第 3 章

CHAPTER 3

Pandas 基础

九层之台，始于垒土。

——《老子》

使用 Python 进行数据挖掘和数据分析时，一个必不可少的利器就是 Pandas 库。利用 Pandas 库可以快速地完成数据读写、数据分片/分组统计、数据整理等操作。Pandas 的所有功能都是构建在两个最基础的数据结构之上：Series 与 DataFrame。本章将围绕这两个基础的数据结构，详细讨论以下内容：

- 如何理解 DataFrame，以及它的基本要素；
- DataFrame 中的数据类型；
- 如何选择 DataFrame 中的某一列；
- 什么是 Series，以及针对 Series 的链式方法；
- 如何对 DataFrame 的列名、行名、索引进行修改；
- 如何选择 DataFrame 中的多列；
- 面向 DataFrame 的操作符。

3.1 什么是 DataFrame

Pandas 中有两种基础数据类型：Series 与 DataFrame。Series 是一种类似于一维数组的对象，由一组数据（各种 NumPy 数据类型）以及一组与之相关的数据标签（即索引）组成。而 DataFrame 则是一个表格型的数据结构，包含一组有序的列，每列可以是不同的值类型（数值、字符串、布尔型等），DataFrame 既有行索引，也有列索引，可以看作是由 Series 组成的字典。在程序中要使用 Pandas，第一步需要导入该库。为了方便，通常在导入 Pandas 时会给它一个别名“pd”，后面如果需要使用该库，只需要输入 pd 就可以了。此外，在 Jupyter Notebook 中，我们还可以根据自己的需要设置数据显示多少行、多少列，超出设置的部分就会用省略号代替。下面的代码完成了 Pandas 库的导入，同时设置 Jupyter Notebook 中显示数据时，最多显示 7 行 6 列，超出部分用省略号代替。

```
>>> import pandas as pd
 # set max display row,columns
>>> pd.set_option('display.max_rows', 7)
>>> pd.set_option('display.max_columns',6)
```

3.1.1 DataFrame 的基本要素

DataFrame 有 3 个基本要素，它们分别是索引、列、数据。首先运行如下代码。

```
>>> retail_data = pd.read_csv("../data/Online_Retail_Fake.csv")
>>> retail_data.head()
```

如果是在 Jupyter Notebook 中运行上述代码，那么会得到如图 3.1 所示的输出。

	InvoiceNo	StockCode	Description	...	UnitPrice	CustomerID	Country
0	536365	85123A	WHITE HANGING HEART T-LIGHT HOLDER	...	2.55	17850.0	United Kingdom
1	536365	71053	WHITE METAL LANTERN	...	NaN	17850.0	United Kingdom
2	536365	84406B	NaN	...	2.75	17850.0	United Kingdom
3	536365	84029G	KNITTED UNION FLAG HOT WATER BOTTLE	...	3.39	17850.0	United Kingdom
4	536365	84029E	RED WOOLLY HOTTIE WHITE HEART.	...	3.39	17850.0	United Kingdom

图 3.1　DataFrame 三要素

上面的代码中，第一行首先使用了 read_csv()函数来将本地目录下的数据文件 Online_Retail_Fake.csv 读入到一个 DataFrame 中。关于读取数据部分，我们在第 8 章会有更深入的介绍。读者现在只需要知道 read_csv()函数可以将 CSV 格式的文件的内容读入到一个 DataFrame 中就可以了。现在数据已经读入到 retail_data 这个 DataFrame 中，我们就可以利用 DataFrame 对象中的方法(函数)head()来查看数据内容，在 Jupyter 中 DataFrame 的列名以及索引均以黑体显示，如图 3.1 所示。一个 DataFrame 包含了 3 个基本组件：索引、列和数据。通过指定对应的索引标签(Index Label)和列可以快速访问一个 DataFrame 的指定行或列的数据。在多个 Series 或 DataFrame 合并时，索引会用来做数据对齐。实际上 DataFrame 把索引和列都当作轴(axis)，一个 DataFrame 有一个竖轴(Index)和一个横轴(Columns)。Pandas 借用了 NumPy 的命名，分别用 0/1 来表示竖/横轴。图 3.1 中最左边一列的 0,1,2,3,4 就是 DataFrame 的索引，我们也把索引(Index)称为 DataFrame 的 0 轴，后面经常会遇到在函数中指定参数 axis＝0 的情况，就是指函数是作用于行。而具体的每一个索引标签(Index Label)对应于一行数据，如图 3.1 中的 3 就对应了数据的第 4 行(Pandas 中的索引也采用了从 0 开始计数的方式)。图中最顶端一行则代表了该数据的列(Columns)，也可以用 axis＝1 表示。例如，StockCode 就是其中一列的名称，正如图中显示的一样，每一列的数据都是同一类型，要么是数字，要么是字符串。虽然 DataFrame 中每一列数据要求是相同类型，但是不同列却可以是不同数据类型。对于数据缺失的情况，Pandas 中采用了 NaN 进行表示。实际数据分析中的数据经常是几千甚至几万行，所以为了方便分析，Jupyter Notebook 提供了设置最多显示多少行、多少列的功能。对于超出设置的部分，Jupyter Notebook 会用省略号进行表示，如图 3.1 所示。Jupyter Notebook 还具有代码补齐功能，按下 Tab 键可以帮助补齐后面的代码。如图 3.2 所示，按下 Tab 键后，Notebook 中会给出对应 retail_data 对象的方法，可以从列表中选择对应方法来补齐代码。

图 3.2 代码补齐

如果对某一个函数的使用存在任何疑问，也可以在函数名称后面输入“?”后按下 Shift＋Enter 组合键运行该行代码来查看函数帮助，如图 3.3 所示。

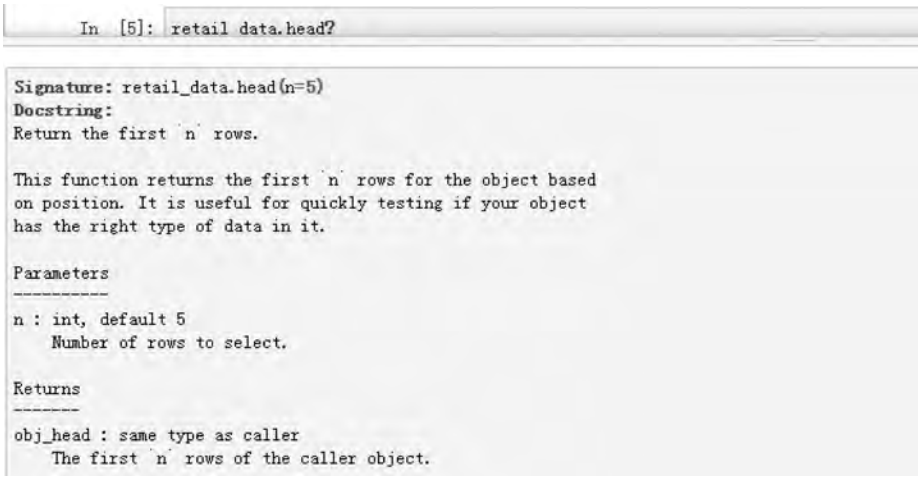

图 3.3 查看函数帮助

由于 index、columns、values 都是 DataFrame 对象中的属性，所以我们可以直接访问这些属性。例如，我们可以用如下代码来获取上述属性的值。

```
>>> index = retail_data.index
>>> columns = retail_data.columns
>>> data = retail_data.values
```

上面 3 行代码直接获取了 retail_data 对象的 index、columns、values 属性，并赋值给了变量 index、columns、data。接下来，运行如下代码依次查看这 3 个变量的值。

```
>>> index
RangeIndex(start = 0, stop = 541909, step = 1)
>>> columns
Index(['InvoiceNo', 'StockCode', 'Description', 'Quantity',
 'InvoiceDate','UnitPrice', 'CustomerID', 'Country'],
       dtype = 'object')
>>> data
array([['536365', '85123A', 'WHITE HANGING HEART T - LIGHT HOLDER', ...,2.55, 17850.0, 'United
Kingdom'],['536365', '71053', 'WHITE METAL LANTERN', ..., nan, 17850.0,'United Kingdom'],
```

```
   ['536365', '84406B', nan, ..., 2.75, 17850.0,
'United Kingdom'], ..., ['581587', '23254',
'CHILDRENS CUTLERY DOLLY GIRL ', ..., 4.15, 12680.0, 'France'],
       ['581587', '23255', 'CHILDRENS CUTLERY CIRCUS PARADE', ...,
  4.15, 12680.0, 'France'], ['581587', '22138',
  'BAKING SET 9 PIECE RETROSPOT ', ..., 4.95,
        12680.0, 'France']], dtype = object)
```

从输出内容可知 index 变量为 RangeIndex 类型，columns 变量为 Index 类型，而 data 变量为 ndarray 类型。通过 issubclass()方法可以发现 RangeIndex 本质上就是 Index 的子类，代码如下。

```
>>> type(index)
pandas.core.indexes.range.RangeIndex
>>> type(columns)
pandas.core.indexes.base.Index
>>> type(data)
numpy.ndarray
>>> issubclass(pd.RangeIndex,pd.Index)
True
```

index 的子类除了 RangeIndex 外，还包括 CategoricalIndex、MultiIndex、IntervalIndex、Int64Index、UInt64Index、Float64Index、RangeIndex、TimedeltaIndex、DatetimeIndex、PeriodIndex 等，在后面的学习中读者也会逐渐接触到它们。我们知道 Pandas 基于 NumPy 而构建，因此它大量地使用了 ndarray，例如，index、columns、data 底层实际都是 ndarray，下面的代码就是很好的说明。

```
>>> index.values
array([ 0, 1, 2, ..., 541906, 541907, 541908], dtype = int64)
>>> columns.values
array(['InvoiceNo', 'StockCode', 'Description', 'Quantity',
'InvoiceDate','UnitPrice', 'CustomerID', 'Country'], dtype = object)
```

3.1.2 数据类型

数据分析中通常将数据简单地分为连续变量和离散变量，而 Pandas 对数据的类型有更详细的分类。表 3.1 给出了 Pandas 中的主要数据类型。

表 3.1 Pandas 中的数据类型

Pandas Type	Python Type	NumPy Type	使用场景
object	str	str	文本
int64	int	int，int8，int16，int32，int64，uint8，uint16，uint32，uint64	整数
float64	float	float	浮点数
bool	bool	bool	布尔类型
datetime64	NA	NA	日期
timedelta[ns]	NA	NA	日期间隔
category	NA	NA	分类变量

通过 3.1.1 节,我们已经知道 DataFrame 的每一列只能是一种数据类型,而不同列则可以是不同数据类型。如果要具体了解一个 DataFrame 对象中列的数据类型,可以通过 dtypes 属性来查看,代码如下。

```
>>> retail_data.dtypes
InvoiceNo        object
StockCode        object
Description      object
Quantity          int64
InvoiceDate      object
UnitPrice       float64
CustomerID      float64
Country          object
dtype: object
```

这里 retail_data 的 UnitPrice 列中每个数据都是 64 位浮点数,而 Quantity 列则都是 64 位整数。对应 object 类型,它可以是任何的 Python 对象,不过通常在数据分析中遇到的 object 类型都是字符串。要方便地查看 DataFrame 中每种数据类型各有几列,可以通过 get_dtype_counts()方法,代码如下。

```
>>> retail_data.get_dtype_counts()
float64    2
int64      1
object     5
dtype: int64
```

3.1.3 了解 Series

Series 就是构成 DataFrame 的列,每一列就是一个 Series。一个 Series 是一个一维的数据类型,其中每一个元素都有一个标签,类似于 NumPy 中元素带标签的数组。其中,标签可以是数字或字符串。如果访问 DataFrame 中的某一列,可以采用如下代码。

```
>>> retail_data['Country']
0          United Kingdom
1          United Kingdom
                ...
541907             France
541908             France
Name: Country, Length: 541909, dtype: object
>>> type(retail_data.Country)
pandas.core.series.Series
```

这一段代码访问的是 Country 列,正如我们所料,返回的是一个 Series。它包含 Index 和数据,同时该 Series 的 Name(即列名)为 Country,数据类型为 object,数据长度为 541909。除了用“[]”操作符来访问某一列,DataFrame 还提供了另一种用“.”操作符来访问列的方法,代码如下。

```
>>> retail_data.Country
```

```
0          United Kingdom
1          United Kingdom
                ...
541907             France
541908             France
Name: Country, Length: 541909, dtype: object
```

通常情况下，我们不建议采用这种方法，原因有二：其一，如果对应的列名包含特殊字符，这种方法可能无效，如果列名为 Country Name，那么无法通过代码 retail_data.Country Name 来访问对应列；其二，如果列名与 DataFrame 的方法重名，那么也会无效，如果数据中某列名为 mean，此时也是无法采用该方法访问列的。获取了某一列数据后，Python 中支持的大部分运算操作符都可以应用于 Series，代码如下。

```
>>> retail_data['UnitPrice']
0          2.55
1           NaN
           ...
541907     4.15
541908     4.95
Name: UnitPrice, Length: 541909, dtype: float64
>>> retail_data['UnitPrice'] + 1
0          3.55
1           NaN
           ...
541907     5.15
541908     5.95
Name: UnitPrice, Length: 541909, dtype: float64
>>> retail_data['UnitPrice'] * 2
0          5.1
1          NaN
          ...
541907     8.3
541908     9.9
Name: UnitPrice, Length: 541909, dtype: float64
>>> retail_data['UnitPrice']> 2
0           True
1          False
           ...
541907      True
541908      True
Name: UnitPrice, Length: 541909, dtype: bool
```

上述代码分别将 UnitPrice 列的每个数据都加 1，乘以 2，将 UnitPrice 列中的每个数据与 2 比较，返回取值为 True/False 的新 Series。

除了数值操作，还可以将一个 Series 与字符串进行比较，例如：

```
>>> country = retail_data['Country']
>>> country == 'France'
0          False
1          False
```

```
            ...
541907      True
541908      True
Name: Country, Length: 541909, dtype: bool
```

这里将 Country 这个 Series 中的每个数据与字符串 France 进行逻辑比较，返回一个新的取值为 True/False 的 Series。数据分析中会大量用到类似的布尔判断来对数据进行过滤，后续章节会对此进行更详细的介绍。

3.1.4 链式方法

在 Python 这一面向对象的高级语言中，一切皆对象，只要是对象，就有着自己的属性和方法，而对这些属性和方法的调用可能会返回新的对象和方法。这种通过“.”操作来顺序调用对象方法的方式称为链式方法。在 Pandas 库中，DataFrame 和 Series 的很多操作都将返回一个新的 DataFrame 或 Series，此时我们又可以对新的 DataFrame 和 Series 进行方法调用，下面来看一段关于链式方法的代码。

```
>>> customers = retail_data['CustomerID']
>>> customers.value_counts().head()
17841.0    7983
14911.0    5903
14096.0    5128
12748.0    4642
14606.0    2782
Name: CustomerID, dtype: int64
>>> retail_data['Country'].value_counts().head()
United Kingdom    495477
Germany             9495
France              8557
EIRE                8196
Spain               2533
Name: Country, dtype: int64
```

上述第一段代码先获取了 CustomerID 列，并将其赋值给了一个新变量 customers。之后用 value_counts()方法统计了 customers 这个 Series 中每个顾客的购物记录总数，最后用 head()方法显示前 5 行。而第二段代码直接对 Country 列中来自不同国家的购物记录进行汇总，之后再显示前 5 行。数据分析中经常会用到链式方法来对 DataFrame 或 Series 中的缺失值进行统计，例如：

```
>>> retail_data['UnitPrice'].isnull()
0         False
1          True
          ...
541907    False
541908    False
Name: UnitPrice, Length: 541909, dtype: bool
>>> retail_data['UnitPrice'].isnull().sum()
3
```

第一段代码使用 isnull()方法来判断 UnitPrice 列中的每一行数据是否缺失，如果是，则返回 True。最终将得到一个新的 Series，它的长度与 UnitPrice 列一样。通过链式方法可以继续调用 sum()函数，该函数将对 Series 求和，这样就可以得出 UnitPrice 列共有 3 行数据缺失。当然，除了采用求和的方式统计总的缺失数据个数，也可以用求平均值函数 mean()来获取缺失数据所占比例，代码如下。

```
>>> retail_data['UnitPrice'].isnull().mean()
5.535984824020269e-06
```

此外，也可以利用 fillna()函数对缺失数据进行填充，之后再利用 mean()函数查看缺失数据，此时可以发现 UnitPrice 中已经没有缺失数据了，代码如下。

```
>>> retail_data['UnitPrice'].fillna(0).isnull().mean()
0.0
```

fillna()函数是处理缺失数据时常用到的一个方法，在后续章节还将更详细地介绍该方法。同时，对于任何函数，在 Jupyter 中都可以通过“?”操作符来获取详细的使用说明，读者可以通过帮助详细了解该函数的使用。

另一个 Pandas 新用户经常会遇到的问题就是 DataFrame 和 Series 中大部分对原有数据的修改操作都是产生一个全新的 Series 或 DataFrame，而不是对原数据的修改。例如，前面的代码 retail_data['UnitPrice'].fillna(0)产生了一个新的 Series，那么对它修改不会对原来的 DataFrame(即 retail_data)产生任何改变。通过运行如下代码可以得到验证。

```
>>> retail_data['UnitPrice'].fillna(0).head()
0    2.55
1    0.00
2    2.75
3    3.39
4    3.39
Name: UnitPrice, dtype: float64
>>> retail_data.head()
0    2.55
1     NaN
2    2.75
3    3.39
4    3.39
Name: UnitPrice, dtype: float64
```

如果想直接在原 DataFrame 上进行修改，通常只需要在函数中加上参数 inplace=True 就可以了。

3.2 索引与列

3.2.1 修改索引与列

在数据分析过程中经常会遇到需要修改 DataFrame 中的索引与列名的问题，这里以 gapminder 数据集为例进行讲解，首先读入该数据，代码如下。

```
>>> gapminder = pd.read_csv("../data/gapminder.csv")
>>> gapminder.head()
```

显示结果如图 3.4 所示。

	Unnamed: 0	1800	1801	...	2015	2016	Life expectancy
0	0	NaN	NaN	...	NaN	NaN	Abkhazia
1	1	28.21	28.20	...	NaN	NaN	Afghanistan
2	2	NaN	NaN	...	NaN	NaN	Akrotiri and Dhekelia
3	3	35.40	35.40	...	NaN	NaN	Albania
4	4	28.82	28.82	...	NaN	NaN	Algeria

5 rows × 219 columns

图 3.4 gapminder 原始显示结果

从数据的显示看出 gapminder 采用了默认的索引值，取值为 0，1，…然而分析中想要的是用 Life expectancy 列的数据作为 gapminder 的索引，因此可以用如下代码完成修改。

```
>>> gapminder = pd.read_csv("../data/gapminder.csv",
    index_col = 'Life expectancy')
>>> gapminder.head()
```

修改结果如图 3.5 所示。

Life expectancy	Unnamed: 0	1800	1801	...	2014	2015	2016
Abkhazia	0	NaN	NaN	...	NaN	NaN	NaN
Afghanistan	1	28.21	28.20	...	NaN	NaN	NaN
Akrotiri and Dhekelia	2	NaN	NaN	...	NaN	NaN	NaN
Albania	3	35.40	35.40	...	NaN	NaN	NaN
Algeria	4	28.82	28.82	...	NaN	NaN	NaN

5 rows × 218 columns

图 3.5 修改结果(1)

通过 index_col='Life expectancy'参数可以让 read_csv()函数在读取数据时就将某列设置为索引。当然，Pandas 也支持在读入数据后再进行修改，代码如下。

```
>>> gapminder = pd.read_csv("../data/gapminder.csv")
>>> gapminder.set_index('Life expectancy')
>>> gapminder.head()
```

修改结果如图 3.6 所示。set_index()函数可以将指定的列设置为对应 DataFrame 对象的 Index，但是从代码输出来看似乎设置没有成功，我们用 gapminder.head()看到的还是原来的 Index。细心的读者应该会记得前面已经指出 Pandas 中大部分函数执行后返回的都是一个新的对象。因此，如果想在原来的对象上修改，有两种方法：一种就是使用 inplace=True 参数；另一种就是将返回的对象赋值给原来的变量，代码如下。

	Unnamed: 0	1800	1801	...	2015	2016	Life expectancy
0	0	NaN	NaN	...	NaN	NaN	Abkhazia
1	1	28.21	28.20	...	NaN	NaN	Afghanistan
2	2	NaN	NaN	...	NaN	NaN	Akrotiri and Dhekelia
3	3	35.40	35.40	...	NaN	NaN	Albania
4	4	28.82	28.82	...	NaN	NaN	Algeria

5 rows × 219 columns

图 3.6　修改结果(2)

```
>>> gapminder = pd.read_csv("../data/gapminder.csv")
>>> gapminder = gapminder.set_index('Life expectancy')
>>> gapminder.head()
```

既然可以设置索引,那么必然有将其恢复到列的方法,如下代码就可以将索引重新恢复到列中,结果如图 3.7 所示。

```
>>> gapminder.reset_index()
```

	Life expectancy	Unnamed: 0	1800	...	2014	2015	2016
0	Abkhazia	0	NaN	...	NaN	NaN	NaN
1	Afghanistan	1	28.21	...	NaN	NaN	NaN
...	...	...	...	...	...	...	...
778	Åland	258	NaN	...	NaN	NaN	NaN
779	South Sudan	259	NaN	...	56.1	56.1	56.1

780 rows × 219 columns

图 3.7　修改结果(3)

修改后返回一个新的 DataFrame,如果想在原来的 DataFrame 上修改,请使用 inplace=True 参数。掌握了索引的修改,接下来看看如何同时修改索引与列名,示例代码如下。

```
>>> gapminder = pd.read_csv("../data/gapminder.csv",
    index_col = 'Life expectancy')
>>> idx_rename = {'Abkhazia':'Abk', 'Afghanistan': 'Afg'}
>>> col_rename = {'Unnamed: 0':'Wrong Column', '1801': '1801Y'}
##再次强调返回的是一个新的 DataFrame 而不是对原有 DataFrame 的修改
>>> gapminder_new = gapminder.rename(index = idx_rename,
    columns = col_rename)
>>> gapminder_new.head()
```

修改结果如图 3.8 所示。上述代码通过 rename()函数完成了对 gapminder 的索引和列名的修改。其中,idx_rename 和 col_rename 均为字典对象,字典的 key 对应了需要修改的内容,value 则是对应要修改成的那个值。同时,index 和 columns 也都可以转换为列表对象,可以通过直接将一个列表赋值给 index 和 columns 的方法来完成修改,具体代码如下。

Life expectancy	Wrong Column	1800	1801Y	...	2014	2015	2016
Abk	0	NaN	NaN	...	NaN	NaN	NaN
Afg	1	28.21	28.20	...	NaN	NaN	NaN
Akrotiri and Dhekelia	2	NaN	NaN	...	NaN	NaN	NaN
Albania	3	35.40	35.40	...	NaN	NaN	NaN
Algeria	4	28.82	28.82	...	NaN	NaN	NaN

5 rows × 218 columns

图 3.8 修改结果(4)

```
>>> index = gapminder.index
>>> columns = gapminder.columns
>>> index_list = index.tolist()
>>> column_list = columns.tolist()
>>> print(index_list[0])
Abkhazia
>>> index_list[0] = 'Abk'
>>> index_list[1] = 'Afg'
>>> column_list[0] = 'Wrong Column'
>>> gapminder.index = index_list
>>> gapminder.columns = column_list
>>> gapminder.head()
```

修改结果如图 3.9 所示。

	Wrong Column	1800	1801	...	2014	2015	2016
Abk	0	NaN	NaN	...	NaN	NaN	NaN
Afg	1	28.21	28.20	...	NaN	NaN	NaN
Akrotiri and Dhekelia	2	NaN	NaN	...	NaN	NaN	NaN
Albania	3	35.40	35.40	...	NaN	NaN	NaN
Algeria	4	28.82	28.82	...	NaN	NaN	NaN

5 rows × 218 columns

图 3.9 修改结果(5)

正如我们所期望的,gapminder 的列名和前两行的索引标签都已经修改为想要的内容。

3.2.2 添加、修改或删除列

在数据分析过程中经常需要添加新的列,或对当前列的数据进行修改,有时还需要删除某些列。本节将一一讨论如何实现它们。首先使用如下代码读入新的数据,并查看前 5 行。

```
>>> retail_data = pd.read_csv("../data/Online_Retail_Fake.csv")
>>> retail_data.head()
```

如图 3.10 所示,retail_data 数据包含了单价 UnitPrice 列和数量 Quantity 列,但是没有总价信息,分析中如果想添加新的一列 TotalPrice 到 retail_data 中,那么可以采取如下方法,结果如图 3.11 所示。

	InvoiceNo	StockCode	Description	...	UnitPrice	CustomerID	Country
0	536365	85123A	WHITE HANGING HEART T-LIGHT HOLDER	...	2.55	17850.0	United Kingdom
1	536365	71053	WHITE METAL LANTERN	...	NaN	17850.0	United Kingdom
2	536365	84406B	NaN	...	2.75	17850.0	United Kingdom
3	536365	84029G	KNITTED UNION FLAG HOT WATER BOTTLE	...	3.39	17850.0	United Kingdom
4	536365	84029E	RED WOOLLY HOTTIE WHITE HEART.	...	3.39	17850.0	United Kingdom

5 rows × 8 columns

图 3.10　读入数据

```
>>> retail_data['Total_price'] = retail_data['UnitPrice'] * \
        retail_data['Quantity']
>>> retail_data[['Total_price','UnitPrice','Quantity']]
```

	Total_price	UnitPrice	Quantity
0	15.30	2.55	6
1	NaN	NaN	6
2	22.00	2.75	8
...	...	...	...
541906	16.60	4.15	4
541907	16.60	4.15	4
541908	14.85	4.95	3

541909 rows × 3 columns

图 3.11　添加新列

从上述代码可以看出，添加新列是非常简单的，直接给该列赋值就可以完成。上面的第二行代码表示只取这3列数据，在第4章数据筛选中还会更详细地介绍，现在只需要理解['Total_price','UnitPrice','Quantity']代表只取这个列表中的列就可以了。默认情况下采用上述代码增加的新列都会作为最后一列，如果想在DataFrame的中间增加列，此时就需要采用insert()方法。例如，如果要在CustomerID列后添加刚才的数据，代码如下。

```
>>> totalPrice_index = retail_data.columns.get_loc
('CustomerID') + 1
>>> totalPrice_index
7
>>> retail_data.insert(loc = totalPrice_index,
                       column = 'New_totalPrice',
                       value = retail_data['Total_price'])
>>> retail_data.head()
```

运行结果如图3.12所示。

	InvoiceNo	StockCode	Description	Quantity	...	CustomerID	New_totalPrice	Country	Total_price
0	536365	85123A	WHITE HANGING HEART T-LIGHT HOLDER	6	...	17850.0	15.300000	United Kingdom	15.300000
1	536365	71053	WHITE METAL LANTERN	6	...	17850.0	27.666699	United Kingdom	27.666699
2	536365	84406B	NaN	8	...	17850.0	22.000000	United Kingdom	22.000000
3	536365	84029G	KNITTED UNION FLAG HOT WATER BOTTLE	6	...	17850.0	20.340000	United Kingdom	20.340000
4	536365	84029E	RED WOOLLY HOTTIE WHITE HEART.	6	...	17850.0	20.340000	United Kingdom	20.340000

5 rows × 10 columns

图 3.12　insert ()插入列

从代码输出可以看出新的New_totalPrice列位于CustomerID后。了解了如何在DataFrame中添加新列后，下面来看一下如何删除列。删除列需要使用drop()方法，如果想删除New_totalPrice列，则可以运行如下代码。

```
>>> retail_data.drop('New_totalPrice',axis = 'columns')
```

运行结果如图 3.13 所示。

	InvoiceNo	StockCode	Description	Quantity	...	UnitPrice	CustomerID	Country	Total_price
0	536365	85123A	WHITE HANGING HEART T-LIGHT HOLDER	6	...	2.550000	17850.0	United Kingdom	15.300000
1	536365	71053	WHITE METAL LANTERN	6	...	4.611117	17850.0	United Kingdom	27.666699
...	...	...	...	...	...	...	...	...	...
541908	581587	22138	BAKING SET 9 PIECE RETROSPOT	3	...	4.950000	12680.0	France	14.850000
541909	581587	22138	Wrong booking	3	...	4.950000	12680.0	France	14.850000

541910 rows × 9 columns

图 3.13 删除列

需要注意的是，drop()方法返回的是一个新的 DataFrame，而不是在原来的 DataFrame 进行列的删除。因此，如果需要保留这个新的 DataFrame，可以使用参数 inplace=True 来指明修改在原来的 DataFrame 上进行。或者也可以将返回的 DataFrame 赋值给一个新的变量。除了采用 drop()方法以外，还可以用如下代码直接在原来的 DataFrame 中删除列。

```
>>> del retail_data['New_totalPrice']
```

除了常规的数学运算，还可以对 DataFrame 的列进行逻辑运算，之后将其赋值给新的列，例如：

```
>>> retail_data['key_customer'] = retail_data['Total_price']> 1000
>>> retail_data['key_customer'].sum()
388
```

通过上述代码将购物总金额超过 1000 的定义为关键客户，首先将 Total_price 列的数据与 1000 比较，此时返回的是一个取值为 True/False 的 Series。然后将该 Series 添加到 retail_data 中，命名该列为 key_customer，对该列求和就可以得出关键客户总数为 388。这种将 DataFrame 的列与某个值进行比较后得到一个取值为布尔值 Series 的方法在后面章节还会多次使用，希望读者能熟练掌握。

3.3 选择多列

前面已经学习了如何选择 DataFrame 中的某一列，那么如果想选择多列，应该怎么操作呢？例如，选择 retail_data 中的 3 列，依次为 Total_price、UnitPrice、Quantity，那么可以用如下代码完成，结果如图 3.14 所示。

```
>>> retail_data[['Total_price','UnitPrice','Quantity']]
```

上述代码中用了一个列表（即['Total_price','UnitPrice','Quantity']）作为变量在 retail_data 中指出想要选择的列，此时将返回一个新的 DataFrame，即使该列表中只有一个元素，返回的也是一个 DataFrame。下面的代码可以帮助我们对此有更好的理解。

```
>>> type(retail_data[['Total_price']])
pandas.core.frame.DataFrame
>>> type(retail_data['Total_price'])
```

	Total_price	UnitPrice	Quantity
0	15.300000	2.550000	6
1	27.666696	4.611116	6
...	...	...	...
541907	16.600000	4.150000	4
541908	14.850000	4.950000	3

541909 rows × 3 columns

图 3.14 选择结果(1)

```
pandas.core.series.Series
```

利用 type()函数进行检查，可以确认输入为列表时返回的是 DataFrame，而直接输入 Total_price 返回的则是 Series。除了通过明确指定列名的方式来选择列以外，还可以根据列的数据类型或列的名称字符串匹配来选择列，代码如下。

```
>>> retail_data.get_dtype_counts()
float64    4
int64      1
object     5
dtype: int64
>>> retail_data.select_dtypes(include = ['float64']).head()
```

运行结果如图 3.15 所示。

	UnitPrice	CustomerID	New_totalPrice	Total_price
0	2.550000	17850.0	15.300000	15.300000
1	4.611116	17850.0	27.666696	27.666696
2	2.750000	17850.0	22.000000	22.000000
3	3.390000	17850.0	20.340000	20.340000
4	3.390000	17850.0	20.340000	20.340000

图 3.15　选择结果(2)

第一段代码统计了 retail_data 中各种数据类型的数量，其中 float64 类型有 4 列。第二段代码则通过 select_dtypes()函数指定选择 float64 数据类型的列。当然，Pandas 也支持一次选择多种数据类型，例如：

```
>>> retail_data.select_dtypes(include = ['float64','int']).head()
```

这段代码同时选择了 float64 和 int 类型的列。除了根据数据类型选择列，Pandas 中还可以根据列名称来进行选择，代码如下。

```
#字符串部分匹配
>>> retail_data.filter(like = 'Price')

#正则表达式
>>> retail_data.filter(regex = '_').head()
```

运行结果如图 3.16 和图 3.17 所示。

	UnitPrice	New_totalPrice
0	2.550000	15.300000
1	4.611116	27.666696
2	2.750000	22.000000
...	...	...
541906	4.150000	16.600000
541907	4.150000	16.600000
541908	4.950000	14.850000

541909 rows × 2 columns

图 3.16　选择结果(3)

	New_totalPrice	Total_price
0	15.300000	15.300000
1	27.666696	27.666696
2	22.000000	22.000000
3	20.340000	20.340000
4	20.340000	20.340000

图 3.17　选择结果(4)

filter()方法有3个互斥的参数，分别为items、regex和like。通过前面的代码已经知道了regex和like参数的用法，接下来看看items参数，该参数实际对应的就是列名。例如，如下代码将选择Total_price和UnitPrice列，结果如图3.18所示。

```
>>> retail_data.filter(items = ['Total_price','UnitPrice']).head()
```

使用items参数的一个好处是，如果items中的列名不存在，此时不会返回KeyError错误，而只是返回包含正确列的DataFrame，代码如下。

```
>>> retail_data.filter(items = ['Total_price','UnitPrice',
    'wrong']).head()
```

	Total_price	UnitPrice
0	15.300000	2.550000
1	27.666696	4.611116
2	22.000000	2.750000
3	20.340000	3.390000
4	20.340000	3.390000

图3.18 选择结果(5)

前面学习了如何选择特定列，利用这一方法还可以对数据中的列进行重排。例如，根据列数据类型(连续变量或离散变量)组织DataFrame中的列，或者将关联的列顺序排列。如果从数据分析角度看，retail_data中UnitPrice、Quantity、Total_price列是一组关联数据，InvoiceNo、InvoiceDate、StockCode、Description列为另一组数据，CustomerID和Country为一组，那么可以采用如下方式完成列的重排。

```
>>> price_info = ['UnitPrice','Quantity','Total_price']
>>> invoice_info = ['InvoiceNo','InvoiceDate','StockCode',
    'Description']
>>> customer_info = ['CustomerID','Country']
>>> columns_new = price_info + invoice_info + customer_info
>>> set(columns_new) == set(retail_data.columns)
True
```

上述代码首先定义了3个列表，分别包含了想要的3组列的名称。然后将3个列表汇总到columns_new这个新的列表，它里面的列名称实际上和原来的retail_data一样，只是列的排序不一样。通过集合比较的输出也可以确认二者是一致的。直接将columns_new这个列表传入到retail_data的“[]”操作符就完成了列的选择，新的DataFrame中列的顺序即是所期望的顺序了，代码如下。

```
>>> retail_data_new = retail_data[columns_new]
>>> retail_data_new.head()
```

运行结果如图3.19所示。

	UnitPrice	Quantity	Total_price	InvoiceNo	InvoiceDate	StockCode	Description	CustomerID	Country
0	2.550000	6	15.300000	536365	2010/12/1 8:26	85123A	WHITE HANGING HEART T-LIGHT HOLDER	17850.0	United Kingdom
1	4.611117	6	27.666699	536365	2010/12/1 8:26	71053	WHITE METAL LANTERN	17850.0	United Kingdom
2	2.750000	8	22.000000	536365	2010/12/1 8:26	84406B	NaN	17850.0	United Kingdom
3	3.390000	6	20.340000	536365	2010/12/1 8:26	84029G	KNITTED UNION FLAG HOT WATER BOTTLE	17850.0	United Kingdom
4	3.390000	6	20.340000	536365	2010/12/1 8:26	84029E	RED WOOLLY HOTTIE WHITE HEART.	17850.0	United Kingdom

图3.19 列的重排