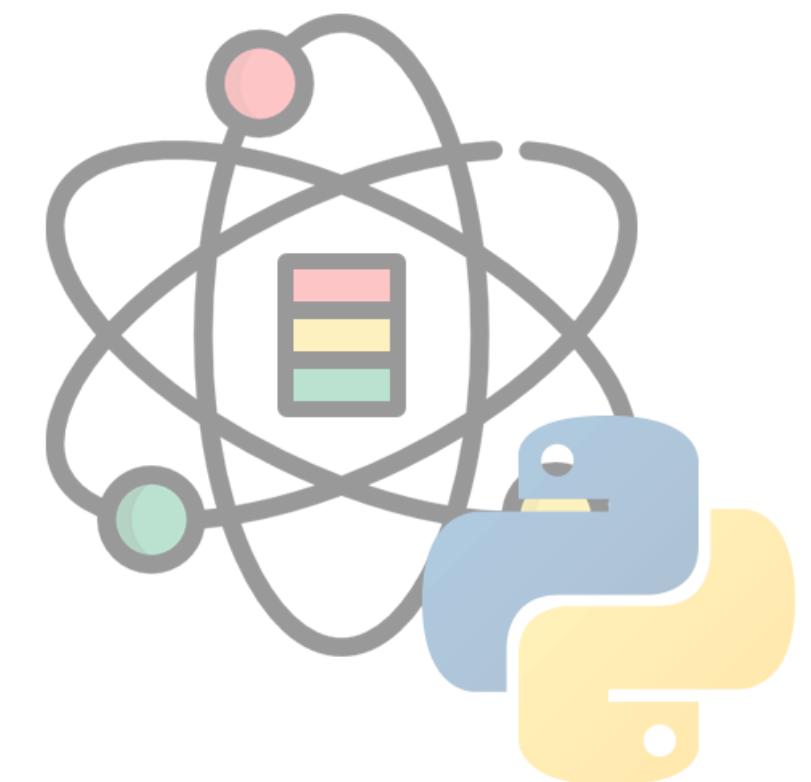


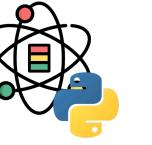
数据分析基础 (下)

Data Analytics Introduction - Part 2

范叶亮 Leo Van



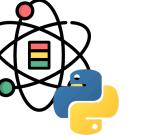
目录



- pandas 简介
- Python 数据载入和存储
- Python 数据规整

pandas 简介

pandas 简介



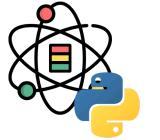
pandas 所包含的数据结构和数据处理工具的设计使得在 Python 中进行数据清洗和分析非常便捷。尽管 pandas 采用了很多 NumPy 的代码风格，但最大的不同在于 pandas 是用来处理表格型或异质类型数据的，而 NumPy 则相反，它更适合处理同质类型的数据。

在后续内容中，我们会使用如下的快捷方式导入 pandas：

```
import pandas as pd
from pandas import Series, DataFrame
```

pandas 中的两个常用的工具数据结构：Series 和 DataFrame 可以为大多数应用提供一个有效易用的基础。

Series



Series 是一种一维的数据型对象，它包含了一个值序列（与 NumPy 中的类型相似），并且包含了数据标签，称之为索引 (index)，最简单的序列可以仅由一个数组构成：

```
s = pd.Series([1, 2, 3])
```

```
s
```

```
0    1
1    2
2    3
dtype: int64
```

交互式环境中 Series 的表示左边为索引，右边为值。由于我们不为数据指定索引，默认生成的索引是从 0 到 $N-1$ (N 是数据的长度)。可以通过 values 属性和 index 属性分别获取 Series 对象的值和索引。

通过添加索引为每个数据点添加标识：

```
s = pd.Series([1, 2, 3], index=['a', 'b', 'c'])
```

```
s
```

```
a    1
b    2
c    3
dtype: int64
```

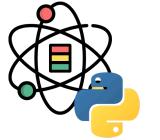
```
s.index
```

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

```
s.values
```

```
array([1, 2, 3])
```

Series



与 NumPy 相比，可以在数据中选取数据时使用标签：

```
s['a']
```

```
1
```

```
s['b'] = 6
```

```
s[['c', 'b', 'a']]
```

```
c 3
```

```
b 6
```

```
a 1
```

```
dtype: int64
```

使用 NumPy 函数或 NumPy 风格操作：

```
s[s > 1]
```

```
b 6
```

```
c 3
```

```
dtype: int64
```

```
s ** 2
```

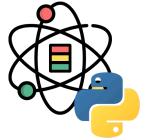
```
a 2
```

```
b 12
```

```
c 6
```

```
dtype: int64
```

Series



Series 可以被看做是一个长度固定且有序的字典，因为它将索引值和数据值按位置配对。

```
'b' in s
```

```
True
```

```
'e' in s
```

```
False
```

可以使用字典生成一个 Series：

```
d = {  
    'Ohio': 35000, 'Texas': 71000,  
    'Oregon': 16000, 'Utah': 5000  
}
```

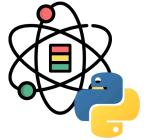
```
s1 = pd.Series(d)  
s1
```

```
Ohio      35000  
Texas     71000  
Oregon    16000  
Utah      5000  
dtype: int64
```

当把字典传递给 Series 时，产生的 Series 的索引将是排序好的字典键。也可以将字典键按照所需的顺序传递给构造函数参数 index，从而生成的 Series 的索引顺序将与传入的字典键相符。

```
states = ['California', 'Ohio', 'Oregon', 'Texas']  
s2 = pd.Series(d, index=states)
```

Series



因为 California 没有出现在 d 的键中，它对应的值是 NaN (Not a Number)，是 pandas 中用于标记缺失值的方式。因为 Utah 不在 d 中，它被排除在结果外。

pandas 使用 isnull 和 notnull 函数检查缺失值：

```
pd.isnull(s2)
```

```
California    True
Ohio        False
Oregon      False
Texas      False
dtype: bool
```

```
pd.notnull(s2)
```

```
California   False
Ohio          True
Oregon        True
Texas        True
dtype: bool
```

isnull 和 notnull 也是 Series 实例方法。

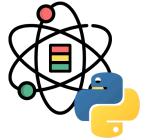
自动对齐索引是 Series 的一个重要特性：

```
s1 + s2
```

```
California      NaN
Ohio        70000.0
Oregon      32000.0
Texas        142000.0
Utah          NaN
dtype: float64
```

该特性与数据库中的 join 操作非常类似。

DataFrame



DataFrame 表示的是矩阵的数据表，它包含已排序的列集合，每一列可以是不同的值类型（数值、字符串、布尔值等）。DataFrame 既有行索引也有列索引，它可以被视为一个共享相同索引的 Series 的字典。在 DataFrame 中，数据被存储为一个以上的二维块，而不是列表、字典或其他一维数组的结合。

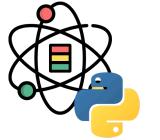
有多种方式可以构建 DataFrame，其中最常用的方式是利用包含等长度列表或 NumPy 数组的字典来构建 DataFrame：

```
data = {  
    'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada', 'Nevada'],  
    'year': [2000, 2001, 2002, 2000, 2001, 2002],  
    'pop': [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]  
}  
df = pd.DataFrame(data)
```

df

	state	year	pop
0	Ohio	2000	1.5
1	Ohio	2001	1.7
2	Ohio	2002	3.6
3	Nevada	2000	2.4
4	Nevada	2001	2.9
5	Nevada	2002	3.2

DataFrame



对于大型 DataFrame, `head` 方法会选出头部的 n 行:

```
df.head(2)
```

	state	year	pop
0	Ohio	2000	1.5
1	Ohio	2001	1.7

DataFrame 可以按照指定列的顺序排序:

```
pd.DataFrame(data, columns=['year', 'state', 'pop'])
```

	year	state	pop
0	2000	Ohio	1.5
1	2001	Ohio	1.7
2	2002	Ohio	3.6
3	2000	Nevada	2.4
4	2001	Nevada	2.9
5	2002	Nevada	3.2

DataFrame 中的一列可以按照字典型标记或属性检索为 Series:

```
df['state']
```

0	Ohio
1	Ohio
2	Ohio
3	Nevada
4	Nevada
5	Nevada

Name: state, dtype: object

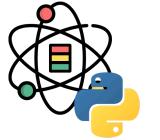
```
df.year
```

0	2000
1	2001
2	2002
3	2000
4	2001
5	2002

Name: year, dtype: int64

请注意, 返回的 Series 与原 DataFrame 有相同的索引, 且 Series 的 `names` 属性也会被合理的设置。

DataFrame



当将列表或数组赋值给一个列时，值的长度必须和 DataFrame 匹配。如果将 Series 赋值给一列时，Series 的索引将会按照 DataFrame 的索引重新排序，并在空缺的地方填充缺失值：

```
val = pd.Series([-1.2, -1.5, -1.7], index=[1, 3, 4])
df['debt'] = val
df
```

```
   state  year  pop  debt
0  Ohio  2000  1.5    NaN
1  Ohio  2001  1.7  -1.2
2  Ohio  2002  3.6    NaN
3 Nevada  2000  2.4  -1.5
4 Nevada  2001  2.9  -1.7
5 Nevada  2002  3.2    NaN
```

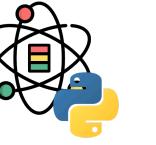
del 方法可以用于移除之前新建的列：

```
del df['debt']
df
```

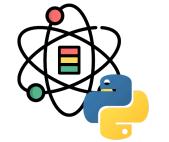
```
   state  year  pop
0  Ohio  2000  1.5
1  Ohio  2001  1.7
2  Ohio  2002  3.6
3 Nevada  2000  2.4
4 Nevada  2001  2.9
5 Nevada  2002  3.2
```

从 DataFrame 中选取的列是数据的视图，不是拷贝。因此，对 Series 的修改会映射到 DataFrame 中。如果需要复制，则应当显示地使用 Series 的 copy 方法。

DataFrame



索引对象



pandas 中的索引对象是用于储存轴标签和其他元数据的（例如：轴名称或标签），在构造 Series 或 DataFrame 时，所使用的任意数组或标签序列都可以在内部转换为索引对象。

```
s = pd.Series(range(3), index=['a', 'b', 'c'])
index = s.index
index
```



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

```
index[1:]
```

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

索引对象是不可变的，因此用户是无法修改索引对象的，这使得在多种数据结构中分享索引对象是安全的。

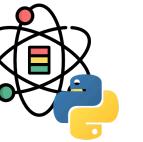
与 Python 集合不同，pandas 索引对象可以包含重复标签：

```
dup_idx = pd.Index(['a', 'a', 'b', 'b'])
dup_idx
```

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

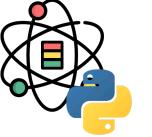
根据重复标签筛选，会选取所有重复标签对应的数据。

索引对象



方法	描述	方法	描述
append	将额外的索引对象粘贴到原索引后，产生一个新的索引	difference	计算两个索引的差集
intersection	计算两个索引的交集	union	计算两个索引的并集
isin	计算表示每一个值是否在传值容器中的布尔数组	delete	将位置 <i>i</i> 的元素删除，并产生新的索引
drop	根据传参删除指定索引值，并产生新的索引	insert	在位置 <i>i</i> 插入元素，并产生新的索引
is_monotonic	如果索引序列递增则返回 True	is_unique	如果索引序列唯一则返回 True
unique	计算索引的唯一值序列		

重建索引



`reindex` 用于创建一个符合新索引的新对象：

```
s = pd.Series([2, 4, 1, 3], index=['d', 'b', 'a', 'c'])  
s
```

```
d    2  
b    4  
a    1  
c    3  
dtype: int64
```

在 DataFrame 中，`reindex` 可以改变行索引，列索引，也可以同时改变二者。当仅传入一个序列时，结果中的行会重建索引，列可以使用 `columns` 关键字重建索引。

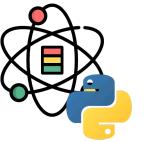
```
df = pd.DataFrame(  
    np.arange(9).reshape((3, 3)),  
    index=['a', 'b', 'c'],  
    columns=['Ohio', 'Texas', 'California'])  
df.reindex(['a', 'b', 'c', 'd'])
```

	Ohio	Texas	California
a	0.0	1.0	2.0
b	3.0	4.0	5.0
c	6.0	7.0	8.0
d	NaN	NaN	NaN

```
df.reindex(columns=['Texas', 'Utah', 'California'])
```

	Texas	Utah	California
a	1	NaN	2
b	4	NaN	5
c	7	NaN	8

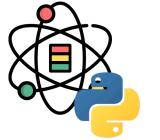
重建索引



`reindex` 方法的参数详见下表:

参数	描述
<code>index</code>	新建作为索引的序列，可以是索引示例或任意其他序列型 Python 数据结构，索引使用时无需复制
<code>method</code>	插值方法， <code>ffill</code> 为向前填充， <code>bfill</code> 为向后填充
<code>fill_value</code>	通过重新索引引入缺失数据时使用的替代值
<code>limit</code>	当前向或后向填充时，所需填充的最大尺寸间隔
<code>tolerance</code>	当前向或后向填充时，所需填充的不精确匹配下的最大尺寸间隔
<code>level</code>	匹配 <code>MultiIndex</code> 级别的简单索引
<code>copy</code>	如果为 <code>True</code> ，即使新索引等于旧索引，也总是复制底层数据

删除条目



drop 方法返回含有指示值或轴向上删除值的新对象：

```
s = pd.Series(np.arange(3.), index=['a', 'b', 'c'])  
s
```

```
a    0.0  
b    1.0  
c    2.0  
dtype: float64
```

```
new_s = s.drop('b')  
new_s
```

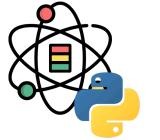
```
a    0.0  
c    2.0  
dtype: float64
```

在 DataFrame 中，索引值可以从轴向上删除：

```
df = pd.DataFrame(  
    np.arange(16).reshape((4, 4)),  
    index=['Ohio', 'Colorado', 'Utah', 'New York'],  
    columns=['one', 'two', 'three', 'four'])  
df
```

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

删除条目



drop 会根据行标签删除值 (轴 0):

```
df.drop(['Colorado', 'Ohio'])
```

	one	two	three	four
Utah	8	9	10	11
New York	12	13	14	15

```
df.drop(['two', 'four'], axis='columns')
```

	one	three
Ohio	0	2
Colorado	4	6
Utah	8	10
New York	12	14

可以通过传递 axis=1 或 axis='columns' 删除列:

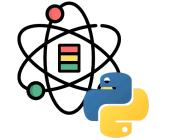
```
df.drop(['two'], axis=1)
```

	one	three	four
Ohio	0	2	3
Colorado	4	6	7
Utah	8	10	11
New York	12	14	15

```
df.drop('Utah', inplace=True)  
df
```

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7
New York	12	13	14	15

索引, 选择和过滤



Series 的索引与 NumPy 数组索引类似:

```
s = pd.Series(  
    np.arange(4.), index=['a', 'b', 'c', 'd'])
```

```
s
```

```
a    0.0  
b    1.0  
c    2.0  
d    3.0  
dtype: float64
```

```
s['b']
```

```
1.0
```

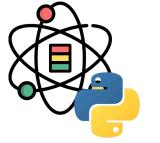
```
s[1]
```

```
1.0
```

```
s[2:4]
```

```
c    2.0  
d    3.0  
dtype: float64
```

索引, 选择和过滤



```
s[['b', 'a']]
```

```
b    1.0  
a    0.0  
dtype: float64
```

```
s[s < 2]
```

```
a    0.0  
b    1.0  
dtype: float64
```

普通的 Python 切片中是不包含尾部的, Series 的切片与之不同:

```
s['b':'c']
```

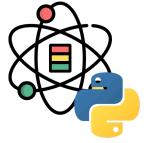
```
b    1.0  
c    2.0  
dtype: float64
```

使用这些方法设置值是会修改 Series 相应的部分:

```
s['b':'c'] = 6  
s
```

```
a    0.0  
b    6.0  
c    6.0  
d    3.0  
dtype: float64
```

索引, 选择和过滤



使用单个值或序列，可以从 DataFrame 中索引出一个或多个列：

```
df = pd.DataFrame(  
    np.arange(16).reshape((4, 4)),  
    index=['Ohio', 'Colorado', 'Utah', 'New York'],  
    columns=['one', 'two', 'three', 'four'])  
df
```

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

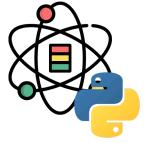
```
df['two']
```

```
Ohio      1  
Colorado  5  
Utah     9  
New York 13  
Name: two, dtype: int64
```

```
df[['three', 'one']]
```

	three	one
Ohio	2	0
Colorado	6	4
Utah	10	8
New York	14	12

索引, 选择和过滤



可以根据一个布尔值数组切片或选择数据:

```
df[:2]
```

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7

```
df[df['three'] > 5]
```

	one	two	three	four
Colorado	4	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

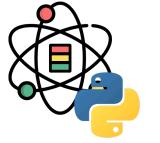
行选择语法 `df[:2]` 非常方便, 传递单个元素或一个列表到 [] 中可以选择列。

通过布尔值也可以对 DataFrame 进行索引, 布尔值 DataFrame 可以是对标量值进行比较产生:

```
df[df < 6] = 0  
df
```

	one	two	three	four
Ohio	0	0	0	0
Colorado	0	0	6	7
Utah	8	9	10	11
New York	12	13	14	15

索引, 选择和过滤



使用 `loc` 和 `iloc` 可以对 DataFrame 在行上的标签索引，轴标签 (`loc`) 或整数标签 (`iloc`) 以 NumPy 风格的语法从 DataFrame 中选出数组的行和列的子集。

```
df.loc['Colorado', ['two', 'three']]
```

```
two      0
three    6
Name: Colorado, dtype: int64
```

```
df.iloc[2, [3, 0, 1]]
```

```
four     11
one      8
two      9
Name: Utah, dtype: int64
```

除了单个标签或标签列表之外，索引功能还可以用于切片：

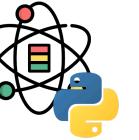
```
df.loc[:'Utah', 'two']
```

```
Ohio      0
Colorado  0
Utah      9
Name: two, dtype: int64
```

```
df.iloc[:, :3][df.three > 5]
```

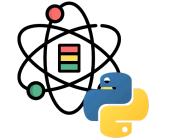
	one	two	three
Colorado	0	0	6
Utah	8	9	10
New York	12	13	14

索引选项



类型	描述
<code>df[val]</code>	从 DataFrame 中选择单列或列序列；特殊情况的便利：布尔数组（过滤行），切片（切片行）或布尔值 DataFrame（根据某些标准设置的值）
<code>df.loc[val]</code>	根据标签选择 DataFrame 的单行或多行
<code>df.loc[:, val]</code>	根据标签选择单列或多列
<code>df.loc[val1, val2]</code>	同时选择行和列的一部分
<code>df.iloc[where]</code>	根据整数位置选择单行或多行
<code>df.iloc[:, where]</code>	根据整数位置选择单列或多列
<code>df.iloc[where_i, where_j]</code>	根据整数位置选择行和列
<code>df.at[label_i, label_j]</code>	根据行、列标签选择单个标量值
<code>df.iat[i, j]</code>	根据行、列整数位置选择单个标量值

算术和数据对齐



不同索引对象之间的算术行为是 pandas 提供的一项重要特性，将对象相加时，如果存在某个索引对不同，则返回结果的索引将是索引对的并集。

```
s1 = pd.Series([1, 2, 3, 4], index=['a', 'c', 'd', 'e'])
s2 = pd.Series([5, 6, 7, 8, 9], index=['a', 'c', 'e', 'f', 'g'])
```

s1

```
a    1
c    2
d    3
e    4
dtype: int64
```

s2

```
a    5
c    6
e    7
f    8
g    9
dtype: int64
```

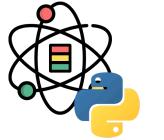
将这些对象相加则会产生：

s1 + s2

```
a    6.0
c    8.0
d    NaN
e    11.0
f    NaN
g    NaN
dtype: float64
```

没有交叠的标签位置上，内部数据对齐会产生缺失值。缺失值会在后续的算数操作上产生影响。

算术和数据对齐



在 DataFrame 中，行和列上都会执行对齐：

```
df1 = pd.DataFrame(  
    np.arange(9).reshape(3, 3), columns=list('bcd'),  
    index=['Ohio', 'Texas', 'Colorado'])  
  
df2 = pd.DataFrame(  
    np.arange(12).reshape(4, 3), columns=list('bde'),  
    index=['Utah', 'Ohio', 'Texas', 'Oregon'])
```

df1

	b	c	d
Ohio	0	1	2
Texas	3	4	5
Colorado	6	7	8

df2

	b	d	e
Utah	0	1	2
Ohio	3	4	5
Texas	6	7	8
Oregon	9	10	11

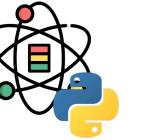
将这些对象加在一起，返回一个 DataFrame，它的索引和列是每个 DataFrame 的索引和列的并集。

df1 + df2

	b	c	d	e
Colorado	NaN	NaN	NaN	NaN
Ohio	3.0	NaN	6.0	NaN
Oregon	NaN	NaN	NaN	NaN
Texas	9.0	NaN	12.0	NaN
Utah	NaN	NaN	NaN	NaN

由于 c 和 e 列并不是两个 DataFrame 共有的列，这两列中产生了缺失值。对于行标签不同的 DataFrame 对象也是如此。

算术和数据对齐



在 `df1` 上使用 `add` 方法，我们将 `df2` 和一个 `fill_value` 作为参数传入：

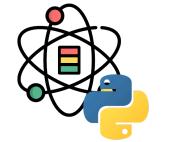
```
df1.add(df2, fill_value=0)
```

	b	c	d	e
Colorado	6.0	7.0	8.0	NaN
Ohio	3.0	1.0	6.0	5.0
Oregon	9.0	NaN	10.0	11.0
Texas	9.0	4.0	12.0	8.0
Utah	0.0	NaN	1.0	2.0

每个方法都有一个以 `r` 开头的副本，副本方法的参数是翻转的，例如：`1 / df1` 和 `df1.rdiv(1)` 是等价的。

方法	描述
<code>add, radd</code>	加法 (+)
<code>sub, rsub</code>	减法 (-)
<code>div, rdiv</code>	除法 (/)
<code>floordiv, rfloordiv</code>	整除 (//)
<code>mul, rmul</code>	乘法 (*)
<code>pow, rpow</code>	幂次方 (**)

算术和数据对齐



DataFrame 和 Series 间的算数操作与 NumPy 中不同维度数组间的操作类似。

```
df = pd.DataFrame(  
    np.arange(12).reshape(4, 3), columns=list('bde'),  
    index=['Utah', 'Ohio', 'Texas', 'Oregon'])  
s = df.iloc[0]
```

df

```
      b   d   e  
Utah  0   1   2  
Ohio  3   4   5  
Texas 6   7   8  
Oregon 9  10  11
```

s

```
      b   0  
      d   1  
      e   2  
Name: Utah, dtype: int64
```

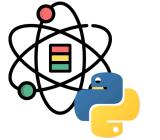
默认情况下， DataFrame 和 Series 数学操作中会将 Series 索引和 DataFrame 的列进行匹配，并广播到各行：

df - s

```
      b   d   e  
Utah  0   0   0  
Ohio  3   3   3  
Texas 6   6   6  
Oregon 9  9   9
```

如果索引值不在 DataFrame 的列中，也不再 Series 的索引中，则对象会重建索引并形成联合。如果想改为在列上进行广播，在行上匹配，则必须使用算法方法中的一种，通过 `axis` 参数进行轴匹配，`axis='index'` 或 `axis=0` 为行匹配。

函数应用和映射



NumPy 的通用函数（逐元素数组方法）对 pandas 对象也有效：

```
df = pd.DataFrame(  
    np.random.randn(4, 3), columns=list('bde'),  
    index=['Utah', 'Ohio', 'Texas', 'Oregon'])  
df
```

```
          b      d      e  
Utah -0.752446 -0.709925 -0.337916  
Ohio  0.726541  2.137040 -2.288782  
Texas 0.674664 -0.896596  2.181261  
Oregon 0.885203 -1.067982 -1.518018
```

```
np.abs(df)
```

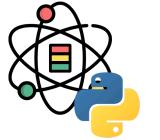
	b	d	e
Utah	0.752446	0.709925	0.337916
Ohio	0.726541	2.137040	2.288782
Texas	0.674664	0.896596	2.181261
Oregon	0.885203	1.067982	1.518018

一个常用的操作是将函数应用到一行或一列的一维数组上， DataFrame 的 apply 方法可以实现该功能：

```
f = lambda x: x.max() - x.min()  
df.apply(f)
```

```
b    1.637649  
d    3.205021  
e    4.470043  
dtype: float64
```

函数应用和映射



通过传递 `axis='columns'` 给 `apply` 函数，函数会被每行调用一次：

```
df.apply(f, axis='columns')
```

```
Utah      0.414531
Ohio      4.425821
Texas     3.077857
Oregon    2.403221
dtype: float64
```

逐元素的 Python 函数也可以，例如对 `df` 中的每个值计算一个格式化的字符串，可以使用 `applymap` 方法：

```
f = lambda x: '{:2f}'.format(x)
df.applymap(f)
```

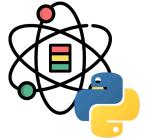
	b	d	e
Utah	-0.752446	-0.709925	-0.337916
Ohio	0.726541	2.137040	-2.288782
Texas	0.674664	-0.896596	2.181261
Oregon	0.885203	-1.067982	-1.518018

使用 `applymap` 作为函数名是因为 Series 有 `map` 方法，可以将一个逐元素的函数应用到 Series 上：

```
df['e'].map(f)
```

Utah	-0.337916
Ohio	-2.288782
Texas	2.181261
Oregon	-1.518018
Name: e, dtype: object	

排序和排名



如需按照行或列索引进行字典型排序，需要使用 `sort_index` 方法，其返回一个新的、排序好的对象：

```
s = pd.Series(range(4), index=list('dabc'))  
s.sort_index()
```

```
a    1  
b    2  
c    3  
d    0  
dtype: int64
```

在 DataFrame 中，可以在各个轴上按索引排序：

```
df = pd.DataFrame(  
    np.arange(8).reshape((2, 4)), index=['two', 'one'],  
    columns=['d', 'a', 'b', 'c'])  
df.sort_index()
```

```
      d  a  b  c  
one  4  5  6  7  
two  0  1  2  3
```

```
df.sort_index(axis=1)
```

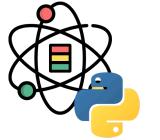
```
      a  b  c  d  
two  1  2  3  0  
one  5  6  7  4
```

数据默认会使用升序排序，也可以使用降序：

```
df.sort_index(axis=1, ascending=False)
```

```
      d  c  b  a  
two  0  3  2  1  
one  4  7  6  5
```

排序和排名



如果根据 Series 的值进行排序，可使用 `sort_values`：

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

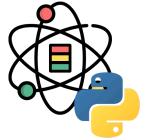
```
2    -3
3     2
0     4
1     7
dtype: int64
```

默认情况下，所有缺失值会排序到尾部：

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

```
4    -3.0
5     2.0
0     4.0
2     7.0
1     NaN
3     NaN
dtype: float64
```

排序和排名



对 DataFrame 排序时，可以使用一列或多列作为排序键，通过设置 `sort_values` 的参数 `by` 实现：

```
df = pd.DataFrame({'b': [7, 4, -3], 'a': [0, 1, 0]})  
df
```

```
   b  a  
0  7  0  
1  4  1  
2 -3  0
```

```
df.sort_values(  
    by='b')
```

```
   b  a  
2 -3  0  
1  4  1  
0  7  0
```

```
df.sort_values(  
    by=['a', 'b'])
```

```
   b  a  
2 -3  0  
0  7  0  
1  4  1
```

排名函数 `rank` 可以对数据点从 1 分配名次：

```
s = pd.Series([2, 1, 4, 2])
```

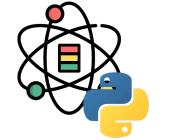
```
s.rank()
```

```
0    2.5  
1    1.0  
2    4.0  
3    2.5  
dtype: float64
```

```
s.rank(method='first')
```

```
0    2.0  
1    1.0  
2    4.0  
3    3.0  
dtype: float64
```

排序和排名



DataFrame 可以对行或列计算排名：

```
df = pd.DataFrame({  
    'b': [4.3, 7, -3, 2],  
    'a': [0, 1, 0, 1],  
    'c': [-2, 5, 8, -2.5]  
})
```

```
df
```

	b	a	c
0	4.3	0	-2.0
1	7.0	1	5.0
2	-3.0	0	8.0
3	2.0	1	-2.5

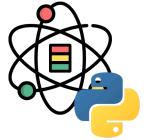
```
df.rank(axis='columns')
```

	b	a	c
0	3.0	2.0	1.0
1	3.0	1.0	2.0
2	1.0	2.0	3.0
3	3.0	2.0	1.0

在排名中打破评级关系的方法如下：

方法	描述
average	默认：在每个组中分配平均排名
min	对整个组使用最小排名
max	对整个组使用最大排名
first	按照值在数据中出现的次序分配排名
dense	类似 min，但排名总是增加 1 而非组元素数量

描述性统计



pandas 装配了一个常用数学、统计学方法的集合，与 NumPy 中类似方法相比，其内建了处理缺失值的功能。

```
df = pd.DataFrame(  
    [[1.4, np.nan], [7.1, -4.5], [np.nan, np.nan], [0.75, -1.3]],  
    index=['a', 'b', 'c', 'd'],  
    columns=['one', 'two']  
)  
df
```

```
  one  two  
a  1.40  NaN  
b  7.10 -4.5  
c  NaN  NaN  
d  0.75 -1.3
```

调用 `sum` 方法返回一个包含列上加和的 Series：

```
df.sum()
```

```
one    9.25  
two   -5.80  
dtype: float64
```

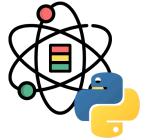
传入 `axis='columns'` 或 `axis=1` 则沿着行求和：

```
df.sum(axis='columns')
```

```
a    1.40  
b    2.60  
c    0.00  
d   -0.55  
dtype: float64
```

通过 `skipna=False` 可以不排除 NA 值。

描述性统计



`idxmax` 和 `idxmin` 返回的是间接统计信息，除了归约方法外，有的方法是积累型方法：

```
df.idxmax()
```

```
one    b
two    d
dtype: object
```

```
df.cumsum()
```

```
one  two
a    1.40  NaN
b    8.50 -4.5
c    NaN   NaN
d    9.25 -5.8
```

还有一类方法不是归约型方法也不是积累型方法，`describe`即其中之一，它一次性产生多个汇总统计：

```
df.describe()
```

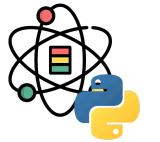
```
one      two
count  3.000000  2.000000
mean   3.083333 -2.900000
std    3.493685  2.262742
min    0.750000 -4.500000
25%   1.075000 -3.700000
50%   1.400000 -2.900000
75%   4.250000 -2.100000
max   7.100000 -1.300000
```

对于非数值类型数据，则产生另一种汇总统计：

```
pd.Series(['a', 'a', 'b', 'c'] * 4).describe()
```

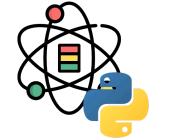
```
count      16
unique      3
top        a
freq       8
dtype: object
```

描述性统计



方法	描述	方法	描述
count	非 NA 值个数	describe	各列汇总值
min, max	计算最小/大值	argmin, argmax	计算最小/大值的索引位置
idxmin, idxmax	计算最小/大值的索引标签	quantile	计算样本的分位数
sum	加和	mean	均值
median	中位数	mad	平均值的平均绝对偏差
prod	连乘	var, std	样本方差/标准差
skew, kurt	偏度/峰度	cumsum	累加
cummin, cummax	累积值的最小/大值	cumprod	累积
diff	计算第一个算术差值	pct_change	计算百分比

相关性和协方差



```
import yfinance as yf
d = {ticker: yf.download(ticker, auto_adjust=False)
      for ticker in ['AAPL', 'IBM', 'MSFT']}
p = pd.DataFrame({ticker: data['Adj Close'][ticker]
                  for ticker, data in d.items()})
v = pd.DataFrame({ticker: data['Volume'][ticker]
                  for ticker, data in d.items()})
```

Series 的 corr 计算的是两个 Series 中的重叠的，非 NA 的，按索引对齐的值的相关性， cov 计算协方差：

```
p['MSFT'].corr(p['IBM'])
```

```
-0.0010077285210880053
```

```
p['MSFT'].cov(p['IBM'])
```

```
-0.02006614347919805
```

DataFrame 的 corr 和 cov 方法会分别以 DataFrame 形式返回相关性和协方差矩阵：

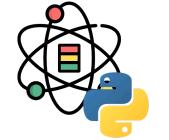
```
p.corr()
```

	AAPL	IBM	MSFT
AAPL	1.000000	0.232372	0.117309
IBM	0.232372	1.000000	-0.001008
MSFT	0.117309	-0.001008	1.000000

```
p.cov()
```

	AAPL	IBM	MSFT
AAPL	18.717995	3.629957	2.798935
IBM	3.629957	13.036918	-0.020066
MSFT	2.798935	-0.020066	30.413456

相关性和协方差



DataFrame 的 `corrwith` 方法可以计算出 DataFrame 行或列与另一个序列或 DataFrame 的相关性。当传入一个 Series 时，会返回一个含有为每列计算相关性值的 Series：

```
p.corrwith(p.IBM)
```

```
AAPL    0.232372
IBM    1.000000
MSFT   -0.001008
dtype: float64
```

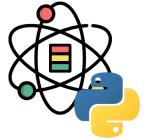
传入一个 DataFrame 时，会计算匹配到列名的相关性数值：

```
p.corrwith(v)
```

```
AAPL    -0.052422
IBM    -0.428928
MSFT   -0.251407
dtype: float64
```

传入 `axis='columns'` 会逐行地进行计算。

唯一值、计数和成员属性



unique 函数可以给出 Series 中的唯一值：

```
s = pd.Series(['c', 'a', 'd', 'a', 'a', 'b', 'b', 'c', 'c'])  
s.unique()
```

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

sort 方法可以对其进行排序，value_counts 可以计算 Series 包含的值的个数：

```
s.value_counts()
```

```
c    3  
a    3  
b    2  
d    1
```

```
Name: count, dtype: int64
```

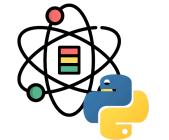
为了方便，返回的 Series 会按照数量降序排序，value_counts 也是有效的 pandas 顶层方法，也可以用于任意数组或序列：

```
pd.value_counts(s.values, sort=False)
```

```
c    3  
a    3  
d    1  
b    2
```

```
Name: count, dtype: int64
```

唯一值、计数和成员属性



`isin` 执行向量化的成员属性检查，还可以将数据集以 Series 或 DataFrame 一列的形式过滤为数据集的值子集：

```
s
```

```
0    c
1    a
2    d
3    a
4    a
5    b
6    b
7    c
8    c
dtype: object
```

```
s.isin(['b', 'c'])
```

```
0    True
1   False
2   False
3   False
4   False
5    True
6    True
7    True
8    True
dtype: bool
```

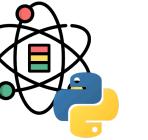
与 `isin` 相关的 `Index.get_indexer` 方法，可以提供一个索引数组，这个索引数组可以将可能非唯一值数组转换为另一个唯一值数组：

```
to_match = pd.Series(['c', 'a', 'b', 'b', 'c', 'a'])
unique_vals = pd.Series(['c', 'b', 'a'])
pd.Index(unique_vals).get_indexer(to_match)
```

```
array([0, 2, 1, 1, 0, 2])
```

数据载入和存储

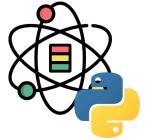
数据读取



将表格型数据读取为 DataFrame 对象是 pandas 的重要特性，下表为常用的数据读取函数：

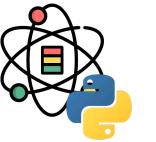
函数	描述
read_csv	从文件、URL 或文件型对象读取分隔好的数据，默认分隔符为 ,
read_table	从文件、URL 或文件型对象读取分隔符的数据，默认分隔符为 \t
read_fwf	从特定宽度格式的文件中读取数据（无分隔符）
read_clipboard	read_table 的剪切板版本
read_excel	从 Excel 的 XLS 或 XLSX 文件中读取表格数据
read_hdf	读取用 pandas 存储的 HDF5 文件
read_html	从 HTML 文件中读取所有表格数据

数据读取



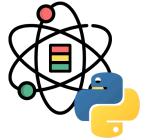
函数	描述
read_json	从 JSON 字符串中读取数据
read_msgpack	读取 MessagePackage 二进制格式的 pandas 数据
read_pickle	读取以 Python pickle 格式存储的任意对象
read_sas	读取存储在 SAS 系统中定制存储格式的 SAS 数据集
read_sql	将 SQL 查询的结果读取为 pandas 的 DataFrame
read_stata	读取 Stata 格式的数据集
read_feather	读取 Feather 二进制格式

数据读取



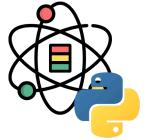
`read_csv` / `read_table` 的函数参数如下表所示：

参数	描述
<code>path</code>	表明文件系统位置的字符串, URL 或文件型对象
<code>sep, delimiter</code>	用于分割每行字段的字符序列或正则表达式
<code>header</code>	用作列名的行号, 默认是 0, 如果没有列名则为 <code>None</code>
<code>index_col</code>	用作结果中行索引的列号或列名, 可以是一个单一的名称/数字, 也可以是一个分层索引
<code>names</code>	结果的列名列表, 和 <code>header=None</code> 一起用
<code>skiprows</code>	从文件开头处起, 需要跳过的行数或行号列表
<code>comment</code>	在行结尾处分隔注释的字符



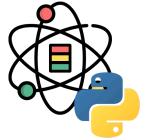
参数	描述
parse_dates	尝试将数据解析为 datetime, 默认是 False, 如果为 True, 将尝试解析所有的列。也可以指定列号或列名列表来进行解析。如果列表的元素是元组或列表, 将会把多个列组合在一起进行解析 (例如: 日期/时间将拆分为两列)
keep_date_col	如果连接列到解析日期上, 保留被连接的列, 默认是 False
converters	包含列名称映射到函数的字典 (例如: {'foo': f} 会把函数 f 应用到 foo 列)
dayfirst	解析非明确日期时, 按照国际格式处理 (例如: 7/6/2012 -> June 7, 2012), 默认为 False
date_parser	用于解析日期的函数

数据读取



参数	描述
<code>nrows</code>	从文件开头处理读入的行数
<code>iterator</code>	返回一个 <code>TextParser</code> 对象, 用于零散地读入文件
<code>chunksize</code>	用于迭代的块大小
<code>skip_footer</code>	忽略文件尾部的行数
<code>verbose</code>	打印各种解析器输出的信息, 比如位于非数值列中的缺失值数量
<code>encoding</code>	Unicode 文本编码 (例如: <code>utf-8</code> 用于表示 UTF-8 编码的文本)
<code>squeeze</code>	如果解析数据只包含一列, 返回一个 <code>Series</code>
<code>thousands</code>	千位分隔符 (例如: <code>,</code> 或 <code>.</code>)

数据写入



使用 DataFrame 的 `to_csv` 方法，可以将数据导出为逗号分隔符文件：

```
df = pd.DataFrame(  
    [['one', 'two', 'three'], [1, 2, 3], [3.0, np.nan, 6.6]],  
    columns=['a', 'b', 'c'])
```

```
df.to_csv('tmp.csv')  
# ,a,b,c  
# 0,one,two,three  
# 1,1,2,3  
# 2,3.0,,6.6
```

通过 `sep` 可以设置分隔符：

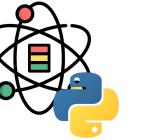
```
import sys  
df.to_csv(sys.stdout, sep='|')  
  
|a|b|c  
0|one|two|three  
1|1|2|3  
2|3.0||6.6
```

`na_rep` 可以对缺失值进行标注，`index` 和 `header` 用于控制是否显示行号和列名：

```
df.to_csv(sys.stdout, na_rep='NULL',  
          index=False, header=False)
```

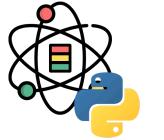
```
one,two,three  
1,2,3  
3.0,NULL,6.6
```

CSV 分隔格式



参数	描述
delimiter	一般用于分隔字段的字符， 默认是 ,
lineterminator	行终止符， 默认是 \r\n, 读取器会忽略行终止符并识别跨平台行终止符
quotechar	用在含有特殊字符字段中的引号， 默认是 "
quoting	引用惯例。选项包括 csv.QUOTE_ALL (引用所有的字段), csv.QUOTE_MINIMAL (只是用特殊字符, 如分隔符), csv.QUOTE_NONNUMERIC 和 csv.QUOTE_NONE (不引用), 默认为 QUOTE_MINIMAL。
skipinitialspace	忽略每个分隔符后的空白， 默认是 False
doublequote	如何处理字段内部的引号， 如果为 True, 则是双引号
escapechar	当引用设置为 csv.QUOTE_NONE 时用于转义分隔符的字符串， 默认是禁用的

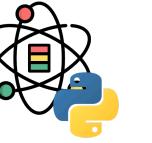
JSON 数据



JSON (JavaScript Object Notation) 是一种常用的数据格式，其基本类型包括：对象（字典）、数组（列表）、字符串、数字、布尔值和空值，对象中的所有键必须是字符串。在 Python 中可以利用内建的 `json` 模块处理 JSON。

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

JSON 数据



利用 `json.loads` 可以将 JSON 字符串转换为 Python 对象：

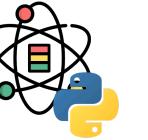
```
import json

res = json.loads(js)
res

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

利用 `json.dumps` 可以将 Python 对象装换回 JSON 字符串。

JSON 数据



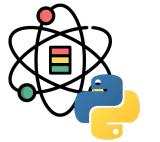
`pandas.read_json` 可以自动将 JSON 数据集按照指定次序转换为 Series 或 DataFrame, `example.json` 数据内容如下:

```
[  
  {"a": 1, "b": 2, "c": 3},  
  {"a": 4, "b": 5, "c": 6},  
  {"a": 7, "b": 8, "c": 9},  
]
```

```
df = pd.read_json('example.json')  
df  
#    a   b   c  
# 0  1   2   3  
# 1  4   5   6  
# 2  7   8   9
```

同样, 从 pandas 数据中导出 JSON 可以使用 Series 和 DataFrame 的 `to_json` 方法。

二进制数据



使用 Python 内建的 `pickle` 序列化模块进行二进制格式操作是存储数据（序列化）最高效，最方便的方法之一。pandas 对象拥有 `to_pickle` 方法可以将数据以 `pickle` 格式写入文件。使用 `read_pickle` 可以方便的读入 `pickle` 序列化后的文件。

HDF5 是一个备受好评的文件格式，用于存储大量的科学数组数据。每个 HDF5 文件可以存储多个数据集并支持元数据，与更简单的格式相比，HDF5 支持多种压缩模式的即时压缩，使得重复模式的数据可以更高效地存储。HDF5 适用于处理不适合在内存中存储的超大型数据，可以使你高效读写大型数组的一小块。

pandas 提供一个高阶的接口，可以简化 `Series` 和 `DataFrame` 的存储。`HDFStore` 类像字典一样工作并处理低级别细节：

```
df = pd.DataFrame({'a': np.random.randn(100)})
store = pd.HDFStore('tmp.h5')
store['obj1'] = df
store['obj1_col'] = df['a']
```

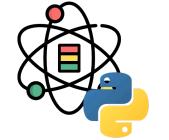
store

```
<class 'pandas.io.pytables.HDFStore'>
File path: tmp.h5
```

`to_hdf` 和 `read_hdf` 函数提供了便捷的操作方法。

数据规整

分层索引



分层索引是 pandas 的重要特性，允许在一个轴向上拥有多个索引层级。分层索引提供了一种在更低维度的形式中处理更高维度数据的方式。

```
s = pd.Series(  
    np.random.randn(9),  
    index=[  
        ['a', 'a', 'a',  
         'b', 'b', 'c',  
         'c', 'd', 'd'],  
        [1, 2, 3,  
         1, 3, 1,  
         2, 2, 3]  
    ]  
)
```

```
s  
a 1 0.637485  
   2 0.208254  
   3 0.073895  
b 1 -1.206255  
   3 1.515729  
c 1 -0.290793  
   2 0.378494  
d 2 0.465555  
   3 -0.787307  
dtype: float64
```

通过分层索引对象，允许简洁地选择出数据的子集：

```
s['b']
```

```
1 -1.206255  
3 1.515729  
dtype: float64
```

```
s['b':'c']
```

```
b 1 -1.206255  
   3 1.515729  
c 1 -0.290793  
   2 0.378494  
dtype: float64
```

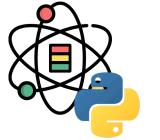
```
s.loc[['b', 'd']]
```

```
b 1 -1.206255  
3 1.515729  
d 2 0.465555  
3 -0.787307  
dtype: float64
```

```
s.loc[:, 2]
```

```
a 0.208254  
c 0.378494  
d 0.465555  
dtype: float64
```

分层索引



分层索引在重塑数据和数组透视表等分组操作中扮演重要角色，例如：使用 `unstack` 方法将数据在 DataFrame 中重新排列：

```
s.unstack()
```

```
1      2      3
a  0.637485  0.208254  0.073895
b -1.206255      NaN  1.515729
c -0.290793  0.378494      NaN
d      NaN  0.465555 -0.787307
```

`unstack` 的反操作是 `stack`

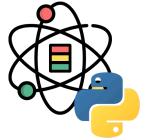
```
s.unstack().stack()
```

在 DataFrame 中，每个轴都可以拥有分层索引：

```
df = pd.DataFrame(
    np.arange(12).reshape((4, 3)),
    index=[['a', 'a', 'b', 'b'], [1, 2, 1, 2]],
    columns=[
        ['Ohio', 'Ohio', 'Colorado'],
        ['Green', 'Red', 'Green']
    ]
)
df
```

	Ohio	Colorado	
	Green	Red	Green
a	1	0	1
	2	3	4
b	1	6	7
	2	9	10
			11

分层索引



分层的层级可以有名称（可以是字符串或 Python 对象），如果层级有名称，这些名称会在控制台中显示：

```
df.index.names = ['key1', 'key2']
df.columns.names = ['state', 'color']
df
```

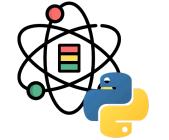
```
state      Ohio      Colorado
color      Green  Red      Green
key1  key2
a      1      0      1      2
      2      3      4      5
b      1      6      7      8
      2      9     10     11
```

通过部分列索引，可以选出列中的组：

```
df['Ohio']
```

	color	Green	Red
	key1	key2	
a	1	0	1
	2	3	4
b	1	6	7
	2	9	10

重排序和层级排序



有时需要重新排列轴上的层级顺序，或者按照特定层级的值对数据进行排序。`swaplevel` 接受两个层级序号或层级名称，返回一个进行了层级变更的新对象：

```
df.swaplevel('key1', 'key2')
```

	state	Ohio	Colorado	
	color	Green	Red	
	key2	key1		
1	a	0	1	2
2	a	3	4	5
1	b	6	7	8
2	b	9	10	11

`sort_index` 只能在单一层级上对数据进行排序，在进行层级变换时，使用 `sort_index` 以使得结果按照层级进行字典排序：

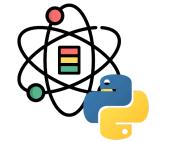
```
df.sort_index(level=1)
```

	state	Ohio	Colorado	
	color	Green	Red	
	key1	key2		
a	1	0	1	2
b	1	6	7	8
a	2	3	4	5
b	2	9	10	11

```
df.swaplevel(0, 1).sort_index(level=0)
```

	state	Ohio	Colorado	
	color	Green	Red	
	key2	key1		
1	a	0	1	2
	b	6	7	8
2	a	3	4	5
	b	9	10	11

按层级进行汇总统计



DataFrame 和 Series 中很多统计性和汇总性统计可以通过 groupby 函数和 level 参数可以指定想要在某个特定的轴上进行聚合，按照层级在行上进行聚合：

```
df.groupby(level='key2').sum()
```

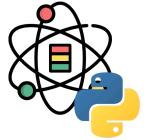
	state	Ohio	Colorado
color	Green	Red	Green
key2			
1		6	8
2		12	14

按照层级在列上进行聚合：

```
df.groupby(level='color', axis=1).sum()
```

color	Green	Red
key1	key2	
a	1	2
	2	8
b	1	14
	2	20

使用列进行索引



DataFrame 的 `set_index` 会生成一个新的 DataFrame, 新的 DataFrame 使用一个或多个列作为索引:

```
df = pd.DataFrame({  
    'a': range(7), 'b': range(7, 0, -1),  
    'c': ['one', 'one', 'one', 'two', 'two', 'two'],  
    'd': [0, 1, 2, 0, 1, 2, 3]})
```

```
df
```

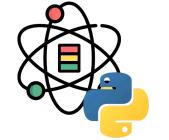
```
   a   b   c   d  
0  0   7  one  0  
1  1   6  one  1  
2  2   5  one  2  
3  3   4  two  0  
4  4   3  two  1  
5  5   2  two  2  
6  6   1  two  3
```

```
df.set_index(['c', 'd'], drop=False)
```

	a	b	c	d	
c	d				
one	0	0	7	one	0
	1	1	6	one	1
	2	2	5	one	2
two	0	3	4	two	0
	1	4	3	two	1
	2	5	2	two	2
	3	6	1	two	3

默认情况下这些列会从 DataFrame 中移除, 通过 `drop=False` 可以将其保留在 DataFrame 中。

数据库风格的连接



合并或连接操作通过一个或多个键来联合数据集，这些操作是关系型数据库的核心内容。pandas 中的 `merge` 函数主要用于将各种 `join` 操作应用到数据上：

```
df1 = pd.DataFrame({  
    'key': ['b', 'b', 'a', 'c', 'a', 'a', 'b'],  
    'data1': range(7)})
```

```
df1
```

```
key  data1  
0    b      0  
1    b      1  
2    a      2  
3    c      3
```

```
df2 = pd.DataFrame({  
    'key': ['a', 'b', 'd'],  
    'data2': range(3)  
})
```

```
key  data2  
0    a      0  
1    b      1  
2    d      2
```

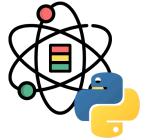
`df1` 的数据多个行的标签为 `a` 和 `b`，而 `df2` 在 `key` 列中每个值仅有一行。调用 `merge` 处理获取连接的对象：

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

	key	data1	data2
0	b	0	1
1	b	1	1
2	a	2	0
3	a	4	0
4	a	5	0
5	b	6	1

上例中并没有指定在哪一列上进行连接，如果没有指定 `merge` 会自动将重叠的列名作为连接的键。

数据库风格的连接



可以显式地指定连接键:

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

	key	data1	data2
0	b	0	1
1	b	1	1
2	a	2	0
3	a	4	0
4	a	5	0
5	b	6	1

如果对象的列名是不同的，可以分别为他们指定列名:

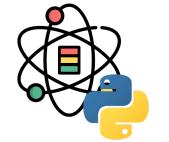
```
df3 = pd.DataFrame({  
    'lkey': ['b', 'b', 'a', 'c', 'a', 'a', 'b'],  
    'data1': range(7)  
})
```

```
df4 = pd.DataFrame({  
    'rkey': ['a', 'b', 'd'],  
    'data2': range(3)  
})  
pd.merge(df3, df4, left_on='lkey', right_on='rkey')
```

	lkey	data1	rkey	data2
0	b	0	b	1
1	b	1	b	1
2	a	2	a	0
3	a	4	a	0
4	a	5	a	0
5	b	6	b	1

默认情况下 `merge` 做的是内连接 `inner`，其他可选项有左连接 `left`、右连接 `right` 和外连接 `outer`。

数据库风格的连接



```
left = pd.DataFrame({  
    'key1': ['foo', 'foo', 'bar'],  
    'key2': ['one', 'two', 'one'], 'lval': [1, 2, 3]  
})  
right = pd.DataFrame({  
    'key1': ['foo', 'foo', 'bar', 'bar'],  
    'key2': ['one', 'one', 'one', 'two'],  
    'rval': [4, 5, 6, 7]  
})  
pd.merge(left, right, on=['key1', 'key2'], how='outer')
```

	key1	key2	lval	rval
0	bar	one	3.0	6.0
1	bar	two	NaN	7.0
2	foo	one	1.0	4.0
3	foo	one	1.0	5.0
4	foo	two	2.0	NaN

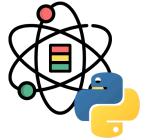
```
pd.merge(left, right, on='key1')
```

	key1	key2_x	lval	key2_y	rval
0	foo	one	1	one	4
1	foo	one	1	one	5
2	foo	two	2	one	4
3	foo	two	2	one	5
4	bar	one	3	one	6
5	bar	one	3	two	7

```
pd.merge(left, right, on='key1', suffixes=('_l', '_r'))
```

	key1	key2_l	lval	key2_r	rval
0	foo	one	1	one	4
1	foo	one	1	one	5
2	foo	two	2	one	4
3	foo	two	2	one	5
4	bar	one	3	one	6
5	bar	one	3	two	7

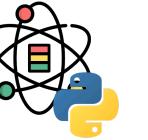
数据库风格的连接



`merge` 函数的参数如下表所示：

参数	描述
<code>left</code>	合并时操作中左边的 DataFrame
<code>right</code>	合并时操作中右边的 DataFrame
<code>how</code>	<code>inner</code> , <code>outer</code> , <code>left</code> , <code>right</code> 之一, 默认为 <code>inner</code>
<code>on</code>	需要连接的列名, 需要在两边的 DataFrame 中都存在。
<code>left_on</code>	左 DataFrame 中用作连接的键
<code>right_on</code>	右 DataFrame 中用作连接的键

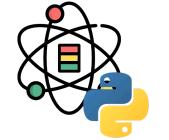
数据库风格的连接



`merge` 函数的参数如下表所示：

参数	描述
<code>left_index</code>	使用 <code>left</code> 的行索引作为它的连接键（如果是 MultiIndex，则为多个键）
<code>right_index</code>	使用 <code>right</code> 的行索引作为它的连接键（如果是 MultiIndex，则为多个键）
<code>sort</code>	通过连接键按字母顺序对合并的数据进行排序， 默认为 <code>True</code>
<code>suffixes</code>	在重叠情况下， 添加到列名后的字符串元组， 默认为 <code>('_x', '_y')</code>
<code>copy</code>	如果为 <code>False</code> ， 则在某些特殊情况下避免将数据复制到结果数据结构中， 默认情况下总是复制
<code>indicator</code>	添加一个特殊的列 <code>_merge</code> ， 指示每一行的来源。值将根据每行中连接数据的来源分别为 <code>left_only</code> , <code>right_only</code> , <code>both</code>

数据库风格的连接



DataFrame 有一个方便的 `join` 实例方法，用于按照索引合并。该方法也可以用于合并多个索引相同或相似但没有重叠列的 DataFrame 对象：

```
left = pd.DataFrame(  
    [[1., 2.], [3., 4.], [5., 6.]],  
    index=['a', 'c', 'e'], columns=['Ohio', 'Nevada'])  
right = pd.DataFrame(  
    [[7., 8.], [9., 10.], [11., 12.], [13., 14.]],  
    index=['b', 'c', 'd', 'e'],  
    columns=['Missouri', 'Alabama'])  
left.join(right, how='outer')
```

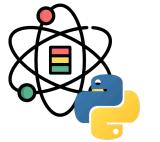
	Ohio	Nevada	Missouri	Alabama
a	1.0	2.0	NaN	NaN
b	NaN	NaN	7.0	8.0
c	3.0	4.0	9.0	10.0
d	NaN	NaN	11.0	12.0
e	5.0	6.0	13.0	14.0

可以向 `join` 方法传递一个 DataFrame 列表：

```
another = pd.DataFrame(  
    [[7.], [8.], [9.], [10.]],  
    index=['a', 'c', 'e', 'f'],  
    columns=['New York'])  
left.join([right, another], how='outer')
```

	Ohio	Nevada	Missouri	Alabama	New York
a	1.0	2.0	NaN	NaN	7.0
c	3.0	4.0	9.0	10.0	8.0
e	5.0	6.0	13.0	14.0	9.0
b	NaN	NaN	7.0	8.0	NaN
d	NaN	NaN	11.0	12.0	NaN
f	NaN	NaN	NaN	NaN	10.0

沿轴向连接



另一种数据组合操作可互换地称为拼接、绑定或堆叠。NumPy 的 `concatenate` 函数可以在 NumPy 数组上实现该功能：

```
arr = np.arange(12).reshape((3, 4))  
arr
```

```
array([[ 0,  1,  2,  3],  
       [ 4,  5,  6,  7],  
       [ 8,  9, 10, 11]])
```

```
np.concatenate([arr, arr], axis=1)
```

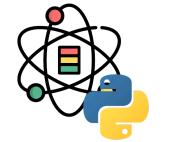
```
array([[ 0,  1,  2,  3,  0,  1,  2,  3],  
       [ 4,  5,  6,  7,  4,  5,  6,  7],  
       [ 8,  9, 10, 11,  8,  9, 10, 11]])
```

在 Series 和 DataFrame 等 pandas 对象上下文中，使用标记的轴可以进一步泛化数据连接。尤其是你还有需要考虑的事情：

- 如果对象在其他轴上的索引不同，我们是否应该将不同的元素组合在这些轴上，还是只使用共享的值（交集）？
- 连接的数据块是否需要在结果对象中被识别？
- “连接轴”是否包含需要保存的数据？

pandas 的 `concat` 函数提供了一种一致性的方式来解决以上问题。

沿轴向连接



```
s1 = pd.Series([0, 1], index=['a', 'b'])  
s2 = pd.Series([2, 3, 4], index=['c', 'd', 'e'])  
s3 = pd.Series([5, 6], index=['f', 'g'])
```

调用 concat 方法会将值和索引连在一起：

```
pd.concat([s1, s2, s3])
```

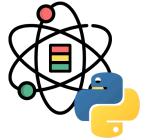
```
a    0  
b    1  
c    2  
d    3  
e    4  
f    5  
g    6  
dtype: int64
```

默认情况下， concat 方法是沿着 axis=0 的轴向生效的，生成另一个 Series。如果设置 axis=1，返回的结果则是一个 DataFrame：

```
pd.concat([s1, s2, s3], axis=1)
```

	0	1	2
a	0.0	NaN	NaN
b	1.0	NaN	NaN
c	NaN	2.0	NaN
d	NaN	3.0	NaN
e	NaN	4.0	NaN
f	NaN	NaN	5.0
g	NaN	NaN	6.0

沿轴向连接



拼接在一起的各部分无法在结果中区分是一个潜在的问题。假设你想在连接轴向上创建一个多层次索引，可以使用 `keys` 参数来实现：

```
res = pd.concat([s1, s1, s3], keys=['one', 'two', 'three'])  
res
```

```
one    a    0  
      b    1  
two    a    0  
      b    1  
three   f    5  
      g    6  
dtype: int64
```

```
res.unstack()
```

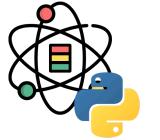
	a	b	f	g
one	0.0	1.0	NaN	NaN
two	0.0	1.0	NaN	NaN
three	NaN	NaN	5.0	6.0

`axis=1` 连接 Series 时，`keys` 为 DataFrame 的列头：

```
pd.concat([s1, s2], axis=1, keys=['one', 'two'])
```

	one	two
a	0.0	NaN
b	1.0	NaN
c	NaN	2.0
d	NaN	3.0
e	NaN	4.0

沿轴向连接



对于 DataFrame 对象：

```
df1 = pd.DataFrame(  
    np.arange(6).reshape((3, 2)),  
    index=['a', 'b', 'c'], columns=['one', 'two'])  
df2 = pd.DataFrame(  
    np.arange(4).reshape((2, 2)),  
    index=['a', 'c'], columns=['three', 'four'])
```

df1

	one	two
a	0	1
b	2	3
c	4	5

df2

	three	four
a	0	1
c	2	3

```
pd.concat([df1, df2], axis=1, keys=['11', '12'])
```

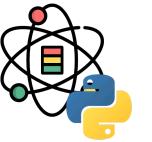
	11	12		
	one	two	three	four
a	0	1	0.0	1.0
b	2	3	NaN	NaN
c	4	5	2.0	3.0

如果传递的是对象的字典而不是列表的话，则字典的键会用于 keys 选项：

```
pd.concat({'11': df1, '12': df2}, axis=1)
```

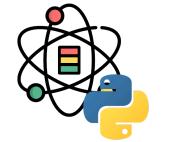
	11	12		
	one	two	three	four
a	0	1	0.0	1.0
b	2	3	NaN	NaN
c	4	5	2.0	3.0

沿轴向连接



参数	描述
objs	需要连接的 pandas 对象列表或字典
axis	连接的轴向, 默认是 0 (沿着行方向)
join	用于指定连接方式 (inner 或 outer)
keys	与要连接的对象关联的值, 沿着连接轴形成分层索引。
levels	在键值传递时, 该参数用于指定多层索引的层级
names	传入 keys 或 levels 时用于多层索引的层级名称
verify_integrity	检查连接对象中的新轴是否重复, 如果是, 则发生异常, 默认为 False
ignore_index	不沿着连接轴保留索引, 而产生一段新的索引

联合重叠数据



另一种数据联合场景，既不是合并，也不是连接。两个数据集的索引全部或部分重叠，考虑 NumPy 的 `where` 函数，这个函数可以进行面向数组的 if-else 等价操作：

```
a = pd.Series([np.nan, 2.5, 0.0, np.nan], index=['d', 'c', 'b', 'a'])  
b = pd.Series([0, np.nan, 2, 3], index=['a', 'b', 'c', 'd'])
```

```
a  
d    NaN  
c    2.5  
b    0.0  
a    NaN  
dtype: float64
```

```
b  
a    0.0  
b    NaN  
c    2.0  
d    3.0  
dtype: float64
```

```
np.where(pd.isnull(a), b, a)
```

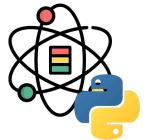
```
array([0. , 2.5, 0. , 3. ])
```

Series 的 `combine_first` 方法等基于上述操作：

```
b.combine_first(a)
```

```
a    0.0  
b    0.0  
c    2.0  
d    3.0  
dtype: float64
```

联合重叠数据



在 DataFrame 中, `combine_first` 逐列进行相同的操作, 可以理解为利用传入的数值替换对象的缺失值:

```
df1 = pd.DataFrame({
    'a': [1., np.nan, 5., np.nan],
    'b': [np.nan, 2., np.nan, 6.],
    'c': range(2, 18, 4)
})
df2 = pd.DataFrame({
    'a': [5., 4., np.nan, 3., 7.],
    'b': [np.nan, 3., 4., 6., 8.]
})
```

df1

	a	b	c
0	1.0	NaN	2
1	NaN	2.0	6
2	5.0	NaN	10
3	NaN	6.0	14

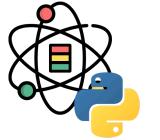
df2

	a	b
0	5.0	NaN
1	4.0	3.0
2	NaN	4.0
3	3.0	6.0
4	7.0	8.0

df1.combine_first(df2)

	a	b	c
0	1.0	NaN	2.0
1	4.0	2.0	6.0
2	5.0	4.0	10.0
3	3.0	6.0	14.0
4	7.0	8.0	NaN

重塑和透视



多层索引在 DataFrame 中提供了一种一致性方式用于重排列数据，包含两个基础操作：

- stack (堆叠), “旋转”或将列中的数据透视到行
- unstack (拆堆), 将行中的数据透视到列

```
df = pd.DataFrame(  
    np.arange(6).reshape((2, 3)),  
    index=pd.Index(['Ohio', 'Colorado'], name='state'),  
    columns=pd.Index(['one', 'two', 'three'], name='num'))  
df
```

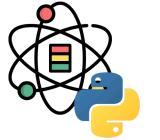
```
  num    one  two  three  
state  
Ohio      0    1    2  
Colorado  3    4    5
```

利用 stack 方法将数据透视到行，产生新的 Series：

```
res = df.stack()  
res
```

```
state      num  
Ohio      one      0  
          two      1  
          three     2  
Colorado  one      3  
          two      4  
          three     5  
dtype: int64
```

重塑和透视



如果层级中的所有值并未包含于每个子分组中，拆分可能会引入缺失值：

```
s1 = pd.Series([0, 1, 2, 3], index=['a', 'b', 'c', 'd'])
s2 = pd.Series([4, 5, 6], index=['c', 'd', 'e'])
df = pd.concat([s1, s2], keys=['one', 'two'])
df
```

```
one  a    0
      b    1
      c    2
      d    3
two  c    4
      d    5
      e    6
dtype: int64
```

默认情况下，堆叠会过滤出缺失值：

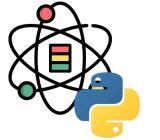
`df.unstack()`

```
          a    b    c    d    e
one  0.0  1.0  2.0  3.0  NaN
two  NaN  NaN  4.0  5.0  6.0
```

`df.unstack().stack()`

```
one  a    0.0
      b    1.0
      c    2.0
      d    3.0
two  c    4.0
      d    5.0
      e    6.0
dtype: float64
```

重塑和透视



在 DataFrame 中拆堆时，被拆堆的层级会变为结果中最低的层级：

```
df = pd.DataFrame(  
    {'left': res, 'right': res + 5},  
    columns=pd.Index(['left', 'right'], name='side'))  
df
```

side	left	right	
state	num		
Ohio	one	0	5
	two	1	6
	three	2	7
Colorado	one	3	8
	two	4	9
	three	5	10

调用 stack 方法时，可以指明需要堆叠的轴向名称：

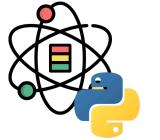
```
df.unstack('state')
```

side	left	right		
state	Ohio	Colorado	Ohio	Colorado
num				
one	0	3	5	8
two	1	4	6	9
three	2	5	7	10

```
df.unstack('state').stack('side')
```

state	Ohio	Colorado	
num	side		
one	left	0	3
	right	5	8
two	left	1	4
	right	6	9
three	left	2	5
	right	7	10

重塑和透视



从一个多层次索引序列中，可以使用 `unstack` 方法将数据重排列后放入一个 DataFrame 中：

```
res.unstack()
```

	num	one	two	three
state				
Ohio	0	1	2	
Colorado	3	4	5	

默认情况下，最内层是已拆堆的（与 `stack` 方法一样），传入层级序号或名称来拆分一个不同的层级：

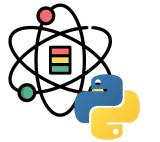
```
res.unstack(0)
```

	state	Ohio	Colorado
num			
one	0	3	
two	1	4	
three	2	5	

```
res.unstack('state')
```

	state	Ohio	Colorado
num			
one	0	3	
two	1	4	
three	2	5	

重塑和透视



DataFrame 中的 `pivot` 和 `melt` 方法提供了将数据“由长到宽”和“由宽到长”变换的支持。

```
df = pd.DataFrame({  
    'key': ['foo', 'bar', 'baz'] * 3,  
    'var': ['A', 'A', 'A', 'B', 'B', 'B', 'C', 'C', 'C'],  
    'val': range(9),  
})  
df
```

	key	var	val
0	foo	A	0
1	bar	A	1
2	baz	A	2
3	foo	B	3
4	bar	B	4
5	baz	B	5
6	foo	C	6
7	bar	C	7
8	baz	C	8

```
pivoted = df.pivot(  
    index='key', columns='var', values='val'  
).reset_index()
```

pivoted

var	key	A	B	C
0	bar	1	4	7
1	baz	2	5	8
2	foo	0	3	6

```
pd.melt(pivoted, ['key'])
```

key	var	value	
0	bar	A	1
1	baz	A	2
2	foo	A	0
3	bar	B	4
4	baz	B	5
5	foo	B	3
6	bar	C	7
7	baz	C	8
8	foo	C	6

感谢倾听

本作品采用  授权

版权所有 © 范叶亮 Leo Van