第5章 Pandas基本用法

Pandas是一个高性能的数据操作和分析工具。它在NumPy的基础上，提供了一种高效的DataFrame数据结构，使得在Python中进行数据清洗和分析非常快捷。Pandas采用了很多NumPy的代码风格，但最大的不同在于Pandas主要用来处理表格型或异质型数据，而NumPy则相反，它更适合处理同质并且是数值类型的数据。事实上大多数时候，使用Pandas多于NumPy。

通常都使用Anaconda发行版安装Pandas，如果自行安装，可以使用如下命令：

```
pip install pandas
conda install pandas
python3 -m pip install --upgrade pandas
```

安装完Pandas后，就可以导入它了，通常会使用下面的惯例进行导入：

```
import pandas as pd
```

有时候也会将它包含的两个重要数据结构Series和DataFrame也单独导入：

```
from pandas import Series, DataFrame
```

可以用如下命令查看当前Pandas的版本信息：

```
pd.__version__
```

Pandas的核心是三大数据结构：Series、DataFrame和Index。绝大多数操作都是围绕这三种结构进行的。

5.1 Series

Series是一个一维的数组对象，它包含一个值序列和一个对应的索引序列。NumPy的一维数组通过隐式定义的整数索引获取元素值，而Series用一种显式定义的索引与元素关联。显式索引让Series对象拥有更强的能力，索引也不再仅仅是整数，还可以是别的类型，如字符串。索引也不需要连续，也可以重复，自由度非常高。

最基本的生成Series对象的方式是使用Series构造器：

In[]：

```
import pandas as pd
```

In[]：

```
s = pd.Series([7, -3,4, -2])
```

In[]：

```
s
```

Out[]：

```
0    7
1   -3
2    4
3   -2
dtype: int64
```

打印时，自动对齐了，看起来比较美观。左边是索引，右边是实际对应的值。默认的索引是 0～$N-1$（N 是数据的长度）。可以通过 values 和 index 属性分别获取 Series 对象的值和索引。

In[]：

```
s.dtype
```

Out[]：

```
dtype('int64')
```

In[]：

```
s.values
```

Out[]：

```
array([ 7, -3,  4, -2], dtype=int64)
```

In[]：

```
s.index
```

Out[]：

```
RangeIndex(start=0, stop=4, step=1)
```

可以在创建 Series 对象的时候指定索引。

In[]：

```
s2 = pd.Series([7, -3,4, -2], index=['d','b','a','c'])
```

In[]：

```
s2
```

Out[]：

```
d    7
```

```
b  -3
a   4
c  -2
dtype: int64
```

In[]:

```
s2.index
```

Out[]:

```
Index(['d', 'b', 'a', 'c'], dtype = 'object')
```

In[]:

```
pd.Series(5, index = list('abcde'))
```

Out[]:

```
a    5
b    5
c    5
d    5
e    5
dtype: int64
```

In[]:

```
pd.Series({2:'a',1:'b',3:'c'}, index = [3,2])        # 通过 index 筛选结果
```

Out[]:

```
3    c
2    a
dtype: object
```

也可以在后期,直接修改 index。

In[]:

```
s
```

Out[]:

```
0    7
1   -3
2    4
3   -2
dtype: int64
```

In[]:

```
s.index = ['a','b','c','d']
```

In[]:

```
s
```

Out[]：

```
a    7
b   -3
c    4
d   -2
dtype: int64
```

类似 Python 的列表和 NumPy 的数组，Series 也可以通过索引获取对应的值。

In[]：

```
s2 = pd.Series([7, -3,4, -2], index = ['d','b','a','c'])
```

In[]：

```
s2['a']
```

Out[]：

```
4
```

In[]：

```
s2[['c','a','d']]
```

Out[]：

```
c   -2
a    4
d    7
dtype: int64
```

也可以对 Seires 执行一些类似 NumPy 的通用函数操作。

In[]：

```
s2[s2 > 0]
```

Out[]：

```
d    7
a    4
dtype: int64
```

In[]：

```
s2 * 2
```

Out[]：

```
d    14
b    -6
a    8
c    -4
dtype: int64
```

In[]:

```
import numpy as np
```

In[]:

```
np.exp(s2)
```

Out[]:

```
d    1096.633158
b       0.049787
a      54.598150
c       0.135335
dtype: float64
```

因为索引可以是字符串，所以从某个角度看，Series 又比较类似 Python 的有序字典，因此可以使用 in 操作。

In[]:

```
'b' in s2
```

Out[]:

```
True
```

In[]:

```
'e'in s2
```

Out[]:

```
False
```

同样地，也会想到使用 Python 的字典来创建 Series。

In[]:

```
dic = {'beijing':35000,'shanghai':71000,'guangzhou':16000,'shenzhen':5000}
```

In[]:

```
s3 = pd.Series(dic)
```

In[]:

```
s3
```

Out[]:

```
beijing      35000
shanghai     71000
guangzhou    16000
shenzhen      5000
dtype: int64
```

In[]:

```
s3.keys() # 同样地,具有类似字典的方法
```

Out[]:

```
Index(['beijing', 'shanghai', 'guangzhou', 'shenzhen'], dtype = 'object')
```

In[]:

```
s3.items()
```

Out[]:

```
< zip at 0x1a5c2d88c88 >
```

In[]:

```
list(s3.items())
```

Out[]:

```
[('beijing', 35000),
('shanghai', 71000),
('guangzhou', 16000),
('shenzhen', 5000)]
```

In[]:

```
s3['changsha'] = 20300
```

In[]:

```
city = ['nanjing', 'shanghai','guangzhou','beijing']
```

In[]:

```
s4 = pd.Series(dic, index = city)
```

In[]:

```
s4
```

Out[]:

```
nanjing           NaN
shanghai      71000.0
guangzhou     16000.0
beijing       35000.0
dtype: float64
```

city 列表中，多了 nanjing，但少了 shenzhen。Pandas 会依据 city 中的关键字去 dic 中查找对应的值，因为 dic 中没有 nanjing，这个值缺失，所以用专门的标记值 NaN 表示。因为 city 中没有 shenzhen，所以在 s4 中也不会存在 shenzhen 这个条目。可以看出，索引很关键，在这里起到了决定性的作用。

在 Pandas 中,可以使用 isnull()和 notnull()函数来检查缺失的数据。

In[]:

```
pd.isnull(s4)
```

Out[]:

```
nanjing       True
shanghai     False
guangzhou    False
beijing      False
dtype: bool
```

In[]:

```
pd.notnull(s4)
```

Out[]:

```
nanjing      False
shanghai      True
guangzhou     True
beijing       True
dtype: bool
```

In[]:

```
s4.isnull()
```

Out[]:

```
nanjing       True
shanghai     False
guangzhou    False
beijing      False
dtype: bool
```

可以为 Series 对象和其索引设置 name 属性,这有助于标记识别。

In[]:

```
s4.name = 'people'
```

In[]:

```
s4.index.name = 'city'
```

In[]:

```
s4
```

Out[]:

```
city
nanjing          NaN
shanghai     71000.0
```

```
guangzhou    16000.0
beijing      35000.0
Name: people, dtype: float64
```

In[]：

```
s4.index
```

Out[]：

```
Index(['nanjing', 'shanghai', 'guangzhou', 'beijing'], dtype = 'object', name = 'city')
```

5.2 DataFrame

DataFrame 是 Pandas 的核心数据结构，表示的是二维的矩阵数据表，类似关系数据库的结构，每一列可以是不同的值类型，如数值、字符串、布尔值等。DataFrame 既有行索引，也有列索引，它可以被看作一个共享相同索引的 Series 的字典。

5.2.1 创建 DataFrame 对象

In[]：

```
dates = ['2021 - 01 - 01','2021 - 01 - 02','2021 - 01 - 03',
       '2021 - 01 - 04','2021 - 01 - 05','2021 - 01 - 06']
dates = pd.to_datetime(dates)
dates
```

Out[]：

```
DatetimeIndex(['2021 - 01 - 01','2021 - 01 - 02','2021 - 01 - 03',
               '2021 - 01 - 04','2021 - 01 - 05','2021 - 01 - 06'],
dtype = 'datetime64[ns]',freq = None)
```

In[]：

```
df = pd.DataFrame(np.random.randn(6,4),index = dates,columns = list('ABCD'))
df
```

输出结果如图 5-1 所示。

	A	B	C	D
2021-01-01	-0.248793	-0.090349	0.182044	3.971109
2021-01-02	-0.583737	-1.648617	0.873211	-0.848886
2021-01-03	-0.061564	-0.877747	0.347661	-0.829144
2021-01-04	-0.070906	-0.078362	-0.369856	-0.811988
2021-01-05	0.245959	2.130982	0.327305	0.615350
2021-01-06	-0.757878	-0.162731	-0.557198	-0.580417

图 5-1 创建 DataFrame 对象

5.2.2 查看 DataFrame 对象

In[]：

```
df.head(3)
```

前 3 行输出结果如图 5-2 所示。

In[]：

```
df.tail(4)
```

后 4 行输出结果如图 5-3 所示。

In[]：

```
df.columns
```

	A	B	C	D
2021-01-01	-0.248793	-0.090349	0.182044	3.971109
2021-01-02	-0.583737	-1.648617	0.873211	-0.848886
2021-01-03	-0.061564	-0.877747	0.347661	-0.829144

图 5-2 查看 DataFrame 对象前 3 行

	A	B	C	D
2021-01-03	-0.061564	-0.877747	0.347661	-0.829144
2021-01-04	-0.070906	-0.078362	-0.369856	-0.811988
2021-01-05	0.245959	2.130982	0.327305	0.615350
2021-01-06	-0.757878	-0.162731	-0.557198	-0.580417

图 5-3 查看 DataFrame 对象后 4 行

Out[]：

```
Index(['A', 'B', 'C', 'D'], dtype = 'object')
```

In[]：

```
df.index
```

Out[]：

```
DatetimeIndex(['2021 - 01 - 01','2021 - 01 - 02','2021 - 01 - 03',
               '2021 - 01 - 04','2021 - 01 - 05','2021 - 01 - 06'],
                dtype = 'datetime64[ns]',freq = None)
```

In[]：

```
df.values
```

Out[]：

```
array([[ - 0.24879283, - 0.09034927, 0.18204419, 3.9711095 ],
       [ - 0.58373675, - 1.64861704, 0.87321137, - 0.84888624],
       [ - 0.06156446, - 0.8777472 , 0.34766089, - 0.82914362],
       [ - 0.07090626, - 0.07836195, - 0.36985636, - 0.81198838],
       [ 0.24595941, 2.13098157, 0.32730456, 0.61535036],
       [ - 0.75787777, - 0.16273068, - 0.55719831, - 0.58041744]])
```

In[]：

```
df.describe()  # 查看数值数据的详细信息
```

输出结果如图 5-4 所示。

	A	B	C	D
count	6.000000	6.000000	6.000000	6.000000
mean	-0.246153	-0.121137	0.133861	0.252671
std	0.369538	1.263503	0.522185	1.906285
min	-0.757878	-1.648617	-0.557198	-0.848886
25%	-0.500001	-0.698993	-0.231881	-0.824855
50%	-0.159850	-0.126540	0.254674	-0.696203
75%	-0.063900	-0.081359	0.342572	0.316408
max	0.245959	2.130982	0.873211	3.971109

图 5-4　数值数据的详细信息 1

5.2.3　DataFrame 对象的索引与切片

1. DataFrame 行与列的单独操作

```
df[1:3]                          # 行操作
df['A']                          # 列操作
df[['A','C']]                    # 多列操作
df[df['A']>0]                    # bool 值操作
```

2. 标签索引与切片

loc 利用 index 的名称获取想要的行(或列)。

```
df.loc[:,'A']                    # 提取 A 列数据
df.loc[:,'A':'C']                # 提取 A～C 列数据
df.loc[dates[0:2],'A':'C']       # 提取 0、1 行的 A～C 列数据
df.loc[dates[0],'A']             # 提取 0 行的 A 列数据
df.at[dates[0],'A']              # 提取 0 行的 A 列数据
df.loc[df.loc[:,'A']>0]          # 提取 A 列大于 0 的行
```

3. 位置索引与切片

iloc 利用 index 的具体位置(所以它只能是整数型参数)获取想要的行(或列)。

```
df.iloc[2]                       # 提取行为 2(第 3 行)的数据
df.iloc[:,2]                     # 提取列为 2(第 3 列)的数据
df.iloc[[1,4],[2,3]]             # 提取行为 1、4,列为 2、3 的数据
df.iloc[1:4,2:4]                 # 提取行为 1～3,列为 2、3 的数据
df.iloc[3,3]                     # 提取行为 3,列为 3 的数据
df.iat[3,3]                      # 提取行为 3,列为 3 的数据
df.loc[:,df.iloc[3]>0]           # 提取所有行,列为 3 的大于 0 的数据
```

4. DataFrame 的操作

1）转置

```
df.T
```

2）排序与排名

```
df.sort_index(axis = 0, ascending = False)          # 按行标签排序
df.sort_index(axis = 1, ascending = False)          # 按列标签排序
df.sort_values(by = 'C')                            # 按某列排序
df.sort_values(by = '2016 - 01 - 05',axis = 1)      # 按某行排序
```

3）增加列

```
s1 = pd.Series([1,2,3,4,5,6],index = pd.date_range('20160101',periods = 6))
df['E'] = s1
```

4）增加行

方式 1：

```
df1 = pd.DataFrame({'A':[1,2,3],'B':[4,5,6],'C':[7,8,9]},\
                   index = pd.date_range('20160110',periods = 3))
df.append(df1)
```

方式 2：

```
data = np.random.randn(1,4)
date = pd.to_datetime('20160107')
df.loc[date,] = data
```

方式 3：

```
pd.concat([df,df1],join = 'inner')
```

5）删除操作

```
df.drop(dates[1:3])                     # 删除 1、2 行的数据,df 并没有删除
df.drop('A',axis = 1)                   # 删除 A 列的数据,注意删除列时,需要指定 axis = 1
del df['A']                             # 删除 A 列的数据,df 并没有删除
df.drop(dates[1:3],inplace = True)      # 删除 1、2 行的数据,df 相应行删除
```

6）替换操作

```
df.loc[dates[2],'C'] = 0
df.iloc[0,2] = 0
df.loc[:,'B'] = np.arange(0,len(df))    # 替换 B 列
df.loc[date,] = data                    # 替换 date 所在行的数据
```

5. DataFrame 的运算

1）简单运算

Series 与 Series 运算：匹配规则为 index 与 index，如不能匹配，就用 NaN。

In[]：

```
s1 = pd.Series([1,2,3],index = list('ABC'))
```

```
s2 = pd.Series([4,5,6],index = list('BCD'))
s1 + s2
```

Out[]：

```
A    NaN
B    6.0
C    8.0
D    NaN
dtype: float64
```

Series 与 DataFrame 运算：匹配规则为 index 与 column，如不能匹配，就用 NaN，每行元素都进行列操作。

In[]：

```
df1 = pd.DataFrame(np.arange(1,13).reshape(3,4),
                   index = list('abc'),columns = list('ABCD'))
df1 - s1
```

Out[]：

```
     A   B     C   D
a  0.0  0.0  0.0  NaN
b  4.0  4.0  4.0  NaN
c  8.0  8.0  8.0  NaN
```

DataFrame 与 DataFrame 运算：匹配规则为 index 与 column 同时匹配，如不能匹配，就用 NaN，每行元素都进行列操作。

In[]：

```
df2 = pd.DataFrame(np.arange(1,13).reshape(4,3),
                   index = list('bcde'),columns = list('CDE'))
df1 * df2
```

Out[]：

```
    A    B    C    D    E
a  NaN  NaN  NaN  NaN  NaN
b  NaN  NaN  7.0  16.0 NaN
c  NaN  NaN 44.0  60.0 NaN
d  NaN  NaN  NaN  NaN  NaN
e  NaN  NaN  NaN  NaN  NaN
```

2）函数的应用和映射

函数基本形式为：

```
DataFrame.apply(func,axis = 0)
```

In[]：

```
df0 = pd.DataFrame(np.random.rand(6,4),
                   index = pd.date_range('20160101',periods = 6),
                     columns = list('ABCD'))
```

```
df0.apply(max,axis = 0)
```

Out[]：

```
A    0.945795
B    0.695932
C    0.923623
D    0.594854
dtype: float64
```

也可以使用自定义函数：

```
f = lambda x: x.max() - x.min()
df0.apply(f,axis = 1)
```

6. 数据规整化

1）使用 pd.concat()函数实现简易合并

In[]：

```
ser1 = pd.Series(['A', 'B', 'C'], index = [1, 2, 3])
ser2 = pd.Series(['D', 'E', 'F'], index = [4, 5, 6])
pd.concat([ser1, ser2])
```

Out[]：

```
1    A
2    B
3    C
4    D
5    E
6    F
dtype: object
```

2）合并与连接

（1）一对一连接。

In[]：

```
df1 = pd.DataFrame({
 'employee' : ['Bob', 'Jake', 'Lisa', 'Sue'],
 'group' : ['Accounting', 'Engineering', 'Engineering', 'HR']
})
 df2 = pd.DataFrame({
 'employee' : ['Lisa', 'Bob', 'Jake', 'Sue'],
 'hire_data' : [2004, 2008, 2012, 2014]
})
 print(df1); print(df2)
```

Out[]：

```
  employee        group
0      Bob   Accounting
1     Jake  Engineering
2     Lisa  Engineering
```

```
3      Sue          HR
  employee  hire_data
0      Lisa       2004
1       Bob       2008
2      Jake       2012
3       Sue       2014
```

In[]：

```
df3 = pd.merge(df1, df2)
df3
```

Out[]：

```
     employee     group    hire_data
0       Bob   Accounting       2008
1      Jake  Engineering       2012
2      Lisa  Engineering       2004
3       Sue           HR       2014
```

（2）多对一连接。

In[]：

```
df4 = pd.DataFrame({
 'group' : ['Accounting', 'Engineering', 'HR'],
 'supervisor' : ['Carly', 'Guido', 'Steve']
    })
    print(pd.merge(df3, df4))
```

Out[]：

```
    employee         group  hire_data supervisor
0        Bob     Accounting       2008       Carly
1       Jake     Engineering      2012       Guido
2       Lisa     Engineering      2004       Guido
3        Sue              HR      2014       Steve
```

（3）多对多连接。

In[]：

```
df5 = pd.DataFrame({
 'group' : ['Accounting', 'Accounting', 'Engineering', 'Engineering', 'HR', 'HR'],
 'skills' : ['math', 'spreadsheets', 'coding', 'linux', 'spreadsheets', 'organization']
 })
 print(pd.merge(df1, df5))
```

Out[]：

```
  employee       group         skills
0      Bob   Accounting           math
1      Bob   Accounting  spreadsheets
2     Jake  Engineering         coding
3     Jake  Engineering          linux
```

```
4     Lisa  Engineering        coding
5     Lisa  Engineering         linux
6      Sue           HR  spreadsheets
7      Sue           HR  organization
```

(4) 设置数据合并的键。

In[]:

```
print(pd.merge(df1, df2, on = 'employee'))
```

Out[]:

```
  employee         group  hire_data
0      Bob    Accounting       2008
1     Jake   Engineering       2012
2     Lisa   Engineering       2004
3      Sue            HR       2014
```

In[]:

```
df3 = pd.DataFrame({
     'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
     'salary': [70000, 80000, 120000, 90000]
     })
    print(df1)
    print(df3)
    print(pd.merge(df1, df3, left_on = 'employee', right_on = 'name'))
```

Out[]:

```
  employee         group
0      Bob    Accounting
1     Jake   Engineering
2     Lisa   Engineering
3      Sue            HR
  name   salary
0   Bob   70000
1  Jake   80000
2  Lisa  120000
3   Sue   90000
  employee         group  name  salary
0      Bob    Accounting   Bob   70000
1     Jake   Engineering  Jake   80000
2     Lisa   Engineering  Lisa  120000
3      Sue            HR   Sue   90000
```

3) 透视表

```
pivot_table(data,values = None,index = None,columns = None,aggfunc = 'mean',fill_value = None,
margins = False,dropna = True,argins_name = 'All')
```

index：透视表的行索引，必要参数，如果想要设置多层次索引，使用列表[ ]。

values：对目标数据进行筛选，默认是全部数据，可以通过 values 参数设置想要展示的数据列。

columns：透视表的列索引，非必要参数，同 index 使用方式一样。

aggfunc：对数据聚合时进行的函数操作，默认是求平均值，也可以用 sum、count 等。

margins：为 True 时，会添加行/列的总计，默认对行/列求和。

fill_value：对空值进行填充。

dropna：默认开启去重。

margins_name：margins＝True 时，设定 margins 行/列的名称。

In[]：

```
df = pd.DataFrame({"A": ["foo", "foo", "foo", "foo", "foo",
                         "bar", "bar", "bar", "bar"],
                   "B": ["one", "one", "one", "two", "two",
                         "one", "one", "two", "two"],
                   "C": ["small", "large", "large", "small",
                         "small", "large", "small", "small",
                         "large"],
                   "D": [1, 2, 2, 3, 3, 4, 5, 6, 7]})
```

	A	B	C	D
0	foo	one	small	1
1	foo	one	large	2
2	foo	one	large	2
3	foo	two	small	3
4	foo	two	small	3
5	bar	one	large	4
6	bar	one	small	5
7	bar	two	small	6
8	bar	two	large	7

图 5-5　创建 DataFrame 对象

输出结果如图 5-5 所示。

In[]：

```
table1 = pd.pivot_table(df, values='D', index=['A', 'B'],columns=['C'],
    aggfunc=np.sum)
table1
```

Out[]：

```
  C      large  small
A    B
bar  one  4.0    5.0
     two  7.0    6.0
foo  one  4.0    1.0
     two  NaN    6.0
```

In[]：

```
table2 = pd.pivot_table(df, values='D', index=['A', 'B'],
              aggfunc=np.sum)
table2
```

Out[]：

```
          D
A     B
bar  one   9
     two  13
foo  one   5
     two   6
```

In[]:

```
table3 = pd.pivot_table(df, values = 'D',columns = ['C'],
                aggfunc = np.sum)
table3
```

Out[]:

```
C    large  small
D       15     18
```

In[]:

```
table4 = pd.pivot_table(df, values = 'D', index = ['A', 'B'],columns = ['C'],
aggfunc = np.sum,margins = True,margins_name = 'total')
          table4
```

Out[]:

```
  C       large  small  total
A    B
bar  one   4.0     5.0     9
     two   7.0     6.0    13
foo  one   4.0     1.0     5
     two   NaN     6.0     6
total     15.0    18.0    33
```

7. 保存结果

```
data.to_csv('cleanfile.csv',encoding = 'utf-8')
```

5.3 应用举例

下面按照数据挖掘中数据处理的步骤,通过一个实际案例——泰坦尼克数据集分析,讲解 DataFrame 的用法。

数据集各字段意义如下。

PassengerId:乘客编号。

Survived:存活情况(1 表示存活,0 表示死亡)。

Pclass:客舱等级。

Name:乘客姓名。

Sex:性别。

Age:年龄。

SibSp:同乘的兄弟姐妹/配偶数。

Parch:同乘的父母/小孩数。

Ticket:船票编号。

Fare:船票价格。

Cabin:客舱号。

Embarked：登船港口（出发地点为S表示英国南安普敦；途经地点为C表示法国瑟堡市，Q表示爱尔兰昆士敦）。

5.3.1 数据读取

读取CSV文件格式如下。

```
pd.read_csv(filepath,encoding,sep,header,names,usecols,index_col,skiprows,nrows…)
```

参数说明如下：

filepath：文件存储路径，可以用r""进行非转义限定，路径最好是纯英文（文件名也是），不然会经常遇到编码错误的问题，最方便的做法是直接将文件存储在Pandas默认的路径下，直接输入文件名即可。

encoding：Pandas默认编码是UTF-8，如果同样读取默认UFT-8的TXT或者JSON格式文件，则可以忽略这个参数，如果是CSV文件，且数据中有中文时，则要指定encoding='gbk'。

sep：指定分隔符形式，CSV文件默认用逗号分隔，可以忽略这个参数，如果是其他分隔方式，则要填写。

header：指定第一行是否是列名。通常有三种用法：忽略或header=0（表示数据第一行为列名）以及header=None（表明数据没有列名），常与names搭配使用。

names：指定列名，通常用一个字符串列表表示，当header=0时，用names可以替换掉数据中的第一行作为列名；如果header=None，用names可以增加一行作为列名；如果没有header参数，则用names会增加一行作为列名，原数据的第一行仍然保留。

usecols：一个字符串列表，可以指定读取的列名。

index_col：一个字符串列表，指定哪几列作为索引。

skiprows：跳过多少行再读取数据，通常数据不太干净，需要去除掉表头才会用到。

nrows：仅读取多少行，后面的处理也都仅限于读取的这些行。

In[]：

```
import pandas as pd
import numpy as np
df = pd.read_csv("Titanic.csv")
df.shape
```

Out[]：

```
(891, 12)
```

In[]：

```
df.head()  # 显示前5行
```

输出结果如图5-6所示。

In[]：

```
df.dtypes  # 查看数据类型
```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S

图 5-6 显示前 5 行数据

Out[]：

```
PassengerId      int64
Survived         int64
Pclass           int64
Name            object
Sex             object
Age            float64
SibSp            int64
Parch            int64
Ticket          object
Fare           float64
Cabin           object
Embarked        object
dtype: object
```

Pandas 能够探测到数值类型，因此已经有一些数据存为整数值。当它检测到双精度值时，会自动将其转换为 float 类型。这里有两个更加实用的命令。

In[]：

```
df.info()  # 查看数据信息
```

Out[]：

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 12 columns):
PassengerId     891 non-null int64
Survived        891 non-null int64
Pclass          891 non-null int64
Name            891 non-null object
Sex             891 non-null object
Age             714 non-null float64
SibSp           891 non-null int64
Parch           891 non-null int64
Ticket          891 non-null object
Fare            891 non-null float64
Cabin           204 non-null object
Embarked        889 non-null object
```

```
dtypes: float64(2), int64(5), object(5)
memory usage: 83.6 + KB
```

可以直观地知道这里有 891 项(行)，大部分变量都是完整的(891 为 non-null)。但实际上 Age、Cabin、Embarked 列在某处都有空值。

In[]：

```
df.describe()  # 查看数值数据的详细信息
```

输出结果如图 5-7 所示。

	PassengerId	Survived	Pclass	Age	SibSp	Parch	Fare
count	891.000000	891.000000	891.000000	714.000000	891.000000	891.000000	891.000000
mean	446.000000	0.383838	2.308642	29.699118	0.523008	0.381594	32.204208
std	257.353842	0.486592	0.836071	14.526497	1.102743	0.806057	49.693429
min	1.000000	0.000000	1.000000	0.420000	0.000000	0.000000	0.000000
25%	223.500000	0.000000	2.000000	20.125000	0.000000	0.000000	7.910400
50%	446.000000	0.000000	3.000000	28.000000	0.000000	0.000000	14.454200
75%	668.500000	1.000000	3.000000	38.000000	1.000000	0.000000	31.000000
max	891.000000	1.000000	3.000000	80.000000	8.000000	6.000000	512.329200

图 5-7　数值数据的详细信息 2

Pandas 列出了所有数值列，而且快速地算出了均值、标准差、最小值和最大值。非常方便。但要注意的是，在 Age 中有很多缺失值，Pandas 是怎么处理它们的呢？它会遗留下很多空值。

从数据中可以看出，年龄的最大值是 80。如果想观察数据中的特定子集，如 Pclass 和 Age 列，Pandas 使用 a[list]的形式即可。

In[]：

```
df[['Sex','Pclass','Age']]
```

Out[]：

```
   Sex  Pclass Age
 0 male  3     22.0
1 female  1    38.0
2 female  3    26.0
3 female  1    35.0
4 male    3    35.0
5 male    3    NaN
...
...
[891 rows x 3 columns]
```

同样可以通过设置条件来过滤数据。

In[]：

```
df[df['Age']> 60].loc[0:100,]  # 前 100 行中年龄大于 60 岁的记录
```

输出结果如图 5-8 所示。

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
33	34	0	2	Wheadon, Mr. Edward H	male	66.0	0	0	C.A. 24579	10.5000	NaN	S
54	55	0	1	Ostby, Mr. Engelhart Cornelius	male	65.0	0	1	113509	61.9792	B30	C
96	97	0	1	Goldschmidt, Mr. George B	male	71.0	0	0	PC 17754	34.6542	A5	C

图 5-8 前 100 行中年龄大于 60 岁的记录

5.3.2 数据清洗

数据清洗主要分为如下三步。

(1) 重复值处理——删除。

(2) 缺失值处理——删除,填充(均值、众数、中位数、前后相邻值)和插值(拉格朗日插值、牛顿插值)。

(3) 异常值处理——进行描述性分析+散点图+箱型图定位异常值,处理方法为删除,将其视为缺失值。

1. 重复值处理

In[]:

```
df.duplicated().value_counts()
```

Out[]:

```
False    891
dtype: int64
```

可以看出,数据集没有重复值。如果 DataFrame 中存在重复的行或者几行中某几列的值重复,这时候需要去掉重复行。示例如下:

```
data.drop_duplicates(subset = ['A','B'],keep = 'first',inplace = True)
```

代码中 subset 对应的值是列名,表示只考虑 A 和 B 两列,将这两列对应值相同的行进行去重。默认值为 subset=None,表示考虑所有列。keep='first'表示保留第一次出现的重复行,是默认值。keep 另外两个取值为'last'和 False,分别表示保留最后一次出现的重复行和去除所有重复行。

inplace=True 表示直接在原来的 DataFrame 上删除重复项,而默认值 False 表示生成一个副本。

2. 缺失值处理

缺失值查找:先通过 isnull()函数看一下是否有空值,结果是有空值的地方显示为 True,没有空值的地方显示为 False;再通过 isnull().any()直接看每一列是否有空值,只要这一列有一个空值,结果就为 True;如果想具体看哪几行有空值,可以再用 df.isnull().values==True 来定位。

In[]:

```
df.isnull().any()
```

Out[]：

```
PassengerId     False
Survived        False
Pclass          False
Name            False
Sex             False
Age             True
SibSp           False
Parch           False
Ticket          False
Fare            False
Cabin           True
Embarked        True
dtype: bool
```

可以看出，Age、Cabin、Embarked 列都有空值。

也可以用另外一种方法查看缺失值情况。

In[]：

```
import missingno
missingno.matrix(df,figsize = (30,5))
```

输出结果如图 5-9 所示。

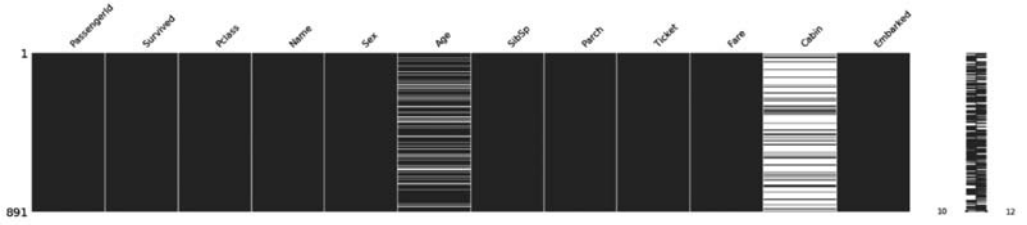

图 5-9　通过 missingno 查看缺失值情况

下面对缺失值进行处理。

(1) 删除缺失值。

```
dropna(axis,subset,how,thresh,inplace)
```

参数说明如下。

axis：删除行或列，行为 0 或 index，列为 1 或 column，默认为行。

subset：删除某几列的缺失值，可选，默认为所有列。

how：any 或 all。any 表明只要出现一个缺失值就删除，all 表示所有列均为缺失值才删除。

thresh：缺失值的数量标准，达到这个阈值才会删除。

inplace：是否直接在原文件中修改。

In[]：

```
df.dropna(how = 'any', axis = 0,inplace = True)
```

```
df.isnull().any()
```

Out[]：

```
PassengerId    False
Survived       False
Pclass         False
Name           False
Sex            False
Age            False
SibSp          False
Parch          False
Ticket         False
Fare           False
Cabin          False
Embarked       False
dtype: bool
```

(2) 缺失值填充。

```
fillna(value,method,{},limit,inplace,axis)
```

参数说明如下。

value：可以传入一个字符串或数字替代 NaN，值可以是指定的或者平均值、众数或中位数等。

method：有 ffill(用前一个填充)和 bfill(用后一个填充)两种。

{}：可以根据不同的列填充不同的值，列为键，填充值为值。

limit：限定填充的数量。

inplace：是否直接在原文件中修改。

axis：填充的方向，默认为 0，按行填充。

接下来对 Sex 分组，用各组乘客的平均年龄填充各组中的缺失年龄，使用 Embarked 变量的众数填充缺失值。

In[]：

```
fillna_Titanic = []
    for i in df.Sex.unique():
     update = df.loc[df.Sex == i,]
     update.fillna(value = {'Age': df.Age[df.Sex == i].mean()}, inplace = True)
     fillna_Titanic.append(update)
    df = pd.concat(fillna_Titanic)
    df.fillna(value = {'Embarked':df.Embarked.mode()[0]}, inplace = True)
    df.isnull().sum() # 查看各列是否有空值
```

Out[]：

```
PassengerId      0
Survived         0
Pclass           0
Name             0
Sex              0
```

```
Age              0
SibSp            0
Parch            0
Ticket           0
Fare             0
Cabin          687
Embarked         0
dtype: int64
```

Cabin 客舱号有缺失，PassengerId、Ticket、Cabin 与是否获救无关，可以删除。

In[]：

```
data.drop(['PassengerId', 'Ticket','Cabin'],axis = 1,inplace = True)
```

5.3.3 数据规整

数据规整是在数据清洗完毕后，将其调整为适合分析的结构，为后续的深入分析做准备，主要分为以下几类。

索引和列名调整：设定新索引，筛选想要的列，更改列名。

数据排序：根据索引或列进行排序。

数据格式调整：更改数据类型，更改数据内容（去除空格标点符号/截取/替换/统一数据单位等），增加用于分析的辅助列。

数据拼接：行堆叠和列拼接。

数据透视：行或列维度转换。

1. 将性别转换为数值，便于后续进行分析

In[]：

```
df.loc[df['Sex'] == 'male','Sex'] = 0
df.loc[df['Sex'] == 'female','Sex'] = 1
```

或者：

```
df['Sex'] = df['Sex'].map( {'female': 1, 'male': 0} ).astype(int)
```

2. 将登录港口转换为数值，便于后续分析

In[]：

```
print(df.Embarked.unique())
df.loc[df['Embarked'] == 'S','Embarked'] = 0
df.loc[df['Embarked'] == 'C','Embarked'] = 1
df.loc[df['Embarked'] == 'Q','Embarked'] = 2
```

Out[]：

```
[0,2,1]
```

In[]：

```
df.describe()
```

输出结果如图 5-10 所示。

	Survived	Pclass	Sex	Age	SibSp	Parch	Fare	Embarked
count	891.000000	891.000000	891.000000	891.000000	891.000000	891.000000	891.000000	891.000000
mean	0.383838	2.308642	0.352413	29.736034	0.523008	0.381594	32.204208	0.361392
std	0.486592	0.836071	0.477990	13.014897	1.102743	0.806057	49.693429	0.635673
min	0.000000	1.000000	0.000000	0.420000	0.000000	0.000000	0.000000	0.000000
25%	0.000000	2.000000	0.000000	22.000000	0.000000	0.000000	7.910400	0.000000
50%	0.000000	3.000000	0.000000	30.000000	0.000000	0.000000	14.454200	0.000000
75%	1.000000	3.000000	1.000000	35.000000	1.000000	0.000000	31.000000	1.000000
max	1.000000	3.000000	1.000000	80.000000	8.000000	6.000000	512.329200	2.000000

图 5-10 数值数据的详细信息 3

3. 建立透视表

(1) 查看不同性别的存活率。

In[]:

```
table = pd.pivot_table(df, index = ["Sex"], values = "Survived")
print(table)
```

Out[]:

```
        Survived
Sex
0       0.188908
1       0.742038
```

可见女性的存活率很高。

参数说明如下。

df：要传入的数据。

index：values to group by in the rows，也就是透视表建立时要根据哪些字段进行分组，这里只依据性别分组。

values：对哪些字段进行聚合操作，因为这里只关心不同性别下的存活率情况，所以 values 只需要传入一个值"survived"。

将所有乘客按性别分为男、女两组后，对"survived"字段开始进行聚合，默认的聚合函数是"mean"，也就是求每个性别组下所有成员的"survived"的均值，即可分别求出男、女两组各自的平均存活率。

(2) 添加列索引：pclass 为客舱级别，共有 1、2、3 三个级别，1 级别最高。

In[]:

```
table = pd.pivot_table(df, index = ["Sex"], values = "Survived")
print(table)
```

Out[]:

```
Pclass      1          2          3
```

```
Sex
0   0.368852   0.157407   0.135447
1   0.968085   0.921053   0.500000
```

可以发现，无论是男性还是女性，客舱级别越高，存活率越高；在各个舱位中，女性的生还概率都远大于男性；一、二等舱的女性生还率接近，且远大于三等舱；一等舱的男性生还率大于二、三等舱，二、三等舱的男性生还率接近。

（3）多级行索引：添加一个行级分组索引。

In[]：

```
table = pd.pivot_table(df, index = ["Sex","Pclass"], values = "Survived")
print(table)
```

Out[]：

```
            Survived
Sex Pclass
0   1       0.368852
    2       0.157407
    3       0.135447
1   1       0.968085
    2       0.921053
    3       0.500000
```

添加一个行级索引"Pclass"后，现在透视表具有二层行级索引和一层列级索引。仔细观察透视表发现，与上面的“添加一个列级索引”在分组聚合效果上是一样的，都是将每个性别组中的成员再次按照客票级别划分为 3 个小组。

（4）将年龄以 18 岁为界，生成一个和年龄有关的透视表。

In[]：

```
def generate_age_label(row):
    age = row["Age"]
    if age < 18:
    return "minor"
    else:
    return "adult"
age_labels = df.apply(generate_age_label, axis = 1)
df['age_labels'] = age_labels
table = df.pivot_table(index = "age_labels", values = "Survived")
print(table)
```

Out[]：

```
            Survived
age_labels
adult       0.361183
minor       0.539823
```

可以发现，未成年组存活率较高。

(5) 性别(Sex)、客舱级别(Pclass)、登船港口(Embarked)与生还率关系。

In[]:

```
df.pivot_table(values = "Survived",index = "Sex",columns = ["Pclass","Embarked"],
aggfunc = np.mean)
```

输出结果如图 5-11 所示。

Pclass	1			2			3		
Embarked	0	1	2	0	1	2	0	1	2
Sex									
0	0.35443	0.404762	0.0	0.154639	0.2	0.0	0.128302	0.232558	0.076923
1	0.96000	0.976744	1.0	0.910448	1.0	1.0	0.375000	0.652174	0.727273

图 5-11 统计结果

可以看出,Sex=0 且 Embarked=1 时,即男性在途经地点 1: C=法国瑟堡市(Cherbourg)登船时,在各个客舱级别中的存活率相对较高。

习 题

探索鸢尾花数据。

(1) 将数据集保存为变量 iris。

(2) 创建数据框的列名称['sepal_length','sepal_width', 'petal_length', 'petal_width', 'class']。

(3) 数据框中有缺失值吗?

(4) 将 petal_length 列的第 10～19 行设置为缺失值。

(5) 将 petal_length 列的缺失值全部替换为 1.0。

(6) 删除 class 列。

(7) 将数据框前 3 行设置为缺失值。

(8) 删除有缺失值的行。

(9) 重新设置索引。