第 5 章

数 据 转 换

5.1 基础知识

本节主要介绍一下计算机的 3 种程序结构及 Pandas 中常用的 for 循环语句。

5.1.1 程序结构

计算机的程序设计中有 3 种基本的结构：顺序结构、分支结构和循环结构。

1. 顺序结构

顺序结构是程序设计中最简单的结构，它的执行顺序是自上而下的，即依次执行，代码如下：

```
import pandas as pd
df = pd.read_excel('demo_.xlsx')
a = df.nunique() == 5
cols = a[a].index
cols
```

代码依据编写的顺序依次被执行(先定义变量再调用变量)，输出的结果如下：

```
Index(['Name','BMI'],dtype = 'object')
```

2. 分支结构

分支结构主要是使用 if 条件语句。其语句是：if 关键字后紧跟条件测试表达式。

当语句中存在条件判断时，需要做分支选择。当条件符合时，选择哪个；当条件不符合时，返回的结果又是哪个。分支结构有单选(if)、二选一(if-else)、多选一(if-elif-else)。

除了简单的 if 语句、if-else、if-elif-else 语句，也经常会用到 if 语句的嵌套。例如：if 语句中嵌套 if-else，if-else 中嵌套 if-else；又例如：for 循环中的 if 语句、while 循环中的 if 语句等。

在每个 if 或 elif 语句的后面，加冒号后换行缩进，再开始另外一行代码。缩进编写是 Python 代码的特色之一。

3. 循环结构

在 Python 数据分析中，循环语句是必不可少的语句。常见的循环有 for 循环与 while 循环。for 循环一般用于循环次数确定的场合，例如列表循环、字典循环等，而 while 循环一般用于循环次数不确定的场合。在 Pandas 数据分析过程中，使用最多的循环为 for 循环。

在 Pandas 的 for 循环使用过程中，会经常嵌套分支结构，从而形成了类似 for-if、for-if-else、for-if-for、for-if(break/continue)、for-pass 等循环结构。

在计算机编程过程中，对于某些很有规律的操作(很容易找到共性的操作)，最简单的实现方式就是采用循环。“循环”，其实是操作过程中某些共性的总结。通过代码化循环操作，可以极大地提高工作效率，代码如下：

```
df = pd.read_excel('demo_.xlsx')
[i for i in df.Score]
```

输出的结果如下：

```
['A', 'A', 'B', 'C', 'B', 'B', 'A']
```

当然，循环也是有代价的，因为逐行循环迭代的原因，从而可使计算机的运行速度大受拖累。例如在 Pandas 中常用的下标索引循环就是一个明显的例子。正如第 4 章所讲，在 Pandas 中，NumPy 的向量化函数的速度是最快的，如果能用 NumPy 或 Pandas 的向量化函数解决的问题，就尽量不要采用循环迭代的方式。

5.1.2 循环语句

1. for-if 循环

在 for 语句后面跟上一个 if 判断语句，用于筛选不符合条件的值，代码如下：

```
df = pd.read_excel('demo_.xlsx')
a = df['Score']          #将 df['Score']赋值给变量 a
l = []
for i in a:
    if not i in l:       #筛选出列表中的重复元素
        l.append(i)      #只有当列表中不存在该值时才会被追加
l
```

输出的结果如下：

```
['A', 'B', 'C']
```

2. for-if-else 循环

分支语句中 else 子句用于执行当其他的条件不满足时的情景，代码如下：

```
df = pd.read_excel('demo_.xlsx',nrows = 3) #nrows = 3, 前面的 3 行
for i in df['Score']:
    if i == 'A':
        print("优")          #当 df['Score'] == 'A'时,显示为"优"
    elif i == 'B':
        print("良")          #当 df['Score'] == 'B'时,显示为"良"
    else:
        print("中")          #当 df['Score']不为'A'或'B'时,显示为"中"
```

输出的结果如下：

```
优
优
良
```

在 Python 中，else 分支子句可配合 for、while 等循环语句使用，还能配合 try…except 语句进行异常处理使用。

3. for…if…for 循环

在二维数组遍历过程中，经常会使用双层 for 循环。如果在循环的过程中需要加上条件判断，则可在 for 语句后面加上一个 if 判断语句，代码如下：

```
df = pd.read_excel('demo_.xlsx')
l = []
for i in df['Score']:
    if i == "A":
        for j in df['Name']:
            if j == 'Kim':
                l.append(i + "_" + j)
l
```

输出的结果如下：

```
['A_Kim', 'A_Kim', 'A_Kim', 'A_Kim', 'A_Kim', 'A_Kim']
```

4. for…if…continue，终止本次循环，进入下一次循环

continue 对前面的 for 起作用，用于终止本次循环，进入下一次循环。其应用原理：跳过符合本次 if 条件筛选的操作，进入下一次 for 循环操作，直至结束，代码如下：

```
lt = ['Kim','Jim','Joe','Tom']
for i in lt:
    if i == 'Tom':
        continue
print(i)
```

输出的结果如下：

```
Kim
Jim
Joe
```

5. for…if…break，终止当前循环

break 跳出的是 for 循环，结束当前的循环，代码如下：

```
lt = ['Kim','Jim','Joe','Tom']
for i in lt:
    if i == 'Joe':
        break
    print(i)
```

输出的结果如下：

```
Kim
Jim
```

注意：*在多层循环中，一个 break 语句只向外跳一层。*

6. for…pass，占位

pass 只是为了保持程序结构的完整性，不会做任何操作，代码如下：

```
lt = ['Kim','Jim','Joe','Tom']
for i in lt:
    print(i)
else:
    pass  #占位
```

输出的结果如下：

```
Kim
Jim
Joe
Tom
```

5.2 映射函数

以下是 map、apply、applymap 这 3 个函数的异同点比较。

(1) map 是 Python 的原生态函数，apply 和 applymap 是 Pandas 的函数。

(2) apply 与 map 的作用对象是 Series,遍历的是 Series 中的每个值。applymap 的作用对象是 DataFrame 中的每个元素,遍历的是 DataFrame 的每个值。

(3) apply 与 map 可以作用于 Series,而 applymap 不可以。

(4) apply 与 applymap 可以通过 axis 来确定作用的方向,而 map 不可以。

(5) apply 与 applymap 可以作用于整个 DataFrame,而 map 不可以。

5.2.1 map()

语法:map(function,iterable,…)。

结果:根据提供的函数对指定序列进行映射。

参数:function 表示函数;iterable 表示一个或多个序列。

说明:map 是 Python 的内置函数。

map()会根据提供的字典或函数,对指定的对象做映射,代码如下:

```
df = pd.read_excel('demo_.xlsx',nrows = 3)
df["状况"] = df["Score"].map({"A":"优","B":"良","C":"中"})
df
```

输出的结果如下:

	Date	Name	City	Age	WorkYears	Weight	BMI	Score	状况
0	2020-12-12	Joe	Beijing	76	35	56	18.86	A	优
1	2020-12-12	Kim	Shanghai	32	12	85	21.27	A	优
2	2020-12-13	Jim	Shenzhen	55	23	72	20.89	B	良

map 的应用场景较为广泛,后续章节会有大量的演示案例可供参考。

5.2.2 apply()

语法:apply(func,axis=0,broadcast=False,raw=False,reduce=None,args=(),** kwds)。

结果:函数作为一个对象,可作为参数传递给其他参数,并且可作为函数的返回值。

参数:func 表示函数、自定义函数或匿名函数,它是 apply 最有用的第一参数。axis 决定第一参数 func 起作用的轴方向,默认值为 axis=0。

apply()是 Pandas 中最为灵活、使用频率最高的函数之一。它可以作用于 Series 或者整个 DataFrame,功能也是自动遍历整个 Series 或者 DataFrame。它可以通过 axis 参数来控制方向,作用于行或者作用于列。后续的章节有大量的 apply()的应用案例。

apply()函数都是以 Series 为单位的,但它可以作用于 DataFrame。apply 主要有 3 种应用方式:第 1 种是用匿名函数(lambda),第 2 种是用自定义函数(def),第 3 种是用函数(例如 Python 函数、NumPy 函数等)。

1. apply 作用于 Series

1）匿名函数实现方式

以下是 apply 的匿名函数的实现方式，代码如下：

```
df = pd.read_excel('demo_.xlsx',nrows = 3)
df['等级'] = ( df['Score']
    .apply(lambda x: "优" if x == 'A' else x)
    .apply(lambda x: "良" if x == 'B' else x)
    .apply(lambda x: "中" if x == 'C' else x))
df
```

输出的结果如下：

```
        Date      Name    City    Age  WorkYears  Weight   BMI    Score  等级
0  2020-12-12    Joe    Beijing   76      35        56    18.86    A     优
1  2020-12-12    Kim    Shanghai  32      12        85    21.27    A     优
2  2020-12-13    Jim    Shenzhen  55      23        72    20.89    B     良
```

在 NumPy 与 Pandas 中，二维数据 axis 默认为 0。以下代码是 axis＝1 的应用举例，axis 用于控制不同 Series 间的组间求和，代码如下：

```
df_ = pd.read_excel('demo_.xlsx')
df_['求和'] = df_.iloc[:,3:7].apply(lambda x: x.sum(),axis = 1)
df_
```

输出的结果如下：

```
        Date      Name    City    Age  WorkYears  Weight   BMI    Score  求和
0  2020-12-12    Joe    Beijing   76      35        56.0   18.86    A    185.86
1  2020-12-12    Kim    Shanghai  32      12        85.0   21.27    A    150.27
2  2020-12-13    Jim    Shenzhen  55      23        72.0   20.89    B    170.89
3  2020-12-13    Tom      NaN     87      33        NaN    21.22    C    141.22
4  2020-12-14    Jim    Guangzhou 93      42        59.0   20.89    B    214.89
5  2020-12-14    Kim    Xiamen    78      36        65.0    NaN     B    179.00
6  2020-12-15    Sam    Suzhou    65      32        69.0   22.89    A    188.89
```

匿名函数是指没有命名的函数，通常在某些需要函数的场合使用，但这个场合的这个函数一般只使用一次，所以就不需特意为其命名。

注意：匿名函数中不能出现 for 或 while 循环语句，因为匿名函数表达式的返回值只能有一个返回值（匿名函数的参数可以为一个或多个，如果参数为多个，则需要用逗号分开）。

2）自定义函数的实现方式

以下是自定义函数的实现方式，代码如下：

```
df = pd.read_excel('demo_.xlsx',nrows = 3)    #取前面的 3 行数据
def AA(s):
    if s == 'A':
        return "优"
    elif s == 'B':
        return "良"
    elif s == 'C':
        return "中"
df['等级'] = df['Score'].apply(AA)
df
```

输出的结果如下：

```
        Date      Name    City    Age  WorkYears  Weight   BMI    Score
0  2020 - 12 - 12  Joe   Beijing   76      35        56    18.86    A
1  2020 - 12 - 12  Kim   Shanghai  32      12        85    21.27    A
2  2020 - 12 - 13  Jim   Shenzhen  55      23        72    20.89    B
```

函数可理解为带名字的代码块，用于完成具体的工作。当需要在程序中多次执行同一个任务时，可以通过事先创建一个函数，然后在使用的过程中实现对这个函数的调用。创建函数（也可以称为定义函数），语法如下：

```
def [函数名]([输入参数] = [默认值]):
    [代码]
    return([输出值])
```

注意：对于自定义函数，即使这个函数没有参数，也必须保留空括号()。如果这个函数带有多个参数，则各参数间应使用逗号(“,”)分隔。

2. apply 作用于 DataFrame

以下是函数（NumPy 的通用函数）的实现方式，代码如下：

```
df = pd.read_excel('demo_.xlsx')
df.iloc[:,3:7] = df.iloc[:,3:7].apply(np.square)
df
```

输出的结果如下：

```
        Date      Name    City     Age  WorkYears  Weight     BMI     Score
0  2020 - 12 - 12  Joe   Beijing   5776    1225     3136.0  355.6996    A
1  2020 - 12 - 12  Kim   Shanghai  1024     144     7225.0  452.4129    A
2  2020 - 12 - 13  Jim   Shenzhen  3025     529     5184.0  436.3921    B
```

```
3  2020-12-13  Tom      NaN      7569  1089    NaN    450.2884  C
4  2020-12-14  Jim   Guangzhou   8649  1764  3481.0   436.3921  B
5  2020-12-14  Kim     Xiamen    6084  1296  4225.0      NaN    B
6  2020-12-15  Sam     Suzhou    4225  1024  4761.0   523.9521  A
```

apply 的第一参数接收的是函数，这个函数可以是前面所讲的自定义函数或匿名函数，也可以是上面的 NumPy 函数等。在本例中，np.square 函数被当作参数传入。

apply 可以通过 axis 来控制作用的行与列，代码如下：

```
df.iloc[:3,3:].apply(lambda x:x.name + '-' + x.astype(str))
#df.iloc[:3,3:].apply(lambda x:x.index + '-' + x.astype(str),axis = 1)
#以上两行代码输出的结果是相同的
```

输出的结果如下：

```
      Age         WorkYears          Weight           BMI          Score
0  Age-5776   WorkYears-1225   Weight-3136.0   BMI-355.6996   Score-A
1  Age-1024    WorkYears-144   Weight-7225.0   BMI-452.4129   Score-A
2  Age-3025    WorkYears-529   Weight-5184.0   BMI-436.3921   Score-B
```

这里给读者留一个问题，供大家思考：为什么代码中第 1 个是 axis=0，而另一个是 axis=1，明明控制的方向不同，但结果为什么最后却还相同呢？

在 apply 中允许函数嵌套。例如 apply 中的 lambda 嵌套，代码如下：

```
df = pd.read_excel('demo_.xlsx')
df.iloc[:,3:7] = (df.iloc[:,3:7]
    .apply(lambda x: x.apply(
            lambda y:str(y) + '-隐退'if y>=65 else y)
      if sum(x)>=185 else x,axis = 1)
)
df
```

输出的结果如下：

```
      Date       Name     City        Age       WorkYears     Weight      BMI    Score
0  2020-12-12   Joe    Beijing   76.0-隐退      35.0          56.0        18.86    A
1  2020-12-12   Kim    Shanghai     32.0        12.0          85.0        21.27    A
2  2020-12-13   Jim    Shenzhen     55.0        23.0          72.0        20.89    B
3  2020-12-13   Tom      NaN        87.0        33.0           NaN        21.22    C
4  2020-12-14   Jim    Guangzhou  93.0-隐退      42.0          59.0        20.89    B
5  2020-12-14   Kim    Xiamen       78.0        36.0          65.0         NaN     B
6  2020-12-15   Sam    Suzhou     65.0-隐退      32.0       69.0-隐退      22.89    A
```

在 DataFrame 中，通过切片(df.iloc[:,3:7])可用新值替换原有的值。

5.2.3 applymap()

df.applymap(func)为其语法结构，返回的值为 DataFrame。

1. 通用函数

第一参数的函数可以是通用函数，代码如下：

```
df = pd.read_excel('demo_.xlsx')
df.applymap(str).info()
```

结果展示如下：

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 7 entries, 0 to 6
Data columns (total 8 columns):
 #  Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   Date       7 non-null      object
 1   Name       7 non-null      object
 2   City       7 non-null      object
 3   Age        7 non-null      object
 4   WorkYears  7 non-null      object
 5   Weight     7 non-null      object
 6   BMI        7 non-null      object
 7   Score      7 non-null      object
dtypes: object(8)
memory usage: 576.0+ Bytes
```

原来的其他数据类型已全部转换为 object 类型。

2. 自定义函数

第一参数的函数可以是自定义函数，代码如下：

```
df = pd.read_excel('demo_.xlsx')
def AA(x):
    return "demo_" + str(x)
df.applymap(AA)
```

输出的结果如图 5-1 所示。

3. 匿名函数

第一参数的函数也可以是匿名函数，代码如下：

```
df = pd.read_excel('demo_.xlsx')
df.applymap(lambda x: x * 10 if isinstance(x, int) and x > 50 else x)
```

	Date	Name	City	Age	WorkYears	Weight	BMI	Score
0	demo_2020/12/12	demo_Joe	demo_Beijing	demo_76	demo_35	demo_56.0	demo_18.86	demo_A
1	demo_2020/12/12	demo_Kim	demo_Shanghai	demo_32	demo_12	demo_85.0	demo_21.27	demo_A
2	demo_2020/12/13	demo_Jim	demo_Shenzhen	demo_55	demo_23	demo_72.0	demo_20.89	demo_B
3	demo_2020/12/13	demo_Tom	demo_nan	demo_87	demo_33	demo_nan	demo_21.22	demo_C
4	demo_2020/12/14	demo_Jim	demo_Guangzhou	demo_93	demo_42	demo_59.0	demo_20.89	demo_B
5	demo_2020/12/14	demo_Kim	demo_Xiamen	demo_78	demo_36	demo_65.0	demo_nan	demo_B
6	demo_2020/12/15	demo_Sam	demo_Suzhou	demo_65	demo_32	demo_69.0	demo_22.89	demo_A

图 5-1　类型转换与文本加工

输出的结果如下：

```
        Date  Name     City   Age  WorkYears  Weight   BMI  Score
0  2020-12-12   Joe  Beijing   760         35    56.0  18.86      A
1  2020-12-12   Kim  Shanghai   32         12    85.0  21.27      A
2  2020-12-13   Jim  Shenzhen  550         23    72.0  20.89      B
3  2020-12-13   Tom      NaN   870         33     NaN  21.22      C
4  2020-12-14   Jim  Guangzhou 930         42    59.0  20.89      B
5  2020-12-14   Kim  Xiamen    780         36    65.0    NaN      B
6  2020-12-15   Sam  Suzhou    650         32    69.0  22.89      A
```

isinstance()是 Python 的内置函数，用来判断对象是否是已知的类型，类似于 type()。它与 type() 的区别：

(1) type()不会认为子类是一种父类类型，不考虑继承关系。

(2) isinstance()会认为子类是一种父类类型，考虑继承关系。如果要判断两种类型是否相同，则推荐使用 isinstance()。

5.3　各类转换

5.3.1　数据类型转换

在 Pandas 中，数据类型的转换在 DataFrame 层面多用 convert_dtypes()，而在 Series 层面多用 astype()。

1. convert_dtypes()

convert_dtypes()可以自动推断数据类型并进行转换。对于 int、float、datetime 类型的数据，全部会自动转换为 64 位，代码如下：

```
df_ = pd.read_excel('demo_.xlsx').convert_dtypes()
df_.info()
```

输出的结果如下：

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 7 entries, 0 to 6
Data columns (total 8 columns):
 #Column      Non-Null Count   Dtype
--  ----      --------------   -----
0   Date        7 non-null     datetime64[ns]
1   Name        7 non-null     string
2   City        6 non-null     string
3   Age         7 non-null     Int64
4   WorkYears   7 non-null     Int64
5   Weight      6 non-null     Int64
6   BMI         6 non-null     Float64
7   Score       7 non-null     string
dtypes: Float64(1), Int64(3), datetime64[ns](1), string(3)
memory usage: 604.0 Bytes
```

特别说明：在 NumPy 中是有 str 和 object 区分的，其中 dtype('S')对应于 str，dtype('O')对应于 object，但是，在 Pandas 中 str 和 object 类型都对应于 dtype('O')类型，二者间其实在转换后并无实质性区别。Pandas 中的 object 类型对应的就是 Python 中的 str 字符类型。

在 Pandas 中，支持的数据类型有 float、int、bool、datetime64、timedelta[ns]、category、object 等。例如，float 其实是 Python 内置的数据类型，可以在 NumPy 及 Pandas 中直接使用，但在 NumPy 及 Pandas 中也可以使用与其对等的数据类型。

在导入数据的过程中可对数据类型进行转换，代码如下：

```
dft = pd.read_excel(
    'demo_.xlsx',
    dtype={
        'Date': 'datetime64',
        'Age': 'object',
        'WorkYears': 'object',
        'Weight': 'str',
        'Score': 'category'
    }
)
dft.info()
```

输出的结果如下：

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 7 entries, 0 to 6
Data columns (total 8 columns):
```

```
 #Column      Non-Null Count  Dtype
---  ------  --------------  -----
 0   Date        7 non-null      datetime64[ns]
 1   Name        7 non-null      object
 2   City        6 non-null      object
 3   Age         7 non-null      object
 4   WorkYears   7 non-null      object
 5   Weight      6 non-null      object
 6   BMI         6 non-null      float64
 7   Score       7 non-null      category
dtypes: category(1), datetime64[ns](1), float64(1), object(5)
memory usage: 551.0+ Bytes
```

对比 info()方法与 dtypes 属性的显示,代码如下:

```
dft.dtypes
```

输出的结果如下:

```
Date          datetime64[ns]
Name                  object
City                  object
Age                   object
WorkYears             object
Weight                object
BMI                  float64
Score               category
dtype: object
```

在上面的代码中,. info()是方法,. dtypes 是属性。在 Pandas 中,当初次面对一个 DataFrame 数据时,经常会用到. info()方法或. dtypes 属性来对数据的整体结构及各字段的数据类型进行一个直观的了解。

注意:当了解的对象是 Series 时,用的是. dtype 属性;当了解的对象是 DataFrame 时,使用的是. dtypes 属性。

上面的两例在数据导入的过程中对数据类型进行了自动识别或指定。其实,当数据导入或运行之后,仍旧可以对数据类型按需求进行转换,代码如下:

```
def f(x):
    return x * 6
df['Age'].apply(f, convert_dtype=False)
```

输出的结果如下:

```
0    456
1    192
2    330
3    522
4    558
5    468
6    390
Name: Age, dtype: object
```

2. astype()

在 Pandas 中 astype()用于对数据类型进行强制转换。当数据经过强制类型转换后，convert_dtype 将不再起作用，代码如下：

```
def f(x):
   return x * 6

df['Weight'].astype('str').apply(f, convert_dtype = True)
#df['Weight'].astype('str').apply(f, convert_dtype = False)
#df['Weight'].astype('str').apply(f)
#以上 3 行代码输出的结果是相同的
```

输出的结果如下：

```
0    56.056.056.056.056.056.0
1    85.085.085.085.085.085.0
2    72.072.072.072.072.072.0
3          nannannannannannan
4    59.059.059.059.059.059.0
5    65.065.065.065.065.065.0
6    69.069.069.069.069.069.0
Name: Weight, dtype: object
```

在 Python 中，字符串 * n 意味着重复 n 次。

现以 df['Age']为例，先将数值强制转换为文本型，然后强制转换为整型，代码如下：

```
df = pd.read_excel('demo_.xlsx')
df['Age'] = df['Age'].astype(str)  #修改某列数据类型
df['Age'] = df['Age'].astype(np.int64)
```

二次转换后的数据类型对比如图 5-2 所示。

astype()在数据类型的强制转换过程中遵循着“就高不就低”的原则。例如：int32 转 float64，会新增小数位；float64 转 int32，会将小数点截掉；string 转 float64，数值型字符串会被强制转换为浮点型。Python、NumPy、Pandas 中的数据类型对照表见表 5-1。

```
Date          object          Date          object
Name          object          Name          object
City          object          City          object
Age           object          Age            int32
WorkYears      int64          WorkYears      int64
Weight       float64          Weight       float64
BMI          float64          BMI          float64
Score         object          Score         object
dtype: object                 dtype: object
```

图 5-2　强制转换数据类型

表 5-1　数据类型对照表

Pandas 中的数据类型	NumPy 中的数据类型	Python 中的数据类型	用途
object	string_,unicode_	str	文本
int64	int_,int8_,int16,int32,int64m uint8,uint16,uint32,uint64	int	整型
float64	float_,float16,float32,float64	float	浮点型
bool	bool_	bool	True/False
datetime64	datetime64[ns]	NA	日期/时间
timedelta[ns]	NA	NA	时间差
category	NA	NA	类别型文本

关于 NumPy 中的所有数据类型及详细分类，参阅表 5-1 的 NumPy 数据类型对照表。

以下案例通过.assign()方法在 DataFrame 中新建列，然后对新建的列用.astype()来强制指定数据类型。相关代码如下：

```
#导入数据,只取 Age、WorkYears 两列中的前 3 行数据
df_ = pd.read_excel('demo_.xlsx',
                    usecols = ['Age','WorkYears'],
                    nrows = 3)

#对 Age、WorkYears 两列进行数据类型指定
(df_.assign(
            Age = df['Age'].astype(np.int16),
            WorkYears = df['WorkYears'].astype(np.int16))
    ).dtypes
```

输出的结果如下：

```
Age          int16
WorkYears    int16
dtype: object
```

下面的例子中，df['City']和 df['Name']原先就存在，但 df['nCity']原先并不存在，所以就变成了新增列 df['nCity']，强制更改 df['Name']列，而 df['City']列数据类型保持不

变，代码如下：

```
df = pd.read_excel('demo_.xlsx',
                   usecols = ['Name','City'],
                   nrows = 3)
(df.assign(
         nCity = df.City.astype('category'),
         Name = df.Name.astype('category'))
            ).info()
```

输出的结果如下：

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3 entries, 0 to 2
Data columns (total 3 columns):
 #Column   Non-Null Count   Dtype
--  ----   ------------    -----
0   Name     3 non-null      category
1   City     3 non-null      object
2   nCity    3 non-null      category
dtypes: category(2), object(1)
memory usage: 422.0+ Bytes
```

除此之后，还可以对 groupby()的对象进行强制数据类型转换，代码如下：

```
(
   pd.read_excel('demo_.xlsx')
    .groupby(['City', 'Name'])['Weight']
    .mean()
    .astype(int)
)
```

输出的结果如下：

```
City        Name
Beijing     Joe     56
Guangzhou   Jim     59
Shanghai    Kim     85
Shenzhen    Jim     72
Suzhou      Sam     69
Xiamen      Kim     65
Name: Weight, dtype: int32
```

注意：如果要转换的列存在空值，则为避免报错的可能，须先对空值进行填充 fillna(0)，再做数据类型的强制转换 astype()。

5.3.2 数据结构转换

在 Pandas 中，会经常用到 stack()与 unstack()做索引的互换。

stack()可将数据的列“旋转”为行，unstack()可将数据的行“旋转”为列，stack()与 unstack()为一组逆运算操作。如果不指定旋转的索引级别，stack()与 unstack()默认对最内层进行操作(level=－1)，这里的－1 是指倒数第一层。

stack 与 unstack 是互逆的过程，stack 与 unstack 的区别在于行与列谁放在前面的问题。stack 采用行放前列放后的方式，而 unstack 采用列放前行放后的方式。

注意：df.stack()的返回值为 Series，df.unstack()的返回值为 DataFrame。

1. stack()

语法：df.stack(level=－1，dropna=True)。

结果：Stack the prescribed level(s) from columns to index。

DataFrame 的列标签为单层索引，将其转换为行，代码如下：

```
df = pd.read_excel('demo_.xlsx')
df.stack()
```

结果呈现及图解说明如图 5-3 所示。

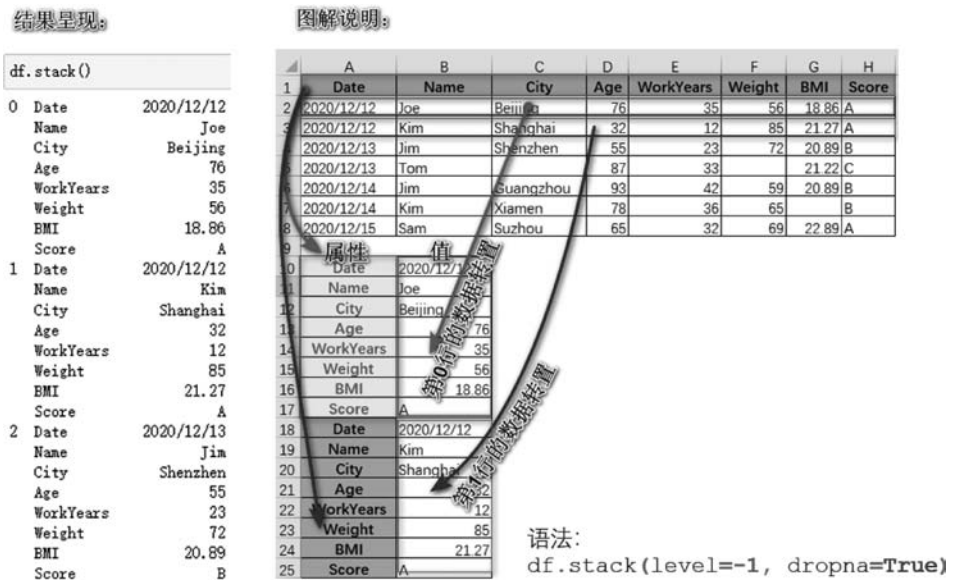

图 5-3 图解 stack()工作原理

以上输出的结果与 Excel 中的 Power Query 的以下 M 代码是等效的，代码如下：

```
let
    源 = Excel.CurrentWorkbook(){[Name="表 1"]}[Content],
    逆透视 = Table.UnpivotOtherColumns(源, {}, "属性", "值")
in
    逆透视
```

对堆叠后的数据重设索引列，使之数据结构由 Series 转变为 DataFrame，代码如下：

```
df.stack().reset_index()
```

对比堆叠数据及堆叠后重设索引的数据，如图 5-4 所示。

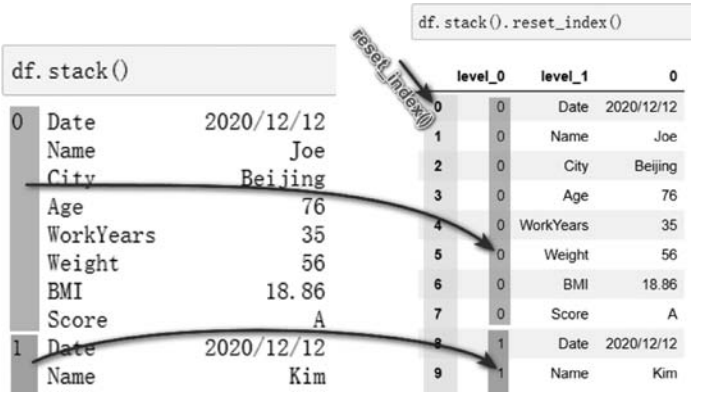

图 5-4　对比堆叠数据

图 5-4 中 DataFrame 的列名不直观、不好理解，现对 columns 重新命名，代码如下：

```
df.stack().reset_index().rename(
    columns={
        'level_0': '索引',    #将'level_0'列改名为'索引'
        'level_1': '属性',    #将'level_1'列改名为'属性'
        0: '值'})             #将 0 列改名为'值'

#以下代码与上面的代码输出的结果是一样的
#df.stack().rename_axis(['索引', '属性']).reset_index(name='值')
```

重命名列前与重命名列后的对比如图 5-5 所示。

2. unstack()

语法：Series. unstack(level＝－ 1，fill_value＝None)，返回 DataFrame 或拆堆后的 Series。

经过堆叠(stack)后再拆堆(unstack)的 DataFrame，返回的值为原 DataFrame，因为 stack 与 unstack 是一对互逆操作，代码如下：

```
df.stack().reset_index()
```

```
df.stack().reset_index().rename(
    columns={
        'level_0': '索引', #将'level_0'列改名为'索引'
        'level_1': '属性', #将'level_1'列改名为'属性'
        0: '值'})          #将0列改名为'值'
```

	level_0	level_1	0
0	0	Date	2020/12/12
1	0	Name	Joe
2	0	City	Beijing
3	0	Age	76
4	0	WorkYears	35
5	0	Weight	56
6	0	BMI	18.86
7	0	Score	A

	索引	属性	值
0	0	Date	2020/12/12
1	0	Name	Joe
2	0	City	Beijing
3	0	Age	76
4	0	WorkYears	35
5	0	Weight	56
6	0	BMI	18.86
7	0	Score	A

图 5-5　重命名列

```
df = pd.read_excel('demo_.xlsx')
df.stack().unstack()
```

输出的结果如图 5-6 所示，为原 DataFrame。

	Date	Name	City	Age	WorkYears	Weight	BMI	Score
0	2020/12/12	Joe	Beijing	76	35	56	18.86	A
1	2020/12/12	Kim	Shanghai	32	12	85	21.27	A
2	2020/12/13	Jim	Shenzhen	55	23	72	20.89	B
3	2020/12/13	Tom	NaN	87	33	NaN	21.22	C
4	2020/12/14	Jim	Guangzhou	93	42	59	20.89	B
5	2020/12/14	Kim	Xiamen	78	36	65	NaN	B
6	2020/12/15	Sam	Suzhou	65	32	69	22.89	A

图 5-6　堆叠与拆堆

unstack()操作会使窄表变成宽表。为了方便理解它的用法，只取(['Name','City','Score'])这三列数据来做观察，代码如下：

```
dfu = pd.read_excel('demo_.xlsx')[['Name', 'City', 'Score']]
dfu = dfu.set_index(['Name', 'City']) [:3]
dfu
```

以上代码等效于下面的代码，两者输出的结果完全一致，代码如下：

```
dfu = pd.read_excel('demo_.xlsx',
                    usecols = ['Name', 'City', 'Score'],
```

```
                index_col = [0,1],
                nrows = 3)
dfu
```

以上代码的详细语法将在第 7 章讲解。输出的结果如下：

```
Name    City    Score
Joe   Beijing     A
Kim   Shanghai    A
Jim   Shenzhen    B
```

重设索引，代码如下：

```
dfu.reset_index()
```

输出的结果如下：

```
   Name    City    Score
0  Joe   Beijing     A
1  Kim   Shanghai    A
2  Jim   Shenzhen    B
```

运用 unstack()方法，将行值转换为列值，代码如下：

```
dfu.reset_index().unstack()
```

前面两个步骤的图解说明如图 5-7 所示。

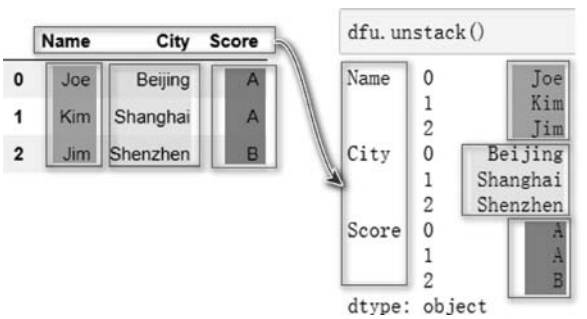

图 5-7　图解 unstack()运行原理(1)

对获取的数据进行拆堆(unstack)操作，代码如下：

```
dfui = pd.read_excel('demo_.xlsx',
                index_col = [0,1],                    #指定索引列
```

```
                usecols = ['Name', 'City', 'Score'],   #指定需导入的列
                nrows = 3).unstack('City')             #指定数据范围.拆堆操作
dfui
```

运行过程的原理及结果如图 5-8 所示。

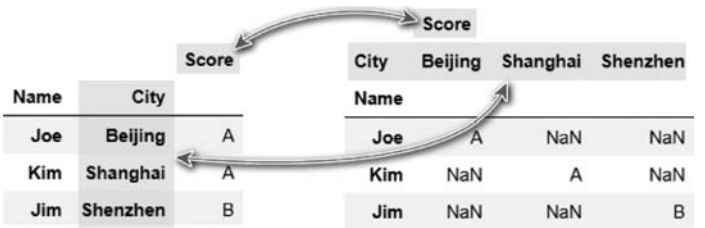

图 5-8　图解 unstack()运行原理(2)

3. melt()

语法：dfl.melt(id_vars,value_vars,var_name,value_name='value',col_level,ignore_index=True,)

结果：返回值为 DataFrame。

作用：逆透视 DataFrame,将宽表变成窄表。

参数：melt()的参数说明如表 5-2 所示。

表 5-2　melt()的参数说明

参　数	对　象	参数说明
id_vars	tuple、list or ndarray。作为标识符变量使用的列	optional
value_vars	tuple、list or ndarray。如果未指定,则使用所有列	
var_name	scalar(标量)。用于“变量”列的名称。如果没有,则使用' frame.columns.name '或“变量”	
value_name	scalar。默认值用于“值”列的名称	default 'value'
col_level	int or str。如果列是一个多层索引,那么使用指定层级去 melt	optional
ignore_index	bool。如果值为 True,则忽略原始索引而重新索引。如果值为 False,则保留原索引	default True

在上述参数中,id 是 identifier(标识符)的简写；var 是 variable(变量)的简写；col 是 column(column)的简写；level 是 multi_index 中的 level；index 是指 DataFrame 的 index。

DataFrame 的 melt()操作,代码如下：

```
dfl = pd.read_excel('demo_.xlsx').iloc[:2,:4]
dfl
dfl.melt()
```

前面两个步骤的图解说明如图 5-9 所示。

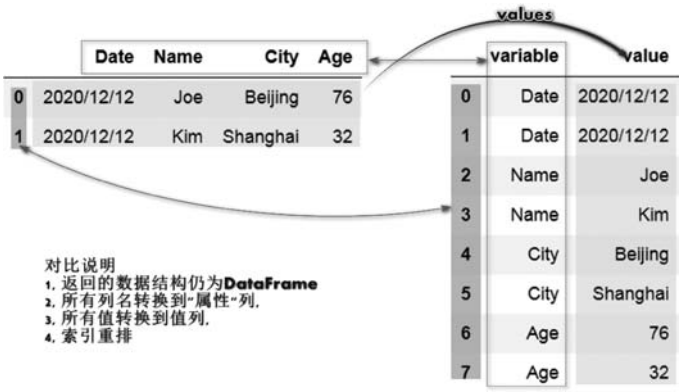

图 5-9　图解 melt()运行原理

DataFrame 的 melt()与 stack()操作比较。逐行运行以下代码：

```
dfl = pd.read_excel('demo_.xlsx').iloc[:2,:4]
dfl.melt()
dfl.stack()
```

图解说明 df.melt()方法与 df.stack()的差异性，如图 5-10 所示。

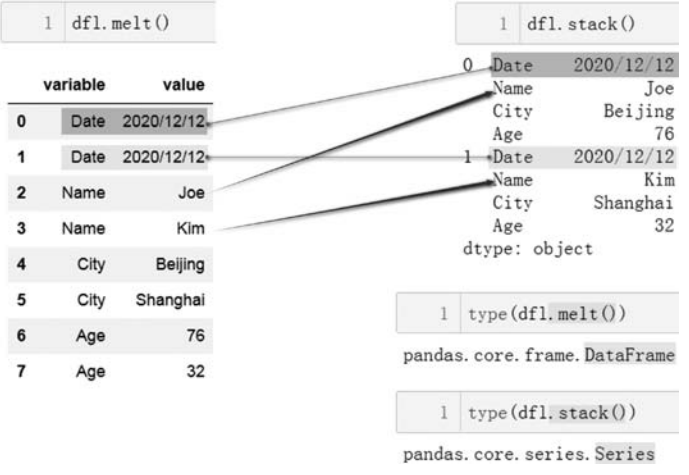

图 5-10　比较 melt()方法与 stack()方法

以下两种运行方式都是允许的，代码如下：

```
dfl = pd.read_excel('demo_.xlsx').iloc[:2,:4]
dfl.melt(id_vars = 'City')          #方式一
pd.melt(dfl,'City')                 #方式二
```

以上两种方式输出的结果是一样的，输出的结果如下：

```
     City    variable                  value
0   Beijing     Date    2020-12-12  00:00:00
1  Shanghai     Date    2020-12-12  00:00:00
2   Beijing     Name                    Joe
3  Shanghai     Name                    Kim
4   Beijing      Age                     76
5  Shanghai      Age                     32
```

继续运行的代码如下：

```
pd.melt(df1,id_vars=['Date'],value_vars=['City','Name'])
#'Date'作为分组依据,['City','Name']为属性列
```

输出的结果如下：

```
        Date      variable    value
0  2020-12-12     City     Beijing
1  2020-12-12     City    Shanghai
2  2020-12-12     Name        Joe
3  2020-12-12     Name        Kim
```

运行的代码如下：

```
pd.melt(df1,value_vars=['City','Name'])
#无任何分组依据,['City','Name']为属性列
```

输出的结果如下：

```
   variable    value
0     City     Beijing
1     City    Shanghai
2     Name        Joe
3     Name        Kim
```

运行的代码如下：

```
#未使用 var_name、value_name 参数
df1.melt(id_vars=['City'],
   value_vars=['Date', 'Name', 'Age'])
```

继续运行的代码如下：

```
#使用了 var_name、value_name 参数
dfl.melt(id_vars = ['City'],
                    value_vars = ['Date', 'Name', 'Age'],
                    var_name = '属性',
                    value_name = '值')
```

图解 var_name、value_name 两参数的使用，对比结果如图 5-11 所示。

	City	variable	value
0	Beijing	Date	2020/12/12
1	Shanghai	Date	2020/12/12
2	Beijing	Name	Joe
3	Shanghai	Name	Kim
4	Beijing	Age	76
5	Shanghai	Age	32

	City	属性	值
0	Beijing	Date	2020/12/12
1	Shanghai	Date	2020/12/12
2	Beijing	Name	Joe
3	Shanghai	Name	Kim
4	Beijing	Age	76
5	Shanghai	Age	32

图 5-11　图解 var_name、value_name 两参数的使用

运行的代码如下：

```
pd.melt(dfl,'City').pivot('City','variable','value').reset_index()
```

输出的结果如下：

```
variable     City     Age       Date       Name
    0       Beijing   76    2020-12-12    Joe
    1      Shanghai   32    2020-12-12    Kim
```

4. pivot()

语法：df.pivot(index,columns,values)。

结果：返回值 DataFrame。

对 DataFrame 的 pivot()应用，代码如下：

```
dfl = pd.read_excel('demo_.xlsx').iloc[:2,:4]
dfl.pivot("Date","Name","Age")                                  #方法一
#dfl.pivot(index = "Date",columns = "Name",values = "Age")      #方法二
#dfl.pivot("Date","Name")["Age"]                                #方法三
```

方法一、方法二、方法三输出的结果完全一样。对比 Excel 的透视表，理解 Pandas pivot()的运行原理，如图 5-12 所示。

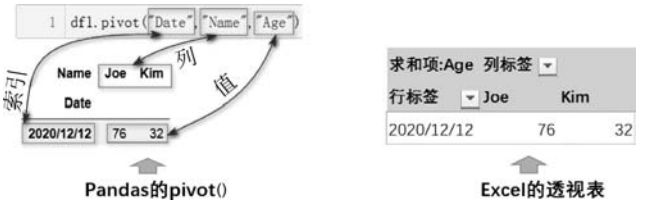

图 5-12　pivot()的运行原理

代码如下：

```
dfl = pd.read_excel('demo_.xlsx').iloc[:2,:4]
dfl.pivot(index = "Date",columns = ['City',"Name"],values = "Age")
```

输出的结果如下：

```
    City       Beijing  Shanghai
    Name         Joe       Kim
    Date
2020-12-12     76        32
```

运行的代码如下：

```
dfl.pivot(index = "Date",columns = 'City',values = ["Age","Name"])
```

输出的结果如下：

```
                Age                Name
    City       Beijing  Shanghai  Beijing  Shanghai
    Date
2020-12-12     76        32        Joe       Kim
```

5. transpose()

在 Pandas 中，df.transpose()方法与 df.T 属性的效果是完全一致的。以所选择的数据的最左上角为基点，进行行列转换位置，代码如下：

```
dfl = pd.read_excel('demo_.xlsx').iloc[:2,:4]
dfl
dfl.transpose()
```

此做法相当于 Excel 中的“复制”→“粘贴”→（粘贴选项）转置。输出的结果如图 5-13 所示。

相关原理说明，如图 5-14 所示。

图 5-13　数据的转置

图 5-14　转置的运行原理

5.3.3　文本格式转换

1. %运算符

语法：(格式模板)%(值组)。

结果：返回的值为字符串。

参数：格式模板，一组或多组以%标志的字符串，如果有两个或两个以上的值，则需要用小括号括起来。值组，即要格式化的字符串或元组。

逐行运行代码：

```
"%s" % "18.86", type("%s" % "18.86")
#'%s', 字符串格式
"%d" % 18.86, type("%d" % 18.86)
#'%d', 整数格式
"%f" % 18.86, type("%f" % 18.86)
#'%d', 浮点格式(默认为小数点后 6 位)
```

格式模板中最常用的是 s、d、f 字符串。输出的结果如下：

```
('18.86', str)
('18', str)
('18.860000', str)
```

当字符串格式化处理的场景较为复杂时，可以采用字典格式（对字典中的键 key 的顺序没有要求，只需键值对应），代码如下：

```
'%(Name)s,%(City)s,%(Age).2f' % {'City':'Beijing', 'Name':'Joe','Age':76}
'%(Name)s 来自一线城市 %(City)s,今年%(Age)d岁 体能%(BMI).2f' % \
{"Name": "Joe","City": "Beijing","Age": 76,"BMI": 18.86}
```

输出的结果如下：

```
'Joe,Beijing,76.00'
'Joe 来自一线城市 Beijing,今年 76 岁 体能 18.86'
```

字符的宽度设置，逐行运行代码如下：

```
'%+6s' % 'Joe'
'%-6s' % 'Joe'
'%+6d' % 18.86
'%-6d' % 18.86
'%06d' % 18.86
'%(Age)+6s, %(BMI)06d' % {"Age":76,"BMI":18.86}
```

可供选择的参数：＋、－、''（空格）、0。其中，＋表示右对齐；－表示左对齐；''表示在正数的左侧填充空格；0 表示填充 0。＋6 表示右对齐，字符宽度为 6（当宽度不足时，从左侧添加空格占位）。输出的结果如下：

```
'  Joe'
'Joe  '
'  +18'
'18   '
'000018'
'   76,000018'
```

字符的精度设置，逐行运行的代码如下：

```
'%f' % 76            #浮点数的默认精度为 6
'%.2f' % 76          #小数点后保留 2 位
'%.4f' % 18.86       #小数点后保留 4 位
'%.4s' % 'Beijing'   #取字符串的前 4 个
```

输出的结果如下：

```
'76.000000'
'76.0000'
```

```
'18.8600'
'Beij'
```

2. format()函数

语法：format(value,format_spec)。

参数：value表示要切换的数据；format_spec为格式控制标志，包括fill、align、sign、#和0、width、千位符、precision、type这8个可选字段，这些字段是可以组合使用的。

逐行运行的代码如下：

```
format(18.86)      #等价于str(18.86)
# format默认将其他数据类型转换为字符型
format(76,'d')
format(18.86,'f')
format(76,'>05')
format(76,'0>5')
format(76/18.86,'%')
format(76/18.86,'.2%')
```

输出的结果如下：

```
'18.86'
'76'
'18.860000'
'00076'
'00076'
'402.969247%'
'402.97%'
```

3. str.format()方法

语法：{参数序号:格式控制标志}.format(位置参数,关键字参数)。

参数序号：位置参数或关键字参数传递过来的参数变量，可以为空值。

格式控制标志：用来控制参数显示时的格式，和format()函数的format_spec参数是一样的。

逐行运行的代码如下：

```
'{}现居住于{},年龄{},体重{}BMI{}'.format('Joe','Beijing',76,56,18.86)

'{0},体重{3}BMI{4}现居住于{1},年龄{2}'.format('Joe','Beijing',76,56,18.86)

'{0},体重{3}BMI{4:.4f},现居住于{1},年龄{2}'.format('Joe','Beijing',76,56,18.86)

'"2012/12/12"登记结果:{0},体重{3}BMI{4}现居住于{1},年龄{2}'.format('Joe','Beijing',76,56,
18.86)
```

输出的结果如下：

```
'Joe 现居住于 Beijing,年龄 76,体重 56BMI18.86'
'Joe,体重 56BMI18.86 现居住于 Beijing,年龄 76'
'Joe,体重 56BMI18.8600,现居住于 Beijing,年龄 76'
'"2012/12/12"登记结果:Joe,体重 56BMI18.86 现居住于 Beijing,年龄 76'
```

5.3.4 style 样式转换

语法：类别（type)为属性(property)，使用时后面不用加括号()。

结果：返回一个样式对象(Styler object)。

作用：对特定数据的突出显示、数值的格式化、迷你条形图的使用等，提高数据的“颜值”。可以通过 styler.applymap、styler.apply 等将样式功能传递到数据中，也可以通过 Styler.background_gradient 实现数据的热力图功能等(例如：对 seaborn 中 light_palette 的调用)。

注意：df.style 输出的是一个 Styler 对象(不是 DataFrame)。

对 DataFrame 的索引进行隐藏，代码如下：

```
df = pd.read_excel(r"demo_.xlsx")
df
df.style.hide_index()  #隐藏索引
```

索引列被隐藏，输出的结果如图 5-15 所示。

Date	Name	City	Age	WorkYears	Weight	BMI	Score
2020/12/12	Joe	Beijing	76	35	56.000000	18.860000	A
2020/12/12	Kim	Shanghai	32	12	85.000000	21.270000	A
2020/12/13	Jim	Shenzhen	55	23	72.000000	20.890000	B
2020/12/13	Tom	nan	87	33	nan	21.220000	C
2020/12/14	Jim	Guangzhou	93	42	59.000000	20.890000	B
2020/12/14	Kim	Xiamen	78	36	65.000000	nan	B
2020/12/15	Sam	Suzhou	65	32	69.000000	22.890000	A

图 5-15　隐藏索引列

对 DataFrame 中指定的列进行隐藏，代码如下：

```
df.style.hide_columns(['City','Date'])  #隐藏列
```

指定的列被隐藏，输出的结果如图 5-16 所示。

对 DataFrame 中的 null 值用黄色填充，代码如下：

```
dft = pd.read_excel(r"demo_.xlsx").select_dtypes('number')
dft
dft.style.highlight_null('yellow')
```

结果如图 5-17 所示。

	Name	Age	WorkYears	Weight	BMI	Score
0	Joe	76	35	56.000000	18.860000	A
1	Kim	32	12	85.000000	21.270000	A
2	Jim	55	23	72.000000	20.890000	B
3	Tom	87	33	nan	21.220000	C
4	Jim	93	42	59.000000	20.890000	B
5	Kim	78	36	65.000000	nan	B
6	Sam	65	32	69.000000	22.890000	A

图 5-16　隐藏列

	Age	WorkYears	Weight	BMI
0	76	35	56.000000	18.860000
1	32	12	85.000000	21.270000
2	55	23	72.000000	20.890000
3	87	33	nan	21.220000
4	93	42	59.000000	20.890000
5	78	36	65.000000	nan
6	65	32	69.000000	22.890000

图 5-17　高亮显示 nan 值

如果想对手头的数据负数标红，正数及 0 用黑色标记（由于手头的数据没有负数，所以先将一部分数据处理变成负数），则代码如下：

```
dft = dft.fillna(0).applymap(lambda x: x if x > 50 else x - 60)
def AA(val):
    color = 'red' if val < 0 else 'black'
    return 'color: %s' % color
dft.style.applymap(AA)
```

输出的结果如图 5-18 所示。

	Age	WorkYears	Weight	BMI
0	76	-25	56.000000	-41.140000
1	-28	-48	85.000000	-38.730000
2	55	-37	72.000000	-39.110000
3	87	-27	-60.000000	-38.780000
4	93	-18	59.000000	-39.110000
5	78	-24	65.000000	-60.000000
6	65	-28	69.000000	-37.110000

图 5-18　标示所有负值

设置自定义函数，对 DataFrame 进行样式设置，代码如下：

```
def BB(s):
    mx = s == s.max()
```

```
    return ['background-color: yellow' if v else '' for v in mx]dft.style.apply(BB)
# 突出显示每列中的最大值
```

输出的结果如图 5-19 所示。

采用链式写法,将以上两个代码自定义函数放在一行语句中,代码如下:

```
dft.style.applymap(AA).apply(BB)    #链式写法
```

输出的结果如图 5-20 所示。

	Age	WorkYears	Weight	BMI
0	76	-25	56.000000	-41.140000
1	-28	-48	85.000000	-38.730000
2	55	-37	72.000000	-39.110000
3	87	-27	-60.000000	-38.780000
4	93	-18	59.000000	-39.110000
5	78	-24	65.000000	-60.000000
6	65	-28	69.000000	-37.110000

图 5-19　高亮显示每列的最大值

	Age	WorkYears	Weight	BMI
0	76	-25	56.000000	-41.140000
1	-28	-48	85.000000	-38.730000
2	55	-37	72.000000	-39.110000
3	87	-27	-60.000000	-38.780000
4	93	-18	59.000000	-39.110000
5	78	-24	65.000000	-60.000000
6	65	-28	69.000000	-37.110000

图 5-20　负值标识及高亮显示

对数据进行百分比格式设置,代码如下:

```
dft = pd.read_excel(r"demo_.xlsx").select_dtypes('number')
dft
dft.style.format("{:.2%}")
dft.style.format("{:.2%}", na_rep="-")
```

运行以上代码,输出的结果对比如图 5-21 所示。

	Age	WorkYears	Weight	BMI
0	7600.00%	3500.00%	5600.00%	1886.00%
1	3200.00%	1200.00%	8500.00%	2127.00%
2	5500.00%	2300.00%	7200.00%	2089.00%
3	8700.00%	3300.00%	nan%	2122.00%
4	9300.00%	4200.00%	5900.00%	2089.00%
5	7800.00%	3600.00%	6500.00%	nan%
6	6500.00%	3200.00%	6900.00%	2289.00%

	Age	WorkYears	Weight	BMI
0	7600.00%	3500.00%	5600.00%	1886.00%
1	3200.00%	1200.00%	8500.00%	2127.00%
2	5500.00%	2300.00%	7200.00%	2089.00%
3	8700.00%	3300.00%	-	2122.00%
4	9300.00%	4200.00%	5900.00%	2089.00%
5	7800.00%	3600.00%	6500.00%	-
6	6500.00%	3200.00%	6900.00%	2289.00%

图 5-21　文本格式设置

采用字典方式,对不同的列采用不同的样式显示,代码如下:

```
dft = pd.read_excel(r"demo_.xlsx").select_dtypes('number')

(
    dft.style.format(
        {'Age': "{:0.0f}",
         'WorkYears': "{:0.3f}",
         'Weight': "$ {:0.2f}",
         'BMI': '{:+.2f}'})
)
```

输出的结果如图 5-22 所示。

	Age	WorkYears	Weight	BMI
0	76	35.000	$56.00	+18.86
1	32	12.000	$85.00	+21.27
2	55	23.000	$72.00	+20.89
3	87	33.000	$nan	+21.22
4	93	42.000	$59.00	+20.89
5	78	36.000	$65.00	+nan
6	65	32.000	$69.00	+22.89

图 5-22　对多列数据的不同样式设置

逐行运行以下代码，给指定列的数据添加数据条，代码如下：

```
dft.style.bar(subset = ['Age', 'BMI'], color = 'lightblue')
dft.style.bar(subset = ['Age', 'BMI'], color = 'lightblue').set_precision(2)
```

.set_precision(2)语句用于将数据的精度设置为 2。输出的结果对比如图 5-23 所示。

	Age	WorkYears	Weight	BMI
0	76	35	56.000000	18.860000
1	32	12	85.000000	21.270000
2	55	23	72.000000	20.890000
3	87	33	nan	21.220000
4	93	42	59.000000	20.890000
5	78	36	65.000000	nan
6	65	32	69.000000	22.890000

	Age	WorkYears	Weight	BMI
0	76	35	56.00	18.86
1	32	12	85.00	21.27
2	55	23	72.00	20.89
3	87	33	nan	21.22
4	93	42	59.00	20.89
5	78	36	65.00	nan
6	65	32	69.00	22.89

图 5-23　添加数据条

5.4　本章回顾

计算机中有顺序结构、分支结构和循环结构 3 种基本的循环结构，用于处理重复的、有规律的操作。在 Pandas 中使用最多的是 for 循环语句。

在数据清洗的 ETL(清洗、转换、加载)过程中,本章属于 T(转换)环节。在数据的清洗与分析的过程中,经常会用到数据类型、数据结构或数据样式的转换。若转换的过程相对复杂,则可能会在转换语句中用到循环结构。

Pandas 的强项在于数据的处理,但它也带有一些简单的数据表格美颜功能。在数据处理的过程中,可通过 style 属性进行一些较为常见的样式设置(例如:空值颜色填充、负值颜色标红、数据条等)。