DataAnalysisLearning-Data Cleaning and Preparation

July 5, 2019

In [1]:
```
# -*-* codig: utf-8 -*-
# @author: tongzi
# @description: Data Cleanning and Preparation
# @created date: 2019/07/04
# @license
```

In [2]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
```

```
C:\Anaconda3\lib\importlib\_bootstrap.py:219: RuntimeWarning: numpy.ufunc size changed, may inc
  return f(*args, **kwds)
C:\Anaconda3\lib\importlib\_bootstrap.py:219: RuntimeWarning: numpy.ufunc size changed, may inc
  return f(*args, **kwds)
C:\Anaconda3\lib\importlib\_bootstrap.py:219: RuntimeWarning: numpy.ufunc size changed, may inc
  return f(*args, **kwds)
C:\Anaconda3\lib\importlib\_bootstrap.py:219: RuntimeWarning: numpy.ufunc size changed, may inc
  return f(*args, **kwds)
C:\Anaconda3\lib\importlib\_bootstrap.py:219: RuntimeWarning: numpy.ufunc size changed, may inc
  return f(*args, **kwds)
```

In this chapter, we will discuss tools for missing data, duplicate data, string manipulation, and some other analytical data transformations.

0.0.1 Handling Missing Data

Missing data occurs commonly in many data analysis applications. One of the goal of pandas is to make working with missing data as painless as possible. For example, all of the descriptive statistic on pandas objects exclude missing data by default.
For numeric data, pandas uses the float-point value **NaN** (Not a Number) to represent missing data. We call this a sentinel value that can be easily detected:

In [3]:
```
string_data = pd.Series(['aardvark', 'artichoke', np.nan, 'avocado'])
```

In [4]:
```
string_data
```

```
Out[4]: 0     aardvark
        1    artichoke
        2          NaN
        3      avocado
        dtype: object
```

```
In [5]: string_data.isnull()
```

```
Out[5]: 0    False
        1    False
        2     True
        3    False
        dtype: bool
```

Below is a table that lists some functions related to missing data handling:

0.0.2 Filtering Out Missing Data

```
In [6]: from numpy import nan as NA

In [7]: data = pd.Series([1, NA, 3.5, 7])

In [8]: data.dropna()
```

```
Out[8]: 0    1.0
        2    3.5
        3    7.0
        dtype: float64
```

As we can see above, the *dropna*() method returns the Series with only the non-null data and index values. This is equivalent to:

```
In [9]: data[data.notnull()]
```

```
Out[9]: 0    1.0
        2    3.5
        3    7.0
        dtype: float64
```

With DataFrame, things are a bit more different, we may want to drop rows or columns that are all NA or only those containing any NAs. *dropna*() method by default drops any row containing a missing value:

```
In [10]: data = pd.DataFrame([[1., 6.5, 3.], [1., NA, NA],
     ....: [NA, NA, NA], [NA, 6.5, 3.]])

In [11]: data
```

```
Out[11]:      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
```

```
In [12]: data.dropna()
```

```
Out[12]:      0    1    2
         0  1.0  6.5  3.0
```

DataFrame*dropna*()

```
In [15]: #
         data.dropna(axis='columns')
```

```
Out[15]: Empty DataFrame
         Columns: []
         Index: [0, 1, 2, 3]
```

```
In [16]: data
```

```
Out[16]:      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
```

Passing *how*='all', will only drop rows that are all NA:

```
In [17]: data.dropna(how='all')
```

```
Out[17]:      0    1    2
         0  1.0  6.5  3.0
         1  1.0  NaN  NaN
         3  NaN  6.5  3.0
```

```
In [18]: #
         data[4] = NA
```

```
In [19]: data
```

```
Out[19]:      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
```

```
In [20]: data.dropna(axis='columns', how='all')
```

```
Out[20]:      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
```

DataFrame*thresh*

```
In [21]: df = pd.DataFrame(np.random.randn(7, 3))

In [22]: df

Out[22]:           0         1         2
         0 -0.377788 -0.099363  0.773704
         1 -0.807716 -0.697565  0.309078
         2 -1.002575 -0.002337  0.970096
         3  1.768859 -0.646667  0.167954
         4  0.687819  3.742656 -0.482712
         5 -0.035491  0.267234  1.161451
         6 -0.654256 -0.612303  2.564999

In [23]: # 41NaN
         df.iloc[:4, 1] = NA

In [26]: # 22NaN
         df.iloc[:2, 2] = NA

In [27]: df

Out[27]:           0         1         2
         0 -0.377788       NaN       NaN
         1 -0.807716       NaN       NaN
         2 -1.002575       NaN  0.970096
         3  1.768859       NaN  0.167954
         4  0.687819  3.742656 -0.482712
         5 -0.035491  0.267234  1.161451
         6 -0.654256 -0.612303  2.564999

In [28]: # NaN
         df.dropna()

Out[28]:           0         1         2
         4  0.687819  3.742656 -0.482712
         5 -0.035491  0.267234  1.161451
         6 -0.654256 -0.612303  2.564999

In [29]: # thresh
         #
         df.dropna(thresh=2)
```

```
Out[29]:           0         1         2
         2 -1.002575       NaN  0.970096
         3  1.768859       NaN  0.167954
         4  0.687819  3.742656 -0.482712
         5 -0.035491  0.267234  1.161451
         6 -0.654256 -0.612303  2.564999
```

0.0.3 Filling In Missing Data

Rather than filtering out missing data (potentially discarding other data along with it), we may want to fill in the 'hole' in any number of ways. For most purposes, the *fillna*() method is the workhorse function to use:

```
In [30]: # 0
         df.fillna(0)
```

```
Out[30]:           0         1         2
         0 -0.377788  0.000000  0.000000
         1 -0.807716  0.000000  0.000000
         2 -1.002575  0.000000  0.970096
         3  1.768859  0.000000  0.167954
         4  0.687819  3.742656 -0.482712
         5 -0.035491  0.267234  1.161451
         6 -0.654256 -0.612303  2.564999
```

```
In [32]: #
         # 1102200
         df.fillna({1:10, 2:200})
```

```
Out[32]:           0          1           2
         0 -0.377788  10.000000  200.000000
         1 -0.807716  10.000000  200.000000
         2 -1.002575  10.000000    0.970096
         3  1.768859  10.000000    0.167954
         4  0.687819   3.742656   -0.482712
         5 -0.035491   0.267234    1.161451
         6 -0.654256  -0.612303    2.564999
```

fillna() returns a new object, but we can modify the existing object in-place by passing the argument *inplace*:

```
In [33]: df.fillna(0, inplace=True)
```

```
In [34]: df
```

```
Out[34]:           0         1         2
         0 -0.377788  0.000000  0.000000
         1 -0.807716  0.000000  0.000000
         2 -1.002575  0.000000  0.970096
```

```
3  1.768859  0.000000  0.167954
4  0.687819  3.742656 -0.482712
5 -0.035491  0.267234  1.161451
6 -0.654256 -0.612303  2.564999
```

The same interpolations available for reindexing can be used for *fillna*(): >*fillna*()

```
In [35]: df = pd.DataFrame(np.random.randn(6, 3))

In [36]: # implicite indexing
         df.iloc[2:, 1] = NA # 21NaN

In [39]: # implicite indexing
         df.iloc[4:, 2] = NA # 42NaN

In [40]: df

Out[40]:           0         1         2
         0 -0.018569  0.250788  0.468610
         1 -1.147574  1.921718  1.831329
         2 -1.351726       NaN -0.247973
         3  0.824898       NaN  0.608928
         4 -0.540984       NaN       NaN
         5 -0.748586       NaN       NaN

In [41]: # 'ffill'forward fill,
         df.fillna(method='ffill')

Out[41]:           0         1         2
         0 -0.018569  0.250788  0.468610
         1 -1.147574  1.921718  1.831329
         2 -1.351726  1.921718 -0.247973
         3  0.824898  1.921718  0.608928
         4 -0.540984  1.921718  0.608928
         5 -0.748586  1.921718  0.608928

In [42]: # 'ffill'forward fill,,
         # 2
         df.fillna(method='ffill', limit=2)

Out[42]:           0         1         2
         0 -0.018569  0.250788  0.468610
         1 -1.147574  1.921718  1.831329
         2 -1.351726  1.921718 -0.247973
         3  0.824898  1.921718  0.608928
         4 -0.540984       NaN  0.608928
         5 -0.748586       NaN  0.608928
```

With the *fillna*(), we can do lots of things with a little creativity. For example, we might pass the mean or median value of a Series:

```
In [43]: data = pd.Series([1., NA, 3.5, NA, 7])

In [44]: data

Out[44]: 0    1.0
         1    NaN
         2    3.5
         3    NaN
         4    7.0
         dtype: float64

In [45]: #(1.0 + 3.5 + 7.0) / 3
         data.fillna(data.mean())

Out[45]: 0    1.000000
         1    3.833333
         2    3.500000
         3    3.833333
         4    7.000000
         dtype: float64

In [48]: (1.0 + 3.5 + 7.0) / 3

Out[48]: 3.8333333333333335
```

fillna()

0.0.4 Data Transformation

So far in this chapter, we've been concerned with rearranging data. Filtering, cleaning and other transformations are another class of important operations.

Remobing Duplicates Duplicate row may be found in a DataFrame for any number of reasons. For example:

```
In [49]: data = pd.DataFrame({'k1': ['one', 'two'] * 3 + ['two'],
         ....: 'k2': [1, 1, 2, 3, 3, 4, 4]})

In [50]: data

Out[50]:     k1  k2
         0  one   1
         1  two   1
         2  one   2
         3  two   3
         4  one   3
         5  two   4
         6  two   4
```

The DataFrame method *duplicated*() returns a Series indicating whether each row is a dulicate (has been observed in a previous row) or not:

```
In [51]: data.duplicated()

Out[51]: 0    False
         1    False
         2    False
         3    False
         4    False
         5    False
         6     True
         dtype: bool
```

Relately, the DataFrame *drop_duplicates*() returns a DataFrame where the duplicated array is False:

```
In [52]: data.drop_duplicates()

Out[52]:     k1  k2
         0  one   1
         1  two   1
         2  one   2
         3  two   3
         4  one   3
         5  two   4
```

```
In [53]: data['v1'] = range(7)

In [54]: data

Out[54]:     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

In [56]: # k1
         data.drop_duplicates(['k1'])

Out[56]:     k1  k2  v1
         0  one   1   0
         1  two   1   1
```

duplicated()*drop_duplicates*()*keep*='last'

```
In [57]: data.drop_duplicates(['k1', 'k2'], keep='last')

Out[57]:     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
```

Transformationg Data Using a Function or Mapping SeriesDataFrame

```
In [58]: data = pd.DataFrame({'food': ['bacon', 'pulled pork', 'bacon',
         ....: 'Pastrami', 'corned beef', 'Bacon',
         ....: 'pastrami', 'honey ham', 'nova lox'],
         ....: 'ounces': [4, 3, 12, 6, 7.5, 8, 3, 5, 6]})

In [59]: data

Out[59]:           food  ounces
         0        bacon     4.0
         1  pulled pork     3.0
         2        bacon    12.0
         3     Pastrami     6.0
         4  corned beef     7.5
         5        Bacon     8.0
         6     pastrami     3.0
         7    honey ham     5.0
         8     nova lox     6.0
```

>Suppose we want to add a column indicating the type of the animal that each food came from.

```
In [60]: meat_to_animal = {
         'bacon': 'pig',
         'pulled pork': 'pig',
         'pastrami': 'cow',
         'corned beef': 'cow',
         'honey ham': 'pig',
         'nova lox': 'salmon'
         }
```

The *map*() method on a Series accepts a function or dict-like containing a mapping, but here we have a problem in that some of the meats are capitalized and others are not. Thus, we need to convert each value to lowercase using the *Series.str.lower*() method:

```
In [61]: lowercased = data['food'].str.lower()

In [62]: lowercased
```

```
Out[62]: 0          bacon
         1    pulled pork
         2          bacon
         3       pastrami
         4    corned beef
         5          bacon
         6       pastrami
         7      honey ham
         8       nova lox
         Name: food, dtype: object
```

```
In [63]: # animal
         data['animal'] = lowercased.map(meat_to_animal)
```

```
In [64]: data
```

```
Out[64]:           food  ounces  animal
         0        bacon     4.0     pig
         1  pulled pork     3.0     pig
         2        bacon    12.0     pig
         3     Pastrami     6.0     cow
         4  corned beef     7.5     cow
         5        Bacon     8.0     pig
         6     pastrami     3.0     cow
         7    honey ham     5.0     pig
         8     nova lox     6.0  salmon
```

We could also pass a function that does the same work:

```
In [67]: data['food'].map(lambda x: meat_to_animal[x.lower()])
```

```
Out[67]: 0       pig
         1       pig
         2       pig
         3       cow
         4       cow
         5       pig
         6       cow
         7       pig
         8    salmon
         Name: food, dtype: object
```

Using *map*() method is a convenient way to perform element-wise transformation and other related data cleaning-related operations.

Replacing Values *fillna*()*map*()*replace*()Let's consider this Series:

```
In [69]: data = pd.Series([1., -999., 2., -999., -1000., 3.])
```

```
In [70]: data

Out[70]: 0       1.0
         1    -999.0
         2       2.0
         3    -999.0
         4   -1000.0
         5       3.0
         dtype: float64
```

-999.0pandas*replace*(): inplace=True.

```
In [71]: data.replace(-999, np.nan)

Out[71]: 0       1.0
         1       NaN
         2       2.0
         3       NaN
         4   -1000.0
         5       3.0
         dtype: float64
```

If we want to replace multile values at once, we instead pass a list and then the substitute value ():

```
In [72]: data.replace([-999, -1000], np.nan)

Out[72]: 0    1.0
         1    NaN
         2    2.0
         3    NaN
         4    NaN
         5    3.0
         dtype: float64
```

To use a different replacement for each value, pass a list of substitute:

```
In [74]: # -999np.nan
         # -10000
         data.replace([-999, -1000], [np.nan, 0])

Out[74]: 0    1.0
         1    NaN
         2    2.0
         3    NaN
         4    0.0
         5    3.0
         dtype: float64
```

The argument passed can also be a dict:

```
In [75]: data.replace({-999: np.nan, -1000:0})

Out[75]: 0    1.0
         1    NaN
         2    2.0
         3    NaN
         4    0.0
         5    3.0
         dtype: float64
```

The *data.replace*() is distinct from *data.str.replace*(), which performs string substitute element-wise.

Renaming Axis Indexes SeriesDataFrame

```
In [76]:  data = pd.DataFrame(np.arange(12).reshape((3, 4)),
         ....: index=['Ohio', 'Colorado', 'New York'],
         ....: columns=['one', 'two', 'three', 'four'])

In [77]: data

Out[77]:           one  two  three  four
         Ohio        0    1      2     3
         Colorado    4    5      6     7
         New York    8    9     10    11
```

Series*map*()

```
In [79]: #
         data.index.map(lambda x: x.upper())

Out[79]: Index(['OHIO', 'COLORADO', 'NEW YORK'], dtype='object')



In [81]: data.index = data.index.map(lambda x: x.upper())

In [82]: data

Out[82]:           one  two  three  four
         OHIO        0    1      2     3
         COLORADO    4    5      6     7
         NEW YORK    8    9     10    11
```

If we want to create a transformed version of dataset without modifying the original, a useful method is *rename*():

```
In [85]: #
         #
         data.rename(index=str.title, columns=str.upper)
```

```
Out[85]:           ONE  TWO  THREE  FOUR
         Ohio        0    1      2     3
         Colorado    4    5      6     7
         New York    8    9     10    11
```

rename()

```
In [88]: data

Out[88]:           one  two  three  four
         OHIO        0    1      2     3
         COLORADO    4    5      6     7
         NEW YORK    8    9     10    11

In [89]: data.rename(index={'NEW YORK': 'Nanning'},
                    columns={'four': 4})

Out[89]:           one  two  three   4
         OHIO        0    1      2   3
         COLORADO    4    5      6   7
         Nanning     8    9     10  11
```

rename() method saves us from the chore () of copying the DataFrame manually and assigning to its index and columns attributes. By passing the argument *inplace*, we can modify a dataset in-place:

```
In [90]: data.rename(index={'OHIO':'Liuzhou'}, inplace=True)

In [91]: data

Out[91]:           one  two  three  four
         Liuzhou     0    1      2     3
         COLORADO    4    5      6     7
         NEW YORK    8    9     10    11
```

Discretization and Binning

Continuous data is discretized or otherwise separated into "bins" for analysis. Suppose we have data about a group of people in a study, and we want to group them into discrete age buckets ():

```
In [92]: ages = [20, 22, 25, 27, 21, 23, 37, 31, 61, 45, 41, 32]
```

Let's divide these into bins of 18 to 25, 26 to 35, 36 to 60, and finally 61 and older. To do so, we have to use *cut*(), a method in pandas:

```
In [93]: bins = [18, 25 ,35, 60, 100]

In [94]: cats = pd.cut(ages, bins)

In [95]: cats
```

```
Out[95]: [(18, 25], (18, 25], (18, 25], (25, 35], (18, 25], ..., (25, 35], (60, 100], (35, 60]
         Length: 12
         Categories (4, interval[int64]): [(18, 25] < (25, 35] < (35, 60] < (60, 100]]
```

The object pandas returns is a special Categorical object. The object's *codes* attribute contains a categories array specifying the distinct category names along with a labeling for the *ages* data:

```
In [97]: # 40,1,2,3,4
         cats.codes

Out[97]: array([0, 0, 0, 1, 0, 0, 2, 1, 3, 2, 2, 1], dtype=int8)

In [99]: #
         cats.categories

Out[99]: IntervalIndex([(18, 25], (25, 35], (35, 60], (60, 100]],
                       closed='right',
                       dtype='interval[int64]')
```

pandas.value_counts()

```
In [100]: pd.value_counts(cats)

Out[100]: (18, 25]     5
          (35, 60]     3
          (25, 35]     3
          (60, 100]    1
          dtype: int64
```

Consistent with mathematical notation for intervals (), a parenthesis means that the side is open, while the square bracket mean it is closed (inclusive). We can change which side is closed by passing *right*=False:

```
In [101]: pd.cut(ages, [18, 26, 36, 61, 100], right=False)

Out[101]: [[18, 26), [18, 26), [18, 26), [26, 36), [18, 26), ..., [26, 36), [61, 100), [36, 61
          Length: 12
          Categories (4, interval[int64]): [[18, 26) < [26, 36) < [36, 61) < [61, 100)]
```

We can also pass our own bin names by passing a list or an array to the *labels* option:

```
In [102]: group_names = ["Youth", "YoungAdult", "MiddleAged", "Senior"]

In [103]: pd.cut(ages, bins, labels=group_names)

Out[103]: [Youth, Youth, Youth, YoungAdult, Youth, ..., YoungAdult, Senior, MiddleAged, Middle
          Length: 12
          Categories (4, object): [Youth < YoungAdult < MiddleAged < Senior]
```

If we pass an integer number of bins to the *cut*() method instead of explicit edges, it will compute the equal-length bins based on the miminum and maximum values in the data. Consider the case of some uniformly distributed data chopped into fourths:

```
In [104]: data = np.random.rand(20)

In [106]: # data4bins
          # 2
          pd.cut(data, 4, precision=2)

Out[106]: [(0.0023, 0.25], (0.49, 0.74], (0.0023, 0.25], (0.0023, 0.25], (0.0023, 0.25], ...,
          Length: 20
          Categories (4, interval[float64]): [(0.0023, 0.25] < (0.25, 0.49] < (0.49, 0.74] < ((
```

The *precision*=2 option limits the decimal precision to two digits.

A closely related function, *qcut*(), bins the data on sample quantiles. Depending on the distribution of the data, using *cut*() method will not usually result in each bin having same number of data point. Since *qcut*() uses sample quantiles instead, by definition you will obtain roughly equal-size bins:

```
In [107]: data = np.random.randn(1000) # normal distribution

In [116]: cats = pd.qcut(data, 4) # cut into quartile

In [117]: cats

Out[117]: [(-0.00142, 0.631], (-3.009, -0.611], (-0.611, -0.00142], (-0.611, -0.00142], (-0.00
          Length: 1000
          Categories (4, interval[float64]): [(-3.009, -0.611] < (-0.611, -0.00142] < (-0.0014

In [118]: pd.value_counts(cats)

Out[118]: (0.631, 3.77]         250
          (-0.00142, 0.631]     250
          (-0.611, -0.00142]    250
          (-3.009, -0.611]      250
          dtype: int64
```

Similar to *cut*(), we can pass our own quantiles (numbers between 0 and 1, inclusive):

```
In [119]: pd.qcut(data, [0, 0.1, 0.5, 0.6, 0.9, 1.])

Out[119]: [(-0.00142, 0.215], (-3.009, -1.253], (-1.253, -0.00142], (-1.253, -0.00142], (-0.00
          Length: 1000
          Categories (5, interval[float64]): [(-3.009, -1.253] < (-1.253, -0.00142] < (-0.0014
```

Detecting and Filtering Outliers ()

```
In [120]: data = pd.DataFrame(np.random.randn(1000, 4))

In [121]: data.describe()
```

```
Out[121]:                 0            1            2            3
          count  1000.000000  1000.000000  1000.000000  1000.000000
          mean      0.062161     0.010908    -0.012074     0.010671
          std       0.994697     1.007757     0.964354     0.999827
          min      -3.299686    -2.952604    -3.579078    -3.577651
          25%      -0.620926    -0.645775    -0.700070    -0.668896
          50%       0.014344    -0.017357     0.012803     0.038414
          75%       0.764888     0.677508     0.626660     0.709005
          max       3.192125     3.342354     3.205287     2.542400
```

Suppose we want to find values in one of the columns exceding 3 in absolute value:

```
In [122]: col = data[2]

In [123]: col[np.abs(col) > 3]

Out[123]: 189    3.109010
          389   -3.579078
          824    3.205287
          Name: 2, dtype: float64
```

3DataFrame*any*()

```
In [124]: data[(np.abs(data) > 3).any(1)]

Out[124]:            0         1         2         3
          189 -0.513485  0.464559  3.109010  0.958465
          389 -0.377858  0.341131 -3.579078 -0.906682
          450  0.516619  0.099933 -0.835361 -3.163249
          599 -2.139750  1.593949  1.175514 -3.435382
          638 -1.383712  3.318922 -0.770849 -2.257575
          668 -0.805004  0.233830  0.284846 -3.577651
          735 -3.299686  0.157850  1.325712 -0.321169
          824  0.266576  2.287888  3.205287  1.195037
          844 -0.756320  3.342354 -2.250097 -1.664915
          852  3.192125 -0.897974  0.696980  0.707895
          953  1.806269  0.304860 -0.447591 -3.053805
```

[-3, 3]3

```
In [125]: data[np.abs(data) > 3] = np.sign(data) * 3

In [126]: data.describe()

Out[126]:                 0            1            2            3
          count  1000.000000  1000.000000  1000.000000  1000.000000
          mean      0.062269     0.010246    -0.011809     0.011901
          std       0.993141     1.005683     0.961369     0.995834
          min      -3.000000    -2.952604    -3.000000    -3.000000
          25%      -0.620926    -0.645775    -0.700070    -0.668896
          50%       0.014344    -0.017357     0.012803     0.038414
          75%       0.764888     0.677508     0.626660     0.709005
          max       3.000000     3.000000     3.000000     2.542400
```

```
In [130]: test = np.random.randint(-10, 10, size=(3,3))

In [136]: test = pd.DataFrame(test)

In [137]: test

Out[137]:    0  1  2
          0 -8 -2 -9
          1 -4  7 -3
          2  0  8 -8

In [139]: test[np.abs(test) > 3] = np.sign(test) * 3

In [140]: test

Out[140]:    0  1  2
          0 -3 -2 -3
          1 -3  3 -3
          2  0  3 -3

In [141]: np.sign(test) * 3

Out[141]:    0  1  2
          0 -3 -3 -3
          1 -3  3 -3
          2  0  3 -3
```

np.sign(data)data-11(shape)data

Permutation and Random Sampling Permuting (randomly reordering) a Series or the rows of a DataFrame is easy to do using the *numpy.random.permutation*(). Calling *permutation*() with the length of the axis you want to permuate produces an array of integers indicating the new ordering:

```
In [142]: df = pd.DataFrame(np.arange(5 * 4).reshape((5, 4)))

In [143]: df

Out[143]:     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

In [146]: sampler = np.random.permutation(5)

In [147]: sampler

Out[147]: array([3, 2, 0, 1, 4])
```

The array can be used in iloc-based indexing or the equivalent function *take*() function:

```
In [148]: df.iloc[sampler]

Out[148]:     0   1   2   3
          3  12  13  14  15
          2   8   9  10  11
          0   0   1   2   3
          1   4   5   6   7
          4  16  17  18  19

In [149]: df.take(sampler)

Out[149]:     0   1   2   3
          3  12  13  14  15
          2   8   9  10  11
          0   0   1   2   3
          1   4   5   6   7
          4  16  17  18  19
```

To select a random subset without replacement, we can use the *sample*() method on Series and DataFrame:

```
In [152]: #
          df.sample(n=3)

Out[152]:     0   1   2   3
          4  16  17  18  19
          2   8   9  10  11
          3  12  13  14  15

In [156]: df.sample(n=2, axis=1)

Out[156]:     2   0
          0   2   0
          1   6   4
          2  10   8
          3  14  12
          4  18  16
```

To generate a sample with replacement (to allow repeat values), pass the argument *replace*=True to sample:

```
In [157]: choices = pd.Series([5, 7, -1, 6, 4])

In [159]: draws = choices.sample(n=10, replace=True)

In [160]: draws
```

```
Out[160]: 0    5
          3    6
          2   -1
          4    4
          1    7
          3    6
          2   -1
          4    4
          0    5
          3    6
          dtype: int64
```

Computing Indicator/Dummy Variables DataFrame*k*(*k*)*k*01*pandas.get_dummies*()

```
In [161]: df = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'b'],
          .....: 'data1': range(6)})

In [162]: df

Out[162]:   key  data1
          0   b      0
          1   b      1
          2   a      2
          3   c      3
          4   a      4
          5   b      5

In [163]: pd.get_dummies(df['key'])

Out[163]:    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
```

*df key*a, b, c*pandas.get_dummies*()010aa(df)
,*pandas.get_dummies*()*prefix*

```
In [164]: dummies = pd.get_dummies(df['key'], prefix='key_')

In [165]: dummies

Out[165]:    key__a  key__b  key__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
```

df['data1']

```
In [174]: df['data1'] # Series

Out[174]: 0    0
          1    1
          2    2
          3    3
          4    4
          5    5
          Name: data1, dtype: int64

In [175]: df[['data1']] # DataFrame

Out[175]:    data1
          0      0
          1      1
          2      2
          3      3
          4      4
          5      5

In [176]: df[['data1']].join(dummies) # DataFrame

Out[176]:    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 [179]: #
          data.rename(columns={0:'one', 1:'two', 2:'three', 3:'four'}, inplace=True)

In [185]: # 'one''two'DataFrame
          dd = data[['one', 'four']]

In [186]: # 'one'Series
          data['one']

Out[186]: 0     -1.486255
          1     -0.176648
          2     -1.150700
          3     -0.769186
          4     -0.882545
          5      0.043969
          6     -0.941941
          7      0.919140
```

8	-0.591679
9	0.603175
10	-0.005484
11	0.294564
12	-1.446958
13	-0.513096
14	0.037718
15	-0.107953
16	0.613648
17	0.156419
18	-0.725193
19	-0.385183
20	0.641327
21	-1.110199
22	-0.237171
23	0.072884
24	0.460074
25	-0.541733
26	0.057325
27	1.426806
28	-1.200798
29	-0.142353
	...
970	0.445650
971	-0.055439
972	1.682989
973	0.123819
974	2.127047
975	0.710947
976	0.279677
977	-0.419060
978	2.621613
979	0.270127
980	0.776597
981	0.041169
982	-0.337197
983	0.828886
984	-0.569314
985	0.886491
986	1.532338
987	1.209169
988	0.263185
989	0.709655
990	0.224568
991	-0.319106
992	1.229200
993	1.168087
994	2.138031

```
995    0.211693
996    1.655879
997    0.591293
998   -1.533716
999    0.300527
Name: one, Length: 1000, dtype: float64
```

If a row in a DataFrame belongs to multiple categories, things are a bit more complicated.

```
In [189]: # 'one''four'data.columns
          data.columns.get_indexer(['one', 'four'])

Out[189]: array([0, 3], dtype=int64)
```

A useful recipe for staticstical applications is to combine *get_dummies*() with a discretization function like *cut*():

```
In [190]: rng = np.random.RandomState(12345) #

In [191]: values = rng.rand(10)

In [192]: values

Out[192]: array([0.92961609, 0.31637555, 0.18391881, 0.20456028, 0.56772503,
                 0.5955447 , 0.96451452, 0.6531771 , 0.74890664, 0.65356987])

In [193]: #
          #
          bins = [0, 0.2, 0.4, 0.6, 0.8, 1.0]

In [194]: cats = pd.cut(values, bins)

In [195]: cats

Out[195]: [(0.8, 1.0], (0.2, 0.4], (0.0, 0.2], (0.2, 0.4], (0.4, 0.6], (0.4, 0.6], (0.8, 1.0],
          Categories (5, interval[float64]): [(0.0, 0.2] < (0.2, 0.4] < (0.4, 0.6] < (0.6, 0.8]

In [196]: pd.get_dummies(cats)

Out[196]:    (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
```

We create a random instance with seed=12345 to make the result deterministic.

String Manipulation

Python has long been a popular raw data manipulation language in part due to its ease of use for string and text processing. Many text operations are made simple with the string object's built-in methods. For more complex pattern matching and text manipulation, regular expressions may be needed. *pandas adds to the mix by enabling us to apply string and regular expressions concisely on whole arrays of data, additionally handling the annoyance of missin data.

String Object Methods In many string munging and scripting applications, built-in methods are sufficient. For example, a comma-separated string can be broken into pieces with *split*():

```
In [197]: val = 'a,b,    guido'

In [198]: val.split(',')

Out[198]: ['a', 'b', '    guido']
```

split() oftem combined with *strip*() to strim whitespace (including line breaks):

```
In [199]: pieces = [x.strip() for x in val.split(',')]

In [200]: pieces

Out[200]: ['a', 'b', 'guido']
```

These strings could be concatenated together with a two-colons delimiter using addition:

```
In [201]: first, second, third = pieces

In [202]: first + "::" + second + "::" + third

Out[202]: 'a::b::guido'
```

But this isn't a practical general method. A faster and more Pythonic is to pass a tuple or list to the *join*() method on the string "::":

```
In [203]: "::".join(pieces)

Out[203]: 'a::b::guido'
```

Python*inindex*()*find*()

```
In [204]: 'guido' in val

Out[204]: True

In [205]: val.index(',')

Out[205]: 1

In [206]: val.find(":")
```

```
Out[206]: -1
```

index()*find*()*index*()*find*()-1

Relatedly, *count*() method returns the number of occurences of a particular substring:

```
In [207]: val.count(',')

Out[207]: 2
```

replace() method will substitute () the occurences of one pattern for another. It is common to use delete patterns, too, by passing am empty string:

```
In [208]: # val',''::'
          val.replace(',', '::')

Out[208]: 'a::b::    guido'

In [209]: # val''
          val.replace(',', '')

Out[209]: 'ab    guido'
```

Python built-in string methods:

Regular Expressions Regular expressions probide a flexible way to search and match (often more comlex) string pattern in text. Python's built-in module *re* is responsible for applying regular expression to strings.

The *re* module falls into three categories: pattern matching, substitution, and splitting. Naturally these are all related; a regex (short for regular expression) describes a pattern to locate in the text, which can be used for many purposes. Let's look at an example:

Suppose we want to split a string with a varaible number of whitespaces (tabs, spaces and newlines). The regex describing one or more whitespace characters is +:

```
In [210]: import re

In [211]: text = 'foo      bar\t  baz \tqux'

In [212]: re.split('\s+', text)

Out[212]: ['foo', 'bar', 'baz', 'qux']
```

re.split('+', text)*split*()text*re.compile*()regex:

```
In [213]: regex = re.compile('\s+')

In [214]: regex.split(text)
```

```
Out[214]: ['foo', 'bar', 'baz', 'qux']
```

re.findall()

```
In [215]: regex.findall(text)
```

```
Out[215]: ['      ', '\t  ', ' \t']
```

Creating a regex object with *re.compile*() method is highly recommended if want to apply the same expression to many strings; doing so will save many CPU cycles.

match()*search*()*findall*()*findall*()*search*()*match*()

Let's consider a block of text and a regular expression capable of identifying most email addresses:

```
In [216]: text = """Dave dave@google.com
          Steve steve@gmail.com
          Rob rob@gmail.com
          Ryan ryan@yahoo.com
          """
```

```
In [223]: pattern = r'[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}'
```

```
In [224]: #
          # re.IGNORECASE makes the regex case-insensitive
          regex = re.compile(pattern, flags=re.IGNORECASE)
```

findall():

```
In [225]: regex.findall(text)
```

```
Out[225]: ['dave@google.com', 'steve@gmail.com', 'rob@gmail.com', 'ryan@yahoo.com']
```

searc()match:

```
In [226]: m = regex.search(text)
```

```
In [227]: m
```

```
Out[227]: <re.Match object; span=(5, 20), match='dave@google.com'>
```

```
In [228]: text[m.start():m.end()]
```

```
Out[228]: 'dave@google.com'
```

```
In [230]: print(regex.match(text))
```

```
None
```

regex.match() returns None, as it only matches if the pattern occurs at the begining of the string.

Relatedly, *sub*() method will return a new string with occurences of the string replaced by the new string:

```
In [231]: print(regex.sub("redacted", text))

Dave redacted
Steve redacted
Rob redacted
Ryan redacted
```

(parenthesis)

```
In [233]: pattern = r'([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\.([A-Z]{2,4})'

In [237]: regex = re.compile(pattern, flags=re.IGNORECASE)
```

A match object produced by this modified regex returns a tuple of the pattern components with its *group*() method:

```
In [238]: m = regex.match('tongzi@126.com')

In [239]: m.groups()

Out[239]: ('tongzi', '126', 'com')
```

While *findall*() returns a list of tuples when the pattern has groups:

```
In [240]: regex.findall(text)

Out[240]: [('dave', 'google', 'com'),
           ('steve', 'gmail', 'com'),
           ('rob', 'gmail', 'com'),
           ('ryan', 'yahoo', 'com')]
```

sub() also has access to groups in each match using special synbols like \1 and \2. The symbol \1 corresponds to the first matched group, \2 corresponds to the second, and so forth:

```
In [241]: print(regex.sub(r'User: \1, Domain: \2, Suffix: \3', text))

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

Table below provides a brief summay:

Vectorized String Functions in pandas Cleaning up a messy dataset for analysis often requires a lot of string munging and regularization. To complicate matters, a column containing strings will sometimes have missing data:

```
In [242]: data = {'Dave': 'dave@google.com', 'Steve': 'steve@gmail.com',
     .....: 'Rob': 'rob@gmail.com', 'Wes': np.nan}

In [243]: data = pd.Series(data)

In [244]: data

Out[244]: Dave     dave@google.com
          Steve    steve@gmail.com
          Rob        rob@gmail.com
          Wes                  NaN
          dtype: object

In [245]: data.isnull()

Out[245]: Dave     False
          Steve    False
          Rob      False
          Wes       True
          dtype: bool
```

We can apply string and regular expression methods that can be applied (passing a lambda or other function) to each value using *data.map*(), but it will fail on NA values. To code with this, Series has array-oriented methods for string operations that skip NA values. These methods are accessed through Series's *str* attribute. For example, we can check whether each email address has 'gmail' in it with *str.contains*():

```
In [246]: data.str.contains('gmail')

Out[246]: Dave     False
          Steve     True
          Rob       True
          Wes        NaN
          dtype: object
```

Regular expressions can be used, too, along with any *re* options like *re.IGNORECASE*:

```
In [248]: pattern

Out[248]: '([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\\.([A-Z]{2,4})'

In [249]: data.str.findall(pattern, flags=re.IGNORECASE)

Out[249]: Dave     [(dave, google, com)]
          Steve    [(steve, gmail, com)]
          Rob        [(rob, gmail, com)]
          Wes                        NaN
          dtype: object
```

There are a couple of ways to do vectorized element retrieval. Either use *str.get()* or index into the *str* attribute:

```
In [250]: matches = data.str.match(pattern, flags=re.IGNORECASE)

In [260]: matches

Out[260]: Dave     True
          Steve    True
          Rob      True
          Wes       NaN
          dtype: object

In [255]: data['ceprei'] = 'cepreitest  software@ceprei.biz'

In [256]: data

Out[256]: Dave                          dave@google.com
          Steve                         steve@gmail.com
          Rob                             rob@gmail.com
          Wes                                       NaN
          ceprei    cepreitest  software@ceprei.biz
          dtype: object

In [257]: data.str.match(pattern, flags=re.IGNORECASE)

Out[257]: Dave       True
          Steve      True
          Rob        True
          Wes         NaN
          ceprei    False
          dtype: object
```

cepreicepreitest*data.str.match*()False

To access elements in the embedded lists, we can pass an index to either of these functions:

```
In [258]: # matchesSeries
          matches.str.get(1)

Out[258]: Dave    NaN
          Steve   NaN
          Rob     NaN
          Wes     NaN
          dtype: float64

In [261]: matches.str[0]

Out[261]: Dave    NaN
          Steve   NaN
          Rob     NaN
          Wes     NaN
          dtype: float64
```

```
In [262]: data.str[:5]

Out[262]: Dave      dave@
          Steve     steve
          Rob       rob@g
          Wes         NaN
          ceprei    cepre
          dtype: object
```

Partial listing of vectorized string methods: