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

7.1 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
string_data = pd.Series(['aardvark', 'artichoke', np.nan, 'avocado'])
```

In [2]:
```
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
```

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

Out[5]:
```
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.

Table 7-1. 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.

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

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

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

Out[8]:
```
0    1.0
2    3.5
4    7.0
dtype: float64
```

This is equivalent to:

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

Out[9]:
```
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 [10]:
```
data = pd.DataFrame([[1., 6.5, 3.], [1., NA, NA],[NA, NA, NA], [NA, 6.5, 3.]])
```

In [11]:
```
cleaned = data.dropna()
```

In [12]:
```
data
```

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

In [13]:
```
cleaned
```

Out[13]:

	0	1	2
0	1.0	6.5	3.0

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

In [14]:
```
data.dropna(how='all')
```

Out[14]:

	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 [15]:
```
data[4] = NA
```

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

Out[16]:

	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 [17]:
```
data.dropna(axis=1, how='all')
```

Out[17]:

	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 [18]:
```
 df = pd.DataFrame(np.random.randn(7, 3))
```

In [19]:
```
df.iloc[:4, 1] = NA
```

In [20]:
```
df.iloc[:2, 2] = NA
df
```

Out[20]:

	0	1	2
0	1.716190	NaN	NaN
1	0.758215	NaN	NaN
2	1.458738	NaN	-0.222721
3	0.018101	NaN	-1.843687
4	1.125204	1.644142	-0.209151
5	-1.465942	0.080446	-0.432103
6	-1.684639	-1.514129	-2.032758

In [21]:
```
df.dropna()
```

Out[21]:

	0	1	2
4	1.125204	1.644142	-0.209151
5	-1.465942	0.080446	-0.432103
6	-1.684639	-1.514129	-2.032758

In [22]:
```
df.dropna(thresh=2)
```

Out[22]:

	0	1	2
2	1.458738	NaN	-0.222721
3	0.018101	NaN	-1.843687
4	1.125204	1.644142	-0.209151
5	-1.465942	0.080446	-0.432103
6	-1.684639	-1.514129	-2.032758

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 [23]:
```
df.fillna(0)
```

Out[23]:

	0	1	2
0	1.716190	0.000000	0.000000
1	0.758215	0.000000	0.000000
2	1.458738	0.000000	-0.222721
3	0.018101	0.000000	-1.843687
4	1.125204	1.644142	-0.209151
5	-1.465942	0.080446	-0.432103
6	-1.684639	-1.514129	-2.032758

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

In [24]:
```
df.fillna({1: 0.5, 2: 0})
```

Out[24]:

	0	1	2
0	1.716190	0.500000	0.000000
1	0.758215	0.500000	0.000000
2	1.458738	0.500000	-0.222721
3	0.018101	0.500000	-1.843687
4	1.125204	1.644142	-0.209151
5	-1.465942	0.080446	-0.432103
6	-1.684639	-1.514129	-2.032758

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

In [25]:
```
 _ = df.fillna(0, inplace=True)
```

In [26]:
```
df
```

Out[26]:

	0	1	2
0	1.716190	0.000000	0.000000
1	0.758215	0.000000	0.000000
2	1.458738	0.000000	-0.222721
3	0.018101	0.000000	-1.843687
4	1.125204	1.644142	-0.209151
5	-1.465942	0.080446	-0.432103
6	-1.684639	-1.514129	-2.032758

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

In [27]:
```
df = pd.DataFrame(np.random.randn(6, 3))
```

In [28]:
```
df.iloc[2:, 1] = NA
```

In [29]:
```
df.iloc[4:, 2] = NA
```

In [30]:
```
df
```

Out[30]:

	0	1	2
0	-0.403981	2.596961	0.373623
1	-0.576694	0.154581	1.233917
2	-0.045495	NaN	0.360516
3	-1.928080	NaN	1.081916
4	1.323195	NaN	NaN
5	1.184422	NaN	NaN

In [31]:
```
df.fillna(method='ffill')
```

Out[31]:

	0	1	2
0	-0.403981	2.596961	0.373623
1	-0.576694	0.154581	1.233917
2	-0.045495	0.154581	0.360516
3	-1.928080	0.154581	1.081916
4	1.323195	0.154581	1.081916
5	1.184422	0.154581	1.081916

In [32]:
```
df.fillna(method='ffill', limit=2)
```

Out[32]:

	0	1	2
0	-0.403981	2.596961	0.373623
1	-0.576694	0.154581	1.233917
2	-0.045495	0.154581	0.360516
3	-1.928080	0.154581	1.081916
4	1.323195	NaN	1.081916
5	1.184422	NaN	1.081916

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 [33]:
```
data = pd.Series([1., NA, 3.5, NA, 7])
```

In [34]:
```
data.fillna(data.mean())
```

Out[34]:
```
0    1.000000
1    3.833333
2    3.500000
3    3.833333
4    7.000000
dtype: float64
```

Table 7-2. 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

7.2 Data Transformation

Removing Duplicates

In [35]:
```
data = pd.DataFrame({'k1': ['one', 'two'] * 3 + ['two'],'k2': [1, 1, 2, 3, 3, 4, 4]
data
```

Out[35]:

	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 boolean Series indicating whether each row is a duplicate (has been observed in a previous row) or not:

In [36]:
```
data.duplicated()
```

Out[36]:
```
0    False
1    False
2    False
3    False
4    False
5    False
6     True
dtype: bool
```

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

In [37]:
```
data.drop_duplicates()
```

Out[37]:

	k1	k2
0	one	1
1	two	1
2	one	2
3	two	3
4	one	3
5	two	4

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 [38]:
```
data['v1'] = range(7)
```

In [39]:
```
data
```

Out[39]:

	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 [40]:
```
data.drop_duplicates(['k1'])
```

Out[40]:

	k1	k2	v1
0	one	1	0
1	two	1	1

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

In [41]:
```
data.drop_duplicates(['k1', 'k2'], keep='last')
```

Out[41]:

	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

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 meat:

In [42]:
```
data = pd.DataFrame({'food': ['bacon', 'pulled pork', 'bacon','Pastrami', 'corned b
```

In [43]:
```
data
```

Out[43]:

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

In [44]:
```
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 object containing a mapping, but here we have a small problem in that some of the meats are capitalized and others are not. Thus, we need to convert each value to lowercase using the str.lower Series method

In [45]:
```
lowercased = data['food'].str.lower()
```

In [46]:
```
lowercased
```

Out[46]:
```
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 [47]:
```
data['animal'] = lowercased.map(meat_to_animal)
```

In [48]:
```
data
```

Out[48]:

	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 have passed a function that does all the work:

In [49]:
```
data['food'].map(lambda x: meat_to_animal[x.lower()])
```

Out[49]:
```
0       pig
1       pig
2       pig
3       cow
4       cow
5       pig
6       cow
7       pig
8    salmon
Name: food, dtype: object
```

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

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 [50]:
```
data = pd.Series([1., -999., 2., -999., -1000., 3.])
```

In [51]:
```
data
```

Out[51]:
```
0       1.0
1    -999.0
2       2.0
3    -999.0
4   -1000.0
5       3.0
dtype: float64
```

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 [52]:
```
data.replace(-999, np.nan)
```

Out[52]:
```
0       1.0
1       NaN
2       2.0
3       NaN
4   -1000.0
5       3.0
dtype: float64
```

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

In [53]:
```
data.replace([-999, -1000], np.nan)
```

Out[53]:
```
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 substitutes:

In [54]:
```
data.replace([-999, -1000], [np.nan, 0])
```

Out[54]:
```
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 [55]:
```
data.replace({-999: np.nan, -1000: 0})
```

Out[55]:
```
0    1.0
1    NaN
2    2.0
3    NaN
4    0.0
5    3.0
dtype: float64
```

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.

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 [56]:
```
data = pd.DataFrame(np.arange(12).reshape((3, 4)),index=['Ohio', 'Colorado', 'New Y
```

In [57]:
```
data
```

Out[57]:

	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 [58]:
```
transform = lambda x: x[:4].upper()
```

In [59]:
```
data.index.map(transform)
```

Out[59]:
```
Index(['OHIO', 'COLO', 'NEW '], dtype='object')
```

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

In [60]:
```
data.index = data.index.map(transform)
```

In [61]:
```
data
```

Out[61]:

	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 [62]:
```
data.rename(index=str.title, columns=str.upper)
```

Out[62]:

	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 [63]:
```
data.rename(index={'OHIO': 'INDIANA'},columns={'three': 'peekaboo'})
```

Out[63]:

	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 [64]:
```
data.rename(index={'OHIO': 'INDIANA'}, inplace=True)
```

In [65]:
```
data
```

Out[65]:

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

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 [66]:
```
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 [67]:
```
bins = [18, 25, 35, 60, 100]
```

In [68]:
```
cats = pd.cut(ages, bins)
cats
```

Out[68]:
```
[(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 [69]:
```
cats.codes
```

Out[69]:
```
array([0, 0, 0, 1, 0, 0, 2, 1, 3, 2, 2, 1], dtype=int8)
```

In [70]:
```
cats.categories
```

Out[70]:
```
IntervalIndex([(18, 25], (25, 35], (35, 60], (60, 100]],
              closed='right',
              dtype='interval[int64]')
```

In [71]:
```
pd.value_counts(cats)
```

Out[71]:
```
(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 [72]:
```
pd.cut(ages, [18, 26, 36, 61, 100], right=False)
```

Out[72]:
```
[[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 [73]:
```
group_names = ['Youth', 'YoungAdult', 'MiddleAged', 'Senior']
```

In [74]:
```
pd.cut(ages, bins, labels=group_names)
```

Out[74]:
```
['Youth', 'Youth', 'Youth', 'YoungAdult', 'Youth', ..., 'YoungAdult', 'Senior', 'M
iddleAged', 'MiddleAged', 'YoungAdult']
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 [75]:
```
data = np.random.rand(20)
```

In [76]:
```
data
```

Out[76]:
```
array([0.73537446, 0.45852963, 0.242461  , 0.6798634 , 0.21123872,
       0.51646697, 0.01183484, 0.23061445, 0.01605296, 0.59781633,
       0.61254167, 0.15803606, 0.84163006, 0.48022942, 0.37551369,
       0.43149055, 0.73851027, 0.92030394, 0.38014579, 0.06505485])
```

In [77]:
```
pd.cut(data, 4, precision=2)
```

Out[77]:
```
[(0.69, 0.92], (0.24, 0.47], (0.24, 0.47], (0.47, 0.69], (0.011, 0.24], ..., (0.2
4, 0.47], (0.69, 0.92], (0.69, 0.92], (0.24, 0.47], (0.011, 0.24]]
Length: 20
Categories (4, interval[float64]): [(0.011, 0.24] < (0.24, 0.47] < (0.47, 0.69] <
(0.69, 0.92]]
```

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 [78]:
```
data = np.random.randn(1000) # Normally distributed
```

In [79]:
```
data
```

```
Out[79]: array([-1.14487773e+00,  7.88131466e-01, -8.60069474e-01, -9.51275040e-01,
                 9.89472165e-01,  8.26985859e-01, -1.42521374e-01, -6.22141446e-01,
                 1.33237531e+00,  1.00816123e+00,  1.43794115e-01,  1.07646329e+00,
                 6.81086622e-06, -8.58980451e-01,  3.16332186e-01, -9.34486234e-01,
                -2.78908437e-01, -2.48190208e-01, -7.95160838e-01, -3.40100686e-01,
                 8.86265755e-01,  1.39985044e+00, -5.46725605e-01, -1.22923968e+00,
                 8.70843388e-01, -1.38884475e+00, -4.43779483e-02, -9.14219558e-01,
                -4.93813310e-01, -9.12873467e-02,  3.94067834e-01, -8.72498594e-01,
                -9.25801681e-01, -7.79585447e-01, -3.52987508e-01,  1.88336451e+00,
                -6.89650465e-02, -4.34866855e-01,  1.10144202e+00, -8.45829224e-01,
                -7.85241005e-01,  1.40796477e-01,  5.89757591e-01,  1.73502550e+00,
                -3.88629648e-01, -5.76089447e-01,  6.14487853e-01,  2.14663187e+00,
                -3.04455616e-01, -2.08384938e+00,  7.84835849e-01,  4.94567626e-01,
                -2.70977991e-01, -3.23368366e-01, -7.30330820e-01,  3.85269553e-02,
                -5.15071217e-01,  2.27754353e-01,  8.67796666e-01, -8.43942595e-01,
                 4.85330326e-01, -1.68874984e+00,  2.93073206e-01,  1.08227112e+00,
                -6.49844242e-01,  1.26549718e+00,  1.85900058e+00,  1.04469810e+00,
                -1.69826196e+00, -1.90255954e-01, -4.33675173e-01,  2.29151625e-01,
                 6.81639525e-01, -8.77995212e-01,  1.16964485e+00,  6.81571526e-01,
                 1.83510240e-01, -3.44317976e-01,  1.28673095e-01,  7.29470660e-01,
                 6.94506187e-01,  2.73199701e-01, -6.46863484e-01, -6.27937470e-02,
                 5.80648812e-01, -1.76305484e-01,  2.84233285e-01,  1.59297840e+00,
                 5.60090194e-01, -3.29622027e-01, -5.66396840e-01, -1.11953934e+00,
                -6.21548181e-01,  1.15310508e+00, -2.20285395e-01,  2.79687812e-01,
                -5.53054471e-01, -4.54048692e-01,  7.87224906e-01, -1.91881084e-01,
                 1.19715328e+00, -3.89444801e+00, -6.93798174e-03, -7.82865846e-01,
                -1.51342677e-01,  8.23739010e-01,  2.02035898e-02,  1.14679183e+00,
                 1.91793040e+00,  2.80559734e-01, -4.80547970e-01, -9.06333995e-01,
                 1.00818351e+00,  4.17534055e-01,  4.70324197e-01,  3.45927937e-01,
                -1.30831875e+00,  1.69752032e+00, -1.28253637e+00, -2.26855783e+00,
                -1.66826220e+00, -3.89504185e-01,  6.17208429e-01,  8.57018849e-01,
                -1.82551822e+00,  5.63839813e-02,  8.41685262e-01,  6.68928117e-01,
                -6.09378541e-01,  1.07187056e+00, -5.75472737e-01,  7.07349787e-01,
                 2.00846475e-01,  2.35228445e-02, -6.12696751e-01, -1.17572811e+00,
                -1.21984318e+00,  1.25997314e+00, -1.10855884e+00, -3.66651268e-01,
                 2.97881658e-01,  1.26725127e+00, -9.92265952e-01, -7.48236878e-01,
                -7.90726654e-01, -4.18744552e-01, -9.93105347e-01,  8.98690144e-01,
                -4.36235065e-01, -7.41909315e-01, -1.32337372e+00, -1.18492932e+00,
                -6.93475013e-01, -1.43973035e+00, -1.66425101e+00,  1.61370815e-01,
                -4.48889755e-01, -4.35068246e-02, -4.32747410e-01,  4.68781454e-01,
                -1.01621869e+00, -3.68559697e-01, -4.64050737e-01, -1.43964128e+00,
                 4.38894841e-01,  1.38343485e+00, -1.01836547e+00,  5.45872286e-01,
                 1.07584519e+00, -1.26576123e+00, -1.14932801e+00, -1.00224387e-01,
                -4.84286661e-01, -9.89080764e-02,  3.21581255e-01,  2.24639397e-01,
                -6.69405665e-01,  3.11927233e-01,  5.42904688e-01,  1.81479128e-01,
                -1.56783108e+00, -6.50412494e-01,  1.55924147e-01,  8.54666529e-01,
                 1.30265619e+00, -1.45917201e-01,  5.59558788e-01, -1.13095881e+00,
                 3.36237708e-01,  1.50128730e+00,  9.06680776e-01,  5.75482264e-02,
                -1.99733232e+00,  6.98613030e-01, -1.74815316e+00, -5.25239256e-01,
                 4.53193072e-01,  6.02839145e-01,  1.18795996e+00, -1.27267187e+00,
                 6.05369561e-01,  1.87530149e-01, -8.37285301e-01, -1.07135001e-01,
                 4.69206665e-01, -1.12431783e+00,  5.44923522e-03, -1.82452235e-01,
                -1.13382086e-01, -1.51838572e-01,  1.26468432e+00,  1.44630124e+00,
                -1.15508163e+00,  9.25575345e-01, -1.42146990e+00, -6.65699195e-01,
                 1.17088719e+00, -1.42193844e+00, -2.86030698e-01,  2.26534711e-01,
                 7.28279489e-01,  7.37535812e-01,  4.02168913e-01,  1.18858165e+00,
```

```
-8.52529257e-01, -1.48471017e+00,  7.20002531e-01, -3.80990379e-01,
-1.75499246e-01,  2.75065701e-01, -1.17863106e+00,  2.93584657e-01,
 7.20155681e-01,  7.32088202e-01,  5.64054112e-01, -7.55689713e-01,
 1.12636626e+00, -7.69810076e-01, -9.17409076e-02,  1.80139279e+00,
-3.45479904e-01, -1.97756697e+00, -4.24732119e-01, -4.71064664e-01,
 7.33237021e-01,  4.53634333e-01, -1.30057055e+00, -1.85875541e-01,
-8.36610256e-01, -4.65364300e-01, -1.90547835e+00,  1.10345691e+00,
 5.14034992e-01, -1.27606210e+00, -1.35026672e+00, -9.20693819e-01,
 6.55908015e-01, -1.77046513e+00, -2.79095718e-01,  1.24951912e+00,
 3.99958599e-01, -1.75438866e-01, -9.81725551e-02, -8.55334057e-01,
-1.77647427e+00,  1.91496967e+00,  1.15886719e+00, -1.34271123e+00,
-1.08069169e-02,  1.02369735e+00, -2.96683068e-01, -1.40064704e+00,
-3.66732431e-01,  9.01910978e-01, -5.86564630e-01, -5.28111942e-01,
 1.83212516e+00,  1.47894011e+00,  3.19503086e-01,  6.99811789e-01,
-8.44057672e-01,  3.16165834e-01,  1.89839916e+00,  3.12556024e-01,
-3.38237748e-01,  1.60828063e+00, -9.01107515e-01,  8.35403361e-02,
-1.17893869e+00,  4.04793802e-01, -1.04394722e+00,  4.39071975e-01,
 2.22350718e-01,  8.92179408e-01,  2.04651744e-02, -4.46258617e-01,
-1.24197494e-01, -1.91559169e+00,  1.38602535e+00, -6.71066840e-03,
-1.07611435e+00, -1.17862077e-01, -5.97395798e-01, -3.67966727e-01,
-3.60187214e-01,  6.26873419e-01,  7.05349705e-01, -5.10362604e-01,
-1.03452171e+00, -2.65894269e-01,  1.39878760e+00,  2.45964734e-02,
 6.88349897e-01,  1.54531792e+00,  7.58255420e-01,  1.78770953e-01,
 6.47840815e-02,  1.78104076e-01, -3.00104085e-01, -6.98252114e-01,
-1.16100640e+00, -2.84057148e-01, -1.81529512e+00,  1.31770108e+00,
-2.65805643e-01,  1.88741510e+00, -7.50717360e-01,  7.17558157e-01,
-5.04040075e-01, -5.02530657e-01,  3.02484239e-01,  9.60832646e-01,
-3.32260866e-01,  1.15689659e-01,  9.28019863e-01, -4.88059759e-02,
-4.62822474e-02,  2.34778542e+00, -1.85661408e-01, -2.95766183e-01,
-1.21310511e+00,  1.17851319e+00, -6.30292065e-01,  1.04727567e-02,
-3.05636940e-01, -3.99780005e-01,  6.58753595e-01, -1.49727632e+00,
-2.54484756e+00,  6.34720928e-01, -3.01376930e-01,  4.88618171e-01,
-1.08396965e+00,  1.89985254e-01, -1.81140881e+00, -1.18361265e+00,
-6.71367480e-01,  2.16957135e-01,  4.58682994e-01,  5.98056847e-01,
 4.42820379e-01, -2.15679121e-01,  5.38250577e-01,  1.11419315e+00,
 7.93659002e-01, -8.85114921e-01,  7.16603746e-01,  7.64798011e-01,
 6.83789584e-01,  1.84944609e-01, -1.30863121e+00, -4.39798318e-01,
-6.26089177e-01, -1.84573690e+00,  2.07653474e+00, -5.88742596e-01,
-9.84468349e-01, -3.62991801e-01, -1.75788974e+00, -1.60723470e-01,
-1.86191658e+00, -2.30047273e-01, -1.57432925e+00,  1.89255845e+00,
 1.03463306e+00,  9.83588100e-01, -1.12237794e+00,  8.79535892e-02,
-4.90063277e-01,  1.56930792e+00, -2.39151276e-01,  1.85506094e-01,
-1.37359207e+00,  2.33537388e-02,  1.18782329e+00,  1.10190123e+00,
-1.47235114e-01,  1.14869585e+00, -1.02737204e+00, -1.89397360e+00,
 8.34953791e-01, -1.01801950e+00, -5.05262629e-01, -1.16026724e+00,
-4.88241705e-01, -8.48720047e-01,  5.96240880e-01, -4.74856491e-01,
-6.88351862e-01,  3.23517996e-01, -5.39849770e-01,  1.14375038e-02,
-1.17576032e+00, -1.36753538e+00,  2.42454568e-01, -1.69101478e+00,
-6.31588357e-01, -2.11886798e-01,  5.59566280e-01,  1.26246957e+00,
 8.82494142e-01,  6.37304324e-02, -1.45573481e+00, -3.38646690e-02,
-7.19721926e-01, -2.32201767e-01,  2.97363021e-01,  1.02961688e+00,
-1.44271821e+00,  1.02960333e+00,  7.38897569e-01, -1.16372566e+00,
-6.96547638e-02,  1.47493655e+00,  2.65934151e-01,  1.19886822e+00,
-9.40390970e-01, -5.61192423e-01,  1.57520303e-01,  1.22169293e+00,
 1.22754426e+00, -1.37364511e+00, -4.28473171e-01, -3.20334150e-01,
 9.02400406e-01,  7.04972936e-02, -1.14228622e-01, -4.59765599e-01,
```

```
 2.26679101e-01, -2.43843583e-01, -1.09140594e+00,  6.20412492e-01,
 1.82985072e+00, -9.98225209e-01, -5.06946352e-01,  7.89148177e-01,
 6.22283319e-02, -1.05265705e+00, -7.68855572e-01, -1.71345824e+00,
 1.40238199e+00, -1.14094201e+00,  1.04846114e+00, -7.97688643e-01,
-3.18407121e-01,  7.86037503e-02, -7.10983915e-01, -1.36420872e+00,
-2.90790562e-01,  4.52635985e-01,  2.20610831e+00,  6.15540778e-01,
 6.63946586e-01,  5.85965829e-01,  3.19465376e-01,  4.78415050e-01,
 1.28585807e+00,  8.45275812e-01,  8.66222546e-01,  1.28112055e-02,
 1.23800931e+00, -1.80135267e+00, -2.53513139e-01, -5.17379825e-01,
 2.30548201e-01,  5.49356286e-01, -3.73927385e-01, -1.30553628e-01,
-1.20383738e-01,  3.81496237e-01, -2.64599344e-01, -3.52004680e-01,
-4.27404531e-01, -1.14123687e+00,  1.87558368e-01, -8.98441851e-01,
-1.11717057e+00,  2.29739743e+00, -2.24299196e+00, -1.47212854e+00,
 3.59530881e-01, -2.78620374e-02,  1.12782241e+00, -7.49976347e-01,
 1.14535936e+00,  2.19607792e+00,  1.13100981e+00,  5.54737511e-01,
-1.74295046e+00,  1.48167293e-01,  8.02849710e-01,  1.02598440e+00,
 7.96122658e-01, -1.03842811e-01,  1.59691866e-01,  4.76715944e-01,
 4.36711497e-01,  1.64625761e-01,  1.05564010e+00, -1.73136829e-01,
 1.68182291e-01, -5.34716849e-01, -7.67284940e-01, -2.64034876e-01,
-1.05951025e+00,  4.03243918e-01, -9.69020439e-01,  6.13009770e-01,
-6.54014985e-01, -8.60790009e-01,  9.25822998e-02,  1.29096633e-01,
 1.25448654e-01,  1.72473522e+00, -3.64581002e-01,  1.30063672e+00,
 4.85161690e-01,  1.10123726e+00,  2.21859775e-01, -5.21901575e-01,
 1.69042402e-01, -1.25022415e+00,  1.43568132e-02,  6.06104926e-01,
 1.88114562e-01,  5.99225029e-02,  3.68477495e-01, -1.23846728e+00,
 8.87056801e-01, -6.50987237e-01, -6.46598830e-01, -5.91076770e-01,
-1.18885538e+00, -6.23392418e-01, -7.39152520e-01, -2.45823276e+00,
-3.27338400e-01,  5.60506737e-01,  8.74289462e-01,  1.35391534e+00,
 2.49407323e-01,  4.16730853e-01, -1.02952874e-01, -1.70259301e+00,
-7.96296559e-01,  1.63351190e-01,  4.86712566e-01,  7.47290006e-01,
-1.17636838e+00,  7.18431000e-01, -2.36707616e+00,  1.79679888e+00,
 4.75099965e-01,  7.63185972e-02, -3.98727696e-01,  1.17733644e+00,
 7.29811598e-01, -2.89814716e-01,  4.35179741e-01, -8.49411365e-01,
 2.63466059e+00,  2.06639187e+00, -8.73053532e-01, -7.30916166e-01,
 1.49135225e+00, -2.09707474e-01,  9.33857552e-01,  6.55651728e-01,
-1.08658455e+00,  8.41907143e-01, -1.52605255e+00, -2.15386701e+00,
 1.82754125e+00,  2.71292250e-01, -2.35549875e+00, -8.67465705e-01,
-1.19765282e+00, -2.21807697e+00, -1.47615657e-01, -1.75548704e-01,
-1.88690836e+00,  1.83817075e+00,  6.06436893e-01,  1.90749087e+00,
 3.10165425e-02,  3.53685432e-01, -1.22836327e+00, -9.55558788e-01,
-5.83741152e-01,  1.97834512e-01, -7.77327508e-01,  1.19238550e+00,
-1.04795884e+00, -8.59991585e-01, -2.82412115e-01, -1.59515601e-01,
-8.32883429e-01, -1.84230366e-01, -1.07019512e+00,  9.76877513e-01,
-1.06667775e+00,  2.26194406e-01, -2.15607830e-02,  5.91067785e-01,
-8.61240635e-01,  4.94849414e-01,  1.35433305e+00, -2.25817939e+00,
-1.15878866e+00, -4.45188020e-01,  8.17584928e-01, -1.50043078e+00,
 4.88200798e-01,  1.71979092e-01,  6.75488400e-01, -4.75042426e-01,
 9.50917942e-03,  1.69101133e+00, -9.69188768e-01,  1.56568140e+00,
 5.85715762e-01,  1.35108411e+00,  8.64077179e-01, -5.34738959e-01,
-3.27059373e+00,  6.14547288e-01,  1.00186507e+00, -1.56596277e+00,
 1.81255589e+00, -2.01729316e+00, -1.07448377e+00, -1.21136475e+00,
 1.31280281e+00,  9.57475199e-01, -8.09659153e-01, -1.22717241e+00,
-8.86465509e-03,  4.69111403e-01,  1.46740425e+00, -1.30495570e+00,
 2.31637475e-01,  4.72372039e-01,  5.04553309e-02, -2.19079288e-01,
 1.19772092e-01, -5.97138230e-01, -1.55788571e+00,  5.21515581e-01,
-2.21714823e+00,  1.53024124e+00,  6.25739470e-01,  1.97271686e+00,
```

```
 1.42612307e+00, -1.58168806e+00,  5.09183190e-01,  4.13507757e-01,
-1.05386553e-01,  7.18786347e-02,  4.60370366e-01, -4.13819697e-01,
-5.92285611e-01, -6.38804531e-01,  6.57750828e-03,  6.36463672e-01,
-4.20669375e-02, -2.05353312e+00,  3.47914613e-01,  1.11498529e-01,
 8.74204114e-02, -5.17456945e-01, -9.66688255e-02,  6.41353459e-01,
-4.25041708e-02,  9.85290661e-01, -4.07129747e-01, -5.93118235e-01,
-1.13368711e+00, -1.76028713e+00,  3.26347100e-01,  8.73819796e-02,
 3.26434644e-01, -1.49911445e-01,  1.41624917e+00, -4.21769074e-01,
-4.21817406e-01,  9.41369874e-01,  2.02029459e+00, -2.31519089e-01,
-4.33389948e-01,  1.88718626e+00, -6.92738212e-01, -1.75710747e+00,
-6.36399982e-01,  8.07388607e-01,  5.46756258e-01, -2.54306774e-01,
 2.52633587e-01,  1.35006680e+00, -1.96507196e+00,  4.02530311e-01,
 1.26530071e-01, -5.27758200e-01,  2.02593762e+00,  4.64919274e-01,
 9.05609017e-01, -7.88802342e-01,  4.86743573e-01,  1.69074988e-01,
-7.86629086e-01,  6.05906672e-01, -3.23582788e-01,  1.16129673e+00,
 3.42157317e-01,  6.69164562e-01, -1.31306121e+00, -5.50799894e-01,
-1.09657076e+00,  5.71811063e-02,  1.54970962e-01, -1.00919866e+00,
-4.07218891e-01, -8.18444660e-01,  1.10970761e+00,  6.35501832e-01,
 8.76500317e-01,  5.48586699e-01,  9.41389188e-01,  1.47913665e+00,
-4.24779300e-01, -3.01924302e-02, -1.53510778e+00, -1.58441707e+00,
-6.93820380e-01,  1.36245267e-01, -1.18529373e+00,  3.52925773e-01,
-5.84781933e-01, -5.75363521e-01, -1.17427574e-01, -5.96139823e-02,
 6.28439683e-01,  9.04837142e-01,  2.01491323e+00,  2.95868090e-02,
 3.23581045e-01,  8.68136455e-02,  1.47128762e+00, -1.49466716e+00,
-2.89031536e-01,  2.46259558e-01, -1.25184051e+00, -9.80261099e-01,
 3.01439042e-01,  8.90522949e-01, -2.92382355e-01, -1.59646346e-02,
-1.68907962e+00, -8.45507037e-01,  4.33696425e-01, -1.87332987e+00,
 2.20775785e+00,  1.74397346e-01,  1.33615013e+00, -6.34950607e-01,
 1.32794973e+00,  1.38502509e+00,  5.98899933e-02, -1.65359596e+00,
 7.32623587e-01,  3.43471932e-01,  1.70473418e+00, -9.49173187e-01,
 7.04346675e-01, -6.95151360e-01,  1.55344917e+00,  1.09433054e+00,
-2.75536084e-01,  8.03230309e-01,  1.94286507e-01, -2.17371729e+00,
-2.67117686e-01, -4.66756438e-01, -7.30312501e-01, -2.24027329e+00,
 1.61478509e+00,  1.05729881e+00,  5.65257926e-01, -2.22519203e+00,
-4.35556338e-01, -1.01245728e+00,  1.77305285e+00,  1.44250787e+00,
-9.62324540e-01, -4.61506124e-01, -3.07512515e-01,  4.01069282e-01,
 8.81099013e-01,  8.73910314e-01,  3.58185399e-01,  1.00952855e-01,
 1.39845210e+00,  2.50420547e+00,  2.44374412e-01,  4.14740626e-01,
-1.07076444e+00,  1.01581086e+00, -1.77134891e-01, -9.54773653e-01,
-9.51609928e-01, -9.13407078e-03,  1.63612179e-01,  1.93595087e+00,
 9.20240067e-01,  4.71957995e-01, -1.66562686e+00, -5.12859031e-01,
 6.05041973e-01,  2.80353681e-01,  7.88115552e-01,  1.67404274e+00,
 5.20094721e-02, -1.34678116e+00,  2.09358443e+00,  4.65625280e-02,
 1.09840043e+00, -8.50882367e-01, -1.84943935e+00,  2.01560734e-01,
-7.44869125e-01, -4.92008465e-01, -2.18067893e-01,  2.56194173e-01,
-6.90637484e-01,  1.62101053e+00, -8.54981660e-01,  2.85613273e-01,
-2.95374765e+00, -1.26267253e+00,  1.01311500e+00, -7.70794967e-01,
 6.73579520e-01, -2.30168497e+00, -9.18947461e-01, -9.72102846e-01,
 5.60856909e-01, -9.76068698e-01,  2.23308945e-01, -3.61284078e-01,
 8.48993854e-01, -6.46716285e-01, -1.48181589e+00,  8.99913672e-01,
-1.11233632e+00,  5.49820099e-01,  1.05799369e+00,  3.57045262e-01,
-7.31219460e-01,  9.14731673e-02, -2.00979639e+00, -1.99929987e+00,
-2.51914363e+00,  1.70884160e-01, -5.95681505e-01, -1.85471574e+00,
 1.68634577e+00, -6.69755177e-01, -3.17421513e-01, -2.80944752e-01,
 1.47620116e+00,  2.54716794e-01, -1.32435921e+00,  3.46993826e-01,
 1.19226604e+00,  2.43814441e-01,  8.48517399e-01,  3.29617497e-01,
```

```
       -1.80146115e+00,  2.16510179e+00, -3.31936476e-01,  8.78876501e-01,
        5.80177326e-01, -7.51342106e-01, -1.16429490e-05,  6.76531509e-01,
       -1.05410620e+00, -1.84638980e-01, -1.03156076e+00,  1.24611634e+00,
       -6.35667598e-01, -8.73832316e-01,  1.19153177e+00,  1.37057413e-01,
        1.26433418e+00, -8.42911876e-01, -9.24273849e-02, -1.94697737e+00,
        3.96742857e-01, -1.52191519e+00,  7.72956250e-01,  1.59919694e+00,
        2.64484163e-01,  2.83071063e-01, -7.44960787e-01,  1.29556161e+00,
       -2.16941947e+00, -1.52174024e+00,  1.18516781e-01,  8.78244680e-01,
       -7.66841995e-01,  5.99250013e-01,  6.58342458e-01, -8.23997914e-01,
       -1.66670827e-01,  7.12648511e-01, -7.59035829e-01, -1.77589154e+00,
       -7.07088279e-01,  5.79196218e-01,  5.72757788e-01,  9.20240682e-01,
       -1.52095649e-01, -1.15988159e+00,  1.12014503e+00,  1.20372177e+00,
        6.46799846e-01,  6.16496347e-01, -3.12907526e-01, -6.28375500e-01,
       -6.51354445e-01,  3.43369331e-01,  4.96397260e-01, -9.04302217e-02,
        2.46560320e-01,  5.14580835e-01,  5.89140304e-01,  8.74585945e-02,
       -1.33901892e+00, -1.49161466e+00, -1.34182496e+00,  1.92320232e-01,
        1.60880315e-01,  6.10340735e-01, -9.56475510e-02, -7.14102790e-01,
       -1.28372929e+00, -5.42519321e-01,  1.82582305e-01, -9.83571392e-01,
        7.44812564e-01, -3.83901074e-01, -3.06752302e-01, -1.45761222e-01,
       -1.73004737e-01, -1.95807145e+00,  6.32397694e-02, -1.00621271e+00,
       -7.12266765e-01,  1.04283928e-01, -9.75509857e-01,  1.05910291e+00,
        9.37888352e-01,  2.06338270e+00, -6.64237938e-01,  4.51443018e-01,
       -3.90434545e-01,  1.04140850e+00, -1.27166734e+00, -1.11063358e+00,
        6.27386952e-01, -1.53219511e-02,  5.05961276e-01,  4.03308600e-01,
       -1.69520307e-01,  1.89149142e-01, -9.43299192e-01,  5.45041981e-01,
       -3.12825522e-01,  7.22016392e-01,  6.61768865e-01, -1.82060319e+00])
```

In [80]:
```python
cats = pd.qcut(data, 4) # Cut into quartiles
```

In [81]:
```python
cats
```

Out[81]:
```
[(-3.895, -0.74], (0.63, 2.635], (-3.895, -0.74], (-3.895, -0.74], (0.63, 2.635],
..., (-0.009, 0.63], (-0.74, -0.009], (0.63, 2.635], (0.63, 2.635], (-3.895, -0.7
4]]
Length: 1000
Categories (4, interval[float64]): [(-3.895, -0.74] < (-0.74, -0.009] < (-0.009,
0.63] < (0.63, 2.635]]
```

In [84]:
```python
pd.value_counts(cats)
```

Out[84]:
```
(0.63, 2.635]      250
(-0.009, 0.63]     250
(-0.74, -0.009]    250
(-3.895, -0.74]    250
dtype: int64
```

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

In [85]:
```python
pd.qcut(data, [0, 0.1, 0.5, 0.9, 1.])
```

Out[85]:
```
[(-1.308, -0.009], (-0.009, 1.192], (-1.308, -0.009], (-1.308, -0.009], (-0.009, 
1.192], ..., (-0.009, 1.192], (-1.308, -0.009], (-0.009, 1.192], (-0.009, 1.192], 
(-3.895, -1.308]]
Length: 1000
Categories (4, interval[float64]): [(-3.895, -1.308] < (-1.308, -0.009] < (-0.009, 
1.192] < (1.192, 2.635]]
```

Detecting and Filtering Outliers

Filtering or transforming outliers is largely a matter of applying array operations. Consider a DataFrame with some normally distributed data:

In [86]:
```
data = pd.DataFrame(np.random.randn(1000, 4))
```

In [87]:
```
data.describe()
```

Out[87]:

	0	1	2	3
count	1000.000000	1000.000000	1000.000000	1000.000000
mean	-0.022110	-0.043906	-0.037331	-0.054070
std	1.014601	1.007231	1.015098	0.954552
min	-3.182219	-3.420015	-3.368210	-2.689002
25%	-0.696666	-0.714993	-0.727664	-0.699930
50%	-0.036531	-0.052193	-0.068682	-0.045962
75%	0.655823	0.636319	0.658800	0.585116
max	3.686829	3.830218	3.063229	2.890968

Suppose you wanted to find values in one of the columns exceeding 3 in absolute value:

In [88]:
```
col = data[2]
```

In [89]:
```
col
```

Out[89]:
```
0      -1.009705
1       1.020642
2       0.003759
3       0.947368
4       2.037514
          ...   
995    -0.065644
996    -0.119238
997     2.334061
998    -2.849141
999     0.052484
Name: 2, Length: 1000, dtype: float64
```

In [90]:
```
col[np.abs(col) > 3]
```

Out[90]:
```
479   -3.368210
694    3.063229
Name: 2, dtype: float64
```

To select all rows having a value exceeding 3 or –3, you can use the any method on a boolean DataFrame:

In [91]:
```
data[(np.abs(data) > 3).any(1)]
```

Out[91]:

	0	1	2	3
52	1.374740	3.385276	-1.629590	1.183052
104	-3.182219	0.191703	0.838626	-0.155851
280	-0.682679	3.830218	1.963083	-1.602334
293	0.440736	-3.055962	-0.747369	-0.386925
321	-0.930775	-3.306304	1.968698	0.281366
325	3.686829	0.940408	0.183505	-0.853655
371	0.817982	3.102559	0.601612	0.036518
398	-3.027455	2.411899	-0.397356	0.705882
468	3.136631	0.837253	-0.433790	-2.087663
479	0.767217	-0.410861	-3.368210	0.982346
680	3.120405	-0.349164	-0.746571	0.065153
683	-3.026335	-0.981515	-0.143394	0.437291
694	-0.587708	0.343727	3.063229	-0.019740
929	1.751564	-3.420015	1.512408	-1.189738

Values can be set based on these criteria. Here is code to cap values outside the interval –3 to 3:

In [92]:
```
data[np.abs(data) > 3] = np.sign(data) * 3
```

In [93]:
```
data.describe()
```

Out[93]:

	0	1	2	3
count	1000.000000	1000.000000	1000.000000	1000.000000
mean	-0.022818	-0.044442	-0.037026	-0.054070
std	1.010820	1.000364	1.013763	0.954552
min	-3.000000	-3.000000	-3.000000	-2.689002
25%	-0.696666	-0.714993	-0.727664	-0.699930
50%	-0.036531	-0.052193	-0.068682	-0.045962
75%	0.655823	0.636319	0.658800	0.585116
max	3.000000	3.000000	3.000000	2.890968

The statement np.sign(data) produces 1 and –1 values based on whether the values in data are positive or negative:

In [94]:
```
np.sign(data).head()
```

Out[94]:

	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

Permutation and Random Sampling

Permuting (randomly reordering) a Series or the rows in a DataFrame is easy to do using the numpy.random.permutation function. Calling permutation with the length of the axis you want to permute produces an array of integers indicating the new ordering:

In [95]:
```
df = pd.DataFrame(np.arange(5 * 4).reshape((5, 4)))
df
```

Out[95]:

	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 [96]:
```
sampler = np.random.permutation(5)
sampler
```

Out[96]:
```
array([3, 2, 1, 4, 0])
```

That array can then be used in iloc-based indexing or the equivalent take function:

In [97]:
```
df.take(sampler)
```

Out[97]:

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

To select a random subset without replacement, you can use the sample method on Series and DataFrame:

In [98]:
```
df.sample(n=3)
```

Out[98]:

	0	1	2	3
0	0	1	2	3
2	8	9	10	11
1	4	5	6	7

To generate a sample with replacement (to allow repeat choices), pass replace=True to sample:

In [99]:
```
choices = pd.Series([5, 7, -1, 6, 4])
```

In [100…
```
draws = choices.sample(n=10, replace=True)
draws
```

Out[100…

```
4    4
1    7
2   -1
4    4
1    7
2   -1
1    7
3    6
2   -1
3    6
dtype: int64
```

Computing Indicator/Dummy Variables

Another type of transformation for statistical modeling or machine learning applications is converting a categorical variable into a "dummy" or "indicator" matrix. If a column in a DataFrame has k distinct values, you would derive a matrix or DataFrame with k columns containing all 1s and 0s. pandas has a get_dummies function for doing this, though devising one yourself is not difficult. Let's return to an earlier example DataFrame:

In [101…

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

In [102…

```
df
```

Out[102…

	key	data1
0	b	0
1	b	1
2	a	2
3	c	3
4	a	4
5	b	5

In [103…

```
pd.get_dummies(df['key'])
```

Out[103…

	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

In some cases, you may want to add a prefix to the columns in the indicator DataFrame, which can then be merged with the other data. get_dummies has a prefix argument for doing this:

In [104…
```
dummies = pd.get_dummies(df['key'], prefix='key')
```

In [105…
```
df_with_dummy = df[['data1']].join(dummies)
```

In [106…
```
df_with_dummy
```

Out[106…

	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

If a row in a DataFrame belongs to multiple categories, things are a bit more complicated. Let's look at the MovieLens 1M dataset, which is investigated in more detail in Chapter 14:

In [107…
```
mnames = ['movie_id', 'title', 'genres']
```

In [108…
```
movies = pd.read_table('movies.dat', sep='::',header=None, names=mnames)
```

```
C:\Users\ankit19.gupta\OneDrive - Reliance Corporate IT Park Limited\Desktop\Practic
e_Code\Python_Practice\Python_For_Data_Analysis\myenv\lib\site-packages\pandas\io\pa
rsers.py:767: ParserWarning: Falling back to the 'python' engine because the 'c' eng
ine does not support regex separators (separators > 1 char and different from '\s+'
are interpreted as regex); you can avoid this warning by specifying engine='python'.
  return read_csv(**locals())
```

In [109…
```
movies[:10]
```

Out[109…

	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

Adding indicator variables for each genre requires a little bit of wrangling. First, we extract the list of unique genres in the dataset:

In [110…
```
all_genres = []
```

In [111…
```
for x in movies.genres:
    all_genres.extend(x.split('|'))
```

In [112…
```
genres = pd.unique(all_genres)
```

In [113…
```
genres
```

Out[113…
```
array(['Animation', "Children's", 'Comedy', 'Adventure', 'Fantasy',
       'Romance', 'Drama', 'Action', 'Crime', 'Thriller', 'Horror',
       'Sci-Fi', 'Documentary', 'War', 'Musical', 'Mystery', 'Film-Noir',
       'Western'], dtype=object)
```

One way to construct the indicator DataFrame is to start with a DataFrame of all zeros:

In [114…
```
zero_matrix = np.zeros((len(movies), len(genres)))
```

In [115…
```
zero_matrix
```

Out[115…
```
array([[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.]])
```

In [116…
```
dummies = pd.DataFrame(zero_matrix, columns=genres)
```

```
dummies
```

Out[116...

	Animation	Children's	Comedy	Adventure	Fantasy	Romance	Drama	Action	Crin
0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	C
1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	C
2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	C
3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	C
4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	C
...	...	...	...	...	...	...	...	...	
3878	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	C
3879	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	C
3880	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	C
3881	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	C
3882	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	C

3883 rows × 18 columns

Now, iterate through each movie and set entries in each row of dummies to 1. To do this, we use the dummies.columns to compute the column indices for each genre:

In [117...
```
gen = movies.genres[0]
```

In [118...
```
gen.split('|')
```

Out[118...
```
['Animation', "Children's", 'Comedy']
```

In [119...
```
dummies.columns.get_indexer(gen.split('|'))
```

Out[119...
```
array([0, 1, 2], dtype=int64)
```

Then, we can use .iloc to set values based on these indices:

In [120...
```
for i, gen in enumerate(movies.genres):
    indices = dummies.columns.get_indexer(gen.split('|'))
    dummies.iloc[i, indices] = 1
```

Then, as before, you can combine this with movies:

In [121...
```
movies_windic = movies.join(dummies.add_prefix('Genre_'))
```

In [122...
```
movies_windic.iloc[0]
```

Out[122…

```
movie_id                                   1
title                       Toy Story (1995)
genres           Animation|Children's|Comedy
Genre_Animation                            1
Genre_Children's                           1
Genre_Comedy                               1
Genre_Adventure                            0
Genre_Fantasy                              0
Genre_Romance                              0
Genre_Drama                                0
Genre_Action                               0
Genre_Crime                                0
Genre_Thriller                             0
Genre_Horror                               0
Genre_Sci-Fi                               0
Genre_Documentary                          0
Genre_War                                  0
Genre_Musical                              0
Genre_Mystery                              0
Genre_Film-Noir                            0
Genre_Western                              0
Name: 0, dtype: object
```

For much larger data, this method of constructing indicator vari- ables with multiple membership is not especially speedy. It would be better to write a lower-level function that writes directly to a NumPy array, and then wrap the result in a DataFrame.

A useful recipe for statistical applications is to combine get_dummies with a discreti- zation function like cut:

In [123…

```
np.random.seed(12345)
```

In [124…

```
values = np.random.rand(10)
```

In [125…

```
values
```

Out[125…

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

In [126…

```
bins = [0, 0.2, 0.4, 0.6, 0.8, 1]
```

In [127…

```
pd.get_dummies(pd.cut(values, bins))
```

Out[127…

	(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 String Manipulation

String Object Methods

In [128…

```
val = 'a,b, guido'
```

In [129…

```
val.split(',')
```

Out[129…

```
['a', 'b', ' guido']
```

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

In [130…

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

Out[130…

```
['a', 'b', 'guido']
```

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

In [131…

```
first, second, third = pieces
```

In [132…

```
first + '::' + second + '::' + third
```

Out[132…

```
'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 [133…

```
'::'.join(pieces)
```

Out[133…
```
'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 [134…
```
'guido' in val
```

Out[134…
```
True
```

In [135…
```
val.index(',')
```

Out[135…
```
1
```

In [137…
```
val.find(':')
```

Out[137…
```
-1
```

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

In [138…
```
val.index(':')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-138-2c016e7367ac> in <module>
----> 1 val.index(':')

ValueError: substring not found
```

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

In [139…
```
val.count(',')
```

Out[139…
```
2
```

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

In [141…
```
val.replace(',', '::')
```

Out[141…
```
'a::b:: guido'
```

In [142…
```
val.replace(',', '')
```

Out[142…
```
'ab guido'
```

Table 7-3. 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 su□x.

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 □rst 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.

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; I'll give a number of examples of its use here.

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 [144…
```
import re
```

In [145…
```
text = "foo bar\t baz \tqux"
```

In [146…
```
re.split('\s+', text)
```

Out[146…
```
['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 [147…
```
regex = re.compile('\s+')
```

In [148…
```
regex.split(text)
```

Out[148…
```
['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 [149…
```
regex.findall(text)
```

Out[149…
```
[' ', '\t ', ' \t']
```

To avoid unwanted escaping with \ in a regular expression, use raw string literals like r'C:\x' instead of the equivalent 'C:\x'.

Creating a regex object with re.compile is highly recommended if you intend to apply the same expression to many strings; doing so will save CPU cycles.

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 [150…
```
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)
```

In [152…
```
regex
```

Out[152…
```
re.compile(r'[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}', re.IGNORECASE|re.UNICODE)
```

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

In [153…
```
regex.findall(text)
```

Out[153…
```
['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 [154…
```
m = regex.search(text)
```

In [155…
```
m
```

Out[155…
```
<_sre.SRE_Match object; span=(5, 20), match='dave@google.com'>
```

In [156…
```
text[m.start():m.end()]
```

Out[156…
```
'dave@google.com'
```

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

In [157…
```
print(regex.match(text))
```

```
None
```

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

In [158…
```
 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 [159…
```
pattern = r'([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\.([A-Z]{2,4})'
```

In [160…
```
regex = re.compile(pattern, flags=re.IGNORECASE)
```

A match object produced by this modified regex returns a tuple of the pattern com- ponents with its groups method:

In [161…
```
m = regex.match('wesm@bright.net')
m.groups()
```
Out[161…
```
('wesm', 'bright', 'net')
```

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

In [162…
```
regex.findall(text)
```
Out[162…
```
[('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 [163…
```
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
```

Table 7-4. 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

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 [164...
```
data = {'Dave': 'dave@google.com', 'Steve': 'steve@gmail.com','Rob': 'rob@gmail.com
```

In [165...
```
data = pd.Series(data)
data
```

Out[165...
```
Dave     dave@google.com
Steve    steve@gmail.com
Rob        rob@gmail.com
Wes                  NaN
dtype: object
```

In [166...
```
data.isnull()
```

Out[166...
```
Dave     False
Steve    False
Rob      False
Wes       True
dtype: bool
```

You can apply 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 [167...
```
data.str.contains('gmail')
```

Out[167...
```
Dave     False
Steve     True
Rob       True
Wes        NaN
dtype: object
```

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

In [168...
```
pattern
```

Out[168...
```
'([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\\.([A-Z]{2,4})'
```

In [169...
```
data.str.findall(pattern, flags=re.IGNORECASE)
```

Out[169...
```
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 [170…
```
matches = data.str.match(pattern, flags=re.IGNORECASE)
```

In [171…
```
matches
```

Out[171…
```
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 [172…
```
matches.str.get(1)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-172-bd3a29697b06> in <module>
----> 1 matches.str.get(1)

~\OneDrive - Reliance Corporate IT Park Limited\Desktop\Practice_Code\Python_Practic
e\Python_For_Data_Analysis\myenv\lib\site-packages\pandas\core\generic.py in __getat
tr__(self, name)
   5135             or name in self._accessors
   5136         ):
-> 5137             return object.__getattribute__(self, name)
   5138         else:
   5139             if self._info_axis._can_hold_identifiers_and_holds_name(name):

~\OneDrive - Reliance Corporate IT Park Limited\Desktop\Practice_Code\Python_Practic
e\Python_For_Data_Analysis\myenv\lib\site-packages\pandas\core\accessor.py in __get_
_(self, obj, cls)
    185             # we're accessing the attribute of the class, i.e., Dataset.geo
    186             return self._accessor
--> 187         accessor_obj = self._accessor(obj)
    188         # Replace the property with the accessor object. Inspired by:
    189         # https://www.pydanny.com/cached-property.html

~\OneDrive - Reliance Corporate IT Park Limited\Desktop\Practice_Code\Python_Practic
e\Python_For_Data_Analysis\myenv\lib\site-packages\pandas\core\strings.py in __init_
_(self, data)
   2098
   2099     def __init__(self, data):
-> 2100         self._inferred_dtype = self._validate(data)
   2101         self._is_categorical = is_categorical_dtype(data.dtype)
   2102         self._is_string = data.dtype.name == "string"

~\OneDrive - Reliance Corporate IT Park Limited\Desktop\Practice_Code\Python_Practic
e\Python_For_Data_Analysis\myenv\lib\site-packages\pandas\core\strings.py in _valida
te(data)
   2155
   2156         if inferred_dtype not in allowed_types:
-> 2157             raise AttributeError("Can only use .str accessor with string val
ues!")
   2158         return inferred_dtype
   2159

AttributeError: Can only use .str accessor with string values!
```

In [173…

```
matches.str[0]
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-173-10bdd22fd8b2> in <module>
----> 1 matches.str[0]

~\OneDrive - Reliance Corporate IT Park Limited\Desktop\Practice_Code\Python_Practic
e\Python_For_Data_Analysis\myenv\lib\site-packages\pandas\core\generic.py in __getat
tr__(self, name)
   5135             or name in self._accessors
   5136         ):
-> 5137             return object.__getattribute__(self, name)
   5138         else:
   5139             if self._info_axis._can_hold_identifiers_and_holds_name(name):

~\OneDrive - Reliance Corporate IT Park Limited\Desktop\Practice_Code\Python_Practic
e\Python_For_Data_Analysis\myenv\lib\site-packages\pandas\core\accessor.py in __get_
_(self, obj, cls)
    185             # we're accessing the attribute of the class, i.e., Dataset.geo
    186             return self._accessor
--> 187         accessor_obj = self._accessor(obj)
    188         # Replace the property with the accessor object. Inspired by:
    189         # https://www.pydanny.com/cached-property.html

~\OneDrive - Reliance Corporate IT Park Limited\Desktop\Practice_Code\Python_Practic
e\Python_For_Data_Analysis\myenv\lib\site-packages\pandas\core\strings.py in __init_
_(self, data)
   2098 
   2099     def __init__(self, data):
-> 2100         self._inferred_dtype = self._validate(data)
   2101         self._is_categorical = is_categorical_dtype(data.dtype)
   2102         self._is_string = data.dtype.name == "string"

~\OneDrive - Reliance Corporate IT Park Limited\Desktop\Practice_Code\Python_Practic
e\Python_For_Data_Analysis\myenv\lib\site-packages\pandas\core\strings.py in _valida
te(data)
   2155 
   2156         if inferred_dtype not in allowed_types:
-> 2157             raise AttributeError("Can only use .str accessor with string val
ues!")
   2158         return inferred_dtype
   2159 

AttributeError: Can only use .str accessor with string values!
```

You can similarly slice strings using this syntax:

In [174…
```
data.str[:5]
```

Out[174…
```
Dave     dave@
Steve    steve
Rob      rob@g
Wes        NaN
dtype: object
```

Table 7-5. 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

In [ ]: