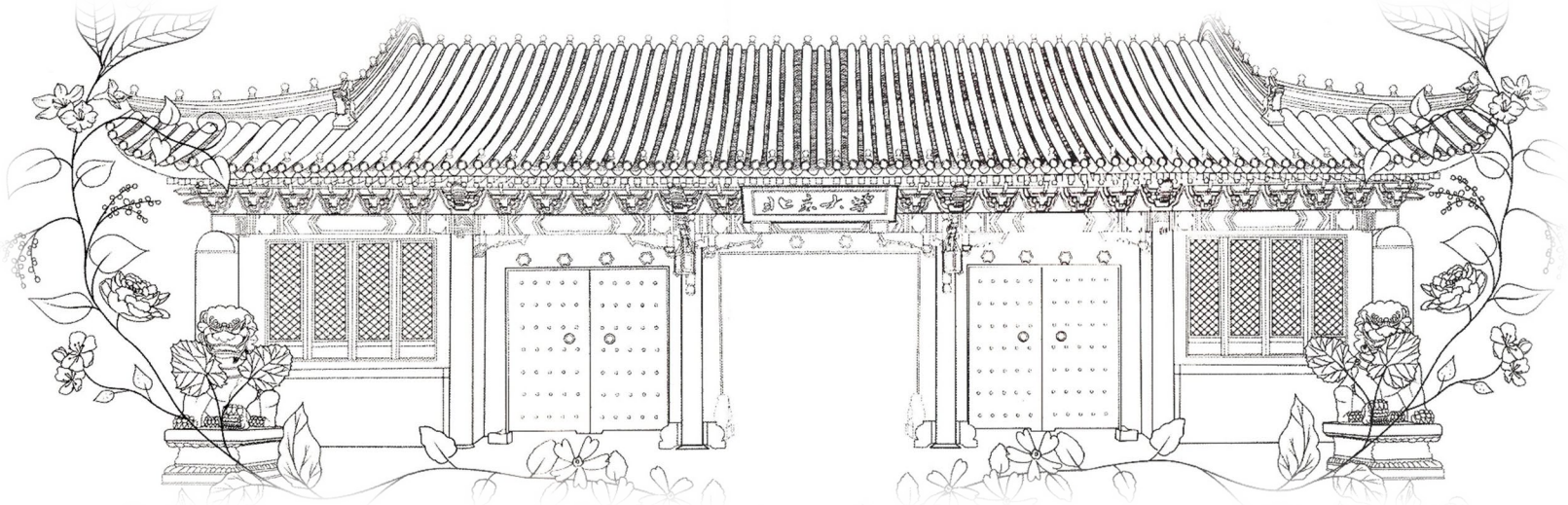
第 1 章 Python数据分析基础 III

北京大学信息管理系

2021/3/22



本节内容

01 数据读写

02 数据清洗

03 数据聚合与分组

本节内容

01 数据读写

02 数据清洗

03 数据聚合与分组

数据读写

- 在进行实际的数据分析任务时，**读取数据**往往是第一步；在进行数据分析时，**数据导出和存储**也是很重要的内容。
- Python内置有文件读写的功能。
- 利用**write**写入文件：

```
f = open('test_file.txt', 'w')
for i in range(5):
    f.write('hello myfile, this is line%d!' %i)
f.close()
```

数据读写

- 写入结果如下：

test_file.txt

```
1 Hello myfile, this is line 0!
2 Hello myfile, this is line 1!
3 Hello myfile, this is line 2!
4 Hello myfile, this is line 3!
5 Hello myfile, this is line 4!
6
```

数据读写

- 如果需要在文件尾部追加内容，可以利用open函数的a参数：

```
f = open('test_file.txt', 'a')
for i in range(3):
    f.write('Hello again, this is additional line %d!\n' %i)
f.close()
```

数据读写

- 写入结果如下：

test_file.txt

```
1 Hello myfile, this is line 0!
2 Hello myfile, this is line 1!
3 Hello myfile, this is line 2!
4 Hello myfile, this is line 3!
5 Hello myfile, this is line 4!
6 Hello again, this is additional line 0!
7 Hello again, this is additional line 1!
8 Hello again, this is additional line 2!
9
```

数据读写

- Python内置的文件读取功能支持简单顺序读取(read)、每次读取一行(readline)和按行读取所有内容(readlines)。
- **read**

```
f = open('test_file.txt', 'r')
content = f.read(20)
print(content)
print('+'*40)
content_2 = f.read()
print(content_2)
f.close()
```

```
Hello myfile, this i
++++++++++++++++++++++++++++++++++++++++
s line 0!
Hello myfile, this is line 1!
Hello myfile, this is line 2!
Hello myfile, this is line 3!
Hello myfile, this is line 4!
Hello again, this is additional line 0!
Hello again, this is additional line 1!
Hello again, this is additional line 2!
```

数据读写

- **readlines**

```
f = open('test_file.txt', 'r')
content = f.readlines()
i = 0
for line in content:
    print('%d:%s'%(i, line), end='')
    i+=1
f.close()
```

```
0:Hello myfile, this is line 0!
1:Hello myfile, this is line 1!
2:Hello myfile, this is line 2!
3:Hello myfile, this is line 3!
4:Hello myfile, this is line 4!
5:Hello again, this is additional line 0!
6:Hello again, this is additional line 1!
7:Hello again, this is additional line 2!
```

数据读写

- **readline**

```
f = open('test_file.txt', 'r')
content = f.readline()
print("1:%s"%content)
content = f.readline()
print("2:%s"%content)
f.close()
```

```
1:Hello myfile, this is line 0!

2:Hello myfile, this is line 1!
```

Pandas数据读取

- Pandas提供了丰富的数据读取接口，可供我们选择：

Function	Description
`read_csv`	Load delimited data from a file, URL, or file-like object; use comma as default delimiter
`read_table`	Load delimited data from a file, URL, or file-like object; use tab (`'\t'`) as default delimiter
`read_fwf`	Read data in fixed-width column format (i.e., no delimiters)
`read_clipboard`	Version of `read_table` that reads data from the clipboard; useful for converting tables from web pages
`read_excel`	Read tabular data from an Excel XLS or XLSX file
`read_hdf`	Read HDF5 files written by pandas
`read_html`	Read all tables found in the given HTML document
`read_json`	Read data from a JSON (JavaScript Object Notation) string representation
`read_msgpack`	Read pandas data encoded using the MessagePack binary format
`read_pickle`	Read an arbitrary object stored in Python pickle format

Pandas数据读取

- Pandas提供了丰富的数据读取接口，可供我们选择：

Function	Description
read_sas	Read a SAS dataset stored in one of the SAS system's custom storage formats
read_sql	Read the results of a SQL query (using SQLAlchemy) as a pandas DataFrame
read_stata	Read a dataset from Stata file format
read_feather	Read the Feather binary file format

Pandas数据读取

➢Pandas提供的这些函数具有多种选项，以**read_csv**为例：

pandas.read_csv

pandas.read_csv(*filepath_or_buffer, sep=<object object>, delimiter=None, header='infer', names=None, index_col=None, usecols=None, squeeze=False, prefix=None, mangle_dupe_cols=True, dtype=None, engine=None, converters=None, true_values=None, false_values=None, skipinitialspace=False, skiprows=None, skipfooter=0, nrows=None, na_values=None, keep_default_na=True, na_filter=True, verbose=False, skip_blank_lines=True, parse_dates=False, infer_datetime_format=False, keep_date_col=False, date_parser=None, dayfirst=False, cache_dates=True, iterator=False, chunksize=None, compression='infer', thousands=None, decimal='.', lineterminator=None, quotechar='"', quoting=0, doublequote=True, escapechar=None, comment=None, encoding=None, dialect=None, error_bad_lines=True, warn_bad_lines=True, delim_whitespace=False, low_memory=True, memory_map=False, float_precision=None, storage_options=None*) [source]

Read a comma-separated values (csv) file into DataFrame.

Also supports optionally iterating or breaking of the file into chunks.

Additional help can be found in the online docs for IO Tools.

Pandas数据读取

- 这些选项可以划分为以下几个大类：
- **索引：**将一个或多个列当做返回的DataFrame处理，以及是否从文件、用户获取列名。
- **类型推断和数据转换：**包括用户定义值的转换、和自定义的缺失值标记列表等。
- **日期解析：**包括组合功能，比如将分散在多个列中的日期时间信息组合成结果中的单个列。
- **迭代：**支持对大文件进行逐块迭代。
- **不规整数据问题：**跳过一些行、页脚、注释或其他一些不重要的东西（比如由成千上万个逗号隔开的数值数据）。

Pandas数据读取

- 必要时，可以先利用**!cat**（Linux或Mac系统）或**!type**（Windows系统）命令，查看文件内容：

- Windows系统

```
In [6]: !type test_file.txt
Hello myfile, this is line 0!
Hello myfile, this is line 1!
Hello myfile, this is line 2!
Hello myfile, this is line 3!
Hello myfile, this is line 4!
Hello again, this is additional line 0!
Hello again, this is additional line 1!
Hello again, this is additional line 2!
```

```
In [8]: !type examples\ex1.csv
a, b, c, d, message
1, 2, 3, 4, hello
5, 6, 7, 8, world
9, 10, 11, 12, foo
```

- Linux或Mac系统

```
In [6]: !cat test_file.txt
Hello myfile, this is line 0!
Hello myfile, this is line 1!
Hello myfile, this is line 2!
Hello myfile, this is line 3!
Hello myfile, this is line 4!
Hello again, this is additional line 0!
Hello again, this is additional line 1!
Hello again, this is additional line 2!
```

```
In [8]: !cat examples/ex1.csv
a, b, c, d, message
1, 2, 3, 4, hello
5, 6, 7, 8, world
9, 10, 11, 12, foo
```

注意不同系统路径分隔符的不同！

Pandas数据读取

- 可以发现，文件是以逗号“，”作为字段分隔符，因而可以直接调用pandas.read_csv进行读取成为DataFrame：

```
In [9]: df = pd.read_csv('examples/ex1.csv')

In [10]: df
Out[10]:
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo
```

- 同样，可以调用pandas.read_table，并指定分隔符为“，”：

```
In [11]: pd.read_table('examples/ex1.csv', sep=',')
Out[11]:
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo
```

- 默认情况下，会将第一行解析为列索引，生成RangeIndex作为行索引。

Pandas数据读取

- 面对没有标题行的数据时，需要利用**header**参数声明没有列名。

```
In [12]: !type examples\ex2.csv
1,2,3,4,hello
5,6,7,8,world
9,10,11,12,foo
```

```
In [13]: pd.read_csv('examples/ex2.csv', header=None)
Out[13]:
   0   1   2   3      4
0  1   2   3   4  hello
1  5   6   7   8  world
2  9  10  11  12    foo
```

- 这种情况，也可以利用names定义列名：

```
In [14]: pd.read_csv('examples/ex2.csv', names=['a', 'b', 'c', 'd', 'message'])
Out[14]:
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo
```

Pandas数据读取

- 如果想指定某些列作为行名，可以使用**index_col**参数进行指定。

```
In [15]: names = ['a', 'b', 'c', 'd', 'message']

In [16]: pd.read_csv('examples/ex2.csv', names=names, index_col='message')
Out[16]:
         a   b   c   d
message
hello    1   2   3   4
world    5   6   7   8
foo      9  10  11  12
```

- 这样操作，等价于读入数据后，再调用**set_index**。

```
In [16]: names = ['a', 'b', 'c', 'd', 'message']

In [17]: pd.read_csv('examples/ex2.csv', names=names).set_index('message')
```

Out[17]:

	a	b	c	d
message				
hello	1	2	3	4
world	5	6	7	8
foo	9	10	11	12

Pandas数据读取

- 如果希望将多个列做成一个层次化索引，只需传入由列编号或列名组成的列表即可：

```
In [17]: !cat examples/csv_mindex.csv
key1,key2,value1,value2
one,a,1,2
one,b,3,4
one,c,5,6
one,d,7,8
two,a,9,10
two,b,11,12
two,c,13,14
two,d,15,16
```

```
In [18]: parsed = pd.read_csv('examples/csv_mindex.csv',
    ....:                     index_col=['key1', 'key2'])

In [19]: parsed
Out[19]:
           value1  value2
key1 key2
one  a          1       2
     b          3       4
     c          5       6
     d          7       8
two  a          9      10
     b         11      12
     c         13      14
     d         15      16
```

Pandas数据读取

- 对于没有使用固定的分隔符去分隔字段的数据，需要考虑用正则表达式定义分隔符参数**sep**。

```
In [20]: list(open('examples/ex3.txt'))
Out[20]:
['            A         B         C\n',
 'aaa -0.264438 -1.026059 -0.619500\n',
 'bbb  0.927272  0.302904 -0.032399\n',
 'ccc -0.264273 -0.386314 -0.217601\n',
 'ddd -0.871858 -0.348382  1.100491\n']

In [21]: result = pd.read_table('examples/ex3.txt', sep='\s+')

In [22]: result
Out[22]:
            A         B         C
aaa -0.264438 -1.026059 -0.619500
bbb  0.927272  0.302904 -0.032399
ccc -0.264273 -0.386314 -0.217601
ddd -0.871858 -0.348382  1.100491
```

由于列名比数据行的数量少，所以read_table推断第一列应该是DataFrame的索引。

如果少多列呢？

Pandas数据读取

- 可以利用**skiprows**参数指定跳过的行。

```
In [23]: !cat examples/ex4.csv
# hey!
a,b,c,d,message
# just wanted to make things more difficult for you
# who reads CSV files with computers, anyway?
1,2,3,4,hello
5,6,7,8,world
9,10,11,12,foo
In [24]: pd.read_csv('examples/ex4.csv', skiprows=[0, 2, 3])
Out[24]:
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo
```

Pandas数据读取

- 对于有缺失的数据，pandas会自动在缺失位置填上**NaN**，常用的缺失值和空值符号（如NA和Null），pandas都会自动识别判断。

```
In [25]: !cat examples/ex5.csv
something,a,b,c,d,message
one,1,2,3,4,NA
two,5,6,,8,world
three,9,10,11,12,foo
In [26]: result = pd.read_csv('examples/ex5.csv')
In [27]: result
Out[27]:
  something  a   b     c   d message
0       one  1   2   3.0   4     NaN
1       two  5   6   NaN   8   world
2     three  9  10  11.0  12     foo

In [28]: pd.isnull(result)
Out[28]:
   something      a      b      c      d  message
0      False  False  False  False  False     True
1      False  False  False   True  False    False
2      False  False  False  False  False    False
```

Pandas数据读取

- 如果遇到不常用或自定义的缺失值标记形式，可以对na_values参数传递**列表（list）**或者**集合（set）**进行定义。

```
In [29]: result = pd.read_csv('examples/ex5.csv', na_values=['NULL'])

In [30]: result
Out[30]:
  something  a   b     c   d message
0       one  1   2   3.0   4     NaN
1       two  5   6   NaN   8   world
2     three  9  10  11.0  12     foo
```

- 如果每列具有不同的缺失值标记形式，可以传递以列名为键的**字典**。

```
In [31]: sentinels = {'message': ['foo', 'NA'], 'something': ['two']}

In [32]: pd.read_csv('examples/ex5.csv', na_values=sentinels)
Out[32]:
  something  a   b     c   d message
0       one  1   2   3.0   4     NaN
1       NaN  5   6   NaN   8   world
2     three  9  10  11.0  12     NaN
```

Pandas数据读取

- 对于有缺失的数据，pandas会自动在确实位置填上**NaN**，常用的缺失值和空值符号（如NA和Null），pandas都会自动识别判断。

```
In [25]: !cat examples/ex5.csv
something,a,b,c,d,message
one,1,2,3,4,NA
two,5,6,,8,world
three,9,10,11,12,foo
In [26]: result = pd.read_csv('examples/ex5.csv')
In [27]: result
Out[27]:
  something  a   b     c   d message
0       one  1   2   3.0   4     NaN
1       two  5   6   NaN   8   world
2     three  9  10  11.0  12     foo

In [28]: pd.isnull(result)
Out[28]:
   something      a      b      c      d  message
0      False  False  False  False  False     True
1      False  False  False   True  False    False
2      False  False  False  False  False    False
```

Pandas数据读取

Table 6-2. Some read_csv/read_table function arguments

Argument	Description
`path`	String indicating filesystem location, URL, or file-like object
`sep` or `delimiter`	Character sequence or regular expression to use to split fields in each row
`header`	Row number to use as column names; defaults to 0 (first row), but should be `None` if there is no header row
`index_col`	Column numbers or names to use as the row index in the result; can be a single name/number or a list of them for a hierarchical index
`names`	List of column names for result, combine with `header=None`
`skiprows`	Number of rows at beginning of file to ignore or list of row numbers (starting from 0) to skip.
`na_values`	Sequence of values to replace with NA.
`comment`	Character(s) to split comments off the end of lines.
`parse_dates`	Attempt to parse data to `datetime`; `False` by default. If `True`, will attempt to parse all columns. Otherwise can specify a list of column numbers or name to parse. If element of list is tuple or list, will combine multiple columns together and parse to date (e.g., if date/time split across two columns).
`keep_date_col`	If joining columns to parse date, keep the joined columns; `False` by default.
`converters`	Dict containing column number of name mapping to functions (e.g., `{'foo': f}` would apply the function `f` to all values in the `'foo'` column).

Pandas数据读取

Argument	Description
converters	Dict containing column number of name mapping to functions (e.g., {'foo': f} would apply the function f to all values in the 'foo' column).
dayfirst	When parsing potentially ambiguous dates, treat as international format (e.g., 7/6/2012 -> June 7, 2012); False by default.
date_parser	Function to use to parse dates.
nrows	Number of rows to read from beginning of file.
iterator	Return a TextParser object for reading file piecemeal.
chunksize	For iteration, size of file chunks.
skip_footer	Number of lines to ignore at end of file.
verbose	Print various parser output information, like the number of missing values placed in non-numeric columns.
encoding	Text encoding for Unicode (e.g., 'utf-8' for UTF-8 encoded text).
squeeze	If the parsed data only contains one column, return a Series.
thousands	Separator for thousands (e.g., ',' or '.').

Pandas数据读取

➢ 如果需要读取一部分数据，可以通过**nrows**参数指定读取的行数。

```
In [34]: result = pd.read_csv('examples/ex6.csv')

In [35]: result
Out[35]:
           one       two     three      four key
0     0.467976 -0.038649 -0.295344 -1.824726   L
1    -0.358893  1.404453  0.704965 -0.200638   B
2    -0.501840  0.659254 -0.421691 -0.057688   G
3     0.204886  1.074134  1.388361 -0.982404   R
4     0.354628 -0.133116  0.283763 -0.837063   Q
...        ...       ...       ...       ...  ..
9995  2.311896 -0.417070 -1.409599 -0.515821   L
9996 -0.479893 -0.650419  0.745152 -0.646038   E
9997  0.523331  0.787112  0.486066  1.093156   K
9998 -0.362559  0.598894 -1.843201  0.887292   G
9999 -0.096376 -1.012999 -0.657431 -0.573315   0
[10000 rows x 5 columns]
```

```
In [36]: pd.read_csv('examples/ex6.csv', nrows=5)
Out[36]:
        one       two     three      four key
0  0.467976 -0.038649 -0.295344 -1.824726   L
1 -0.358893  1.404453  0.704965 -0.200638   B
2 -0.501840  0.659254 -0.421691 -0.057688   G
3  0.204886  1.074134  1.388361 -0.982404   R
4  0.354628 -0.133116  0.283763 -0.837063   Q
```

Pandas数据读取

- 如果要分块读取数据，可以通过**chunksize**参数指定每块的行数。

```
In [37]: chunker = pd.read_csv('examples/ex6.csv', chunksize=1000)

In [38]: chunker
Out[38]: <pandas.io.parsers.TextFileReader at 0x7f6b1e2672e8>
```

- 得到的**文本文件读取对象**，可以通过迭代的方式访问。

```
tot = pd.Series([])
for piece in chunker:
    tot = tot.add(piece['key'].value_counts(), fill_value=0)

tot = tot.sort_values(ascending=False)
```

对每个数据块中每个key的取值个数进行统计；

并通过add方法进行累加，得到全部数据汇总。

```
In [40]: tot[:10]
Out[40]:
E    368.0
X    364.0
L    346.0
O    343.0
Q    340.0
M    338.0
J    337.0
F    335.0
K    334.0
H    330.0
dtype: float64
```

Pandas数据写入文件

- 可以利用**to_csv**直接将数据写成逗号分隔的文本文件：

```
In [41]: data = pd.read_csv('examples/ex5.csv')

In [42]: data
Out[42]:
  something  a   b     c   d message
0       one  1   2   3.0   4     NaN
1       two  5   6   NaN   8   world
2     three  9  10  11.0  12     foo
```

```
In [43]: data.to_csv('examples/out.csv')

In [44]: !cat examples/out.csv
,something,a,b,c,d,message
0,one,1,2,3.0,4,
1,two,5,6,,8,world
2,three,9,10,11.0,12,foo
```

Pandas数据写入文件

- 如果要指定分隔符，同样是利用**sep**参数：

```
In [45]: import sys

In [46]: data.to_csv(sys.stdout, sep='|')
|something|a|b|c|d|message
0|one|1|2|3.0|4|
1|two|5|6||8|world
2|three|9|10|11.0|12|foo
```

将csv写入到sys.stdout代表将结果打印到控制台。

- 有缺失值时，可以利用**na_rep**参数指定缺失值的代表符号：

```
In [47]: data.to_csv(sys.stdout, na_rep='NULL')
,something,a,b,c,d,message
0,one,1,2,3.0,4,NULL
1,two,5,6,NULL,8,world
2,three,9,10,11.0,12,foo
```

Pandas数据写入文件

- 可以利用**index**参数和**header**参数分别指定是否保留行、列索引（默认设定是都保留）：

```
In [48]: data.to_csv(sys.stdout, index=False, header=False)
one,1,2,3.0,4,
two,5,6,,8,world
three,9,10,11.0,12,foo
```

- 如果想按一定顺序保留一部分列，则可将列名的列表传递给**header**参数：

```
In [49]: data.to_csv(sys.stdout, index=False, columns=['a', 'b', 'c'])
a,b,c
1,2,3.0
5,6,
9,10,11.0
```

Pandas数据写入文件

- 对于Series而言，同样具有**to_csv**方法：

```
In [50]: dates = pd.date_range('1/1/2000', periods=7)

In [51]: ts = pd.Series(np.arange(7), index=dates)

In [52]: ts.to_csv('examples/tseries.csv')

In [53]: !cat examples/tseries.csv
2000-01-01,0
2000-01-02,1
2000-01-03,2
2000-01-04,3
2000-01-05,4
2000-01-06,5
2000-01-07,6
```

处理JSON数据

- **JSON**（JavaScript Object Notation的简称）已经成为通过HTTP请求在Web浏览器和其他应用程序之间发送数据的标准格式之一。
- 它比表格型文本格式（如CSV）要**灵活**很多。如果通过API从服务器获取数据，很可能会遇到JSON格式。
- JSON的数据内容以键值对的形式进行定义，其语法与Python代码非常接近，只在空值等一些细微地方存在差别。

```
obj = """
{"name": "Wes",
 "places_lived": ["United States", "Spain", "Germany"],
 "pet": null,
 "siblings": [{"name": "Scott", "age": 30, "pets": ["Zeus", "Zuko"]},
              {"name": "Katie", "age": 38,
               "pets": ["Sixes", "Stache", "Cisco"]}]
}
"""
```

处理JSON数据

- 读取JSON数据可以利用Python标准库**json**。
- 通过**json.loads**即可将JSON字符串转换成Python形式：

```
In [62]: import json

In [63]: result = json.loads(obj)

In [64]: result
Out[64]:
{'name': 'Wes',
 'pet': None,
 'places_lived': ['United States', 'Spain', 'Germany'],
 'siblings': [{'age': 30, 'name': 'Scott', 'pets': ['Zeus', 'Zuko']},
  {'age': 38, 'name': 'Katie', 'pets': ['Sixes', 'Stache', 'Cisco']}]}
```

- 通过**json.dumps**即可将Python对象转换成JSON格式：

```
In [65]: asjson = json.dumps(result)
```

处理JSON数据

- 可以通过向DataFrame构造器传入一个字典的列表（就是原先的JSON对象），并选取数据字段的子集的方式，将JSON对象转化为DataFrame。

```
In [64]: result
Out[64]:
{'name': 'Wes',
 'pet': None,
 'places_lived': ['United States', 'Spain', 'Germany'],
 'siblings': [{'age': 30, 'name': 'Scott', 'pets': ['Zeus', 'Zuko']},
  {'age': 38, 'name': 'Katie', 'pets': ['Sixes', 'Stache', 'Cisco']}]}

In [66]: siblings = pd.DataFrame(result['siblings'], columns=['name', 'age'])

In [67]: siblings
Out[67]:
    name  age
0  Scott   30
1  Katie   38
```

处理JSON数据

- 也可以利用pd.read_json自动将特定格式的JSON数据集转换为Series或DataFrame。

```
In [68]: !cat examples/example.json
[{"a": 1, "b": 2, "c": 3},
 {"a": 4, "b": 5, "c": 6},
 {"a": 7, "b": 8, "c": 9}]

In [69]: data = pd.read_json('examples/example.json')

In [70]: data
Out[70]:
   a  b  c
0  1  2  3
1  4  5  6
2  7  8  9
```

pandas.read_json的默认选项假设JSON数组中的每个对象是表格中的一行。

处理JSON数据

- 可以使用**to_json**方法，将数据从pandas输出到JSON。

```
In [70]: data
Out[70]:
   a  b  c
0  1  2  3
1  4  5  6
2  7  8  9

In [71]: print(data.to_json())
{"a":{"0":1,"1":4,"2":7},"b":{"0":2,"1":5,"2":8},"c":{"0":3,"1":6,"2":9}}

In [72]: print(data.to_json(orient='records'))
[{"a":1,"b":2,"c":3},{"a":4,"b":5,"c":6},{"a":7,"b":8,"c":9}]
```

to_json的默认将每列作为一个元素；
可以指定参数orient='records'，使得以每行作为一个元素（此时，行索引信息会丢失）。

本节内容

01 数据读写

02 数据清洗

03 数据聚合与分组

数据清洗

➢ 数据清洗主要的操作有两种：

1. **缺失值处理**
2. **数据转换**

其中，缺失值处理已经在上一节ppt介绍过；

数据转换中的数据重排已经有过介绍；

本节主要介绍**数据过滤**、**数据清理**、以及其他的转换方式。

去重

- DataFrame的**duplicated**方法返回一个布尔型Series，表示各行是否是重复行。

```
In [45]: data = pd.DataFrame({'k1': ['one', 'two'] * 3 + ['two'],
   ....:                      'k2': [1, 1, 2, 3, 3, 4, 4]})
```

```
In [46]: data
Out[46]:
    k1  k2
0  one   1
1  two   1
2  one   2
3  two   3
4  one   3
5  two   4
6  two   4
```

```
In [47]: data.duplicated()
Out[47]:
0    False
1    False
2    False
3    False
4    False
5    False
6     True
dtype: bool
```

- 可利用**drop_duplicates**方法，剔除重复行。

```
In [48]: data.drop_duplicates()
Out[48]:
    k1  k2
0  one   1
1  two   1
2  one   2
3  two   3
4  one   3
5  two   4
```

去重

```
In [46]: data
Out[46]:
    k1  k2
0  one   1
1  two   1
2  one   2
3  two   3
4  one   3
5  two   4
6  two   4
```

- 以上两个方法默认会对所有列进行比对，如果需要只对比若干列，可以传递**列索引**或**列索引构成的列表**。

```
In [49]: data['v1'] = range(7)

In [50]: data.drop_duplicates(['k1'])
Out[50]:
    k1  k2  v1
0  one   1   0
1  two   1   1

In [51]: data.drop_duplicates(['k1', 'k2'], keep='last')
Out[51]:
    k1  k2  v1
0  one   1   0
1  two   1   1
2  one   2   2
3  two   3   3
4  one   3   4
6  two   4   6
```

duplicated和drop_duplicates默认保留的是第一个出现的值组合。传入keep='last'则保留最后一个

利用函数或映射进行转换

- 在需要根据某些数据对应规则进行转换时，可以利用函数或映射。如下面的肉类数据。

```
In [52]: data = pd.DataFrame({'food': ['bacon', 'pulled pork', 'bacon',
   ....:                               'Pastrami', 'corned beef', 'Bacon',
   ....:                               'pastrami', 'honey ham', 'nova lox'],
   ....:                      'ounces': [4, 3, 12, 6, 7.5, 8, 3, 5, 6]})
```

```
In [53]: data
Out[53]:
          food  ounces
0        bacon     4.0
1  pulled pork     3.0
2        bacon    12.0
3     Pastrami     6.0
4  corned beef     7.5
5        Bacon     8.0
6     pastrami     3.0
7    honey ham     5.0
8     nova lox     6.0
```

- 假设我们需要根据各种肉的来源动物添加一列，我们首先需要一个肉到动物的映射：

```
meat_to_animal = {
  'bacon': 'pig',
  'pulled pork': 'pig',
  'pastrami': 'cow',
  'corned beef': 'cow',
  'honey ham': 'pig',
  'nova lox': 'salmon'
}
```

利用函数或映射进行转换

- 这时可以调用Series的**map**方法，将一个**函数**或**含有映射关系的字典型对象**作为参数传递给**map**方法，即可进行转换：

```
In [55]: lowercased = data['food'].str.lower()

In [56]: lowercased
Out[56]:
0          bacon
1    pulled pork
2          bacon
3       pastrami
4    corned beef
5          bacon
6       pastrami
7      honey ham
8       nova lox
Name: food, dtype: object
```

需要注意的是，Python是对大小写敏感的，为避免意外情况，进行map时最好进行大小写统一：

利用函数或映射进行转换

- 这时可以调用Series的**map**方法，将一个**函数**或**含有映射关系的字典型对象**作为参数传递给**map**方法，即可进行转换：

```
In [53]: data
Out[53]:
          food  ounces
0        bacon     4.0
1  pulled pork     3.0
2        bacon    12.0
3     Pastrami     6.0
4  corned beef     7.5
5        Bacon     8.0
6     pastrami     3.0
7   honey ham      5.0
8    nova lox      6.0
```

需要注意的是，Python是对大小写敏感的，为避免意外情况，map之前最好进行大小写统一：

```
In [55]: lowercased = data['food'].str.lower()

In [56]: lowercased
Out[56]:
0          bacon
1    pulled pork
2          bacon
3       pastrami
4    corned beef
5          bacon
6       pastrami
7     honey ham
8       nova lox
Name: food, dtype: object
```

利用函数或映射进行转换

```
meat_to_animal = {
  'bacon': 'pig',
  'pulled pork': 'pig',
  'pastrami': 'cow',
  'corned beef': 'cow',
  'honey ham': 'pig',
  'nova lox': 'salmon'
}
```

- 这时可以调用Series的**map**方法，将一个**函数**或**含有映射关系的字典型对象**作为参数传递给**map**方法，即可进行转换：

```
In [57]: data['animal'] = lowercased.map(meat_to_animal)

In [58]: data
Out[58]:
          food  ounces  animal
0        bacon     4.0     pig
1  pulled pork     3.0     pig
2        bacon    12.0     pig
3     Pastrami     6.0     cow
4  corned beef     7.5     cow
5        Bacon     8.0     pig
6     pastrami     3.0     cow
7    honey ham     5.0     pig
8     nova lox     6.0  salmon
```

```
In [56]: lowercased
Out[56]:
0          bacon
1    pulled pork
2          bacon
3       pastrami
4    corned beef
5          bacon
6       pastrami
7      honey ham
8       nova lox
Name: food, dtype: object
```

利用函数或映射进行转换

- 也可以考虑给**map**传入一个函数，该函数将作用Series中的每个元素，并返回一个与原Series具有相同index的新Series。

```
In [59]: data['food'].map(lambda x: meat_to_animal[x.lower()])
Out[59]:
0       pig
1       pig
2       pig
3       cow
4       cow
5       pig
6       cow
7       pig
8    salmon
Name: food, dtype: object
```

将每一个元素作为键，映射为转换字典中对应的值。

替换值

- 利用**fillna**方法填充缺失数据可以看做值替换的一种特殊情况。在更一般的情况下，可以调用**replace**完成定制化替换：

```
In [60]: data = pd.Series([1., -999., 2., -999., -1000., 3.])

In [61]: data
Out[61]:
0       1.0
1    -999.0
2       2.0
3    -999.0
4   -1000.0
5       3.0
dtype: float64
```

```
In [62]: data.replace(-999, np.nan)
Out[62]:
0       1.0
1       NaN
2       2.0
3       NaN
4   -1000.0
5       3.0
dtype: float64
```

如要原地修改，则需传入参数inplace=True

替换值

```
In [61]: data
Out[61]:
0       1.0
1    -999.0
2       2.0
3    -999.0
4   -1000.0
5       3.0
dtype: float64
```

- 可以通过传入**列表**或**字典**一次性替换多个值：

```
In [63]: data.replace([-999, -1000], np.nan)
Out[63]:
0    1.0
1    NaN
2    2.0
3    NaN
4    NaN
5    3.0
dtype: float64
```

```
In [64]: data.replace([-999, -1000], [np.nan, 0])
Out[64]:
0    1.0
1    NaN
2    2.0
3    NaN
4    0.0
5    3.0
dtype: float64
```

```
In [65]: data.replace({-999: np.nan, -1000: 0})
Out[65]:
0    1.0
1    NaN
2    2.0
3    NaN
4    0.0
5    3.0
dtype: float64
```


重命名索引

- 跟Series中的值一样，**轴标签**也可以通过**函数**或**映射**进行转换，从而得到一个新的不同标签的对象。轴还可以被就地修改，而无需新建一个数据结构。

```
In [66]: data = pd.DataFrame(np.arange(12).reshape((3, 4)),
   ....:                     index=['Ohio', 'Colorado', 'New York'],
   ....:                     columns=['one', 'two', 'three', 'four'])
```

- 轴索引也有一个map方法，进行转换（重命名）：

```
In [67]: transform = lambda x: x[:4].upper()

In [68]: data.index.map(transform)
Out[68]: Index(['OHIO', 'COLO', 'NEW '], dtype='object')
```

- 可以通过赋值实现原地修改：

```
In [69]: data.index = data.index.map(transform)
```

```
In [70]: data
Out[70]:
      one  two  three  four
OHIO    0    1      2     3
COLO    4    5      6     7
NEW     8    9     10    11
```

替换值

- 可以利用**rename**进行转换，生成新的数据：

```
In [71]: data.rename(index=str.title, columns=str.upper)
Out[71]:
       ONE  TWO  THREE  FOUR
Ohio     0    1      2     3
Colo     4    5      6     7
New      8    9     10    11
```

- 通过传递给**index**和**columns**参数传递字典，实现部分标签的修改：

```
In [72]: data.rename(index={'OHIO': 'INDIANA'},
   ....:             columns={'three': 'peekaboo'})
Out[72]:
          one  two  peekaboo  four
INDIANA     0    1         2     3
COLO        4    5         6     7
NEW         8    9        10    11
```

- 可以利用通过参数**inplace=True**进行原地修改：

```
In [73]: data.rename(index={'OHIO': 'INDIANA'}, inplace=True)
```

本节内容

01 数据读写

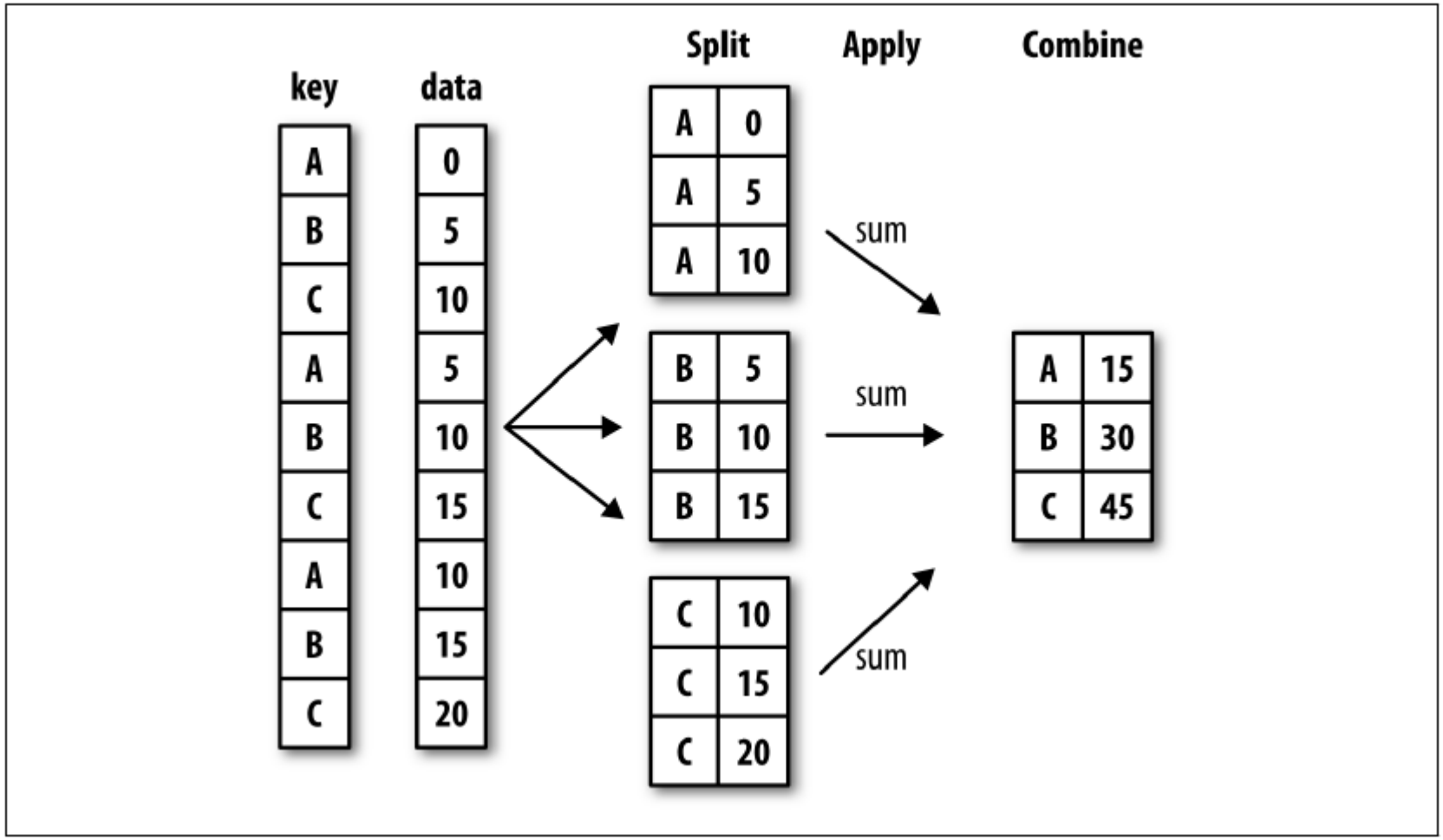
02 数据清洗

03 数据聚合与分组

数据聚合与分组

- 在进行数据分析时，往往需要对数据集**进行分组**并**对各组应用一个函数**（无论是聚合还是转换）。在将数据集加载、融合、准备好之后，通常就是计算**分组统计**或**生成透视表**。
- Pandas提供了一个灵活高效的**groupby**功能，便于以一种自然的方式对数据集进行切片、切块、摘要等操作。
- 具体地，通过pandas的内置功能，可以实现：
  1. 使用一个或多个键（形式可以是函数、数组或DataFrame列名）分割pandas对象。
  2. **计算分组的统计量**，比如数量、平均值或标准差，或自定义函数。
  3. **应用组内转换或其他运算**，如规范化、回归、排名或选取子集等。
  4. 计算**透视表**或**交叉表**。
  5. 执行**分位数分析**以及其它统计分组分析。

GroupBy机制


key
data
Split
Apply
Combine
A
B
C
A
B
C
A
B
C
0
5
10
5
10
15
10
15
20
A 0
A 5
A 10
B 5
B 10
B 15
C 10
C 15
C 20
sum
sum
sum
A 15
B 30
C 45

数据聚合与分组

- 给定具体的数据：

```
In [10]: df = pd.DataFrame({'key1' : ['a', 'a', 'b', 'b', 'a'],
   ....:                    'key2' : ['one', 'two', 'one', 'two', 'one'],
   ....:                    'data1' : np.random.randn(5),
   ....:                    'data2' : np.random.randn(5)})

In [11]: df
Out[11]:
      data1     data2 key1 key2
0 -0.204708  1.393406    a  one
1  0.478943  0.092908    a  two
2 -0.519439  0.281746    b  one
3 -0.555730  0.769023    b  two
4  1.965781  1.246435    a  one
```

数据聚合与分组

```
In [11]: df
Out[11]:
      data1     data2 key1 key2
0 -0.204708  1.393406    a  one
1  0.478943  0.092908    a  two
2 -0.519439  0.281746    b  one
3 -0.555730  0.769023    b  two
4  1.965781  1.246435    a  one
```

- 要按key1进行分组，并计算data1列的平均值，可以通过访问data1，并根据key1调用**groupby**：

```
In [12]: grouped = df['data1'].groupby(df['key1'])

In [13]: grouped
Out[13]: <pandas.core.groupby.SeriesGroupBy object at 0x7faa31537390>
```

- 变量grouped是一个**GroupBy对象**。它实际上还没有进行任何计算，只是含有一些有关分组键df[‘key1’]的中间数据而已。可以在此基础上进行各种计算，例如求平均值：

```
In [14]: grouped.mean()
Out[14]:
key1
a    0.746672
b   -0.537585
Name: data1, dtype: float64
```

数据（Series）根据分组键进行了聚合，产生了一个新的Series，其索引为key1列中的唯一值。

数据聚合与分组

```
In [11]: df
Out[11]:
      data1     data2 key1 key2
0 -0.204708  1.393406    a  one
1  0.478943  0.092908    a  two
2 -0.519439  0.281746    b  one
3 -0.555730  0.769023    b  two
4  1.965781  1.246435    a  one
```

- 一次传入多个数组的列表，就会得到经过层次化索引的结果：

```
In [15]: means = df['data1'].groupby([df['key1'], df['key2']]).mean()

In [16]: means
Out[16]:
key1  key2
a     one     0.880536
      two     0.478943
b     one    -0.519439
      two    -0.555730
Name: data1, dtype: float64
```

```
In [17]: means.unstack()
Out[17]:
key2       one       two
key1
a     0.880536  0.478943
b    -0.519439 -0.555730
```

数据聚合与分组

```
In [11]: df
Out[11]:
      data1     data2 key1 key2
0 -0.204708  1.393406    a  one
1  0.478943  0.092908    a  two
2 -0.519439  0.281746    b  one
3 -0.555730  0.769023    b  two
4  1.965781  1.246435    a  one
```

- 这以上例子中，分组键均为Series。实际上，分组键可以是任何**长度适当的数组**：

```
In [18]: states = np.array(['Ohio', 'California', 'California', 'Ohio', 'Ohio'])

In [19]: years = np.array([2005, 2005, 2006, 2005, 2006])

In [20]: df['data1'].groupby([states, years]).mean()
Out[20]:
California  2005    0.478943
            2006   -0.519439
Ohio        2005   -0.380219
            2006    1.965781
Name: data1, dtype: float64
```

数据聚合与分组

```
In [11]: df
Out[11]:
      data1     data2 key1 key2
0 -0.204708  1.393406    a  one
1  0.478943  0.092908    a  two
2 -0.519439  0.281746    b  one
3 -0.555730  0.769023    b  two
4  1.965781  1.246435    a  one
```

- 通常，分组信息就位于相同的要处理DataFrame中。于是，可以将列名（可以是字符串、数字或其他Python对象）用作分组键：

```
In [21]: df.groupby('key1').mean()
Out[21]:
         data1     data2
key1
a     0.746672  0.910916
b    -0.537585  0.525384

In [22]: df.groupby(['key1', 'key2']).mean()
Out[22]:
               data1     data2
key1 key2
a    one    0.880536  1.319920
     two    0.478943  0.092908
b    one   -0.519439  0.281746
     two   -0.555730  0.769023
```

- **size**方法返回每组的样本个数：

```
In [23]: df.groupby(['key1', 'key2']).size()
Out[23]:
key1  key2
a     one     2
      two     1
b     one     1
      two     1
dtype: int64
```

分组迭代

```
In [11]: df
Out[11]:
      data1     data2 key1 key2
0 -0.204708  1.393406    a  one
1  0.478943  0.092908    a  two
2 -0.519439  0.281746    b  one
3 -0.555730  0.769023    b  two
4  1.965781  1.246435    a  one
```

- GroupBy对象支持迭代，可以产生一组二元元组（由分组名和数据块组成）：

```
In [24]: for name, group in df.groupby('key1'):
   ....:     print(name)
   ....:     print(group)
   ....:
a
      data1     data2 key1 key2
0 -0.204708  1.393406    a  one
1  0.478943  0.092908    a  two
4  1.965781  1.246435    a  one
b
      data1     data2 key1 key2
2 -0.519439  0.281746    b  one
3 -0.555730  0.769023    b  two
```

- 可以将这些数据片段做成一个字典：

```
In [26]: pieces = dict(list(df.groupby('key1')))

In [27]: pieces['b']
Out[27]:
      data1     data2 key1 key2
2 -0.519439  0.281746    b  one
3 -0.555730  0.769023    b  two
```

分组迭代

```
In [11]: df
Out[11]:
      data1     data2 key1 key2
0 -0.204708  1.393406    a  one
1  0.478943  0.092908    a  two
2 -0.519439  0.281746    b  one
3 -0.555730  0.769023    b  two
4  1.965781  1.246435    a  one
```

- 对于多重键的情况，外层元组的第一个元素将会是由键值组成的元组：

```
In [25]: for (k1, k2), group in df.groupby(['key1', 'key2']):
   ....:     print((k1, k2))
   ....:     print(group)
   ....:
('a', 'one')
      data1     data2 key1 key2
0 -0.204708  1.393406    a  one
4  1.965781  1.246435    a  one
('a', 'two')
      data1     data2 key1 key2
1  0.478943  0.092908    a  two
('b', 'one')
      data1     data2 key1 key2
2 -0.519439  0.281746    b  one
('b', 'two')
     data1     data2 key1 key2
3 -0.55573  0.769023    b  two
```

分组后选取一列或列的子集

```
In [11]: df
Out[11]:
      data1     data2 key1 key2
0 -0.204708  1.393406    a  one
1  0.478943  0.092908    a  two
2 -0.519439  0.281746    b  one
3 -0.555730  0.769023    b  two
4  1.965781  1.246435    a  one
```

- 对于由DataFrame产生的GroupBy对象，如果用一个（单个字符串）或一组（字符串数组）列名对其进行索引，就能实现选取部分列进行聚合的目的。
- 在进行较大规模数据分析时，可以只选取一部分进行计算。为了方便程序编写，pandas提供了以下**语法糖**：

```
df.groupby('key1')['data1']
df.groupby('key1')[['data2']]
```

are syntactic sugar for:

```
df['data1'].groupby(df['key1'])
df[['data2']].groupby(df['key1'])
```

例子：

```
In [31]: df.groupby(['key1', 'key2'])[['data2']].mean()
Out[31]:
              data2
key1 key2
a    one   1.319920
     two   0.092908
b    one   0.281746
     two   0.769023
```

使用字典或Series分组

- 考虑数据：

```
In [35]: people = pd.DataFrame(np.random.randn(5, 5),
   ....:                       columns=['a', 'b', 'c', 'd', 'e'],
   ....:                       index=['Joe', 'Steve', 'Wes', 'Jim', 'Travis'])

In [36]: people.iloc[2:3, [1, 2]] = np.nan # Add a few NA values

In [37]: people
Out[37]:
              a         b         c         d         e
Joe    1.007189 -1.296221  0.274992  0.228913  1.352917
Steve  0.886429 -2.001637 -0.371843  1.669025 -0.438570
Wes   -0.539741       NaN       NaN -1.021228 -0.577087
Jim    0.124121  0.302614  0.523772  0.000940  1.343810
Travis -0.713544 -0.831154 -2.370232 -1.860761 -0.860757
```

使用字典或Series分组

```
In [37]: people
Out[37]:
               a         b         c         d         e
Joe     1.007189 -1.296221  0.274992  0.228913  1.352917
Steve   0.886429 -2.001637 -0.371843  1.669025 -0.438570
Wes    -0.539741       NaN       NaN -1.021228 -0.577087
Jim     0.124121  0.302614  0.523772  0.000940  1.343810
Travis -0.713544 -0.831154 -2.370232 -1.860761 -0.860757
```

- 假设已知列的分组关系，并希望根据分组计算列的和：

```
In [38]: mapping = {'a': 'red', 'b': 'red', 'c': 'blue',
   ....:            'd': 'blue', 'e': 'red', 'f' : 'orange'}
```

- 可以直接传递字典给**groupby**:

```
In [39]: by_column = people.groupby(mapping, axis=1)

In [40]: by_column.sum()
Out[40]:
            blue       red
Joe     0.503905  1.063885
Steve   1.297183 -1.553778
Wes    -1.021228 -1.116829
Jim     0.524712  1.770545
Travis -4.230992 -2.405455
```

请注意，字典中多余的键“f”不影响程序正常工作。

使用字典或Series分组

```
In [37]: people
Out[37]:
               a         b         c         d         e
Joe     1.007189 -1.296221  0.274992  0.228913  1.352917
Steve   0.886429 -2.001637 -0.371843  1.669025 -0.438570
Wes    -0.539741       NaN       NaN -1.021228 -0.577087
Jim     0.124121  0.302614  0.523772  0.000940  1.343810
Travis -0.713544 -0.831154 -2.370232 -1.860761 -0.860757
```

- 同样可以直接传递**Series**进行分组（**Series**可以被看做一个固定大小的映射）：

```
In [41]: map_series = pd.Series(mapping)

In [42]: map_series
Out[42]:
a       red
b       red
c      blue
d      blue
e       red
f    orange
```

```
In [43]: people.groupby(map_series, axis=1).count()
Out[43]:
        blue  red
Joe        2    3
Steve      2    3
Wes        1    2
Jim        2    3
Travis     2    3
```

使用函数分组

- 比起使用字典或Series，使用Python函数是一种更原生的方法定义分组映射。任何被当做分组键的函数都会在各个索引值上被调用一次，其返回值就会被用作分组名称。

```
In [44]: people.groupby(len).sum()
Out[44]:
          a         b         c         d         e
3  0.591569 -0.993608  0.798764 -0.791374  2.119639
5  0.886429 -2.001637 -0.371843  1.669025 -0.438570
6 -0.713544 -0.831154 -2.370232 -1.860761 -0.860757

In [45]: key_list = ['one', 'one', 'one', 'two', 'two']

In [46]: people.groupby([len, key_list]).min()
Out[46]:
              a         b         c         d         e
3 one -0.539741 -1.296221  0.274992 -1.021228 -0.577087
  two  0.124121  0.302614  0.523772  0.000940  1.343810
5 one  0.886429 -2.001637 -0.371843  1.669025 -0.438570
6 two -0.713544 -0.831154 -2.370232 -1.860761 -0.860757
```

根据索引级别分组

- 层次化索引数据集最方便的地方就在于它能够根据轴索引的一个级别进行聚合：

```
In [47]: columns = pd.MultiIndex.from_arrays([['US', 'US', 'US', 'JP', 'JP'],
   ....:                                      [1, 3, 5, 1, 3]],
   ....:                                      names=['cty', 'tenor'])

In [48]: hier_df = pd.DataFrame(np.random.randn(4, 5), columns=columns)
In [49]: hier_df
Out[49]:
cty          US                            JP
tenor         1         3         5         1         3
0      0.560145 -1.265934  0.119827 -1.063512  0.332883
1     -2.359419 -0.199543 -1.541996 -0.970736 -1.307030
2      0.286350  0.377984 -0.753887  0.331286  1.349742
3      0.069877  0.246674 -0.011862  1.004812  1.327195
In [50]: hier_df.groupby(level='cty', axis=1).count()
Out[50]:
cty  JP  US
0     2   3
1     2   3
2     2   3
3     2   3
```

要根据级别分组，使用level关键字传递级别序号或名字。

数据聚合

- 聚合（**Aggregations**）指的是任何能够从数组产生标量值的数据转换过程，比如之前例子用到的mean、count、min以及sum等。常用的经过优化的聚合函数如下：

Table 10-1. Optimized groupby methods

Function name	Description
`count`	Number of non-NA values in the group
`sum`	Sum of non-NA values
`mean`	Mean of non-NA values
`median`	Arithmetic median of non-NA values
`std, var`	Unbiased (n – 1 denominator) standard deviation and variance
`min, max`	Minimum and maximum of non-NA values
`prod`	Product of non-NA values
`first, last`	First and last non-NA values

数据聚合

- 聚合还可采用自行定义的聚合运算，或分组对象上已经定义好的任何方法，利于分位数**quantile**：

```
In [51]: df
Out[51]:
      data1     data2 key1 key2
0 -0.204708  1.393406    a  one
1  0.478943  0.092908    a  two
2 -0.519439  0.281746    b  one
3 -0.555730  0.769023    b  two
4  1.965781  1.246435    a  one

In [52]: grouped = df.groupby('key1')

In [53]: grouped['data1'].quantile(0.9)
Out[53]:
key1
a    1.668413
b   -0.523068
Name: data1, dtype: float64
```

```
In [54]: def peak_to_peak(arr):
   ....:     return arr.max() - arr.min()

In [55]: grouped.agg(peak_to_peak)
Out[55]:
         data1     data2
key1
a     2.170488  1.300498
b     0.036292  0.487276
```

数据聚合

- **describe**方法同样适用于此：

```
In [56]: grouped.describe()
Out[56]:
     data1                                                                \
     count      mean       std       min       25%       50%       75%
key1
a      3.0  0.746672  1.109736 -0.204708  0.137118  0.478943  1.222362
b      2.0 -0.537585  0.025662 -0.555730 -0.546657 -0.537585 -0.528512
               data2                                                      \
           max count      mean       std       min       25%       50%
key1
a     1.965781   3.0  0.910916  0.712217  0.092908  0.669671  1.246435
b    -0.519439   2.0  0.525384  0.344556  0.281746  0.403565  0.525384

           75%       max
key1
a     1.319920  1.393406
b     0.647203  0.769023
```

多列数据聚合

tips.csv

```
total_bill,tip,smoker,day,time,size
16.99,1.01,No,Sun,Dinner,2
10.34,1.66,No,Sun,Dinner,3
21.01,3.5,No,Sun,Dinner,3
23.68,3.31,No,Sun,Dinner,2
24.59,3.61,No,Sun,Dinner,4
25.29,4.71,No,Sun,Dinner,4
8.77,2.0,No,Sun,Dinner,2
26.88,3.12,No,Sun,Dinner,4
15.04,1.96,No,Sun,Dinner,2
14.78,3.23,No,Sun,Dinner,2
10.27,1.71,No,Sun,Dinner,2
35.26,5.0,No,Sun,Dinner,4
15.42,1.57,No,Sun,Dinner,2
18.43,3.0,No,Sun,Dinner,4
14.83,3.02,No,Sun,Dinner,2
```

- 对Series或DataFrame列的聚合运算其实就是使用**aggregate**（使用自定义函数）或调用诸如**mean**、**std**之类的方法。对**不同的列**使用**不同的聚合函数**，或**一次应用多个函数**都是可以实现的。
- 首先读入数据，并计算衍生指标：

```
In [57]: tips = pd.read_csv('examples/tips.csv')

# Add tip percentage of total bill
In [58]: tips['tip_pct'] = tips['tip'] / tips['total_bill']

In [59]: tips[:6]
Out[59]:
   total_bill   tip smoker  day    time  size    tip_pct
0       16.99  1.01     No  Sun  Dinner     2  0.059447
1       10.34  1.66     No  Sun  Dinner     3  0.160542
2       21.01  3.50     No  Sun  Dinner     3  0.166587
3       23.68  3.31     No  Sun  Dinner     2  0.139780
4       24.59  3.61     No  Sun  Dinner     4  0.146808
5       25.29  4.71     No  Sun  Dinner     4  0.186240
```

多列数据聚合

```
In [59]: tips[:6]
Out[59]:
   total_bill   tip smoker  day    time  size   tip_pct
0       16.99  1.01     No  Sun  Dinner     2  0.059447
1       10.34  1.66     No  Sun  Dinner     3  0.160542
2       21.01  3.50     No  Sun  Dinner     3  0.166587
3       23.68  3.31     No  Sun  Dinner     2  0.139780
4       24.59  3.61     No  Sun  Dinner     4  0.146808
5       25.29  4.71     No  Sun  Dinner     4  0.186240
```

- 首先根据day和smoker对tips进行分组：

```
In [60]: grouped = tips.groupby(['day', 'smoker'])
In [61]: grouped_pct = grouped['tip_pct']
```

```
In [62]: grouped_pct.agg('mean')
Out[62]:
day   smoker
Fri   No        0.151650
      Yes       0.174783
Sat   No        0.158048
      Yes       0.147906
Sun   No        0.160113
      Yes       0.187250
Thur  No        0.160298
      Yes       0.163863
Name: tip_pct, dtype: float64
```

- 函数名可以以字符串的形式传入：

```
In [63]: grouped_pct.agg(['mean', 'std', peak_to_peak])
Out[63]:
                mean       std  peak_to_peak
day  smoker
Fri  No      0.151650  0.028123      0.067349
     Yes     0.174783  0.051293      0.159925
Sat  No      0.158048  0.039767      0.235193
     Yes     0.147906  0.061375      0.290095
Sun  No      0.160113  0.042347      0.193226
     Yes     0.187250  0.154134      0.644685
Thur No      0.160298  0.038774      0.193350
     Yes     0.163863  0.039389      0.151240
```

多列数据聚合

- 可以传入一个由(name, function)元组组成的列表，则各元组的第一个元素就会被用作DataFrame的列名，第二个元素作为函数名：

```
In [64]: grouped_pct.agg([('foo', 'mean'), ('bar', np.std)])
Out[64]:
                  foo       bar
day  smoker
Fri  No      0.151650  0.028123
     Yes     0.174783  0.051293
Sat  No      0.158048  0.039767
     Yes     0.147906  0.061375
Sun  No      0.160113  0.042347
     Yes     0.187250  0.154134
Thur No      0.160298  0.038774
     Yes     0.163863  0.039389
```

多列数据聚合

- 可以指定对各列计算相同的聚合函数：

```
In [65]: functions = ['count', 'mean', 'max']

In [66]: result = grouped['tip_pct', 'total_bill'].agg(functions)

In [67]: result
Out[67]:
             tip_pct                     total_bill
               count      mean       max      count       mean    max
day  smoker
Fri  No            4  0.151650  0.187735          4  18.420000  22.75
     Yes          15  0.174783  0.263480         15  16.813333  40.17
Sat  No           45  0.158048  0.291990         45  19.661778  48.33
     Yes          42  0.147906  0.325733         42  21.276667  50.81
Sun  No           57  0.160113  0.252672         57  20.506667  48.17
     Yes          19  0.187250  0.710345         19  24.120000  45.35
Thur No           45  0.160298  0.266312         45  17.113111  41.19
     Yes          17  0.163863  0.241255         17  19.190588  43.11
```

- 结果DataFrame拥有层次化的列，这相当于分别对各列进行聚合，然后用concat将结果组装到一起，使用列名用作keys参数.

多列数据聚合

- 上述结果同样可以通过传入带有自定义名称的一组元组实现：

```
In [69]: ftuples = [('Durchschnitt', 'mean'), ('Abweichung', np.var)]

In [70]: grouped['tip_pct', 'total_bill'].agg(ftuples)
Out[70]:
                tip_pct                total_bill
           Durchschnitt Abweichung Durchschnitt  Abweichung
day  smoker
Fri  No        0.151650   0.000791    18.420000   25.596333
     Yes       0.174783   0.002631    16.813333   82.562438
Sat  No        0.158048   0.001581    19.661778   79.908965
     Yes       0.147906   0.003767    21.276667  101.387535
Sun  No        0.160113   0.001793    20.506667   66.099980
     Yes       0.187250   0.023757    24.120000  109.046044
Thur No        0.160298   0.001503    17.113111   59.625081
     Yes       0.163863   0.001551    19.190588   69.808518
```

多列数据聚合

- 如果需要对一个列或不同的列应用不同的函数。具体的办法是向**agg**传入一个从列名映射到函数的字典

```
In [71]: grouped.agg({'tip' : np.max, 'size' : 'sum'})
Out[71]:
                tip  size
day  smoker
Fri  No       3.50     9
     Yes      4.73    31
Sat  No       9.00   115
     Yes     10.00   104
Sun  No       6.00   167
     Yes      6.50    49
Thur No       6.70   112
     Yes      5.00    40
```

```
In [72]: grouped.agg({'tip_pct' : ['min', 'max', 'mean', 'std'],
   ....:              'size' : 'sum'})
Out[72]:
                tip_pct                               size
                    min       max      mean       std  sum
day  smoker
Fri  No        0.120385  0.187735  0.151650  0.028123    9
     Yes       0.103555  0.263480  0.174783  0.051293   31
Sat  No        0.056797  0.291990  0.158048  0.039767  115
     Yes       0.035638  0.325733  0.147906  0.061375  104
Sun  No        0.059447  0.252672  0.160113  0.042347  167
     Yes       0.065660  0.710345  0.187250  0.154134   49
Thur No        0.072961  0.266312  0.160298  0.038774  112
     Yes       0.090014  0.241255  0.163863  0.039389   40
```