CHAPTER 5

Data Cleaning and Preparation

During the course of doing data analysis and modeling, a significant amount of time is spent on data preparation: loading, cleaning, transforming, and rearranging. Such tasks are often reported to take up 80% or more of an analyst's time. Sometimes the way that data is stored in files or databases is not in the right format for a particular task. Many researchers choose to do ad hoc processing of data from one form to another using a general-purpose programming language, like Python, Perl, R, or Java, or Unix text-processing tools like sed or awk. Fortunately, pandas, along with the built-in Python language features, provides you with a high-level, flexible, and fast set of tools to enable you to manipulate data into the right form.

5.1 Handling Missing Data

Missing data occurs commonly in many data analysis applications. One of the goals of pandas is to make working with missing data as painless as possible. For example, all of the descriptive statistics on pandas objects exclude missing data by default.

The way that missing data is represented in pandas objects is somewhat imperfect, but it is functional for a lot of users. For numeric data, pandas uses the floating-point value **NaN** (Not a Number) to represent missing data. We call this a *sentinel value* that can be easily detected:

In [1]:
```
import pandas as pd
import numpy as np
```

In [2]:
```
string_data = pd.Series(['aardvark', 'artichoke', np.nan, 'avocado'])
string_data
```

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

In [3]:
```
string_data.isnull()
```

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

In pandas, we've adopted a convention used in the R programming language by referring to missing data as **NA**, which stands for *not available*. In statistics applications, NA data may either be data that does not exist or that exists but was not observed (through problems with data collection, for example). When cleaning up data for analysis, it is often important to do analysis on the missing data itself to identify data collection problems or potential biases in the data caused by missing data.

The built-in Python None value is also treated as NA in object arrays:

In [4]:
```
string_data[0] = None
string_data.isnull()
```

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

There is work ongoing in the pandas project to improve the internal details of how missing data is handled, but the user API functions, like pandas.isnull, abstract away many of the annoying details. See Table 5-1 for a list of some functions related to missing data handling.

Table 5.1: NA handling methods

Argument	Description
`dropna`	Filter axis labels based on whether values for each label have missing data, with varying thresholds for how much missing data to tolerate.
`fillna`	Fill in missing data with some value or using an interpolation method such as `'ffill'` or `'bfill'`.
`isnull`	Return boolean values indicating which values are missing/NA.
`notnull`	Negation of `isnull`.

5.1.1 Filtering Out Missing Data

There are a few ways to filter out missing data. While you always have the option to do it by hand using pandas.isnull and boolean indexing, the **dropna** can be helpful. On a Series, it returns the Series with only the non-null data and index values:

In [5]:
```
from numpy import nan as NA

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

Out[5]:
```
0    1.0
1    NaN
2    3.5
3    NaN
4    7.0
dtype: float64
```

In [6]:
```
data.dropna()
```

Out[6]:
```
0    1.0
2    3.5
4    7.0
dtype: float64
```

This is equivalent to:

In [7]:
```
data[data.notnull()]
```

Out[7]:
```
0    1.0
2    3.5
4    7.0
dtype: float64
```

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

In [8]:
```
data = pd.DataFrame([[1., 6.5, 3.], [1., NA, NA],[NA, NA, NA], [NA, 6.5, 3.]])
data
```

Out[8]:

	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 [9]:
```
cleaned = data.dropna()
cleaned
```

Out[9]:

	0	1	2
0	1.0	6.5	3.0

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

In [10]:
```
data.dropna(how='all')
```

Out[10]:

	0	1	2
0	1.0	6.5	3.0
1	1.0	NaN	NaN
3	NaN	6.5	3.0

To drop columns in the same way, pass **axis=1**:

In [11]:
```
data[4] = NA
data
```

Out[11]:

	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 [12]:
```
data.dropna(axis=1, how='all')
```

Out[12]:

	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

A related way to filter out DataFrame rows tends to concern time series data. Suppose you want to keep only rows containing a certain number of observations. You can indicate this with the **thresh** argument:

In [13]:
```
df = pd.DataFrame(np.random.randn(7, 3))
df.iloc[:4, 1] = NA
df.iloc[:2, 2] = NA
df
```

Out[13]:

	0	1	2
0	0.278166	NaN	NaN
1	0.277251	NaN	NaN
2	-0.646550	NaN	-0.063453
3	0.172160	NaN	0.880198
4	-0.458739	0.495164	-0.152519
5	-0.967685	-0.484373	-0.643886
6	0.065862	0.342122	0.726409

In [14]:
```
df.dropna()
```

Out[14]:

	0	1	2
4	-0.458739	0.495164	-0.152519
5	-0.967685	-0.484373	-0.643886
6	0.065862	0.342122	0.726409

In [15]:
```
df.dropna(thresh=2)
# it will drop any rows that contain less than 2 observation values
```

Out[15]:

	0	1	2
2	-0.646550	NaN	-0.063453
3	0.172160	NaN	0.880198
4	-0.458739	0.495164	-0.152519
5	-0.967685	-0.484373	-0.643886
6	0.065862	0.342122	0.726409

5.1.2 Filling In Missing Data

Rather than filtering out missing data (and potentially discarding other data along with it), you may want to fill in the “holes” in any number of ways. For most purposes, the **fillna** method is the workhorse function to use. Calling **fillna** with a constant replaces missing values with that value:

In [16]:
```
df.fillna(0)
```

Out[16]:

	0	1	2
0	0.278166	0.000000	0.000000
1	0.277251	0.000000	0.000000
2	-0.646550	0.000000	-0.063453
3	0.172160	0.000000	0.880198
4	-0.458739	0.495164	-0.152519
5	-0.967685	-0.484373	-0.643886
6	0.065862	0.342122	0.726409

Calling **fillna** with a dict, you can use a different fill value for each column:

In [17]:
```
df
```

Out[17]:

	0	1	2
0	0.278166	NaN	NaN
1	0.277251	NaN	NaN
2	-0.646550	NaN	-0.063453
3	0.172160	NaN	0.880198
4	-0.458739	0.495164	-0.152519
5	-0.967685	-0.484373	-0.643886
6	0.065862	0.342122	0.726409

In [18]:
```
df.fillna({1: 0.5, 2: 0})
```

Out[18]:

	0	1	2
0	0.278166	0.500000	0.000000
1	0.277251	0.500000	0.000000
2	-0.646550	0.500000	-0.063453
3	0.172160	0.500000	0.880198
4	-0.458739	0.495164	-0.152519
5	-0.967685	-0.484373	-0.643886
6	0.065862	0.342122	0.726409

fillna returns a new object, but you can modify the existing object *in-place*:

In [19]:
```
df.fillna(0, inplace=True)
df
```

Out[19]:

	0	1	2
0	0.278166	0.000000	0.000000
1	0.277251	0.000000	0.000000
2	-0.646550	0.000000	-0.063453
3	0.172160	0.000000	0.880198
4	-0.458739	0.495164	-0.152519
5	-0.967685	-0.484373	-0.643886
6	0.065862	0.342122	0.726409

The same interpolation methods available for reindexing can be used with **fillna**:

In [20]:
```
df = pd.DataFrame(np.random.randn(6, 3))
df.iloc[2:, 1] = NA
df.iloc[4:, 2] = NA
df
```

Out[20]:

	0	1	2
0	-1.044206	0.604230	0.447860
1	0.904291	0.188787	-0.774222
2	0.329654	NaN	-1.259086
3	-1.784109	NaN	0.362695
4	-0.017220	NaN	NaN
5	-1.129725	NaN	NaN

In [21]:
```
df.fillna(method='ffill')
```

Out[21]:

	0	1	2
0	-1.044206	0.604230	0.447860
1	0.904291	0.188787	-0.774222
2	0.329654	0.188787	-1.259086
3	-1.784109	0.188787	0.362695
4	-0.017220	0.188787	0.362695
5	-1.129725	0.188787	0.362695

In [22]:
```
df.fillna(method='ffill', limit=2)
```

Out[22]:

	0	1	2
0	-1.044206	0.604230	0.447860
1	0.904291	0.188787	-0.774222
2	0.329654	0.188787	-1.259086
3	-1.784109	0.188787	0.362695
4	-0.017220	NaN	0.362695
5	-1.129725	NaN	0.362695

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

In [23]:
```
data = pd.Series([1., NA, 3.5, NA, 7])
data
```

Out[23]:
```
0    1.0
1    NaN
2    3.5
3    NaN
4    7.0
dtype: float64
```

In [24]:
```
data.fillna(data.mean())
```

Out[24]:
```
0    1.000000
1    3.833333
2    3.500000
3    3.833333
4    7.000000
dtype: float64
```

See Table 5-2 for a reference on **fillna**.

Table 5.2: fillna function arguments

Argument	Description
`value`	Scalar value or dict-like object to use to fill missing values
`method`	Interpolation; by default `'ffill'` if function called with no other arguments
`axis`	Axis to fill on; default `axis=0`
`inplace`	Modify the calling object without producing a copy
`limit`	For forward and backward filling, maximum number of consecutive periods to fill

In [25]:
```
https://chrisalbon.com/python/data_wrangling/pandas_missing_data/
```

```
  File "<ipython-input-25-c8c14180c1e1>", line 1
    https://chrisalbon.com/python/data_wrangling/pandas_missing_data/
           ^
SyntaxError: invalid syntax
```

5.2 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.

5.2.1 Removing Duplicates

Duplicate rows may be found in a DataFrame for any number of reasons. Here is an example:

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

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

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

Relatedly, **drop_duplicates** returns a DataFrame where the duplicated array is False:

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

Both of these methods by default consider all of the columns; alternatively, you can specify any subset of them to detect duplicates. Suppose we had an additional column of values and wanted to filter duplicates only based on the 'k1' column:

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

In [ ]:
```
data.drop_duplicates(['k1'])
```

duplicated and **drop_duplicates** by default keep the first observed value combination. Passing *keep='last'* will return the last one:

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

5.2.2 Transforming Data Using a Function or Mapping

For many datasets, you may wish to perform some transformation based on the values in an array, Series, or column in a DataFrame. Consider the following hypothetical data collected about various kinds of vegetables:

In [ ]:
```
data = pd.DataFrame({'vege': ['cabbage', 'carrot', 'cabbage','Lettuce', 'potato', 'Cabbage',
                              'lettuce', 'tomato', 'cucumber'],  'weight': [4, 3, 12, 6, 7.5, 8, 3,
5, 6]})
data
```

Suppose you wanted to add a column indicating the seller that each food came from. Let's write down a mapping of each distinct vegetables type to the seller:

In [ ]:
```
vege_to_seller = {'cabbage': 'David',  'carrot': 'David',  'lettuce': 'Ahmad',  'potato': 'Ahmad',
                  'tomato': 'David',  'cucumber': 'Tina'}
```

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

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

In [ ]:
```
data['seller'] = lowercased.map(vege_to_seller)
data
```

We could also have passed a function that does all the work:

In [ ]:
```
data['vege'].map(lambda x: vege_to_seller[x.lower()])
```

Using **map** is a convenient way to perform element-wise transformations and other data cleaning–related operations.

5.2.3 Replacing Values

Filling in missing data with the **fillna** method is a special case of more general value replacement. As you've already seen, **map** can be used to modify a subset of values in an object but **replace** provides a simpler and more flexible way to do so. Let's consider this Series:

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

The -999 values might be sentinel values for missing data. To replace these with NA values that pandas understands, we can use replace, producing a new Series (unless you pass inplace=True):

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

If you want to replace multiple values at once, you instead pass a list and then the substitute value:

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

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

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

The argument passed can also be a dict:

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

The **data.replace** method is distinct from **data.str.replace**, which performs string substitution element-wise. We look at these string methods on Series later in the chapter.

5.2.4 Renaming Axis Indexes

Like values in a Series, axis labels can be similarly transformed by a function or mapping of some form to produce new, differently labeled objects. You can also modify the axes in-place without creating a new data structure. Here's a simple example:

In [27]:
```
data = pd.DataFrame(np.arange(12).reshape((3, 4)), index=['Ohio', 'Colorado', 'New York'],
                    columns=['one', 'two', 'three', 'four'])
data
```

Out[27]:

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

Like a Series, the axis indexes have a **map** method:

In [30]:
```
transform = lambda x: x[:4].upper()
data.index.map(transform)
```

Out[30]:
```
Index(['OHIO', 'COLO', 'NEW '], dtype='object')
```

You can assign to **index**, modifying the DataFrame in-place:

In [32]:
```
data.index = data.index.map(transform)
data
```

Out[32]:

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

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

In [33]:
```
data.rename(index=str.title, columns=str.upper)
```

Out[33]:

	ONE	TWO	THREE	FOUR
Ohio	0	1	2	3
Colo	4	5	6	7
New	8	9	10	11

Notably, **rename** can be used in conjunction with a dict-like object providing new values for a subset of the axis labels:

In [34]:
```
 data.rename(index={'OHIO': 'INDIANA'},columns={'three': 'peekaboo'})
```

Out[34]:

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

rename saves you from the chore of copying the DataFrame manually and assigning to its **index** and **columns** attributes. Should you wish to modify a dataset in-place, pass *inplace=True*:

In [35]:
```
data.rename(index={'OHIO': 'INDIANA'}, inplace=True)
data
```

Out[35]:

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

5.2.5 Discretization and Binning

Continuous data is often discretized or otherwise separated into “bins” for analysis. Suppose you have data about a group of people in a study, and you want to group them into discrete age buckets:

In [36]:
```
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, you have to use **cut**, a function in pandas:

In [37]:
```
bins = [18, 25, 35, 60, 100]
cats = pd.cut(ages, bins)
cats
```

Out[37]:
```
[(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]): [(18, 25] < (25, 35] < (35, 60] < (60, 100]]
```

The object pandas returns is a special **Categorical** object. The output you see describes the bins computed by **pandas.cut**. You can treat it like an array of strings indicating the bin name; internally it contains a **categories** array specifying the distinct category names along with a labeling for the **ages** data in the codes attribute:

In [38]:
```
cats.codes
```

Out[38]:
```
array([0, 0, 0, 1, 0, 0, 2, 1, 3, 2, 2, 1], dtype=int8)
```

In [39]:
```
cats.categories
```

Out[39]:
```
IntervalIndex([(18, 25], (25, 35], (35, 60], (60, 100]]
              closed='right',
              dtype='interval[int64]')
```

In [40]:
```
pd.value_counts(cats)
```

Out[40]:
```
(18, 25]     5
(35, 60]     3
(25, 35]     3
(60, 100]    1
dtype: int64
```

Note that **pd.value_counts(cats)** are the bin counts for the result of **pandas.cut**.

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

In [41]:
```
 pd.cut(ages, [18, 26, 36, 61, 100], right=False)
```

Out[41]:
```
[[18, 26), [18, 26), [18, 26), [26, 36), [18, 26), ..., [26, 36), [61, 100), [36, 61), [36, 61),
[26, 36)]
Length: 12
Categories (4, interval[int64]): [[18, 26) < [26, 36) < [36, 61) < [61, 100)]
```

You can also pass your own bin names by passing a list or array to the **labels** option:

In [42]:
```
group_names = ['Youth', 'YoungAdult', 'MiddleAged', 'Senior']
pd.cut(ages, bins, labels=group_names)
```

Out[42]:
```
[Youth, Youth, Youth, YoungAdult, Youth, ..., YoungAdult, Senior, MiddleAged, MiddleAged, YoungAd
ult]
Length: 12
Categories (4, object): [Youth < YoungAdult < MiddleAged < Senior]
```

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

In [43]:
```
data = np.random.rand(20)
data
```

Out[43]:
```
array([0.88871342, 0.7306573 , 0.97697839, 0.24678705, 0.90640686,
       0.41670186, 0.76636268, 0.12230495, 0.00641049, 0.78483762,
       0.68942646, 0.67022781, 0.7899601 , 0.6836309 , 0.73432307,
       0.9790324 , 0.37388424, 0.34704298, 0.27370018, 0.44901268])
```

In [44]:
```
pd.cut(data, 4, precision=2)
```

Out[44]:
```
[(0.74, 0.98], (0.49, 0.74], (0.74, 0.98], (0.0054, 0.25], (0.74, 0.98], ..., (0.74, 0.98], (0.25
, 0.49], (0.25, 0.49], (0.25, 0.49], (0.25, 0.49]]
Length: 20
Categories (4, interval[float64]): [(0.0054, 0.25] < (0.25, 0.49] < (0.49, 0.74] < (0.74, 0.98]]
```

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

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

In [45]:
```
data = np.random.randn(1000)  # Normally distributed
data
```

```
Out[45]: array([ 1.81389040e+00,  1.83171166e+00, -1.46090093e+00, -1.46494872e+00,
                 1.22069655e+00,  1.27331085e+00, -1.57138524e+00, -9.12192374e-01,
                -7.93686660e-01,  1.77952016e-02, -3.07787711e-01,  2.50148415e-01,
                 3.62276737e-01, -2.44382802e+00, -5.84712699e-01,  2.77142894e-01,
                -8.13877473e-01, -1.31209589e+00,  6.38475677e-01, -2.40648092e-01,
                 5.18671433e-01, -7.90198280e-01, -2.04114683e-01,  1.36790083e+00,
                 9.57333122e-01,  8.14495198e-01,  6.47352960e-01, -1.25555211e+00,
                 7.87696121e-01,  2.23638577e-01,  1.37274989e+00, -5.51518052e-01,
                -2.35942447e+00, -1.02150221e+00, -4.38211546e-02,  1.10764963e+00,
                -1.40081121e+00,  1.62619488e+00, -9.34641621e-01, -1.97916044e-01,
                -7.02721110e-01,  1.35653875e+00, -9.71408665e-01, -7.21039208e-01,
                -2.16739468e+00,  4.81818157e-01,  1.99808752e+00,  1.19684390e+00,
                 6.54475672e-01,  1.06811644e-01, -2.98626982e-02,  1.03666852e+00,
                 3.25844833e-01,  1.39008305e+00, -1.67270459e+00, -6.07703285e-01,
                 9.43238433e-01,  1.09082298e-01,  4.33352557e-01, -1.35641061e+00,
                -3.75232332e-02,  5.17692596e-01,  7.86531045e-01,  1.05331960e+00,
                 8.89927808e-01,  8.12729108e-01, -2.85824425e-01,  1.24775243e+00,
                -1.73033930e+00,  1.06862198e+00, -9.44814383e-01,  1.81145792e+00,
                 1.31536488e+00,  8.46477684e-01, -2.94491136e-01, -1.00268594e+00,
                -4.87326467e-01,  6.30519928e-01, -9.12120361e-01, -3.68599065e-01,
                 6.64129443e-01, -9.24724265e-01,  4.41305925e-01,  4.89558405e-01,
                -3.40815515e-02,  4.45665970e-01,  3.67300581e-01,  1.01305151e+00,
                 9.23527771e-01,  5.11332437e-01, -1.44623346e+00,  5.84509047e-01,
                -8.29966702e-01, -7.40514214e-01,  2.58005883e-02,  1.48827225e+00,
                -8.68884372e-01,  1.00706874e+00, -4.00501916e-01,  6.57544225e-01,
                -4.46051797e-01,  4.70105735e-01,  5.51283695e-01, -2.37651000e-01,
                -2.23869929e+00, -1.51356905e+00,  2.34997734e-01, -4.54217520e-01,
                 1.78796764e-01,  1.59695035e-01, -4.89619241e-01,  6.62978076e-01,
                 4.69413382e-01,  8.16151154e-01,  1.19669902e+00, -1.26321790e-01,
                 1.69708356e+00,  5.86087047e-01, -1.29432200e-01,  5.62038237e-01,
                -2.04676462e+00, -9.22239883e-01,  1.63578489e+00,  1.45305459e+00,
                -1.01439555e+00, -1.70304861e+00,  6.01732121e-02, -5.99792122e-01,
                -7.77681474e-01, -7.07006677e-01,  8.13543747e-01,  5.17537117e-02,
                -4.68517732e-01, -7.97818672e-01,  2.42786851e-01, -1.08405016e+00,
                 4.91712983e-01,  1.48533327e+00, -1.15692159e+00, -4.86670061e-02,
                 4.13955653e-01, -3.95364438e-01, -2.72597445e-02,  1.11476191e-01,
                 1.48649476e+00,  3.57238690e-01, -3.17144076e-01, -7.05166157e-01,
                 4.54471825e-01, -3.62210630e-01, -4.63170891e-01,  6.44659971e-01,
                -2.94079142e-01,  1.79462175e+00,  3.64220426e-01,  1.49279691e+00,
                 4.31646671e-02,  7.72075666e-01,  3.29134066e-01,  1.50270338e+00,
                -1.18857955e+00, -2.77345949e-02, -1.26355888e+00,  1.53280918e+00,
                -4.97123916e-01,  1.91478975e+00,  3.84781445e-01, -1.17991545e+00,
                 1.04342669e+00, -1.07839663e+00,  6.93254713e-01, -1.27704320e+00,
                -4.91161874e-01, -3.68585341e-01, -2.08794912e-01, -4.59257706e-02,
                -8.14906715e-01, -1.26866075e+00,  1.69940216e+00,  9.50685635e-01,
                 1.00024867e+00,  2.58658270e-02, -1.03306004e+00, -3.92513576e-01,
                 1.72153745e+00,  1.59192306e+00,  1.18686162e+00, -6.14233042e-01,
                -9.39216639e-01, -8.27531086e-02, -8.87301697e-02,  5.69511763e-01,
                -9.67286851e-01, -7.91045084e-01, -5.02414014e-01, -7.32990836e-01,
                -6.58401528e-01,  5.43582515e-01,  5.59984554e-01,  1.82771416e+00,
                 3.21487956e+00,  5.72529038e-01, -1.02625624e+00,  7.39350265e-01,
                -4.10646250e-01,  2.75654989e-01, -8.91762265e-01,  9.47794851e-01,
                 1.19747915e+00,  3.96373692e-01,  9.55042908e-01, -3.12371145e-01,
                -1.54449844e+00, -3.40409066e-02,  3.62121896e-01,  1.40513602e-01,
                -1.22122618e+00,  2.91951833e-01,  5.05061376e-01,  8.17133568e-01,
                -1.73916480e+00,  1.95028192e+00, -4.51909370e-01, -7.26391668e-01,
                -1.09713477e+00, -1.05101963e-01,  7.69328037e-01,  3.10350574e-01,
                -8.08137675e-01, -8.24597450e-01, -6.92151234e-01,  1.23633951e-01,
                 8.27627403e-01, -1.04587771e+00,  4.29520359e-01, -1.32413316e+00,
                -1.10166680e+00,  7.74711183e-01, -7.42864430e-01, -2.15814893e-01,
                 4.33526344e-01,  6.32780160e-01,  1.91144097e-01,  2.11429928e-01,
                 2.74408767e-01, -1.74920430e+00, -1.01682421e+00, -6.48791517e-01,
                 9.69021748e-01, -4.15615849e-01, -1.64125765e+00, -1.52718970e-01,
                 9.29447572e-01,  1.65837098e-01,  8.15468158e-01,  5.43143606e-02,
                -3.08331696e-01,  1.37772666e+00,  2.97713090e-02, -1.29202339e-01,
                 3.96601948e-01, -7.70140429e-01,  2.83590322e-01,  2.55857175e-01,
                 1.39247492e+00,  9.99593704e-02, -1.85107538e+00,  9.01515009e-01,
                 1.48466862e+00,  2.50811769e+00,  8.34039328e-01,  7.43794565e-02,
                 5.66477259e-01,  6.29921631e-01, -6.73106327e-01, -3.29161958e-01,
                -4.45037321e-01, -7.48391818e-01,  1.67408970e+00, -1.57103715e+00,
                 1.45742274e+00,  1.21259324e-01,  5.39242732e-01,  3.74451182e-02,
                -1.02155238e+00, -3.70072149e-02, -5.71582028e-01,  6.38792527e-01,
                -7.95235641e-01,  1.44479705e+00,  3.24643773e-01, -6.54357418e-01,
                -4.64209711e-01,  5.28976811e-01, -2.48607625e-01, -2.13800009e-01,
                 1.36601240e+00, -1.37504793e+00,  1.25936467e+00, -6.35335486e-01,
                 1.04144345e+00,  7.05919931e-01,  1.73231158e-02,  1.09775774e+00,
                -4.16599127e-01, -3.38234569e-01,  8.95300746e-01, -1.67605857e+00,
                 2.32886105e-01, -3.48137224e-01,  1.12818884e+00, -1.04282536e+00,
                 3.09246239e-01,  2.82927507e+00,  8.35563454e-01,  1.15887502e+00,
                 2.87563157e-01,  9.76508565e-01,  9.77009652e-01, -6.26300116e-01,
                -6.38367646e-01, -1.54231916e+00,  3.89859051e-02,  1.02413672e+00,
                -3.01884516e-01,  1.42221689e+00, -1.14499889e+00, -2.53354780e-01,
```

In [46]:
```
cats = pd.qcut(data, 4)  # Cut into quartiles
cats
```

Out[46]:
```
[(0.665, 3.215], (0.665, 3.215], (-3.424, -0.69], (-3.424, -0.69], (0.665, 3.215], ..., (-0.69, 0
.0404], (0.0404, 0.665], (0.0404, 0.665], (-0.69, 0.0404], (0.0404, 0.665]]
Length: 1000
Categories (4, interval[float64]): [(-3.424, -0.69] < (-0.69, 0.0404] < (0.0404, 0.665] < (0.665,
3.215]]
```

In [47]:
```
pd.value_counts(cats)
```

Out[47]:
```
(0.665, 3.215]     250
(0.0404, 0.665]    250
(-0.69, 0.0404]    250
(-3.424, -0.69]    250
dtype: int64
```

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

In [48]:
```
pd.qcut(data, [0, 0.1, 0.5, 0.9, 1.])
```

Out[48]:
```
[(1.243, 3.215], (1.243, 3.215], (-3.424, -1.235], (-3.424, -1.235], (0.0404, 1.243], ..., (-1.23
5, 0.0404], (0.0404, 1.243], (0.0404, 1.243], (-1.235, 0.0404], (0.0404, 1.243]]
Length: 1000
Categories (4, interval[float64]): [(-3.424, -1.235] < (-1.235, 0.0404] < (0.0404, 1.243] < (1.24
3, 3.215]]
```

5.3 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. Most text operations are made simple with the string object's built-in methods. For more complex pattern matching and text manipulations, regular expressions may be needed. pandas adds to the mix by enabling you to apply string and regular expressions concisely on whole arrays of data, additionally handling the annoyance of missing data.

5.3.1 String Object Methods

In many string munging and scripting applications, built-in string methods are sufficient. As an example, a comma-separated string can be broken into pieces with **split**:

In [50]:
```
val = 'a,b,  guido'
val.split(',')
```

Out[50]:
```
['a', 'b', '  guido']
```

split is often combined with **strip** to trim whitespace (including line breaks):

In [51]:
```
pieces = [x.strip() for x in val.split(',')]
pieces
```

Out[51]:
```
['a', 'b', 'guido']
```

These substrings could be concatenated together with a two-colon delimiter using addition:

In [52]:
```
first, second, third = pieces
first + '::' + second + '::' + third
```

Out[52]:
```
'a::b::guido'
```

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

In [53]:
```
'::'.join(pieces)
```

Out[53]:
```
'a::b::guido'
```

Other methods are concerned with locating substrings. Using Python's **in** keyword is the best way to detect a substring, though **index** and *find* can also be used:

In [54]:
```
'guido' in val
```

Out[54]:
```
True
```

In [55]:
```
val.index(',')
```

Out[55]:
```
1
```

In [56]:
```
val.find(':')
```

Out[56]:
```
-1
```

Note the difference between **find** and **index** is that index raises an exception if the string isn't found (versus returning –1):

In [57]:
```
val.index(':')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-57-2e127bf43f77> in <module>()
----> 1 val.index(':')

ValueError: substring not found
```

Relatedly, **count** returns the number of occurrences of a particular substring:

In [58]:
```
val.count(',')
```

Out[58]:
```
2
```

replace will substitute occurrences of one pattern for another. It is commonly used to delete patterns, too, by passing an empty string:

In [59]:
```
val.replace(',', '::')
```

Out[59]:
```
'a::b::  guido'
```

In [60]:
```
val.replace(',', '')
```

Out[60]:
```
'ab  guido'
```

See Table 5-3 for a listing of some of Python's string methods. Regular expressions can also be used with many of these operations, as you'll see.

Table 5.3: Python built-in string methods

Argument	Description
`count`	Return the number of non-overlapping occurrences of substring in the string.
`endswith`	Returns `True` if string ends with suffix.
`startswith`	Returns `True` if string starts with prefix.
`join`	Use string as delimiter for concatenating a sequence of other strings.
`index`	Return position of first character in substring if found in the string; raises `ValueError` if not found.
`find`	Return position of first character of *first* occurrence of substring in the string; like `index`, but returns –1 if not found.
`rfind`	Return position of first character of *last* occurrence of substring in the string; returns –1 if not found.
`replace`	Replace occurrences of string with another string.
`strip`, `rstrip`, `lstrip`	Trim whitespace, including newlines; equivalent to `x.strip()` (and `rstrip`, `lstrip`, respectively) for each element.
`split`	Break string into list of substrings using passed delimiter.
`lower`	Convert alphabet characters to lowercase.
`upper`	Convert alphabet characters to uppercase.
`casefold`	Convert characters to lowercase, and convert any region-specific variable character combinations to a common comparable form.
`ljust`, `rjust`	Left justify or right justify, respectively; pad opposite side of string with spaces (or some other fill character) to return a string with a minimum width.

5.3.2 Regular Expressions

Regular expressions provide a flexible way to search or match (often more complex) string patterns in text. A single expression, commonly called a *regex*, is a string formed according to the regular expression language. Python's built-in **re** module is responsible for applying regular expressions to strings.

The **re** module functions fall into three categories: pattern matching, substitution, and splitting. Naturally these are all related; a *regex* describes a pattern to locate in the text, which can then be used for many purposes. Let's look at a simple example:

Suppose we wanted to split a string with a variable number of whitespace characters (tabs, spaces, and newlines). The *regex* describing one or more whitespace characters is **\s+**:

```
In [61]: import re
         text = "foo    bar\t baz  \tqux"
         re.split('\s+', text)
Out[61]: ['foo', 'bar', 'baz', 'qux']
```

When you call **re.split('\s+', text)**, the regular expression is first compiled, and then its split method is called on the passed text. You can compile the *regex* yourself with **re.compile**, forming a reusable *regex* object:

```
In [62]: regex = re.compile('\s+')
         regex.split(text)
Out[62]: ['foo', 'bar', 'baz', 'qux']
```

If, instead, you wanted to get a list of all patterns matching the *regex*, you can use the **findall** method:

```
In [63]: regex.findall(text)
Out[63]: ['    ', '\t ', '  \t']
```

match and **search** are closely related to **findall**. While **findall** returns all matches in a string, **search** returns only the first match. More rigidly, **match** only matches at the beginning of the string. As a less trivial example, let's consider a block of text and a regular expression capable of identifying most email addresses:

In [65]:
```
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 makes the regex case-insensitive
regex = re.compile(pattern, flags=re.IGNORECASE)
```

Using **findall** on the text produces a list of the email addresses:

In [66]:
```
regex.findall(text)
```

Out[66]:
```
['dave@google.com', 'steve@gmail.com', 'rob@gmail.com', 'ryan@yahoo.com']
```

search returns a special match object for the first email address in the text. For the preceding *regex*, the match object can only tell us the start and end position of the pattern in the string:

In [67]:
```
m = regex.search(text)
m
```

Out[67]:
```
<_sre.SRE_Match object; span=(5, 20), match='dave@google.com'>
```

In [68]:
```
text[m.start():m.end()]
```

Out[68]:
```
'dave@google.com'
```

regex.match returns None, as it only will match if the pattern occurs at the start of the string:

In [69]:
```
print(regex.match(text))
```

```
None
```

Relatedly, **sub** will return a new string with occurrences of the pattern replaced by the a new string:

In [70]:
```
print(regex.sub('REDACTED', text))
```

```
Dave REDACTED
Steve REDACTED
Rob REDACTED
Ryan REDACTED
```

Suppose you wanted to find email addresses and simultaneously segment each address into its three components: username, domain name, and domain suffix. To do this, put parentheses around the parts of the pattern to segment:

In [71]:
```
pattern = r'([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\.([A-Z]{2,4})'
regex = re.compile(pattern, flags=re.IGNORECASE)
```

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

In [72]:
```
m = regex.match('wesm@bright.net')
m.groups()
```

Out[72]:
```
('wesm', 'bright', 'net')
```

findall returns a list of tuples when the pattern has groups:

```
In [73]: regex.findall(text)
```

```
Out[73]: [('dave', 'google', 'com'),
          ('steve', 'gmail', 'com'),
          ('rob', 'gmail', 'com'),
          ('ryan', 'yahoo', 'com')]
```

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

```
In [74]: 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
```

There is much more to regular expressions in Python, most of which is outside our scope. Table 5-4 provides a brief summary.

Table 5.4: Regular expression methods

Argument	Description
`findall`	Return all non-overlapping matching patterns in a string as a list
`finditer`	Like `findall`, but returns an iterator
`match`	Match pattern at start of string and optionally segment pattern components into groups; if the pattern matches, returns a match object, and otherwise None
`search`	Scan string for match to pattern; returning a match object if so; unlike `match`, the match can be anywhere in the string as opposed to only at the beginning
`split`	Break string into pieces at each occurrence of pattern
`sub, subn`	Replace all (sub) or first n occurrences (subn) of pattern in string with replacement expression; use symbols `\1, \2, ...` to refer to match group elements in the replacement string

5.3.3 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 [75]: data = {'Dave': 'dave@google.com', 'Steve': 'steve@gmail.com', 'Rob': 'rob@gmail.com', 'Wes': np.na
         n}
         data = pd.Series(data)
         data
```

```
Out[75]: Dave     dave@google.com
         Steve    steve@gmail.com
         Rob        rob@gmail.com
         Wes                  NaN
         dtype: object
```

```
In [76]: data.isnull()
```

```
Out[76]: Dave     False
         Steve    False
         Rob      False
         Wes       True
         dtype: bool
```

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

In [77]:
```
data.str.contains('gmail')
```

Out[77]:
```
Dave     False
Steve     True
Rob       True
Wes        NaN
dtype: object
```

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

In [78]:
```
pattern
```

Out[78]:
```
'([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\\.([A-Z]{2,4})'
```

In [79]:
```
data.str.findall(pattern, flags=re.IGNORECASE)
```

Out[79]:
```
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 [80]:
```
matches = data.str.match(pattern, flags=re.IGNORECASE)
matches
```

Out[80]:
```
Dave     True
Steve    True
Rob      True
Wes       NaN
dtype: object
```

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

In [87]:
```
matches.str.get(1)
```

Out[87]:
```
Dave    NaN
Steve   NaN
Rob     NaN
Wes     NaN
dtype: float64
```

In [82]:
```
matches.str[0]
```

Out[82]:
```
Dave    NaN
Steve   NaN
Rob     NaN
Wes     NaN
dtype: float64
```

You can similarly slice strings using this syntax:

In [88]:
```
data.str[:5]
```

Out[88]:
```
Dave     dave@
Steve    steve
Rob      rob@g
Wes        NaN
dtype: object
```

See Table 5-5 for more pandas string methods.

Table 5.5: Partial listing of vectorized string methods

Method	Description
`cat`	Concatenate strings element-wise with optional delimiter
`contains`	Return boolean array if each string contains pattern/regex
`count`	Count occurrences of pattern
`extract`	Use a regular expression with groups to extract one or more strings from a Series of strings; the result will be a DataFrame with one column per group
`endswith`	Equivalent to `x.endswith(pattern)` for each element
`startswith`	Equivalent to `x.startswith(pattern)` for each element
`findall`	Compute list of all occurrences of pattern/regex for each string
`get`	Index into each element (retrieve *i*-th element)
`isalnum`	Equivalent to built-in `str.alnum`
`isalpha`	Equivalent to built-in `str.isalpha`
`isdecimal`	Equivalent to built-in `str.isdecimal`
`isdigit`	Equivalent to built-in `str.isdigit`
`islower`	Equivalent to built-in `str.islower`
`isnumeric`	Equivalent to built-in `str.isnumeric`
`isupper`	Equivalent to built-in `str.isupper`
`join`	Join strings in each element of the Series with passed separator
`len`	Compute length of each string
`lower, upper`	Convert cases; equivalent to `x.lower()` or `x.upper()` for each element
`match`	Use `re.match` with the passed regular expression on each element, returning matched groups as list
`pad`	Add whitespace to left, right, or both sides of strings
`center`	Equivalent to `pad(side='both')`
`repeat`	Duplicate values (e.g., `s.str.repeat(3)` is equivalent to `x * 3` for each string)
`replace`	Replace occurrences of pattern/regex with some other string
`slice`	Slice each string in the Series
`split`	Split strings on delimiter or regular expression
`strip`	Trim whitespace from both sides, including newlines
`rstrip`	Trim whitespace on right side
`lstrip`	Trim whitespace on left side