Pandas

—简介

pandas含有使数据分析工作变得更快更简单的高级数据结构和操作工具。它是基于NumPy构建的，让以NumPy为中心的应用变得更加简单。

这节是对pandas的一个简单的介绍，详细的介绍请参考：*Cookbook* 。

http://pandas.pydata.org/pandas-docs/stable/cookbook.html#cookbook

习惯上，会按下面格式引入所需要的包：

```
>>> import pandas as pd
>>> import numpy as np
>>> import matplotlib.pyplot as plt
```

一、创建对象

Series

可以通过传递一个list对象来创建一个Series，pandas会默认创建整型索引：

```
>>>s = pd.Series([1,3,5,np.nan,6,8])

>>> s
0    1
1    3
2    5
3  NaN
4    6
5    8
dtype: float64
```

一、创建对象

通常希望所创建的Series带有一个可以对各个数据点进行标记的索引：

```
>>>s = pd.Series([1,3,5,np.nan,6,8],index=['a','b','c','d','e','f'])
>>>s
a    1
b    3
c    5
d  NaN
e    6
f    8
dtype: float64
```

到目前为止，可能觉得 Series 对象和一维 NumPy 数组基本可以等价交换，但两者间的本本质差异其实是索引： NumPy 数组通过隐式定

一、创建对象

义的整数索引获取数值，而 Pandas 的 Series 对象用一种显式定义的索引与数值关联。

显式索引的定义让 Series 对象拥有了更强的能力。例如，索引不再仅仅是整数，还可以是任意想要的类型。如果需要，完全可以用字符串定义索引，也可以使用不连续或不按顺序的索引：

```
>>> data = pd.Series([0.25, 0.5, 0.75, 1.0],index=[2, 5, 3, 7])
>>> data
2    0.25
5    0.50
3    0.75
7    1.00
dtype: float64
```

一、创建对象

可以把 Pandas 的 Series 对象看成一种特殊的 Python 字典：

```
>>> population_dict = {'California': 38332521,
'Texas': 26448193,
'New York': 19651127,
'Florida': 19552860,
'Illinois': 12882135}
>>> population = pd.Series(population_dict)
>>> population
California    38332521
Florida       19552860
Illinois      12882135
New York      19651127
Texas         26448193
dtype: int64
```

一、创建对象

```
>>> population['California']
38332521

>>> population['California':'Illinois']
California    38332521
Florida       19552860
Illinois      12882135
dtype: int64
```

一、创建对象

DataFrame

DataFrame是一个表格型的数据结构，它含有一组有序的列，每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，它可以被看做由Series组成的字典（共用同一个索引）。跟其他类似的数据结构相比（如R），DataFrame中面向行和面向列的操作基本上是平衡的。其实，DataFrame中的数据是以一个或多个二维块存放的（而不是列表、字典或别的一维数据结构）。

一、创建对象

通过传递一个numpy array，时间索引以及列标签来创建一个DataFrame：

```
>>>dates = pd.date_range('20130101', periods=6)

>>> dates
DatetimeIndex(['2013-01-01', '2013-01-02', '2013-01-03',
'2013-01-04',
               '2013-01-05', '2013-01-06'],
              dtype='datetime64[ns]', freq='D', tz=None)

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

>>> df
```

一、创建对象

通过传递一个能够被转换成类似序列结构的字典对象来创建一个DataFrame：

```
>>> df = pd.DataFrame({ 'A' : 1.,
                        'B' : pd.Timestamp('20130102'),
                        'C' :
pd.Series(1,index=list(range(4)),dtype='float32'),
                        'D' : np.array([3] * 4,dtype='int32'),
                        'E' :
pd.Categorical(["test","train","test","train"]),
                        'F' : 'foo' })

>>> df
```

一、创建对象

查看不同列的数据类型：

```
>>> df.dtypes
A          float64
B    datetime64[ns]
C          float32
D            int32
E         category
F           object
dtype: object
```

如果使用的是IPython，使用Tab自动补全功能会自动识别所有的属性以及自定义的列，下图中是所有能够被自动识别的属性的一个子集：

```
>>> df.<TAB>
```

一、创建对象

Pandas的Index对象

Series 和 DataFrame 对象都使用便于引用和调整的显式索引。Pandas 的 Index 对象是一个很有趣的数据结构，可以将它看作是一个不可变数组或有序集合：

```
>>> ind = pd.Index([2, 3, 5, 7, 11])
>>> ind
Int64Index([2, 3, 5, 7, 11], dtype='int64')
>>> ind[1]
3
>>> ind[::2]
Int64Index([2, 5, 11], dtype='int64')
>>> print(ind.size, ind.shape, ind.ndim, ind.dtype)
(5, (5L,), 1, dtype('int64'))
```

一、创建对象

Index 对象的不可变特征使得多个 DataFrame 和数组之间进行索引共享时更加安全，尤其是可以避免因修改索引时粗心大意而导致的副作用。

```
>>> ind[1] = 0
```

Pandas 对象被设计用于实现许多操作，如连接（join）数据集，其中会涉及许多集合操作。 Index 对象遵循 Python 标准库的集合（set）数据结构的许多习惯用法，包括并集、交集、差集等：

一、创建对象

```
>>> indA = pd.Index([1, 3, 5, 7, 9])
ind = pd.Index([2, 3, 5, 7, 11])

>>> indA & ind          # 交集
Int64Index([3, 5, 7], dtype='int64')

>>> indA | ind          # 并集
Int64Index([1, 2, 3, 5, 7, 9, 11], dtype='int64')

>>> indA ^ ind          # 异或
Int64Index([1, 2, 9, 11], dtype='int64‘)

# 这些操作还可以通过调用对象方法来实现，例如
>>> indA.intersection(ind)
Int64Index([3, 5, 7], dtype='int64')
```

Index的方法和属性 #index. index.is

方法	说明
append	连接另一个Index对象，产生一个新的Index
diff	计算差集，并得到一个Index
intersection	计算交集
union	计算并集
isin	计算一个指示各值是否都包含在参数集合中的布尔型数组
delete	删除索引i处的元素，并得到新的Index
drop	删除传入的值，并得到新的Index
insert	将元素插入到索引i处，并得到新的Index
is_monotonic	当各元素均大于等于前一个元素时，返回True
is.unique	当Index没有重复值时，返回True
unique	计算Index中唯一值的数组

二、查看数据

1、查看frame中头部和尾部的行：

```
>>> df.head()

>>> df.tail(3)
```

2、显示索引、列和底层的numpy数据：

```
>>> df.index

DatetimeIndex(['2013-01-01', '2013-01-02',
'2013-01-03', '2013-01-04',
               '2013-01-05', '2013-01-06'],
              dtype='datetime64[ns]', freq='D',
tz=None)
```

二、查看数据

```
>>>df.columns
Index([u'A', u'B', u'C', u'D'], dtype='object')

>>> df.values
array([[-1.02712343, -0.01454347,  0.36682575, -
0.57462714],
       [-0.91022264, -0.43360699,  1.34085819, -0.62465058],
       [-0.55523497, -1.76107749, -0.21041818, -0.54550316],
       [-3.02961207, -0.64198084, -0.06241634, -0.50250544],
       [ 0.50100655,  1.38876526,  1.16139828,  1.06322606],
       [-0.05581814, -0.06085931, -0.61268583, -0.43812153]])
```

二、查看数据

3、 describe()函数对于数据的快速统计汇总：

```
>>>  df.describe()
```

4、 对数据的转置：

```
>>>  df.T
```

5、 按轴进行排序

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

6、 按值进行排序

```
>>> df.sort_index(by='B')
```

三、选择

虽然标准的Python/Numpy的选择和设置表达式都能够直接派上用场，但是作为工程使用的代码，推荐使用经过优化的pandas数据访问方式： .at, .iat, .loc, .iloc 和 .ix

1、 获取

选择一个单独的列，这将会返回一个Series，等同于df.A：

三、选择

```
>>>  df['A']
2013-01-01  -1.027123
2013-01-02  -0.910223
2013-01-03  -0.555235
2013-01-04  -3.029612
2013-01-05   0.501007
2013-01-06  -0.055818
Freq: D, Name: A, dtype: float64
```

通过[]进行选择，这将会对行进行切片

```
>>> df[0:3]

>>> df['20130102':'20130104']
```

三、选择

2、 通过标签选择

使用标签来获取一个交叉的区域

```
>>>  df.loc[dates[0]]
A   -1.027123
B   -0.014543
C    0.366826
D   -0.574627
Name: 2013-01-01 00:00:00, dtype: float64
```

通过标签来在多个轴上进行选择

```
>>> df.loc[:,['A','B']]
```

三、选择

标签切片

```
>>> df.loc['20130102':'20130104',['A','B']]
```

对于返回的对象进行维度缩减

```
>>>df.loc['20130102',['A','B']]
A   -0.910223
B   -0.433607
Name: 2013-01-02 00:00:00, dtype: float64
```

获取一个标量（ 快速访问一个标量）

```
>>> df.loc[dates[0],'A']
-1.0271234279484225

>>> df.at[dates[0],'A']
-1.0271234279484225
```

三、选择

3、 通过位置选择

通过传递数值进行位置选择（选择的是行）

```
>>> df.iloc[3]
A   -3.029612
B   -0.641981
C   -0.062416
D   -0.502505
Name: 2013-01-04 00:00:00, dtype: float64
```

通过数值进行切片，与numpy/python中的情况类似

```
>>>  df.iloc[3:5,0:2]
```

三、选择

通过指定一个位置的列表，与numpy/python中的情况类似

```
>>> df.iloc[[1,2,4],[0,2]]
```

对行进行（列）切片

```
>>> df.iloc[1:3,:]

>>> df.iloc[:,1:3]
```

获取特定的值

```
>>> df.iloc[1,1]
-0.4336069868405485 5

>>> df.iat[1,1]
-0.4336069868405485 5
```

三、选择

4、 布尔索引

使用一个单独列的值来选择数据：

```
>>> df[df.A > 0]
```

使用where操作来选择数据：

```
>>> df[df > 0]    # df.where(df>0)
```

使用isin()方法来过滤：

```
>>> df2 = df.copy()

>>> df2['E'] = ['one', 'one','two','three','four','three']

>>> df2

>>> df2[df2['E'].isin(['two','four'])]
```

三、选择

5、设置

设置一个新的列：

```
>>> s1 = pd.Series([1,2,3,4,5,6],
index=pd.date_range('20130102', periods=6))

>>>  s1
2013-01-02   1
2013-01-03   2
2013-01-04   3
2013-01-05   4
2013-01-06   5
2013-01-07   6
Freq: D, dtype: int64

>>>  df['F'] = s1
```

三、选择

通过标签设置新的值：

```
>>> df.at[dates[0],'A'] = 0
```

通过位置设置新的值：

```
>>> df.iat[0,1] = 0
```

通过一个numpy数组设置一组新值：

```
>>> df.loc[:,'D'] = np.array([5] * len(df))

>>> df
```

通过where操作来设置新的值：

```
>>> df2 = df.copy()
>>> df2[df2 > 0] = -df2
>>> df2
>>> df
```

四、 缺失值处理

缺失数据（missing data)在大部分数据分析应用中都很常见。pandas的设计目标之一就是让缺失数据的处理任务尽量轻松。例如，pandas对象上的所有描述统计都排除了缺失数据。

pandas使用NaN (Not a Number)表示浮点和非浮点数组中的缺失数据。它只是一 个便于被检测出来的标记而已：

```
>>>
string_data=pd.Series(['aardvark','artichoke',np.nan,'avocado'])
```

四、 缺失值处理

```
>>> string_data
0     aardvark
1    artichoke
2          NaN
3      avocado
dtype: object
```

```
>>> string_data.isnull()
0    False
1    False
2     True
3    False
dtype: bool
```

Python内置的None值也会被当做NA处理

```
>>> string_data[0] = None

>>>string_data.isnull()
0     True
1    False
2     True
3    False
dtype: bool
```

四、 缺失值处理

NA处理方法:

方法	说明
dropna	根据各标签的值中是否存在缺失数据对轴标签进行过滤，可通过阈值调节对缺失值的容忍度
fillna	用指定值或插值方法（如ffill或bfill)填充缺失数据
isnull	返回一个含有布尔值的对象，这些布尔值表示哪些值是缺失值/NA，该对象的类型与源类型一样
notnull	isnull 的否定式

四、 缺失值处理

1、滤除缺失数据

过滤掉缺失数据的办法有很多种。纯手工操作永远都是一个办法，但dropna可能会更实用一些。对于一个Series，dropna返回一个仅含非空数据和索引值的Series:

```
>>> from numpy import nan as NA
>>> from pandas import Series,DataFrame
>>> data = Series([1, NA, 3.5, NA, 7])
>>> data.dropna()
0    1.0
2    3.5
4    7.0
dtype: float64
```

四、 缺失值处理

当然，也可以通过布尔型索引达到这个目的:

```
>>>data[data.notnull()]
0    1.0
2    3.5
4    7.0
dtype: float64
```

对于DataFrame对象，dropna默认丢弃任何含有缺失值的行：

```
>>> data = DataFrame([[1., 6.5, 3.],[1., NA,NA],
      [NA, NA, NA], [NA, 6.5, 3.1]])

>>> cleaned = data.dropna()
>>> data
>>> cleaned
```

四、 缺失值处理

传入how='all'将只丢弃全为NA的那些行：

```
>>> data.dropna(how='all')
#要用这种方式丢弃列，只需传入axis=1即可:
>>> data[4] = NA
>>> data
>>> data.dropna(axis=1, how='all')
```

另一个滤除DataFrame行的问题涉及时间序列数据。假设只想留下一部分观测数据， 可以用thresh参数实现此目的：

```
>>> df = DataFrame(np.random.randn(7, 3))
>>> df.iloc[:4, 1] = NA; df.iloc[:2, 2] = NA

>>> df
>>> df.dropna(thresh=3)
```

四、 缺失值处理

2、填充缺失数据

若不想滤除缺失数据（有可能会丢弃跟它有关的其他数据），而是希望通过其他方式填补那些“空洞”。对于大多数情况而言，fillna方法是最主要的函数。通过一个常数调用fillna就会将缺失值替换为那个常数值：

```
>>> df.fillna(0)
```

若是通过一个字典调用fillna,就可以实现对不同的列填充不同的值

```
>>> df.fillna({1: 0.5, 2: -1})
```

四、 缺失值处理

fillna默认会返回新对象，但也可以对现有对象进行就地修改：

```
#总是返回被填允对象的引用
>>> ddf= df.fillna(0, inplace=True)
>>> df
```

对reindex有效的那些插值方法也可用于fillna：

```
>>> df = DataFrame(np.random.randn(6, 3))
>>> df.iloc[2:,1] = NA; df.iloc[4:,2] = NA

>>> df
>>> df.fillna(method='ffill')
>>> df.fillna(method='ffill', limit=2)
```

四、 缺失值处理

可以利用fillna实现许多别的功能。比如可以传入Series 的平均值或中位数：

```
>>> data = Series([1., NA, 3.5, NA, 7])

>>> data.fillna(data.mean())
0    1.000000
1    3.833333
2    3.500000
3    3.833333
4    7.000000
dtype: float64
```

四、 缺失值处理

fillna函数的参数：

参数	说明
value	用于填充缺失值的标量值或字典对象
method	插值方式。如果函数调用时未指定其他参数的话，默认为“ffill”
axis	待填充的轴，默认axis=0
inplace	修改调用者对象而不产生副本
limit	(对于前向和后向填充)可以连续填充的最大数量

五、 基本功能

1、算术运算和数据对齐

pandas最重要的一个功能是，它可以对不同索引的对象进行算术运算。在将对象相加时， 如果存在不同的索引对，则结果的索引就是该索引对的并集。

```
>>> s1 = Series([7.3, -2.5, 3.4, 1.5],index=['a', 'c', 'd', 'e'])

>>> s2 = Series([-2.1, 3.6, -1.5, 4, 3.1], index=['a', 'c', 'e', 'f', 'g'])

>>> s1

>>> s2
```

五、 基本功能

将它们相加就会产生：

```
>>>s1+s2
a    5.2
c    1.1
d    NaN
e    0.0
f    NaN
g    NaN
dtype: float64
```

自动的数据对齐操作在不重叠的索引处引入NA值。缺失值会在算术运算过程中传播。

五、 基本功能

对于DataFrame,对齐操作会同时发生在行和列上:

```
>>> df1 = DataFrame(np.arange(9,).reshape((3, 3)),
columns=list('bcd'),
        index=['Ohio', 'Texas', 'Colorado'])

>>> df2 = DataFrame(np.arange(12,).reshape((4, 3)),
columns=list('bde'),
        index=['Utah', 'Ohio', 'Texas', 'Oregon'])

>>> df1
>>> df2
>>> df1+ df2
#把它们相加后将会返回一个新的DataFrame，其索引和列为原来那两个DataFrame的并集.
```

五、 基本功能

在算术方法中填充值：

在对不同索引的对象进行算术运算时，可能希望当一个对象中某个轴标签在另一个对象中找不到时填充一个特殊值（比如0）：

```
>>> df1 = DataFrame(np.arange(12).reshape((3, 4)), columns=list('abcd'))

>>> df2 = DataFrame(np.arange(20.).reshape((4, 5)), columns=list('abcde'))

>>> df1
>>> df2
>>> df1 + df2
# 将它们相加时，没有重叠的位置就会产生NA值.
```

五、 基本功能

使用df1的add方法，传入df2以及一个fill_value参数：

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

与此类似，在对Series或DataFrame重新索引时，也可以指定一个填充值：

```
>>> df1.reindex(columns=df2.columns, fill_value=0)
```

灵活的算术方法

add	用于加法(+)的方法
sub	用于减法(-）的方法
div	用于除法(/)的方法
mul	用于乘法(*)的方法

五、 基本功能

DataFrame和Series之间的运算：

跟NumPy数组一样，DataFrame和Series之间算术运算也是有明确规定的。先来看一个具有启发性的例子，计算一个二维数组与其某行之间的差：

```
>>> arr = np.arange(12.).reshape((3, 4))
>>> arr
array([[  0.,   1.,   2.,   3.],
       [  4.,   5.,   6.,   7.],
       [  8.,   9.,  10.,  11.]])
>>> arr[0]
array([ 0.,  1.,  2.,  3.])
>>> arr - arr[0]
array([[ 0.,  0.,  0.,  0.],
       [ 4.,  4.,  4.,  4.],
       [ 8.,  8.,  8.,  8.]])
```

五、 基本功能

这就叫做广播（broadcasting)。DataFrame和Series之间的运算差不多也是如此：

```
>>> frame = DataFrame(np.arange(12.).reshape((4, 3)),
columns=list('bde'),
            index=['Utah', 'Ohio', 'Texas', 'Oregon'])

>>> series = frame.iloc[0]
>>> frame
>>> series
>>> frame - series
```

默认情况下，DataFrame和Series之间的算术运算会将Series的索引匹配到DataFrame的列，然后沿着行一直向下广播.

五、 基本功能

如果某个索引值在DataFrame的列或Series的索引中找不到，则参与运算的两个对象就会被重新索引以形成并集：

```
>>> series2 = Series(range(3), index=['b', 'e', 'f'])
>>> frame + series2
```

如果希望匹配行且在列上广播，则必须使用算术运算方法。例如：

```
>>> series3 = frame['d']
>>> series3
>>> frame.sub(series3,  axis=0)
```

传入的轴号就是希望匹配的轴。在本例中目的是匹配DateFrame的行索引并进行广播。

五、 基本功能

2、函数应用和映射

NumPy的ufuncs (元素级数组方法）也可用于操作pandas对象:

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

>>> frame

>>> np.abs(frame)
```

五、 基本功能

另一个常见的操作是，将函数应用到由各列或行所形成的一维数组上。DataFrame的apply方法即可实现此功能：

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

>>> frame.apply(f)

>>> frame.apply(f, axis=1)
```

许多最为常见的数组统计功能都被实现成DataFrame的方法（如sum和mean），因此无需使用apply方法。

五、 基本功能

除标量值外，传递给apply的函数还可以返回由多个值组成的Series:

```
>>>def f(x):
    return Series([x.min(), x.max()], index=['min',
'max'])

>>>frame

>>>frame.apply(f)
```

五、 基本功能

此外，元素级的Python函数也是可以用的。假如想得到frame中各个浮点值的格式化字符串，使用applymap即可：

```
>>> format = lambda x: '%.2f' % x

>>> frame.applymap(format)
```

之所以叫做applymap，是因为Series有一个用于应用元素级函数的map方法：

```
>>> frame['e'].map(format)
```

五、 基本功能

3、 统计与汇总（相关操作通常情况下不包括缺失值）

```
>>> df.mean()
A   -0.674980
B   -0.251460
C    0.330594
D    5.000000
F    3.000000
dtype: float64


>>> df.sum()

#传入axis=1将会按行进行求和运算：
>>> df.sum(axis=1)
```

五、 基本功能

其他轴上进行相同的操作：

```
>>> df.mean(1)   # df.mean(0)
2013-01-01    1.341706
2013-01-02    1.199406
2013-01-03    0.894654
2013-01-04    0.853198
2013-01-05    2.410234
2013-01-06    1.854127
Freq: D, dtype: float64



>>> df.mean(axis=1)
```

五、 基本功能

对于拥有不同维度，需要对齐的对象进行操作。Pandas会自动的沿着指定的维度进行广播：

```
>>> s = pd.Series([1,3,5,np.nan,6,8], index=dates).shift(2)

>>> s
2013-01-01   NaN
2013-01-02   NaN
2013-01-03     1
2013-01-04     3
2013-01-05     5
2013-01-06   NaN
Freq: D, dtype: float64

>>> df.sub(s, axis='index')
```

五、 基本功能

描述和汇总统计：

方法	说明
count	非NA值的数量
describe	针对Series或各DataFrame列计算汇总统计
min,max	计算最小值和最大值
argmin,argmax	计算能够获取到最小值和最大值的索引位置（整数)
idxmin,idxmax	计算能够获取到最小值和最大值的索引值
quantile	计算样本的分位数（0到 1）
sum	值的总和
mean	值的平均数
media	值的算术中位数（50%分位数)
mad	根据平均值计算平均绝对离差
var	样本值的方差

五、 基本功能

描述和汇总统计（续）

方法	说明
std	样本值的标准差
skew	样本值的偏度（三阶矩）
kurt	样本值的峰度（四阶矩）
cumsum	样本值的累计和
cummin,cummax	样本值的累计最大值和累计最小
cumprod	样本值的累计积
diff	计算一阶差分（对时间序列很有用)
pct_change	计算百分数变化

五、 基本功能

直方图：

```
>>> s = pd.Series(np.random.randint(0, 7, size=10))
>>> s
0    0
1    2
2    4
3    3
4    2
5    2
6    4
7    6
8    0
9    2
dtype: int32

>>> s.value_counts()
2    4
4    2
0    2
6    1
3    1
dtype: int64
```

五、 基本功能

4、字符串方法

Series对象在其str属性中配备了一组字符串处理方法，可以很容易的应用到数组中的每个元素，如下段代码所示。

```
>>> s = pd.Series(['A', 'B', 'C', 'Aaba', 'Baca', np.nan, 'CABA', 'dog', 'cat'])
>>>  s.str.lower()
0       a
1       b
2       c
3    aaba
4    baca
5     NaN
6    caba
7     dog
8     cat
dtype: object、
```

六、合并

Pandas提供了大量的方法能够轻松的对Series，DataFrame和Panel对象进行各种符合各种逻辑关系的合并操作。

1、 Concat（相同字段的表首尾相接）

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

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

六、合并

```
>>> df = pd.DataFrame(np.random.randn(10, 4))
>>> df
>>> pieces = [df[:3], df[3:7], df[7:]] # break it into pieces. pieces[1]
>>> pieces
>>> pd.concat(pieces)    # pd.concat([pieces[1], pieces[0]], axis=0)
```

定义一个能够创建 DataFrame 某种形式的函数：

```
>>> def make_df(cols, ind):
    data = {c: [str(c) + str(i) for i in ind] for c in cols}
    return pd.DataFrame(data, ind)

>>> make_df('ABC', range(3))
```

六、合并

```
>>> df3 = make_df('AB', [0, 1])
>>> df4 = make_df('CD', [0, 1])
>>> print(df3); print(df4); print(pd.concat([df3, df4],
axis=1))

   A   B
0  A0  B0
1  A1  B1
   C   D
0  C0  D0
1  C1  D1
   A   B   C   D
0  A0  B0  C0  D0
1  A1  B1  C1  D1
```

六、合并

索引重复：Pandas 在合并时会保留索引

```
>>> x = make_df('AB', [0, 1])
>>> y = make_df('AB', [2, 3])
>>> y.index = x.index        # 复制索引
>>> print(x); print(y); print(pd.concat([x, y]))

   A   B
0  A0  B0
1  A1  B1
   A   B
0  A2  B2
1  A3  B3
   A   B
0  A0  B0
1  A1  B1
0  A2  B2
1  A3  B3
```

六、合并

忽略索引：有时索引无关紧要，那么合并时就可以忽略它们， 可以通过设置 ignore_index 参数来实现。如果将参数设置为True，那么合并时将会创建一个新的整数索引。

```
>>> print(pd.concat([x, y], ignore_index=True))

    A   B
0  A0  B0
1  A1  B1
2  A2  B2
3  A3  B3
```

六、合并

增加多级索引：另一种处理索引重复的方法是通过 keys 参数为数据源设置多级索引标签，这样结果数据就会带上多级索引：

```
>>> print(x); print(y); print(pd.concat([x, y],
keys=['x', 'y']))
   A   B
0  A0  B0
1  A1  B1
   A   B
0  A2  B2
1  A3  B3
     A   B
x 0  A0  B0
  1  A1  B1
y 0  A2  B2
  1  A3  B3
```

六、合并

2、 Append

因为直接进行数组合并的需求非常普遍，所以 Series 和DataFrame 对象都支持 append 方法，让你通过最少的代码实现合并功能。

```
>>>  df = pd.DataFrame(np.random.randn(8, 4), columns=['A','B','C','D'])
>>> df
>>> s = df.iloc[3]
>>> df.append(s, ignore_index=True)
>>> df.append(s, ignore_index=False)

>>> print(x); print(y);
print(x.append(y,ignore_index=True))
```

六、合并

3、merge

```
>>> left = pd.DataFrame({'key': ['foo', 'foo'], 'lval': [1, 2]})
>>> right = pd.DataFrame({'key': ['foo', 'foo'], 'rval': [4, 5]})
>>> left
>>> right
>>> pd.merge(left, right, on='key')

# Another example that can be given is:
>>> left = pd.DataFrame({'key': ['foo', 'bar'], 'lval': [1, 2]})
>>> right = pd.DataFrame({'key': ['foo', 'bar'], 'rval': [4, 5]})
>>> left
>>> right
>>> pd.merge(left, right, on='key')
```

pd.merge() 函数实现了三种数据连接的类型：一对一、多对一和多对多。根据不同的数据连接需求进行不同的操作。

六、合并

一对一连接：

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

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

六、合并

```
>>>  df3 = pd.merge(df1, df2)
>>> df3
employee        group  hire_date
0      Bob   Accounting       2008
1     Jake  Engineering       2012
2     Lisa  Engineering       2004
3      Sue           HR       2014
```

pd.merge() 方法会发现两个 DataFrame 都有“employee”列，并会自动以这列作为键进行连接，合并结果是一个新的DataFrame。需要注意的是，虽然 df1 与 df2 中“employee”列的位置是不一样的，但是 pd.merge() 函数会正确处理这个问题。另外还需要注意的是，pd.merge() 会默认丢弃原来的行索引，不过也可以自定义。

六、合并

多对一连接：多对一连接是指在需要连接的两个列中，有一列的值有重复。通过多对一连接获得的结果 DataFrame 将会保留重复值。

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

六、合并

```
>>> print(df3); print(df4); print(pd.merge(df3, df4))

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

         group supervisor
0   Accounting      Carly
1  Engineering      Guido
2           HR      Steve

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

六、合并

多对多连接：如果左右两个输入的共同列都包含重复值，那么合并的结果就是一种多对多连接。

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

>>> print(df1); print(df5); print(pd.merge(df1, df5))
```

六、合并

```
   employee        group
0       Bob   Accounting
1      Jake  Engineering
2      Lisa  Engineering
3       Sue           HR
         group       skills
0   Accounting         math
1   Accounting   spreadsheets
2  Engineering       coding
3  Engineering        linux
4           HR   spreadsheets
5           HR   organization
  employee        group       skills
0       Bob   Accounting         math
1       Bob   Accounting   spreadsheets
2      Jake  Engineering       coding
3      Jake  Engineering        linux
4      Lisa  Engineering       coding
5      Lisa  Engineering        linux
6       Sue           HR   spreadsheets
7       Sue           HR   organization
```

六、合并

设置数据合并的键：1）、直接将参数 on 设置为一个列名字符串或者一个包含多列名称的列表：

```
>>> print(df1); print(df2); print(pd.merge(df1, df2, on='employee'))
  employee        group
0      Bob   Accounting
1     Jake  Engineering
2     Lisa  Engineering
3      Sue           HR
  employee  hire_date
0     Lisa       2004
1      Bob       2008
2     Jake       2012
3      Sue       2014
  employee        group  hire_date
0      Bob   Accounting       2008
1     Jake  Engineering       2012
2     Lisa  Engineering       2004
3      Sue           HR       2014
```

六、合并

2）**left_on**与**right_on**参数：有时需要合并两个列名不同的数据集：

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

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

六、合并

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

>>> pd.merge(df1, df3, left_on="employee",
right_on="name").drop('name', axis=1)

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

获取的结果中会有一个多余的列，可以通过 DataFrame 的 drop() 方法将这列去掉。

六、合并

3）left_index与right_index参数：

除了合并列之外，你可能还需要合并索引。

```
>>> df1a = df1.set_index('employee')
df2a = df2.set_index('employee')
print(df1a); print(df2a)

                group
employee
Bob         Accounting
Jake       Engineering
Lisa       Engineering
Sue                HR
              hire_date
employee
Lisa             2004
Bob              2008
Jake             2012
Sue              2014
```

六、合并

```
>>> print(df1a); print(df2a);
print(pd.merge(df1a, df2a, left_index=True, right_index=True))
                group
employee
Bob         Accounting
Jake        Engineering
Lisa        Engineering
Sue                 HR
          hire_date
employee
Lisa            2004
Bob             2008
Jake            2012
Sue             2014
                group  hire_date
employee
Bob         Accounting        2008
Jake        Engineering       2012
Lisa        Engineering       2004
Sue                 HR        2014
```

六、合并

设置数据连接的集合操作规则：how 参数支持的数据连接方式还有‘inner’、 'outer'、'left' 和 'right'。

```
>>> df6 = pd.DataFrame({'name': ['Peter', 'Paul', 'Mary'],
'food': ['fish', 'beans', 'bread']}, columns=['name', 'food'])
df7 = pd.DataFrame({'name': ['Mary', 'Joseph'],
'drink': ['wine', 'beer']}, columns=['name', 'drink'])
print(df6); print(df7); print(pd.merge(df6, df7))
    name   food
0  Peter   fish
1   Paul  beans
2   Mary  bread
    name drink
0   Mary  wine
1 Joseph  beer
   name   food   drink
0  Mary   bread   wine
```

六、合并

```
>>> print(df6); print(df7); print(pd.merge(df6, df7,
how='outer'))

    name   food
0  Peter   fish
1   Paul  beans
2   Mary  bread

    name drink
0   Mary  wine
1 Joseph  beer

    name   food   drink
0  Peter   fish     NaN
1   Paul  beans     NaN
2   Mary  bread    wine
3 Joseph    NaN   beer
```

六、合并

重复列名：**suffixes**参数：若输出结果中有两个重复的列名， pd.merge() 函数会自动为它们增加后缀 _x 或 _y，当然也可以通过 suffixes 参数自定义后缀名：

```
>>> df8 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
'rank': [1, 2, 3, 4]})
df9 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
'rank': [3, 1, 4, 2]})
print(df8); print(df9); print(pd.merge(df8, df9, on="name"))
   name  rank
0   Bob     1
1  Jake     2
2  Lisa     3
3   Sue     4
   name  rank
0   Bob     3
1  Jake     1
2  Lisa     4
3   Sue     2
```

六、合并

```
name  rank_x  rank_y
0   Bob       1       3
1  Jake       2       1
2  Lisa       3       4
3   Sue       4       2

>>>print(pd.merge(df8, df9, on="name", suffixes=["_L", "_R"]))
   name  rank_L  rank_R
0   Bob       1       3
1  Jake       2       1
2  Lisa       3       4
3   Sue       4       2
```

七、分组

对于”group by”操作，通常是指以下一个或多个操作步骤：

（Splitting）按照一些规则将数据分为不同的组；

（ Applying）对于每组数据分别执行一个函数；

（Combining）将结果组合到一个数据结构中；

七、分组

```
>>> df = pd.DataFrame({'A' : ['foo', 'bar', 'foo', 'bar',
                              'foo', 'bar', 'foo', 'foo'],
                    'B' : ['one', 'one', 'two', 'three',
                              'two', 'two', 'one', 'three'],
                    'C' : np.random.randn(8),
                    'D' : np.random.randn(8)})

 >>> df

>>> df.groupby('A').sum()

>>> df.groupby(['A','B']).sum()
```

八、数据重塑（reshape）

1、 Stack

```
>>> tuples = list(zip(*[['bar', 'bar', 'baz', 'baz',
                        'foo', 'foo', 'qux', 'qux'],
                        ['one', 'two', 'one', 'two',
                        'one', 'two', 'one', 'two']]))

>>> index = pd.MultiIndex.from_tuples(tuples,
names=['first', 'second'])
>>> index

>>> df = pd.DataFrame(np.random.randn(8, 2),
index=index, columns=['A', 'B'])
>>> df

>>> df2 = df[:4]
>>> df2
```

八、数据重塑（reshape）

```
>>>stacked = df2.stack()
>>> stacked
first  second
bar    one     A    0.045397
               B   -0.379312
       two     A   -0.404804
               B    0.184046
baz    one     A    0.477272
               B   -0.144071
       two     A    0.516476
               B    1.183518
dtype: float64

>>> stacked.unstack()
>>> stacked.unstack(1)
>>> stacked.unstack(0)
```

八、数据重塑（reshape）

2、 数据透视表

```
>>> df = pd.DataFrame({'A' : ['one', 'one', 'two', 'three'] * 3,
                       'B' : ['A', 'B', 'C'] * 4,
                       'C' : ['foo', 'foo', 'foo', 'bar', 'bar', 'bar'] * 2,
                       'D' : np.random.randn(12),
                       'E' : np.random.randn(12)})

>>> df

>>> pd.pivot_table(df, values='D', index=['A', 'B'], columns=['C'])

>>> pd.pivot_table?
```

九、时间序列

Pandas在对频率转换进行重新采样时拥有简单、强大且高效的功能（如将按秒采样的数据转换为按1分钟为单位进行采样的数据）。这种操作在金融领域非常常见。

```
>>> rng = pd.date_range('1/1/2012', periods=200, freq='S')

>>> ts = pd.Series(np.random.randint(0, 50, len(rng)), index=rng)

>>> ts.resample('1Min')    # ts.resample?

>>> ts.resample('1Min').sum()
```

九、时间序列

1、 时区表示：

```
>>>rng = pd.date_range('3/6/2012 00:00', periods=5, freq='D')

>>> ts = pd.Series(np.random.randn(len(rng)), rng)
>>> ts
2012-03-06    0.082352
2012-03-07    2.937241
2012-03-08    0.500599
2012-03-09   -0.883261
2012-03-10   -0.078990
Freq: D, dtype: float64

>>> ts_utc = ts.tz_localize('UTC')
>>> ts_utc
2012-03-06 00:00:00+00:00    0.082352
2012-03-07 00:00:00+00:00    2.937241
2012-03-08 00:00:00+00:00    0.500599
2012-03-09 00:00:00+00:00   -0.883261
2012-03-10 00:00:00+00:00   -0.078990
Freq: D, dtype: float64
```

九、时间序列

2、 时区转换：

```
>>> ts_utc.tz_convert('US/Eastern')
2012-03-05 19:00:00-05:00    0.082352
2012-03-06 19:00:00-05:00    2.937241
2012-03-07 19:00:00-05:00    0.500599
2012-03-08 19:00:00-05:00   -0.883261
2012-03-09 19:00:00-05:00   -0.078990
Freq: D, dtype: float64
```

3、 时间跨度转换：

```
>>> rng = pd.date_range('1/1/2012', periods=5, freq='M')
>>> ts = pd.Series(np.random.randn(len(rng)), index=rng)
>>> ts
2012-01-31   -1.004425
2012-02-29    0.692854
2012-03-31    0.562317
2012-04-30   -0.520005
2012-05-31   -1.109106
Freq: M, dtype: float64
```

九、时间序列

```
>>> ps = ts.to_period()

>>> ps
2012-01   -1.004425
2012-02    0.692854
2012-03    0.562317
2012-04   -0.520005
2012-05   -1.109106
Freq: M, dtype: float64

>>> ps.to_timestamp()
2012-01-01   -1.004425
2012-02-01    0.692854
2012-03-01    0.562317
2012-04-01   -0.520005
2012-05-01   -1.109106
Freq: MS, dtype: float64
```

九、时间序列

4、 时期和时间戳之间的转换使得可以使用一些方便的算术函数。

```
>>>
prng = pd.period_range('1990Q1', '2000Q4', freq='Q-NOV')

>>> ts = pd.Series(np.random.randn(len(prng)), prng)

>>> ts.index = (prng.asfreq('M', 'e') + 1).asfreq('H', 's') + 9

>>> ts.head()
1990-03-01 09:00    1.211725
1990-06-01 09:00   -0.969415
1990-09-01 09:00    0.184478
1990-12-01 09:00    0.206094
1991-03-01 09:00    0.756307
Freq: H, dtype: float64
```

十、 Categorical

从0.15版本开始，pandas可以在DataFrame中支持Categorical类型的数据。

```
>>> df = pd.DataFrame({"id":[1,2,3,4,5,6],
"raw_grade":['a', 'b', 'b', 'a', 'a', 'e']})
```

1、将raw_grade转换为Categorical数据类型：

```
>>> df["grade"] = df["raw_grade"].astype("category")
>>> df["grade"]
0    a
1    b
2    b
3    a
4    a
5    e
Name: grade, dtype: category
Categories (3, object): [a, b, e]
```

十、 Categorical

2、 将Categorical类型数据重命名为更有意义的名称：

```
>>> df["grade"].cat.categories = ["very good", "good", "very bad"]
```

3、对类别进行重新排序，增加缺失的类别：

```
>>> df["grade"] = df["grade"].cat.set_categories(["very bad", "bad", "medium", "good", "very good"])

>>> df["grade"]
0    very good
1         good
2         good
3    very good
4    very good
5     very bad
Name: grade, dtype: category
Categories (5, object): [very bad, bad, medium, good, very good]
```

十、 Categorical

4、 排序是按照Categorical的顺序进行的而不是按照字典顺序进行：

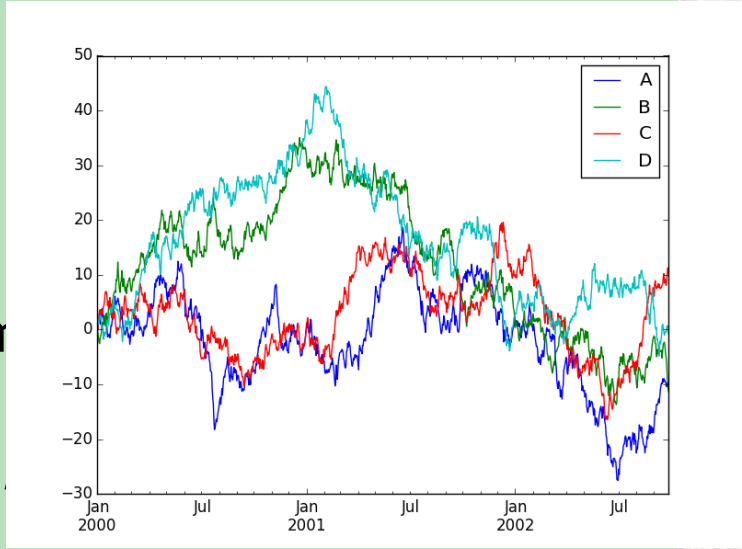
```
>>> df.sort_index(by="grade")
```

5、 对Categorical列进行排序时存在空的类别：

```
>>> df.groupby("grade").size()
grade
very bad      1
bad         NaN
medium      NaN
good          2
very good     3
dtype: float64
```

十一、画图

```
>>> ts = pd.Series(np.random.randn(1000),
index=pd.date_range('1/1/2000', periods=1000))

>>> ts = ts.cumsum()

>>> ts.plot()


>>> df = pd.DataFrame(np.random
index=ts.index,
                  columns=['A', 'B', 'C'

>>> df = df.cumsum()

>>> plt.figure(); df.plot(); plt.legend(loc='best')
```



十二、导入和保存数据

CSV

写入csv文件：

```
>>> df.to_csv('foo.csv')
```

从csv文件中读取：

```
>>> pd.read_csv('foo.csv')
```

写入HDF5存储：从HDF5存储中读取：

```
>>> df.to_hdf('foo.h5','df')
>>> pd.read_hdf('foo.h5','df')
```

写入excel文件：从excel文件中读取：

```
>>> df.to_excel('foo.xlsx', sheet_name='Sheet1')
>>> pd.read_excel('foo.xlsx', 'Sheet1',
index_col=None, na_values=['NA'])
```