第二十二章 Pandas 数据分析库

Pandas 是基于 Numpy 的一套数据分析工具，该工具是为了解决数据分析任务而创建的。Pandas 纳入了大量标准的数据模型，提供了高效地操作大型数据集所需的工具。Pandas 提供了大量能使我们快速便捷地处理数据的函数和方法。它是使 Python 成为强大而高效的数据分析环境的重要因素之一。

通过阅读本章，您可以：

- 掌握 Anaconda 环境的安装及使用
- 了解什么是 Pandas
- 掌握 Series 对象基本操作
- 掌握 DataFrame 对象的基本操作
- 掌握缺值处理
- 掌握 Series 对象和 DataFrame 的拼接
- 掌握 merge 的使用

22.1 Pandas 开发环境搭建

Pandas 是第三方程序库，所以在使用 Pandas 之前必须安装 Pandas 库。但是如果使用 Anaconda Python 开发环境，那么 Pandas 已经集成到 Anaconda 环境中，不需要再安装。

22.1.1 Anaconda 下载和安装

Anaconda 已经自动的安装了 Jupter Notebook 及其他工具，还有 python 中超过 180 个科学包及其依赖项。现在就看一下 Anaconda 的具体安装流程。

1) 进入官网（https://www.anaconda.com/products/individual），单击 Download 按钮，如图 22-1 所示。

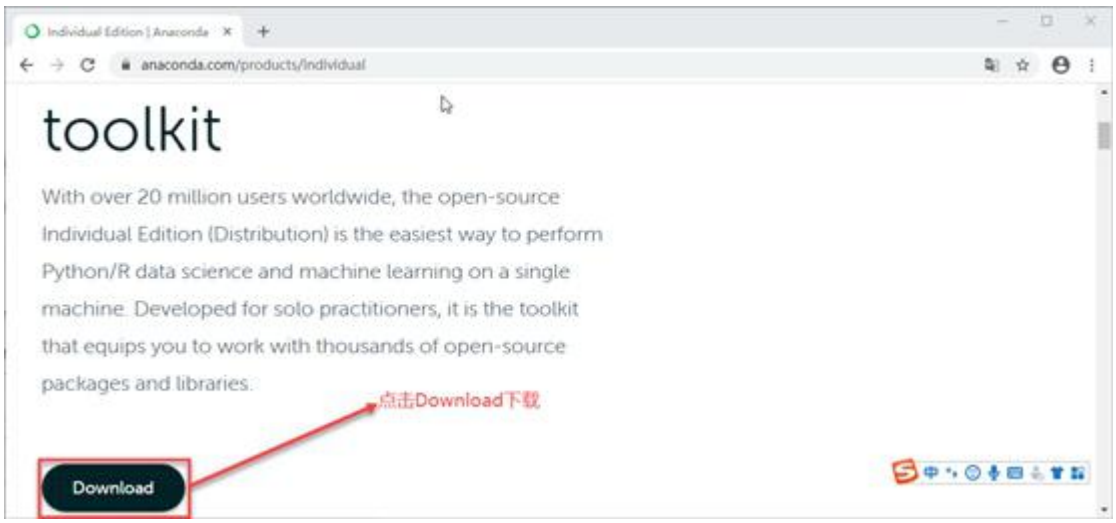

图 22-1 Anaconda 官网

2) 根据电脑系统类型及 Python 环境下载对应版本，因为在 2020 年之后官方就不再支持 Python2 了，所以建议大家选择 Python3，本书的代码也是基于 Python3，然后根据电脑的操作系统位数（32Bit/64Bit）选择对应版本，如图 22-2 所示。

图 22-2 Anaconda 下载

3) 下载后保存到电脑里，双击安装包打开后进行安装，如图 22-3、22-4、22-5 依次单击相应按钮。

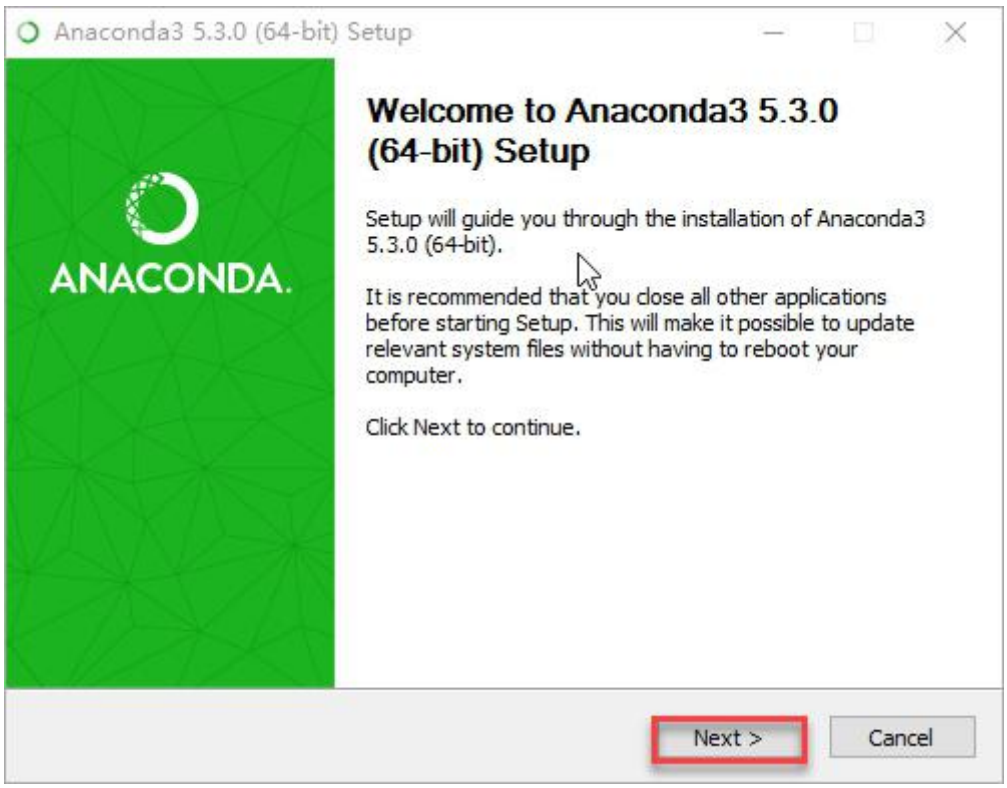

图 22-3 双击安装文件

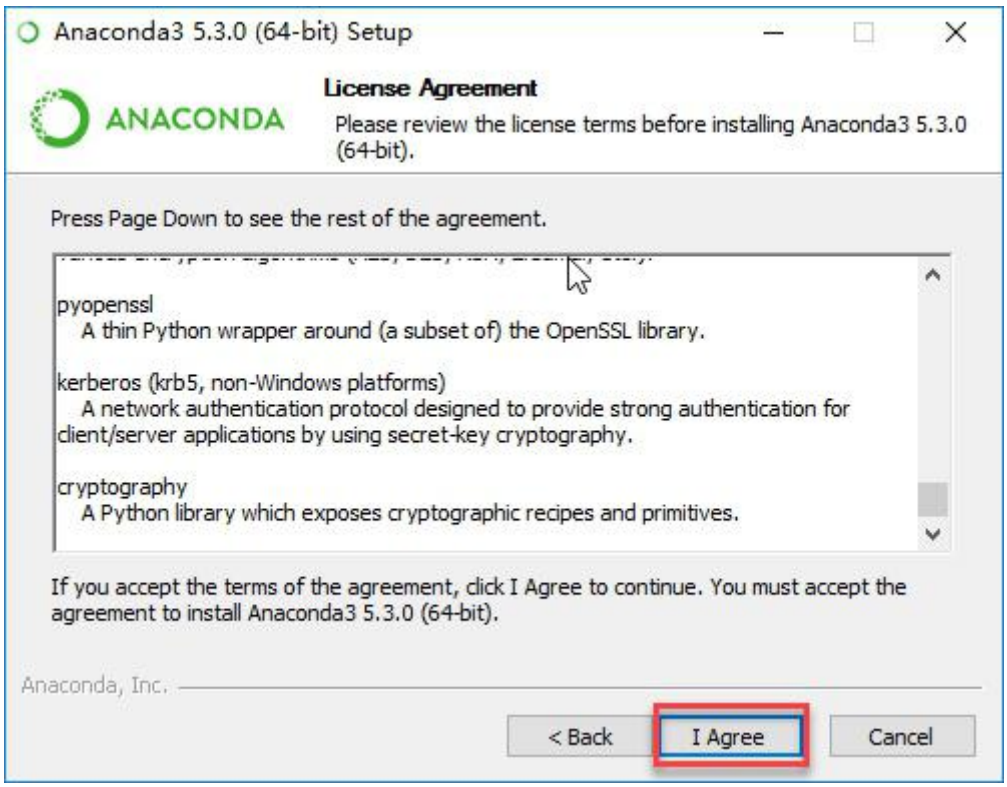

图 22-4 同意协议

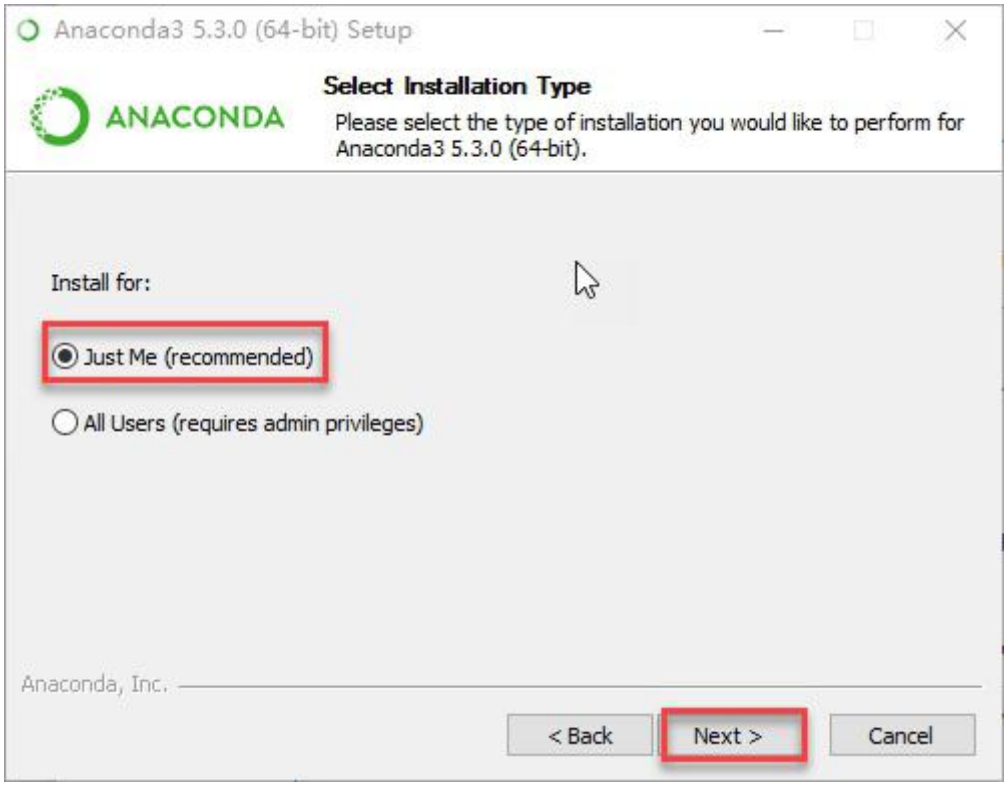

图 22-5 选择用户

4) 选择安装路径，不需要添加环境变量，然后点击 Next 按钮，并在弹出的对话框中勾选相应选项即可。

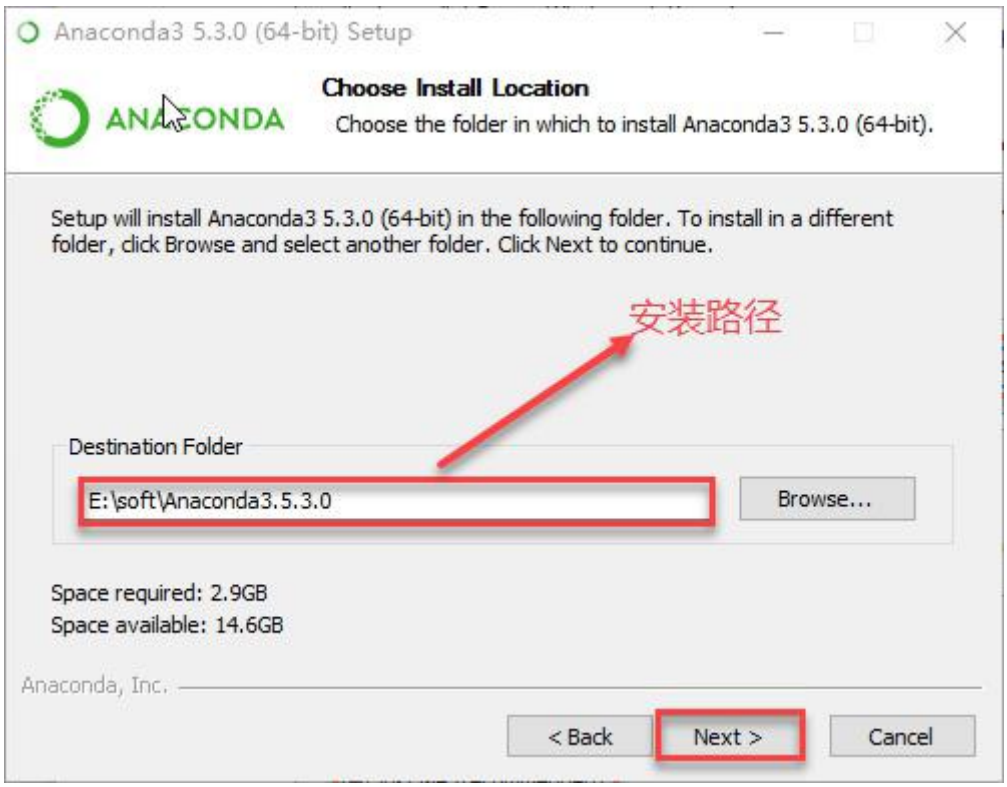

图 22-6 选择安装路径

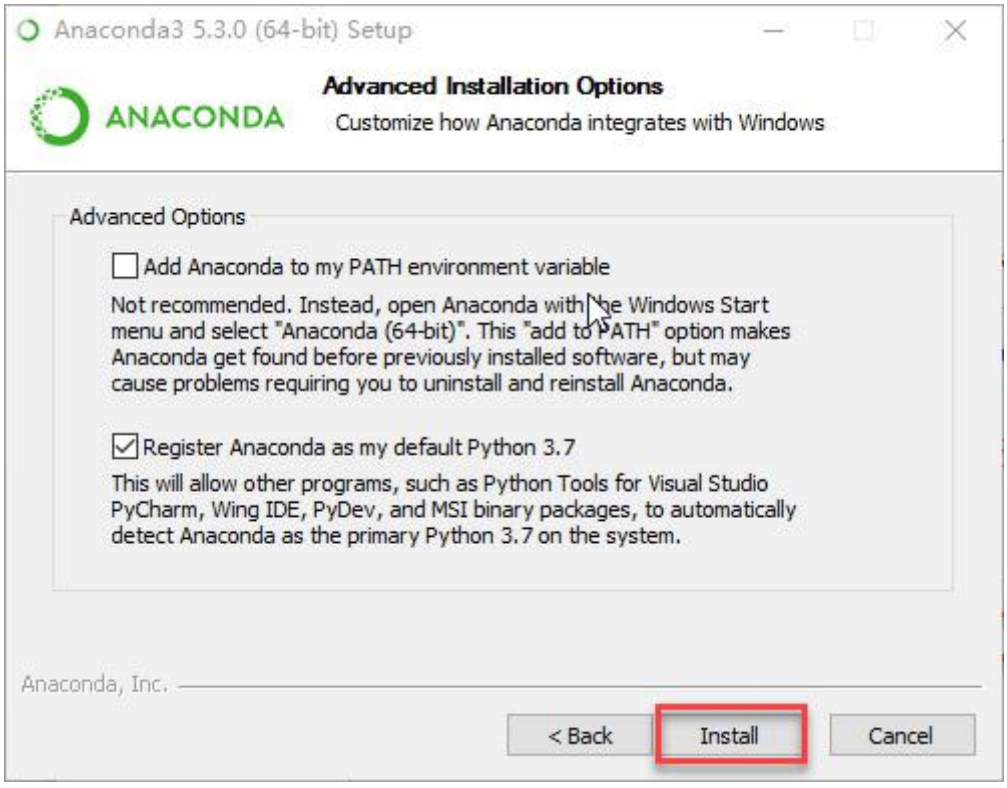

图 22-7 不添加环境变量

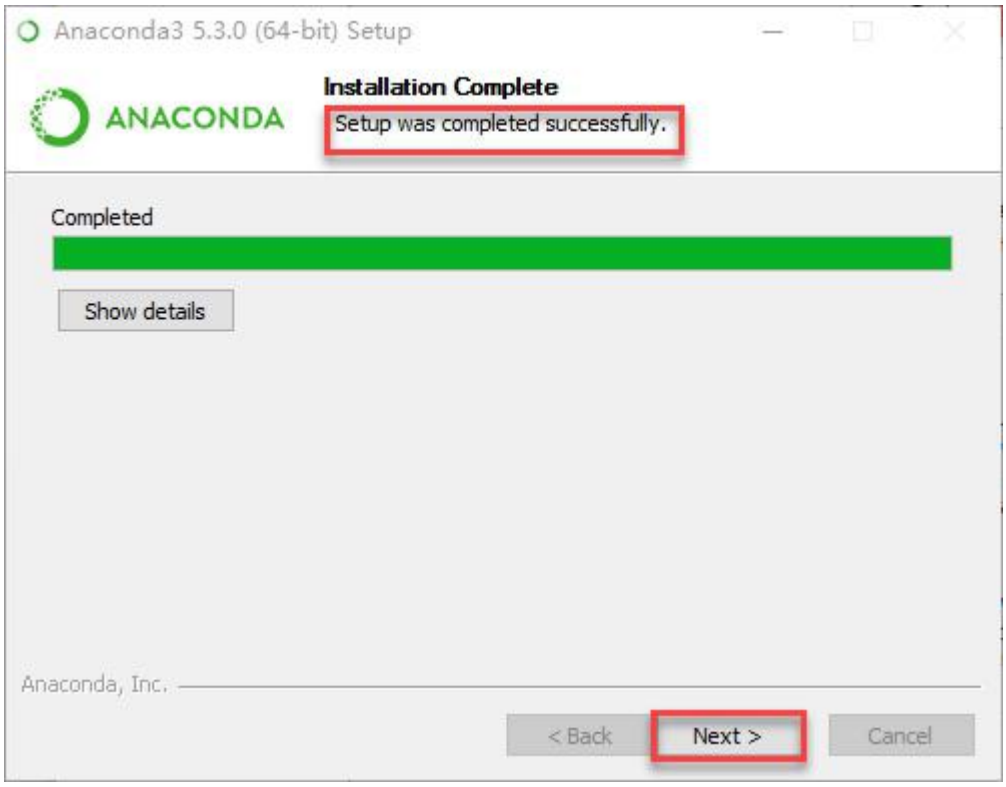

图 22-8 安装成功

5) 单击并勾选如图 22-9、22-10 所示按钮。

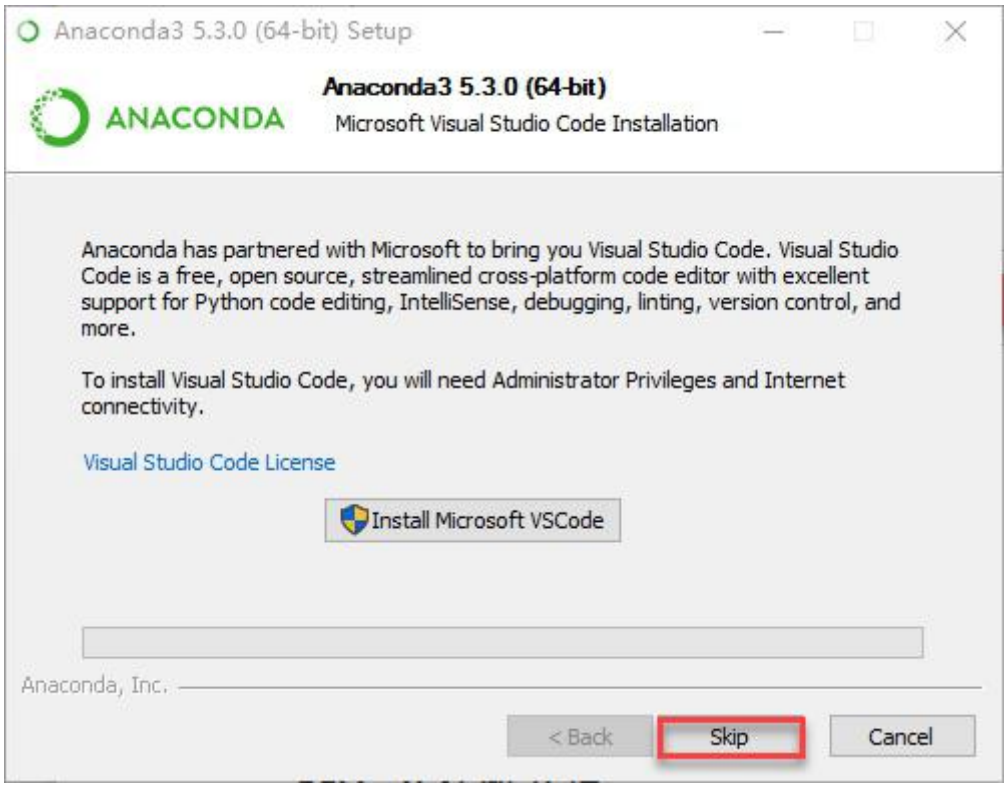

图 22-9 跳过

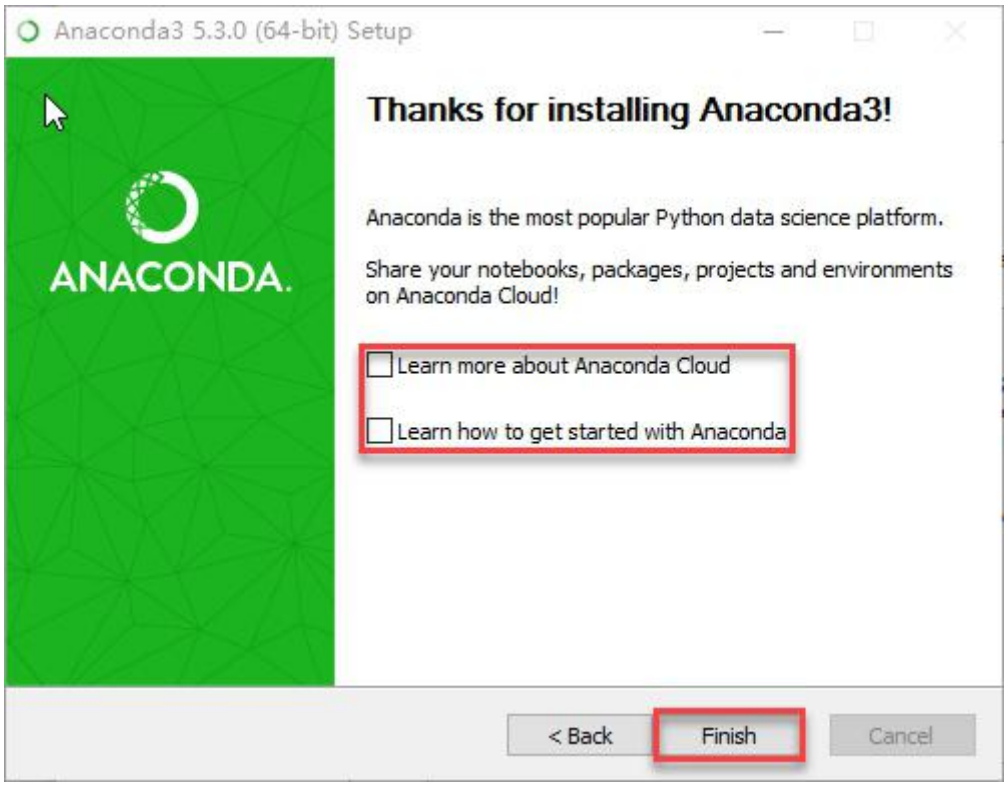

图 22-10 安装完成

6) 完成上述操作后，开始界面就会看到如图 22-11 所示新添加的程序，这就表示 Anoconda 安装好了。单击 Jupyter Notebook 打开，会弹出一个黑框，按 Enter 键后会让你选择哪个浏览器打开，建议选择 Chrome 浏览器。

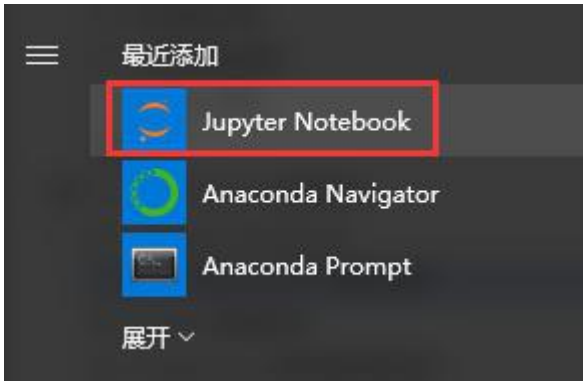

图 22-11 开始界面新增程序

7) 当看到如图 22-12 所示界面时，表示环境已经配置好了。

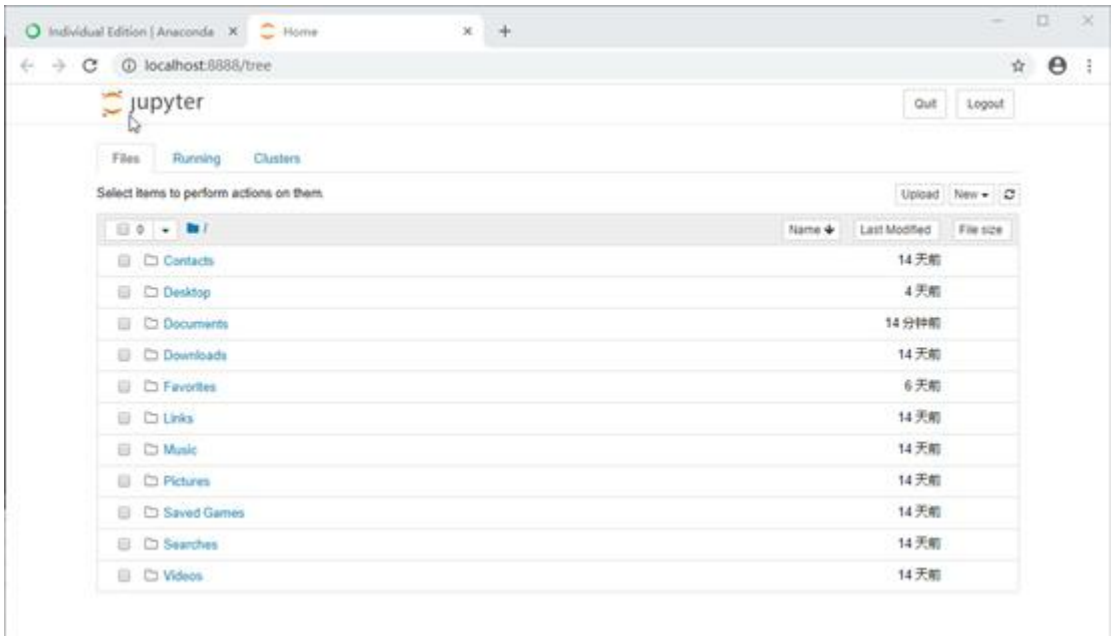

图 22-12 启动 Jupyter Notebook 界面

22.1.2 新建 Jupyter Notebook 文件

打开 Jupyter Notebook 后单击右上角的 New 按钮，在下拉列表中选择 Python3 选项来创建一个 Python 文件，如图 22-13 所示。

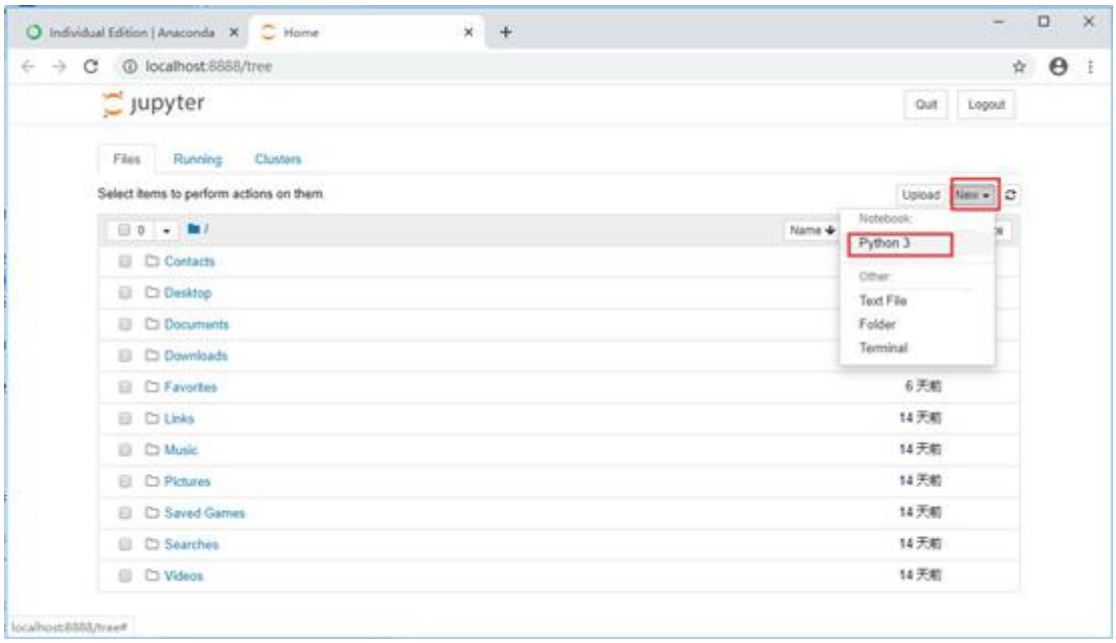

图 22-13 新建 Jupyter Notebook 文件

当你看到如图 22-14 所示的界面就表示新建了一个 Jupyter Notebook 文件。

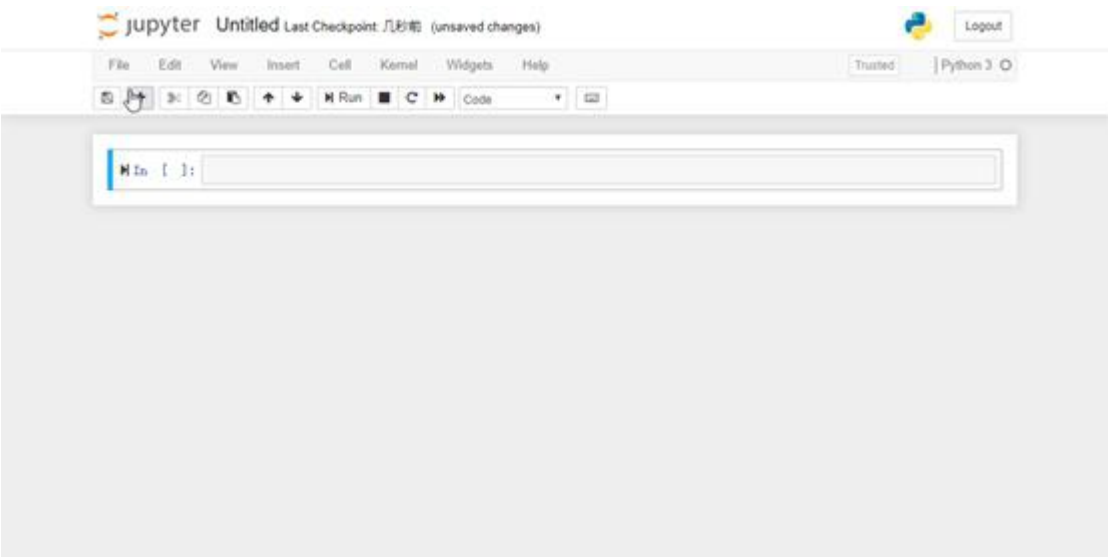

图 22-14 Jupyter Notebook 文件

22.1.3 运行第一段代码

在图 22-14 的代码框中输入 print(“hello Python”)，然后单击 Run 按钮，或者按 Ctrl+Enter 组合键，就会输出 hello Python，这就表示你的第一段代码运行成功。当你想换一个代码框输入代码时，你可以通过单击左上角的“+”按钮来新增代码框。执行效果如图 22-15 所示。

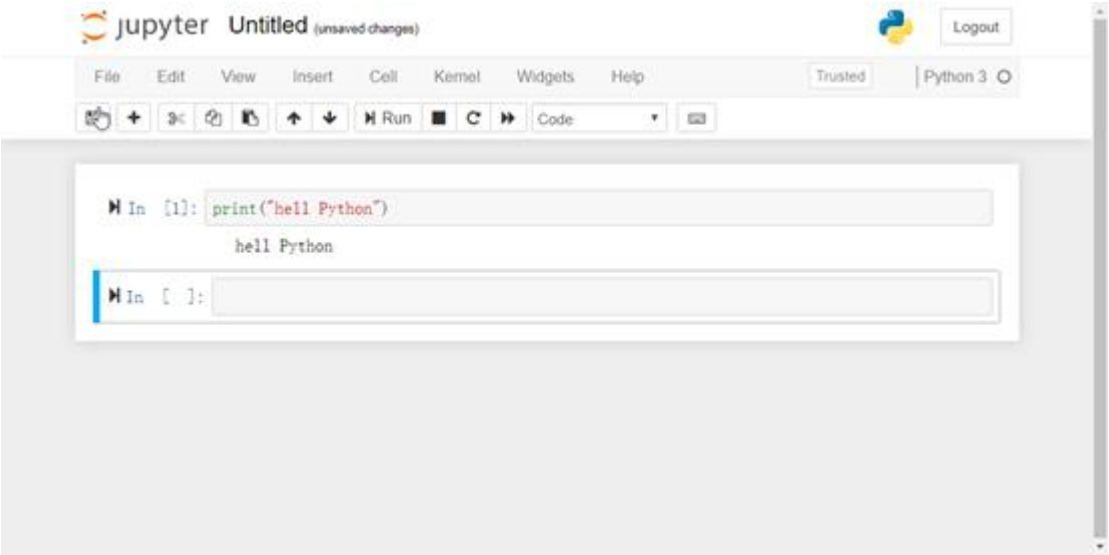

图 22-15 运行第一段代码

22.1.4 重命名 Jupyter Notebook 文件

当新建一个 Jupyter Notebook 文件时，该文件名默认为 Untitled（类似于 Excel 中的工作簿），可以单击 File→Rename 对该文件进行重命名，如图 22-16 所示。

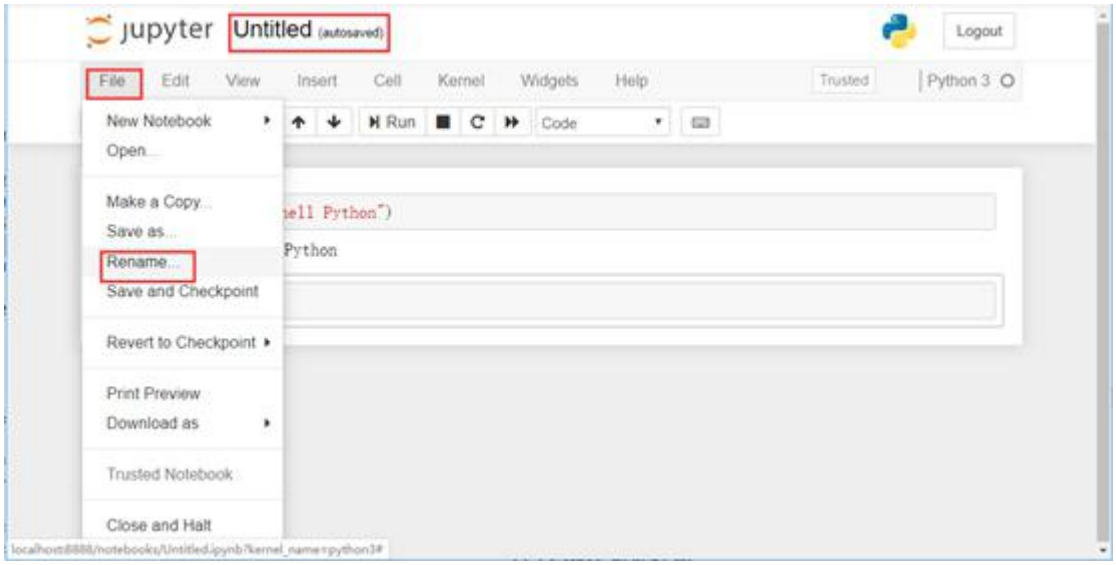

图 22-16 重命名 Jupyter Notebook 文件

22.1.5 保存 Jupyter Notebook 文件

保存文件有两种方式：

- 单击 File→Save and Checkpoint 保存文件，但是这种方法会将文件保存到默认路径下，且文件默认格式为 ipynb，ipynb 是 Jupyter Notebook 的专属文件格式。
- 选择 Download as 选项对文件进行保存，它相当于 Excel 中的“另存为”，可以自己选择保存路径及保存格式，如图 22-17 所示。

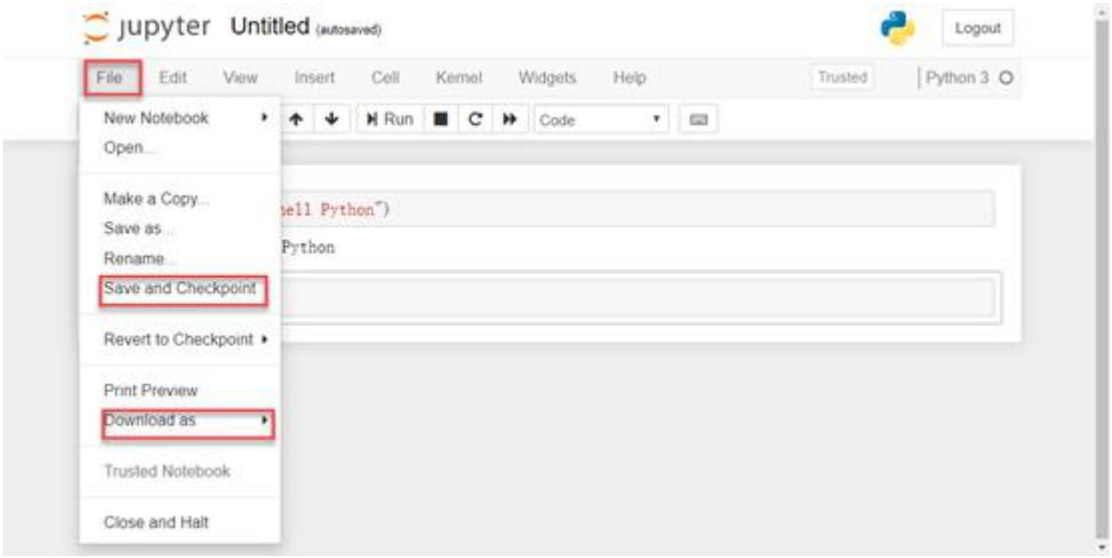

图 22-17 保存 Jupyter Notebook 文件

22.2 Pandas 数据类型

Pandas 中两个重要的数据类型：Series 和 DataFrame。Series 表示数据列表，DataFrame 表示二维数据集。

22.2.1 Series 对象创建

【示例 22-1】使用列表创建 Series 对象

```
import pandas as pd
data=pd.Series([4,3,5,6,1])
data
```

运行结果如图 22-18 所示：

```
In [2]: import pandas as pd

In [3]: data=pd.Series([4,3,5,6,1])
        data

Out[3]: 0    4
        1    3
        2    5
        3    6
        4    1
        dtype: int64
```

图 22-18 示例 22-1 运行效果图

series 对象包装的是 numpy 中的一维数组，实际上是将一个一维数组与一个索引名称捆绑在一起了。

pandas 中两个重要的属性 values 和 index，values 是 Series 对象的原始数据。index 对应了 Series 对象的索引对象。

【示例 22-2】Series 对象中两个重要的属性 values 和 index

```
data.values
data.index
```

运行结果如图 22-19 所示：

```
data.index
RangeIndex(start=0, stop=5, step=1)
data.values
array([4, 3, 5, 6, 1], dtype=int64)
```

图 22-19 示例 22-2 运行效果图

【示例 22-3】创建 Series 对象时候，指定 index 属性

```
data=pd.Series([5,4,6,3,1],index=['one','two','three','four','five'])
data
```

运行结果如图 22-20 所示：

```
In [10]: data=pd.Series([5,4,6,3,1],index=['one','two','three','four','five'])
         data
Out[10]: one      5
         two      4
         three    6
         four     3
         five     1
         dtype: int64
```

图 22-20 示例 22-3 运行效果图

【示例 22-4】创建 Series 对象时候，使用 list 列表指定 index 属性

```
data=pd.Series([4,3,2,1],index=list('abcd'))
data
```

运行结果如图 22-21 所示：

```
Out[14]: a    4
         b    3
         c    2
         d    1
         dtype: int64
```

图 22-21 示例 22-4 运行效果图

【示例 22-5】使用字典创建 Series 对象，默认将 key 作为 index 属性

```
population_dict={'bj':3000,'gz':1500,'sh':2800,'sz':1200}
```

```
population_series=pd.Series(population_dict)
population_series
```

运行结果如图 22-22 所示：

```
In [2]: population_dict={'bj':3000,'gz':1500,'sh':2800,'sz':1200}
        population_series=pd.Series(population_dict)
        population_series

Out[2]: bj    3000
        gz    1500
        sh    2800
        sz    1200
        dtype: int64
```

图 22-22 示例 22-5 运行效果图

【示例 22-6】使用字典创建 Series 对象，又指定了 index 属性值，如果 key 不存在，则值为 NaN

```
sub_series=pd.Series(population_dict,index=['bj','sh']) #如果存在取交集
sub_series
sub_series=pd.Series(population_dict,index=['bj','xa']) #如果不存在则值为 NaN
sub_series
```

运行结果如图 22-23 所示：

```
In [22]: sub_series=pd.Series(population_dict,index=['bj','sh'])
         sub_series

Out[22]: bj    3000
         sh    2800
         dtype: int64
```

```
In [23]: sub_series=pd.Series(population_dict,index=['bj','xa'])
         sub_series

Out[23]: bj    3000.0
         xa       NaN
         dtype: float64
```

图 22-23 示例 22-6 运行效果图

【示例 22-7】标量与 index 属性创建 Series

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

运行结果如图 22-24 所示：

```
In [25]: data=pd.Series(10,index=[4,3,2,5])
         data
Out[25]: 4    10
         3    10
         2    10
         5    10
         dtype: int64
```

图 22-24 示例 22-7 运行效果图

22.2.2 DataFrame 对象创建

将两个 series 对象作为字典的值，就可以创建一个 DataFrame 对象。

【示例 22-8】创建 DataFrame 对象

```
population_dict={'beijing':3000,'shanghai':1200,'guangzhou':1800}
area_dict={'beijing':300,'guangzhou':200,'shanghai':180}
import pandas as pd
population_series=pd.Series(population_dict)
area_series=pd.Series(area_dict)
citys=pd.DataFrame({'area':area_series,'population':population_series})
citys
```

运行结果如图 22-25 所示：

Out[4]:

	area	population
beijing	300	3000
guangzhou	200	1800
shanghai	180	1200

图 22-25 示例 22-8 运行效果图

【示例 22-9】查看 DataFrame 对象的 values、index 和 columns 属性

```
citys.index
citys.values
citys.columns
```

运行结果如图 22-26 所示：

```
In [6]: citys.values
Out[6]: array([[ 300, 3000],
               [ 200, 1800],
               [ 180, 1200]], dtype=int64)

In [7]: citys.index
Out[7]: Index(['beijing', 'guangzhou', 'shanghai'], dtype='object')

In [8]: citys.columns
Out[8]: Index(['area', 'population'], dtype='object')
```

图 22-26 示例 22-9 运行效果图

【示例 22-10】使用列表创建 DataFrame 对象

```
population_dict={'beijing':3000,'shanghai':1200,'guangzhou':1800}
area_dict={'beijing':300,'shanghai':180,'guangzhou':200}
data=pd.DataFrame([population_dict,area_dict])
print(data) #将‘beijing’ ‘shanghai’ ‘guangzhou’作为表头
```

运行结果如图 22-27 所示：

	beijing	guangzhou	shanghai
0	3000	1800	1200
1	300	200	180

图 22-27 示例 22-10 运行效果图

【示例 22-11】创建 DataFrame 对象，指定 index 属性

```
population_dict={'beijing':3000,'guangzhou':1800,'shanghai':1200}
area_dict={'beijing':300,'shanghai':180,'guangzhou':200}
data=pd.DataFrame([area_dict,population_dict],index=['area','population'])
```

```
data
```

运行结果如图 22-28 所示：

	beijing	guangzhou	shanghai
area	300	200	180
population	3000	1800	1200

图 22-28 示例 22-11 运行效果图

【示例 22-12】创建 DataFrame 对象，指定列索引 columns

```
population_series=pd.Series(population_dict)
pd.DataFrame(population_series,columns=['population'])
```

运行结果如图 22-29 所示：

	population
beijing	3000
guangzhou	1800
shanghai	1200

图 22-29 示例 22-12 运行效果图

【示例 22-13】使用列表创建 Dataframe 对象

```
pd.DataFrame([{'a':i,'b':i*2} for i in range(3)])
```

运行结果如图 22-30 所示：

```
In [23]: pd.DataFrame([{'a':i,'b':i*2} for i in range(3)])
```

Out[23]:

	a	b
0	0	0
1	1	2
2	2	4

图 22-30 示例 22-13 运行效果图

【示例 22-14】使用一个二维数组并指定 columns 和 index 创建 DataFrame 对象

```
import numpy as np
```

```
pd.DataFrame(np.random.randint(0,10,(3,2)),columns=list('ab'),index=list('efg'))
```

运行结果如图 22-31 所示：

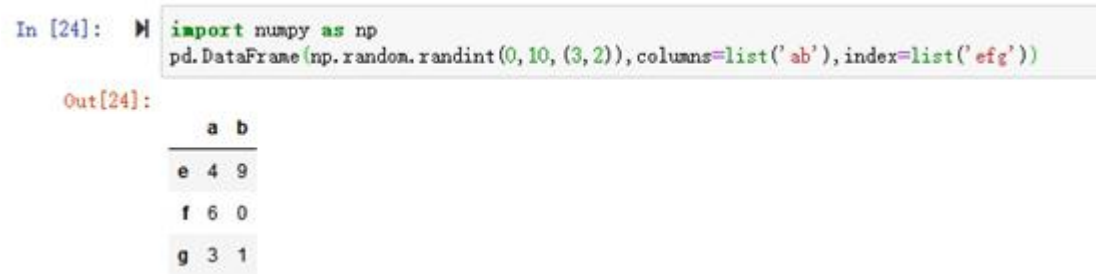

图 22-31 示例 22-14 运行效果图

22.2.3 获取 Series 对象的值

【示例 22-15】Series 对象的切片、索引

```
import numpy as np
import pandas as pd
data=pd.Series([4,3,25,2,3],index=list('abcde'))
display('根据 key 获取：',data['a'])
display('切片获取：',data['a':'d'])
display('索引获取：',data[1])
display('索引切片：',data[2:4])
```

运行结果如图 22-32 所示：

```
'根据key获取：'
4
'切片获取：'
a    4
b    3
c   25
d    2
dtype: int64
'索引获取：'
3
'索引切片：'
c   25
d    2
dtype: int64
```

图 22-32 示例 22-15 运行效果图

从示例 22-15 运行结果如图 22-32 所示，可以看出 Series 与 ndarray 数组都可以通过索引访问元素，Series 对象的索引分为位置索引和标签索引。不同之处，标签索引进行切片时候是左闭右闭，而位置索引是左闭右开。

【示例 22-16】位置索引与标签索引相同的问题

```
data=pd.Series([5,6,7,8],index=[1,2,3,4])
data[1]
```

如示例 22-16 所示，位置索引与标签索引有相同值 1，这时候 data[1]就不知道是按哪个来获取，此时要使用 loc、iloc。其中 loc 表示的是标签索引，iloc 表示的是位置索引。

【示例 22-17】Series 对象中 loc 与 iloc 的使用

```
data=pd.Series([5,3,2,5,9],index=[1,2,3,4,5])
data.loc[1]
data.iloc[1]
```

运行结果如图 22-33 所示：

```
In [44]: data=pd.Series([5,3,2,5,9],index=[1,2,3,4,5])
         data
Out[44]: 1    5
         2    3
         3    2
         4    5
         5    9
         dtype: int64

In [45]: data.loc[1]
Out[45]: 5

In [46]: data.iloc[1]
Out[46]: 3
```

图 22-33 示例 22-17 运行效果图

22.2.4 获取 DataFrame 的值

1) 选择某一列/某几列

DataFrame 对象非常容易获取数据集中指定列的数据。只需要在表 df 后面的括号中指明要选择的列名即可。如果要获取一列，则只需要传入一个列名；如果是同时选择多列，则传入多个列名即可，多个列名用一个 list 存放。

【示例 22-18】创建 DataFrame 对象

```
import numpy as np
import pandas as pd
data=pd.DataFrame(np.arange(12).reshape(3,4),index=list('abc'),columns=list('ABCD'))
```

运行结果如图 22-34 所示：

```
In [3]: import numpy as np
        import pandas as pd
        data=pd.DataFrame(np.arange(12).reshape(3,4),index=list('abc'),columns=list('ABCD'))
        data
Out[3]:
```

	A	B	C	D
a	0	1	2	3
b	4	5	6	7
c	8	9	10	11

图 22-34 示例 22-18 运行效果图

【示例 22-19】获取 DataFrame 对象中某一列/某几列

```
print('获取‘B’列：')
print(data['B'])
print('获取‘A’‘C’两列：')
print(data[['A','C']])
```

运行结果如图 22-35 所示：

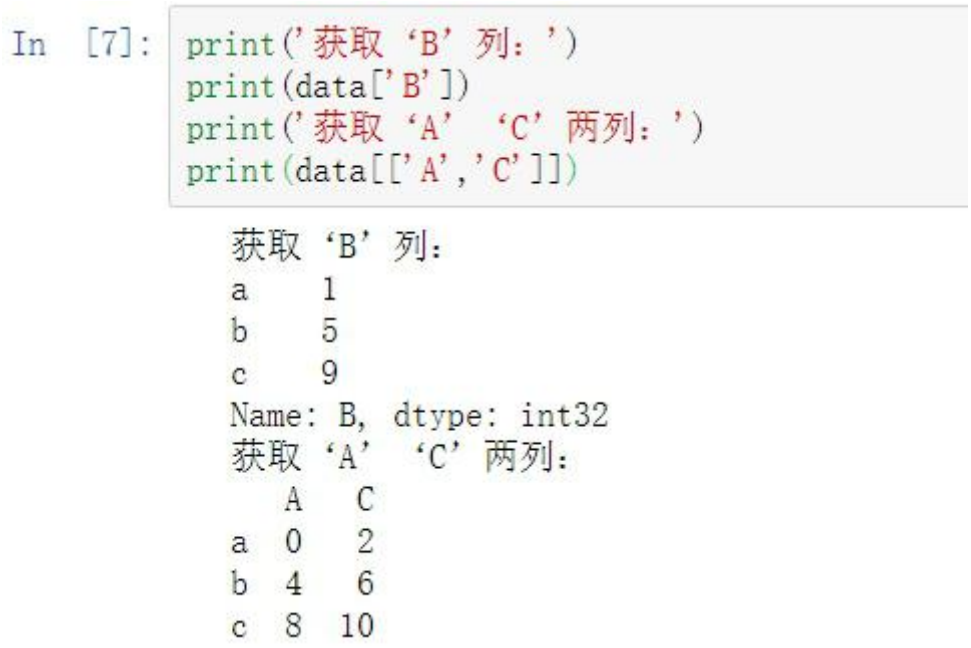

图 22-35 示例 22-19 运行效果图

DataFrame 对象获取列，除了传入具体的列名，还可以传入具体列的位置，通过传入位置来获取数据时需要用到 iloc 方法。

【示例 22-20】通过传入位置获取 DataFrame 对象中某一列/某几列

```
print('获取第 1 列：')
print(data.iloc[:,0])
print('获取第 1 列和第 3 列：')
print(data.iloc[:,[0,2]])
```

运行结果如图 22-36 所示：

```
[10]: print('获取第1列：')
      print(data.iloc[:,0])
      print('获取第1列和第3列：')
      print(data.iloc[:,[0,2]])
```

```
获取第1列：
a    0
b    4
c    8
Name: A, dtype: int32
获取第1列和第3列：
   A   C
a  0   2
b  4   6
c  8  10
```

图 22-36 示例 22-20 运行效果图

从上面的示例中可以看到，iloc 后的方括号中逗号之前的部分表示要获取的行的位置。只输入一个冒号，不输入任何数值表示获取所有的行；逗号之后的方括号表示要获取的列的位置，列的位置同样也是从 0 开始计数。

2) 选择连续的某几列

我们将通过列名选择数据的方式叫做普通索引，传入列的位置选择数据的方式叫做位置索引。获取连续的某几列，用普通索引和位置索引都可以做到。因为要获取的列是连续的，所以直接对列进行切片。

【示例 22-21】获取 DataFrame 对象中连续几列

```
print('获取 B C D 三列，使用普通索引获取：')
print(data.loc[:,'B':'D'])
print('获取 B C D 三列，使用位置索引获取：')
print(data.iloc[:,1:4])
```

运行结果如图 22-37 所示：

```
print('获取B C D三列，使用普通索引获取：')
print(data.loc[:,'B':'D'])
print('获取B C D三列，使用位置索引获取：')
print(data.iloc[:,1:4])
```

```
获取B C D三列，使用普通索引获取：
   B   C   D
a  1   2   3
b  5   6   7
c  9  10  11
获取B C D三列，使用位置索引获取：
   B   C   D
a  1   2   3
b  5   6   7
c  9  10  11
```

图 22-37 示例 22-21 运行效果图

从上面的示例可以看到，loc 和 iloc 后的方括号中逗号之前的表示选择的行，当只传入一个冒号时，表示选择所有行，逗号后面表示要选择列。data.loc[:,'B':'D']表示选择从 B 列开始到 D 列之间的值（包括 B 列也包括 D 列），data.iloc[:,1:4]表示选择第 2 列到第 5 列之间的值（包括第 1 列但不包括第 5 列）。

3) 选择某一行/某几行

获取行的方式主要有两种，一种是普通索引，即传入具体行索引的名称，需要用到 loc 方法；另外一种是位置索引，即传入具体的行数，需要用到 iloc 方法。

【示例 22-22】DataFrame 对象中选择某一行/某几行

```
print('获取 a 行,普通索引获取：')
print(data.loc['a'])
print('获取 a c 行,普通索引获取：')
print(data.loc[['a','c']])
print('获取第 1 行，位置索引获取：')
print(data.iloc[0])
print('获取第 1 行第 3 行，位置索引获取：')
print(data.iloc[[0,2]])
```

运行结果如图 22-38 所示：

```
获取a行,普通索引获取:
A    0
B    1
C    2
D    3
Name: a, dtype: int32
获取a c行,普通索引获取:
   A  B   C   D
a  0  1   2   3
c  8  9  10  11
获取第1行，位置索引获取:
A    0
B    1
C    2
D    3
Name: a, dtype: int32
获取第1行第3行，位置索引获取:
   A  B   C   D
a  0  1   2   3
c  8  9  10  11
```

图 22-38 示例 22-22 运行效果图

4) 选择连续的某几行

选择连续的某几行和选择连续某几列类似，只要把连续行的位置用一个区间表示即可。

【示例 22-23】DataFrame 对象中选择某一行/某几行

```
print('选择 a 行  b 行，使用普通索引：')
print(data.loc['a':'b'])
print('选择第 1 行  第 2 行，使用位置索引：')
print(data.iloc[0:2])
```

运行结果如图 22-39 所示：

```
选择a行 b行，使用普通索引:
   A  B  C  D
a  0  1  2  3
b  4  5  6  7
选择第1行 第2行，使用位置索引:
   A  B  C  D
a  0  1  2  3
b  4  5  6  7
```

图 22-39 示例 22-23 运行效果图

5) 行列同时选择

【示例 22-24】同时选择连续的部分行和部分列

```
print('同时获取 a b 行，A B 列，使用普通索引：')
print(data.loc['a':'b','A':'B'])
print('同时获取 a b 行，A B 列，使用位置索引：')
print(data.iloc[0:2,0:2])
```

运行结果如图 22-40 所示：

```
同时获取a b行，A B列，使用普通索引：
   A  B
a  0  1
b  4  5
同时获取a b行，A B列，使用位置索引：
   A  B
a  0  1
b  4  5
```

图 22-40 示例 22-24 运行效果图

【示例 22-25】同时选择不连续的部分行和部分列

```
print('同时获取 a c 行，ABD 列，使用普通索引：')
print(data.loc[['a','c'],['A','B','D']])
print('同时获取 a c 行，ABD 列，使用位置索引：')
print(data.iloc[[0,2],[0,1,3]])
```

运行结果如图 22-41 所示：

```
同时获取a c行，ABD列，使用普通索引：
   A  B   D
a  0  1   3
c  8  9  11
同时获取a c行，ABD列，使用位置索引：
   A  B   D
a  0  1   3
c  8  9  11
```

图 22-41 示例 22-25 运行效果图

22.2.5 Series 的方法

Series 对象中有很多常用的方法可以对数据进行各种处理。例如，mean 方法可以对某一列数据取平均数，min 方法获取最小值，max 方法获取最大值，std 方法获取标准差。

【示例 22-26】使用 mean、min、max、std 等方法对数据集进行各种运算，最后对数据集进行排序操作

```
import pandas as pd
data = pd.DataFrame({
    'Name':['zs','lisi','ww'],
    'Sno':['1001','1002','1003'],
    'Sex':['man','woman','man'],
    'Age':[17,18,19],
    'Score':[80,97,95]
},columns=['Sno','Sex','Age','Score'],index=['zs','lisi','ww'])
display('数据集',data)
ages = data['Age']
display('获取数据集中 Age 列的所有',ages)
print('计算 Age 列的平均值：',ages.mean())
print('计算 Age 列的最大值：',ages.max())
print('计算 Age 列的最小值：',ages.min())
print('计算 Age 列的标准差：',ages.std())
display('对 Age 进行降序排序：',ages.sort_values(ascending=False))
```

运行结果如图 22-42 所示：

'数据集'

	Sno	Sex	Age	Score
zs	1001	man	17	80
lisi	1002	woman	18	97
ww	1003	man	19	95

'获取数据集中Age列的所有'

```
zs      17
lisi    18
ww      19
Name: Age, dtype: int64

计算Age列的平均值:  18.0
计算Age列的最大值:  19
计算Age列的最小值:  17
计算Age列的标准差:  1.0

'对Age进行降序排序:'

ww      19
lisi    18
zs      17
Name: Age, dtype: int64
```

图 22-42 示例 22-26 运行效果图

22.2.6 Series 的条件过滤

Series 对象也可以像 SQL 语句一样，通过指定条件来过滤数据。

【示例 22-27】Series 对象指定条件来过滤数据

```
import pandas as pd
data = pd.DataFrame({
    'Name':['zs','lisi','ww'],
    'Sno':['1001','1002','1003'],
    'Sex':['man','woman','man'],
    'Age':[17,18,19],
```

```
    'Score':[80,97,95]
},columns=['Sno','Sex','Age','Score'],index=['zs','lisi','ww'])
display('数据集',data)
scores = data['Score']
display('筛选出成绩大于平均值的数据：',scores[scores>scores.mean()])
```

运行结果如图 22-43 所示：

'数据集'

	Sno	Sex	Age	Score
zs	1001	man	17	80
lisi	1002	woman	18	97
ww	1003	man	19	95

'筛选出成绩大于平均值的数据：'

```
lisi    97
ww      95
Name: Score, dtype: int64
```

图 22-43 示例 22-27 运行效果图

22.2.7 DataFrame 的条件过滤

DataFrame 与 Series 类似，也可以使用条件进行过滤。

【示例 22-28】DataFrame 对象指定条件来过滤数据

```
import pandas as pd
data = pd.DataFrame({
    'Name':['zs','lisi','ww'],
    'Sno':['1001','1002','1003'],
    'Sex':['man','woman','man'],
    'Age':[17,18,19],
    'Score':[80,97,95]
},columns=['Sno','Sex','Age','Score'],index=['zs','lisi','ww'])
display('数据集',data)
scores = data['Score']
```

```
display('输出数据中所有成绩大于平均值的记录',data[scores>scores.mean()])
display('获取成绩大于平均值得所有记录，只显示Sno Age Score三列：',data[scores>scores.mean()].loc[:,['Sno','Age','Score']])
```

运行结果如图 22-44 所示：

'数据集'

	Sno	Sex	Age	Score
zs	1001	man	17	80
lisi	1002	woman	18	97
ww	1003	man	19	95

'输出数据中所有成绩大于平均值的记录'

	Sno	Sex	Age	Score
lisi	1002	woman	18	97
ww	1003	man	19	95

'获取成绩大于平均值得所有记录，只显示Sno Age Score三列：'

	Sno	Age	Score
lisi	1002	18	97
ww	1003	19	95

图 22-44 示例 22-28 运行效果图

22.3 处理缺失值

缺失值就是由某些原因导致部分数据为空，对于为空的这部分数据我们一般有两种处理方式，一种是删除，即把含有缺失值的数据删除；另外一种是填充，即把缺失的那部分数据用某个值代替。

22.3.1 缺失值查看

对缺失值进行处理，首先要把缺失值找出来，也就是查看哪列有缺失值。直接调用 info() 方法就会返回每一列的缺失情况。

【示例 22-29】缺失值查看

```
df=pd.DataFrame([[1,2,np.nan],[4,np.nan,6],[5,6,7]])
df.info()
```

运行结果如图 22-45 所示：

```
df=pd.DataFrame([[1, 2, np.nan], [4, np.nan, 6], [5, 6, 7]])
df
```

	0	1	2
0	1	2.0	NaN
1	4	NaN	6.0
2	5	6.0	7.0

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3 entries, 0 to 2
Data columns (total 3 columns):
0    3 non-null int64
1    2 non-null float64
2    2 non-null float64
dtypes: float64(2), int64(1)
memory usage: 152.0 bytes
```

图 22-45 示例 22-29 运行效果图

Pandas 中缺失值用 NaN 表示，从用 info()方法的结果来看，索引 1 这一列是 1 2 non-null float64，表示这一列有 2 个非空值，而应该是 3 个非空值，说明这一列有 1 个空值。

还可以用 isnull()方法来判断哪个值是缺失值，如果是缺失值则返回 True，如果不是缺失值返回 False。

【示例 22-30】获取所有缺失值

```
data=pd.Series([3,4,np.nan,1,5,None])
print('isnull()方法判断是否是缺值：')
print(data.isnull())
print('获取缺值：')
print(data[data.isnull()])
print('获取非空值')
```

```
print(data[data.notnull()])
```

运行结果如图 22-46 所示：

```
isnull()方法判断是否是缺值：
0    False
1    False
2     True
3    False
4    False
5     True
dtype: bool
获取缺值：
2   NaN
5   NaN
dtype: float64
获取非空值
0    3.0
1    4.0
3    1.0
4    5.0
dtype: float64
```

图 22-46 示例 22-30 运行效果图

22.3.2 缺失值删除

缺失值分为两种，一种是一行中某个字段是缺失值；另一种是一行中的字段全部为缺失值，即为一个空白行。调用 dropna()方法删除缺失值，dropna()方法默认删除含有缺失值的行，也就是只要某一行有缺失值就把这一行删除。如果想按列为单位删除缺失值，需要传入参数 axis=’columns’。

【示例 22-31】删除缺失值

```
df=pd.DataFrame([[1,2,np.nan],[4,np.nan,6],[5,6,7]])
print('默认为以行为单位剔除:')
display(df.dropna())
print('以列为单位剔除:')
display(df.dropna(axis='columns'))
```

运行结果如图 22-47 所示：

默认为以行为单位剔除：

	0	1	2
2	5	6.0	7.0

以列为单位剔除：

	0
0	1
1	4
2	5

图 22-47 示例 22-31 运行效果图

如果想删除空白行，需要给 dropna()方法中传入参数 how='all'即可，默认值是 any。

【示例 22-32】删除空白行

```
df=pd.DataFrame([[1,2,np.nan],[4,np.nan,6],[5,6,7]])
print('所有为 nan 时候才剔除:')
display(df.dropna(how='all'))
print('默认情况，只要有就剔除')
display(df.dropna(how='any'))
```

运行结果如图 22-48 所示：

所有为nan时候才剔除：

	0	1	2
0	1	2.0	NaN
1	4	NaN	6.0
2	5	6.0	7.0

默认情况，只要有就剔除

	0	1	2
2	5	6.0	7.0

图 22-48 示例 22-32 运行效果图

22.3.3 缺失值填充

上面介绍了缺失值的删除，但是数据是宝贵的，一般情况下只要数据缺失比例不高（不大于 30%），尽量别删除，而是选择填充。

调用 fillna()方法对数据表中的所有缺失值进行填充，在 fillna()方法中输入要填充的值。还可以通过 method 参数使用前一个数和后一个数来进行填充。

【示例 22-33】Series 对象缺失值填充

```
data=pd.Series([3,4,np.nan,1,5,None])
print('以 0 进行填充:')
display(data.fillna(0))
print('以前一个数进行填充:')
display(data.fillna(method='ffill'))
print('以后一个数进行填充:')
display(data.fillna(method='bfill'))
print('先按前一个，再按后一个')
display(data.fillna(method='bfill').fillna(method='ffill'))
```

运行结果如图 22-49 所示：

```
以0进行填充:

0    3.0
1    4.0
2    0.0
3    1.0
4    5.0
5    0.0
dtype: float64

以前一个数进行填充:

0    3.0
1    4.0
2    4.0
3    1.0
4    5.0
5    5.0
dtype: float64

以后一个数进行填充:

0    3.0
1    4.0
2    1.0
3    1.0
4    5.0
5    NaN
dtype: float64
```

图 22-49 示例 22-33 运行效果图

【示例 22-34】DataFrame 对象缺失值填充

```
df=pd.DataFrame([[1,2,np.nan],[4,np.nan,6],[5,6,7]])
print('使用数值 0 来填充：')
display(df.fillna(0))
print('使用行的前一个数来填充：')
display(df.fillna(method='ffill'))
print('使用列的后一个数来填充：')
display(df.fillna(method='bfill' ,axis=1))
```

运行结果如图 22-50 所示：

使用数值0来填充：

	0	1	2
0	1	2.0	0.0
1	4	0.0	6.0
2	5	6.0	7.0

使用行的前一个数来填充：

	0	1	2
0	1	2.0	NaN
1	4	2.0	6.0
2	5	6.0	7.0

使用列的后一个数来填充：

	0	1	2
0	1.0	2.0	NaN
1	4.0	6.0	6.0
2	5.0	6.0	7.0

图 22-50 示例 22-34 运行效果图

【示例 22-35】列的平均值来填充

```
df=pd.DataFrame([[1,2,np.nan],[4,np.nan,6],[5,6,7]])
for i in df.columns:
    df[i]=df[i].fillna(np.nanmean(df[i]))
df
```

运行结果如图 22-51 所示：

	0	1	2
0	1	2.0	6.5
1	4	4.0	6.0
2	5	6.0	7.0

图 22-51 示例 22-35 运行效果图

22.4 拼接

【示例 22-36】Series 对象拼接

```
ser1=pd.Series([1,2,3],index=list('ABC'))
ser2=pd.Series([4,5,6],index=list('DEF'))
pd.concat([ser1,ser2])
```

运行结果如图 22-52 所示：

```
In [4]: ser1=pd.Series([1,2,3],index=list('ABC'))
        ser2=pd.Series([4,5,6],index=list('DEF'))
        pd.concat([ser1,ser2])

Out[4]: A    1
        B    2
        C    3
        D    4
        E    5
        F    6
        dtype: int64
```

图 22-52 示例 22-36 运行效果图

【示例 22-37】两个 df 对象拼接，默认找相同的列索引进行合并

```
def make_df(cols,index):
    data={c:[str(c)+str(i) for i in index] for c in cols}
    return pd.DataFrame(data,index=index)
df1=make_df('AB',[1,2])
df2=make_df('AB',[3,4])
pd.concat([df1,df2])
```

运行结果如图 22-53 所示：

```
In [5]: df1=make_df('AB',[1,2])
        df2=make_df('AB',[3,4])
        pd.concat([df1,df2])
```

Out[5]:

	A	B
1	A1	B1
2	A2	B2
3	A3	B3
4	A4	B4

图 22-53 示例 22-37 运行效果图

【示例 22-38】两个 df 对象拼接，添加 axis 参数

```
df1=make_df('AB',[1,2])
df2=make_df('AB',[3,4])
pd.concat([df1,df2],axis=1) #或者 pd.concat([df1,df2],axis='columns')
```

运行结果如图 22-54 所示：

```
In [6]: pd.concat([df1,df2],axis=1)
```

Out[6]:

	A	B	A	B
1	A1	B1	NaN	NaN
2	A2	B2	NaN	NaN
3	NaN	NaN	A3	B3
4	NaN	NaN	A4	B4

```
In [7]: pd.concat([df1,df2],axis='columns')
```

Out[7]:

	A	B	A	B
1	A1	B1	NaN	NaN
2	A2	B2	NaN	NaN
3	NaN	NaN	A3	B3
4	NaN	NaN	A4	B4

图 22-54 示例 22-38 运行效果图

【示例 22-39】两个 df 对象拼接，索引重复问题

```
x=make_df('AB',[1,2])
y=make_df('AB',[1,2])
pd.concat([x,y])
```

运行结果如图 22-55 所示：

```
In [9]: x=make_df('AB',[1,2])
        y=make_df('AB',[1,2])
        pd.concat([x,y])
```

Out[9]:

	A	B
1	A1	B1
2	A2	B2
1	A1	B1
2	A2	B2

图 22-55 示例 22-39 运行效果图

【示例 22-40】两个 df 对象拼接，解决索引重复问题加 ignore_index 属性

```
x=make_df('AB',[1,2])
y=make_df('AB',[1,2])
pd.concat([x,y],ignore_index=True)
```

运行结果如图 22-56 所示：

```
In [10]: pd.concat([x,y],ignore_index=True)
```

Out[10]:

	A	B
0	A1	B1
1	A2	B2
2	A1	B1
3	A2	B2

图 22-56 示例 22-40 运行效果图

【示例 22-41】两个 df 对象拼接，解决索引重复问题，加 keys 属性

```
x=make_df('AB',[1,2])
y=make_df('AB',[1,2])
pd.concat([x,y],keys=list('xy'))
```

运行结果如图 22-57 所示：

```
In [12]: pd.concat([x,y],keys=list('xy'))
Out[12]:
```

		A	B
x	1	A1	B1
	2	A2	B2
y	1	A1	B1
	2	A2	B2

图 22-57 示例 22-41 运行效果图

【示例 22-42】两个 df 对象拼接，join 内连接做交集

```
a=make_df('ABC',[1,2])
b=make_df('BCD',[3,4])
pd.concat([a,b],join='inner')
```

运行结果如图 22-58 所示：

```
In [18]: pd.concat([a,b],join='inner')
Out[18]:
```

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

图 22-58 示例 22-42 运行效果图

22.5 merge 的使用

pandas 中的 merge 和 concat 类似，但主要是用于两组有 key column 的数据，统一索引的数据。通常也被用在 Database 的处理当中。

合并时有 4 种方式 how = ['left', 'right', 'outer', 'inner']，默认值 how='inner'。

【示例 22-43】merge 的使用，默认以 how=’inner’进行合并

```
left=pd.DataFrame({'key':['k0','k1','k2','k3'],
                   'A':['A0','A1','A2','A3'],
                   'B':['B0','B1','B2','B3'],
                  })
right=pd.DataFrame({'key':['k0','k1','k4','k3'],
                   'C':['C0','C1','C2','C3'],
                   'D':['D0','D1','D2','D3'],
                  })
result=pd.merge(left,right)
```

运行结果如图 22-59 所示：

```
left=pd.DataFrame({'key':['k0','k1','k2','k3'],
                   'A':['A0','A1','A2','A3'],
                   'B':['B0','B1','B2','B3'],
                  })
right=pd.DataFrame({'key':['k0','k1','k4','k3'],
                   'C':['C0','C1','C2','C3'],
                   'D':['D0','D1','D2','D3'],
                  })
result=pd.merge(left,right,how='inner')
```

```
result
```

]:

	key	A	B	C	D
0	k0	A0	B0	C0	D0
1	k1	A1	B1	C1	D1
2	k3	A3	B3	C3	D3

图 22-59 示例 22-43 运行效果图

【示例 22-44】merge 的使用，参数 how=’outer’进行合并

```
left=pd.DataFrame({'key':['k0','k1','k2','k3'],
                   'A':['A0','A1','A2','A3'],
                   'B':['B0','B1','B2','B3'],
```

```
                        })
right=pd.DataFrame({'key':['k0','k1','k4','k3'],
                            'C':['C0','C1','C2','C3'],
                            'D':['D0','D1','D2','D3'],
                        })
pd.merge(left,right,how='outer')
```

运行结果如图 22-60 所示：

```
left=pd.DataFrame({'key':['k0','k1','k2','k3'],
                   'A':['A0','A1','A2','A3'],
                   'B':['B0','B1','B2','B3'],
                  })
right=pd.DataFrame({'key':['k0','k1','k4','k3'],
                    'C':['C0','C1','C2','C3'],
                    'D':['D0','D1','D2','D3'],
                   })
pd.merge(left,right,how='outer')
```

]:

	key	A	B	C	D
0	k0	A0	B0	C0	D0
1	k1	A1	B1	C1	D1
2	k2	A2	B2	NaN	NaN
3	k3	A3	B3	C3	D3
4	k4	NaN	NaN	C2	D2

图 22-60 示例 22-44 运行效果图

【示例 22-45】merge 的使用，参数 how=’left’进行合并

```
left=pd.DataFrame({'key':['k0','k1','k2','k3'],
                            'A':['A0','A1','A2','A3'],
                            'B':['B0','B1','B2','B3'],
                        })
right=pd.DataFrame({'key':['k0','k1','k4','k3'],
                            'C':['C0','C1','C2','C3'],
                            'D':['D0','D1','D2','D3'],
```

```
                    })
pd.merge(left,right,how='left')
```

运行结果如图 22-61 所示：

```
left=pd.DataFrame({'key':['k0','k1','k2','k3'],
                   'A':['A0','A1','A2','A3'],
                   'B':['B0','B1','B2','B3'],
                  })
right=pd.DataFrame({'key':['k0','k1','k4','k3'],
                    'C':['C0','C1','C2','C3'],
                    'D':['D0','D1','D2','D3'],
                   })
pd.merge(left,right,how='left')
```

4]:

	key	A	B	C	D
0	k0	A0	B0	C0	D0
1	k1	A1	B1	C1	D1
2	k2	A2	B2	NaN	NaN
3	k3	A3	B3	C3	D3

图 22-61 示例 22-45 运行效果图

【示例 22-46】行索引不同， how=’right’进行合并

```
left=pd.DataFrame({'key':['k0','k1','k2','k3'],
                   'A':['A0','A1','A2','A3'],
                   'B':['B0','B1','B2','B3'],
                  })
right=pd.DataFrame({'key':['k0','k1','k4','k3'],
                    'C':['C0','C1','C2','C3'],
                    'D':['D0','D1','D2','D3'],
                   })
pd.merge(left,right,how='right')
```

运行结果如图 22-62 所示：

```
left=pd.DataFrame({'key':['k0','k1','k2','k3'],
                   'A':['A0','A1','A2','A3'],
                   'B':['B0','B1','B2','B3'],
                  })
right=pd.DataFrame({'key':['k0','k1','k4','k3'],
                   'C':['C0','C1','C2','C3'],
                   'D':['D0','D1','D2','D3'],
                  })
pd.merge(left,right,how='right')
```

]:

	key	A	B	C	D
0	k0	A0	B0	C0	D0
1	k1	A1	B1	C1	D1
2	k3	A3	B3	C3	D3
3	k4	NaN	NaN	C2	D2

图 22-62 示例 22-46 运行效果图

习题

一、简答题

1. pandas 库的重要特征是什么？
2. 如何计算 Series 的标准偏差？
3. 在 pandas 中创建 DataFrame 有哪些不同方式？
4. 如何获得数字序列的最小值，第 25，中位数，第 75 和最大值？
5. 如何将 DataFrame 转换为 Excel 文件？

二、编码题

1. 设有如下 data 数据，完成操作：

```
data =  {‘xiaoming’:10,’xiaolan’:80,’black’:77}
```

a. 创建 Series 对象。

b. 获取 values、index 属性。

c. 获取’xiaolan’的值，分别用 loc 和 iloc 获取。

d. 使用切片获取值为 10,80 的数据。

2. 设有如下 data 数据，完成操作：

```
data = {'animal': ['cat', 'cat', 'snake', 'dog', 'dog', 'cat', 'snake', 'cat', 'dog', 'dog'],
        'age': [2.5, 3, 0.5, np.nan, 5, 2, 4.5, np.nan, 7, 3],
        'visits': [1, 3, 2, 3, 2, 3, 1, 1, 2, 1],
        'priority': ['yes', 'yes', 'no', 'yes', 'no', 'no', 'no', 'yes', 'no', 'no']}
labels = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']
```

a. 创建 DataFrame 对象。

b. 获取前三行的数据。

c. 获取所有行，'animal'和'age'列。

d. 获取 3,4,8 行，'animal'和'age'列。

e. 获取 'visits'的值大于等于 3 的数据。

f. 获取'animal'的值是’cat’的数据。

g. 获取'age'不为空的数据。