CHAPTER 7

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.

If you identify a type of data manipulation that isn't anywhere in this book or elsewhere in the pandas library, feel free to share your use case on one of the Python mailing lists or on the pandas GitHub site. Indeed, much of the design and implementation of pandas have been driven by the needs of real-world applications.

In this chapter I discuss tools for missing data, duplicate data, string manipulation, and some other analytical data transformations. In the next chapter, I focus on combining and rearranging datasets in various ways.

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 sufficient for most real-world use. For data with `float64` dtype, pandas uses the floating-point value `NaN` (Not a Number) to represent missing data.

We call this a *sentinel value*: when present, it indicates a missing (or *null*) value:

```
In [14]: float_data = pd.Series([1.2, -3.5, np.nan, 0])

In [15]: float_data
Out[15]:
0    1.2
1   -3.5
2    NaN
3    0.0
dtype: float64
```

The `isna` method gives us a Boolean Series with `True` where values are null:

```
In [16]: float_data.isna()
Out[16]:
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 [17]: string_data = pd.Series(["aardvark", np.nan, None, "avocado"])

In [18]: string_data
Out[18]:
0    aardvark
1         NaN
2        None
3     avocado
dtype: object

In [19]: string_data.isna()
Out[19]:
0    False
1     True
2     True
3    False
dtype: bool

In [20]: float_data = pd.Series([1, 2, None], dtype='float64')

In [21]: float_data
Out[21]:
```

```
0    1.0
1    2.0
2    NaN
dtype: float64

In [22]: float_data.isna()
Out[22]:
0    False
1    False
2     True
dtype: bool
```

The pandas project has attempted to make working with missing data consistent across data types. Functions like `pandas.isna` abstract away many of the annoying details. See Table 7-1 for a list of some functions related to missing data handling.

Table 7-1. NA handling object methods

Method	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"`.
`isna`	Return Boolean values indicating which values are missing/NA.
`notna`	Negation of `isna`, returns `True` for non-NA values and `False` for NA values.

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.isna` and Boolean indexing, `dropna` can be helpful. On a Series, it returns the Series with only the nonnull data and index values:

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

In [24]: data.dropna()
Out[24]:
0    1.0
2    3.5
4    7.0
dtype: float64
```

This is the same thing as doing:

```
In [25]: data[data.notna()]
Out[25]:
0    1.0
2    3.5
4    7.0
dtype: float64
```

With DataFrame objects, there are different ways to remove missing data. You may want to drop rows or columns that are all NA, or only those rows or columns containing any NAs at all. `dropna` by default drops any row containing a missing value:

```
In [26]: data = pd.DataFrame([[1., 6.5, 3.], [1., np.nan, np.nan],
   ....:                      [np.nan, np.nan, np.nan], [np.nan, 6.5, 3.]])

In [27]: data
Out[27]: 
     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 [28]: data.dropna()
Out[28]: 
     0    1    2
0  1.0  6.5  3.0
```

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

```
In [29]: data.dropna(how="all")
Out[29]: 
     0    1    2
0  1.0  6.5  3.0
1  1.0  NaN  NaN
3  NaN  6.5  3.0
```

Keep in mind that these functions return new objects by default and do not modify the contents of the original object.

To drop columns in the same way, pass `axis="columns"`:

```
In [30]: data[4] = np.nan

In [31]: data
Out[31]: 
     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 [32]: data.dropna(axis="columns", how="all")
Out[32]: 
     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
```

Suppose you want to keep only rows containing at most a certain number of missing observations. You can indicate this with the `thresh` argument:

```
In [33]: df = pd.DataFrame(np.random.standard_normal((7, 3)))

In [34]: df.iloc[:4, 1] = np.nan

In [35]: df.iloc[:2, 2] = np.nan

In [36]: df
Out[36]:
          0         1         2
0 -0.204708       NaN       NaN
1 -0.555730       NaN       NaN
2  0.092908       NaN  0.769023
3  1.246435       NaN -1.296221
4  0.274992  0.228913  1.352917
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741

In [37]: df.dropna()
Out[37]:
          0         1         2
4  0.274992  0.228913  1.352917
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741

In [38]: df.dropna(thresh=2)
Out[38]:
          0         1         2
2  0.092908       NaN  0.769023
3  1.246435       NaN -1.296221
4  0.274992  0.228913  1.352917
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741
```

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 [39]: df.fillna(0)
Out[39]:
          0         1         2
0 -0.204708  0.000000  0.000000
1 -0.555730  0.000000  0.000000
2  0.092908  0.000000  0.769023
3  1.246435  0.000000 -1.296221
4  0.274992  0.228913  1.352917
```

```
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741
```

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

```
In [40]: df.fillna({1: 0.5, 2: 0})
Out[40]:
          0         1         2
0 -0.204708  0.500000  0.000000
1 -0.555730  0.500000  0.000000
2  0.092908  0.500000  0.769023
3  1.246435  0.500000 -1.296221
4  0.274992  0.228913  1.352917
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741
```

The same interpolation methods available for reindexing (see Table 5-3) can be used with `fillna`:

```
In [41]: df = pd.DataFrame(np.random.standard_normal((6, 3)))

In [42]: df.iloc[2:, 1] = np.nan

In [43]: df.iloc[4:, 2] = np.nan

In [44]: df
Out[44]:
          0         1         2
0  0.476985  3.248944 -1.021228
1 -0.577087  0.124121  0.302614
2  0.523772       NaN  1.343810
3 -0.713544       NaN -2.370232
4 -1.860761       NaN       NaN
5 -1.265934       NaN       NaN

In [45]: df.fillna(method="ffill")
Out[45]:
          0         1         2
0  0.476985  3.248944 -1.021228
1 -0.577087  0.124121  0.302614
2  0.523772  0.124121  1.343810
3 -0.713544  0.124121 -2.370232
4 -1.860761  0.124121 -2.370232
5 -1.265934  0.124121 -2.370232

In [46]: df.fillna(method="ffill", limit=2)
Out[46]:
          0         1         2
0  0.476985  3.248944 -1.021228
1 -0.577087  0.124121  0.302614
2  0.523772  0.124121  1.343810
3 -0.713544  0.124121 -2.370232
```

```
4 -1.860761        NaN -2.370232
5 -1.265934        NaN -2.370232
```

With `fillna` you can do lots of other things such as simple data imputation using the median or mean statistics:

```
In [47]: data = pd.Series([1., np.nan, 3.5, np.nan, 7])

In [48]: data.fillna(data.mean())
Out[48]: 
0    1.000000
1    3.833333
2    3.500000
3    3.833333
4    7.000000
dtype: float64
```

See Table 7-2 for a reference on `fillna` function arguments.

Table 7-2. `fillna` function arguments

Argument	Description
`value`	Scalar value or dictionary-like object to use to fill missing values
`method`	Interpolation method: one of `"bfill"` (backward fill) or `"ffill"` (forward fill); default is `None`
`axis`	Axis to fill on (`"index"` or `"columns"`); default is `axis="index"`
`limit`	For forward and backward filling, maximum number of consecutive periods to fill

7.2 Data Transformation

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

Removing Duplicates

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

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

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

The DataFrame method `duplicated` returns a Boolean Series indicating whether each row is a duplicate (its column values are exactly equal to those in an earlier row) or not:

```
In [51]: data.duplicated()
Out[51]:
0    False
1    False
2    False
3    False
4    False
5    False
6     True
dtype: bool
```

Relatedly, `drop_duplicates` returns a DataFrame with rows where the `duplicated` array is `False` filtered out:

```
In [52]: data.drop_duplicates()
Out[52]:
    k1  k2
0  one   1
1  two   1
2  one   2
3  two   3
4  one   3
5  two   4
```

Both 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 based only on the `"k1"` column:

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

In [54]: data
Out[54]:
    k1  k2  v1
0  one   1   0
1  two   1   1
2  one   2   2
3  two   3   3
4  one   3   4
5  two   4   5
6  two   4   6

In [55]: data.drop_duplicates(subset=["k1"])
Out[55]:
    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 [56]: data.drop_duplicates(["k1", "k2"], keep="last")
Out[56]:
    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 [57]: data = pd.DataFrame({"food": ["bacon", "pulled pork", "bacon",
   ....:                               "pastrami", "corned beef", "bacon",
   ....:                               "pastrami", "honey ham", "nova lox"],
   ....:                      "ounces": [4, 3, 12, 6, 7.5, 8, 3, 5, 6]})

In [58]: data
Out[58]:
          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:

```
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 (also discussed in "Function Application and Mapping" on page 158) accepts a function or dictionary-like object containing a mapping to do the transformation of values:

```
In [60]: data["animal"] = data["food"].map(meat_to_animal)

In [61]: data
Out[61]:
          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 [62]: def get_animal(x):
   ....:     return meat_to_animal[x]

In [63]: data["food"].map(get_animal)
Out[63]:
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 [64]: data = pd.Series([1., -999., 2., -999., -1000., 3.])

In [65]: data
Out[65]:
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:

```
In [66]: data.replace(-999, np.nan)
Out[66]:
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 [67]: data.replace([-999, -1000], np.nan)
Out[67]:
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 [68]: data.replace([-999, -1000], [np.nan, 0])
Out[68]:
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 dictionary:

```
In [69]: data.replace({-999: np.nan, -1000: 0})
Out[69]:
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 element-wise string substitution. 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 [70]: data = pd.DataFrame(np.arange(12).reshape((3, 4)),
   ....:                     index=["Ohio", "Colorado", "New York"],
   ....:                     columns=["one", "two", "three", "four"])
```

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

```
In [71]: def transform(x):
   ....:     return x[:4].upper()

In [72]: data.index.map(transform)
Out[72]: Index(['OHIO', 'COLO', 'NEW '], dtype='object')
```

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

```
In [73]: data.index = data.index.map(transform)

In [74]: data
Out[74]: 
      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 [75]: data.rename(index=str.title, columns=str.upper)
Out[75]: 
      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 dictionary-like object, providing new values for a subset of the axis labels:

```
In [76]: data.rename(index={"OHIO": "INDIANA"},
   ....:             columns={"three": "peekaboo"})
Out[76]: 
         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 new values to its `index` and `columns` attributes.

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 [77]: 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 `pandas.cut`:

```
In [78]: bins = [18, 25, 35, 60, 100]

In [79]: age_categories = pd.cut(ages, bins)

In [80]: age_categories
Out[80]: 
[(18, 25], (18, 25], (18, 25], (25, 35], (18, 25], ..., (25, 35], (60, 100], (35,
 60], (35, 60], (25, 35]]
Length: 12
Categories (4, interval[int64, right]): [(18, 25] < (25, 35] < (35, 60] < (60, 10
0]]
```

The object pandas returns is a special Categorical object. The output you see describes the bins computed by `pandas.cut`. Each bin is identified by a special (unique to pandas) interval value type containing the lower and upper limit of each bin:

```
In [81]: age_categories.codes
Out[81]: array([0, 0, 0, 1, 0, 0, 2, 1, 3, 2, 2, 1], dtype=int8)

In [82]: age_categories.categories
Out[82]: IntervalIndex([(18, 25], (25, 35], (35, 60], (60, 100]], dtype='interval
[int64, right]')

In [83]: age_categories.categories[0]
Out[83]: Interval(18, 25, closed='right')

In [84]: pd.value_counts(age_categories)
Out[84]: 
(18, 25]     5
```

```
(25, 35]     3
(35, 60]     3
(60, 100]    1
dtype: int64
```

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

In the string representation of an interval, a parenthesis means that the side is *open* (exclusive), while the square bracket means it is *closed* (inclusive). You can change which side is closed by passing right=False:

```
In [85]: pd.cut(ages, bins, right=False)
Out[85]:
[[18, 25), [18, 25), [25, 35), [25, 35), [18, 25), ..., [25, 35), [60, 100), [35,
 60), [35, 60), [25, 35)]
Length: 12
Categories (4, interval[int64, left]): [[18, 25) < [25, 35) < [35, 60) < [60, 100
)]
```

You can override the default interval-based bin labeling by passing a list or array to the labels option:

```
In [86]: group_names = ["Youth", "YoungAdult", "MiddleAged", "Senior"]

In [87]: pd.cut(ages, bins, labels=group_names)
Out[87]:
['Youth', 'Youth', 'Youth', 'YoungAdult', 'Youth', ..., 'YoungAdult', 'Senior', '
MiddleAged', 'MiddleAged', 'YoungAdult']
Length: 12
Categories (4, object): ['Youth' < 'YoungAdult' < 'MiddleAged' < 'Senior']
```

If you pass an integer number of bins to pandas.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 [88]: data = np.random.uniform(size=20)

In [89]: pd.cut(data, 4, precision=2)
Out[89]:
[(0.34, 0.55], (0.34, 0.55], (0.76, 0.97], (0.76, 0.97], (0.34, 0.55], ..., (0.34
, 0.55], (0.34, 0.55], (0.55, 0.76], (0.34, 0.55], (0.12, 0.34]]
Length: 20
Categories (4, interval[float64, right]): [(0.12, 0.34] < (0.34, 0.55] < (0.55, 0
.76] <
                                           (0.76, 0.97]]
```

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

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

```
In [90]: data = np.random.standard_normal(1000)

In [91]: quartiles = pd.qcut(data, 4, precision=2)

In [92]: quartiles
Out[92]:
[(-0.026, 0.62], (0.62, 3.93], (-0.68, -0.026], (0.62, 3.93], (-0.026, 0.62], ...
, (-0.68, -0.026], (-0.68, -0.026], (-2.96, -0.68], (0.62, 3.93], (-0.68, -0.026]
]
Length: 1000
Categories (4, interval[float64, right]): [(-2.96, -0.68] < (-0.68, -0.026] < (-0
.026, 0.62] <
                                           (0.62, 3.93]]

In [93]: pd.value_counts(quartiles)
Out[93]:
(-2.96, -0.68]     250
(-0.68, -0.026]    250
(-0.026, 0.62]     250
(0.62, 3.93]       250
dtype: int64
```

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

```
In [94]: pd.qcut(data, [0, 0.1, 0.5, 0.9, 1.]).value_counts()
Out[94]:
(-2.9499999999999997, -1.187]    100
(-1.187, -0.0265]                400
(-0.0265, 1.286]                 400
(1.286, 3.928]                   100
dtype: int64
```

We'll return to pandas.cut and pandas.qcut later in the chapter during our discussion of aggregation and group operations, as these discretization functions are especially useful for quantile and group analysis.

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 [95]: data = pd.DataFrame(np.random.standard_normal((1000, 4)))

In [96]: data.describe()
Out[96]:
                 0            1            2            3
count  1000.000000  1000.000000  1000.000000  1000.000000
mean      0.049091     0.026112    -0.002544    -0.051827
```

```
std       0.996947     1.007458     0.995232     0.998311
min      -3.645860    -3.184377    -3.745356    -3.428254
25%      -0.599807    -0.612162    -0.687373    -0.747478
50%       0.047101    -0.013609    -0.022158    -0.088274
75%       0.756646     0.695298     0.699046     0.623331
max       2.653656     3.525865     2.735527     3.366626
```

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

```
In [97]: col = data[2]

In [98]: col[col.abs() > 3]
Out[98]:
41    -3.399312
136   -3.745356
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 [99]: data[(data.abs() > 3).any(axis="columns")]
Out[99]:
            0         1         2         3
41   0.457246 -0.025907 -3.399312 -0.974657
60   1.951312  3.260383  0.963301  1.201206
136  0.508391 -0.196713 -3.745356 -1.520113
235 -0.242459 -3.056990  1.918403 -0.578828
258  0.682841  0.326045  0.425384 -3.428254
322  1.179227 -3.184377  1.369891 -1.074833
544 -3.548824  1.553205 -2.186301  1.277104
635 -0.578093  0.193299  1.397822  3.366626
782 -0.207434  3.525865  0.283070  0.544635
803 -3.645860  0.255475 -0.549574 -1.907459
```

The parentheses around `data.abs() > 3` are necessary in order to call the any method on the result of the comparison operation.

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

```
In [100]: data[data.abs() > 3] = np.sign(data) * 3

In [101]: data.describe()
Out[101]:
                 0            1            2            3
count  1000.000000  1000.000000  1000.000000  1000.000000
mean      0.050286     0.025567    -0.001399    -0.051765
std       0.992920     1.004214     0.991414     0.995761
min      -3.000000    -3.000000    -3.000000    -3.000000
25%      -0.599807    -0.612162    -0.687373    -0.747478
50%       0.047101    -0.013609    -0.022158    -0.088274
```

```
75%       0.756646    0.695298    0.699046    0.623331
max       2.653656    3.000000    2.735527    3.000000
```

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

```
In [102]: np.sign(data).head()
Out[102]:
     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 possible 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 [103]: df = pd.DataFrame(np.arange(5 * 7).reshape((5, 7)))

In [104]: df
Out[104]:
    0   1   2   3   4   5   6
0   0   1   2   3   4   5   6
1   7   8   9  10  11  12  13
2  14  15  16  17  18  19  20
3  21  22  23  24  25  26  27
4  28  29  30  31  32  33  34

In [105]: sampler = np.random.permutation(5)

In [106]: sampler
Out[106]: array([3, 1, 4, 2, 0])
```

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

```
In [107]: df.take(sampler)
Out[107]:
    0   1   2   3   4   5   6
3  21  22  23  24  25  26  27
1   7   8   9  10  11  12  13
4  28  29  30  31  32  33  34
2  14  15  16  17  18  19  20
0   0   1   2   3   4   5   6

In [108]: df.iloc[sampler]
Out[108]:
    0   1   2   3   4   5   6
```

```
3  21  22  23  24  25  26  27
1   7   8   9  10  11  12  13
4  28  29  30  31  32  33  34
2  14  15  16  17  18  19  20
0   0   1   2   3   4   5   6
```

By invoking take with axis="columns", we could also select a permutation of the columns:

```
In [109]: column_sampler = np.random.permutation(7)

In [110]: column_sampler
Out[110]: array([4, 6, 3, 2, 1, 0, 5])

In [111]: df.take(column_sampler, axis="columns")
Out[111]:
    4   6   3   2   1   0   5
0   4   6   3   2   1   0   5
1  11  13  10   9   8   7  12
2  18  20  17  16  15  14  19
3  25  27  24  23  22  21  26
4  32  34  31  30  29  28  33
```

To select a random subset without replacement (the same row cannot appear twice), you can use the sample method on Series and DataFrame:

```
In [112]: df.sample(n=3)
Out[112]:
    0   1   2   3   4   5   6
2  14  15  16  17  18  19  20
4  28  29  30  31  32  33  34
0   0   1   2   3   4   5   6
```

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

```
In [113]: choices = pd.Series([5, 7, -1, 6, 4])

In [114]: choices.sample(n=10, replace=True)
Out[114]:
2   -1
0    5
3    6
1    7
4    4
0    5
4    4
0    5
4    4
4    4
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 `pandas.get_dummies` function for doing this, though you could also devise one yourself. Let's consider an example DataFrame:

```
In [115]: df = pd.DataFrame({"key": ["b", "b", "a", "c", "a", "b"],
   .....:                    "data1": range(6)})

In [116]: df
Out[116]: 
  key  data1
0   b      0
1   b      1
2   a      2
3   c      3
4   a      4
5   b      5

In [117]: pd.get_dummies(df["key"])
Out[117]: 
   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. `pandas.get_dummies` has a prefix argument for doing this:

```
In [118]: dummies = pd.get_dummies(df["key"], prefix="key")

In [119]: df_with_dummy = df[["data1"]].join(dummies)

In [120]: df_with_dummy
Out[120]: 
   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
```

The `DataFrame.join` method will be explained in more detail in the next chapter.

If a row in a DataFrame belongs to multiple categories, we have to use a different approach to create the dummy variables. Let's look at the MovieLens 1M dataset, which is investigated in more detail in Chapter 13:

```
In [121]: mnames = ["movie_id", "title", "genres"]

In [122]: movies = pd.read_table("datasets/movielens/movies.dat", sep="::",
   .....:                        header=None, names=mnames, engine="python")

In [123]: movies[:10]
Out[123]:
   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
```

pandas has implemented a special Series method `str.get_dummies` (methods that start with `str.` are discussed in more detail later in Section 7.4, "String Manipulation," on page 227) that handles this scenario of multiple group membership encoded as a delimited string:

```
In [124]: dummies = movies["genres"].str.get_dummies("|")

In [125]: dummies.iloc[:10, :6]
Out[125]:
   Action  Adventure  Animation  Children's  Comedy  Crime
0       0          0          1           1       1      0
1       0          1          0           1       0      0
2       0          0          0           0       1      0
3       0          0          0           0       1      0
4       0          0          0           0       1      0
5       1          0          0           0       0      1
6       0          0          0           0       1      0
7       0          1          0           1       0      0
8       1          0          0           0       0      0
9       1          1          0           0       0      0
```

Then, as before, you can combine this with `movies` while adding a `"Genre_"` to the column names in the `dummies` DataFrame with the `add_prefix` method:

```
In [126]: movies_windic = movies.join(dummies.add_prefix("Genre_"))

In [127]: movies_windic.iloc[0]
Out[127]:
movie_id                                       1
```

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

For much larger data, this method of constructing indicator variables 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 `pandas.get_dummies` with a discretization function like `pandas.cut`:

```
In [128]: np.random.seed(12345) # to make the example repeatable

In [129]: values = np.random.uniform(size=10)

In [130]: values
Out[130]:
array([0.9296, 0.3164, 0.1839, 0.2046, 0.5677, 0.5955, 0.9645, 0.6532,
       0.7489, 0.6536])

In [131]: bins = [0, 0.2, 0.4, 0.6, 0.8, 1]

In [132]: pd.get_dummies(pd.cut(values, bins))
Out[132]:
   (0.0, 0.2]  (0.2, 0.4]  (0.4, 0.6]  (0.6, 0.8]  (0.8, 1.0]
0           0           0           0           0           1
1           0           1           0           0           0
2           1           0           0           0           0
3           0           1           0           0           0
4           0           0           1           0           0
5           0           0           1           0           0
```

```
6           0           0           0           0           1
7           0           0           0           1           0
8           0           0           0           1           0
9           0           0           0           1           0
```

We will look again at `pandas.get_dummies` later in "Creating dummy variables for modeling" on page 245.

7.3 Extension Data Types

This is a newer and more advanced topic that many pandas users do not need to know a lot about, but I present it here for completeness since I will reference and use extension data types in various places in the upcoming chapters.

pandas was originally built upon the capabilities present in NumPy, an array computing library used primarily for working with numerical data. Many pandas concepts, such as missing data, were implemented using what was available in NumPy while trying to maximize compatibility between libraries that used NumPy and pandas together.

Building on NumPy led to a number of shortcomings, such as:

- Missing data handling for some numerical data types, such as integers and Booleans, was incomplete. As a result, when missing data was introduced into such data, pandas converted the data type to `float64` and used `np.nan` to represent null values. This had compounding effects by introducing subtle issues into many pandas algorithms.
- Datasets with a lot of string data were computationally expensive and used a lot of memory.
- Some data types, like time intervals, timedeltas, and timestamps with time zones, could not be supported efficiently without using computationally expensive arrays of Python objects.

More recently, pandas has developed an *extension type* system allowing for new data types to be added even if they are not supported natively by NumPy. These new data types can be treated as first class alongside data coming from NumPy arrays.

Let's look at an example where we create a Series of integers with a missing value:

```
In [133]: s = pd.Series([1, 2, 3, None])

In [134]: s
Out[134]:
0    1.0
```

```
1    2.0
2    3.0
3    NaN
dtype: float64

In [135]: s.dtype
Out[135]: dtype('float64')
```

Mainly for backward compatibility reasons, Series uses the legacy behavior of using a `float64` data type and `np.nan` for the missing value. We could create this Series instead using `pandas.Int64Dtype`:

```
In [136]: s = pd.Series([1, 2, 3, None], dtype=pd.Int64Dtype())

In [137]: s
Out[137]:
0       1
1       2
2       3
3    <NA>
dtype: Int64

In [138]: s.isna()
Out[138]:
0    False
1    False
2    False
3     True
dtype: bool

In [139]: s.dtype
Out[139]: Int64Dtype()
```

The output `<NA>` indicates that a value is missing for an extension type array. This uses the special `pandas.NA` sentinel value:

```
In [140]: s[3]
Out[140]: <NA>

In [141]: s[3] is pd.NA
Out[141]: True
```

We also could have used the shorthand `"Int64"` instead of `pd.Int64Dtype()` to specify the type. The capitalization is necessary, otherwise it will be a NumPy-based nonextension type:

```
In [142]: s = pd.Series([1, 2, 3, None], dtype="Int64")
```

pandas also has an extension type specialized for string data that does not use NumPy object arrays (it requires the pyarrow library, which you may need to install separately):

```
In [143]: s = pd.Series(['one', 'two', None, 'three'], dtype=pd.StringDtype())

In [144]: s
Out[144]:
0      one
1      two
2     <NA>
3    three
dtype: string
```

These string arrays generally use much less memory and are frequently computationally more efficient for doing operations on large datasets.

Another important extension type is `Categorical`, which we discuss in more detail in Section 7.5, "Categorical Data," on page 235. A reasonably complete list of extension types available as of this writing is in Table 7-3.

Extension types can be passed to the Series `astype` method, allowing you to convert easily as part of your data cleaning process:

```
In [145]: df = pd.DataFrame({"A": [1, 2, None, 4],
   .....:                    "B": ["one", "two", "three", None],
   .....:                    "C": [False, None, False, True]})

In [146]: df
Out[146]:
     A      B      C
0  1.0    one  False
1  2.0    two   None
2  NaN  three  False
3  4.0   None   True

In [147]: df["A"] = df["A"].astype("Int64")

In [148]: df["B"] = df["B"].astype("string")

In [149]: df["C"] = df["C"].astype("boolean")

In [150]: df
Out[150]:
      A      B      C
0     1    one  False
1     2    two   <NA>
2  <NA>  three  False
3     4   <NA>   True
```

Table 7-3. pandas extension data types

Extension type	Description
`BooleanDtype`	Nullable Boolean data, use `"boolean"` when passing as string
`CategoricalDtype`	Categorical data type, use `"category"` when passing as string
`DatetimeTZDtype`	Datetime with time zone
`Float32Dtype`	32-bit nullable floating point, use `"Float32"` when passing as string
`Float64Dtype`	64-bit nullable floating point, use `"Float64"` when passing as string
`Int8Dtype`	8-bit nullable signed integer, use `"Int8"` when passing as string
`Int16Dtype`	16-bit nullable signed integer, use `"Int16"` when passing as string
`Int32Dtype`	32-bit nullable signed integer, use `"Int32"` when passing as string
`Int64Dtype`	64-bit nullable signed integer, use `"Int64"` when passing as string
`UInt8Dtype`	8-bit nullable unsigned integer, use `"UInt8"` when passing as string
`UInt16Dtype`	16-bit nullable unsigned integer, use `"UInt16"` when passing as string
`UInt32Dtype`	32-bit nullable unsigned integer, use `"UInt32"` when passing as string
`UInt64Dtype`	64-bit nullable unsigned integer, use `"UInt64"` when passing as string

7.4 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.

Python Built-In 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 [151]: val = "a,b,  guido"

In [152]: val.split(",")
Out[152]: ['a', 'b', '  guido']
```

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

```
In [153]: pieces = [x.strip() for x in val.split(",")]

In [154]: pieces
Out[154]: ['a', 'b', 'guido']
```