Depth x 3 种Contour = 12 组

8.3.9 列值转化为行索引

set_index()方法将列值转换为行索引，如果append参数为False(默认值)，则删除当前的行索引; 若为True, 则为当前的索引添加新的级别。

例如，将soils_mean中的Group列设置为行索引，返回数据框具有了3级行索引。

soils_mean.head()

		Group	pH	N
Depth	Contour			
0-10	Depression	9	5.3525	0.17825
	Slope	5	5.5075	0.21900
	Top	1	5.3325	0.19550
10-30	Depression	10	4.8800	0.08025
	Slope	6	5.2825	0.10100

soils_mean.set_index('Group', append=True).head()

			pH	N
Depth	Contour	Group		
0-10	Depression	9	5.3525	0.17825
	Slope	5	5.5075	0.21900
	Top	1	5.3325	0.19550
10-30	Depression	10	4.8800	0.08025
	Slope	6	5.2825	0.10100

8.3.10 行索引转化（还原）为列

reset_index()方法将行索引转换为列，level参数可以指定被转换为列的级别(默认所有级别)。若只希望从行索引中删除某个级别，可以设置drop参数为True。

例：将soils_mean中的1级行索引Contour转化为列，返回的数据框为单级行索引。

```
soils_mean.head()
```

		Group	pH	N
Depth	Contour			
0-10	Depression	9	5.3525	0.17825
	Slope	5	5.5075	0.21900
	Top	1	5.3325	0.19550
10-30	Depression	10	4.8800	0.08025
	Slope	6	5.2825	0.10100

```
soils_mean.reset_index(level='Contour',
                       drop=True).head()
```

Depth	Group	pH	N
0-10	9	5.3525	0.17825
0-10	5	5.5075	0.21900
0-10	1	5.3325	0.19550
10-30	10	4.8800	0.08025
10-30	6	5.2825	0.10100

```
soils_mean.reset_index(
                    level='Contour').head()
```

Depth	Contour	Group	pH	N
0-10	Depression	9	5.3525	0.17825
0-10	Slope	5	5.5075	0.21900
0-10	Top	1	5.3325	0.19550
10-30	Depression	10	4.8800	0.08025
10-30	Slope	6	5.2825	0.10100

8.3.11 行索引和列索引的相互转换

stack()方法把指定级别的列索引转换为行索引，而**unstack()**则把行索引转换为列索引。例如：将行索引中的第1级转换为列索引的第1级，所得到的结果中行索引为单级索引，而列索引为两级索引。

soils_mean.head()

		Group	pH	N
Depth	Contour			
0-10	Depression	9	5.3525	0.17825
	Slope	5	5.5075	0.21900
	Top	1	5.3325	0.19550
10-30	Depression	10	4.8800	0.08025
	Slope	6	5.2825	0.10100

```
# 将行索引中的第1级转换为列索引的第1级
df=soils_mean.unstack(level=1)
print(df.head())
```

	Group			pH			N		
Contour	Depression	Slope	Top	Depression	Slope	Top	Depression	Slope	Top
Depth									
0-10	9	5	1	5.3525	5.5075	5.3325	0.17825	0.21900	0.19550
10-30	10	6	2	4.8800	5.2825	4.8500	0.08025	0.10100	0.11750
30-60	11	7	3	4.3625	4.2675	4.2050	0.05050	0.06075	0.07950
60-90	12	8	4	4.1725	3.9275	3.8925	0.04025	0.04300	0.05775

8.3.11 行索引和列索引的相互转换

对前面得到的df调用方法，将其列索引中的第1级转换为行索引的第1级，所得到的结果中列索引为**stack()**单级索引，而行索引为两级索引。返回的数据框与soils_mean完全相同。

df.stack(1).head()

		Group	pH	N
Depth	Contour			
0-10	Depression	9	5.3525	0.17825
	Slope	5	5.5075	0.21900
	Top	1	5.3325	0.19550
10-30	Depression	10	4.8800	0.08025
	Slope	6	5.2825	0.10100

df.stack(1).head()

df.head()

	Group			pH			N		
Contour	Depression	Slope	Top	Depression	Slope	Top	Depression	Slope	Top
Depth									
0-10	9	5	1	5.3525	5.5075	5.3325	0.17825	0.21900	0.19550
10-30	10	6	2	4.8800	5.2825	4.8500	0.08025	0.10100	0.11750
30-60	11	7	3	4.3625	4.2675	4.2050	0.05050	0.06075	0.07950
60-90	12	8	4	4.1725	3.9275	3.8925	0.04025	0.04300	0.05775

8.3.11 行索引和列索引的相互转换

无论是stack()还是unstack()，当所有的索引被转换到同一个轴上时，将得到一个 Series对象。

soils_mean.head()

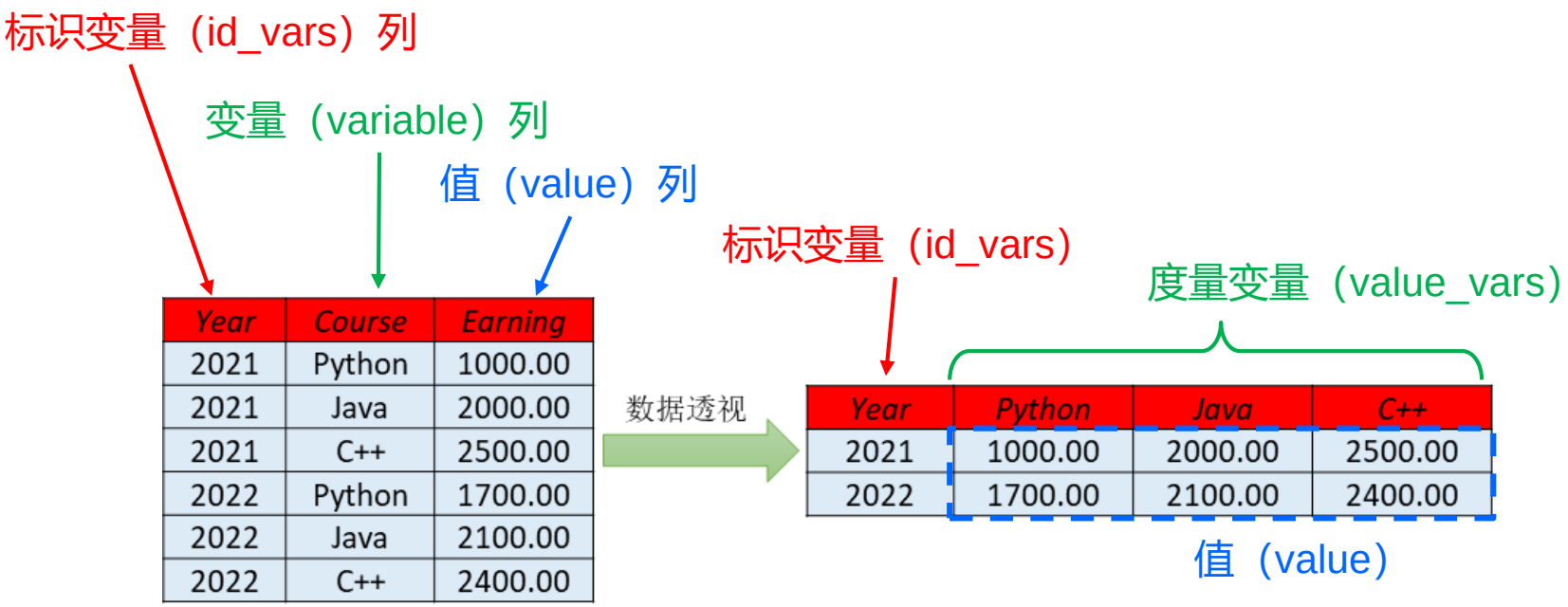
Depth	Contour	Group	pH	N
0-10	Depression	9	5.3525	0.17825
	Slope	5	5.5075	0.21900
	Top	1	5.3325	0.19550
10-30	Depression	10	4.8800	0.08025
	Slope	6	5.2825	0.10100

soils_mean.stack().head(10)

```
Depth  Contour
0-10   Depression  Group    9.00000
                   pH       5.35250
                   N        0.17825
       Slope       Group    5.00000
                   pH       5.50750
                   N        0.21900
       Top         Group    1.00000
                   pH       5.33250
                   N        0.19550
10-30  Depression  Group   10.00000
dtype: float64
```

8.3.12 数据透视与数据融合

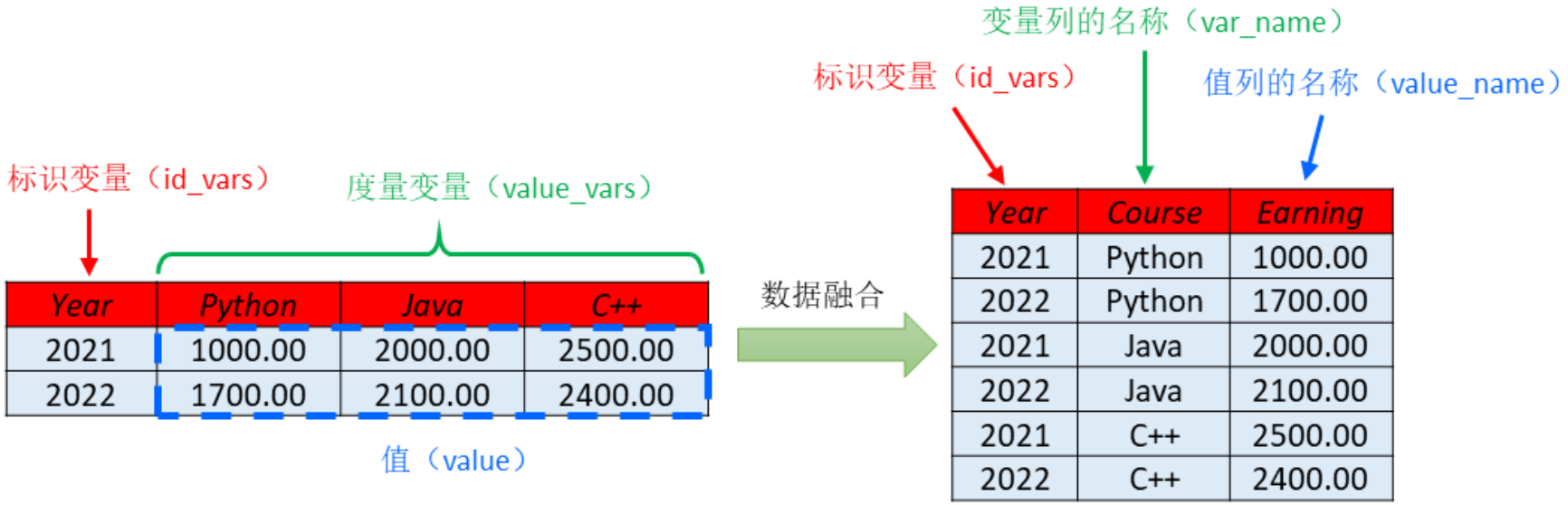
数据透视（pivot）的过程如图所示，对左表以Year为索引，按照Course列来透视Earning数据，从而得到右表，它以Year作为行索引，Course列的不同取值作为右表中的列名，某个年份的不同课程的收入在横向的一个行中显示。



特点：数据从（垂向）长格式转换为（横向）宽格式

8.3.12 数据透视与数据融合

数据融合（melt），也叫**数据逆透视**，把数据从宽格式转换为长格式。图中左表的Python列、Java列和C++列的名称作为右表中Course列的取值，原先在一行中的多列数据按课程名分裂成了多行，每行显示某年中一门课程的收入。



Year	Python	Java	C++
2021	1000.00	2000.00	2500.00
2022	1700.00	2100.00	2400.00

Year	Course	Earning
2021	Python	1000.00
2022	Python	1700.00
2021	Java	2000.00
2022	Java	2100.00
2021	C++	2500.00
2022	C++	2400.00

特点：数据从（横向）宽格式转换为（垂向）长格式，数据行数增加了。

8.3.12 数据透视与数据融合

pivot()可以将DataFrame中的3列数据分别作为行索引、列索引和元素值，将这三列数据转换为二维表格。例如：

```
data = [ [2021, 'Python', 1000, 80], [2021, 'Java', 2000, 140], [2021, 'C++', 2500, 185],
         [2022, 'Python', 1700, 93], [2022, 'Java', 2100, 162], [2022, 'C++', 2400, 205] ]
df = pd.DataFrame(data=data, columns=['Year', 'Course', 'Earning', 'StudentNum'])
# Earning指标在Year和Course索引上的数据透视
df1 = df.pivot( index='Year',  # 指定作为新表的行索引的列
                columns='Course',  # 指定作为新表的列索引的列
                values='Earning'  # 指定作为新表的元素值的列
)
```

df

	Year	Course	Earning	StudentNum
0	2021	Python	1000	80
1	2021	Java	2000	140
2	2021	C++	2500	185
3	2022	Python	1700	93
4	2022	Java	2100	162
5	2022	C++	2400	205

df1

Course	C++	Java	Python
Year			
2021	2500	2000	1000
2022	2400	2100	1700

8.3.12 数据透视与数据融合

pivot()的values参数一般只指定一列数据，但也可以指定多列。若不指定values参数，就将剩余的列都当作元素值列，得到多级列索引。

```
df2 = df.pivot(index='Year', columns='Course')
# 相当于
# df2 = df.pivot(index='Year', columns='Course', values=['Earning', 'StudentNum'])
```

df

	Year	Course	Earning	StudentNum
0	2021	Python	1000	80
1	2021	Java	2000	140
2	2021	C++	2500	185
3	2022	Python	1700	93
4	2022	Java	2100	162
5	2022	C++	2400	205

df1

	Earning			StudentNum		
Course	C++	Java	Python	C++	Java	Python
Year						
2021	2500	2000	1000	185	140	80
2022	2400	2100	1700	205	162	93

得到与unstack操作相似的结果

8.3.12 数据透视与数据融合

数据透视时，如果需要对数据进行聚合操作，则需要使用pivot_table方法。该方法和pivot方法的行为相似，但是会对值进行聚合操作，因此只能处理数值数据。

pivot_table(self, values=None, index=None, columns=None, aggfunc='mean', fill_value=None)

- values：指定需要透视并聚合的数据列。
- index：指定分组器，该列数据作为透视数据框的行索引。
- columns：指定分组器，该列数据作为透视数据框列索引。
- aggfunc：指定聚合的函数，默认为'mean'，可以是'sum'（或np.sum）等。
- fill_value：指定用于填充缺失值的值。

8.3.12 数据透视与数据融合

例如：Earning指标在Year和Course索引上的数据透视

```
data = [ [2021, 'Python', 1000, 80], [2021, 'Java', 2000, 140], [2021, 'C++', 2500, 185],
         [2021, 'Python', 600, 50], [2022, 'Python', 1700, 93], [2022, 'Java', 2100, 162],
         [2022, 'C++', 2400, 205], [2022, 'Java', 700, 26] ]
df = pd.DataFrame(data=data, columns=['Year', 'Course', 'Earning', 'StudentNum'])
# 数据透视
df1 = df.pivot_table(index='Year', # 指定作为新表的行索引的列
      columns='Course', # 指定作为新表的列索引的列
      values='Earning', # 指定作为新表的元素值的列
      aggfunc='sum' # 指定聚合函数
)
```

df

	Year	Course	Earning	StudentNum
0	2021	Python	1000	80
1	2021	Java	2000	140
2	2021	C++	2500	185
3	2021	Python	600	50
4	2022	Python	1700	93
5	2022	Java	2100	162
6	2022	C++	2400	205
7	2022	Java	700	26

df1

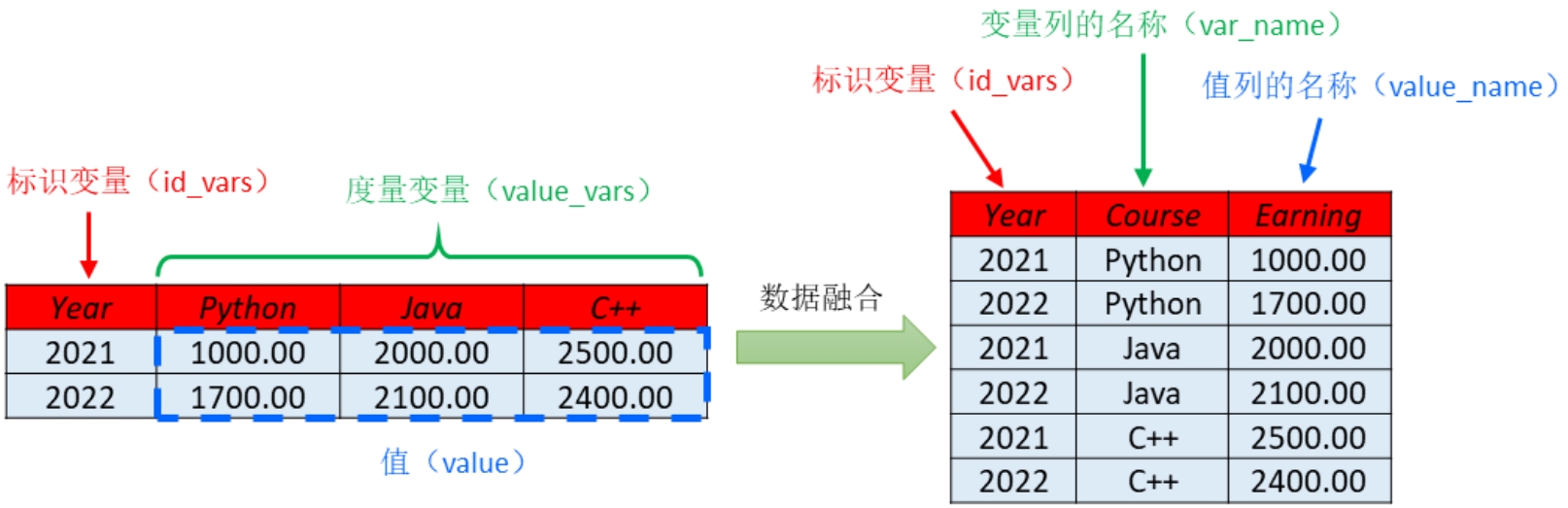
Course	C++	Java	Python
Year			
2021	2500	2000	1600
2022	2400	2800	1700

8.3.12 数据透视与数据融合

melt()方法实现了融合数据功能，它将数据从宽格式转换为长格式。

melt(self, id_vars=None, value_vars=None, var_name=None, value_name='value')

- id_vars：作为标识变量的列，可以是一列，也可以是多列。
- value_vars：作为值的列，也就是指定需要分裂的度量变量的索引列表。
- var_name：默认值是variable，对长格式中度量变量的列名所在的列进行命名，也就是列索引作为一列的值后，指定该列的索引（列名）。
- value_name：默认值是value，对长格式中度量变量的列值所在的列进行命名，也就是行中的值分裂到一列后，指定该列的索引（列名）。



Year	Python	Java	C++
2021	1000.00	2000.00	2500.00
2022	1700.00	2100.00	2400.00

Year	Course	Earning
2021	Python	1000.00
2022	Python	1700.00
2021	Java	2000.00
2022	Java	2100.00
2021	C++	2500.00
2022	C++	2400.00

8.3.12 数据透视与数据融合

例如，上图所示的数据融合的具体实现为：

```
data = [[2021, 1000, 2000, 2500], [2022, 1700, 2100, 2400]]
df = pd.DataFrame(data=data, columns=['Year', 'Python', 'Java', 'C++'])
df1 = df.melt(
    id_vars='Year', # 指定作为标识变量的列
   value_vars=['Python', 'Java', 'C++'], # 指定需要分裂的度量变量的列索引集合
   var_name='Course', # 指定度变量的名称构成的新列的索引（列名）
   value_name='Earning' # 指定度变量的值构成的新列的索引（列名）
)
```

df

	Year	Python	Java	C++
0	2021	1000	2000	2500
1	2022	1700	2100	2400

df1

	Year	Course	Earning
0	2021	Python	1000
1	2022	Python	1700
2	2021	Java	2000
3	2022	Java	2100
4	2021	C++	2500
5	2022	C++	2400

8.4 数值运算函数

pandas提供了各种数值运算方法，例如 max()、 min()、 mean()、 std()等。这些函数都有如下3个常用参数：

- axis: 指定运算对应的轴。
- level: 指定运算对应的索引级别。
- skipna: 运算是否自动跳过NaN。

```
# pandas DataFrame控制台打印输出设置浮点数小数位数
pd.options.display.float_format = '{:.2f}'.format
# 或者
# pd.options.display.float_format = lambda x:'%.2f'%x
print(df_soil.mean()) # 在第0轴上计算平均值，即每列的平均值
print(df_soil.mean(level=0)) #第0级行索引上分组计算各列平均值
print(df_soil.mean(level=1)) #在第1级行索引上分组计算各列平均值
print(df_soil.mean(axis=1)) #第1轴上计算平均值，即每行的平均值
```

```
Measures
pH        5.20
Dens      1.18
Ca       10.60
Conduc    3.14
dtype: float64
```

Measures	pH	Dens	Ca	Conduc
Depth				
0-10	5.40	1.01	12.11	1.63
10-30	5.00	1.35	9.10	4.66

Measures	pH	Dens	Ca	Conduc
Contour				
Depression	5.12	1.17	9.12	3.48
Slope	5.40	1.20	10.88	3.48
Top	5.09	1.17	11.81	2.48

```
Depth   Contour
0-10    Depression    4.62
        Slope         5.21
        Top           5.27
10-30   Depression    4.82
        Slope         5.26
        Top           5.00
dtype: float64
```

8.4 数值运算函数

pandas提供了各种数值运算方法，例如 max()、 min()、 mean()、 std()等。这些函数都有如下3个常用参数：

- axis: 指定运算对应的轴。
- level: 指定运算对应的索引级别。
- skipna: 运算是否自动跳过NaN。

```
>>> df=pd.DataFrame([[1,2,3],[4,5,6]],columns=['a','b','c'])
>>> df
   a  b  c
0  1  2  3
1  4  5  6
```

```
>>> df.sum(axis=0)  # 求列和
a    5
b    7
c    9
dtype: int64

>>> df.sum(axis=1)  # 求行和
0    6
1   15
dtype: int64
```

```
>>> df.sum(axis=1)>10  # 行和是否大于10
0    False
1     True
dtype: bool

>>> df[df.sum(axis=1)>10]  # 选出行和大于10的行
   a  b  c
1  4  5  6
```

8.4 数值运算函数

除了支持**加减乘除等运算符**之外，Pandas还提供了 add()、 sub() 、 mul() 、 div()、 mod()等与二元运算符对应的函数。可以通过**axis、level**和 **fill_value**等参数控制其运算行为, fill_value参数用于指定不存在的值或NaN时使用的默认值。

Measures		pH	Dens	Ca	Conduc	Date	Name
Depth	Contour						
0-10	Depression	5.3525	0.9775	10.6850	1.4725	2015-05-26	Lois
	Slope	5.5075	1.0500	12.2475	2.0500	2015-04-30	Roy
	Top	5.3325	1.0025	13.3850	1.3725	2015-05-21	Roy
10-30	Depression	4.8800	1.3575	7.5475	5.4800	2015-03-21	Lois
	Slope	5.2825	1.3475	9.5150	4.9100	2015-02-06	Diana
	Top	4.8500	1.3325	10.2375	3.5825	2015-04-11	Diana

```
df_soil.pH + 1  # 单列+1
df_soil.loc[:, ['pH', 'Ca']] * 0.5  # 两列*05
df_soil.loc[(slice(None), 'Top'), ['pH', 'Ca']] * 0.5  # 第1级行索引为Top的行中两列*05
df_soil.iloc[0:2, 0:2] * 0.5  # 头两行中前2列*05（注意文本和日期类型不知支持该概运算）
```

Depth	Contour	
0-10	Depression	6.35
	Slope	6.51
	Top	6.33
10-30	Depression	5.88
	Slope	6.28
	Top	5.85

Name: pH, dtype: float64

Measures		pH	Ca
Depth	Contour		
0-10	Depression	2.68	5.34
	Slope	2.75	6.12
	Top	2.67	6.69
10-30	Depression	2.44	3.77
	Slope	2.64	4.76
	Top	2.42	5.12

Measures		pH	Ca
Depth	Contour		
0-10	Top	2.67	6.69
10-30	Top	2.42	5.12

Measures		pH	Dens
Depth	Contour		
0-10	Depression	2.68	0.49
	Slope	2.75	0.53

8.4 数值运算函数


```
Depth   Contour
0-10    Depression    9.62
        Slope        14.70
        Top          13.39
10-30   Depression    6.79
        Slope        11.42
        Top          10.24
dtype: float64
```

```
Depth   Contour
0-10    Depression    9.62
        Slope        14.70
        Top           0.00
10-30   Depression    6.79
        Slope        11.42
        Top           0.00
dtype: float64
```

```
Measures              pH     Ca
Depth Contour
0-10  Depression  2.68  21.37
      Slope       2.75  24.50
      Top         2.67  26.77
10-30 Depression  2.44  15.10
      Slope       2.64  19.03
      Top         2.42  20.48
```

```
s = pd.Series(dict(Depression=0.9, Slope=1.2))
df_soil.Ca.mul(s, level=1, fill_value=1)
df_soil.Ca.mul(s, level=1, fill_value=0)
s2 = pd.Series(dict(pH=0.5, Ca=2))
df_soil[['pH','Ca']].mul(s2)# 两列乘不同的系数
```

Measures		pH	Dens	Ca	Conduc	Date	Name
Depth	Contour						
0-10	Depression	5.3525	0.9775	10.6850	1.4725	2015-05-26	Lois
	Slope	5.5075	1.0500	12.2475	2.0500	2015-04-30	Roy
	Top	5.3325	1.0025	13.3850	1.3725	2015-05-21	Roy
10-30	Depression	4.8800	1.3575	7.5475	5.4800	2015-03-21	Lois
	Slope	5.2825	1.3475	9.5150	4.9100	2015-02-06	Diana
	Top	4.8500	1.3325	10.2375	3.5825	2015-04-11	Diana

8.5 字符串处理

- Series对象提供了大量的字符串处理方法，例如upper(),capitalize(),split(),len(),cat(),+运算符，**replace(),strip(),find(),match(),startswith(), endswith(),** map()等。因数量众多，因此Pandas使用了对象str来包装这些方法。

取子串则使用切片操作，例如：s.str[2:]

```
s = pd.Series(['Python Programming', 'thank you!', '我爱UPC'])
print(s.str.upper()) # 每个元素转化为大写字母
print(s.str.len()) # Unicode字符数(实际字符个数)
print(s.str.encode('GBK').str.len()) # 编码后的字节数(一个汉字占用2字节)
print(s.str.encode('UTF-8').str.len()) # 编码后的字节数(一个汉字占用3字节)
print(s.str.split(' ')) # 每个元素分割成列表
# 也可以在序列上使用map()函数,它将针对每个元素运算的函数运用到整个序列之上。
print(s.map(lambda x: x.capitalize())) # 每个元素的首字母大写
```

```
0    PYTHON PROGRAMMING
1          THANK YOU!
2               我爱UPC
dtype: object
```

```
0    18
1    10
2     5
dtype: int64
```

```
0    18
1    10
2     7
dtype: int64
```

```
0    18
1    10
2     9
dtype: int64
```

```
0    [Python, Programming]
1          [thank, you!]
2                [我爱UPC]
dtype: object
```

```
0    Python programming
1          Thank you!
2               我爱upc
dtype: object
```

8.5 字符串处理

- **示例：**从网站（example.python-scraping.com）爬取到如下数据，读入DataFrame内，并将其中的第二列转化为int类型。

```
name,population
Antigua and Barbuda,"86,754"
Antarctica,0
Anguilla,"13,254"
Angola,"13,068,161"
```

- 问题1：双引号是否需要处理？
- 问题2：逗号如何处理？

```
import pandas as pd
import numpy as np

header = ['name', 'population']
data = [['Antigua and Barbuda', '86,754'],
        ['Antarctica', '0'],
        ['Anguilla', '13,254'],
        ['Angola', '13,068,161']]
df = pd.DataFrame(data, columns=header)
df=pd.read_csv('scrapy_countries_or_districts.csv').head(4)
print(df)

print(df.dtypes)
```

```
                   name  population
0  Antigua and Barbuda      86,754
1            Antarctica           0
2              Anguilla      13,254
3                Angola  13,068,161
name          object
population    object
dtype: object
```

8.5 字符串处理

- **示例：**从网站（example.python-scraping.com）爬取到如下数据，读入DataFrame内，并将其中的第二列转化为int类型。

```
s = df['population']
#s1 = s.map(lambda x: x.replace(',', '') )
s1 = s.str.replace(',', '')  # 替换字符串
print(s1)
df['population'] = s1.astype(np.int)  # 转换数据类型，并更新列
print(df)

print(df.dtypes)
```

思考：如果数据中含有单位，例如'12km', 如何剔除单位?

```
0       86754
1           0
2       13254
3    13068161
Name: population, dtype: object
```

```
[   86754        0    13254 13068161]
```

```
                   name  population
0  Antigua and Barbuda       86754
1            Antarctica           0
2              Anguilla       13254
3                Angola    13068161
```

```
name          object
population     int32
dtype: object
```

8.5 字符串处理

- **DataFrame和Series对象的replace方法可用于元素替换，元素类型可以是任意的，该方法替换灵活。**

```
replace(self, to_replace=None, value=None, inplace=False,
limit=None, regex=False, method='pad')
```

- **to_replace**：需要被替换的元素，可以是标量（此时用value参数指定的值替换），或是列表（value也是等长的列表），还可以是字典(不同的元素用不同的值替换，此时value无效；DataFrame不同列中的特定值用value替换)。
- **regex**：在字符串替换时，决定是否使用正则表达式。
- **method**: {'pad', 'ffill', 'bfill', None}，当to_replace参数是标量、列表或元组，且value参数是None时，替换元素的方法。

Scalar `to_replace` and `value`

```
>>> s = pd.Series([0, 1, 2, 3, 4])
>>> s.replace(0, 5)
0    5
1    1
2    2
3    3
4    4
dtype: int64
```

```
>>> df = pd.DataFrame({'A': [0, 1, 2, 3, 4],
    ...                     'B': [5, 6, 7, 8, 9],
    ...                     'C': ['a', 'b', 'c', 'd', 'e']})
>>> df.replace(0, 5)
   A  B  C
0  5  5  a
1  1  6  b
2  2  7  c
3  3  8  d
4  4  9  e
```

List-like `to_replace`

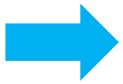
```
>>> df
   A  B  C
0  0  5  a
1  1  6  b
2  2  7  c
3  3  8  d
4  4  9  e
```

```
>>> df.replace([0, 1, 2, 3], 4)
   A  B  C
0  4  5  a
1  4  6  b
2  4  7  c
3  4  8  d
4  4  9  e
```

```
>>> df.replace([0, 1, 2, 3], [4, 3, 2, 1])
   A  B  C
0  4  5  a
1  3  6  b
2  2  7  c
3  1  8  d
4  4  9  e
```

List-like `to_replace`

```
>>> s
0    5
1    1
2    2
3    3
4    4
dtype: int64
```



```
>>> s.replace([1, 2], method='bfill')
0    0
1    3
2    3
3    3
4    4
dtype: int64
```

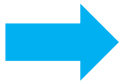
dict-like `to_replace`

```
>>> df
   A  B  C
0  0  5  a
1  1  6  b
2  2  7  c
3  3  8  d
4  4  9  e
```

```
>>> df.replace({0: 10, 1: 100})
     A  B  C
0   10  5  a
1  100  6  b
2    2  7  c
3    3  8  d
4    4  9  e
```

dict-like `to_replace`

```
>>> df
   A  B  C
0  0  5  a
1  1  6  b
2  2  7  c
3  3  8  d
4  4  9  e
```



```
>>> df.replace({'A': 0, 'B': 5}, 100)
     A    B  C
0  100  100  a
1    1    6  b
2    2    7  c
3    3    8  d
4    4    9  e
```

```
>>> df
   A  B  C
0  0  5  a
1  1  6  b
2  2  7  c
3  3  8  d
4  4  9  e
```

```
>>> df.replace({'A': {0: 100, 4: 400}})
     A  B  C
0  100  5  a
1    1  6  b
2    2  7  c
3    3  8  d
4  400  9  e
```

Regular expression `to_replace`

```
>>> df = pd.DataFrame({'A': ['bat', 'foo', 'bait'],
...                    'B': ['abc', 'bar', 'xyz']})
>>> df.replace(to_replace=r'^ba.$', value='new', regex=True)
      A    B
0   new  abc
1   foo  new
2  bait   xyz
```

```
>>> df.replace({'A': r'^ba.$'}, {'A': 'new'}, regex=True)
      A    B
0   new  abc
1   foo  bar
2  bait   xyz
```

```
>>> df.replace(regex=r'^ba.$', value='new')
      A    B
0   new  abc
1   foo  new
2  bait   xyz
```

Regular expression `to_replace`

```
>>> df
     A    B
0   bat  abc
1   foo  bar
2  bait  xyz
```

```
>>> df.replace(regex={r'^ba.$': 'new', 'foo': 'xyz'})
     A    B
0  new  abc
1  xyz  new
2  bait  xyz
```

```
>>> df.replace(regex=[r'^ba.$', 'foo'], value='new')
     A    B
0  new  abc
1  new  new
2  bait  xyz
```

Compare the behavior of ``s.replace({'a': None})`` and ``s.replace('a', None)`` to understand the peculiarities of the `to_replace` parameter

```
>>> s = pd.Series([10, 'a', 'a', 'b', 'a'])
```

```
0      10
1    None
2    None
3       b
4    None
dtype: object
```

s.replace({'a': None})

s.replace(to_replace={'a': None}, value=None, method=None)

```
0    10
1    10
2    10
3     b
4     b
dtype: object
```

s.replace('a', None)

s.replace(to_replace='a', value=None, method='pad')

pad：衬垫

一个字符串列替换的例子

地区,人口数量
北京,21893095
天津,13866009
河北,74610235
山西,34915616
内蒙古,24049155
辽宁,42591407
吉林,24073453
黑龙江,31850088
上海,24870895
江苏,84748016
浙江,64567588
…
香港,7500957
澳门,683100
台湾,23833611

使用pyechart包中的map模块绘图，省份、直辖市、自治区、特别行政区名称需使用标准全称，例如：台湾省、北京市、西藏自治区、香港特别行政区

```
# 读入作图数据
data: pd.DataFrame = pd.read_csv('2020年中国各省人口.csv')
# 省份、自治区、直辖市、特别行政区采用标准名称
data['地区'] = data['地区'] + '省'  # 地区名后面统一加上'省'，之后在替换特殊名称
special = {'北京省': '北京市', '天津省': '天津市', '上海省': '上海市', '重庆省': '重庆市',  # 直辖市
           '西藏省': '西藏自治区', '新疆省': '新疆维吾尔自治区', '内蒙古省': '内蒙古自治区',
           '宁夏省': '宁夏回族自治区', '广西省': '广西壮族自治区',  # 自治区
           '香港省': '香港特别行政区', '澳门省': '澳门特别行政区',  # 特别行政区
          }
data['地区'].replace(to_replace=special, inplace=True)  # 根据不同的特殊值替换
```

data

	地区	人口数量
0	北京	21893095
1	天津	13866009
2	河北	74610235
3	山西	34915616
4	内蒙古	24049155
5	辽宁	42591407
6	吉林	24073453
7	黑龙江	31850088
8	上海	24870895

data

	地区	人口数量
0	北京市	21893095
1	天津市	13866009
2	河北省	74610235
3	山西省	34915616
4	内蒙古自治区	24049155
5	辽宁省	42591407
6	吉林省	24073453
7	黑龙江省	31850088
8	上海市	24870895

8.6 日期处理

- 日期类型（datatime）转换函数：pd.to_datetime()，例如：

  df['publish_time'] = **pd.to_datetime**(df['publish_time'])

- Pandas的 **dt 模块**用于处理日期类型数据，常见用途是提取年月日时分秒

属性	说明
year	datetime 的年
month	datetime 的月
day	datetime 的日
hour	datetime 的小时
minute	datetime 的分钟
second	datetime 的秒
microsecond	datetime 的微秒
nanosecond	datetime 的纳秒
date	返回 datetime.date（不包含时区信息）
time	返回 datetime.time（不包含时区信息）
timetz	返回带本地时区信息的 datetime.time

属性	说明
dayofyear	一年里的第几天
weekofyear	一年里的第几周
week	一年里的第几周
dayofweek	一周里的第几天，Monday=0, Sunday=6
weekday	一周里的第几天，Monday=0, Sunday=6
weekday_name	这一天是星期几（如，Friday）
quarter	日期所处的季节：Jan-Mar = 1，Apr-Jun = 2 等
days_in_month	日期所在的月有多少天
is_month_start	逻辑判断是不是月初（由频率定义）
is_month_end	逻辑判断是不是月末（由频率定义）
is_quarter_start	逻辑判断是不是季初（由频率定义）
is_quarter_end	逻辑判断是不是季末（由频率定义）
is_year_start	逻辑判断是不是年初（由频率定义）
is_year_end	逻辑判断是不是年末（由频率定义）
is_leap_year	逻辑判断是不是日期所在年是不是闰年

```
# 先转化为datetime类型,默认format='%Y-%m-%d %H:%M:%S'
df['datetime'] = pd.to_datetime(df['datetime'], errors='coerce')
df['date'] = df['datetime'].dt.date  # 转化提取年-月-日
df['year'] = df['datetime'].dt.year.fillna(0).astype('int')  # 转化提取年 ,
# 如果有NaN元素则默认转化float64型,
# 要转换数据类型则需要先填充空值,再做数据类型转换
df['month'] = df['datetime'].dt.month.fillna(0).astype('int')  # 转化提取月
df['%Y_%m'] = df['year'].map(str) + '-' + df['month'].map(str)  # 转化获取年-月
df['day'] = df['datetime'].dt.day.fillna(0).astype('int')  # 转化提取天
df['hour'] = df['datetime'].dt.hour.fillna(0).astype('int')  # 转化提取小时
df['minute'] = df['datetime'].dt.minute.fillna(0).astype('int')  # 转化提取分钟
df['second'] = df['datetime'].dt.second.fillna(0).astype('int')  # 转化提取秒
df['dayofyear'] = df['datetime'].dt.dayofyear.fillna(0).astype('int')  # 一年中的第n天
df['weekofyear'] = df['datetime'].dt.weekofyear.fillna(0).astype('int')  # 一年中的第n周
# 周几, 一周里的第几天, Monday=0, Sunday=6
df['weekday'] = df['datetime'].dt.weekday.fillna(0).astype('int')
df['quarter'] = df['datetime'].dt.quarter.fillna(0).astype('int')  # 季度
```

8.6 日期处理

- 另一种方法：利用字符串截取原理，提取年月日时分秒

```
df['datetime'] = pd.to_datetime(df['datetime'],
                                errors='coerce').fillna('0000-00-00 00:00:00')
# 转化为字符串类型, 利用截取字符串获取年月日时分秒
df['date'] = df['datetime'].astype('str').str[0:10]  # 截取年-月-日
df['%Y-%m'] = df['datetime'].astype('str').str[0:7]  # 截取年-月
df['day'] = df['datetime'].astype('str').str[8:10]  # 截取日

df['hour'] = df['datetime'].astype('str').str[11:13]  # 截取小时
```

8.7 缺失数据的处理

Pandas使用NaN（Not-a-Number）和NaT（Not-a-Time）表示缺失值（missing values），NaT专门表示时间类型的缺失值。缺失值，也叫空值，可记为NA（Not Available），non-NA则表示非空值。np.nan、pd.NA和pd.NaT都是代表缺失值的符号常量。

在数据框中，与NA相关的方法常用的有：

- **where()**方法：不满足条件的元素设置为NA;
- **isnull()**和 **notnull()**方法：用于判断元素值是否为NA;
- **count()**方法：返回每行或每列的non-NA元素的个数;
- **dropna()**方法：删除包含NA的行或列;
- **ffill()**、**bfill()**和**interpolate()**方法：分别用NA之前的元素值，之后的元素值，或前后元素的插值填充该NA元素;
- **fillna()**方法：将NA值填充为value参数指定的值。

8.7 缺失数据的处理

- where()方法

where()方法将**不满足**指定条件的元素设置为NA，返回Series或DataFrame。由于整数列无法使用NA, 因此如果整数类型的列出现缺失数据，则会被自动转换为浮点数类型。

```
s = pd.Series([2, 0, 4, 8, 1])
print(s)
print(s > 1)
s1= s.where(s > 1)  # 小于等于1的元素置为NA，返回一个Series
print(s1)
```

s	s > 1	s1= s.where(s > 1)
0 2 1 0 2 4 3 8 4 1 dtype: int64	0 True 1 False 2 True 3 True 4 False dtype: bool	0 2.0 1 NaN 2 4.0 3 8.0 4 NaN dtype: float64

8.7 缺失数据的处理

- where()方法

```
np.random.seed(0)  # 固定随机数种子，每次运行程序产生的伪随机系列固定
A = np.random.randint(0, 10, (10, 3))
df = pd.DataFrame(data=A, columns=['A', 'B', 'C'])
df1 = df.where(df > 2)  # 小于等于2的元素全部置为NA，返回一个DataFrame
```

df

	A	B	C
0	5	0	3
1	3	7	9
2	3	5	2
3	4	7	6
4	8	8	1
5	6	7	7
6	8	1	5
7	9	8	9
8	4	3	0
9	3	5	0

df1 = df.where(df > 2)

	A	B	C
0	5	NaN	3.0
1	3	7.0	9.0
2	3	5.0	NaN
3	4	7.0	6.0
4	8	8.0	NaN
5	6	7.0	7.0
6	8	NaN	5.0
7	9	8.0	9.0
8	4	3.0	NaN
9	3	5.0	NaN

注意：where()方法设置inplace参数为True，则修改这个df本身而不是返回一个新的df。即：
df.**where**(df > 2, **inplace**=True)
等效于 df=df.**where**(df_int > 2)

8.7 缺失数据的处理

- isnull()和 notnull()方法

isnull()和 notnull()方法用于判断元素值是否为NA，返回布尔型Series或DataFrame。

s1		s1.**isnull**()	
0	2.0	0	False
1	NaN	1	True
2	4.0	2	False
3	8.0	3	False
4	NaN	4	True
dtype: float64		dtype: bool	

df1

	A	B	C
0	5	NaN	3.0
1	3	7.0	9.0
2	3	5.0	NaN
3	4	7.0	6.0
4	8	8.0	NaN
5	6	7.0	7.0
6	8	NaN	5.0
7	9	8.0	9.0
8	4	3.0	NaN
9	3	5.0	NaN

df1.**isnull**()

	A	B	C
0	False	True	False
1	False	False	False
2	False	False	True
3	False	False	False
4	False	False	True
5	False	False	False
6	False	True	False
7	False	False	False
8	False	False	True
9	False	False	True

8.7 缺失数据的处理

- count()方法

count()方法返回每行或每列的non-NA元素的个数，axis参数表示在0轴（对应列）或1轴（对应行）上统计。

s1

```
0    2.0
1    NaN
2    4.0
3    8.0
4    NaN
dtype: float64
```

s1.count()

```
3
```

df1

	A	B	C
0	5	NaN	3.0
1	3	7.0	9.0
2	3	5.0	NaN
3	4	7.0	6.0
4	8	8.0	NaN
5	6	7.0	7.0
6	8	NaN	5.0
7	9	8.0	9.0
8	4	3.0	NaN
9	3	5.0	NaN

df1.count()

```
A    10
B     8
C     6
dtype: int64
```

df1.count(axis=1)

```
0    2
1    3
2    2
3    3
4    2
5    3
6    2
7    3
8    2
9    2
dtype: int64
```

注意其他对象的count()方法：

```
>>> 'abc'.count('ab')
1
>>> ['abcde','e','a','e'].count('e')
2
```

8.7 缺失数据的处理

- dropna方法
  - 对于包含NA元素的数据，最简单的办法就是调用dropna()以删除包含NA的行或列，当全部使用默认参数（axis=0）时，将删除包含NA的所有行。
  - 如果使用thresh参数，则保留至少有thresh个non-NA元素的行或列。

```
print(df1.dropna()) # 删除包含NA的行
print(df1.dropna(axis=1)) # 删除包含NA的列
print(df1.dropna(axis=1, thresh=7) ) # 保留至少含有7个non-NA的列
```

df1

	A	B	C
0	5	NaN	3.0
1	3	7.0	9.0
2	3	5.0	NaN
3	4	7.0	6.0
4	8	8.0	NaN
5	6	7.0	7.0
6	8	NaN	5.0
7	9	8.0	9.0
8	4	3.0	NaN
9	3	5.0	NaN

df1.dropna()

	A	B	C
1	3	7.0	9.0
3	4	7.0	6.0
5	6	7.0	7.0
7	9	8.0	9.0

df1.dropna(axis=1, thresh=7)

	A	B
0	5	NaN
1	3	7.0
2	3	5.0
3	4	7.0
4	8	8.0
5	6	7.0
6	8	NaN
7	9	8.0
8	4	3.0
9	3	5.0

8.7 缺失数据的处理

- ffill()、bfill()和interpolate()方法

ffill()、bfill()和interpolate()方法分别使用NA之前的元素值，之后的元素值，或前后元素的插值填充该NA元素。 填充顺序，默认bfill()为backward, 其他为forward。

```
print(df1.ffill())  # 在0轴（列）上填充NA之前的元素值
print(df1.ffill(axis=1))  # 在1轴（行）上填充NA之前的元素值
print(df1.bfill())  # 在0轴（列）上填充NA之后的元素值
print(df1.interpolate())  # 在0轴（列）上填充NA之前后元素的插值
```

df1

	A	B	C
0	10	3.0	NaN
1	10	NaN	3.0
2	19	7.0	5.0
3	18	3.0	3.0
4	12	6.0	NaN
5	14	6.0	9.0
6	13	8.0	4.0
7	17	6.0	NaN
8	15	NaN	NaN
9	15	3.0	NaN

df1.ffill()

	A	B	C
0	10	3.0	NaN
1	10	3.0	3.0
2	19	7.0	5.0
3	18	3.0	3.0
4	12	6.0	3.0
5	14	6.0	9.0
6	13	8.0	4.0
7	17	6.0	4.0
8	15	6.0	4.0
9	15	3.0	4.0

df1.ffill(axis=1)

	A	B	C
0	10.0	3.0	3.0
1	10.0	10.0	3.0
2	19.0	7.0	5.0
3	18.0	3.0	3.0
4	12.0	6.0	6.0
5	14.0	6.0	9.0
6	13.0	8.0	4.0
7	17.0	6.0	6.0
8	15.0	15.0	15.0
9	15.0	3.0	3.0

df1.bfill()

	A	B	C
0	10	3.0	3.0
1	10	7.0	3.0
2	19	7.0	5.0
3	18	3.0	3.0
4	12	6.0	9.0
5	14	6.0	9.0
6	13	8.0	4.0
7	17	6.0	NaN
8	15	3.0	NaN
9	15	3.0	NaN

df1.interpolate()

	A	B	C
0	10	3.0	NaN
1	10	5.0	3.0
2	19	7.0	5.0
3	18	3.0	3.0
4	12	6.0	6.0
5	14	6.0	9.0
6	13	8.0	4.0
7	17	6.0	4.0
8	15	4.5	4.0
9	15	3.0	4.0

8.7 缺失数据的处理

- ffill()、bfill()和interpolate()方法

```
df_nan.iloc[5,2]=np.nan   # 修改元素值为nan
print(df_nan)
print(df_nan.interpolate())
```

	A	B	C
0	10	3.0	NaN
1	10	NaN	3.0
2	19	7.0	5.0
3	18	3.0	3.0
4	12	6.0	NaN
5	14	6.0	9.0
6	13	8.0	4.0
7	17	6.0	NaN
8	15	NaN	NaN
9	15	3.0	NaN

df_nan

	A	B	C
0	10	3.0	NaN
1	10	NaN	3.0
2	19	7.0	5.0
3	18	3.0	3.0
4	12	6.0	NaN
5	14	6.0	NaN
6	13	8.0	4.0
7	17	6.0	NaN
8	15	NaN	NaN
9	15	3.0	NaN

df_nan. interpolate()

	A	B	C
0	10	3.0	NaN
1	10	5.0	3.000000
2	19	7.0	5.000000
3	18	3.0	3.000000
4	12	6.0	3.333333
5	14	6.0	3.666667
6	13	8.0	4.000000
7	17	6.0	4.000000
8	15	4.5	4.000000
9	15	3.0	4.000000

8.7 缺失数据的处理

- fillna()方法

  fillna()方法将NA值填充为value参数指定的值。value参数若为字典，对不同的列使用不同的值填充NA。

```
np.random.seed(41)
A = np.random.randint(0, 10, (10, 3))
df_int = pd.DataFrame(data=A, columns=list("ABC"))
df_int["A"] += 10  # 修改第一列元素：各元素+10
df_nan = df_int.where(df_int > 2)  # 小于等于2的元素全部置为NA，返回一个DataFrame
```

df_nan

	A	B	C
0	10	3.0	NaN
1	10	NaN	3.0
2	19	7.0	5.0
3	18	3.0	3.0
4	12	6.0	NaN
5	14	6.0	9.0
6	13	8.0	4.0
7	17	6.0	NaN
8	15	NaN	NaN
9	15	3.0	NaN

df_nan.**fillna**(value='-9999')

	A	B	C
0	10	3	-9999
1	10	-9999	3
2	19	7	5
3	18	3	3
4	12	6	-9999
5	14	6	9
6	13	8	4
7	17	6	-9999
8	15	-9999	-9999
9	15	3	-9999

df_nan.**fillna**(value={'B':-9999,'C':0})

	A	B	C
0	10	3.0	0.0
1	10	-9999.0	3.0
2	19	7.0	5.0
3	18	3.0	3.0
4	12	6.0	0.0
5	14	6.0	9.0
6	13	8.0	4.0
7	17	6.0	0.0
8	15	-9999.0	0.0
9	15	3.0	0.0

8.7 缺失数据的处理

各种聚合方法的skipna参数默认为True, 因此计算时将忽略NA元素，注意每行或每列是单独运算的。**如果需要忽略包含NA的整行**，需要先调用dropna()。若将skipna参数设置为False, 则包含NA的行或列的运算结果为NA。

df_nan

	A	B	C
0	10	3.0	NaN
1	10	NaN	3.0
2	19	7.0	5.0
3	18	3.0	3.0
4	12	6.0	NaN
5	14	6.0	9.0
6	13	8.0	4.0
7	17	6.0	NaN
8	15	NaN	NaN
9	15	3.0	NaN

df_nan.sum()

```
A    143.0
B     42.0
C     24.0
dtype: float64
```

df_nan.sum(**skipna**=False)

```
dtype: float64
A    143.0
B      NaN
C      NaN
dtype: float64
```

df_nan.**dropna**(axis=0).**sum**()

```
A    64.0
B    24.0
C    21.0
dtype: float64
```

8.8 删除指定数据和重复数据

- Series的drop方法和drop_duplicates()方法分别用于删除序列中指定的数据或重复的数据。
- DataFrame的drop方法和drop_duplicates()方法分别用于删除数据框中指定的行或重复行。

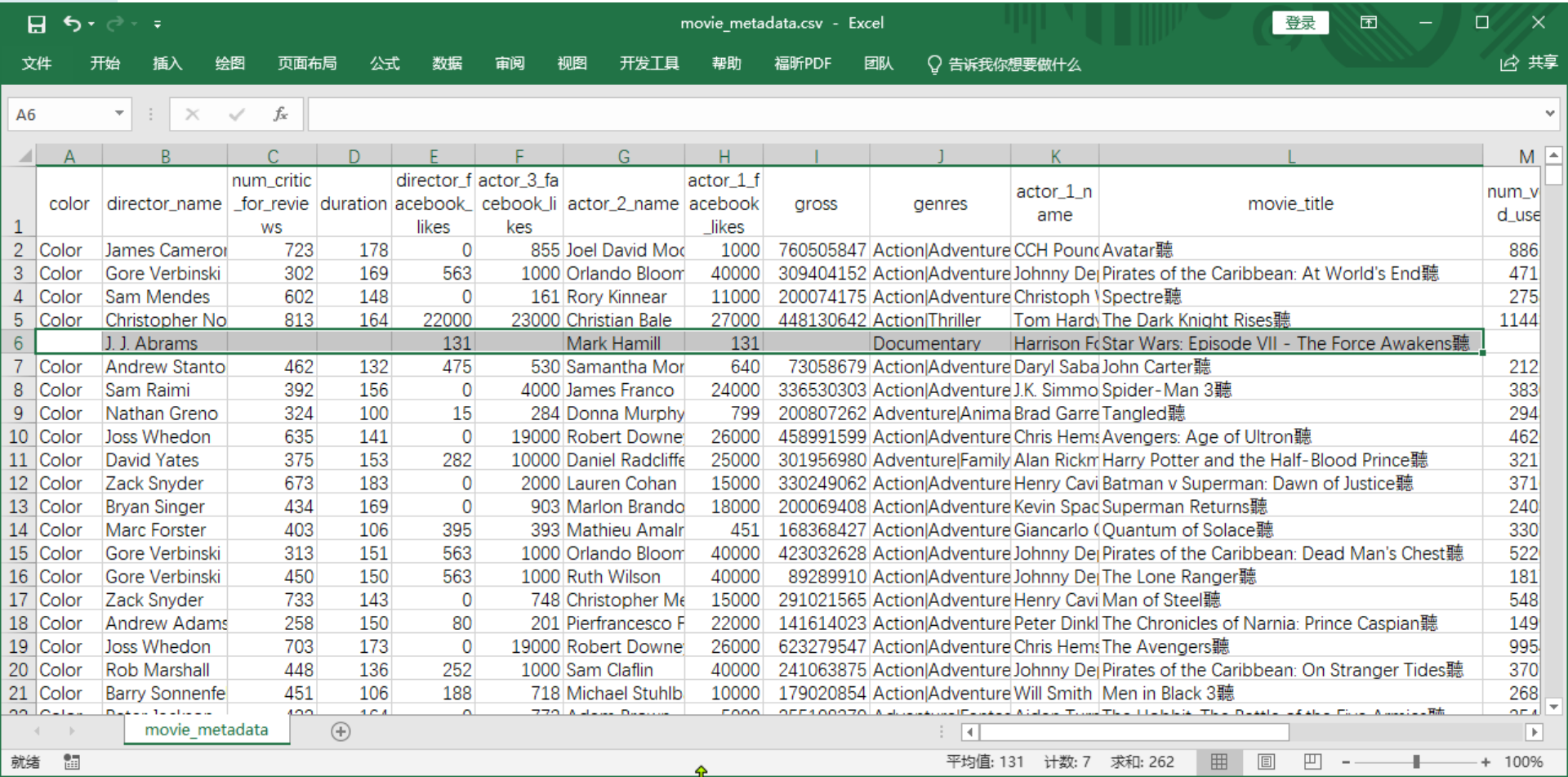
```
drop_duplicates(self, keep='first', inplace=False) -> 'Series | None'
    Return Series with duplicate values removed.

    Parameters
    ----------
    keep : {'first', 'last', False}, default 'first'
        Method to handle dropping duplicates:
        - 'first' : Drop duplicates except for the first occurrence.
        - 'last' : Drop duplicates except for the last occurrence.
        - False : Drop all duplicates.
    inplace : bool, default ``False``
        If ``True``, performs operation inplace and returns None.
```

8.9 数据清洗基本流程及示例

- Pandas 提供功能强大的类库，不管数据处于什么状态，它可以帮助我们通过清洗数据，最后得到清晰明了的数据。
- Pandas数据清洗基本流程如下：
  - 准备工作（导入pandas包、准备好要清洗的数据等）
  - 检查数据
  - 处理缺失数据（添加默认值、删除不完整的行或列等）
  - 删除重复数据
  - 规范化数据类型
  - 必要的转换（错别字、英文单词时大小写的不统一、输入了额外的空格等）
  - 重命名列名
  - 保存结果（保存到文件或数据库）

8.9.1 数据清洗示例之一

- 本示例所使用的数据集：movie_metadata.csv 包含了电影的很多信息，包括影名、演员、导演、预算、总收入，以及 IMDB 评分和上映时间。



movie_metadata.csv - Excel

	A	B	C	D	E	F	G	H	I	J	K	L	M
1	color	director_name	num_critic_for_reviews	duration	director_facebook_likes	actor_3_facebook_likes	actor_2_name	actor_1_facebook_likes	gross	genres	actor_1_name	movie_title	num_vd_use
2	Color	James Cameron	723	178	0	855	Joel David Moo	1000	760505847	Action|Adventure	CCH Pounc	Avatar聽	886
3	Color	Gore Verbinski	302	169	563	1000	Orlando Bloom	40000	309404152	Action|Adventure	Johnny De	Pirates of the Caribbean: At World's End聽	471
4	Color	Sam Mendes	602	148	0	161	Rory Kinnear	11000	200074175	Action|Adventure	Christoph	Spectre聽	275
5	Color	Christopher No	813	164	22000	23000	Christian Bale	27000	448130642	Action|Thriller	Tom Hardy	The Dark Knight Rises聽	1144
6		J. J. Abrams			131		Mark Hamill	131		Documentary	Harrison Fo	Star Wars: Episode VII - The Force Awakens聽	
7	Color	Andrew Stanto	462	132	475	530	Samantha Mor	640	73058679	Action|Adventure	Daryl Saba	John Carter聽	212
8	Color	Sam Raimi	392	156	0	4000	James Franco	24000	336530303	Action|Adventure	J.K. Simmo	Spider-Man 3聽	383
9	Color	Nathan Greno	324	100	15	284	Donna Murphy	799	200807262	Adventure|Anima	Brad Garre	Tangled聽	294
10	Color	Joss Whedon	635	141	0	19000	Robert Downe	26000	458991599	Action|Adventure	Chris Hems	Avengers: Age of Ultron聽	462
11	Color	David Yates	375	153	282	10000	Daniel Radcliffe	25000	301956980	Adventure|Family	Alan Rickm	Harry Potter and the Half-Blood Prince聽	321
12	Color	Zack Snyder	673	183	0	2000	Lauren Cohan	15000	330249062	Action|Adventure	Henry Cavi	Batman v Superman: Dawn of Justice聽	371
13	Color	Bryan Singer	434	169	0	903	Marlon Brando	18000	200069408	Action|Adventure	Kevin Spac	Superman Returns聽	240
14	Color	Marc Forster	403	106	395	393	Mathieu Amalr	451	168368427	Action|Adventure	Giancarlo (	Quantum of Solace聽	330
15	Color	Gore Verbinski	313	151	563	1000	Orlando Bloom	40000	423032628	Action|Adventure	Johnny De	Pirates of the Caribbean: Dead Man's Chest聽	522
16	Color	Gore Verbinski	450	150	563	1000	Ruth Wilson	40000	89289910	Action|Adventure	Johnny De	The Lone Ranger聽	181
17	Color	Zack Snyder	733	143	0	748	Christopher Me	15000	291021565	Action|Adventure	Henry Cavi	Man of Steel聽	548
18	Color	Andrew Adams	258	150	80	201	Pierfrancesco F	22000	141614023	Action|Adventure	Peter Dinkl	The Chronicles of Narnia: Prince Caspian聽	149
19	Color	Joss Whedon	703	173	0	19000	Robert Downe	26000	623279547	Action|Adventure	Chris Hems	The Avengers聽	995
20	Color	Rob Marshall	448	136	252	1000	Sam Claflin	40000	241063875	Action|Adventure	Johnny De	Pirates of the Caribbean: On Stranger Tides聽	370
21	Color	Barry Sonnenfe	451	106	188	718	Michael Stuhlb	10000	179020854	Action|Adventure	Will Smith	Men in Black 3聽	268

movie_metadata

就绪 平均值: 131 计数: 7 求和: 262 100%

8.9.1 数据清洗示例之一

- 准备工作（安装并导入pandas包、下载数据集、加载数据集）
- 检查数据

```
# 加载数据集
df = pd.read_csv('movie_metadata.csv')
print(df.head())
```

	color	director_name	...	aspect_ratio	movie_facebook_likes
0	Color	James Cameron	...	1.78	33000
1	Color	Gore Verbinski	...	2.35	0
2	Color	Sam Mendes	...	2.35	85000
3	Color	Christopher Nolan	...	2.35	164000
4	NaN	J. J. Abrams	...	NaN	0

用Pandas 的方法查看数据，也可以通过传统的 Excel 程序查看数据。此时，我们可以开始记录数据上的问题，然后想相应办法解决问题。

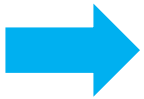
8.9.1 数据清洗示例之一

- 处理缺失数据：添加默认值

应该去掉那些不友好的 NaN 值。但是，应该用什么值替换呢？检查一下“country”列，这一列非常简单，然而有一些电影没有提供地区，所以有些数据的值是NaN。在本案例中，我们推断地区并不是很重要，所以可以使用“”空字符串或其他默认值（例如None Given）。

```
# 空值处理：添加默认值
print(df.head().country)
df.country = df.country.fillna('')
# df.country = df.country.fillna('None Given')
print(df.head().country)
```

```
0    USA
1    USA
2     UK
3    USA
4    NaN
Name: country, dtype: object
```



```
0    USA
1    USA
2     UK
3    USA
4
Name: country, dtype: object
```

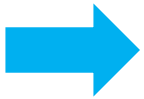
8.9.1 数据清洗示例之一

- 处理缺失数据：添加默认值

有些数值类型的数据可以用平均值代替，比如，电影的时长。这并不是最优解，但这个持续时间是根据其他数据估算出来的。这样就不会因为像 0 或者 NaN 这样的值在我们分析的时候而抛错。

```
# 用平均值(取整)填充空值
print(df.head(7).duration)
df.duration = df.duration.fillna(int(df.duration.mean()))
print(df.head(7).duration)
```

```
0    178.0
1    169.0
2    148.0
3    164.0
4      NaN
5    132.0
6    156.0
Name: duration, dtype: float64
```



```
0    178.0
1    169.0
2    148.0
3    164.0
4    107.0
5    132.0
6    156.0
Name: duration, dtype: float64
```

8.9.1 数据清洗示例之一

- 处理缺失数据：删除不完整的行、列

```
print(df.shape)
# df.dropna(how='all', inplace=True) # 删除一整行的值都为 nan 的行

# 删除非空值少于5 个的数据行
df.dropna(thresh=5, inplace=True)

# 删除电影上映时间为nan的数据行
# subset 参数允许我们选择想要检查的列。如果是多个列，可使用列名的 list 作为参数。
df.dropna(subset=['title_year'], inplace=True)
print(df.shape)
```

参数axis =0(默认)—操作行， axis=1—操作列

(5043, 28) → (4935, 28)

8.9.1 数据清洗示例之一

- 删除重复数据

  影名应该是唯一的，因此可以根据movie_title列去重。

```
print(df.shape)
df.drop_duplicates(subset=['movie_title'], inplace=True)  # 根据movie_title列去重
print(df.shape)
```

(4935, 28) → (4811, 28)

- 规范化数据类型

  当读取 csv 中的title_year列的整数数字时，由于该列存在空值，因此该列类型被Pandas识别为浮点类型。可以在删除空值后，将该列转换为整型。

```
print(df['title_year'].dtype)
df['title_year'] = df['title_year'].astype(np.int64)  # title_year列转换为int64
print(df['title_year'].dtype)
```

float64 → int64

8.9.1 数据清洗示例之一

- 必要的变换

  人工录入的数据可能，可能存在单位不统一、错别字、英文单词时大小写的不统一、输入了额外的空格等情形，都需要进行一些必要的变换。

```
# 删除首尾空格（包括HTML转义字符 ；表示non-breaking space，unicode编码为u'\xa0'）
print(df['movie_title'][0], len(df['movie_title'][0])) # 输出：AVATAR 7
print(df['movie_title'][0].endswith('\xa0')) # 输出：True
df['movie_title'] = df['movie_title'].str.strip()
print(df['movie_title'][0], len(df['movie_title'][0])) # 输出：AVATAR 6
print(df['movie_title'][0].endswith('\xa0')) # 输出：False

# 将数据中所有的 movie_title 列改成大写
print(df.movie_title.head(2))
# movie_title 列改成大写
df['movie_title'] = df['movie_title'].str.upper()
print(df.movie_title.head(2))
```

```
0                                      Avatar
1    Pirates of the Caribbean: At World's End
Name: movie_title, dtype: object
```

```
0                                      AVATAR
1    PIRATES OF THE CARIBBEAN: AT WORLD'S END
Name: movie_title, dtype: object
```

```
>>> '\u00a0'
'\xa0'
>>> '\u00a0'=='\xa0'
True
```

```
>>> ord('\u00a0')
160
>>> ord('\xa0')
160
```

8.9.1 数据清洗示例之一

- 重命名列

  某些数据列可能是由计算机生成的，因此列名有可能也是计算机按照一定计算规律生成的。这些列名对于我们阅读可能不够友好，因此需要重命名。

```
# 重命名列
print('重命名列----------')
df.rename(
    columns={'title_year': 'release_date',
             'movie_facebook_likes': 'facebook_likes'},
    inplace=True)
print(df[['release_date', 'facebook_likes']].head(2))
```

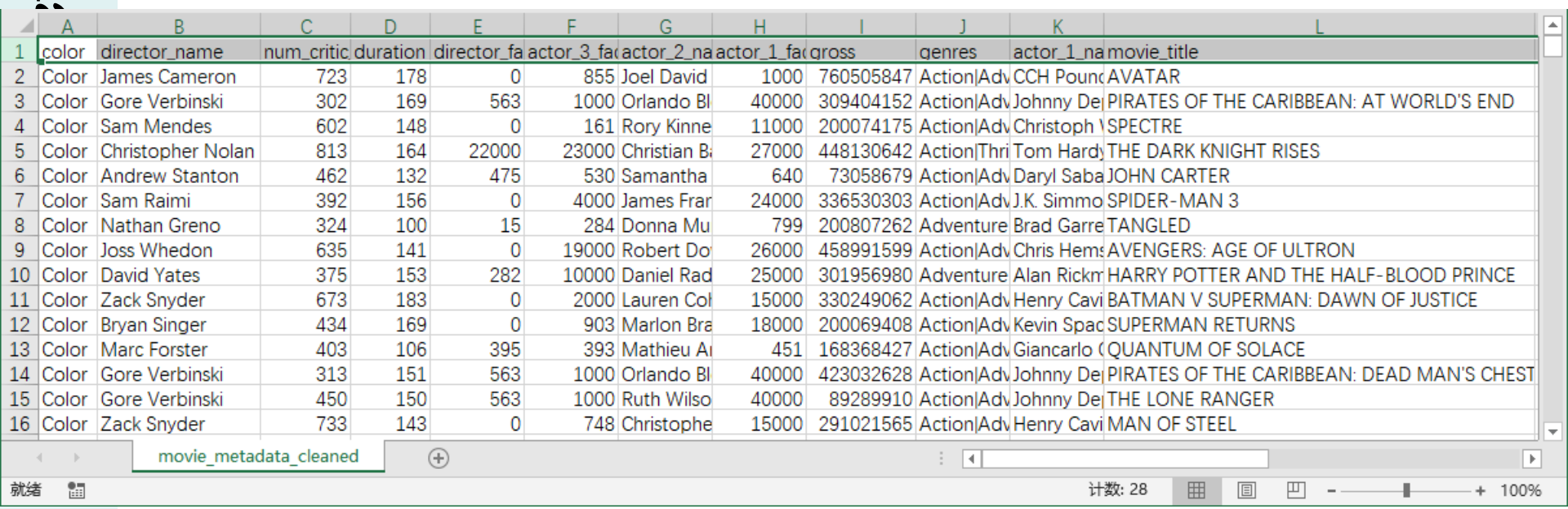
```
   release_date  facebook_likes
0          2009           33000
1          2007               0
```

8.9.1 数据清洗示例之一

- 保存结果

  数据清洗后，通常要保存起来，以便后续其它程序的处理。本例用pd.to_csv方法将结果保存为csv文件。

```
# 保存结果
df.to_csv('movie_metadata_cleaned.csv', header=True, index=False, encoding='utf-8')
```



	A	B	C	D	E	F	G	H	I	J	K	L
1	color	director_name	num_critic	duration	director_fa	actor_3_fa	actor_2_na	actor_1_fa	gross	genres	actor_1_na	movie_title
2	Color	James Cameron	723	178	0	855	Joel David	1000	760505847	Action|Adv	CCH Pounc	AVATAR
3	Color	Gore Verbinski	302	169	563	1000	Orlando Bl	40000	309404152	Action|Adv	Johnny De	PIRATES OF THE CARIBBEAN: AT WORLD'S END
4	Color	Sam Mendes	602	148	0	161	Rory Kinne	11000	200074175	Action|Adv	Christoph	SPECTRE
5	Color	Christopher Nolan	813	164	22000	23000	Christian Ba	27000	448130642	Action|Thri	Tom Hardy	THE DARK KNIGHT RISES
6	Color	Andrew Stanton	462	132	475	530	Samantha	640	73058679	Action|Adv	Daryl Saba	JOHN CARTER
7	Color	Sam Raimi	392	156	0	4000	James Frar	24000	336530303	Action|Adv	J.K. Simmo	SPIDER-MAN 3
8	Color	Nathan Greno	324	100	15	284	Donna Mu	799	200807262	Adventure	Brad Garre	TANGLED
9	Color	Joss Whedon	635	141	0	19000	Robert Do	26000	458991599	Action|Adv	Chris Hems	AVENGERS: AGE OF ULTRON
10	Color	David Yates	375	153	282	10000	Daniel Rad	25000	301956980	Adventure	Alan Rickm	HARRY POTTER AND THE HALF-BLOOD PRINCE
11	Color	Zack Snyder	673	183	0	2000	Lauren Col	15000	330249062	Action|Adv	Henry Cavi	BATMAN V SUPERMAN: DAWN OF JUSTICE
12	Color	Bryan Singer	434	169	0	903	Marlon Bra	18000	200069408	Action|Adv	Kevin Spac	SUPERMAN RETURNS
13	Color	Marc Forster	403	106	395	393	Mathieu Ar	451	168368427	Action|Adv	Giancarlo (	QUANTUM OF SOLACE
14	Color	Gore Verbinski	313	151	563	1000	Orlando Bl	40000	423032628	Action|Adv	Johnny De	PIRATES OF THE CARIBBEAN: DEAD MAN'S CHEST
15	Color	Gore Verbinski	450	150	563	1000	Ruth Wilso	40000	89289910	Action|Adv	Johnny De	THE LONE RANGER
16	Color	Zack Snyder	733	143	0	748	Christophe	15000	291021565	Action|Adv	Henry Cavi	MAN OF STEEL

movie_metadata_cleaned

就绪 计数: 28 100%

8.9.2 数据清洗示例之二

- 数据集patient_heart_rate.csv是描述不同个体在不同时间的心跳情况，包括患者姓名、年龄、体重、性别和不同时间段的心率。该数据集很小，包括空白行在内只有11行、10列数据。

	A	B	C	D	E	F	G	H	I	J	K
1	1	Mickéy Mousé	56	70kgs	72	69	71	-	-	-	
2	2	Donald Duck	34	154.89lbs	-	-	-	85	84	76	
3	3	Mini Mouse	16		-	-	-	65	69	72	
4	4	Scrooge McDuck		78kgs	78	79	72	-	-	-	
5	5	Pink Panther	54	198.658lbs	-	-	-	69		75	
6	6	Huey McDuck	52	189lbs	-	-	-	68	75	72	
7	7	Dewey McDuck	19	56kgs	-	-	-	71	78	75	
8	8	Scööpy Doo	32	78kgs	78	76	75	-	-	-	
9											
10	9	Huey McDuck	52	189lbs	-	-	-	68	75	72	
11	10	Louie McDuck	12	45kgs	-	-	-	92	95	87	
12											
13											

8.9.2 数据清洗示例之二

- 准备工作（安装并导入pandas包、下载数据集、加载数据集）
- 检查数据

```
# 加载数据集
df = pd.read_csv('patient_heart_rate.csv')
print(df)
```

```
     1     Mickéy Mousé    56        70kgs   72   69   71     -   -.1   -.2
0  2.0      Donald Duck  34.0   154.891bs    -    -    -   85    84    76
1  3.0       Mini Mouse  16.0          NaN   -    -    -   65    69    72
2  4.0  Scrooge McDuck    NaN        78kgs   78   79   72    -     -     -
3  5.0     Pink Panther  54.0  198.6581bs    -    -    -   69   NaN    75
4  6.0      Huey McDuck  52.0        189lbs   -    -    -   68    75    72
5  7.0     Dewey McDuck  19.0         56kgs   -    -    -   71    78    75
6  8.0       Scööpy Doo  32.0         78kgs   78   76   75    -     -     -
7  NaN              NaN   NaN           NaN  NaN  NaN  NaN  NaN   NaN   NaN
8  9.0      Huey McDuck  52.0        189lbs   -    -    -   68    75    72
9 10.0     Louie McDuck  12.0         45kgs   -    -    -   92    95    87
```

8.9.2 数据清洗示例之二

- 准备工作（安装并导入pandas包、下载数据集、加载数据集）
- 检查数据

```
# 加载数据集
df = pd.read_csv('patient_heart_rate.csv', header=None)
print(df)
```

```
      0                 1     2            3    4    4    7    8    9
0   1.0     Mickéy Mousé  56.0        70kgs   72   69   71    -    -    -
1   2.0      Donald Duck  34.0   154.891bs     -    -    -   85   84   76
2   3.0       Mini Mouse  16.0          NaN    -    -    -   65   69   72
3   4.0  Scrooge McDuck    NaN        78kgs   78   79   72    -    -    -
4   5.0     Pink Panther  54.0  198.6581bs     -    -    -   69  NaN   75
5   6.0      Huey McDuck  52.0      1891bs     -    -    -   68   75   72
6   7.0     Dewey McDuck  19.0        56kgs    -    -    -   71   78   75
7   8.0      Scööpy Doo   32.0        78kgs   78   76   75    -    -    -
8   NaN              NaN   NaN          NaN  NaN  NaN  NaN  NaN  NaN  NaN
9   9.0      Huey McDuck  52.0      1891bs     -    -    -   68   75   72
10  10.0    Louie McDuck  12.0        45kgs    -    -    -   92   95   87
```

8.9.2 数据清洗示例之二

- 分析数据问题
  - 没有列名。
  - 姓名列应该拆分为FisrtName和LastName两列。
  - 体重列数据的单位不统一，包括kgs和lbs两种单位。
  - 存在缺失值，例如分时段心率列和体重列。
  - 存在空行和存在重复数据。
  - 表格记录数据方便，但是不便于数据分析。性别与分时段心率混在一起，实际上应该进行数据融合，也就是水平显示应改为垂直显示。

	1	Mickéy Mousé	56	70kgs	72	69	71	-	-.1	-.2
0	2.0	Donald Duck	34.0	154.89lbs	-	-	-	85	84	76
1	3.0	Mini Mouse	16.0	NaN	-	-	-	65	69	72
2	4.0	Scrooge McDuck	NaN	78kgs	78	79	72	-	-	-
3	5.0	Pink Panther	54.0	198.658lbs	-	-	-	69	NaN	75
4	6.0	Huey McDuck	52.0	189lbs	-	-	-	68	75	72
5	7.0	Dewey McDuck	19.0	56kgs	-	-	-	71	78	75
6	8.0	Scööpy Doo	32.0	78kgs	78	76	75	-	-	-
7	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
8	9.0	Huey McDuck	52.0	189lbs	-	-	-	68	75	72
9	10.0	Louie McDuck	12.0	45kgs	-	-	-	92	95	87

8.9.2 数据清洗示例之二

- 数据目标格式

	1	Mickéy Mousé	56	70kgs	72	69	71	-	-.1	-.2
0	2.0	Donald Duck	34.0	154.891bs	-	-	-	85	84	76
1	3.0	Mini Mouse	16.0	NaN	-	-	-	65	69	72
2	4.0	Scrooge McDuck	NaN	78kgs	78	79	72	-	-	-
3	5.0	Pink Panther	54.0	198.6581bs	-	-	-	69	NaN	75
4	6.0	Huey McDuck	52.0	1891bs	-	-	-	68	75	72
5	7.0	Dewey McDuck	19.0	56kgs	-	-	-	71	78	75
6	8.0	Scööpy Doo	32.0	78kgs	78	76	75	-	-	-
...	...									

	firstname	lastname	age	weight	puls_rate	sex	hour
47	Mini	Mouse	16	71	72	f	12-18
31	Pink	Panther	54	90	69	f	00-06
40	Pink	Panther	54	90	76	f	06-12
49	Pink	Panther	54	90	75	f	12-18
3	Scrooge	McDuck	34	78	78	m	00-06
12	Scrooge	McDuck	34	78	79	m	06-12
21	Scrooge	McDuck	34	78	72	m	12-18
...	...						

usecols=range(1, 10), # 第1列不读取

8.9.2 数据清洗示例之二

- 没有列名：加载数据集时指定列索引

```
names = ['name', 'age', 'weight', 'm0006', 'm0612', 'm1218', 'f0006', 'f0612',
'f1218']
df = pd.read_csv('patient_heart_rate.csv',
                 usecols=range(1, 10), # 第1列不读取
                 names=names) # 数据中不包含列名，指定列索引
print(df)
```

	name	age	weight	m0006	m0612	m1218	f0006	f0612	f1218
0	Mickéy Mousé	56.0	70kgs	72	69	71	-	-	-
1	Donald Duck	34.0	154.89lbs	-	-	-	85	84	76
2	Mini Mouse	16.0	NaN	-	-	-	65	69	72
3	Scrooge McDuck	NaN	78kgs	78	79	72	-	-	-
4	Pink Panther	54.0	198.6581bs	-	-	-	69	NaN	75
5	Huey McDuck	52.0	189lbs	-	-	-	68	75	72
6	Dewey McDuck	19.0	56kgs	-	-	-	71	78	75
7	Scööpy Doo	32.0	78kgs	78	76	75	-	-	-
8	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
9	Huey McDuck	52.0	189lbs	-	-	-	68	75	72
10	Louie McDuck	12.0	45kgs	-	-	-	92	95	87

8.9.2 数据清洗示例之二

- 删除空白行和重复行

	name	age	weight	m0006	m0612	m1218	f0006	f0612	f1218
0	Mickéy Mousé	56.0	70kgs	72	69	71	-	-	-
1	Donald Duck	34.0	154.89lbs	-	-	-	85	84	76
2	Mini Mouse	16.0	NaN	-	-	-	65	69	72
3	Scrooge McDuck	NaN	78kgs	78	79	72	-	-	-
4	Pink Panther	54.0	198.658lbs	-	-	-	69	NaN	75
5	Huey McDuck	52.0	189lbs	-	-	-	68	75	72
6	Dewey McDuck	19.0	56kgs	-	-	-	71	78	75
7	Scööpy Doo	32.0	78kgs	78	76	75	-	-	-
8	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
9	Huey McDuck	52.0	189lbs	-	-	-	68	75	72
10	Louie McDuck	12.0	45kgs	-	-	-	92	95	87

8.9.2 数据清洗示例之二

- 删除空白行和重复行

```
df.dropna(how='all', inplace=True)  # 删除空白行
df.drop_duplicates(inplace=True)  # 去重
print(df)
```

	name	age	weight	m0006	m0612	m1218	f0006	f0612	f1218
0	Mickéy Mousé	56.0	70kgs	72	69	71	-	-	-
1	Donald Duck	34.0	154.89lbs	-	-	-	85	84	76
2	Mini Mouse	16.0	NaN	-	-	-	65	69	72
3	Scrooge McDuck	NaN	78kgs	78	79	72	-	-	-
4	Pink Panther	54.0	198.658lbs	-	-	-	69	NaN	75
5	Huey McDuck	52.0	189lbs	-	-	-	68	75	72
6	Dewey McDuck	19.0	56kgs	-	-	-	71	78	75
7	Scööpy Doo	32.0	78kgs	78	76	75	-	-	-
10	Louie McDuck	12.0	45kgs	-	-	-	92	95	87

8.9.2 数据清洗示例之二

- 拆分列
  - Name 列包含了两个参数 firstname 和 lastname。为了达到数据整洁目的，决定将 name 列拆分成 firstname 和 lastname。
  - 可以使用 str.split(expand=True), 将列拆成两列，再将原来的 name 列删除

```
# 拆分姓名，删除原数据列
df[['firstname', 'lastname']] = df['name'].str.split(expand=True)  # name列拆分为两列
df.drop('name', axis=1, inplace=True)  # 删除name列
#并调整列序
s = df.lastname  # 暂存lastname列
del df['lastname']  # 从df中移除lastname列
df.insert(0, column='lastname', value=s) # lastname列作为第0列插入
s = df.firstname  # 暂存firstname列
del df['firstname']  # 从df中移除firstname列
df.insert(0, column='firstname', value=s) # firstname列作为第0列插入
print(df)
```

```
tmp_series = df['name'].str.split()  # 返回一个Series，元素是字符串列表，例如['Mickéy','Mousé']
tmp_df = df['name'].str.split(expand=True)  # 返回一个DataFrame,包含两列
```

8.9.2 数据清洗示例之二

- 拆分列
  - Name 列包含了两个参数 firstname 和 lastname。为了达到数据整洁目的，决定将 name 列拆分成 firstname 和 lastname。
  - 可以使用 str.split(expand=True), 将列拆成两列，再将原来的 name 列删除

	firstname	lastname	age	weight	m0006	m0612	m1218	f0006	f0612	f1218
0	Mickéy	Mousé	56.0	70kgs	72	69	71	-	-	-
1	Donald	Duck	34.0	154.89lbs	-	-	-	85	84	76
2	Mini	Mouse	16.0	NaN	-	-	-	65	69	72
3	Scrooge	McDuck	NaN	78kgs	78	79	72	-	-	-
4	Pink	Panther	54.0	198.658lbs	-	-	-	69	NaN	75
5	Huey	McDuck	52.0	189lbs	-	-	-	68	75	72
6	Dewey	McDuck	19.0	56kgs	-	-	-	71	78	75
7	Scööpy	Doo	32.0	78kgs	78	76	75	-	-	-
10	Louie	McDuck	12.0	45kgs	-	-	-	92	95	87

8.9.2 数据清洗示例之二

- 列数据的单位不统一

  Weight 列的单位不统一，有的单位是 kgs，有的单位是 lbs。

```
# 获取 weight 列中单位为 lbs 的数据行(布尔值)
rows_with_lbs = df['weight'].str.endswith('lbs')  # 返回 bool类型Series, Nan值endswith测试后仍然为Nan
print(rows_with_lbs)

rows_with_lbs = rows_with_lbs.fillna(False)  # Nan值填充为False
print(rows_with_lbs)
```

```
0     False
1      True
2       NaN
3     False
4      True
5      True
6     False
7     False
10    False
Name: weight, dtype: object
```

fillna

```
0     False
1      True
2     False
3     False
4      True
5      True
6     False
7     False
10    False
Name: weight, dtype: bool
```

8.9.2 数据清洗示例之二

- 列数据的单位不统一

  Weight 列的单位不统一, 有的单位是 kgs，有的单位是 lbs。

```
print(df[rows_with_lbs]) # 显示从 df 中选出 weight 列的单位是 lbs 的行
```

	firstname	lastname	age	weight	m0006	m0612	m1218	f0006	f0612	f1218
1	Donald	Duck	34.0	154.891bs	-	-	-	85	84	76
4	Pink	Panther	54.0	198.6581bs	-	-	-	69	NaN	75
5	Huey	McDuck	52.0	1891bs	-	-	-	68	75	72

8.9.2 数据清洗示例之二

- 列数据的单位不统一

  统一单位，将单位是 lbs 的数据转换成 kgs，并移除单位kgs。

```
for index, lbs_row in df[rows_with_lbs].iterrows():  # 遍历各行对应的元组(index，Series)
    weight = int(float(lbs_row['weight'][:-3]) / 2.2)  # 单位换算
    df.at[index, 'weight'] = f'{weight}kgs'  # 修改元素值

df['weight'] = df['weight'].str[0:-3]  # 移除单位kgs
print(df)
```

	firstname	lastname	age	weight	m0006	m0612	m1218	f0006	f0612	f1218
0	Mickéy	Mousé	56.0	70	72	69	71	-	-	-
1	Donald	Duck	34.0	70	-	-	-	85	84	76
2	Mini	Mouse	16.0	NaN	-	-	-	65	69	72
3	Scrooge	McDuck	NaN	78	78	79	72	-	-	-
4	Pink	Panther	54.0	90	-	-	-	69	NaN	75
5	Huey	McDuck	52.0	85	-	-	-	68	75	72
6	Dewey	McDuck	19.0	56	-	-	-	71	78	75
7	Scööpy	Doo	32.0	78	78	76	75	-	-	-
10	Louie	McDuck	12.0	45	-	-	-	92	95	87

8.9.2 数据清洗示例之二

- 列数据的单位不统一

  统一单位，将单位是 lbs 的数据转换成 kgs，并移除单位kgs。

或者直接遍历各行：

```
for index, row in df.iterrows(): # 遍历各行对应的元组(index，Series)
    if pd.notna(row['weight']) and row['weight'].endswith('lbs'):
        df.at[index, 'weight']=f"{int(float(row['weight'][0:-3])/2.2)}kgs"

df['weight'] = df['weight'].str[0:-3]  # 移除单位kgs
```

或者：

```
for i in range(len(df)):
    if pd.notna(df.at[i,'weight']) and df.at[i, 'weight'].endswith('lbs'):
        df.at[i, 'weight'] = f"{int(float(df.at[i,'weight'][0:-3])/2.2)}kgs"

df['weight'] = df['weight'].str[0:-3]  # 移除单位kgs
```

8.9.2 数据清洗示例之二

- 有些列头应该是数据，而不应该是列名参数

有一些列头是由性别和时间范围组成的，这些数据有可能是在处理收集的过程中进行了行列转换，或者收集器的固定命名规则。这些值应该被分解为性别（m, f），小时单位的时间范围（00-06，06-12，12-18）。

	firstname	lastname	age	weight	sex_hour	puls_rate
6	Dewey	McDuck	19.0	56	m0006	-
15	Dewey	McDuck	19.0	56	m0612	-
24	Dewey	McDuck	19.0	56	m1218	-
33	Dewey	McDuck	19.0	56	f0006	71
42	Dewey	McDuck	19.0	56	f0612	78
51	Dewey	McDuck	19.0	56	f1218	75
1	Donald	Duck	34.0	70	m0006	-
10	Donald	Duck	34.0	70	m0612	-
19	Donald	Duck	34.0	70	m1218	-
28	Donald	Duck	34.0	70	f0006	85
37	Donald	Duck	34.0	70	f0612	84
46	Donald	Duck	34.0	70	f1218	76

- puls_rate是数据行转换为列后的新列标签
- sex_hour是列索引标签转换为列后的新列标签

8.9.2 数据清洗示例之二

```
# 数据融合
id_vars = ['firstname', 'lastname', 'age', 'weight']
df = pd.melt(df,
             id_vars=id_vars, # 标识变量列
             var_name='sex_hour', # 度量变量列命名
             value_name='puls_rate') # 值列命名
df.sort_values(by=id_vars, inplace=True) # 数据框按指定列的值排序
print(df.head(12))
```

	firstname	lastname	age	weight	sex_hour	puls_rate
6	Dewey	McDuck	19.0	56	m0006	-
15	Dewey	McDuck	19.0	56	m0612	-
24	Dewey	McDuck	19.0	56	m1218	-
33	Dewey	McDuck	19.0	56	f0006	71
42	Dewey	McDuck	19.0	56	f0612	78
51	Dewey	McDuck	19.0	56	f1218	75
1	Donald	Duck	34.0	70	m0006	-
10	Donald	Duck	34.0	70	m0612	-
19	Donald	Duck	34.0	70	m1218	-
28	Donald	Duck	34.0	70	f0006	85
37	Donald	Duck	34.0	70	f0612	84
46	Donald	Duck	34.0	70	f1218	76

- sex_hour是列索引标签转换为列后的新列标签
- puls_rate是数据行转换为列后的新列标签

8.9.2 数据清洗示例之二

```
# 删除没有心率的数据
row_with_dashes = (df['puls_rate'] == '-')  # 注意序列中可能含有nan值
row_with_dashes = row_with_dashes.fillna(False)  # 将Nan值设置为false

df.drop(index=df[row_with_dashes].index, inplace=True)  # 按行标签删除行
print(df.head(12))
```

	firstname	lastname	age	weight	sex_hour	puls_rate
33	Dewey	McDuck	19.0	56	f0006	71
42	Dewey	McDuck	19.0	56	f0612	78
51	Dewey	McDuck	19.0	56	f1218	75
28	Donald	Duck	34.0	70	f0006	85
37	Donald	Duck	34.0	70	f0612	84
46	Donald	Duck	34.0	70	f1218	76
32	Huey	McDuck	52.0	85	f0006	68
41	Huey	McDuck	52.0	85	f0612	75
50	Huey	McDuck	52.0	85	f1218	72
35	Louie	McDuck	12.0	45	f0006	92
44	Louie	McDuck	12.0	45	f0612	95
53	Louie	McDuck	12.0	45	f1218	87

8.9.2 数据清洗示例之二

- 有些列头应该是数据，而不应该是列名参数

有一些列头是有性别和时间范围组成的，这些数据有可能是在处理收集的过程中进行了行列转换，或者收集器的固定命名规则。这些值应该被分解为性别（m,f），小时单位的时间范围（00-06，06-12，12-18）。

```
# 将sex_hour列拆分成sex和hour两列
df['sex'] = df['sex_hour'].str[0]  # 新增sex列

df['hour'] = df['sex_hour'].str[1:3] + "-" + df['sex_hour'].str[3:5]  # 新增hour列

del df['sex_hour']  # 删除sex_hour列
print(df.head())
```

	firstname	lastname	age	weight	puls_rate	sex	hour
33	Dewey	McDuck	19.0	56	71	f	00-06
42	Dewey	McDuck	19.0	56	78	f	06-12
51	Dewey	McDuck	19.0	56	75	f	12-18
28	Donald	Duck	34.0	70	85	f	00-06
37	Donald	Duck	34.0	70	84	f	06-12

	firstname	lastname	age	weight	sex_hour	puls_rate
33	Dewey	McDuck	19.0	56	f0006	71
42	Dewey	McDuck	19.0	56	f0612	78

8.9.2 数据清洗示例之二

- 规范数据类型

想将age、weight和puls_rate三列类型设置为整型。由于NaN值不能转化为整型， 因此，这些可能含有空值的列不能直接转化为整型，而是先转化为浮点型，然后填充空值，最后再转化为整型。

```
# 转化为浮点型
df['age'] = df['age'].astype(float, errors='ignore')
df['weight'] = df['weight'].astype(float, errors='ignore')
df['puls_rate'] = df['puls_rate'].astype(float, errors='ignore')
print(df.tail(10))
```

	firstname	lastname	age	weight	puls_rate	sex	hour
47	Mini	Mouse	16.0	NaN	72.0	f	12-18
31	Pink	Panther	54.0	90.0	69.0	f	00-06
40	Pink	Panther	54.0	90.0	NaN	f	06-12
49	Pink	Panther	54.0	90.0	75.0	f	12-18
3	Scrooge	McDuck	NaN	78.0	78.0	m	00-06
12	Scrooge	McDuck	NaN	78.0	79.0	m	06-12
21	Scrooge	McDuck	NaN	78.0	72.0	m	12-18
7	Scöopy	Doo	32.0	78.0	78.0	m	00-06
16	Scöopy	Doo	32.0	78.0	76.0	m	06-12
25	Scöopy	Doo	32.0	78.0	75.0	m	12-18

8.9.2 数据清洗示例之二

- 规范数据类型

想将age、weight和puls_rate三列类型设置为整型。由于NaN值不能转化为整型， 因此，这些可能含有空值的列不能直接转化为整型，而是先转化为浮点型，然后填充空值，最后再转化为整型。

```
# 将age、weight和puls_rate三列分别用平均值填充空值
df.age = df.age.fillna(df.age.mean())
df.weight = df.weight.fillna(df.weight.mean())
df.puls_rate = df.puls_rate.fillna(df.puls_rate.mean())
print(df.tail(10))
```

	firstname	lastname	age	weight	puls_rate	sex	hour
47	Mini	Mouse	16.000	71.5	72.000000	f	12-18
31	Pink	Panther	54.000	90.0	69.000000	f	00-06
40	Pink	Panther	54.000	90.0	76.076923	f	06-12
49	Pink	Panther	54.000	90.0	75.000000	f	12-18
3	Scrooge	McDuck	34.375	78.0	78.000000	m	00-06
12	Scrooge	McDuck	34.375	78.0	79.000000	m	06-12
21	Scrooge	McDuck	34.375	78.0	72.000000	m	12-18
7	Scöopy	Doo	32.000	78.0	78.000000	m	00-06
16	Scöopy	Doo	32.000	78.0	76.000000	m	06-12
25	Scöopy	Doo	32.000	78.0	75.000000	m	12-18

8.9.2 数据清洗示例之二

- 规范数据类型

想将age、weight和puls_rate三列类型设置为整型。由于NaN值不能转化为整型， 因此，这些可能含有空值的列不能直接转化为整型，而是先转化为浮点型，然后填充空值，最后再转化为整型。

```
# 最后转化为整型
df['age'] = df['age'].astype(int)
df['weight'] = df['weight'].astype(int)
df['puls_rate'] = df['puls_rate'].astype(int)
print(df.tail(10))
```

	firstname	lastname	age	weight	puls_rate	sex	hour
47	Mini	Mouse	16	71	72	f	12-18
31	Pink	Panther	54	90	69	f	00-06
40	Pink	Panther	54	90	76	f	06-12
49	Pink	Panther	54	90	75	f	12-18
3	Scrooge	McDuck	34	78	78	m	00-06
12	Scrooge	McDuck	34	78	79	m	06-12
21	Scrooge	McDuck	34	78	72	m	12-18
7	Scöopy	Doo	32	78	78	m	00-06
16	Scöopy	Doo	32	78	76	m	06-12
25	Scöopy	Doo	32	78	75	m	12-18

8.9.2 数据清

- 保存结果

```
# 重新编号行索引，从1开始
df.index = range(1, len(df)+1)
df.index.name = 'id'  # 设置行索引名

df.to_csv('patient_heart_rate_cleaned.csv',
          header=True,
          index=True,
          encoding='utf-8')
```

patient_heart_rate_cleaned.csv

```
id,firstname,lastname,age,weight,puls_rate,sex,hour
1,Dewey,McDuck,19,56,71,f,00-06
2,Dewey,McDuck,19,56,78,f,06-12
3,Dewey,McDuck,19,56,75,f,12-18
4,Donald,Duck,34,70,85,f,00-06
5,Donald,Duck,34,70,84,f,06-12
6,Donald,Duck,34,70,76,f,12-18
7,Huey,McDuck,52,85,68,f,00-06
8,Huey,McDuck,52,85,75,f,06-12
9,Huey,McDuck,52,85,72,f,12-18
10,Louie,McDuck,12,45,92,f,00-06
11,Louie,McDuck,12,45,95,f,06-12
12,Louie,McDuck,12,45,87,f,12-18
13,Mickéy,Mousé,56,70,72,m,00-06
14,Mickéy,Mousé,56,70,69,m,06-12
15,Mickéy,Mousé,56,70,71,m,12-18
16,Mini,Mouse,16,71,65,f,00-06
17,Mini,Mouse,16,71,69,f,06-12
18,Mini,Mouse,16,71,72,f,12-18
19,Pink,Panther,54,90,69,f,00-06
20,Pink,Panther,54,90,76,f,06-12
21,Pink,Panther,54,90,75,f,12-18
22,Scrooge,McDuck,34,78,78,m,00-06
23,Scrooge,McDuck,34,78,79,m,06-12
24,Scrooge,McDuck,34,78,72,m,12-18
25,Scöopy,Doo,32,78,78,m,00-06
26,Scöopy,Doo,32,78,76,m,06-12
27,Scöopy,Doo,32,78,75,m,12-18
```

谢谢大家！