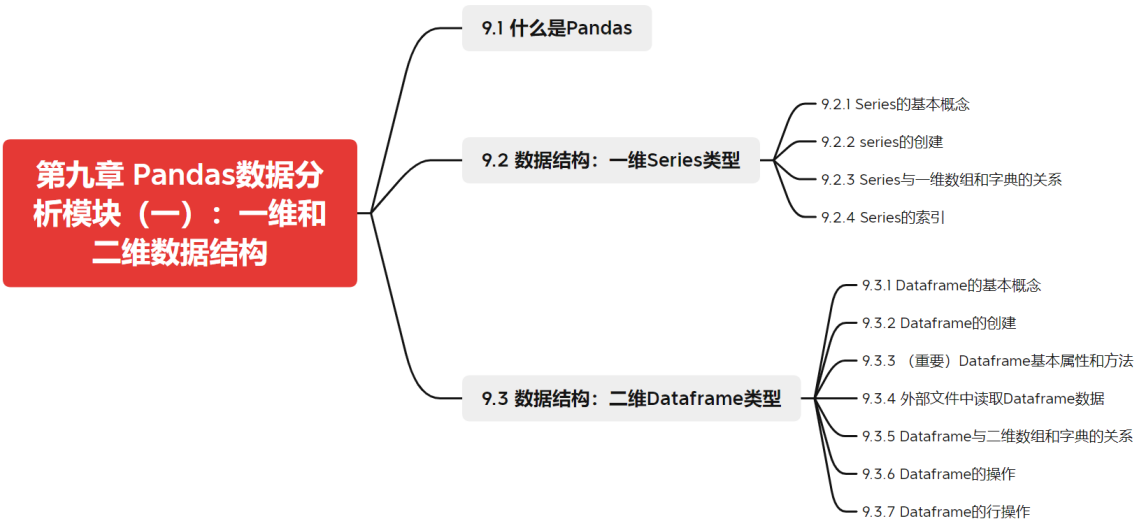
第9章 Pandas数据分析模块（一）：一维和二维数据结构

思维导图



9.1 什么是Pandas

任务：为何需要Pandas模块

> 知识点：
>
> - NumPy模块的核心数据结构是`ndarray`，它为Python提供了**多维数组的存储和处理方法。**
> - 多维是通过轴(`axes`)对数据展开操作。每一个维度相应地被称为一个轴（`axes`），`ndim`是轴的总量，`shape`描述了每个轴上的数据规模。
> - NumPy模块为多维数组提供高效的数值计算，例如广播、通用函数、聚合操作、四大索引等技术。

> 需求分析：
>
> - NumPy模块仅适合处理**结构化数据**，即数值类型的，不含缺失值的数据。

> - 在现实应用中，存在大量的**非结构化数据**，比如时间序列、带标签、含有缺失值等数据；此外，需要对**表格类型**数据进行分组、数据透视等加工处理。对于这类问题，NumPy模块并不适合处理。

为了解决上述需求，Pandas模块在NumPy模块的基础上，专门针对**非结构化类型数据**提供高效的数据处理。Pandas是一个强大的分析结构化数据的工具集；它继承了NumPy的优势(提供高性能的矩阵运算)；用于数据挖掘和数据分析，同时也提供数据清洗功能。

任务：Pandas的主要特点

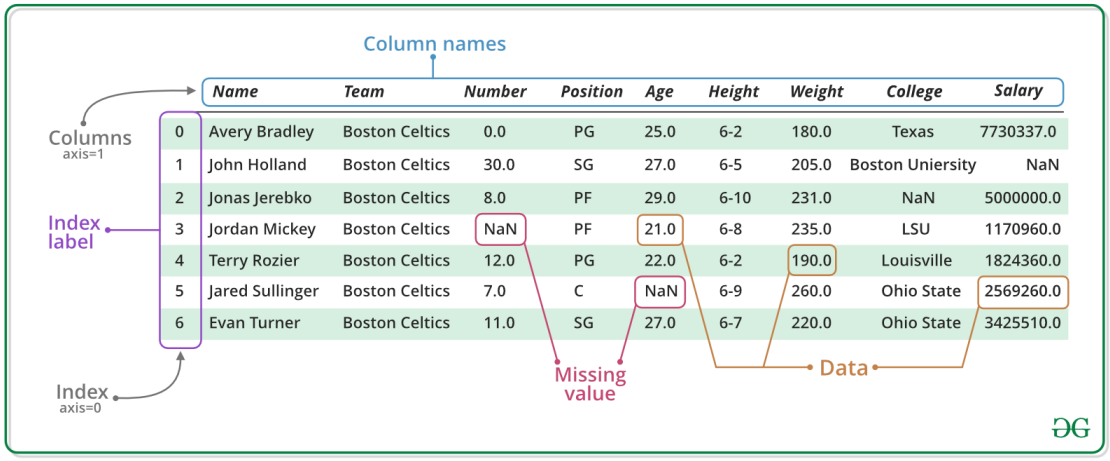
Pandas是一个开放源码的Python库，它使用强大的数据结构提供高性能的数据操作和分析工具。Pandas用于广泛的领域，包括数据分析，人工智能，金融，经济，统计，分析等学术和商业领域。

> 知识点：
>
> - 带标签的数据结构：高效的一维`Series`和二维`DataFrame`数据结构，具有默认和自定义的索引。
> - 多样化数据格式：将数据从**不同文件格式**加载到内存中的数据对象的工具，例如`txt`、`CSV`、`SQL`数据库等。
> - 缺失值：轻松处理**缺失值**数据(以`NaN`表示)以及非浮点数据。
> - 数据重构：按数据**分组**进行聚合和转换，**数据透视**功能。
> - 个性化索引：提供高效的标签切片、花式索引、简单索引和多级索引。
> - 支持**时间序列**数据，数据重采样功能。

任务：Pandas的三大数据结构

> 知识点：
>
> - 系列(`Series`)类型：一维数据结构，它是由一组数据以及与之相关的数据标签(即索引)组成。
> - 数据帧(`DataFrame`)类型：二维数据结构，它是Pandas中的一个**表格型**的数据结构，包含有一组有序的列，每列可以是不同的值类型(数值、字符串、布尔型等)。`DataFrame`即有行索引也有列索引，可以被看做是由`Series`组成的字典。
> - 面板(`Panel`)类型：三维数据结构，用于高维的数据描述。
> - 理解：`Series`是基本数据的容器、`DataFrame`是`Series`的容器，`Panel`是`DataFrame`的容器。

> 知识点：
>
> - `Series`和`DataFrame`是Pandas用的最多的数据结构。
> - `Series`是一维数据结构。例如，下面的`Series`是整数集合。
>
> | | | | | | | | | | |
> |---|---|---|---|---|---|---|---|---|---|
> | 10 | 23 | 56 | 17 | 52 | 61 | 73 | 90 | 26 | 72 |
>
> - `DataFrame`是二维数据结构。例如，表格数据，带标签，异构混合类型。



	Name	Team	Number	Position	Age	Height	Weight	College	Salary
0	Avery Bradley	Boston Celtics	0.0	PG	25.0	6-2	180.0	Texas	7730337.0
1	John Holland	Boston Celtics	30.0	SG	27.0	6-5	205.0	Boston Uniersity	NaN
2	Jonas Jerebko	Boston Celtics	8.0	PF	29.0	6-10	231.0	NaN	5000000.0
3	Jordan Mickey	Boston Celtics	NaN	PF	21.0	6-8	235.0	LSU	1170960.0
4	Terry Rozier	Boston Celtics	12.0	PG	22.0	6-2	190.0	Louisville	1824360.0
5	Jared Sullinger	Boston Celtics	7.0	C	NaN	6-9	260.0	Ohio State	2569260.0
6	Evan Turner	Boston Celtics	11.0	SG	27.0	6-7	220.0	Ohio State	3425510.0

任务：Pandas的导包

Anaconda自带安装Numpy、Pandas、Matplot模块。测试工作环境是否有安装好了Pandas，导入相关包如下：

```
# 领域内惯用的三个模块导包统一简写命名方式
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
print(pd.__version__)
```

> 知识点：
>
> - 建议在Jupyter的第一个`cell`添加三个模块的导包语句。之后的cell，不再需要重复导包。
> - **本章所有的例子，都默认已经导包。**

9.2 数据结构：一维Series类型

9.2.1 Series的基本概念

任务：Series的基本概念

> 知识点：
>
> - `Series`是带索引的一维数据结构，数据类型可以是整数，字符串，浮点数，Python对象等。
> - 类似于字典，`Series`的索引可以是任何的**不可变类型**。
> - 类似于NumPy数组，`Series`数据必须是**相同**类型。

任务：Series的构造函数

`Series`的创建函数描述如下：

```
pandas.Series(data, index, dtype, copy)
```

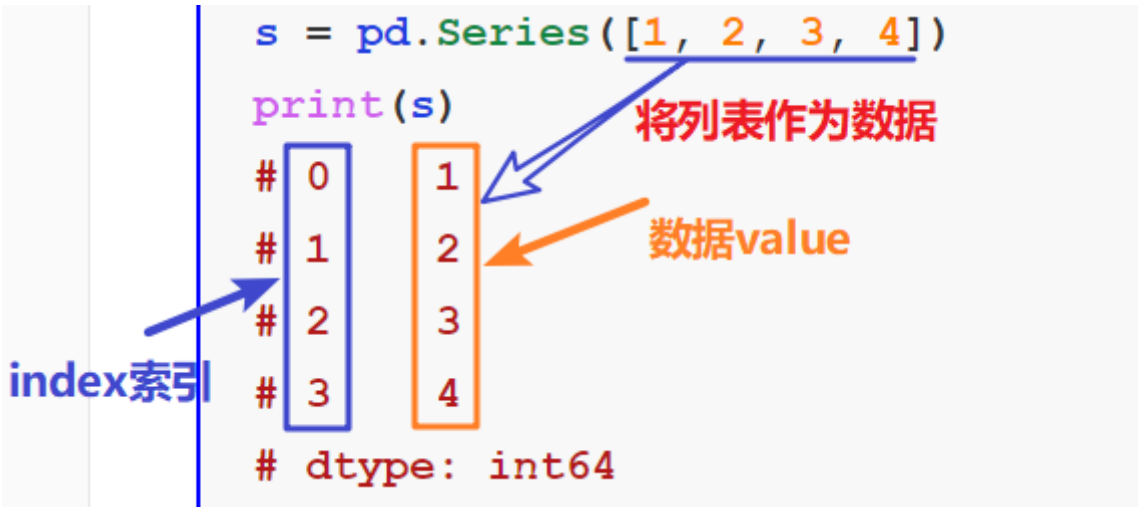
构造函数的参数如下：

- `data`：数据。数组`ndarray`，列表`list`、字典、标量或常量。

- `index`：索引值必须是**唯一**的(不可变类型)，与**数据的长度相同。** 默认`np.arange(n)`如果没有索引被传递。
- `dtype`：`dtype`用于数据类型。如果没有，Pandas会根据数据类型进行推断。
- `copy`：复制数据，默认为`false`。

例子：创建一个`Series`对象。

```
s = pd.Series([1, 2, 3, 4])
print(s)
# 0    1
# 1    2
# 2    3
# 3    4
# dtype: int64
```



> 知识点：
>
> - `Series`的左侧是`index`(索引)。
> - 调用`Series`构造函数时，若不指明`index`，那么Pandas会默认使用从0开始依次递增的数值作为`index`数组。
> - `Series`的右侧是`index`对应的`values`，即封装的数据。

任务：获取标签和数据

> 知识点：
>
> - `index`属性：获取**标签**索引。标签索引类型是`Index`，或它的子类型。
> - `values`属性：获取数据。类型是`ndarray`。

```
s = pd.Series([1, 2, 3, 4], index=['a','b','c','d'])
# 获得标签索引
print(s.index)
#Index(['a', 'b', 'c', 'd'], dtype='object')

# 获得数据，一维数组
print(type(s.values), s.values)
# <class 'numpy.ndarray'> [1 2 3 4]
```

9.2.2 series的创建

任务：创建一个空`Series`

```
s = pd.Series()
print(s)
# Series([], dtype: float64)
```

`Series`有3中创建方式，从**标量**、从**数组**(可以是列表、元组和NumPy数组)和从**字典**。

任务：从标量创建`Series`

> 知识点：
>
> - 如果数据是标量值，则**必须**提供`index`索引参数。
> - 将重复该标量值，以匹配索引的长度。

```
print(pd.Series(5, index=[100, 200, 300]))
# 100    5
# 200    5
# 300    5
# dtype: int64
```

任务：从`ndarray`数组创建`Series`

> 知识点：
>
> - 如果数据是`ndarray`，则传递的索引必须具有相同的长度。
> - 如果没有传递索引值，那么默认索引将是范围(`n`)，其中`n`是数组长度，即`[0,1,…,len(data)-1]`。

```
data = np.array(['a','b','c','d'])
s = pd.Series(data)
print(s)
# 0   a
# 1   b
# 2   c
# 3   d
# dtype: object
```

这里没有传递任何索引，因此默认情况下，它分配了从`0`到`len(data)-1`的索引，即：`0`到`3`。

任务：从字典创建创建`Series`

> 知识点：
>
> - 字典(`dict`)可以作为输入传递。
> - 如果没有指定`index`，将字典的`key`用于构建`Series`的索引。
> - `Series`元素的顺序按传递的字典`key`顺序排列。

```
data = {'a' : 0., 'b' : 1., 'c' : 2.}
s = pd.Series(data)
print(s)
# a 0.0
# b 1.0
# c 2.0
# dtype: float64
```

> 知识点：如果传递`index`参数。根据`index`，
>
> - 将字典中与相匹配的`key`对应的元素取出。
> - 没匹配的`key`对应的元素使用`NaN`(不是数字)填充。
> - `Series`元素的顺序按`index`顺序排列。

```
data = {'a' : 0., 'b' : 1., 'c' : 2.}
s = pd.Series(data,index=['b','c','d','a'])
print(s)
# b 1.0
# c 2.0
# d NaN
# a 0.0
# dtype: float64
```

> 知识点：通过`index`可以对字典数据进行筛选。

```
# 保留index指定的索引的字典元素
print(pd.Series({2:'a', 1:'b', 3:'c'}, index=[3, 2]))
# 3    c
# 2    a
# dtype: object
```

9.2.3 Series与一维数组和字典的关系

任务：Serise是通用的NumPy数组

> 知识点：`Series`和NumPy的一维数组的本质区别在于索引：
>
> - 类似列表，NumPy数组通过**隐式**整数索引获取数值。
> - 类似字典，`Series`使用显示索引`index`与数值关联。
> -
>   - 显式索引让`Series`拥有了更强的数据处理能力。
>   - 索引不仅仅是整数，可以是任意不可变类型(类似于字典的`key`)。

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

# 类似字典，可以通过索引来访问元素。
data['b']# 0.5
# 可以使用不连续的索引。
data = pd.Series([0.25, 0.5, 0.75, 1.0], index=[2, 5, 3, 7])
print(data)
print(data[5])
```

任务：Series是特殊的字典

> 知识点：
>
> - 可以把`Series`看成是一种特殊的Python字典。
> - - 字典是将`key`映射到`value`的数据结构。
>   - `Series`是一组索引映射到一组**类型值**的数据结构。
> - 相对于，NumPy数组的底层实现是通过优化编译的，它的效率比普通的Python列表更高效。
> - `Series`是建立在NumPy基础上.因此，它的操作会比Python字典更高效。

例子：可以直接将字典转换为`Series`。

```
population_dict = {'California': 38332521, 'Texas': 26448193,
                   'New York': 19651127,'Florida': 19552860,
                   'Illinois': 12882135}
population = pd.Series(population_dict)
print(population)
```

> 知识点：
>
> - 用字典创建`Series`时，其索引默认按照顺序排列。
> - 与字典不同，`Series`是**有序**的，因此，它支持切片操作。

```
# 类似字典，通过标签来访问元素
print(population['California'])
# 支持切片操作。字典不能使用切片因为是无序的
print(population['California':'Illinois'])
```

9.2.4 Series的索引

`Series`共有两种访问元素方式：**隐式位置索引**和**显式标签索引**。

任务：隐式位置索引

> 知识点：类似于`ndarray`数组和列表，`Series`可以根据位置来**隐式索引**元素。

例子：检索第一个元素。由于数组是从零开始计数的，那么第一个元素存储在零位置。

```
s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])
print(s[0])
# 1
```

例子：使用切片索引`Series`的前三个元素。

```
s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])
print(s[:3])
# a  1
# b  2
# c  3
# dtype: int64
```

例子：使用切片索引`Series`的最后三个元素。

```
s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])
print(s[-3:])
# c  3
# d  4
# e  5
# dtype: int64
```

任务：显式标签索引

> 知识点：类似于字典，`Series`可以通过**标签**来**显式索引**元素。

例子：使用标签索引来访问单个元素。

```
s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])
print(s['a'])
# 1
```

例子：使用标签索引来访问多个元素。

```
s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])
print(s[['a','c','d']])
# a  1
# c  3
# d  4
# dtype: int64
```

例子：如果不包含指定的标签，则会出现异常。

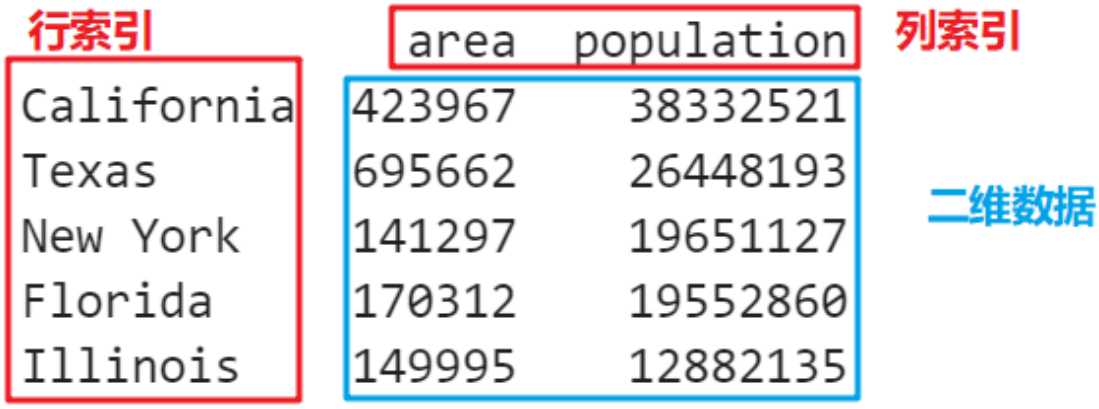
```
s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])
print(s['f'])
# KeyError: 'f'
```

9.3 数据结构：二维Dataframe类型

9.3.1 Dataframe的基本概念

任务：Dataframe的基本概念

> 知识点：
>
> - `DataFrame`是带索引的二维数据结构，数据以行和列的表格方式排列。本质上，`DataFrame`就是一个**表格数据**。
> - 每个列**必须是相同**数据类型，不同列**可以是不同**类型。
> - 类似于`Series`，`DataFrame`可以使用**标记**对行和列进行索引，也可以对行和列执行算术运算。



可以将`DataFrame`的**每个列**看作是一个`Series`。

任务：DataFrame的构造函数

`DataFrame`的构造函数描述如下：

```
pandas.DataFrame(data, index, columns, dtype, copy)
```

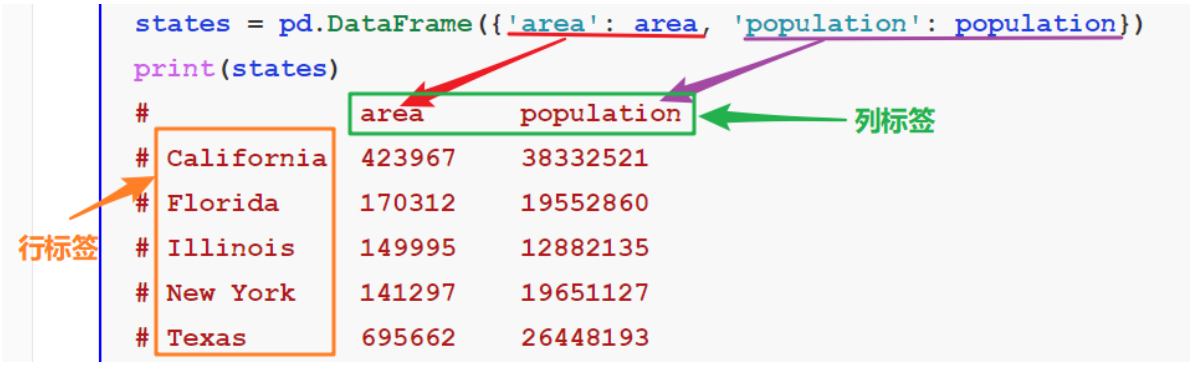
构造函数的参数如下：

- `data`：数据。数据采取各种形式，如`ndarray`，`series`，`map`，`lists`，`dict`、`DataFrame`等。
- `index`：行标签。索引值必须是**唯一**的(不可变类型)，与**数据的长度相同。** 默认`np.arange(n)`如果没有索引被传递。
- `columns`列标签。默认`np.arange(n)`如果没有索引被传递。
- `dtype`：每列的数据类型。如果没有，Pandas会根据数据类型进行推断。
- `copy`：复制数据，默认为`false`。

例子：由2个`Series`创建`DataFrame`。

```
# 创建Series
```

```python
area_dict = {'California': 423967, 'Texas': 695662, 'New York': 141297,
             'Florida': 170312, 'Illinois': 149995}
area = pd.Series(area_dict)

# 创建Series
population_dict = {'California': 38332521, 'Texas': 26448193,
                   'New York': 19651127,'Florida': 19552860,
                   'Illinois': 12882135}
population = pd.Series(population_dict)

# 从Series创建DataFrame，作为列
states = pd.DataFrame({'area': area, 'population': population})
print(states)
#             area      population
# California  423967    38332521
# Florida     170312    19552860
# Illinois    149995    12882135
# New York    141297    19651127
# Texas       695662    26448193
```



任务：获取行标签、列标签和数据

> 知识点：
>
> - `index` 属性：获取**行标签**索引。与`Series`类似。
> - `columns` 属性：获取**列标签**索引。
> - `values`属性：获取数据。类型是`ndarray`。
> - 标签索引类型都是`Index`。

```python
# 获得行标签索引
print(states.index)
#Index(['California', 'Florida', 'Illinois', 'New York', 'Texas'],
dtype='object')

# 获得列标签索引
print(states.columns)
# Index(['area', 'population'], dtype='object')

# 获得数据，二维数组
print(type(states.values),'\n', states.values)
# <class 'numpy.ndarray'>
#  [[  423967 38332521]
#  [  695662 26448193]
```

```
#  [  141297 19651127]
#  [  170312 19552860]
#  [  149995 12882135]]
```

9.3.2 Dataframe的创建

任务：创建一个空的DataFrame

```
df = pd.DataFrame()
print(df)
# Empty DataFrame
# Columns: []
# Index: []
```

任务：从列表或嵌套列表的数据中来创建DataFrame

> 知识点：
>
> - 使用**单列表**创建`DataFrame`，此时的`DataFrame`只包含一个列。
> - 使用**嵌套列表**创建`DataFrame`。

例子：从单列表创建`DataFrame`。

```
# 默认行和列标签从0开始计数
data = [6,3,2,5]
df = pd.DataFrame(data)
print(df)
#    0
# 0  6
# 1  3
# 2  2
# 3  5
```

例子：从嵌套单列表创建`DataFrame`，并通过`columns`参数指定列标签，行标签默认从0开始计数。

```
data = [['Alex',10],['Bob',12],['Clarke',13]]
df = pd.DataFrame(data, columns=['Name','Age'])
print(df)
#       Name  Age
# 0    Alex   10
# 1     Bob   12
# 2  Clarke   13
```

例子：通过`dtype=float`参数，将数值类型的列指定`DataFrame`为浮点数，而非数值类型的不受影响。

```
data = [['Alex',10],['Bob',12],['Clarke',13]]
df = pd.DataFrame(data,columns=['Name','Age'],dtype=float)
print(df)
#      Name   Age
# 0    Alex  10.0
# 1     Bob  12.0
# 2  Clarke  13.0
```

任务：从列表嵌套字典的数据中来创建DataFrame

> 知识点：
>
> - 列表嵌套字典：列表作为字典的`values`。
> - 将**相同长度**的列表或`ndarray`对象作为字典的`values`，来创建`DataFrame`。
> - 不需要指定`columns`参数，Pandas自动从字典中提取`key`作为`DataFrame`的列索引。
> - 如果传递了行索引(`index`)，则索引的长度应等于数组的长度。
> - 如果没有传递行索引，则索引默认为`range(n)`，其中`n`为数组长度。

例子：字典的`values`嵌套列表。将字典`data`的`key`作为列索引。行标签默认从0开始计数。

```
data = {'Name':['Tom', 'Jack', 'Steve', 'Ricky'],'Age':[28,34,29,42]}
df = pd.DataFrame(data)
print(df)
#     Name  Age
# 0    Tom   28
# 1   Jack   34
# 2  Steve   29
# 3  Ricky   42
```

例子：字典的`values`嵌套列表。列表长度不一致会触发异常，`Traceback`很长，可以直接翻阅到出错代码位置和最后一行。

```
# 第一个元素长度为4，第2元素长度为5
data = {'Name':['Tom', 'Jack', 'Steve', 'Ricky'],'Age':[28,34,1,29,42]}
df = pd.DataFrame(data)
print(df)
# ValueError: arrays must all be same length
```

> 知识点：`index`参数为每行分配一个行索引。

例子：指定`index`参数。注意，`index`参数的长度必须要和列表长度一致，否则会触发异常。

```
data = {'Name':['Tom', 'Jack', 'Steve', 'Ricky'],'Age':[28,34,29,42]}
df = pd.DataFrame(data, index=['rank1','rank2','rank3','rank4'])
print(df)
#         Name  Age
# rank1    Tom   28
# rank2   Jack   34
```

```
# rank3  Steve   29
# rank4  Ricky   42
# index参数的长度与列表长度不一致，触发异常
data = {'Name':['Tom', 'Jack', 'Steve', 'Ricky'],'Age':[28,34,29,42]}
df = pd.DataFrame(data, index=['rank1','rank2','rank3','rank4','rank5'])
print(df)
# values(列表)的shape为(4, 2)，但是行索引需要列表shape为(5, 2)
# ValueError: Shape of passed values is (4, 2), indices imply (5, 2)
```

任务：从字典嵌套列表的数据中创建DataFrame

> 知识点：
>
> - 字典嵌套列表：字典作为列表元素。
> - 使用字典`key`作为默认的列索引。
> - 每个字典，分别作为`DataFrame`的**行**数据。
> - 字典元素个数**允许不一样**。对于缺少的`key`，在`DataFrame`中使用`NaN`来填充。

例子：使用字典嵌套列表数据来创建`DataFrame`。

```
# 列表的第一个字典缺少key='c'的元素，使用NaN来填充
data = [{'a': 1, 'b': 2},{'a': 5, 'b': 10, 'c': 20}]
df = pd.DataFrame(data)
print(df)
# 每个字典分别作为一行数据
#    a   b     c
# 0  1   2   NaN
# 1  5  10  20.0
```

> 知识点：`index`的长度，必须要和字典个数一致，否则会引起异常。

例子：指定`index`参数。

```
data = [{'a': 1, 'b': 2},{'a': 5, 'b': 10, 'c': 20}]
df = pd.DataFrame(data, index=['first', 'second'])
print(df)
#         a   b     c
# first   1   2   NaN
# second  5  10  20.0
```

> 知识点：
>
> - `columns`参数本质上是用`columns`访问每个字典对应的`key`元素。
> - 如果元素存在，则在相应列上填上对应值。
> - 如果不存在，则用`NaN`填充。
> - 对于非`columns`中的key，`DataFrame`会忽略这些元素。

例子：使用字典嵌套列表数据，指定`index`参数，并使用`columns`参数筛选元素。

```
data = [{'a': 1, 'b': 2},{'a': 5, 'b': 10, 'c': 20}]
# columns中不包含'c'，因此，DataFrame会忽略第2个字典的'c': 20元素
df1 = pd.DataFrame(data, index=['first', 'second'], columns=['a', 'b'])
print(df1)
#         a   b
# first   1   2
# second  5  10
data = [{'a': 1, 'b': 2},{'a': 5, 'b': 10, 'c': 20}]
# 第2个columns为b1，data中，没有关键字为key的元素，所以都用NaN填充。
df2 = pd.DataFrame(data, index=['first', 'second'], columns=['a', 'b1'])
print(df2)
#         a  b1
# first   1 NaN
# second  5 NaN
```

任务：从Series嵌套字典的数据中创建DataFrame

> 知识点：
>
> - `Series`嵌套字典：`Series`作为字典的`values`。
> - 使用字典`key`作为默认的列索引。
> - 每个字典，分别作为`DataFrame`的**列**数据。
> - 行索引是所有`Series`索引的**并集**。对于缺少相应`index`元素的`Series`，在`DataFrame`中使用`NaN`来填充。

例子：使用`Series`嵌套字典的数据创建`DataFrame`。

```
# 行索引是所有Series索引的并集。
d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
     'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}
df = pd.DataFrame(d)
print(df)
#    one  two
# a  1.0    1
# b  2.0    2
# c  3.0    3
# d  NaN    4
```

对于字典第一个元素(`Series`)，由于缺少标签`'d'`对应的元素。因此在`DataFrame`中，`d`行标签，用`NaN`填充。

9.3.3 （重要）Dataframe基本属性和方法

`Dataframe`数据结构主要包含如下的属性和方法：

编号	属性或方法	描述
1	`T`	转置行和列。
2	`axes`	返回一个列，行轴标签和列轴标签作为唯一的成员。
3	`dtypes`	返回此对象中的数据类型(`dtypes`)。
4	`empty`	如果 `DataFrame` 为空，则返回为 `True` ，任何轴的长度都为 `0` 。
5	`ndim`	数组维度大小，默认为2维
6	`shape`	返回表示 `DataFrame` 的维度的元组。
7	`size`	`DataFrame` 中的元素个数。
8	`values`	将 `DataFrame` 中的实际数据作为 `NDarray` 返回。
9	`head()`	返回开头前 `n` 行。
10	`tail()`	返回最后 `n` 行。

首先，从字典中，创建一个 `Dataframe` 对象。

```
d = {'Name':pd.Series(['Tom','James','Ricky','Vin','Steve','Minsu','Jack']),
  'Age':pd.Series([25,26,25,23,30,29,23]),
  'Rating':pd.Series([4.23,3.24,3.98,2.56,3.20,4.6,3.8])}
df = pd.DataFrame(d)
print(df)
#     Name  Age  Rating
# 0    Tom   25    4.23
# 1  James   26    3.24
# 2  Ricky   25    3.98
# 3    Vin   23    2.56
# 4  Steve   30    3.20
# 5  Minsu   29    4.60
# 6   Jack   23    3.80
```

任务：T转置属性

> 知识点：类似于线性代数的转置运算，返回 `DataFrame` 的转置，行和列将交换。

```
print(df.T)
#            0      1      2     3      4      5     6
# Name     Tom  James  Ricky   Vin  Steve  Minsu  Jack
# Age       25     26     25    23     30     29    23
# Rating  4.23   3.24   3.98  2.56    3.2    4.6   3.8
```

任务：axes属性

> 知识点：返回行标签和列标签的 `Index` 对象。

```
# 先行标签，后列标签
print(df.axes)
# [RangeIndex(start=0, stop=7, step=1),
# Index(['Name', 'Age', 'Rating'], dtype='object')]
```

任务：dtypes属性

> 知识点：
>
> - 返回**每列**的数据类型。
> - 看清楚，是每列，而非每行。

```
# 先行标签，后列标签
print(df.dtypes)
# Name       object
# Age         int64
# Rating    float64
# dtype: object
```

任务：empty属性

> 知识点：返回布尔值，表示`DataFrame`是否为空，返回`True`表示`DataFrame`为空。

```
print(df.empty)
# False
```

任务：ndim、shape和size属性

> 知识点：
>
> - `ndim`属性：返回`DataFrame`的维数。由于`DataFrame`是二维的，所以该返回值为2。
> - `shape`属性：返回`DataFrame`的形状元组。元组`(a, b)`，其中`a`表示行数，`b`表示列数。
> - `size`属性：返回`DataFrame`的总元素个数。

```
print(df.ndim)
# 2

print(df.shape)
# (7, 3)

print(df.size)
# 21
```

任务：values属性

> 知识点：将`DataFrame`中的实际数据作为`ndarray`对象返回。

```
print(df.values)
# [['Tom' 25 4.23]
#  ['James' 26 3.24]
#  ['Ricky' 25 3.98]
#  ['Vin' 23 2.56]
#  ['Steve' 30 3.2]
#  ['Minsu' 29 4.6]
#  ['Jack' 23 3.8]]
```

任务：head()和tail()方法

> 知识点：
>
> - 要查看`DataFrame`对象的部分样本，可使用`head()`和`tail()`方法。
> - `head()`：返回前`n`行。默认数量为5，可以传递自定义数值。
> - `tail()`：返回最后`n`行。默认数量为5，可以传递自定义数值。

```
# 前3行
print(df.head(3))
#     Name  Age  Rating
# 0    Tom   25    4.23
# 1  James   26    3.24
# 2  Ricky   25    3.98

# 后2行
print(df.tail(2))
#     Name  Age  Rating
# 5  Minsu   29     4.6
# 6   Jack   23     3.8
```

9.3.4 外部文件中读取Dataframe数据

知识点：

- 通过调用Pandas函数，可以从外部文件中将数据读取为`Dataframe`格式。
- 支持txt、csv、Excel、MySQL等。
- MySQL需要安装数据库，可以暂时跳过。

9.3.5 Dataframe与二维数组和字典的关系

任务：DataFrame的数据是二维数组

> 知识点：
>
> - `Series`类比为带索引的**一维数组**。
> - `DataFrame`类比于带行和列索引的**二维数组**。
> - `DataFrame`的`values`属性本质上就是`numpy.ndarray`的二维数组。

```
# 创建Series
area_dict = {'California': 423967, 'Texas': 695662, 'New York': 141297,
```

```
            'Florida': 170312, 'Illinois': 149995}
area = pd.Series(area_dict)

# 创建Series
population_dict = {'California': 38332521, 'Texas': 26448193,
                   'New York': 19651127,'Florida': 19552860,
                   'Illinois': 12882135}
population = pd.Series(population_dict)

# 从Series创建DataFrame，作为列
states = pd.DataFrame({'area': area, 'population': population})
print(states)
#             area      population
# California  423967    38332521
# Florida     170312    19552860
# Illinois    149995    12882135
# New York    141297    19651127
# Texas       695662 26448193
print(states.index)
# Index(['California', 'Texas', 'New York', 'Florida', 'Illinois'],
dtype='object')
print(states.columns)
# Index(['area', 'population'], dtype='object')

# 观察values，发现类型是'numpy.ndarray'
print(type(states.values))
# <class 'numpy.ndarray'>
print(states.values)
```

因此，`DataFrame` 可以看作一种NumPy二维数组，它的行与列都可以通过索引来获取。

任务：DataFrame是特殊的字典

> 知识点：
>
> - 与 `Series` 类似，`DataFrame` 也是一个特殊的字典。
> - 字典是每个`key`映射`values`，而 `DataFrame` 是每个列索引映射 `Series` 的数据。

例子：通过 `'area'` 的列索引返回返回包含面积数据的 `Series` 对象。

```
print(states['area'])
# California    423967
# Florida       170312
# Illinois      149995
# New York      141297
# Texas         695662
# Name: area, dtype: int64
```

任务(注意)：二维数组和DataFrame的单索引并不一样

> 知识点：
>
> - 在NumPy的二维数组中，`data[0]` 返回**第一行**。
> - 在 `DataFrame` 中，`data['col0']` 返回**第一列**。
> - 为了**避免索引混淆**，最好把 `DataFrame` 看成一种通用字典，而不是二维数组。

9.3.6 Dataframe的操作

任务：索引列

通过指定`columns`列标签，索引列元素。

```
d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
     'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}
df = pd.DataFrame(d)
print(df)

# 索引'one'这列
print(df['one'])
# a    1.0
# b    2.0
# c    3.0
# d    NaN
# Name: one, dtype: float64
```

任务：添加列

> 知识点：类似**字典**的添加新元素方式，可以`Series`作为指定列标签的列添加到`DataFrame`。

例子：为`DataFrame`添加指定列标签的列。

```
d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
     'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}
df = pd.DataFrame(d)

df['three']=pd.Series([10,20,30],index=['a','b','c'])
print(df)
#    one  two  three
# a  1.0    1   10.0
# b  2.0    2   20.0
# c  3.0    3   30.0
# d  NaN    4    NaN
```

例子：将第1和第3列元素相加，并作为新列添加到`DataFrame`。

```
df['four']=df['one']+df['three']
print(df)
#    one  two  three  four
# a  1.0    1   10.0  11.0
# b  2.0    2   20.0  22.0
# c  3.0    3   30.0  33.0
# d  NaN    4    NaN   NaN
```

任务：删除列

> 知识点：使用`del`或`pop()`删除指定**列标签**的列。

```
d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
    'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd']),
    'three' : pd.Series([10,20,30], index=['a','b','c'])}
df = pd.DataFrame(d)
print(df)
```

使用`del`删除列标签为`'one'`的列。

```
del df['one']
print(df)
#    two  three
# a    1   10.0
# b    2   20.0
# c    3   30.0
# d    4    NaN
```

使用`pop()`删除列标签为`'two'`的列，`pop()`弹出的列是`Series`对象。

```
s = df.pop('two')
print(df)
#    three
# a   10.0
# b   20.0
# c   30.0
# d    NaN

print(s)
# a    1
# b    2
# c    3
# d    4
# Name: two, dtype: int64
```

9.3.7 Dataframe的行操作

任务：显式标签索引行

> 知识点：
>
> - 通过向`loc()`方法传递要选择的**行标签索引**。
> - 返回是将`DataFrame`的列标签作为标签的`Series`对象。

```
d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
    'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}
df = pd.DataFrame(d)
print(df.loc['b'])
# one    2.0
# two    2.0
# Name: b, dtype: float64
```

任务：隐式位置索引行

> 知识点：
>
> - 通过向`iloc()`方法传递要选择的**行位置索引**。
> - 返回是将`DataFrame`的列标签作为标签的`Series`对象。

```
d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
    'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}
df = pd.DataFrame(d)
print(df.iloc[2])
# one    3.0
# two    3.0
# Name: c, dtype: float64
```

任务：切片行

> 知识点：类似NumPy数组，`DataFrame`可以使用切片来返回子`DataFrame`。

```
d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
   'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}
df = pd.DataFrame(d)
print(df[2:4])
#    one  two
# c  3.0    3
# d  NaN    4
```

任务：添加行

> 知识点：使用`append()`方法将新行添加到`DataFrame`的尾部。

```
df = pd.DataFrame([[1, 2], [3, 4]], columns = ['a','b'])
df2 = pd.DataFrame([[5, 6], [7, 8]], columns = ['a','b'])
# 将df2追加到df尾部
df = df.append(df2)
print(df)
#    a  b
# 0  1  2
# 1  3  4
# 0  5  6
# 1  7  8
```

任务：删除行

> 知识点：
>
> - 使用`drop()`方法行索引从`DataFrame`中删除行。
> - 如果标签重复，则**会删除多行。**

```
df = pd.DataFrame([[1, 2], [3, 4]], columns = ['a','b'])
df2 = pd.DataFrame([[5, 6], [7, 8]], columns = ['a','b'])
df = df.append(df2)

# 新的df有两个0行标签
print(df)
#    a  b
# 0  1  2
# 1  3  4
# 0  5  6
# 1  7  8

# 删除行标签为0的行。
# 一共有两行被删除，因为这两行包含相同的标签0。
df = df.drop(0)
print(df)
#    a  b
# 1  3  4
# 1  7  8
```

第10章Pandas数据分析模块（二）：索引操作

思维导图

10.1 数据结构：索引Index类型

任务：Index索引基本介绍

> 知识点：
>
> - Pandas专门有一个数据类型`Index`，用于管理标签索引。
> - `Series`和`DataFrame`数据结构：显式标签`Index`索引 + `ndarray`数据。
> - 在创建`Series`和`DataFrame`时，通过参数`index`和`columns`传递索引标签，都会被转换为`Index`类型。
> - 和字典的`key`类似，`Index`对象必须是**不可变类型**序列。

例子：使用整数列表来创建一个`Index`对象。

```
ind = pd.Index([2, 3, 5, 7, 11])
print(ind)
# Int64Index([2, 3, 5, 7, 11], dtype='int64')
```

任务：Index类继承ndarray类

> 知识点：类似于`ndarray`数组，`Index`对象同样支持切片操作。

```
ind = pd.Index([2, 3, 5, 7, 11])
print(ind[1])
# 3
print(ind[::2])
#Int64Index([2, 5, 11], dtype='int64')
```

> 知识点：`Index`继承自`ndarray`，拥有`shape`、`ndim`、`dtype`等属性。

```
ind = pd.Index([2, 3, 5, 7, 11])
print(ind.size, ind.shape, ind.ndim, ind.dtype)
# 5 (5,) 1 int64
```

任务：Index对象必须是不可变数组

> 知识点：
>
> - `Index`对象中的索引数据必须是**不可变**的。
> - 如果对索引进行修改，会触发异常。`TypeError: Index does not support mutable operations`。
> - 设置为不可变的**原因**：`Index`对象的不可变特征使得多个`DataFrame`和数组之间共享使用相同索引时**更加安全**，可以避免因不小心修改索引而导致程序发生逻辑错误。

```
print(ind[1] = 0)
# TypeError: Index does not support mutable operations
```

任务：Index支持集合操作

> 知识点：
>
> - Pandas支持多个`Series`和`DataFrame`进行拼接(join)操作。
> - 因此，`Index`对象也被设计为支持类似集合`set`的逻辑运算，比如并集、交集、差集。

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

print(indA & indB) # 交集
# Int64Index([3, 5, 7], dtype='int64')

print(indA | indB) # 并集
# Int64Index([1, 2, 3, 5, 7, 9, 11], dtype='int64')

print(indA ^ indB) # 异或
# Int64Index([1, 2, 9, 11], dtype='int64')
```

10.2 Pandas的索引和数据选择

任务：需求分析

> 知识点(回顾)：
>
> - Python和NumPy通过**索引运算符**`"[]"`来访问数据。
> - NumPy支持四大索引技术：单值索引(`arr[2, 1]`)、切片(`arr[:, 1:5]`)、掩码索引(`arr[arr > 0]`)、花式索引(`arr[0, [1, 5]]`)。以及它们之间的组合索引(`arr[:, [1, 5]]`)。

> 知识点(术语)：
>
> - 位置索引或**隐式**索引：通过元素的位置来索引元素。范围为0到`len`-1，`len`为相应维度的长度。
> - 标签索引或**显式**索引：通过元素对应的标签(`key`)来索引元素。
> - 注意：在`DataFrame`中，原NumPy的**索引运算符**`"[]"`只支持`DataFrame`的**标签列索引**方式。

    - 支持标签列索引`df['a']`，但不支持位置索引`df[0]`。
    - 不支持多维索引。`df['groups','a']`触发异常。

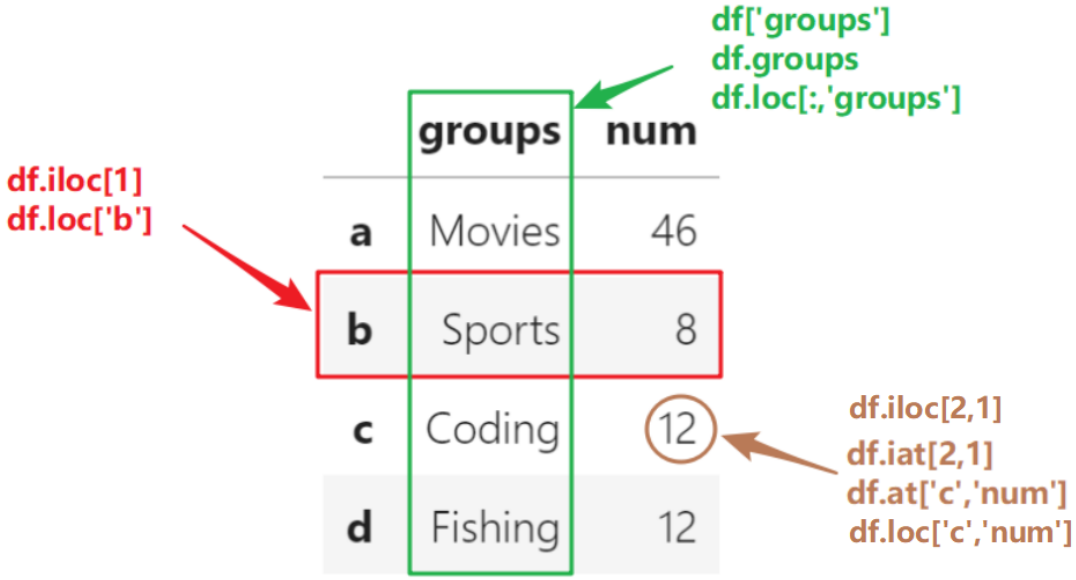
```
groups = ["Movies", "Sports", "Coding", "Fishing"]
num = [46, 8, 12, 12]
dict = {"groups":groups, "num": num}

# 从字典中创建DataFrame
df = pd.DataFrame(dict, index=['a','b','c','d'])

print(df)
#     groups  num
# a   Movies   46
# b   Sports    8
# c   Coding   12
# d  Fishing   12
# 获取"groups"列
print(df["groups"])
# a     Movies
# b     Sports
# c     Coding
# d    Fishing
# Name: groups, dtype: object
# 位置索引触发异常
# 索引运算符[]默认使用标签(关键字)索引
print(df[0])
# KeyError: 0
# 位置索引触发异常
# 索引运算符[]默认使用标签(关键字)索引
print(df['groups','a'])
# KeyError: ('groups', 'a')
```

任务：扩展索引运算符[]，支持多维位置和标签索引

Pandas为了让`Series`和`DataFrame`同时支持**多维的**位置和标签索引方式，对索引运算符`[]`进行了扩展，共提供了3种**索引方式**：显式索引`.loc[]`、隐式索引`.iloc[]`和组合索引`.ix[]`。

> 知识点：
>
> - `.loc[]`：**显式**索引。
>   - 输入为标签，通过标签来索引元素。
>   - 例如，`df.loc["b"]`。
> - `.iloc[]`：**隐式**索引，`i`表示`implicit`。
>   - 输入为位置，通过位置来索引元素。从0开始，到`len`-1结束。
>   - 例如，`df.iloc[1]`。
> - `.ix[]`：**组合**索引。**不建议使用**，已被淘汰。
>   - 不同维度的索引上，使用标签和位置的组合形式。
>   - 例如，`df.iloc["a",0]`。
> - 注意是`[]`而非`()`。

一图了解所有索引用法，先仔细看图，再对着程序观察运行结果。有很多等价形式，看懂他人程序，掌握一种使用方式。



	groups	num
a	Movies	46
b	Sports	8
c	Coding	12
d	Fishing	12

```
# 显式标签索引：获取元素
print(df.loc['c','num'])
print(df.at['c','num'])
# 12

# 显式标签索引：获取行元素
print(df.loc['b'])
# groups    Sports
# num            8
# Name: b, dtype: object

# 显式标签索引：获取列元素
print(df.loc[:,'groups'])
print(df['groups'])
print(df.groups)
# a     Movies
# b     Sports
# c     Coding
# d    Fishing
# Name: groups, dtype: object
# 隐式标签索引：获取元素
print(df.iloc[2,1])
print(df.iat[2,1])
# 12

# 隐式标签索引：获取行元素
print(df.iloc[1])
# groups    Sports
# num            8
# Name: b, dtype: object

# 隐式标签索引：获取列元素
print(df.iloc[:,0])
# a     Movies
```

```
# b     Sports
# c     Coding
# d    Fishing
# Name: groups, dtype: object
```

10.3 显示和隐式索引

10.3.1 显式标签索引loc[]

> 知识点：`df.loc[]`主要基于标签(label)的，包括行标签(`index`)和列标签(`colums`)，即行名称和列名称，可以使用`df.loc[index_name, col_name]`选择指定位置的数据，主要用法有：
>
> - **单索引**：如果`.loc[]`中只有**单标签**，那么**选择行**。如：`df.loc['a']`选择的是`index`为`'a'`行，等价于`df.loc['a',:]`。
> - **切片**：与通常的python切片不同，在最终选择的数据中**包含**切片的`start`和`stop`。如：`df.loc['c':'h']`即包含`'c'`行，也包含`'h'`行。
> - **掩码索引**：用于筛选符合某些条件的行，如：`df.loc[df.A>0.5]`筛选出所有`'A'`列大于0.5的行。
> - **花式索引**：如：`df.loc[['a', 'b', 'c']]`，同样只选择行。

生成DataFrame数据。

```
df = pd.DataFrame(np.random.randn(4, 4),
index = ['a','b','c','d'], columns = ['A', 'B', 'C', 'D'])
print(df)
#           A         B         C         D
# a  0.381720 -0.134319  0.276336  0.927901
# b  0.905183 -0.707632  0.150454 -1.229599
# c -0.165328  1.834757  1.301909 -0.444526
# d -0.434927 -0.951217 -0.823918  0.568995
```

切片+单索引：选择'A'列。

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

切片+单索引：选择'a'行。

```
# 缺省列切片，等价于df.loc['a',:]
print(df.loc['a'])
```

切片+花式索引：选择'A'和'C'列。

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

花式索引(广播)。

```
print(df.loc[['a','b','f','h'],['A','C']])
```

标签切片：从'b'到'd'。包括'd'行。

```
print(df.loc['b':'d':2])
```

获得掩码数组：判断'a'行>0的逻辑结果。返回的是'a'行的列元素逻辑判断。

```
print(df.loc['a']>0)
# 掩码索引：选择'a'行>0的列元素，需要把掩码放在列索引。
print(df.loc[:,df.loc['a']>0])
```

获得掩码数组：判断'A'列>0的逻辑结果。返回的是'A'列的行元素逻辑判断。

```
print(df.A>0)
# 掩码索引：选择'A'列>0的行元素，需要把掩码放在行索引。
# 缺省列切片等价于df.loc[df.A>0,:]
print(df.loc[df.A>0])
```

10.3.2 隐式位置索引iloc[]

> 知识点：`.iloc()` 是基于位置索引，利用元素在各个`axis`上的索引序号位置进行选择，序号超过范围产生`IndexError`。主要用法有：
>
> - 单索引：与`.loc`相同，若只使用一个维度，则默认对**行**索引，下标从 0 开始。如：`df.iloc[5]`，选择第 6 行。
> - 切片：与`.loc`不同，`stop`元素不被选择。如：`df.iloc[0:3]`， 只包含 0，1，2行，不包含第 3 行。**切片时允许序号超过范围**。
> - 隐码索引：使用掩码数组对行筛选，如`df.iloc[np.array(df.A>0.5)]`，`df.iloc[list(df.A>0.5)]`。
> - 花式索引：如`df.iloc[[5, 1, 7]]`，选择第 6 行， 第 2 行， 第 8 行。
> - 上述结论同样适用于列索引或元素索引。

生成DataFrame数据。

```
df = pd.DataFrame(np.random.randn(4, 4),
index = ['a','b','c','d'], columns = ['A', 'B', 'C', 'D'])
print(df)
# 切片：选择从0~2行
print (df.iloc[:3])
# 切片：选择从2~3列
print (df.iloc[:,2:4])
# 切片：选择从1~2行，2~3列
print (df.iloc[1:3, 2:4])
print (df.iloc[[1, 3], [1, 3]])
print (df.iloc[1:3, :])
print (df.iloc[:,1:3])
```

10.3.3 进阶用法

任务：`loc[]`、`iloc[]`和`[]`的主要区别

> 知识点：
>
> - `loc[]`和`iloc[]`，若只使用一个维度，则默认选择**行**。
> - Python的`[]`选择的是**列**，并且必须使用列名。

```
df = pd.DataFrame(np.random.randn(8, 4),
index = ['a','b','c','d','e','f','g','h'], columns = ['A', 'B', 'C', 'D'])
# []选择A列
print(df['A'])

# df.loc[]选择a行
print(df.loc['a'])
```

任务：`.`属性符选择列

> 知识点：
>
> - 使用属性运算符`.`来选择列。
> - 用法`df.列标签`，列标签**不需要加引号。**

```
df = pd.DataFrame(np.random.randn(8, 4),
index = ['a','b','c','d','e','f','g','h'], columns = ['A', 'B', 'C', 'D'])
# []选择A列
print(df['A'])
# df.loc[]选择A列
print(df.loc[:,'A'])
# .属性运算符选择A列
print(df.A)
```

任务：索引运算符[]只能选择列

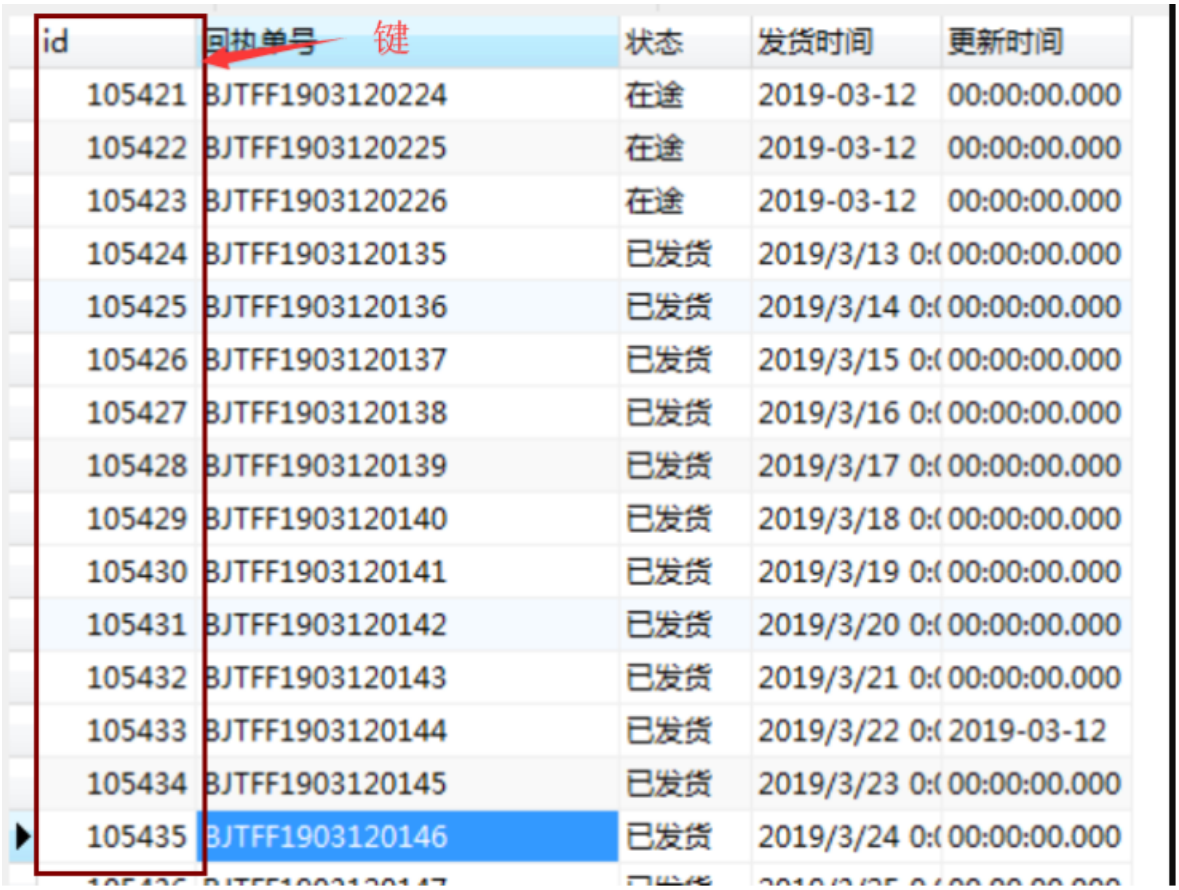
> 知识点：Python的索引运算符`[]`只能输入一个维度，不能用逗号隔开输入两个维度。

```
df = pd.DataFrame(np.random.randn(8, 4),
index = ['a','b','c','d','e','f','g','h'], columns = ['A', 'B', 'C', 'D'])

print(df['a', 'A'])
# 触发异常 KeyError: ('a', 'A')
```

10.4 概念讲解：数据表中的键和关键列

任务：数据库中的键

> 知识点：
>
> - 在关系型数据库中，通常都会为每条行数据，都会分配一个能**唯一标识该行数据**的`id`号。
> - 这个`id`被称为**键**。通过**键**能快速的定位到某条数据，便于后续的查询、删除、修改等操作。
> - 现实中，存在大量关于**键**的使用，例如通过**学号**来唯一标识某个学生。
> - 为防止歧义性，通常要求作为**键不重复**。
> - 数据表中，键所在的列(字段)称为**关键列**。如下面的订单数据表中，`id`唯一标识了每行数据。
> - **数据库是找工作的敲门砖，建议熟练掌握。**



id	回执单号	状态	发货时间	更新时间
105421	BJTFF1903120224	在途	2019-03-12	00:00:00.000
105422	BJTFF1903120225	在途	2019-03-12	00:00:00.000
105423	BJTFF1903120226	在途	2019-03-12	00:00:00.000
105424	BJTFF1903120135	已发货	2019/3/13 0:(	00:00:00.000
105425	BJTFF1903120136	已发货	2019/3/14 0:(	00:00:00.000
105426	BJTFF1903120137	已发货	2019/3/15 0:(	00:00:00.000
105427	BJTFF1903120138	已发货	2019/3/16 0:(	00:00:00.000
105428	BJTFF1903120139	已发货	2019/3/17 0:(	00:00:00.000
105429	BJTFF1903120140	已发货	2019/3/18 0:(	00:00:00.000
105430	BJTFF1903120141	已发货	2019/3/19 0:(	00:00:00.000
105431	BJTFF1903120142	已发货	2019/3/20 0:(	00:00:00.000
105432	BJTFF1903120143	已发货	2019/3/21 0:(	00:00:00.000
105433	BJTFF1903120144	已发货	2019/3/22 0:(	2019-03-12
105434	BJTFF1903120145	已发货	2019/3/23 0:(	00:00:00.000
105435	BJTFF1903120146	已发货	2019/3/24 0:(	00:00:00.000

任务：Dataframe中的键和关键列

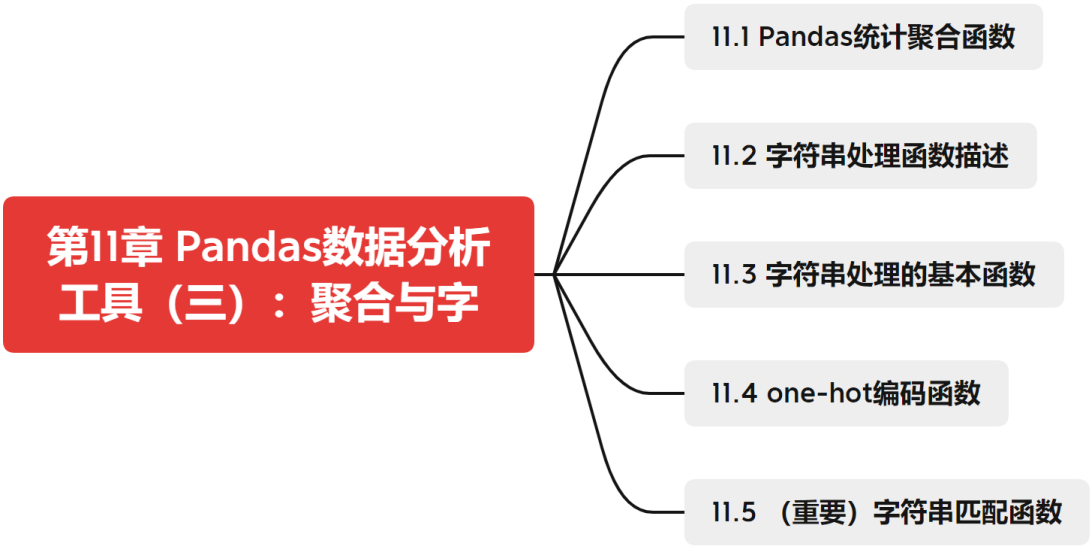
> 知识点：
>
> - `Dataframe`和Excell、关系型数据库类似，都是通过**数据表**的形式储存数据。
> - 在`Dataframe`中，通常选择**行索引**或**某列**数据作为**键**，用来**唯一标识**某行数据。
> - 字典通过**键值对**来管理数据，即用`key`管理`value`，`key`和`value`一一对应。
> - `Dataframe`的**键**是用于管理**整行**数据，**键**位于的列称之为**关键列**。
> - 关键列非常重要，`Dataframe`的进阶操作，合并、分组、数据透视操作都是围绕着关键列进行的。

	method	number	orbital_period	mass	distance	year
0	Radial Velocity	1	269.300000	7.10	77.40	2006
1	Radial Velocity	1	874.774000	2.21	56.95	2008
2	Radial Velocity	1	763.000000	2.60	19.84	2011
3	Radial Velocity	1	326.030000	19.40	110.62	2007
4	Radial Velocity	1	516.220000	10.50	119.47	2009
...	...	...	...	...	...	...
1030	Transit	1	3.941507	NaN	172.00	2006
1031	Transit	1	2.615864	NaN	148.00	2007
1032	Transit	1	3.191524	NaN	174.00	2007
1033	Transit	1	4.125083	NaN	293.00	2008
1034	Transit	1	4.187757	NaN	260.00	2008

通过key 1030索引整条数据

第11章 Pandas数据分析工具（三）：聚合与字符串处理

思维导图



11.1 Pandas统计聚合函数

任务：聚合函数基本介绍

> 知识点：
>
> - Pandas为`DataFrame`提供了一系列的**数据统计聚合**方法。
> - 类似于NumPy的聚合，`DataFrame`也需要为聚合操作指定`axis`。
> - `DataFrame`是**二维**数据结构：行`index`为`axis=0`(默认)；列`columns`为`axis=1`。

下表给出了Pandas提供的聚合函数以及功能描述。它们都拥有一个共同的参数`axis`，用于指定沿哪个`axis`进行聚合操作。

编号	函数	描述
1	`count()`	非空数据的数量
2	`sum()`	所有值之和
3	`mean()`	所有值的平均值
4	`median()`	所有值的中位数
5	`mode()`	值的模值
6	`std()`	值的标准偏差
7	`min()`	所有值中的最小值
8	`max()`	所有值中的最大值
9	`abs()`	绝对值
10	`prod()`	数组元素的乘积
11	`cumsum()`	累计总和
12	`cumprod()`	累计乘积

为演示聚合函数功能，首先创建一个`DataFrame`，这里使用`np.nan`对数据引入缺失值。

```
d = {'Name':pd.Series(['Tom','James','Ricky','Vin','Steve','Minsu','Jack']),
   'Age':pd.Series([25,26,np.nan,23,30,29,23]),
   'Rating':pd.Series([4.23,3.24,3.98,np.nan,3.20,4.6,3.8,])}

df = pd.DataFrame(d)
print(df)
#     Name   Age  Rating
# 0    Tom  25.0    4.23
# 1  James  26.0    3.24
# 2  Ricky   NaN    3.98
# 3    Vin  23.0     NaN
# 4  Steve  30.0    3.20
# 5  Minsu  29.0    4.60
# 6   Jack  23.0    3.80
```

任务：统计信息描述函数describe()

> 知识点：
>
> - `describe()`函数对**每列**获得`DataFrame`**统计信息摘要**：平均值，标准差和IQR值。
> - 没有`axis`参数。
> - 函数排除**字符**列，给出**数值**列的统计摘要。

```
print(df.describe())
#              Age    Rating
# count   6.000000  6.000000
# mean   26.000000  3.841667
# std     2.966479  0.551450
# min    23.000000  3.200000
# 25%    23.500000  3.380000
# 50%    25.500000  3.890000
# 75%    28.250000  4.167500
# max    30.000000  4.600000
```

任务：求和函数sum()

> 知识点：
>
> - 对`DataFrame`沿指定`axis`求和，返回`Series`对象。
> - 默认为`axis=0`，沿行方向求和。
>
> - 如果**全部为字符串**时，则执行拼接；
> - 如果**字符串和数值混合**时，则**忽略字符串和缺失值**，求和数值。

```
print(type(df.sum()))
# <class 'pandas.core.series.Series'>

print(df.sum())
# 对Name列字符串拼接，对其他列数值求和
# Name      TomJamesRickyVinSteveMinsuJack
# Age                                  156
# Rating                             23.05
# dtype: object
print(df.sum(axis=1))
# 忽略字符串和缺失值，只对数值类型求和
# 0    29.23
# 1    29.24
# 2     3.98
# 3    23.00
# 4    33.20
# 5    33.60
# 6    26.80
# dtype: float64
```

任务：聚合方向的实际意义分析

> 知识点：
>
> - **列代表数据的某个特征**(沿`axis=0`)
>   - 通常同列数据都是统一量纲的数值类型。对同列的数据聚合是**有实际统计意义**的。
>   - 例如，对所有的Age求和代表总年龄。
> - 但是**行代表一条样本或记录**(沿`axis=1`)。
>   - 通常不同列数据数据类型和量纲都不一样。对同行数据聚合并**无实际意义**。
>   - 例如，对第一行的，Age和Rating求和毫无意义。
> - 这也就是为何`DataFrame`为何**默认**是沿`axis=0`进行聚合。因此，以后例子都**针对列进行数据统计**。

任务：平均函数mean()

> 知识点：返回每列的均值`Series`对象。**忽略字符串和缺失值**

```
print(df.mean())
# Age       25.857143
# Rating     3.658571
# dtype: float64
```

任务：标准差函数std()

> 知识点：返回**每列**的标准差`Series`对象。

```
print(df.mean())
# Age       2.734262
# Rating    0.698628
# dtype: float64
```

任务：累计总和cumsum()

> 知识点：返回**每列**的累积求和`DataFrame`对象。

```
print(df.cumsum())
#                              Name    Age  Rating
# 0                             Tom   25.0    4.23
# 1                        TomJames   51.0    7.47
# 2                   TomJamesRicky    NaN   11.45
# 3                TomJamesRickyVin   74.0     NaN
# 4           TomJamesRickyVinSteve  104.0   14.65
# 5      TomJamesRickyVinSteveMinsu  133.0   19.25
# 6  TomJamesRickyVinSteveMinsuJack  156.0   23.05
```

11.2 字符串处理函数描述

任务：字符串处理函数的基本概述

> 知识点：
>
> - 在`Dataframe`中，通常**行表示数据的样本**，**列表示数据的特征**(反映了数据的某个指标)。
> - `Dataframe`对象的列是由`Series`对象组成。
> - Pandas对数据分析提供了一系列的字符串和文本数据处理函数。 这些函数对Python字符串函数进行了扩展，**可以处理包含缺失值`NaN`数据**。
> - 对于`Dataframe`的字符串`Series`列，Pandas首先将该列`Series`对象转换为`String`对象，然后执行类似Python的字符串处理函数。

下表总结了19个Pandas提供的字符串和文本数据处理函数。很多都是和Python的字符串处理函数类似，只是处理对象变成了`Series`对象或`Index`标签对象。

函数	描述
`lower()`	将`Series/Index`中的字符串转换为小写。
`upper()`	将`Series/Index`中的字符串转换为大写。
`len()`	计算字符串长度。
`strip()`	删除`Series/Index`中的每个字符串两侧的空格(包括换行符)。
`split(' ')`	用给定的模式拆分每个字符串。
`cat(sep=' ')`	使用给定的分隔符连接系列/索引元素。
`get_dummies()`	返回具有`one-hot`编码的`DataFrame`对象。
`contains(pattern)`	如果元素中包含子字符串，则返回每个元素的布尔值`True`，否则为`False`。
`replace(a,b)`	将值`a`替换为值`b`。
`repeat(value)`	重复每个元素指定的次数。
`count(pattern)`	返回模式中每个元素的出现总数。
`startswith(pattern)`	如果系列/索引中的元素以模式开始，则返回`true`。
`endswith(pattern)`	如果系列/索引中的元素以模式结束，则返回`true`。
`find(pattern)`	返回模式第一次出现的位置。
`findall(pattern)`	返回模式的所有出现的列表。
`swapcase`	变换字母大小写。
`islower()`	检查系列/索引中每个字符串中的所有字符是否小写，返回布尔值
`isupper()`	检查系列/索引中每个字符串中的所有字符是否大写，返回布尔值
`isnumeric()`	检查系列/索引中每个字符串中的所有字符是否为数字，返回布尔值。

任务：字符串处理函数调用方式

> 知识点：
>
> - 首先通过`str`属性将`Series`或`Index`对象转换为`String`对象。
> - 然后，在调用相应的字符串方法。
> - 调用方式为`s.str.字符串函数()`。

11.3 字符串处理的基本函数

任务：创建`Series`对象

由上可知，`Dataframe`对象的字符串处理本质上是对`Series`列对象处理。因此，为演示字符串函数功能，首先创建一个`Series`对象，并使用`np.nan`对数据引入缺失值。

```
s = pd.Series(['Tom', 'William Rick', 'John', 'Alber@t', np.nan,
'1234','SteveMinsu'])
print(s)
# 0             Tom
# 1    William Rick
# 2            John
# 3         Alber@t
# 4             NaN
# 5            1234
# 6      SteveMinsu
# dtype: object
```

任务：lower()和upper()大小写转换函数

> 知识点：
>
> - 将`Series/Index`中的字符串转换为大写或小写。
> - 调用方式`s.str.lower()`或`s.str.upper()`。

```
# 转换为小写
print(s.str.lower())
# 0             tom
# 1    william rick
# 2            john
# 3         alber@t
# 4             NaN
# 5            1234
# 6      steveminsu
# dtype: object
# 转换为大写
print(s.str.upper())
# 0             TOM
# 1    WILLIAM RICK
# 2            JOHN
# 3         ALBER@T
# 4             NaN
# 5            1234
# 6      STEVEMINSU
# dtype: object
```

任务：len()获得字符串长度函数

> 知识点：
>
> - 计算`Series/Index`中的每个字符串长度。
> - 调用方式`s.str.len()`。

```
# 计算每个字符串长度
print(s.str.len())
# 0     3.0
# 1    12.0
# 2     4.0
# 3     7.0
# 4     NaN
# 5     4.0
# 6    10.0
# dtype: float64
```

任务：strip()空白字符删除函数

> 知识点：
>
> - 删除`Series/Index`中的字符串**两侧**的空格(包括换行符)。
> - 调用方式`s.str.strip()`。

```
s = pd.Series(['Tom    ', ' William Rick', 'John   ', 'Alber@t'])
print(s)
# 0          Tom
# 1     William Rick
# 2          John
# 3          Alber@t
# dtype: object

# 仔细观察并对比结果
print(s.str.strip())
# 0             Tom
# 1    William Rick
# 2            John
# 3         Alber@t
# dtype: object
```

任务(重要)：split(pattern)拆分字符串函数

> 知识点：
>
> - 将`Series/Index`中的字符串按`pattern`**分隔符**拆分，返回**嵌套列表**的`Series`对象。
> - 字符串拆分在**数据爬虫**中应用很广。
> - 调用方式`s.str.split(pattern)`。

```
s = pd.Series(['Tom Tom OK', 'William Rick', 'John CC', 'NN Alber@t'])
print(s)
# 0      Tom Tom OK
# 1    William Rick
# 2         John CC
# 3      NN Alber@t
# dtype: object

# 分隔符pattern设置为空格
# 返回的Series对象每个元素为列表
```

```
print(s.str.split(" "))
# 0     [Tom, Tom, OK]
# 1    [William, Rick]
# 2         [John, CC]
# 3      [NN, Alber@t]
# dtype: object
```

任务(重要)：cat(sep=pattern)拼接字符串函数

> 知识点：
>
> - 使用指定**分隔符**关键字参数`sep`拼接`Series/Index`对象元素，返回`String`字符串对象。
> - 字符串拆分在**数据爬虫**中应用很广。
> - 调用方式`s.str.cat(sep=pattern)`。

```
s = pd.Series(['Tom ', ' William Rick', 'John', 'Alber@t'])
print(s)
# 0             Tom
# 1     William Rick
# 2             John
# 3          Alber@t
# dtype: object

# 分隔符' <=> '拼接字符串
print (s.str.cat(sep=' <=> '))
# Tom  <=>  William Rick <=> John <=> Alber@t

# 返回为String对象
print (type(s.str.cat(sep=' <=> ')))
# <class 'str'>
```

任务：islower()、isupper()、isnumeric()检验函数

> 知识点：
>
> - `islower()`：检查`Series`对象元素中的所有字符是否**全小写**。
> - `isupper()`：检查`Series`对象元素中的所有字符是否**全大写**。
> - `isnumeric()`：检查`Series`对象元素中的所有字符是否**全为数值**。

```
s = pd.Series(['TOM1', '1199','William Rick', 'john2', 'Alber@t'])
print(s)
# 0            TOM1
# 1            1199
# 2    William Rick
# 3           john2
# 4         Alber@t

# 忽略数值，只检查字符是否全小写。
print(s.str.islower())
# 0    False
# 1    False
# 2    False
```

```
# 3     True
# 4    False
# dtype: bool

# 忽略数值，只检查字符是否全大写。
print(s.str.isupper())
# 0     True
# 1    False
# 2    False
# 3    False
# 4    False
# dtype: bool

# 是否全数值
print(s.str.isnumeric())
# 0    False
# 1     True
# 2    False
# 3    False
# 4    False
# dtype: bool
```

任务：swapcase()字符大小写转换函数

> 知识点：
>
> - 返回字符大小写转换的`Series`对象。`swap`代表交换。
> - 调用方式`s.str.swapcase()`。

```
s = pd.Series(['T omo ', ' Ri ck', 'aJo honT', 'Alber@t'])
print(s)
# 0      T omo
# 1       Ri ck
# 2    aJo honT
# 3     Alber@t
# dtype: object

print(s.str.swapcase())
# 0      t OMO
# 1       rI CK
# 2    AjO HONt
# 3     aLBER@T
# dtype: object
```

11.4 one-hot编码函数

任务(重要)：get_dummies()离散型特征提取函数

> 背景介绍：
>
> - 在拿到的数据里，经常有**离散型数据**或类别型数据，如下：
>   - 球鞋品牌：Nike、adidas、 Vans、PUMA、CONVERSE。
>   - 性别：男、女。
>   - 颜色：红、黄、蓝、绿。

> - 如何将离散型数据转换为计算机擅长处理的数值类型数据呢？将离散型数据使用`one-hot`编码是常用的方法。

> `one-hot`编码的基本思想：
>
> - 将离散型数据的每一种**取值**都看成一种**状态**，使用二进制表示数据的状态。若数据中有N个不相同的取值，那么可以使用N个位将数据抽象为N种不同状态。
> - `one-hot`编码保证每一个值只会有一种"**激活态**"。即N种状态中只有一个状态位值为1，其他状态位都是0。
>
> 例如，颜色：红、黄、蓝、绿。使用`one-hot`编码对应有
>
> - 红：`[0,0,0,1]`；黄：`[0,0,1,0]`；蓝：`[0,1,0,0]`；绿：`[1,0,0,0]`。

> 知识点：
>
> - `get_dummies()`离散型特征提取函数，返回具有`one-hot`编码的`DataFrame`对象。
> - 在机器学习领域，`one-hot`编码应用非常广泛。
> - 调用方式`s.str.get_dummies()`。

`Series`对象的`one-hot`编码。

```
s = pd.Series(["男","女","女","男"])
print(s)
# 0    男
# 1    女
# 2    女
# 3    男
# dtype: object
print(s.str.get_dummies())
#    女  男
# 0  0  1
# 1  1  0
# 2  1  0
# 3  0  1

print(type(s.str.get_dummies())) # 返回DataFrame类型
# <class 'pandas.core.frame.DataFrame'>
```

`DataFrame`对象的`one-hot`编码。

```
df = pd.DataFrame({"学号":[1001,1002,1003,1004],
                   "性别":["男","女","女","男"],
                   "学历":["本科","硕士","专科","本科"]})
print(df)
#     学号 性别 学历
# 0  1001  男  本科
# 1  1002  女  硕士
# 2  1003  女  专科
# 3  1004  男  本科
print(pd.get_dummies(df))
```

	学号	性别_女	性别_男	学历_专科	学历_本科	学历_硕士
0	1001	0	1	0	1	0
1	1002	1	0	0	0	1
2	1003	1	0	1	0	0
3	1004	0	1	0	1	0

11.5 （重要）字符串匹配函数

字符串匹配函数常用于网络爬虫，进阶内容为**正则表达式。**

任务：contains(pattern)检查字符串函数

> 知识点：
>
> - 检查`Series/Index`对象中的元素是否包含`pattern`字符串。
> - `pattern`字符串指的是匹配的**子字符串。**
> - 返回为`Series`对象，其中元素为相应的布尔值`True`或`False`。
> - 调用方式`s.str.contains(pattern)`。

```
s = pd.Series(['Tom ', ' William Rick', 'Jo hn', 'Alber@t'])
print(s)
# 0             Tom
# 1     William Rick
# 2           Jo hn
# 3         Alber@t
# dtype: object

# 查看是否包含' '的字符串
print(s.str.contains(' '))
# 0     True
# 1     True
# 2     True
# 3    False
# dtype: bool
```

任务：replace(a,b)字符串替换函数

> 知识点：
>
> - 将字符串`a`替换为`b`。
> - 调用方式`s.str.replace(a,b)`。

```
s = pd.Series(['Tom   ', ' William Rick', 'Jo hn', 'Alber@t'])
print(s)
# 0           Tom
# 1     William Rick
# 2           Jo hn
# 3          Alber@t
# dtype: object

# 将' '字符串替换为'#'
print(s.str.replace(' ','#'))
# 0           Tom###
# 1    #William#Rick
# 2           Jo#hn
# 3          Alber@t
# dtype: object
```

任务：repeat(value)重复字符串函数

> 知识点：
>
> - 对`Series/Index`对象中的每个元素重复`value`次。
> - 调用方式`s.str.repeat(value)`。

```
s = pd.Series(['Tom ', ' Rick', 'Jo hn', 'Alber@t'])
print(s)
# 0       Tom
# 1       Rick
# 2      Jo hn
# 3    Alber@t

# 重复3次
print(s.str.repeat(3))
# 0             Tom Tom Tom
# 1           Rick Rick Rick
# 2          Jo hnJo hnJo hn
# 3    Alber@tAlber@tAlber@t
# dtype: object
```

任务：count(pattern)重复字符串函数

> 知识点：
>
> - `Series/Index`对象的元素出现`pattern`的次数。
> - 调用方式`s.str.count(pattern)`。

```
s = pd.Series(['T om ', ' Ri ck', 'Jo hn', 'Alber@t'])
```

```
print(s)
# 0      T om
# 1      Ri ck
# 2      Jo hn
# 3    Alber@t
# dtype: object

# 统计' '出现的次数
print(s.str.count(' '))
# 0    2
# 1    2
# 2    1
# 3    0
# dtype: int64
```

任务：startswith(pattern)和endswith(pattern)字符串检验函数

> 知识点：
>
> - 如果`Series/Index`对象的元素以`pattern`开始或结束，则返回`true`。
> - 区分大小写。
> - 调用方式`s.str.startswith(pattern)`或`s.str.endswith(pattern)`。

```
s = pd.Series(['T om ', ' Ri ck', 'aJo hnT', 'Alber@t'])
print(s)
# 0      T om
# 1      Ri ck
# 2    aJo hnT
# 3    Alber@t
# dtype: object

# 检验是否以"A"字符串开始，区分大小写。
print(s.str.startswith('A'))
# 0    False
# 1    False
# 2    False
# 3     True
# dtype: bool

print(s.str.endswith(' '))
# 0     True
# 1    False
# 2    False
# 3    False
# dtype: bool
```

任务：find(pattern)和findall(pattern)查找字符串函数

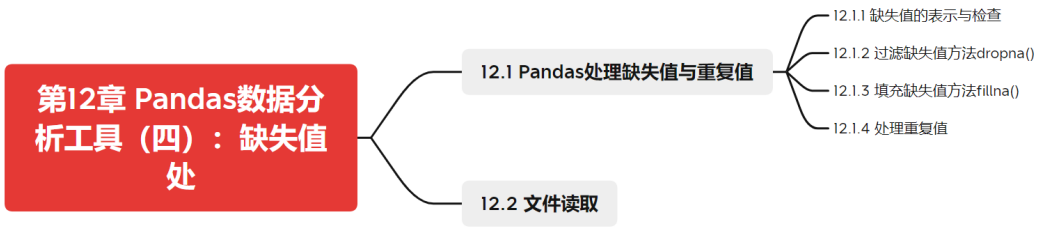
> 知识点：
>
> - `find(pattern)`：返回`pattern`第一次出现的**位置**。如果没有则返回`-1`。
> - `findall(pattern)`：返回`pattern`所有出现的**列表**。如果没有，则返回**空列表**。
> - 常用于网络爬虫，区分大小写，字符串的正则表达式也有该功能。

- 调用方式`s.str.find(pattern)`或`s.str.findall(pattern)`。

```
s = pd.Series(['T omo ', ' Ri ck', 'aJo honT', 'Alber@t'])
print(s)
# 0      T omo
# 1       Ri ck
# 2    aJo honT
# 3     Alber@t
# dtype: object

print(s.str.find('o'))
# 0    2
# 1   -1
# 2    2
# 3   -1
# dtype: int64

print(s.str.findall('o'))
# 0    [o, o]
# 1        []
# 2    [o, o]
# 3        []
# dtype: object
```

第12章 Pandas数据分析工具（四）：缺失值处理与文件处理

思维导图



12.1 Pandas处理缺失值与重复值

12.1.1 缺失值的表示与检查

任务：需求分析

> 知识点：
>
> - **缺失值**是指数据集中的某些样本存在遗漏的特征值。缺失值的存在会影响到数据分析和挖掘的结果。
> - 造成数据缺失的主要原因：数据本身的特性、采样的失误、用户输入的空值、数据处理的异常0/0、对负数开方等。

> 知识点：当遇到缺失值时，通常采三种方法处置：**删除法**，**替换法**和**插补法**。
>
> - **删除法**：对有缺失值的行和列简单的删除。当确实的观测**比例非常低**时，如5%以内，可以直接删除这些缺失的变量。
> - **替换法**：**用某些值直接替换缺失值**。例如，对连续变量，可以使用均值或中位数替换；对离散型变量，可以使用众数替换。
> - **插补法**：根据缺失值相近的数据来预测缺失值。常用的插补法有回归插补法，K近邻法，拉格朗日插补法等。

在本节，将重点介绍Pandas提供的缺失值处理函数。

任务：缺失值的表示

> 知识点：
>
> - Pandas使用浮点值`NaN`(Not a Number)表示浮点数和非浮点数组中的缺失值。
> - Python内置`None`值也会被当作是缺失值。
> - Pandas和NumPy中，使用`np.nan`来表示缺失值。

```
ser = pd.Series([0, 1, 2, np.NaN, 9],
                index=['red', 'blue', 'yellow', 'white', 'green'])
print(ser)
# red       0.0
# blue      1.0
# yellow    2.0
# white     NaN
# green     9.0
# dtype: float64
```

> 知识点：如果变量为`None`，Pandas将缺失值统一作为`NaN`来处理。

```
ser['red'] = None
print(ser)
# red       NaN
# blue      1.0
# yellow    2.0
# white     NaN
# green     9.0
# dtype: float64
```

任务：检查缺失值函数

> 知识点：
>
> - 使用`isnull()`和`notnull()`检测数据中否包含缺失值。
> - 它们也是`Series`和`DataFrame`对象的方法。

例子：`Series`对象的缺失值判断。

```
s = pd.Series(["a","b",np.nan,"c",None])
print(s)
```

```
# 0       a
# 1       b
# 2     NaN
# 3       c
# 4    None
# dtype: object

# 判断缺失值,如果是则返回True,否则返回False
print(s.isnull())
# 0    False
# 1    False
# 2     True
# 3    False
# 4     True
# dtype: bool

# 掩码索引，输出缺失值的索引和值
print(s[s.isnull()])
# 2     NaN
# 4    None
# dtype: object
```

例子：`DataFrame`对象的缺失值判断。

```
a = [[1,np.nan,2],[3,4,None]]
data = pd.DataFrame(a)
#DataFrame的None值变成了NaN
print(data)
#    0    1    2
# 0  1  NaN  2.0
# 1  3  4.0  NaN

print(data.isnull())
#        0      1      2
# 0  False   True  False
# 1  False  False   True

# 全都是NaN，很奇怪的结果，请看下面知识点分析
print(data[data.isnull()])
#     0   1   2
# 0 NaN NaN NaN
# 1 NaN NaN NaN
```

> 知识点：
>
> - 在使用`Series`和`DataFrame`的时候，如果其中有值为`None`，`Series`会输出`None`，而`DataFrame`会输出`NaN`，但是对缺失值判断没有影响。
> - **注意**：`DataFrame`使用`isnull()`方法在输出缺失值的时候全为`NaN`，因为`DataFrame`对于`False`对应的位置，也会使用`NaN`代替输出值，而`Series`对于`False`对应的位置是没有输出值的。

12.1.2 过滤缺失值方法dropna()

任务：dropna()方法的定义

> 知识点：`dropna(axis=0, how='any',inplace=False,thresh=None)`方法。
>
> - 描述：`Series`和`DataFrame`对象的方法，删除缺失值。
> - `axis`: 删除行还是列，0为`index`行，1为`columns`列，默认为0。
> - `how`: 删除方式。
>   - 如果为`any`则列或行中只要有缺失值就被删除。
>   - 如果为`all`则列或行中所有值都为缺失值才删除。
> - `inplace`: 如果为`True`则修改原数据，否则返回数据的`copy`。
> - `thresh`：删除只有大于`thresh`的缺失值行。

任务：Series的缺失值过滤

> 知识点：`dropna()`默认是新建一个`Series`对象，不影响原`Series`数据。

```
s = pd.Series(["a","b",np.nan,"c",None])
# 通过使用notnull()方法来获取非缺失数据
print(s[s.notnull()])
# 0    a
# 1    b
# 3    c
# dtype: object

#使用dropna()方法删除缺失数据,返回一个删除后的Series
print(s.dropna())
# 0    a
# 1    b
# 3    c
# dtype: object

# dropna()默认是新建一个Series对象，不影响原Series数据。
print(s)
# 0       a
# 1       b
# 2     NaN
# 3       c
# 4    None
# dtype: object
```

> 知识点：`inplace`参数为`True`，在原`Series`上进行删除，不会返回新的`Series`。

```
s = pd.Series(["a","b",np.nan,"c",None])
# 通过使用notnull()方法来获取非缺失数据
print(s[s.notnull()])
# 0    a
# 1    b
# 3    c
```

```
# dtype: object

# 通过设置inplace参数为True,在原Series上进行删除,不会返回新的Series
print(s.dropna(inplace=True))
# None (返回为空)

print(s)
# 0    a
# 1    b
# 3    c
# dtype: object
```

任务：DataFrame的缺失值过滤

`DataFrame`删除缺失值相对于`Series`而言就要复杂一些，有时只要包含缺失值的行或列都被删除；有时只有当整行或整列全为缺失值的时才删除。Pandas为上述两种情况提供了相对应的处理方法。

> 知识点：`how`关键字参数默认为`any`，删除含有`NaN`的行和列。

```
a = [[1, np.nan, 2],[9,None,np.nan],[3, 4, None],[5,6,7]]
data = pd.DataFrame(a)
print(data)
#    0    1    2
# 0  1  NaN  2.0
# 1  9  NaN  NaN
# 2  3  4.0  NaN
# 3  5  6.0  7.0

# 使用dropna方法删除含有缺失值的行，默认是行
print(data.dropna())
#    0    1    2
# 3  5  6.0  7.0

# 删除含有缺失值的列
print(data.dropna(axis=1))
#    0
# 0  1
# 1  9
# 2  3
# 3  5
```

> 知识点：删除全为`NaN`的行和列。设置`how`关键字参数为`all`。

```
a = [[1, np.nan, 2],[np.nan,None,np.nan],[3, None, None],[5,None,7]]
data = pd.DataFrame(a)
print(data)
#      0   1    2
# 0  1.0 NaN  2.0
# 1  NaN NaN  NaN
# 2  3.0 NaN  NaN
# 3  5.0 NaN  7.0

# 当行全为NaN的时候,才删除
```

```
print(data.dropna(how="all"))
#      0   1    2
# 0  1.0 NaN  2.0
# 2  3.0 NaN  NaN
# 3  5.0 NaN  7.0

# 当列全为NaN的时候，才删除
print(data.dropna(how="all",axis=1))
#      0    2
# 0  1.0  2.0
# 1  NaN  NaN
# 2  3.0  NaN
# 3  5.0  7.0
```

> 知识点：`DataFrame`的`dropna()`方法的`inplace`参数与`Series`一样，设置为`True`覆盖原数据。

```
a = [[1, np.nan, 2],[np.nan,None,np.nan],[3, None, None],[5,None,7]]
data = pd.DataFrame(a)
print(data)
#      0   1    2
# 0  1.0 NaN  2.0
# 1  NaN NaN  NaN
# 2  3.0 NaN  NaN
# 3  5.0 NaN  7.0

print(data.dropna(how="all", inplace=True))
# None (返回为空)

print(data)
#      0   1    2
# 0  1.0 NaN  2.0
# 2  3.0 NaN  NaN
# 3  5.0 NaN  7.0
```

> 知识点：
>
> - 通过`thresh`关键字参数控制删除`NaN`行的阈值。
> - 只要该行大于`thresh`时才会被删除。
> - 只对行有效，列无效。

```
a = [[1, np.nan, 2],[np.nan,None,np.nan],[3, None, None],[5,None,7]]
data = pd.DataFrame(a)
print(data)
#      0   1    2
# 0  1.0 NaN  2.0
# 1  NaN NaN  NaN
# 2  3.0 NaN  NaN
# 3  5.0 NaN  7.0

# 当行全为NaN的时候,才删除
print(data.dropna(how="all"))
#      0   1    2
```

```
# 0  1.0 NaN  2.0
# 2  3.0 NaN  NaN
# 3  5.0 NaN  7.0

# 通过thresh参数来控制只要大于2个NaN，该行数据就被删除，删除列的时候thresh参数无效。
print(data.dropna(how="all",thresh=2))
#      0   1    2
# 0  1.0 NaN  2.0
# 3  5.0 NaN  7.0
```

12.1.3 填充缺失值方法fillna()

> 知识点：
>
> - 数据都是宝贵的，如果简单通过`dropna()`丢弃数据，会造成很大的浪费。
> - 因为数据越多，对于数据分析得到的结果越可靠。
> - 因此，对缺失值数据使用合适的值来填充可以较好的缓解这问题。

下面介绍使用Pandas的`fillna()`方法来填充缺失数据。

> 知识点：`fillna(value=None,method=None,axis=None,inplace=False)`
>
> - 描述：填充缺失值。
> - `value`：用于填充的值，可为单个值，或者字典（key是列名，value是值）
> - `method`：填充值方式。前向和后向填充。
>   - `ffill`：使用前一个不为空的值填充`forword fill`；
>   - `bfill`：使用后一个不为空的值填充`backword fill`。
> - `axis`：行还是列，0为`index`行，1为`columns`列，默认为0。
> - `inplace`：如果为`True`则修改原数据，否则返回数据的`copy`。

> 知识点：通过`value`参数，指定特殊值来填充缺失值。

```
df = pd.DataFrame([[np.nan, 2, np.nan, 0],
                   [3, 4, np.nan, 1],
                   [np.nan, np.nan, np.nan, 5],
                   [np.nan, 3, np.nan, 4]],
                  columns=list('ABCD'))
print(df)
#      A    B   C  D
# 0  NaN  2.0 NaN  0
# 1  3.0  4.0 NaN  1
# 2  NaN  NaN NaN  5
# 3  NaN  3.0 NaN  4

# 用0替换所有的NaN
print(df.fillna(0))
#     A   B   C   D
# 0   0.0 2.0 0.0 0
# 1   3.0 4.0 0.0 1
# 2   0.0 0.0 0.0 5
# 3   0.0 3.0 0.0 4
```

> 知识点：为`value`参数传递字典，来指定不同列使用不同的填充值。

```
df = pd.DataFrame([[np.nan, 2, np.nan, 0],
                   [3, 4, np.nan, 1],
                   [np.nan, np.nan, np.nan, 5],
                   [np.nan, 3, np.nan, 4]],
                  columns=list('ABCD'))
print(df)
#      A    B   C  D
# 0  NaN  2.0 NaN  0
# 1  3.0  4.0 NaN  1
# 2  NaN  NaN NaN  5
# 3  NaN  3.0 NaN  4

# 为value参数传递字典，来指定不同列使用不同的填充值。
values = {'A': 0, 'B': 1, 'C': 2, 'D': 3}
print(df.fillna(value=values))
#     A   B   C   D
# 0   0.0 2.0 2.0 0
# 1   3.0 4.0 2.0 1
# 2   0.0 1.0 2.0 5
# 3   0.0 3.0 2.0 4
```

> 知识点：`method`参数指定前向填充`ffill`和后向填充`bfill`。

```
df = pd.DataFrame([[np.nan, 2, np.nan, 0],
                   [3, 4, np.nan, 1],
                   [np.nan, np.nan, np.nan, 5],
                   [np.nan, 3, np.nan, 4]],
                  columns=list('ABCD'))
print(df)
#      A    B   C  D
# 0  NaN  2.0 NaN  0
# 1  3.0  4.0 NaN  1
# 2  NaN  NaN NaN  5
# 3  NaN  3.0 NaN  4

# 使用前面的值来填充后面的缺失值
print(df.fillna(method='ffill'))
#     A   B   C   D
# 0   NaN 2.0 NaN 0
# 1   3.0 4.0 NaN 1
# 2   3.0 4.0 NaN 5
# 3   3.0 3.0 NaN 4

# 使用后面的值来填充前面的缺失值
print(df.fillna(method='bfill'))
#      A    B   C  D
# 0  3.0  2.0 NaN  0
# 1  3.0  4.0 NaN  1
# 2  NaN  3.0 NaN  5
# 3  NaN  3.0 NaN  4
```

> 知识点：通过嵌套聚合函数，使用列的平均值来填充缺失值。

```
a = [[1, 2, 2],[3,None,6],[3, 7, None],[5,None,7]]
data = pd.DataFrame(a)
print(data)
#    0    1    2
# 0  1  2.0  2.0
# 1  3  NaN  6.0
# 2  3  7.0  NaN
# 3  5  NaN  7.0

print(data.fillna(data.mean()))
#    0    1    2
# 0  1  2.0  2.0
# 1  3  4.5  6.0
# 2  3  7.0  5.0
# 3  5  4.5  7.0
```

12.1.4 处理重复值

在实际开发中，经常需要对数据进行去重复值操作。根据需求不同，通常会分为两种情况，一种是去除所有重复的行数据，另一种是去除指定列的重复的行数据。

本节将学习重复值处理方法`drop_duplicates()`和`duplicated()`使用。

任务：重复值删除方法drop_duplicates()的定义

> 知识点：
>
> - `drop_duplicates(subset=None, keep='first', inplace=False)`
> - 描述：`Series`或`DataFrame`的方法，去除特定列下面的**重复行**，返回`Series`或`DataFrame`数据。
> - `subset`：选定重复值检测的关键列。默认值为`subset=None`表示检测所有列。
> - `keep`：取值`first`、`last`、`False`，默认值为`first`。分别表示保留第一次、保留最后次、去除所有出现的重复行。
> - `inplace`：取值为`True`或`False`，默认值`False`。表示是否直接在原`DataFrame`上删除重复项表示生成一个副本。
> - 注意：是删除重复值所在的整行数据，而非单独某个重复值。

任务：仅去除重复项

下面以`DataFrame`为例。创建`DataFrame`数据。

```
df1 = pd.DataFrame({'a': [1, 1, 2, 2, 5],
                    'b': [1, 1, 6, 4, 3],
                    'c': [1, 1, 2, 2, 9]})
print(df1)
#    a  b  c
# 0  1  1  1
# 1  1  1  1
# 2  2  6  2
# 3  2  4  2
# 4  5  3  9
```

`drop_duplicates()` 默认参数：检测所有列重复值，保留重复值第一次出现的行数据，并且返回一个新的 `DataFrame` 数据。

```
# 以a、b、c 3列为关键列，删除完全相同的行数据
# 第1和0条行数据，关于a、b、c列，内容相同。
print(df1.drop_duplicates())
#    a  b  c
# 0  1  1  1
# 2  2  6  2
# 3  2  4  2
# 4  5  3  9
```

任务：删除指定关键列的重复项

`drop_duplicates()` 指定 `'a'` 和 `'b'` 列的重复值，保留重复值第一次出现的行数据，并且返回一个新的 `DataFrame` 数据。

```
# 以a、c 2列为关键列
# 第1和0条、第2和3条行数据，关于a、c列，内容相同。
print(df1.drop_duplicates(['a','c'], keep='first',inplace=False))
#    a  b  c
# 0  1  1  1
# 2  2  6  2
# 4  5  3  9
```

任务：inplace=True时直接修改原数据

`drop_duplicates()` 默认参数：检测所有列重复值，保留重复值第一次出现的行数据，并且返回一个新的 `DataFrame` 数据。

```
# 注意这里因为inplace为Ture，直接在原数据上修改，方法返回为空。
# 以a、b 2列为关键列。
# 第1和0条，关于a、c列，内容相同。
print(df1.drop_duplicates(['a','b'], keep='first',inplace=True))
# None
print(df1)
#    a  b  c
# 0  1  1  1
# 1  1  4  1
# 2  3  6  3
# 3  2  4  9
```

任务：找出所有的重复项数据

`df.index.tolist()`：将DataFrame的行索引转换为列表。

```
df1 = pd.DataFrame({'a': [1, 1, 2, 2, 5],
                    'b': [1, 1, 6, 4, 3],
                    'c': [1, 1, 2, 2, 9]})
```

```
print('原始数据：\n',df1)
#     a  b  c
# 0  1  1  1
# 1  1  1  1
# 2  2  6  2
# 3  2  4  2
# 4  5  3  9

print('去掉重复行后：\n', df1.drop_duplicates(['a','c']))
#     a  b  c
# 0  1  1  1
# 2  2  6  2
# 4  5  3  9

# 获取非重复项索引
drop_index = df1.drop_duplicates(['a','c']).index.tolist()

# 使用花式索引，丢弃drop_index索引的行数据(非重复项)
# 被删除的重复项数据
print('去掉的重复行是：\n',df1.drop(drop_index))
#     a  b  c
# 1  1  1  1
# 3  2  4  2
```

任务：获得重复值检测掩码数组

> 知识点：
>
> - `duplicated(subset=None, keep='first')`
> - 描述：`Series`或`DataFrame`的方法，获得特定列下面的行数据是否为**重复行**，返回掩码数组的`Series`对象。
> - `subset`：选定重复值检测的关键列。默认值为`subset=None`表示检测所有列。
> - `keep`： 取值`first`、`last`、`False`，默认值为`first`。分别表示保留第一次、保留最后次、去除所有出现的重复行。

```
df1 = pd.DataFrame({'a': [1, 1, 2, 2, 5],
                    'b': [1, 1, 6, 4, 3],
                    'c': [1, 1, 2, 2, 9]})
print(df1)
#    a  b  c
# 0  1  1  1
# 1  1  1  1
# 2  2  6  2
# 3  2  4  2
# 4  5  3  9

# 检测'a'和'c'，获取重复值检测
print(df1.duplicated(['a','c']))
# 0    False
# 1     True
# 2    False
# 3     True
# 4    False
# dtype: bool
```

```
# 使用掩码索引，获得重复值行数据
print(df1.loc[dup])
```

12.2 文件读取