第 3 章

Pandas 入 门

NumPy 作为 Python 科学计算的基础软件包，它的功能非常强大，但它不支持异构列表数据。什么是异构？简单地说就是指一个整体中包含不同的成分的特性，即这个整体由多个不同的成分构成。

在实际工作及生活中，使用的数据大多数是二维异构列表数据。二维异构列表数据是指在一个二维数据结构中允许不同的列拥有不同的数据类型。Pandas 正是为处理此类数据而生的，它能够灵活、高效地处理和 SQL、Excel 电子表格类似的二维异构列表数据，使 Pandas 迅速成为 Python 的核心数据分析支持库。

以下是 Pandas 的特色及 NumPy 与 Pandas 的区别说明。

1. Pandas 的特色

Pandas 是 Python 生态环境下非常重要的数据分析包。它是一个开源的、有开源协议的库。正因为有了它的存在，基于 Python 的数据分析才大放异彩，为世人瞩目。

Pandas 吸纳了 NumPy 中的很多精华，同时又能支持图表和混杂数据运算。这是强于 NumPy 的地方。因为 NumPy 数组中只能支持单一的某种数据类型（例如：整型或浮点型）。Pandas 是基于 NumPy 构建的数据分析包，但包含了比 ndarray 更高级的数据结构和操作工具。正是因为 series 与 DataFrame 的存在，才使 Pandas 在进行数据分析时，十分便捷与高效。

2. NumPy 和 Pandas 的区别

NumPy 生成的是 ndarray 数组，而 Pandas 则可基于 NumPy 生成两种对象（Series，DataFrame）。Series 是一维数组，它能保存不同种类的数据类型（字符串、boolean 值、数字等），而 NumPy 只能存储同类型数据。DataFrame 是二维的表格型数据结构，DataFrame 的每一列都是一个 Series。

3.1 Series

Series 的中文含义是“次序、顺序、连续”，在后面的所有章节会直接采用英文 Series。DataFrame（结构）与 Series（结构）都有一个 Index，Index 也是 Pandas 的数据结构之一。

注意：在 Pandas 中，DataFrame、Series、Index 这 3 种数据结构的首字母均须大写。

3.1.1 Series 基础知识

1. Series 结构

Series 是由一组同类型的数据和一组与数据对应的 Index(标签)所组成的。也就是说：Series(结构)＝Index(结构)＋ data(类似于一组数组的数据结构)，它是 Pandas 的核心结构之一，也是 DataFrame 结构的基础。Series 的语法说明如下：

```
pd.Series(
    data = None,       # 传入数组,可迭代对象、字典或标量值
    index = None,      # 以数组或列表形式传入自定义索引,若不传值,则按系统默认值
    dtype = None,      # 指定数据类型.如果为 None,则默认为 object
    name = None,       # 自定义 series 的名字,默认为 none
    copy = False,      # 一个布尔值.如果值为 True,则复制输入的数据 data
    fastpath = False,
)
```

作用：通过对各类数据结构转换后，生成 Pandas 的 Series 数据结构。

注意：创建 Series 是用 pd.Series()实现的，而不是用 df.Series()。

pd.Series()语法的图解说明如图 3-1 所示。

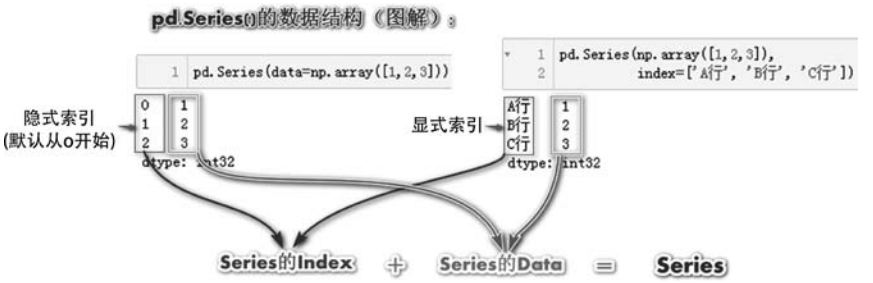

图 3-1 图解说明 pd.Series()

注意：当指定索引时，索引的长度一定要与 Series 中的 data 长度相等，否则会报错。

当输入的是 pd.Series(np.array([1,2,3,]),index=['A 行','B 行','C 行','D 行'])时，会报错，如图 3-2 所示。

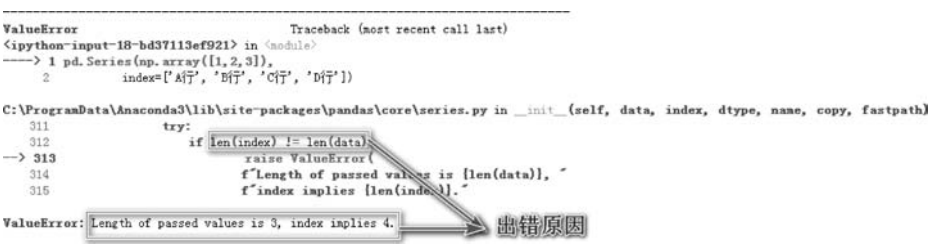

图 3-2 报错提示

以下是 pd.Series()的代码演示：

```
>>> import pandas as pd
>>> #1. Series 的索引默认从 0 开始,步长为 1 的整型序列
>>> print(pd.Series(data = np.array([1,2,3])))
0    1
1    2
2    3
dtype: int32

>>> print(pd.Series(np.array([1,2,3])))
0    1
1    2
2    3
dtype: int32

>>> print(pd.Series(np.array([1,2,3]), index = ['A 行', 'B 行', 'C 行']))
A 行    1
B 行    2
C 行    3
dtype: int32

>>> #2. 当指定索引时,Series 会按指定的索引进行排列
>>> print(pd.Series(np.array([1,2,3]),
...           index = ['A 行', 'B 行', 'C 行'],
...           dtype = float))
A 行    1.0
B 行    2.0
C 行    3.0
dtype: float64

>>> #3. 为 Series 命名是允许的,默认为 none
>>> print( pd.Series(np.array([1,2,3]),
...           index = ['A 行', 'B 行', 'C 行'],
...           dtype = float,
...           name = '第 1 个 Series'))
A 行    1.0
B 行    2.0
C 行    3.0
Name: 第 1 个 Series, dtype: float64

>>> #4. 将 copy 参数设置为 True, 注意 True 的首字母为大写
>>> print( pd.Series(np.array([1,2,3]),
...           index = ['A 行', 'B 行', 'C 行'],
...           dtype = float,
...           name = '第 1 个 Series',
```

```
...            copy = True))
A行    1.0
B行    2.0
C行    3.0
Name: 第 1 个 Series, dtype: float64
```

注意：在 Python 中，True 与 False 的首字母必须大写，否则会报错。

2. Series 与 NumPy 的初步比对

NumPy 是 Pandas 的基础及重要依赖项，所以在 Pandas 中随时可以找到 NumPy 的“身影”。首先导入 NumPy 及 Pandas 的第三方库，代码如下：

```
import numpy as np
import pandas as pd
```

Pandas 与 NumPy 的一些属性对比，如图 3-3 所示。

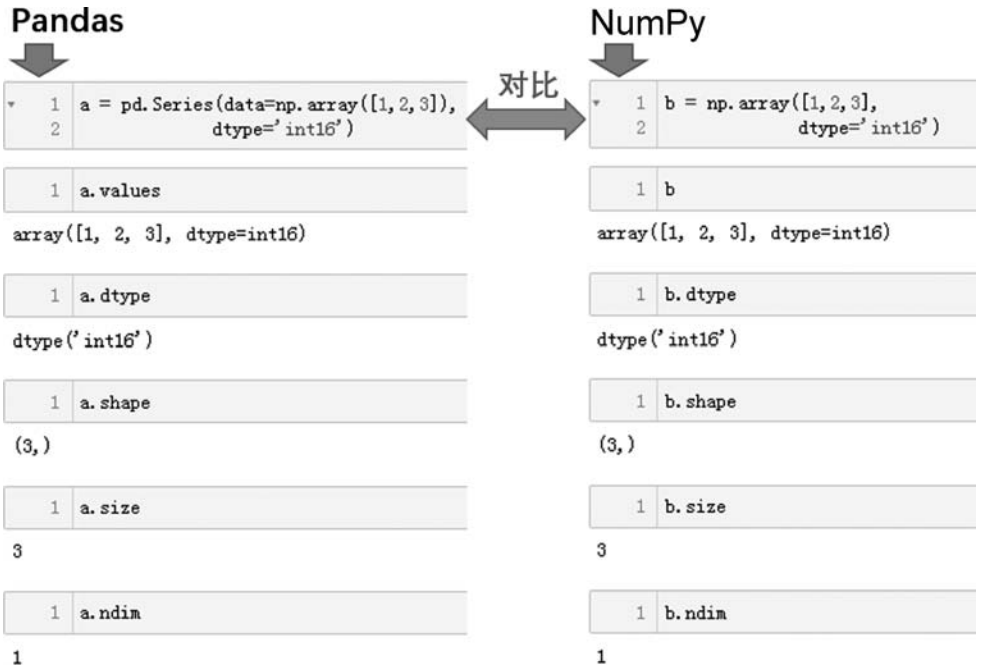

图 3-3　Pandas 与 NumPy 的属性对比

在图 3-3 中，Series.values()与 NumPy 的 array 存在关联性。np.array()相关的属性，例如：dtype、shape、size、ndim、nBytes、T 等属性均适用于 Series 或 DataFrame。当然，Pandas 在 ndarray 的基础上，也新增了不少新的属性，例如：index、memory_usage、hasnans、empty、flags、set_flags、name 等。

3.1.2 Series的构建

Series是一个类似于一维数组的对象，可以把它理解为一组带索引的NumPy一维数组。常见的创建的方式有列表转Series、字典转Series、ndarray转Series。在创建的过程中可能同时要考虑是否须特别指定index、是否须设置数据类型等。

一维数组与Series之间最大的区别在于：Series附带索引列(Index)。Series的索引所附带的信息作为一个有效的标识而经常用到(例如：表间关系型的应用)，这是单纯的一维数组所不具备的。

1. 列表(List)转Series

以下列举了List转Series的5种场景，代码如下：

```
#场景一
>>> pd.Series(list('456'))
0    4
1    5
2    6
dtype: object

#场景二

>>> pd.Series(['a','b','c'],index=[0,1,2])
0    a
1    b
2    c
dtype: object

#场景三
>>> pd.Series(index=list('0a2b'), data=np.arange(4))
0    0
a    1
2    2
b    3
dtype: int32

#场景四
>>> pd.Series([1,2,3])
0    1
1    2
2    3
dtype: int64

#场景五
>>> pd.Series([1, 2, 3, 8], index=[5,2,3,1])
```

```
5    1
2    2
3    3
1    8
dtype: int64
```

下面的这几种情形需特别注意，代码如下：

```
#list * n
>>> pd.Series([2] * 3)
0    2
1    2
2    2
dtype: int64

#list * n
>>> pd.Series([1,2,3] * 3)
0    1
1    2
2    3
3    1
4    2
5    3
6    1
7    2
8    3
dtype: int64

#Series * n
>>> pd.Series([1,2,3,]) * 3
0    3
1    6
2    9
dtype: int64
```

在 pd.Series([2] * 3)与 pd.Series([1,2,3] * 3)中，它们的 * 3 是以列表[2]和[1,2,3]为对象的。在 Python 中，使用数字 n 乘以一个序列会生成新的序列。新序列的内容为原来序列被重复 n 次的结果。例：[2] * 3 的结果为[2,2,2]；[1,2,3]结果为[1,2,3,1,2,3,1,2,3]。

pd.Series([1,2,3,]) * 3 中，它的 * 3 是对 Series 进行操作，而 pd.Series().values 其实是一个 NumPy 数组，而数组是具备广播功能的，所以值为 np.array([3,6,9])。

2. 元组(Tuple)转 Series

元组转 Series，代码如下：

```
>>> pd.Series((1,2,3))
0    1
1    2
2    3
dtype: int64
```

3. 字典(Dist)转 Series

在 Python 中,字典是放在花括号{}内的一系列键值对;在字典中,想存放多少个键值对取决于需求。在字典中,每个键都与一个值相关联,键与值之间用冒号分隔,而键值对之间用逗号分隔。这些对应的值可以是“数字、字符串及其他”,代码如下:

```
>>> pd.Series({
...     "Joe":'Beijing',
...     "Kim":'Shanghai',
...     "Jim":'Guangzhou'
... })
Joe    Beijing
Kim   Shanghai
Jim  Guangzhou
dtype: object
```

4. ndarray 转 Series

以下列举了 4 种应用场景,后 3 种侧重于 Index 的指定,代码如下:

```
#场景一
>>> pd.Series(np.array((1,2,3)))
0    1
1    2
2    3
dtype: int32

#场景二
>>> pd.Series(np.arange(4,9),index = list('abcde'))
a    4
b    5
c    6
d    7
e    8
dtype: int32

#场景三
>>> pd.Series(np.arange(4,9),index = np.arange(6,11))
6    4
7    5
```

```
8     6
9     7
10    8
dtype: int32

#场景四
>>> pd.Series(np.linspace(0,9,5),['A','B','C','D','E'])
A    0.00
B    2.25
C    4.50
D    6.75
E    9.00
dtype: float64
```

5. 常量转 Series

常量转 Series,代码如下:

```
>>> pd.Series(5,index = np.arange(4))
0    5
1    5
2    5
3    5
dtype: int64
```

3.1.3 Series 的常用转换方法

在 Pandas 中,以下转换方法适用于 Series 及 DataFrame。

1. astype()

astype()用于对 Series 中的数据类型进行强制转换,代码如下:

```
>>> s = pd.Series([1, 2, 3],dtype = 'int32')
>>> s.dtype
dtype('int32')

>>> s.astype('uint32').dtype
dtype('uint32')
```

2. convert_dtypes()

convert_dtypes()用于对 DataFrame 或 Series 中的数据类型进行强制转换,代码如下:

```
>>> s = pd.Series([1, 2, 3], dtype = np.dtype("float32"))
>>> s.convert_dtypes()
0    1
1    2
```

```
2    3
dtype: Int64
```

通过 convert_dtypes()，自动转换到最可能的数据类型。这个功能在 DataFrame 的数据导入过程中会经常用到。需要说明的是：在 NumPy 中，当未做数据的子类型指定时，整型数据多数情况下为 32 位；经 convert_dtypes()转换后，数据类型默认为 64 位。在 NumPy 和 Pandas 中，浮点型的数据类型默认为 float64。

3. to_x()

在 Pandas 中，to_x()系列有 pd. to_x()和 df. to_()两大类。常用的有 df. to_numpy()、df. to_list()、pd. to_period()、pd. to_timestamp()、pd. to_numeric()等，代码如下：

```
>>> pd.Series(pd.Categorical(['a', 'b', 'a', 'c'])).to_numpy()
array(['a', 'b', 'a', 'c'], dtype = object)

>>> pd.Series(np.array([1, 2, 3], dtype = 'int')).to_numpy(dtype = object)
array([1, 2, 3], dtype = object)

>>> pd.Series(np.array([1, 2, 3])).to_list()
[1, 2, 3]
```

Series 与 DataFrame 的 to_x()系列中还有很多，例如：to_excel()、to_csv()、to_sql()、to_json()、to_dist()等，这里面的很多方法在接触 DataFrame 时会经常使用到。

在 Pandas 中，read 与 to 是一对黄金搭档。读取文件的方法以 pd. read_x()为主，而写入的方法以 df. to_x()为主，如表 3-1 所示。

表 3-1 Pandas 中常用的读取与写入方法

数据类型	描述符	读取方法	写入方法
文本	CSV	pd. read_csv()	df. to_csv()
文本	JSON	pd. read_json()	df. to_json()
文本	HTML	pd. read_csv()	df. to_csv()
文本	剪切板	pd. read_clipboard()	df. to_clipboard()
二进制	Excel	pd. read_excel()	df. to_excel()
二进制	HDF5	pd. read_hdf()	df. to_hdf()
二进制	PKL	pd. read_pickle()	df. to_pickle()
SQL	SQL	pd. read_sql()	df. to_sql()

3.1.4 Series 的“十八招”

在中国的传统武术中有“十八般武艺”之说，现在一般多用来比喻各种技能。Pandas 也有它的“十八般武艺”，例如：筛选、删除、保留、填充、修改、转换、排序、计算等。

本节所罗列的十八招(含文本处理的“一招九式”)为 Pandas 的“十八般武艺”。以下均以 Series 对象做演示,其目的是让读者先对 Pandas 的常见功能有个大体、直观的了解,达到“纲举目张”的效果。以下十八招来源于笔者日常数据分析的使用流程及经验的累积,同时又兼顾了理解的容易性及招数的实战性。以下是十八招的着力点。

(1) 在行与列的优先序方面,优先处理列。如果知道导出的数据哪一个区间段的列是所需的,完全可以采取切片的方式;如果列名是有规律的,则可以采用 filter 或其他函数的方式,选取所需的列;如果整列为空的,则可以采取 drop 方法。

(2) 对行进行筛选。行筛选的方式主要有指定前/后所保留的行数、条件筛选(例如:any/all、isin/between 等)、比较条件筛选(例如:>、<、=等)等。

(3) 行列处理后的值可能需要进行二次清洗工作。例如:删除(空值、重复值、异常值等);如果某些空值是不能删除的,就需要按指定规则对其进行填充。

(4) 转换环节。首先,对文本进行转换,抽取复合型文本中的数字,对特定文本进行拆分与合并,然后,要对数据进行转换,进入对应的四则运算及格式转换。

(5) 对规整的数据进行各式各样的处理(例如:排序、分组、聚合),也可以在此基础上进行挖掘与图形化呈现。

第 1 招:筛选

filter()是一个 Python 的内置函数,用于过滤掉不符合条件的元素,代码如下:

```
>>> s = pd.Series([1,2,3],index=['A','B1','B2'])
>>> s.filter(like='B')
B1    2
B2    3
dtype: int64
```

filter()是一个很好用且很有用的数据筛选方法。它里面有 items、like、regex 这 3 种互为冲突的选择模式(也就是说:三者只能选一),用于处理不同的选择模式。其中,items 适用于 list-like,like 适用于 str,regex 适用于正则。

(1) item 参数的用法。在 DataFrame 中,items 中的 list-like 用于按名字筛选列。

s. filter(items=['B1','B2'])与 s. filter(like='B')等效。

例:df. filter(items=['Name','City']),用于筛选 df 中的 Name、City 两列。

(2) like 参数的用法。在 filter 中,like 参数可以与 axis 参数一起使用。

例:df. set_index('Name'). filter(like='im',axis=0),筛选出索引中含 im 的所有名字。

(3) regex 参数的用法。regex 参数可以与 axis 参数一起使用,regex 代表的是正则。

df. set_index('Name'). filter(regex='im $ ',axis=0),筛选出索引中含 im 的所有名字。

df. set_index('Name'). filter(regex='e $ ',axis=1),筛选出列名中以 e 结尾的所有

字段。

除了 filter()方法，Pandas 中还有 select_dtypes()方法，此方法用于列筛选，依据指定的数据类型选择对应的列。需要注意的是：在 Series 中查看数据类型用 dtype，但在 DataFrame 中查看数据类型则用 dtypes，代码如下：

```
>>> pd.Series([1,2,3],index=['A','B1','B2']).dtype
dtype('int64')
```

注意：Series 中没有类似 select_dtype()之类的方法。

第 2 招：保留

1. head()

head( )函数只能读取前 5 行数据，也可另行指定前几行。head(2)为前 2 行，代码如下：

```
>>> s = pd.Series([1,2,3],index=['A','B1','B2'])
>>> s.head(2)
A     1
B1    2
dtype: int64
```

2. tail()

tail()方法默认显示数据集的最后 5 行，也可另行指定后几行，代码如下：

```
>>> s.tail(2)
B1    2
B2    3
dtype: int64
```

尽管本书一直在用小数据做演示，但在实际数据处理分析时，肯定会碰到大数据。如果只想查看前几行，则可以用.head()方法；如果想看最后几行，则可以用.tail()方法。

3. 索引与切片

以下是常见的 3 种下标索引方式，代码如下：

```
>>> s = pd.Series(np.random.normal(size=5))
>>> s[0]
0.608344392668116

>>> s.loc[0]
0.608344392668116

>>> s.iloc[0]
0.608344392668116
```

布尔索引也是常用的一种索引方式，代码如下：

```
>>> s = pd.Series(np.arange(3))
>>> s[s > 1]
2    2
dtype: int32
```

以下是切片的应用举例，代码如下：

```
#索引与切片
>>> s = pd.Series(np.random.normal(size = 5))
>>> s[:3]
0  -0.304613
1   1.067088
2  -1.260125
dtype: float64
```

第 3 招：判断

在计算机语言中，判断的返回值只有两种：是(True)或否(False)。在计算机语言中，以 is 开头的函数多为信息函数，其返回值为 True 或 False。

1. isnull()

df.isnull()用于检查数据是否有缺失；df.notnull()用于判断是否未缺失；df.isnull()与 df.notnull()互为逻辑的对立面，代码如下：

```
>>> s = pd.Series(['R','G','B'],index = [1,3,np.nan])
>>> s.isna()
1.0    False
3.0    False
NaN    False
dtype: bool

>>> s.isnull()
1.0    False
3.0    False
NaN    False
dtype: bool

>>> s.notna()     #等价于 s.notnull()
1.0    True
3.0    True
NaN    True
dtype: bool
```

```
>>> s.notna().sum()
3
```

通过缺失值检测(isna 或 notna)后生成布尔值,返回值为 True 或 False。

2. isin()

isin()用于查看 Series 中是否包含某个字符串,返回布尔值,代码如下:

```
>>> s = pd.Series(['R',np.nan,'G'],index = np.arange(3))
>>> s.isin(['R', 'G', 'B', 'nan',36])
0     True
1    False
2     True
dtype: bool

>>> s = pd.Series([2, 0,np.nan])
>>> s.between(1, 4,inclusive = False)
0     True
1    False
2    False
dtype: bool
```

isin()用于接受一个列表,并且判断该列中元素是否在列表中。

3. 比较

在 Pandas 中相同长度的 Series 之间,组间比较是允许的,代码如下:

```
>>> a = pd.Series([1, 2, np.nan, 5], index = ['a', 'b', 'd', 'e'])
>>> b = pd.Series([0, 4, np.nan, 5], index = ['a', 'b', 'd', 'f'])
>>> a.lt(b, fill_value = 0)
a    False
b     True
d    False
e    False
f     True
dtype: bool
```

Pandas 中比较运算及数值运算所对应的方法,如表 3-2 所示。

表 3-2 比较运算及数值运算所对应的方法

运算类别	运算符	方法名称
比较运算	<、>、<=、>=、==、!=	.lt()、.gt()、.le()、.ge()、.eq()、.ne()
数值运算	+、-、*、/、//、%、**	.add、.sub、.mul、.div、.floordiv、.mod、.pow

Series 与 DataFrame 中常用的 6 个比较运算符对应的英文全称及中文含义如表 3-3 所示。

表 3-3 常用的 6 个比较运算符

方法	对应的运算符	英文全称	中文含义
.lt()	<	less then	小于
.gt()	>	greater than	大于
.le()	<=	less than or equal to	小于或等于
.ge()	>=	greater than or equal to	大于或等于
.eq()	==	equal to	等于
.ne()	!=	not equal to	不等于

4. any 与 all

any()用于当 Series 中只要有一个值满足条件时其返回值为 True；all()用于当 Series 中所有值均满足条件时其返回值为 True，代码如下：

```
>>> s_ = pd.Series([9,10,11,12])
>>> s_[s_>8].all()
True

>>> s_[s_>12].any()
False
```

第 4 招：删除

当 Series 存在缺失值时，依据指定的方式进行删除，代码如下：

```
>>> s = pd.Series(['R','G','R', np.nan],index=[1,'A',np.nan,5])
>>> s
1      R
A      G
NaN    R
5      NaN
dtype: object

>>> s.drop(labels=['A',np.nan])
1      R
5    NaN
dtype: object

>>> s.dropna(how='any')
1      R
A      G
NaN    R
dtype: object
```

当 Series 的(axis=0)和 DataFrame 的(行方向 axis=0,列方向 axis=1)存在缺失值时,会按指定的 axis 有针对性地删除空值(how='all',所在方向的所有数据都为缺失值时;how='any',所在方向只要有数据为缺失值时)。

除了用 how='all'或 how='any',还可以采用 thresh=n 的方式,例如:thresh=3,当数据所在方向的有效值低于 3 时,删除该行或该列的数据。

第 5 招:去重

drop_duplicate()方法是用于去除特定列下面的重复行,代码如下:

```
>>> s = pd.Series(['R','G','R', np.nan],index = [1,'A',np.nan,5])
>>> s
1         R
A         G
NaN       R
5         NaN
dtype: object

>>> s_ = s.drop_duplicates(keep = 'last')
>>> s_
A         G
NaN       R
5         NaN
dtype: object

>>> s_.drop_duplicates(keep = False, inplace = True)
>>> s_
A         G
NaN       R
5         NaN
dtype: object

>>> s
1         R
A         G
NaN       R
5         NaN
dtype: object
```

keep 有 first、last、False 三个选项,默认为 first;参数 False 表示删除所有重复值。inplace 参数的默认值为 False,inplace=True 代表就地删除(会直接影响数据源)。

第 6 招:填充

填充的模式主要有 bfill(或 backfill)、ffill(或 pad)、None 这 3 种,代码如下:

```
>>> s = pd.Series(['R','G','B'],index=[1,3,5])
>>> s
1    R
3    G
5    B
dtype: object

>>> s=s.reindex(np.arange(0,7),method='ffill')
>>> s
0    NaN
1      R
2      R
3      G
4      G
5      B
6      B
dtype: object
```

在 Pandas 中：可通过 method='ffill'或 method='pad'实现自动向下空值填充，而自动向上空值填充则可通过 method='bfill'或 method='backfill'实现。

填充模式的图解说明如图 3-4 所示。

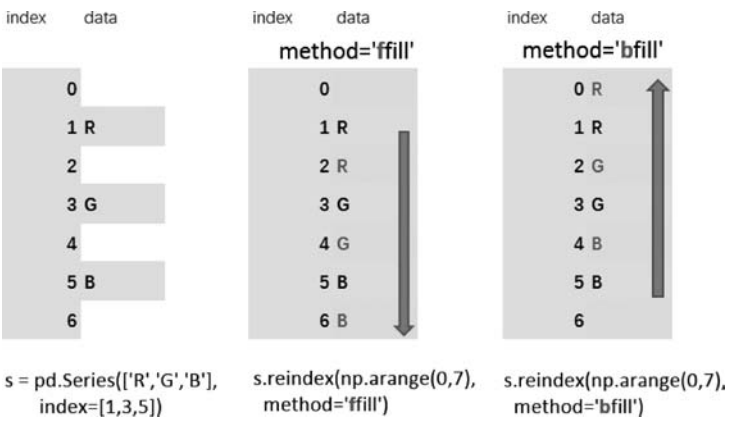

图 3-4　填充模式

第 7 招：修改

在 Pandas 中奉行的是无则新增，有则修改的原则。对于原有数据的重新赋值则意味着修改原有的数据，代码如下：

```
>>> s = pd.Series(['R',np.nan,'G'],index=np.arange(3))
>>> s[5]=83
>>> s
```

```
0      R
1    NaN
2      G
5     83
dtype: object

>>> s[len(s)] = 69     #按索引顺序在最后增加
>>> s
0      R
1    NaN
2      G
5     83
4     69
dtype: object

>>> s.add_prefix('idx_')
idx_0      R
idx_1    NaN
idx_2      G
idx_5     83
idx_4     69
dtype: object

>>> s.add_suffix('_idx')
0_idx      R
1_idx    NaN
2_idx      G
5_idx     83
4_idx     69
dtype: object
```

第 8 招：追加

append()方法用于在列表末尾添加新的对象，代码如下：

```
>>> s1 = pd.Series([1,2,3])
>>> s2 = pd.Series([2] * 3)
>>> s1.append(s2)
0    1
1    2
2    3
0    2
1    2
2    2
dtype: int64
```

第 9 招：变形

经过变形与转换，数据结构会发生改变。常用于数据结构转换的函数有 stack()、unstack()、swaplevel()、ravel()等，代码如下：

```
>>> s = pd.Series(
... [1, 2, 3, 4],
... index = pd.MultiIndex.from_product([
...           ['A', 'B'],['a', 'b']]))
>>> s
A  a    1
   b    2
B  a    3
   b    4
dtype: int64

>>> s.unstack(level = 0)
   A  B
a  1  3
b  2  4

>>> s.unstack(level = -1)
   a  b
A  1  2
B  3  4

>>> s.ravel()
array([1, 2, 3, 4], dtype = int64)

>>> s.reorder_levels([1,0])
a  A    1
b  A    2
a  B    3
b  B    4
dtype: int64

>>> s.swaplevel()
a  A    1
b  A    2
a  B    3
b  B    4
dtype: int64
```

第 10 招：文本

字符串是 Python 中最常用的数据类型之一。相同字符串内或不同字符串间可以进行

字符串运算或格式转换，代码如下：

```
>>> s = pd.Series(['wo', np.nan, 'Wo De', 'Zuguo'])
>>> s.str.upper()
0       WO
1      NaN
2    WO DE
3    ZUGUO
dtype: object

>>> s.str.title()
0       Wo
1      NaN
2    Wo De
3    Zuguo
dtype: object
```

Pandas中的文本操作是一个很值得深入探索的主题，Pandas中的文本操作一般以Series为单位，通过Series的str属性来完成。通过Series.str属性的转换操作，可用于完成一般的字符大小写转换、空值删除等日常性操作，也可用于拼接(cat)、连接(join)、拆分(split)、包含(contains)、匹配(match)、查找(find)、提取(extract)、替换(replace)、重复(repeat)等更多的复杂操作。在这些操作方法中，有些支持正则应用。

第1式：find(查找)

find不支持正则，返回的是索引位置。findall支持正则，返回的是一个列表，代码如下：

```
>>> s = pd.Series(['a1', 'b2', 'c3'])
>>> s.str.find('a')
0    0
1   -1
2   -1
dtype: int64

>>> s.str.findall('a1')
0    [a1]
1      []
2      []
dtype: object

>>> s.str.findall('a1').str[0]
0     a1
1    NaN
2    NaN
dtype: object
```

第 2 式：join(连接)

组内与组间字符串连接的应用举例，代码如下：

```
>>> pd.Series(['Kim','Jim','Joe']).str.join('-')
0    K-i-m
1    J-i-m
2    J-o-e
dtype: object

>>> pd.Series([['Kim','Jim','Joe'],['SH','SZ','BJ']]).str.join('-')
0    Kim-Jim-Joe
1         SH-SZ-BJ
dtype: object
```

s.str.join()只有一个参数(分隔符 sep=)，返回的值是 Series。

第 3 式：cat(拼接)

str.cat()方法主要有两个重要参数，sep=''用于指定分隔符，na_rep=''用于指定空值，代码如下：

```
>>> s = pd.Series(['wo', np.nan, 'Wo De', 'Zuguo'])
>>> s.str.cat(sep = ' ')
'wo Wo De Zuguo'

>>> s.str.cat(sep = ' ', na_rep = '?')
'wo ? Wo De Zuguo'

>>> t = pd.Series(['d', 'a', 'e', 'c'], index = [3, 0, 4, 2])
>>> s.str.cat(t, join = 'left',sep = ' ', na_rep = '-')
0       wo a
1         - -
2    Wo De c
3    Zuguo d
dtype: object
```

第 4 式：repeat(重复)

str.repeat(repeats)用于复制字符串。参数 repeats 可为整型或向量。整型表示每个字符串都复制相同的次数，向量则是可以设置每个元素重复的次数，代码如下：

```
>>> s = pd.Series(['a', 'b', 'c'])
>>> s
0    a
1    b
2    c
```

```
dtype: object

>>> s.str.repeat(repeats = 2)
0    aa
1    bb
2    cc
dtype: object

>>> s.str.repeat(repeats = [1, 2, 3])
0     a
1    bb
2    ccc
dtype: object
```

第 5 式：contains(包含)

str.contains()相当于 SQL 中的 like,用于字符串的模糊筛选,代码如下：

```
>>> s = pd.Series(['wo', 'ai','wo', 'de', 'zuguo'])
>>> s.str.contains('o', regex = False)
0     True
1    False
2     True
3    False
4     True
dtype: bool
```

判断字符是否有包含关系,经常用在数据筛选中,支持正则。

第 6 式：match(匹配)

str.match()的语法为 str.match(pat,case,flags,na),用于确定 Series 对象中的每个字符串是否与正则表达式匹配,代码如下：

```
>>> s = pd.Series(['a1', 'b2', 'c3'])
>>> s.str.match('a')
0     True
1    False
2    False
dtype: bool

>>> s.str.match('a1')
0     True
1    False
2    False
dtype: bool
```

参数 pat 是具有捕获组的正则表达式模式；参数 case 用于区分大小写；参数 flags 可为 re.I、re.G、re.M；na 为缺失值的填充方式。

第 7 式：replace（替换）

str.replace()的语法为 str.replace(pat,repl,regex)，用于字符串的替换。参数 pat 为查找的内容，一般为正则表达式；repl 为要替的字符串，代码如下：

```
>>> (pd.Series(
...  ['wo','ai','wo','de','zuguo'])
...  .str.replace('w.', 'Wo',
...    regex = True))
0       Wo
1       ai
2       Wo
3       de
4    zuguo
dtype: object

>>> (pd.Series(
...  ['wo','ai','wo','de','zuguo'])
...  .str.replace('w.', 'Wo',
...    regex = False))
0       wo
1       ai
2       wo
3       de
4    zuguo
dtype: object
```

可用于文本替换，支持正则。当不想使用正则功能时，可以使用参数 regex＝False 来关闭。

第 8 式：split（拆分）

str.split()的语法结构为 str.split(pat,n,expand)，用于字符串的拆分，类似于 Excel 中的拆分列功能。pat 为字符串或正则表达式，n 默认为－1，expand 默认值为 True，代码如下：

```
>>> pd.Series(['wo ai wo de zuguo']).str.split(" ")
0    [wo, ai, wo, de, zuguo]
dtype: object

>>> pd.Series(['wo ai wo de zuguo']).str.split(n = 4,expand = True)
    0   1   2   3       4
0  wo  ai  wo  de  zuguo
```

第 9 式：exact(提取)

str.extract()的语法为 str.extract(pat,flags,expand),用于从字符数据中抽取匹配的数据。pat 参数为字符串或正则表达式,代码如下：

```
>>> s = pd.Series(['a1', 'b2', 'c3'])
>>> s.str.extract(r'([abc])(\d)')
   0  1
0  a  1
1  b  2
2  c  3

>>> s.str.extract(r'([ab])?(\d)')
     0  1
0    a  1
1    b  2
2  NaN  3

>>> s.str.extract(r'[abc](\d)', expand = True)
   0
0  1
1  2
2  3

>>> s.str.extract(r'[abc](\d)', expand = False)
0    1
1    2
2    3
dtype: object
```

str.extract 可以利用正则将文本中的数据提取出来,从而形成单独的列,支持正则。如果 expand 参数为 True,则返回一个 DataFrame,不管是一列还是多列,当 expand 参数为 False 且只有一列时才会返回一个 Series/Index。

第 11 招：映射

apply 函数是 Pandas 中自由度最高的一个函数,它的语法结构为 df.apply(func,axis,...,** kwds),代码如下：

```
>>> s = pd.Series([1, 2, 3])
>>> def sq(x):
...                     return (0.98 * x) ** 2
s.apply(sq)
0    0.9604
1    3.8416
2    8.6436
```

```
dtype: float64

>>> s = pd.Series([[1, 2, 3],[4,5,6]])
>>> s.apply(np.max)
0    3
1    6
dtype: int64
```

除 apply 之外，后续还会运用较多的映射类函数，如 applymap、map 函数及相关联的 transform、pipe 函数。

第 12 招：排序

sort_values()是 Pandas 中使用频率较高的一种排序方法，代码如下：

```
>>> s = pd.Series([1, 2, np.nan, 5])
>>> s
0    1.0
1    2.0
2    NaN
3    5.0
dtype: float64

>>> s.sort_values(ascending = True)
0    1.0
1    2.0
3    5.0
2    NaN
dtype: float64

>>> s.sort_values(ascending = True,
...               na_position = 'first')
2    NaN
0    1.0
1    2.0
3    5.0
dtype: float64
```

它的语法结构为 DataFrame. sort_values(by, axis＝0, ascending＝True, inplace＝False, kind＝'quicksort', na_position＝'last', ignore_index＝False, key＝None)。

sort_values()的 ascending 参数，默认升序为 True，降序则为 False。如果是列表，则需与 by 所指定的列表数量相同，指明每一列的排序方式。

sort_values()的 na_position 参数，有 first 与 last 两种选择模式，默认值为 last。当指

定的排序列有 nan 值时，nan 值放在序列中第 1 个或最后一个。

第 13 招：计算

加、减、乘、除四则运算是使用频率最高的运算，Pandas 支持不同 Series 间的组间运算，可以采用运算符方式，也可以用函数的方式进行计算，代码如下：

```
>>> s1 = pd.Series([1,2,3])
>>> s2 = pd.Series([2]*3)
>>> s3 = s1+s2*4
>>> s3
0     9
1    10
2    11
dtype: int64

>>> s4 = pd.Series(666, index=[ 1,2,'C'])
>>> s4
1    666
2    666
C    666
dtype: int64

>>> s3 + s4
0      NaN
1    676.0
2    677.0
C      NaN
dtype: float64

>>> s3.add(s4,fill_value=0)
0      9.0
1    676.0
2    677.0
C    666.0
dtype: float64
```

当两个 Series 间不存在空值时，无论是采用运算符方式还是采用函数方式其输出的结果都是一致的，但当两个 Series 间有一个 Series 存在空值或 Index 不一致时，则直接运算后的结果为 nan，可以在 add()及其他函数中用 fill_value=0 解决此问题。

以上代码的图解说明如图 3-5 所示。

Pandas 中，更多的运算函数如表 3-4 所示。

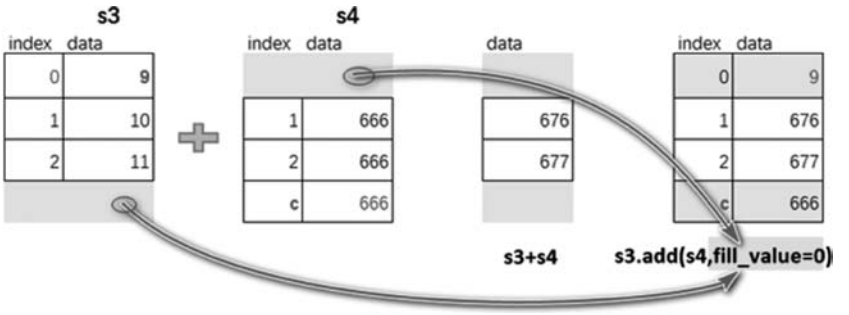

图 3-5　位与字节

表 3-4　Pandas 中的运算函数

	add(),radd()
	sub(),rsub()
	mul(),rmul()
	div(),rdiv(),truediv(),rtruediv(),floordiv(),rfloordiv()
	mod(),rmod()
	pow(),rpow()
	combine(),combine_first()
	round()
	product()
	dot()

上面的 radd()、rsub()等首字母为 r 的函数,其中的 r 代表的是 reverse(反转、使次序颠倒)的意思。例如：pd.Series([1,2,3]).div(0),结果全为 inf；pd.Series([1,2,3]).rdiv(0),结果全为 0.0。其中的差别在于 div(0)中 0 是除数,rdiv(0) 中 0 是被除数。除数为 0,结果为无穷大；被除数为 0(除数不为 0 时),结果为 0。

第 14 招：描述

df.describe()用以完成统计学中的描述性统计分析,用于观测数据的整体趋势,代码如下：

```
>>> s = pd.Series([1,2,3])
>>> s.describe()
count    3.0
mean     2.0
std      1.0
min      1.0
25 %     1.5
50 %     2.0
75 %     2.5
max      3.0
dtype: float64
```

与上述描述性统计分析相对应的函数还有很多。

(1) 与趋势相关的(离中、趋中)：corr()、cov()、kurt()、max()、mean()、median()、min()、mod()、sem()、skew()、std()、var()、kurtosis()、count()、quantile()、nlargest()、nsmallest()、nunique()、value_counts()等。

(2) 与累计相关的：sum()、cummax()、cummin()、cumsum()、cumprod()等。

(3) 与布尔相关的：is_unique()、is_monotonic()、is_monotonic_increasing()、is_monotonic_decreasing()等。

第 15 招：聚合

第 1 式：groupby()

Pandas 的 groupby()功能十分强悍与好用，代码如下：

```
>>> s = pd.Series([1,3,9,2,5],
...     index = ['Kim', 'Jim', 'Joe', 'Tom','Sam'], name = 'ename')
>>> s
Kim    1
Jim    3
Joe    9
Tom    2
Sam    5
Name: ename, dtype: int64

>>> s.groupby(["a", "b", "a", "b",'b']).mean()
a    5.000000
b    3.333333
Name: ename, dtype: float64

>>> s.groupby(level = 0).mean()
Jim    3
Joe    9
Kim    1
Sam    5
Tom    2
Name: ename, dtype: int64

>>> s.groupby(s > 4).mean()
ename
False    2
True     7
Name: ename, dtype: int64
```

它的语法结构为 DataFrame.groupby(by=None,axis=0,level=None,as_index=True,sort=True,group_keys=True,squeeze=False, ** kwargs)。

第 2 式：agg()

agg 是 aggregate 的简写。在 Pandas 中，df.groupby.agg()与 df.groupby.aggregate()是完全等效的。应用 agg()方法后，可以一次性使用多种汇总方式，可以针对不同的列采用不同的汇总方式，可以支持函数的多种写法，代码如下：

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

>>> s.agg('min')
1

>>> s.agg(['min', 'max'])
min    1
max    3
dtype: int64

>>> s.aggregate('min')
1

>>> s.aggregate(['min', 'max'])
min    1
max    3
dtype: int64

>>> s.aggregate(['min', 'max']) == s.agg(['min', 'max'])
min    True
max    True
dtype: bool
```

第 16 招：日期

Pandas 继承了 NumPy 库和 datetime 库中与时间相关的模块，能更高效地处理时间序列数据，代码如下：

```
>>> ts = pd.Series(
...     pd.date_range('2021-03-01',
...         periods=4, freq='2M'))
>>> ts
0   2021-03-31
1   2021-05-31
2   2021-07-31
```

```
3   2021-09-30
dtype: datetime64[ns]

>>> ts = pd.DataFrame(
...     pd.date_range('2021-03-01',
...         periods=4, freq='2M')).set_index(0)
ts.index
DatetimeIndex(['2021-03-31', '2021-05-31', '2021-07-31', '2021-09-30'], dtype=
'datetime64[ns]', name=0, freq=None)
```

第 17 招：时间

时间索引同样可用于序列索引及时间偏移等相关操作，代码如下：

```
>>> ts = pd.Series( index=
...     pd.date_range('2021-03-01',
...         periods=4, freq='36T'),
...         data=[1,2,3,4])

>>> ts.between_time('1:10', '2:45')
2021-03-01 01:12:00    3
2021-03-01 01:48:00    4
Freq: 36T, dtype: int64

>>> ts.shift(periods=1, freq='H')
2021-03-01 01:00:00    1
2021-03-01 01:36:00    2
2021-03-01 02:12:00    3
2021-03-01 02:48:00    4
Freq: 36T, dtype: int64
```

第 18 招：图表

在 Pandas 内可直接调用 plot()方法，代码如下：

```
s = pd.Series([1, 2, 3,4,5,6])
def sq(x):
    return (0.98*x) ** 3
s.apply(sq).plot()
```

输出的图形如图 3-6 所示。

以上十八招可以串起来理解：整体的数据分析是由“清洗、运算、挖掘”三部分完成的。当接触到一个新数据时可以按以下步骤处理。

第 1 步：识别数据类型是否存在空值或异常值等。对空值行统计、填充或删除；对异常值进行去重或按条件筛选、切片等，完成相关清洗工作。

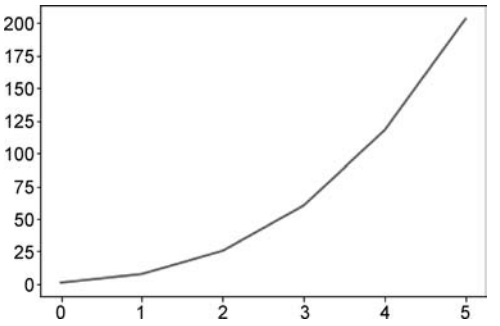

图 3-6　折线图

第 2 步：以目标为导向，对数据进行计算处理与结构转换，完成相关数据的运算工作。

第 3 步：对数据进行描述性分析、探索性挖掘、图形化交互与挖掘工作。

虽然上面列出了这么多招式，但是要强调的是，所有的数据分析与挖掘必须以熟悉业务及明确需求为前提。毕竟，数据分析与挖掘的本质是数据乘以业务，所有（与数据相关的）招式就好比工具，工具一旦离开了它的（业务）应用场景，其（数据分析与挖掘）效果是很难预判的。

3.2　DataFrame

3.2.1　DataFrame 基础知识

DataFrame 是一个表格型的数据结构，它含有一组有序的列。它的每一列可以是不同的值类型，但同一列必须是相同的值类型（数值、日期、object 类型）。其中 object 类型可以保存任何 Python 对象，例如字符串等。

DataFrame 既有行索引，又有列索引。它可以被看作 n 个 Series 组成的字典（共用同一个行索引）。也就是说：DataFrame（结构）＝Index（结构）＋ data（类似于二组数组的数据结构）。以下是 DataFrame 的语法说明，代码如下：

```
pd.DataFrame(
    data = None,          #要传入的数据,必选参数
      #data 可为 ndarray、series、map、lists、dist、常量和另外一个 DataFrame
    index = None,         #行索引,可选参数
      #默认值 np.arange(n),即 0,1,2,3…
    columns = None,       #列索引,可选参数
      #默认为 np.arange(n),即 0,1,2,3…
    dtype = None,         #每列的数据类型,可选参数
    copy = False,         #从 input 输入中复制数据
)
```

作用：通过对各类数据输入或结构转换后，生成 Pandas 的 DataFrame 数据结构。

DataFrame 的结构如图 3-7 所示。

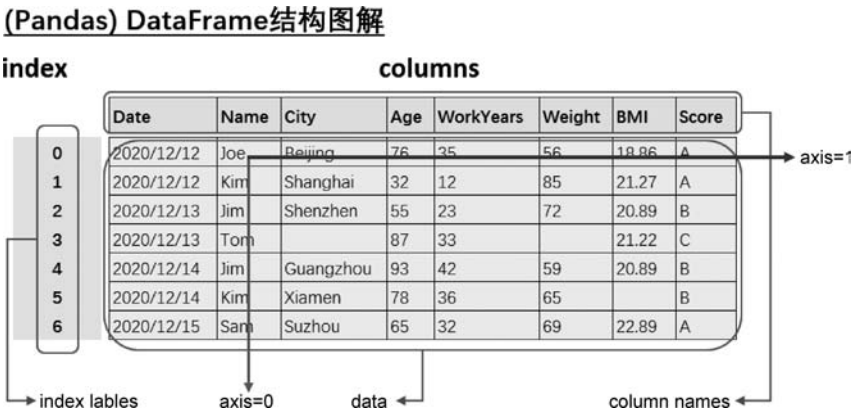

图 3-7 DataFrame 数据结构

Pandas 的 DataFrame 是与 Excel、SQL 等类似的表格型数据结构，由 index、columns、data 三部分组成。其中，DataFrame 的 data 部分与 NumPy 的 ndarray 是一致的。

在 DataFrame 中，axis=0 与 axis="index"是等价的，axis=1 与 axis="columns"是等价的。例如：在 df.iloc[:,3:7].sum(1)中，sum(1)的 1 代表的就是 axis=1 的方向，这是一种简写方式。

3.2.2 创建

1. 文件导入生成

pd.read_excel(io,sheet_name=0,...)默认打开的 sheet_name 是导入的 Excel 对象中的第 1 个电子表格，可以省略不写，代码如下：

```
pd.read_excel('demo_.xlsx')
```

输出的结果如下：

```
      Date        Name     City      Age   WorkYears   Weight    BMI    Score
0 2020-12-12   Joe    Beijing    76       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   55       23        72.0    20.89     B
3 2020-12-13   Tom      NaN      87       33         NaN    21.22     C
4 2020-12-14   Jim    Guangzhou  93       42        59.0    20.89     B
5 2020-12-14   Kim    Xiamen     78       36        65.0     NaN      B
6 2020-12-15   Sam    Suzhou     65       32        69.0    22.89     A
```

为了让读者聚焦于语法及便于理解返回的值，本书的大部分代码演示围绕着上面这 7

行 8 列的数据展开。

pd. read_excel()：

作用：将 Excel 读到 Pandas 的 DataFrame。

说明：Pandas 的核心在于数据分析，而不是数据文件的读取与写入，但是，从外部文件中读写数据，仍属于 Pandas 的重要组成部分。Pandas 提供了很多 API，以支持对外部数据（Excel、CSV、SQL、JSON、HTML、Picklle、HDF 等）的读写。在日常工作中，使用最多的是 pd. read_excel()和 pd. read_csv()两种方式。

以下是 pd. read_excel()的语法说明：

```
pd.read_excel(
    io,                    #相关 Excel 文件的存储路径
    sheet_name = 0,        #要读取的工作表名称
    header = 0,            #用哪一行作为列名
    names = None,          #自定义最终的列名
    index_col = None,      #用作索引的列
    usecols = None,        #需要读取哪些列
    squeeze = False,       #当数据仅包含一列
    dtype = None,          #指定的数据类型
    …… 
)
```

更多的参数说明与解析，会在第 7 章进行讲解。

2. Series 创建

在 Series 构建或转换为 DataFrame 的过程中，可以有多种构建或转换方式。

1）单个 Series 创建

Series 创建 DataFrame，代码如下：

```
pd.Series([32,55,65],name = "Age").to_frame()
#将一个 Seires 转换为只有一列的表格
```

输出的结果如下：

```
    Age
0   32
1   55
2   65
```

2）多个 Series 创建

以下是由多个 Series 合并为一个 DataFrame 的应用，代码如下：

```
>>> s1 = pd.Series(['Kim','Jim','Sam'])
>>> s2 = pd.Series((32,55,65))
```

```
>>> s3 = pd.Series(('Shanghai','Shenzhen','Suzhou'))
>>> pd.DataFrame(zip(s1,s2,s3),
...              columns = ['Name','Age','City'])
>>> #分别将不同的 Series 写在 zip 中以便转换与创建
   Name  Age    City
0   Kim   32   Shanghai
1   Jim   55   Shenzhen
2   Sam   65    Suzhou

>>> pd.DataFrame(zip( *[s1,s2,s3]),
...              columns = ['Name','Age','City'])
>>> #多个 Series 在容器中的转换与创建
   Name  Age    City
0   Kim   32   Shanghai
1   Jim   55   Shenzhen
2   Sam   65    Suzhou
```

3. 字典创建

以下是通过字典数据创建 Dataframe 的应用，代码如下：

```
pd.DataFrame({
    "Name":["Kim",'Jim', 'Sam'],
"Age":[32,55,65],
"City":["Shanghai",'Shenzhen','Suzhou']
})
```

输出的结果如下：

```
  Name  Age      City
0  Kim   32  Shanghai
1  Jim   55  Shenzhen
2  Sam   65    Suzhou
```

4.（二维）列表创建

以下是通过二维列表创建 Dataframe 的应用，代码如下：

```
df = pd.DataFrame([
    ["Kim",'Jim','Sam'],
    [32,55,65],
    ["Shanghai",'Shenzhen','Suzhou']],
    index = ['Name','Age','City'])
df.T
```

输出的结果如下：

```
  Name Age      City
0  Kim  32  Shanghai
1  Jim  55  Shenzhen
2  Sam  65    Suzhou
```

5. 元组创建

以下是通过元组创建 DataFrame 的应用，代码如下：

```
pd.DataFrame(
    data = (
        ('Kim',32,'Shanghai'),
        ('Jim',55,'Shenzhen'),
        ('Sam',65,'Suzhou')),
    columns = ['Name','Age','City']
)
```

输出的结果如下：

```
  Name  Age       City
0  Kim   32   Shanghai
1  Jim   55  Shenzhen
2  Sam   65     Suzhou
```

3.2.3 DataFrame 相关知识

1. to_x()回顾

在 Series 中，有 astype()、convert_dtypes()、to_x()等转换方法。在 DataFrame 中，有 to_numpy()、to_dict()、to_string()等转换方法。

代码如下：

```
>>> df = pd.read_excel('demo_.xlsx').head(3)
>>> df.to_numpy()
array([[Timestamp('2020-12-12 00:00:00'), 'Joe', 'Beijing', 76, 35, 56.0,
        18.86, 'A'],
       [Timestamp('2020-12-12 00:00:00'), 'Kim', 'Shanghai', 32, 12,
        85.0, 21.27, 'A'],
       [Timestamp('2020-12-13 00:00:00'), 'Jim', 'Shenzhen', 55, 23,
        72.0, 20.89, 'B']], dtype=object)

>>> df.to_dict()
{'Date': {0: Timestamp('2020-12-12 00:00:00'), 1: Timestamp('2020-12-12 00:00:00'),
2: Timestamp('2020-12-13 00:00:00')}, 'Name': {0: 'Joe', 1: 'Kim', 2: 'Jim'}, 'City': {0:
'Beijing', 1: 'Shanghai', 2: 'Shenzhen'}, 'Age': {0: 76, 1: 32, 2: 55}, 'WorkYears': {0: 35, 1: 12,
2: 23}, 'Weight': {0: 56.0, 1: 85.0, 2: 72.0}, 'BMI': {0: 18.86, 1: 21.27, 2: 20.89}, 'Score':
{0: 'A', 1: 'A', 2: 'B'}}
```

从输出的结果来看：to_numpy()生成的是 ndarray 对象，to_dict()生成的是 dict(字典)对象，而 to_string()生成的是 str 对象。

如何部分截取 DataFrame 中的内容，如图 3-8 所示。

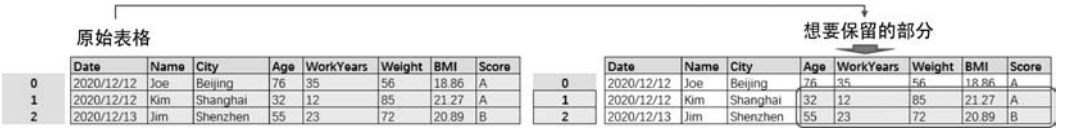

图 3-8 部分截取 DataFrame 中的内容

代码如下：

```
>>> pd.DataFrame(df.iloc[1:, 3:].to_dict())
   Age  WorkYears  Weight   BMI   Score
1   32       12      85.0   21.27    A
2   55       23      72.0   20.89    B

>>> pd.DataFrame(df.to_numpy()[1:, 3:],
...         columns = ['Age', 'WorkYears', 'Weight', 'BMI', 'Score'])
   Age  WorkYears  Weight   BMI   Score
0   32       12       85    21.27    A
1   55       23       72    20.89    B
```

关于 df.to_x()方法，在使用过程中稍留心会发现有 20 多种。能够轻松驾驭数据结构间的相互转换将使数据分析变得更为灵活高效，代码如下：

```
>>> pd.Series(
...     df['Name'].to_numpy(),
...     index = df['City'])
City
Beijing     Joe
Shanghai    Kim
Shenzhen    Jim
dtype: object

>>> pd.DataFrame(
...     pd.Series(df['Name'].to_numpy(),index = df['City']),
...     columns = ["Name"]
... ) #注意："Name"外面的[]不可省，否则会报错
          Name
City
Beijing   Joe
Shanghai  Kim
Shenzhen  Jim
```

2. index()相关

1) index_col 设置

导入时可直接设置索引列,代码如下:

```
>>> pd.read_excel('demo_.xlsx',index_col = 'City').head(2)
                Date   Name  Age  WorkYears  Weight   BMI    Score
City
Beijing   2020 - 12 - 12  Joe   76       35     56.0    18.86     A
Shanghai 2020 - 12 - 12  Kim   32       12     85.0    21.27     A
```

2) set_index 与 reset_index()

先导入数据再设置索引列,代码如下:

```
pd.read_excel('demo_.xlsx').set_index("City")
```

结果如图 3-9 所示。

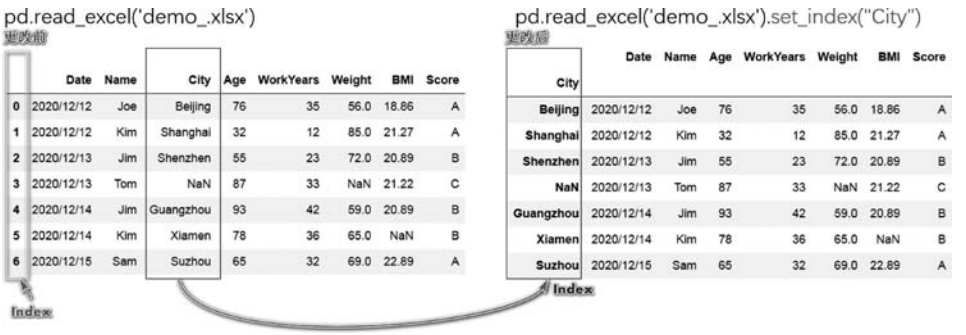

图 3-9 设置索引列

取消原有索引列,重置并将多列设置为索引列,代码如下:

```
df.reset_index().set_index(['Name', 'City'], drop = False)
```

结果如图 3-10 所示。

图 3-10 重设索引列并保留原索引列

DataFrame 的 set_index 函数会将其一个或多个列转换为行索引，并创建一个新的 DataFrame。默认情况下，那些列会从 DataFrame 中移除，但也可以用 drop＝False 将其保留下来。

3. 属性

DataFrame 中的部分属性如图 3-11 所示。

```
1 import pandas as pd
2 df = pd.read_excel('demo_.xlsx')
```

```
1 df.ndim
```

```
2
```

```
1 df.shape
```

```
(7, 8)
```

```
1 df.size
```

```
56
```

```
1 df.index
```

```
RangeIndex(start=0, stop=7, step=1)
```

```
1 df.columns
```

```
Index(['Date', 'Name', 'City', 'Age', 'WorkYears', 'Weight', 'BMI', 'Score'], dtype='object')
```

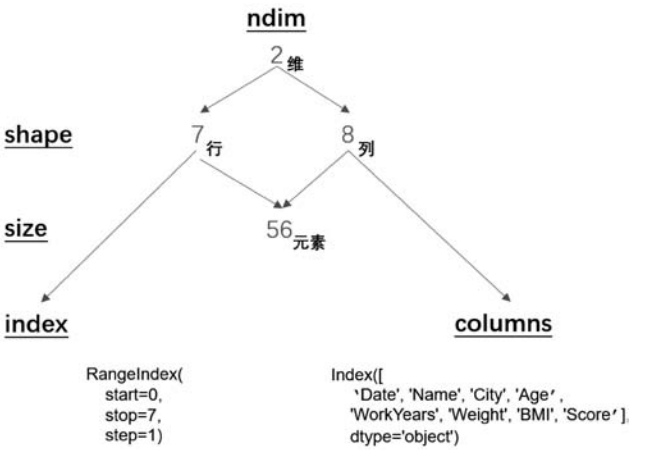

图 3-11 DataFrame 的属性

如果想查看各列的数据类型，可以用 dtypes 属性(注意：查看 Series 的数据类型可采用 dtype，而查看 DataFrame 的数据类型采用的是 dtypes)；如果想知道 DataFrame 中数据类型的总数，则可以在 dtypes 属性后再加上 value_counts()方法。演示代码如下：

```
>>> df = pd.read_excel("demo_.xlsx")
>>> df.dtypes
Date          datetime64[ns]
Name                  object
City                  object
Age                    int64
WorkYears              int64
Weight               float64
BMI                  float64
Score                 object
dtype: object

>>> df.dtypes.value_counts()
object           3
```

```
int64              2
float64            2
datetime64[ns]     1
dtype: int64

>>> df.Date.dtype
dtype('<M8[ns]')

>>> df.info()
<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       int64
4   WorkYears   7 non-null       int64
5   Weight      6 non-null       float64
6   BMI         6 non-null       float64
7   Score       7 non-null       object
dtypes: datetime64[ns](1), float64(2), int64(2), object(3)
memory usage: 576.0+ Bytes
```

从图 3-11 可以发现：这个 DataFrame 共由 7 行 8 列组成(列名有 Date 和 Name 等)，这 8 列中：float64 有 2 列，int64 有 2 列，object 有 4 列(float64、int64、object 为数据类型，可以用.dtype 属性查看)。例如：输入 df.Date.dtype，输出为 dtype('O')，直接查看的是 df['Date']列的数据属性。

3.3 本章回顾

Pandas 在数据处理与分析及图形化呈现方面，功能十分强大。当然，Pandas 之所以能有如此强大的功能与其有着丰富的方法与属性分不开。

十八般武艺并非局限 18 种武艺，十八般武艺只是众多武艺的一个概说。以上列出的十八招及文本操作的一招九式，也只是 Pandas 库里丰富的方法与属性应用的一个缩影。掌握好十八招是 Pandas 入门的必经之路，这中间少不了对语法及参数的掌握及反复使用所形成的肌肉记忆，最后逐步形成自己的知识体系并相互融会贯通，让 Pandas 真正实现由了解到强大再过渡到真正应用上的强大。

Pandas 本身就好比一个有着器灵存在的上品神器，它具备力量法则、空间法则、时间法则等加持于一身，使之具备越阶作战的能力。具备众多的 API 及与第三方库无缝链接，使

之力量备受加持，是力量法则的体现，即管理学所讲的“赋能”。众多应用场景中的各类花式的数据的筛选与转换，是空间法则的体现。Pandas 中强大的时间序列功能，通过各类频率的转换、重采样、时间窗口等功能，让时间拉长、变短、偏移等，是时间法则的体现。同时，它拥有攻击属性和防护属性于一体，攻击属性体现在其强大的数据分析与挖掘能力，防护属性体现在其灵活的数据批量获取与存储、数据的结构与类型转换等。

面对如此功能强大的上品神器，读者就是它的新认主人，是否打算立马炼化它？总之，纸上得来终觉浅，绝知此事须躬行。