Ch7 Data Cleaning and Preparation 数据清洗与准备

In [266]:
```
import pandas as pd
import numpy as np
import re
```

7.1 处理缺失值 Handling Missing Data

In [11]:
```
#pandas使用浮点值Nan-not a number表示缺失值。Nan为容易检测到的标识值。
string_data = pd.Series(['aardvark', 'artichok', np.nan, 'avocado'])
string_data
# 0     aardvark
# 1     artichok
# 2          NaN
# 3      avocado
# dtype: object

#.isnull()用来检查是否为空值：缺失值和none均为空值。
string_data.isnull()
# 0     False
# 1     False
# 2      True
# 3     False
# dtype: bool

string_data[0] = None
string_data.isnull()
# 0      True
# 1     False
# 2      True
# 3     False
# dtype: bool
```

Out[11]:
```
0     True
1    False
2     True
3    False
dtype: bool
```

In [ ]:
```
#【重点：】
#NA处理方法：
# dropna：根据每个标签的值是否是缺失数据来筛选轴标签，根据允许丢失的数据量，来确定阈值
# fillna：用某些值，填充确实的数据或使用插值方法。
# isnull：返回表名那些值是缺失值的布尔值。
# notnull：isnull的反函数。
```

7.1.1 过滤缺失值 Filtering Out Missing Data

In [15]:
```
#.dropna() <=> .notnull() 去掉缺失值
#pandas.isnull和布尔值索引手动过滤缺失值
#dropna在过滤缺失值很有用。在series使用dropna, 会返回series中所有的非空数据及其索引

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

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

Out[15]:
```
0    1.0
2    3.5
4    7.0
dtype: float64
```

In [25]:
```
#当处理data frame对象时，可能想要删除全部为NA或包含有NA的行或列，但dropna默认情况下

data = pd.DataFrame([[1., 6.5, 3.], [1., NA, NA], [NA, NA, NA], [NA, 6.5, 3.]]
cleaned = data.dropna()
data
# 0 1   2
# 0 1.0 6.5 3.0
# 1 1.0 NaN NaN
# 2 NaN NaN NaN
# 3 NaN 6.5 3.0
cleaned
# 0 1   2
# 0 1.0 6.5 3.0

#只删除（所有的值均为NA)的行：传入how = 'all'时，将，保留仅有1.2...n个na的行：
data.dropna(how = 'all')
# 0 1   2
# 0 1.0 6.5 3.0
# 1 1.0 NaN NaN
# 3 NaN 6.5 3.0

#把所有的值均为NA的列删除：传入参数axis = 1
data[4] = NA
data
#   0   1   2   4
# 0 1.0 6.5 3.0 NaN
# 1 1.0 NaN NaN NaN
# 2 NaN NaN NaN NaN
# 3 NaN 6.5 3.0 NaN
data.dropna(axis = 1, how = 'all')
# 0 1   2
# 0 1.0 6.5 3.0
# 1 1.0 NaN NaN
# 2 NaN NaN NaN
# 3 NaN 6.5 3.0

#过滤DataFrame的行的相关方法设计时间序列。 假设只想保留包含一定数量的观察值的行。
#可以用thresh参数来表示：
df = pd.DataFrame(np.random.randn(7,3))
df.iloc[:4, 1] = NA                              #前4行，第2列，设置为NA
df.iloc[:2, 2] = NA                              #前2行，第3列，设置为NA
df
# 0 1   2
# 0 -2.773912   NaN NaN
# 1 1.787779    NaN NaN
# 2 0.498764    NaN 1.108871
# 3 0.134833    NaN 1.313636
# 4 0.362728    -0.119155   0.077606
# 5 -0.742221   -0.851967   -0.233683
# 6 -1.169108   -0.541773   -1.091397
df.dropna()                                      #.dropna()：有一个为NA即drop掉
# 0 1   2
# 4 -0.745122   0.474192    -0.207012
# 5 -1.111911   0.963357    -0.690348
# 6 -0.279375   1.082670    0.432352
df.dropna(thresh = 2)                            #.dropna(thresh = 2)：有2个NA的
# 0 1   2
```

```
# 2 0.146841    NaN -1.218570
# 3 0.040456    NaN 0.516865
# 4 -2.297259   0.030341    -0.211776
# 5 -2.368655   -0.969274   -0.003980
# 6 0.132776    0.410199    0.208535
```

Out[25]:

	0	1	2
2	0.146841	NaN	-1.218570
3	0.040456	NaN	0.516865
4	-2.297259	0.030341	-0.211776
5	-2.368655	-0.969274	-0.003980
6	0.132776	0.410199	0.208535

7.1.2 补全缺失值 Filling In Missing Data

In [71]:
```
#补全漏洞，而不是过滤缺失值时。可以使用fillna方法来补全缺失值
#缺失值可以：填0，字典式，就近填充，平均值/中位数

#1.全部补0
df.fillna(0)
# 0 1   2
# 0 0.772158    0.000000    0.000000
# 1 -2.539879   0.000000    0.000000
# 2 0.146841    0.000000    -1.218570
# 3 0.040456    0.000000    0.516865
# 4 -2.297259   0.030341    -0.211776
# 5 -2.368655   -0.969274   -0.003980
# 6 0.132776    0.410199    0.208535

#2.字典式赋值
#调用fillna时使用字典：第2列填充0.5，第3列填充0：.fillna({n1:a, n2:b})
df.fillna({1:0.5, 2:0})
# 0 1   2
# 0 0.772158    0.500000    0.000000
# 1 -2.539879   0.500000    0.000000
# 2 0.146841    0.500000    -1.218570
# 3 0.040456    0.500000    0.516865
# 4 -2.297259   0.030341    -0.211776
# 5 -2.368655   -0.969274   -0.003980
# 6 0.132776    0.410199    0.208535

#fillna返回的是一个新的对象，也可以修改已经存在的对象：全部填0：.fillna(0, inplace
_ = df.fillna(0, inplace = True)
df
# 0 1   2
# 0 0.772158    0.000000    0.000000
# 1 -2.539879   0.000000    0.000000
# 2 0.146841    0.000000    -1.218570
# 3 0.040456    0.000000    0.516865
# 4 -2.297259   0.030341    -0.211776
# 5 -2.368655   -0.969274   -0.003980
# 6 0.132776    0.410199    0.208535

#用于重建索引的相同的插值方法也可以用于fillna
df = pd.DataFrame(np.random.randn(6,3))
df.iloc[2:, 1] = NA                             #[2:,1]：第3行之后，第一列
df.iloc[4:, 2] = NA                             #[4:,2]：第5行之后，第二列
df
# 0 1   2
# 0 -0.801166   -0.779006   -0.186398
# 1 -0.449003   -0.236377   0.852875
# 2 0.003301    NaN -1.180197
# 3 0.553074    NaN -0.163766
# 4 -1.076717   NaN NaN
# 5 -0.714772   NaN NaN

#3.将缺失值按前面最近的非缺失值填
#.fillna(method = 'ffill')：它将缺失值用其前面的非缺失值进行填充，即缺失值将被其前面
df.fillna(method = 'ffill')
# 0 1   2
# 0 -0.245778   -0.315176   0.159850
# 1 -0.383522   【-0.248150】 1.614308
```

```
# 2 1.135666    -0.248150   1.319412
# 3 1.873568    -0.248150   【1.718205】
# 4 -1.686481   -0.248150   1.718205
# 5 -0.037789   -0.248150   1.718205


#.fillna(method='ffill', limit=n): 只允许连续的最多n个缺失值被填充为其前面的非缺失
df.fillna(method = 'ffill', limit = 2)
# 0 1   2
# 0 -0.057093   -1.515736   -0.059605
# 1 1.262333    1.096798    -1.362441
# 2 0.930699    1.096798    -1.862778
# 3 -1.029882   1.096798    -1.019072
# 4 -0.302298   NaN -1.019072
# 5 0.088165    NaN -1.019072


#4.将series的平均值或中位数用于填充缺失值: .fillna(data.mean()) or .fillna(data.m
data = pd.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
data = pd.Series([1., NA, 3.5, NA, 7])
data.fillna(data.median())
# 0    1.0
# 1    3.5
# 2    3.5
# 3    3.5
# 4    7.0
# dtype: float64
```

Out[71]:

```
0    1.0
1    3.5
2    3.5
3    3.5
4    7.0
dtype: float64
```

In [ ]:

```
#fillna函数参数:
# value: 标量值或字典型对象用于填充缺失值
# method: 插值方法, 如果没有其他参数, 默认是'ffill'
# axis: 需要填充的轴, 默认axis = 0
# inplace: 修改被调用的对象, 而不是生成一个备份
# limit: 用于前向或后向填充式最大的填充范围
```

7.2 数据转换 Data Transformation

7.2.1 删除重复值 Removing Duplicates

In [81]:
```
data = pd.DataFrame({'k1': ['one', 'two'] * 3 + ['two'],
                     'k2': [1, 1, 2, 3, 3, 4, 4]})
data
# k1    k2
# 0 one 1
# 1 two 1
# 2 one 2
# 3 two 3
# 4 one 3
# 5 two 4
# 6 two 4

#.duplicated()检验是否重复
#data frame的duplicated方法返回的事一个布尔值series，这个series反映的是每一行，是否
data.duplicated()
# 0     False
# 1     False
# 2     False
# 3     False
# 4     False
# 5     False
# 6      True
# dtype: bool

#drop_duplicates返回的是data frame，内容是duplicated返回数组为false的部分：
data.drop_duplicates()
#   k1  k2
# 0 one 1
# 1 two 1
# 2 one 2
# 3 two 3
# 4 one 3
# 5 two 4

#.drop_duplicates(['column_name'])：想根据该列，去除重复值：
#增添一个k1列，根据该列，去除重复值：只有第一次出现one two的行留下了

data['v1'] = range(7)
data
# k1    k2  v1
# 0 one 1   0
# 1 two 1   1
# 2 one 2   2
# 3 two 3   3
# 4 one 3   4
# 5 two 4   5
# 6 two 4   6
data.drop_duplicates(['k1'])
# k1    k2  v1
# 0 one 1   0
# 1 two 1   1
```

Out[81]:

	k1	k2	v1
0	one	1	0
1	two	1	1

In [82]:
```
#duplicated和drop_duplicates默认都是保留第一个观测的值。传入参数keep = 'last'将返[
#因为[5], [6]行都是two 4, 传入keep = 'last'后, 保留[6]
data.drop_duplicates(['k1','k2'], keep = 'last')

# 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
```

Out[82]:

	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

7.2.2 使用函数或映射进行数据转换 Transforming Data Using a Function or Mapping

In [93]:
```
data = pd.DataFrame({'food': ['bacon', 'pulled pork', 'bacon',
                              'Pastrmi', 'corned beef', 'Bacon',
                              'pastrami', 'honey ham', 'nova lox'],
                    'ounces':[4, 3, 12, 6, 7.5, 8, 3, 5, 6]})
data
# food  ounces
# 0 bacon   4.0
# 1 pulled pork 3.0
# 2 bacon   12.0
# 3 Pastrmi 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

#假设想添加一列用于表名每种食物的动物肉类型。先写下一个食物和肉类的映射：
meat_to_animal = {
    'bacon': 'pig',
    'pulled pork': 'cow',
    'corned beef': 'pig',
    'honey ham': 'pig',
    'nova lox': 'salmon'
}

#series的map方法，接收一个函数或一个包含映射关系的字典型对象。
#此处有一个问题在于，一些肉类大写了，另一部分肉类没有。
#因此需要series的str.lower将每个值转换为小写：
lowercased = data['food'].str.lower()
lowercased
# 0          bacon
# 1     pulled pork
# 2          bacon
# 3        pastrmi
# 4     corned beef
# 5          bacon
# 6       pastrami
# 7      honey ham
# 8       nova lox
# Name: food, dtype: object

#
data['animal'] = lowercased.map(meat_to_animal)
data
# food  ounces  animal
# 0 bacon   4.0 pig
# 1 pulled pork 3.0 cow
# 2 bacon   12.0    pig
# 3 Pastrmi 6.0 NaN
# 4 corned beef 7.5 pig
# 5 Bacon   8.0 pig
# 6 pastrami    3.0 NaN
# 7 honey ham   5.0 pig
# 8 nova lox    6.0 salmon
```

Out[93]:

	food	ounces	animal
0	bacon	4.0	pig
1	pulled pork	3.0	cow
2	bacon	12.0	pig
3	Pastrmi	6.0	NaN
4	corned beef	7.5	pig
5	Bacon	8.0	pig
6	pastrami	3.0	NaN
7	honey ham	5.0	pig
8	nova lox	6.0	salmon

In [175]:

```
#也可以传入一个能够完成所有工作的函数:
data = pd.DataFrame({'food': ['bacon', 'pulled pork', 'bacon',
                              'Pastrmi', 'corned beef', 'Bacon',
                              'pastrami', 'honey ham', 'nova lox'],
                     'ounces':[4, 3, 12, 6, 7.5, 8, 3, 5, 6]})
meat_to_animal = {
    'bacon': 'pig',
    'pulled pork': 'cow',
    'corned beef': 'pig',
    'honey ham': 'pig',
    'nova lox': 'salmon'
}

#data['food'].map(lambda x: meat_to_animal[x.lower()])
```

7.2.3替代值 Replacing Values

In [107]:
```
#使用fillna填充缺失值，是通用值替换的特殊案例。
#map可以用来修改一个对象中的子集的值，但是replace提供了更为简单的实现。

data = pd.Series([1., -999., 2., -999., -1000., 3.])
data
# 0        1.0
# 1     -999.0
# 2        2.0
# 3     -999.0
# 4    -1000.0
# 5        3.0
# dtype: float64

#.replace(a, np.nan)替换为NA值
#-999可能是缺失值的标志，可以用replace方法生成新的series：把-999 替换成na
data.replace(-999, np.nan)
# 0        1.0
# 1        NaN
# 2        2.0
# 3        NaN
# 4    -1000.0
# 5        3.0
# dtype: float64

#.replace([a,b], np.nan)：如果想要一次替代多个值，可以传入一个列表和替代值：
data.replace([-999, -1000], np.nan)
# 0     1.0
# 1     NaN
# 2     2.0
# 3     NaN
# 4     NaN
# 5     3.0
# dtype: float64

#.replace([a,b], [np.nan,0])：要将不同的值，替换为不同的值，可以传入替代值的列表：
data.replace([-999, -1000], [np.nan, 0])
# 0     1.0
# 1     NaN
# 2     2.0
# 3     NaN
# 4     0.0
# 5     3.0
# dtype: float64
```

Out[107]:
```
0    1.0
1    NaN
2    2.0
3    NaN
4    0.0
5    3.0
dtype: float64
```

7.2.4 重命名轴索引 Renaming Axis Indexes

In [119]:
```
data = pd.DataFrame(np.arange(12).reshape((3,4)),
                    index = ['Ohio', 'Colorado', 'New York'],
                    columns = ['one', 'two', 'three', 'four'])
data
#   one two three   four
# Ohio  0   1   2   3
# Colorado  4   5   6   7
# New York  8   9   10  11

#lambda x: x[:4].upper(): 只将index 前4个字母保留并大写
transform = lambda x: x[:4].upper()
data.index.map(transform)
#Index(['OHIO', 'COLO', 'NEW '], dtype='object')

data.index = data.index.map(transform)
data
#   one two three   four
# OHIO  0   1   2   3
# COLO  4   5   6   7
# NEW   8   9   10  11

#想创建数据集后的版本，不修改原有的数据集，可以用rename：
#data.rename(index = str.title, columns = str.upper)：大写列名
data.rename(index = str.title, columns = str.upper)
# ONE   TWO THREE   FOUR
# Ohio  0   1   2   3
# Colo  4   5   6   7
# New   8   9   10  11

#rename可以和字典型对象使用，为轴标签的子集提供新的值：
#rename可以修改行/列名：
data.rename(index = {'OHIO' : 'INDIANA'},
            columns = {'three' : 'peekaboo'})
# one   two peekaboo    four
# INDIANA   0   1   2   3
# COLO  4   5   6   7
# NEW   8   9   10  11

#rename可以从手动复制data frame并为其分配索引和列属性解放，
#如果想修改原有数据集，rename传入inplace = True：
data.rename(index = {'OHIO': 'INDIANA'}, inplace = True)
data
#   one two three   four
# INDIANA   0   1   2   3
# COLO  4   5   6   7
# NEW   8   9   10  11
```

Out[119]:

	one	two	three	four
INDIANA	0	1	2	3
COLO	4	5	6	7
NEW	8	9	10	11

7.2.5 离散化和分箱 Discretization and Binning

In [138]:

```
#连续值经常需要离散化，或者分离成箱子进行分析。
#假设有某项研究中一组人群的数据，想将他们进行分组，放入离散的年龄框中

ages = [20, 22, 25, 27, 21, 23, 37, 31, 61, 45, 41, 32]

#将年龄分为若干个组，可以使用pands.cut：
bins = [18, 25, 35, 60, 100]
cats = pd.cut(ages, bins)
cats
# [(18, 25], (18, 25], (18, 25], (25, 35], (18, 25], ..., (25, 35], (60, 100],
# Length: 12
# Categories (4, interval[int64, right]): [(18, 25] < (25, 35] < (35, 60] < (6
```

Out[138]:

```
[(18, 25], (18, 25], (18, 25], (25, 35], (18, 25], ..., (25, 35], (60, 100],
(35, 60], (35, 60], (25, 35]]
Length: 12
Categories (4, interval[int64, right]): [(18, 25] < (25, 35] < (35, 60] < (6
0, 100]]
```

In [ ]:

```
#pandas返回的对象是一个特殊的categorical对象，看到的输出描述了由padas.cut计算出的箱
#它在内部包含一个categories数组，指定了不同的类别名称以及codes属性中的ages数据标签：
#cats.codes：这行代码访问Categorical对象的codes属性，返回每个个体所属年龄段的编码。
#编码是根据年龄段的顺序来分配的，从0开始递增，表示该年龄隶属的不同的年龄段。
cats.codes
# array([0, 0, 0, 1, 0, 0, 2, 1, 3, 2, 2, 1], dtype=int8)
cats.categories
# IntervalIndex([(18, 25], (25, 35], (35, 60], (60, 100]], dtype='interval[int

#pd.value_counts(cats)：对pandas.cut的结果中的箱数量的计数。
pd.value_counts(cats)
# (18, 25]     5
# (25, 35]     3
# (35, 60]     3
# (60, 100]    1
# dtype: int64
```

In [141]:
```
#right = False or True: ()[]与数学符号一致，可以通过传递来改变哪一边是封闭的:

pd.cut(ages, [18, 26, 36, 61, 100], right = False)
# [[18, 26), [18, 26), [18, 26), [26, 36), [18, 26), ..., [26, 36), [61, 100),
# Length: 12
# Categories (4, interval[int64, left]): [[18, 26) < [26, 36) < [36, 61) < [61
pd.cut(ages, [18, 26, 36, 61, 100], right = True)
# [(18, 26], (18, 26], (18, 26], (26, 36], (18, 26], ..., (26, 36], (36, 61],
# Length: 12
# Categories (4, interval[int64, right]): [(18, 26] < (26, 36] < (36, 61] < (6
```

Out[141]:
```
[(18, 26], (18, 26], (18, 26], (26, 36], (18, 26], ..., (26, 36], (36, 61],
(36, 61], (36, 61], (26, 36]]
Length: 12
Categories (4, interval[int64, right]): [(18, 26] < (26, 36] < (36, 61] < (6
1, 100]]
```

In [144]:
```
#pd.cut(ages, bins, labels = group_names): 也可以通过向labels传递一个列表或数组来
group_names = ['Youth', 'YoungAdult', 'MiddleAged', 'Senior']
pd.cut(ages, bins, labels = group_names)
# ['Youth', 'Youth', 'Youth', 'YoungAdult', 'Youth', ..., 'YoungAdult', 'Senio
# Length: 12
# Categories (4, object): ['Youth' < 'YoungAdult' < 'MiddleAged' < 'Senior']
```

Out[144]:
```
['Youth', 'Youth', 'Youth', 'YoungAdult', 'Youth', ..., 'YoungAdult', 'Senio
r', 'MiddleAged', 'MiddleAged', 'YoungAdult']
Length: 12
Categories (4, object): ['Youth' < 'YoungAdult' < 'MiddleAged' < 'Senior']
```

In [152]:
```
#如果传给cut整数个的箱，来代替显示的箱边，pandas将根据数据中的最小值和最大值，计算出
#考虑一些均匀分布的数据被切成4份的情况：
data = pd.np.random.rand(20)
pd.cut(data, 4, precision = 2)      #precision = 2选项将十进制精度限制在两位。
# [(0.26, 0.45], (0.26, 0.45], (0.63, 0.81], (0.63, 0.81], (0.63, 0.81], ...,
# Length: 20
# Categories (4, interval[float64, right]): [(0.08, 0.26] < (0.26, 0.45] < (0.

#qcut是一个与分箱密切相关的函数，给予样本分位数进行分箱。取决于数据的分布，使用cut通
#由于qcut使用样本的分位数，可以通过qcut获得等长的箱：
data = np.random.randn(1000)        #正态分布
cats = pd.qcut(data, 4)             #切成4份
cats
# [(-3.2359999999999998, -0.749], (-0.103, 0.668], (-0.749, -0.103], (-0.103,
# Length: 1000
# Categories (4, interval[float64, right]): [(-3.2359999999999998, -0.749] < (

pd.value_counts(cats)
# (-3.395, -0.758]     250
# (-0.758, -0.0739]    250
# (-0.0739, 0.635]     250
# (0.635, 3.465]       250
# dtype: int64
```

```
C:\Users\miran\AppData\Local\Temp\ipykernel_72532\1053101203.py:3: FutureWarn
ing: The pandas.np module is deprecated and will be removed from pandas in a
future version. Import numpy directly instead.
  data = pd.np.random.rand(20)
```

Out[152]:
```
(-3.312, -0.629]     250
(-0.629, -0.0232]    250
(-0.0232, 0.619]     250
(0.619, 3.536]       250
dtype: int64
```

In [154]:
```
#与cut类似，可以传入自定义的分位数（0和1之间的数据，包括边）：
pd.qcut(data, [0, 0.1, 0.5, 0.9, 1.])
# [(-1.244, -0.0232], (-0.0232, 1.311], (-1.244, -0.0232], (-1.244, -0.0232],
# Length: 1000
# Categories (4, interval[float64, right]): [(-3.312, -1.244] < (-1.244, -0.02
```

Out[154]:
```
[(-1.244, -0.0232], (-0.0232, 1.311], (-1.244, -0.0232], (-1.244, -0.0232],
(-3.312, -1.244], ..., (-0.0232, 1.311], (-1.244, -0.0232], (1.311, 3.536],
(-1.244, -0.0232], (-0.0232, 1.311]]
Length: 1000
Categories (4, interval[float64, right]): [(-3.312, -1.244] < (-1.244, -0.023
2] < (-0.0232, 1.311] < (1.311, 3.536]]
```

7.2.6 检测和过滤异常值 Detecting and Filtering Outliers

In [174]:
```
#过滤或转换异常值，是应用数组操作的事情。考虑一个具有正态分布数据的data frame:

data = pd.DataFrame(np.random.randn(1000,4))
data.describe
# <bound method NDFrame.describe of              0         1         2
# 0    1.142038 -0.574816  3.063350 -1.059836
# 1   -0.619458  0.067483  0.428145  0.162779
# 2   -0.530945  1.175043  0.436316  0.087275
# 3   -0.566760  0.458664 -1.694724 -1.877735
# 4   -0.398853  0.088756 -0.527560 -1.042355
# ..        ...       ...       ...       ...
# 995  0.950834  1.608913 -0.994935 -0.232550
# 996 -0.702129 -0.378089  1.062553 -1.119628
# 997 -0.087475 -1.014514 -1.785102 -0.618286
# 998  0.767778  1.262318 -1.531322  0.024828
# 999  0.385158  1.557846  0.036737  0.732690

# [1000 rows x 4 columns]>


#假设想找一列中，绝对值大于3的值:
col = data[2]                                       #它提取了DataFrame中的第二列
col[np.abs(col) > 3]                                #找到第二列中 值>3

# 410   -3.146837
# 730   -3.240782
# 821    3.383893
# Name: 2, dtype: float64

#选出所有值大于3或小于3的行，可以对布尔值data frame使用any方法:
data[(np.abs(data) > 3).any(1)]
# 0 1   2   3
# 9 -3.071123   -0.163088   0.099980    -0.197243
# 149   0.685931    3.260466    0.602843    1.058663
# 289   0.378955    -0.221422   3.091872    -0.356299
# 306   3.031398    -0.662224   0.451169    1.851046
# 496   -3.343232   1.775168    1.242600    -0.395018
# 537   -0.006856   1.272517    3.231086    -1.544452
# 564   -0.831633   0.556131    3.245053    0.249727
# 591   -0.066392   -0.186191   3.247500    -0.469210
# 611   1.668031    0.585434    1.174437    -3.090351
# 664   -0.760357   0.371869    0.634024    3.456374
# 703   -0.629071   -1.126303   -0.247052   -3.633301
# 898   1.193264    -3.205424   0.638124    -0.406197
# 905   0.662059    3.777832    -0.033470   -0.509475
# 969   -0.400612   0.488288    -3.004723   -0.940533

#找到绝对值>3的数所在的行：限制了-3到3之间的数值
data[np.abs(data) > 3] = np.sign(data) * 3
data.describe()
# 0 1   2   3
# count 1000.000000 1000.000000 1000.000000 1000.000000
# mean  -0.001102   -0.041514   0.006544    -0.020336
# std   0.970248    1.023402    0.993441    0.997568
# min   -3.000000   -3.000000   -2.970516   -2.909003
# 25%   -0.689065   -0.713156   -0.670724   -0.695272
# 50%   -0.040071   -0.048061   0.020232    -0.004706
```

```
# 75%   0.651710    0.621874    0.689062    0.644225
# max   3.000000    3.000000    3.000000    3.000000

#np.sign(data)根据数据中的值的正负，分别生成1和-1的数值:
np.sign(data).head()
# 0 1   2   3
# 0 -1.0    -1.0    -1.0    -1.0
# 1 -1.0    -1.0    1.0 1.0
# 2 -1.0    1.0 1.0 1.0
# 3 1.0 1.0 1.0 1.0
# 4 -1.0    1.0 -1.0    1.0
```

```
C:\Users\miran\AppData\Local\Temp\ipykernel_72532\404559273.py:31: FutureWarn
ing: In a future version of pandas all arguments of DataFrame.any and Series.
any will be keyword-only.
  data[(np.abs(data) > 3).any(1)]
```

Out[174]:

	0	1	2	3
0	-1.0	-1.0	-1.0	-1.0
1	-1.0	-1.0	1.0	1.0
2	-1.0	1.0	1.0	1.0
3	1.0	1.0	1.0	1.0
4	-1.0	1.0	-1.0	1.0

7.2.7 置换和随机抽样 Permutation and Random Sampling

In [202]:

```
#使用numpy.random.permutation对data frame的series或行进行置换。
#在调用permutation时，根据想要的轴长度产生新顺序的整数数组:

df = pd.DataFrame(np.arange(5 * 4).reshape((5,4)))
sampler = np.random.permutation(5)
sampler
#array([3, 4, 2, 1, 0])
```

Out[202]:

```
array([4, 0, 1, 2, 3])
```

In [236]:
```
#整数数组可以用在基于iloc索引或等价的take函数中:
df = pd.DataFrame(np.arange(5 * 4).reshape((5,4)))
df
#   0   1   2   3
# 0 0   1   2   3
# 1 4   5   6   7
# 2 8   9   10  11
# 3 12  13  14  15
# 4 16  17  18  19

df.take(sampler)                                                              #它根据随机排列的
# 0 1   2   3
# 4 16  17  18  19
# 0 0   1   2   3
# 1 4   5   6   7
# 2 8   9   10  11
# 3 12  13  14  15
```

Out[236]:

	0	1	2	3
4	16	17	18	19
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11
3	12	13	14	15

In [256]:
```
#要选出一个不含有替代值的随机子集，可以使用series和data frame的sample方法：
#具体而言，sample方法通过n参数指定要抽取的行数。在这个例子中，n=3表示从df中随机抽取3

df.sample(n = 3)
# 0 1   2   3
# 1 4   5   6   7
# 0 0   1   2   3
# 4 16  17  18  19

#有5个数，有放回随机拿10次：
#要生成一个带有替代值的样本（允许有重复选择），将replace = True传入sample方法：
choices = pd.Series([5, 7, -1, 6, 4])
draws = choices.sample(n = 10, replace = True)
draws
# 2   -1
# 1    7
# 0    5
# 0    5
# 0    5
# 2   -1
# 3    6
# 1    7
# 0    5
# 2   -1
```

Out[256]:
```
2   -1
1    7
0    5
0    5
0    5
2   -1
3    6
1    7
0    5
2   -1
dtype: int64
```

7.2.8 计算指标/ 虚拟变量 Computing Indicator/Dummy Variables

In [272]:

```
#将分类变量转换为虚拟或指标矩阵，是一种用于建模ml的转换操作。
#如果data frame中 一列有k个不同的值，可以衍生出一个k列值为1和0的矩阵或data frame.
#pandas有一个get_dummies函数用于实现该功能。

df = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'b'],
                   'data1': range(6)})

pd.get_dummies(df['key'])
# a b   c
# 0 0   1   0
# 1 0   1   0
# 2 1   0   0
# 3 0   0   1
# 4 1   0   0
# 5 0   1   0

#如果想在指标data frame的列上，加入前缀，然后与其他数据合并。
#在data1前，加一列key.
#在get_dummies有一个前缀参数用于实现该功能：
dummies = pd.get_dummies(df['key'], prefix = 'key')  #根据DataFrame对象df中的ke
                                                     #prefix='key'参数指定了虚拟
df_with_dummy = df[['data1']].join(dummies)          #通过df[['data1']]选择df中
df_with_dummy
# data1 key_a   key_b   key_c
# 0 0   0   1   0
# 1 1   0   1   0
# 2 2   1   0   0
# 3 3   0   0   1
# 4 4   1   0   0
# 5 5   0   1   0
```

Out[272]:

	data1	key_a	key_b	key_c
0	0	0	1	0
1	1	0	1	0
2	2	1	0	0
3	3	0	0	1
4	4	1	0	0
5	5	0	1	0

In [284]:
```
#如果data frame中，一行属于多个类别，则情况略微复杂。
mnames = ['movie_id', 'title', 'genres']
movies = pd.read_table('C:/Users/miran/lpthw/movies.dat', sep = '::',
                       header = None, names = mnames)
movies[:10]
# movie_id  title   genres
# 0 1   Toy Story (1995)    Animation|Children's|Comedy
# 1 2   Jumanji (1995)  Adventure|Children's|Fantasy
# 2 3   Grumpier Old Men (1995) Comedy|Romance
# 3 4   Waiting to Exhale (1995)    Comedy|Drama
# 4 5   Father of the Bride Part II (1995)  Comedy
# 5 6   Heat (1995) Action|Crime|Thriller
# 6 7   Sabrina (1995)  Comedy|Romance
# 7 8   Tom and Huck (1995) Adventure|Children's
# 8 9   Sudden Death (1995) Action
# 9 10  GoldenEye (1995)    Action|Adventure|Thriller
```

```
C:\Users\miran\AppData\Local\Temp\ipykernel_72532\1532099098.py:3: ParserWarn
ing: Falling back to the 'python' engine because the 'c' engine does not supp
ort regex separators (separators > 1 char and different from '\s+' are interp
reted as regex); you can avoid this warning by specifying engine='python'.
  movies = pd.read_table('C:/Users/miran/lpthw/movies.dat', sep = '::',
```

Out[284]:

	movie_id	title	genres
0	1	Toy Story (1995)	Animation|Children's|Comedy
1	2	Jumanji (1995)	Adventure|Children's|Fantasy
2	3	Grumpier Old Men (1995)	Comedy|Romance
3	4	Waiting to Exhale (1995)	Comedy|Drama
4	5	Father of the Bride Part II (1995)	Comedy
5	6	Heat (1995)	Action|Crime|Thriller
6	7	Sabrina (1995)	Comedy|Romance
7	8	Tom and Huck (1995)	Adventure|Children's
8	9	Sudden Death (1995)	Action
9	10	GoldenEye (1995)	Action|Adventure|Thriller

In [285]:
```
#为每个电影流派添加指标变量，需要进行一些数据处理。首先提取所有不同的流派列表:
all_generes = []
for x in movies.genres:
    all_generes.extend(x.split('|'))
genres = pd.unique(all_generes)
genres
# array(['Animation', "Children's", 'Comedy', 'Adventure', 'Fantasy',
#        'Romance', 'Drama', 'Action', 'Crime', 'Thriller', 'Horror',
#        'Sci-Fi', 'Documentary', 'War', 'Musical', 'Mystery', 'Film-Noir',
#        'Western'], dtype=object)
```

Out[285]:
```
array(['Animation', "Children's", 'Comedy', 'Adventure', 'Fantasy',
       'Romance', 'Drama', 'Action', 'Crime', 'Thriller', 'Horror',
       'Sci-Fi', 'Documentary', 'War', 'Musical', 'Mystery', 'Film-Noir',
       'Western'], dtype=object)
```

In [301]:
```
#使用全0的data frame是构建data frame的一种方式:
zero_matrix = np.zeros((len(movies), len(genres)))
dummies = pd.DataFrame(zero_matrix, columns = genres)
dummies
# Animation Children's  Comedy  Adventure   Fantasy Romance Drama   Action  Cr
# 0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
# 1 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
# 3882  0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.
# 3883 rows × 18 columns

movies.genres
# 0        Animation|Children's|Comedy
# 1       Adventure|Children's|Fantasy
# 2                     Comedy|Romance
# 3                       Comedy|Drama
# 4                             Comedy
#                         ...
# 3878                          Comedy
# 3879                           Drama
# 3880                           Drama
# 3881                           Drama
# 3882                  Drama|Thriller
# Name: genres, Length: 3883, dtype: object

gen = movies.genres[0]                                          #选择该列的第一个元素
gen.split('|')
# #['Animation', "Children's", 'Comedy']

dummies.columns.get_indexer(gen.split('|'))
# # #array([0, 1, 2], dtype=int64)
```

Out[301]:
```
array([0, 1, 2], dtype=int64)
```

In [309]:
```
#使用.loc根据这些指标来设置值:
for i, gen in enumerate(movies.genres):
    indices = dummies.columns.get_indexer(gen.split('|'))
    dummies.iloc[i, indices] = 1

#可以将结果与movies进行联合:
movies_windic = movies.join(dummies.add_prefix('Genre_'))
movies_windic.iloc[0]
# movie_id                                       1
# title                           Toy Story (1995)
# genres               Animation|Children's|Comedy
# Genre_Animation                              1.0
# Genre_Children's                             1.0
# Genre_Comedy                                 1.0
# Genre_Adventure                              0.0
# Genre_Fantasy                                0.0
# Genre_Romance                                0.0
# Genre_Drama                                  0.0
# Genre_Action                                 0.0
# Genre_Crime                                  0.0
# Genre_Thriller                               0.0
# Genre_Horror                                 0.0
# Genre_Sci-Fi                                 0.0
# Genre_Documentary                            0.0
# Genre_War                                    0.0
# Genre_Musical                                0.0
# Genre_Mystery                                0.0
# Genre_Film-Noir                              0.0
# Genre_Western                                0.0
# Name: 0, dtype: object


#将get_dummies与cut等离散化函数结合使用时统计应用的方法:
np.random.seed(12345)        #将种子设置为固定的数值，比如12345，可以保证每次运行程
values = np.random.rand(10)           #随机的10个数
values
# array([0.92961609, 0.31637555, 0.18391881, 0.20456028, 0.56772503,
#        0.5955447 , 0.96451452, 0.6531771 , 0.74890664, 0.65356987])
bins = [0, 0.2, 0.4, 0.6, 0.8, 1]        #设置5个bins
pd.get_dummies(pd.cut(values, bins))    #把values归在每个bins
#   (0.0, 0.2]  (0.2, 0.4]  (0.4, 0.6]  (0.6, 0.8]  (0.8, 1.0]
# 0 0   0   0   0   1
# 1 0   1   0   0   0
# 2 1   0   0   0   0
# 3 0   1   0   0   0
# 4 0   0   1   0   0
# 5 0   0   1   0   0
# 6 0   0   0   0   1
# 7 0   0   0   1   0
# 8 0   0   0   1   0
# 9 0   0   0   1   0
```

Out[309]:

	(0.0, 0.2]	(0.2, 0.4]	(0.4, 0.6]	(0.6, 0.8]	(0.8, 1.0]
0	0	0	0	0	1
1	0	1	0	0	0
2	1	0	0	0	0
3	0	1	0	0	0
4	0	0	1	0	0
5	0	0	1	0	0
6	0	0	0	0	1
7	0	0	0	1	0
8	0	0	0	1	0
9	0	0	0	1	0

7.3 字符串操作 String Manipulation

7.3.1 字符串对象方法 Python Built-In String Object Methods

In [338]:
```
#一个逗号分隔的字符串，可以使用split拆分
val = 'a,b,  guido'
val.split(',')
#['a', 'b', '  guido']

#split和strip一起使用，用于清除空格（包括换行）：
pieces = [x.strip() for x in val.split(',')]
pieces
#['a', 'b', 'guido']

#这些子字符串可以使用加法与两个冒号分隔符连接在一起:
first, second, third = pieces
first + '::' + second + '::' + third
#'a::b::guido'

#这并不是一个使用的通用方法，在字符串'::' 的join方法传入一个列表或元组更快:
'::'.join(pieces)
#'a::b::guido'

#其他方法涉及子字符串定位。使用in最佳，index，find也可以实现同样功能。
'guido' in val
#True

val.index(',')
# 1

val.find(':')
#-1

#注意find和index的区别，在于index在字符串没有找到时，会抛出一个异常
#val.index(':')
# ---------------------------------------------------------------------------
# ValueError                                Traceback (most recent call last)
# Cell In[322], line 2
#       1 #注意find和index的区别，在于index在字符串没有找到时，会抛出一个异常
# ----> 2 val.index(':')
# ValueError: substring not found
```

Out[338]: -1

In [339]:
```
#count返回的是某个特定的子字符串在字符串中出现的次数:
val.count(',')
#2

#替换分隔符或改分隔符为空格
# #replace将用一种模式替代另一种模式。也用于传入空字符串，来删除某个模式。
val.replace(',','::')
#'a::b::  guido'

val.replace(',','')
#'ab  guido'
```

Out[339]: 'ab guido'

In [340]:
```
#python内建字符串方法:
#count: 返回字符串在字符串中的非重叠出现次数
#endswith:如果字符串以后缀结尾则返回True
#startswith: 如果字符串以后缀开始则返回True
#join: 使用字符串作为间隔符，用于粘合其他字符串的序列
#index: 如果在字符串中找到，则返回子字符串中第一个字符的位置，如果找不到则引发value
#find: 返回字符串中第一个出现子字符的第一个字符的位置，类似index，如果没有找到，则返
#rfind: 返回子字符串在字符串最后一次出现时，第一个字符的位置，如果没有找到，则返回-1
#replace: 使用一个字符串替代另一个字符串
#strip, rstrip, lstrip: 修建空白，包括换行符，相当于对每个元素进行x.strip()
#split: 使用分隔符将字符串拆分为子字符串的列表
#lower: 转换为全小写
#upper: 转换为全大写
#casefold: 转换为小写，并将任何特定于区域的变量字符组合转换为常见的可比较形式
#ljust, rjust: 左对齐或右对齐，用空格填充字符串的相反侧，以返回具有最小宽度的字符串
```

7.3.2 正则表达式 Regular Expressions

In [392]:
```
#正则表达式提供了一种在文本中，灵活查找或匹配字符串模式的方法。
#单个表达式通常被称为regex，根据正则表达式语言形成的字符串。
#python内建的re模块，适用于将正则表达式，应用到字符串上的库。

#re模块有3个主题，模式匹配，替代，拆分。
#eg. 将含有多种空白字符（制表符，空格，换行符）的字符串拆分开。
#描述一个或多个空白字符的正则表达式是\s+

import re
text = "foo    bar\t baz  \tqux"
text
#'foo    bar\t baz  \tqux'
re.split('\s+', text)
#['foo', 'bar', 'baz', 'qux']

#当调用re.split('\s+', text)，正则表达式首先会被编译。然后正则表达式的split方法在传
#可以使用re.complie自行编译，形成一个可复用的正则表达式对象:
regex = re.compile('\s+')
regex.split(text)
#['foo', 'bar', 'baz', 'qux']
```

Out[392]:
```
['dave@google.com', 'steve@gmail.com', 'rob@gmail.com', 'ryan@yahoo.com']
```

In [ ]:
```
#如果想获得的是一个所有匹配正则表达式的模式的列表，可以使用findall方法:
regex.findall(text)
#['    ', '\t ', '  \t']

#如果需要将相同的表达式应用到多个字符串上，推荐使用re.compile穿件一个正则表达式对象
#match和search与findall相关性很大。
#findall返回的是字符串中所有匹配项，而search返回的仅仅第一个匹配项。
#match更为严格，只在字符串的起始位置进行匹配。

text = """Dave dave@google.com
Steve steve@gmail.com
Rob rob@gmail.com
Ryan ryan@yahoo.com
"""
pattern = r'[A-Z0-9.%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}'
#re.IGNORECASE使正则表达式不区分大小写
regex = re.compile(pattern, flags = re.IGNORECASE)

#使用findall会生成一个电子邮件地址的列表:
regex.findall(text)
#['dave@google.com', 'steve@gmail.com', 'rob@gmail.com', 'ryan@yahoo.com']
```

In [359]:

```
#search返回的是文本中第一个匹配到的电邮地址，对于正则表达式，匹配对象只告诉模式在字符
m = regex.search(text)
m
#<re.Match object; span=(5, 20), match='dave@google.com'>
text[m.start():m.end()]
#'dave@google.com'

#regex.match只在模式出现于字符串起始位置时，进行匹配，如果没有匹配到，返回None：
print(regex.match(text))
#None

#sub会返回一个新的字符串，原字符串的模式会被一个新的字符串替代：
print(regex.sub('REDACTED', text))
# None
# Dave REDACTED
# Steve REDACTED
# Rob REDACTED
# Ryan REDACTED
```

```
None
Dave REDACTED
Steve REDACTED
Rob REDACTED
Ryan REDACTED
```

In [364]:

```
#假设项查找电邮地址，将每个地址氛围3个部分，用户名，域名，域名后缀。
pattern = r'([A-Z0-9.%+-]+)@([A-Z0-9.-]+)\.([A-Z]{2,4})'
regex = re.compile(pattern, flags = re.IGNORECASE)
#修改后的正则表达式产生的匹配对象的groups方法，返回的是模式组件的元组
m = regex.match('wesm@bright.net')
m.groups()
#('wesm', 'bright', 'net')

#当模式可以分组时，findall返回的是包含元组的列表：
regex.findall(text)
# [('dave', 'google', 'com'),
#  ('steve', 'gmail', 'com'),
#  ('rob', 'gmail', 'com'),
#  ('ryan', 'yahoo', 'com')]

#sub可以使用特殊符号：\1代表第一个匹配分组，\2代表第二个匹配分组
print(regex.sub(r'Username: \1, Domain: \2, Suffix: \3', text))
# Dave Username: dave, Domain: google, Suffix: com
# Steve Username: steve, Domain: gmail, Suffix: com
# Rob Username: rob, Domain: gmail, Suffix: com
# Ryan Username: ryan, Domain: yahoo, Suffix: com
```

```
Dave Username: dave, Domain: google, Suffix: com
Steve Username: steve, Domain: gmail, Suffix: com
Rob Username: rob, Domain: gmail, Suffix: com
Ryan Username: ryan, Domain: yahoo, Suffix: com
```

In [ ]:
```
#正则表达式方法
# findall: 将字符串中所有非重叠匹配模式一列表形式返回
# finditer: 返回的是迭代器
# match: 字符串其实位置匹配，若匹配上，返回一个匹配对象，否则返回None
# search: 如扫描到了返回匹配对象，search方法匹配事字符串任意位置，不仅仅是字符串的起
# split: 根据模式，将字符串拆分为多个部分
# sub, subn: 用替换表达式替换字符串所有的匹配或低n个出现匹配串
```

7.3.3 pandas中的向量化字符串函数 String Functions in pandas

In [394]:
```
#杂乱的数据用于分析通常需要大量的字符串处理和正则化。

data = {'Dave':'dave@google.com', 'Steve': 'steve@gmail.com',
        'Rob': 'rob@gmail.com', 'Wes': np.nan}
data = pd.Series(data)
data
# Dave     dave@google.com
# Steve    steve@gmail.com
# Rob        rob@gmail.com
# Wes                  NaN
# dtype: object
data.isnull()
# Dave     False
# Steve    False
# Rob      False
# Wes       True
# dtype: bool
```

Out[394]:
```
Dave     False
Steve    False
Rob      False
Wes       True
dtype: bool
```

In [396]:

```
#可以使用data.map将字符串和有效的正则表达式方法（以lambda或其他函数的方式传递），应F
#series有面向数组的方法用于跳过na值的字符串操作。
#str.contain来检查每个电邮地址是否含有'gmail'

data.str.contains('gmail')
# Dave     False
# Steve     True
# Rob       True
# Wes        NaN
# dtype: object

#正则表达式也可以结合任意re模块选项使用，例如IGNORECASE:
pattern
#'([A-Z0-9.%+-]+)@([A-Z0-9.-]+)\\.([A-Z]{2,4})'
data.str.findall(pattern, flags = re.IGNORECASE)
# Dave     [(dave, google, com)]
# Steve    [(steve, gmail, com)]
# Rob        [(rob, gmail, com)]
# Wes                          NaN
# dtype: object
```

Out[396]:

```
Dave     [dave@google.com]
Steve    [steve@gmail.com]
Rob        [rob@gmail.com]
Wes                    NaN
dtype: object
```

In [388]:

```
#多种方法可以进行向量化的元素检索，可以使用str.get或在str内部索引:
matches = data.str.match(pattern, flags = re.IGNORECASE)
matches
# Dave     True
# Steve    True
# Rob      True
# Wes       NaN
# dtype: object

#要访问嵌入式列表的元素，可以将索引传递给这些函数的任意一个:
#matches.str.get(1)
#matches.str[0]
#matches.str[:5]
```

Out[388]:

```
Dave     True
Steve    True
Rob      True
Wes       NaN
dtype: object
```

In [ ]:
```
#部分向量化字符串方法列表:
# cat
# contains
# count
# extract
# endswith
# startwith
# findall
# get
# isalnum
# isalhpa
# isdecimal
# isdigit
# islower
# isnumeric
# isupper
# join
# len
# lower, upper
# match
# pad
# center
# repeat
# replace
# slice
# split
# strip
# rstrip
# lstrip
```

7.5 分类数据 Categorical Data

Background and Motivation

Categorical Extension Type in pandas

Computations with Categoricals

Better performance with categoricals

Categorical Methods

Creating dummy variables for modeling

7.6 Conclusion