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
```

```
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") 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
```

NOTE

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
                                 (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,”) 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
```

NOTE

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".

7.3 Extension Data Types

NOTE

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,”. 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']
```

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

```
In [155]: first, second, third = pieces

In [156]: first + "::" + second + "::" + third
Out[156]: '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 [157]: "::".join(pieces)
Out[157]: '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 [158]: "guido" in val
Out[158]: True

In [159]: val.index(",")
Out[159]: 1

In [160]: val.find(":")
Out[160]: -1
```

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

```
In [161]: val.index(":")
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-161-bea4c4c30248> in <module>
----> 1 val.index(":")
ValueError: substring not found
```

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

```
In [162]: val.count(",")
Out[162]: 2
```

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

```
In [163]: val.replace(",", "::")
Out[163]: 'a::b::  guido'

In [164]: val.replace(",", "")
Out[164]: 'ab  guido'
```

See Table 7-4 for a listing of some of Python’s string methods.

Regular expressions can also be used with many of these operations, as you’ll see.

Table 7-4. Python built-in string methods

Method	Description
count	Return the number of nonoverlapping occurrences of substring in the string
endswith	Return True if string ends with suffix
startswith	Return True if string starts with prefix
join	Use string as delimiter for concatenating a sequence of other strings
index	Return starting index of the first occurrence of passed substring if found in the string; otherwise, raises ValueError if not found
find	Return position of first character of *first* occurrence of substring in the string; like index, but returns –1 if not found
rfind	Return position of first character of *last* occurrence of substring in the string; returns –1 if not found
replace	Replace occurrences of string with another string
strip, rstrip, lstrip	Trim whitespace, including newlines on both sides, on the right side, or on the left side, respectively
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.

NOTE

The art of writing regular expressions could be a chapter of its own and thus is outside the book’s scope. There are many excellent tutorials and references available on the internet and in other books.

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 [165]: import re

In [166]: text = "foo    bar\t baz  \tqux"

In [167]: re.split(r"\s+", text)
Out[167]: ['foo', 'bar', 'baz', 'qux']
```

When you call re.split(r"\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 [168]: regex = re.compile(r"\s+")

In [169]: regex.split(text)
Out[169]: ['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 [170]: regex.findall(text)
Out[170]: ['    ', '\t ', '  \t']
```

> **NOTE**
>
> 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:

```
text = """Dave dave@google.com
Steve steve@gmail.com
Rob rob@gmail.com
Ryan ryan@yahoo.com"""
pattern = r"[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}"

# re.IGNORECASE makes the regex case insensitive
regex = re.compile(pattern, flags=re.IGNORECASE)
```

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

```
In [172]: regex.findall(text)
Out[172]:
['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 [173]: m = regex.search(text)

In [174]: m
Out[174]: <re.Match object; span=(5, 20), match='dave@google.com'>

In [175]: text[m.start():m.end()]
Out[175]: 'dave@google.com'
```

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

```
In [176]: print(regex.match(text))
None
```

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

```
In [177]: 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 [178]: pattern = r"([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\.([A-Z]{2,4})"

In [179]: regex = re.compile(pattern, flags=re.IGNORECASE)
```

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

```
In [180]: m = regex.match("wesm@bright.net")

In [181]: m.groups()
Out[181]: ('wesm', 'bright', 'net')
```

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

```
In [182]: regex.findall(text)
Out[182]: 
[('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 [183]: print(regex.sub(r"Username: \1, Domain: \2, Suffix: \3", text))
Dave Username: dave, Domain: google, Suffix: com
Steve Username: steve, Domain: gmail, Suffix: com
Rob Username: rob, Domain: gmail, Suffix: com
Ryan Username: ryan, Domain: yahoo, Suffix: com
```

There is much more to regular expressions in Python, most of which is outside the book's scope. Table 7-5 provides a brief summary.

Table 7-5. Regular expression methods

Method	Description
findall	Return all nonoverlapping 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, return 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

String Functions in pandas

Cleaning up a messy dataset for analysis often requires a lot of string manipulation. To complicate matters, a column containing strings will sometimes have missing data:

```
In [184]: data = {"Dave": "dave@google.com", "Steve": "steve@gmail.com",
   .....:         "Rob": "rob@gmail.com", "Wes": np.nan}

In [185]: data = pd.Series(data)

In [186]: data
Out[186]: 
Dave     dave@google.com
Steve    steve@gmail.com
Rob        rob@gmail.com
Wes                  NaN
dtype: object

In [187]: data.isna()
```

```
Out[187]:
Dave     False
Steve    False
Rob      False
Wes       True
dtype: bool
```

String and regular expression methods can be applied (passing a lambda or other function) to each value using data.map, but it will fail on the NA (null) values. To cope with this, Series has array-oriented methods for string operations that skip over and propagate 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 [188]: data.str.contains("gmail")
Out[188]:
Dave     False
Steve     True
Rob       True
Wes        NaN
dtype: object
```

Note that the result of this operation has an object dtype. pandas has *extension types* that provide for specialized treatment of strings, integers, and Boolean data which until recently have had some rough edges when working with missing data:

```
In [189]: data_as_string_ext = data.astype('string')

In [190]: data_as_string_ext
Out[190]:
Dave     dave@google.com
Steve    steve@gmail.com
Rob        rob@gmail.com
Wes                 <NA>
dtype: string

In [191]: data_as_string_ext.str.contains("gmail")
Out[191]:
Dave     False
Steve     True
Rob       True
Wes       <NA>
dtype: boolean
```

Extension types are discussed in more detail in Section 7.3, "Extension Data Types,".

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

```
In [192]: pattern = r"([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\.([A-Z]{2,4})"

In [193]: data.str.findall(pattern, flags=re.IGNORECASE)
Out[193]:
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 [194]: matches = data.str.findall(pattern, flags=re.IGNORECASE).str[0]

In [195]: matches
Out[195]:
Dave     (dave, google, com)
Steve    (steve, gmail, com)
Rob        (rob, gmail, com)
Wes                      NaN
dtype: object
```

```
In [196]: matches.str.get(1)
```

```
Out[196]:
Dave     google
Steve     gmail
Rob       gmail
Wes         NaN
dtype: object
```

You can similarly slice strings using this syntax:

```
In [197]: data.str[:5]
Out[197]:
Dave     dave@
Steve    steve
Rob      rob@g
Wes        NaN
dtype: object
```

The str.extract method will return the captured groups of a regular expression as a DataFrame:

```
In [198]: data.str.extract(pattern, flags=re.IGNORECASE)
Out[198]:
           0       1    2
Dave    dave  google  com
Steve  steve   gmail  com
Rob      rob   gmail  com
Wes      NaN     NaN  NaN
```

See Table 7-6 for more pandas string methods.

Table 7-6. Partial listing of Series 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 True or False whether it matches
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

7.5 Categorical Data

This section introduces the pandas Categorical type. I will show how you can achieve better performance and memory use in some pandas operations by using it. I also introduce some tools that may help with using categorical data in statistics and machine learning applications.

Background and Motivation

Frequently, a column in a table may contain repeated instances of a smaller set of distinct values. We have already seen functions like unique and value_counts, which enable us to extract the distinct values from an array and compute their frequencies, respectively:

```
In [199]: values = pd.Series(['apple', 'orange', 'apple',
   .....:                     'apple'] * 2)

In [200]: values
Out[200]:
0    apple
```

```
1    orange
2     apple
3     apple
4     apple
5    orange
6     apple
7     apple
dtype: object

In [201]: pd.unique(values)
Out[201]: array(['apple', 'orange'], dtype=object)

In [202]: pd.value_counts(values)
Out[202]:
apple     6
orange    2
dtype: int64
```

Many data systems (for data warehousing, statistical computing, or other uses) have developed specialized approaches for representing data with repeated values for more efficient storage and computation. In data warehousing, a best practice is to use so-called *dimension tables* containing the distinct values and storing the primary observations as integer keys referencing the dimension table:

```
In [203]: values = pd.Series([0, 1, 0, 0] * 2)

In [204]: dim = pd.Series(['apple', 'orange'])

In [205]: values
Out[205]:
0    0
1    1
2    0
3    0
4    0
5    1
6    0
7    0
dtype: int64

In [206]: dim
Out[206]:
0     apple
1    orange
dtype: object
```

We can use the take method to restore the original Series of strings:

```
In [207]: dim.take(values)
Out[207]:
0     apple
1    orange
0     apple
0     apple
0     apple
1    orange
0     apple
0     apple
dtype: object
```

This representation as integers is called the *categorical* or *dictionary-encoded* representation. The array of distinct values can be called the *categories*, *dictionary*, or *levels* of the data. In this book we will use the terms *categorical* and *categories*. The integer values that reference the categories are called the *category codes* or simply *codes*.

The categorical representation can yield significant performance improvements when you are doing analytics. You can also perform transformations on the categories while leaving the codes unmodified. Some example transformations that can be made at relatively low cost are:

- Renaming categories
- Appending a new category without changing the order or position of the existing categories

Categorical Extension Type in pandas

pandas has a special Categorical extension type for holding data that uses the integer-based categorical representation or *encoding*. This is a popular data compression technique for data with many occurrences of similar values and can provide significantly faster performance with lower memory use, especially for string data.

Let’s consider the example Series from before:

```
In [208]: fruits = ['apple', 'orange', 'apple', 'apple'] * 2

In [209]: N = len(fruits)

In [210]: rng = np.random.default_rng(seed=12345)

In [211]: df = pd.DataFrame({'fruit': fruits,
   .....:                    'basket_id': np.arange(N),
   .....:                    'count': rng.integers(3, 15, size=N),
   .....:                    'weight': rng.uniform(0, 4, size=N)},
   .....:                   columns=['basket_id', 'fruit', 'count', 'weight'])

In [212]: df
Out[212]:
   basket_id   fruit  count    weight
0          0   apple     11  1.564438
1          1  orange      5  1.331256
2          2   apple     12  2.393235
3          3   apple      6  0.746937
4          4   apple      5  2.691024
5          5  orange     12  3.767211
6          6   apple     10  0.992983
7          7   apple     11  3.795525
```

Here, df['fruit'] is an array of Python string objects. We can convert it to categorical by calling:

```
In [213]: fruit_cat = df['fruit'].astype('category')

In [214]: fruit_cat
Out[214]:
0     apple
1    orange
2     apple
3     apple
4     apple
5    orange
6     apple
7     apple
Name: fruit, dtype: category
Categories (2, object): ['apple', 'orange']
```

The values for fruit_cat are now an instance of pandas.Categorical, which you can access via the .array attribute:

```
In [215]: c = fruit_cat.array

In [216]: type(c)
Out[216]: pandas.core.arrays.categorical.Categorical
```

The Categorical object has categories and codes attributes:

```
In [217]: c.categories
Out[217]: Index(['apple', 'orange'], dtype='object')

In [218]: c.codes
Out[218]: array([0, 1, 0, 0, 0, 1, 0, 0], dtype=int8)
```

These can be accessed more easily using the cat accessor, which will be explained soon in “Categorical Methods”.

A useful trick to get a mapping between codes and categories is:

```
In [219]: dict(enumerate(categories))
Out[219]: {0: 'apple', 1: 'orange'}
```

You can convert a DataFrame column to categorical by assigning the converted result:

```
In [220]: df['fruit'] = df['fruit'].astype('category')

In [221]: df["fruit"]
Out[221]:
0     apple
1    orange
2     apple
3     apple
4     apple
5    orange
6     apple
7     apple
Name: fruit, dtype: category
Categories (2, object): ['apple', 'orange']
```

You can also create pandas.Categorical directly from other types of Python sequences:

```
In [222]: my_categories = pd.Categorical(['foo', 'bar', 'baz', 'foo', 'bar'])

In [223]: my_categories
Out[223]:
['foo', 'bar', 'baz', 'foo', 'bar']
Categories (3, object): ['bar', 'baz', 'foo']
```

If you have obtained categorical encoded data from another source, you can use the alternative from_codes constructor:

```
In [224]: categories = ['foo', 'bar', 'baz']

In [225]: codes = [0, 1, 2, 0, 0, 1]

In [226]: my_cats_2 = pd.Categorical.from_codes(codes, categories)

In [227]: my_cats_2
Out[227]:
['foo', 'bar', 'baz', 'foo', 'foo', 'bar']
Categories (3, object): ['foo', 'bar', 'baz']
```

Unless explicitly specified, categorical conversions assume no specific ordering of the categories. So the categories array may be in a different order depending on the ordering of the input data. When using from_codes or any of the other constructors, you can indicate that the categories have a meaningful ordering:

```
In [228]: ordered_cat = pd.Categorical.from_codes(codes, categories,
   .....:                                         ordered=True)

In [229]: ordered_cat
Out[229]:
['foo', 'bar', 'baz', 'foo', 'foo', 'bar']
Categories (3, object): ['foo' < 'bar' < 'baz']
```

The output [foo < bar < baz] indicates that 'foo' precedes 'bar' in the ordering, and so on. An unordered categorical instance can be made ordered with as_ordered:

```
In [230]: my_cats_2.as_ordered()
Out[230]:
['foo', 'bar', 'baz', 'foo', 'foo', 'bar']
Categories (3, object): ['foo' < 'bar' < 'baz']
```

As a last note, categorical data need not be strings, even though I have shown only string examples. A categorical array can consist of any immutable value types.

Computations with Categoricals

Using Categorical in pandas compared with the nonencoded version (like an array of strings) generally behaves the same way. Some parts of pandas, like the groupby function, perform better when working with categoricals. There are also some functions that can utilize the ordered flag.

Let’s consider some random numeric data and use the pandas.qcut binning function. This returns pandas.Categorical; we used pandas.cut earlier in the book but glossed over the details of how categoricals work:

```
In [231]: rng = np.random.default_rng(seed=12345)

In [232]: draws = rng.standard_normal(1000)

In [233]: draws[:5]
Out[233]: array([-1.4238,  1.2637, -0.8707, -0.2592, -0.0753])
```

Let’s compute a quartile binning of this data and extract some statistics:

```
In [234]: bins = pd.qcut(draws, 4)

In [235]: bins
Out[235]:
[(-3.121, -0.675], (0.687, 3.211], (-3.121, -0.675], (-0.675, 0.0134], (-0.675, 0
.0134], ..., (0.0134, 0.687], (0.0134, 0.687], (-0.675, 0.0134], (0.0134, 0.687],
 (-0.675, 0.0134]]
Length: 1000
Categories (4, interval[float64, right]): [(-3.121, -0.675] < (-0.675, 0.0134] <
(0.0134, 0.687] <
                                           (0.687, 3.211]]
```

While useful, the exact sample quartiles may be less useful for producing a report than quartile names. We can achieve this with the labels argument to qcut:

```
In [236]: bins = pd.qcut(draws, 4, labels=['Q1', 'Q2', 'Q3', 'Q4'])

In [237]: bins
Out[237]:
['Q1', 'Q4', 'Q1', 'Q2', 'Q2', ..., 'Q3', 'Q3', 'Q2', 'Q3', 'Q2']
Length: 1000
Categories (4, object): ['Q1' < 'Q2' < 'Q3' < 'Q4']

In [238]: bins.codes[:10]
Out[238]: array([0, 3, 0, 1, 1, 0, 0, 2, 2, 0], dtype=int8)
```

The labeled bins categorical does not contain information about the bin edges in the data, so we can use groupby to extract some summary statistics:

```
In [239]: bins = pd.Series(bins, name='quartile')

In [240]: results = (pd.Series(draws)
   .....:            .groupby(bins)
   .....:            .agg(['count', 'min', 'max'])
   .....:            .reset_index())

In [241]: results
Out[241]:
  quartile  count       min       max
0       Q1    250 -3.119609 -0.678494
1       Q2    250 -0.673305  0.008009
2       Q3    250  0.018753  0.686183
3       Q4    250  0.688282  3.211418
```

The 'quartile' column in the result retains the original categorical information, including ordering, from bins:

```
In [242]: results['quartile']
Out[242]:
0    Q1
1    Q2
```

```
2    Q3
3    Q4
Name: quartile, dtype: category
Categories (4, object): ['Q1' < 'Q2' < 'Q3' < 'Q4']
```

Better performance with categoricals

At the beginning of the section, I said that categorical types can improve performance and memory use, so let’s look at some examples. Consider some Series with 10 million elements and a small number of distinct categories:

```
In [243]: N = 10_000_000

In [244]: labels = pd.Series(['foo', 'bar', 'baz', 'qux'] * (N // 4))
```

Now we convert labels to categorical:

```
In [245]: categories = labels.astype('category')
```

Now we note that labels uses significantly more memory than categories:

```
In [246]: labels.memory_usage(deep=True)
Out[246]: 600000128

In [247]: categories.memory_usage(deep=True)
Out[247]: 10000540
```

The conversion to category is not free, of course, but it is a one-time cost:

```
In [248]: %time _ = labels.astype('category')
CPU times: user 507 ms, sys: 105 ms, total: 612 ms
Wall time: 608 ms
```

GroupBy operations can be significantly faster with categoricals because the underlying algorithms use the integer-based codes array instead of an array of strings. Here we compare the performance of value_counts(), which internally uses the GroupBy machinery:

```
In [249]: %timeit labels.value_counts()
852 ms +- 14.7 ms per loop (mean +- std. dev. of 7 runs, 1 loop each)

In [250]: %timeit categories.value_counts()
29.5 ms +- 465 us per loop (mean +- std. dev. of 7 runs, 10 loops each)
```

Categorical Methods

Series containing categorical data have several special methods similar to the Series.str specialized string methods. This also provides convenient access to the categories and codes. Consider the Series:

```
In [251]: s = pd.Series(['a', 'b', 'c', 'd'] * 2)

In [252]: cat_s = s.astype('category')

In [253]: cat_s
Out[253]:
0    a
1    b
2    c
3    d
4    a
5    b
6    c
7    d
dtype: category
Categories (4, object): ['a', 'b', 'c', 'd']
```

The special *accessor* attribute cat provides access to categorical methods:

```
In [254]: cat_s.cat.codes
Out[254]:
0    0
1    1
2    2
3    3
4    0
5    1
6    2
7    3
dtype: int8

In [255]: cat_s.cat.categories
Out[255]: Index(['a', 'b', 'c', 'd'], dtype='object')
```

Suppose that we know the actual set of categories for this data extends beyond the four values observed in the data. We can use the set_categories method to change them:

```
In [256]: actual_categories = ['a', 'b', 'c', 'd', 'e']

In [257]: cat_s2 = cat_s.cat.set_categories(actual_categories)

In [258]: cat_s2
Out[258]:
0    a
1    b
2    c
3    d
4    a
5    b
6    c
7    d
dtype: category
Categories (5, object): ['a', 'b', 'c', 'd', 'e']
```

While it appears that the data is unchanged, the new categories will be reflected in operations that use them. For example, value_counts respects the categories, if present:

```
In [259]: cat_s.value_counts()
Out[259]:
a    2
b    2
c    2
d    2
dtype: int64

In [260]: cat_s2.value_counts()
Out[260]:
a    2
b    2
c    2
d    2
e    0
dtype: int64
```

In large datasets, categoricals are often used as a convenient tool for memory savings and better performance. After you filter a large DataFrame or Series, many of the categories may not appear in the data. To help with this, we can use the remove_unused_categories method to trim unobserved categories:

```
In [261]: cat_s3 = cat_s[cat_s.isin(['a', 'b'])]

In [262]: cat_s3
Out[262]:
0    a
1    b
4    a
5    b
dtype: category
```

```
Categories (4, object): ['a', 'b', 'c', 'd']

In [263]: cat_s3.cat.remove_unused_categories()
Out[263]:
0    a
1    b
4    a
5    b
dtype: category
Categories (2, object): ['a', 'b']
```

See Table 7-7 for a listing of available categorical methods.

Table 7-7. Categorical methods for Series in pandas

Method	Description
add_categories	Append new (unused) categories at end of existing categories
as_ordered	Make categories ordered
as_unordered	Make categories unordered
remove_categories	Remove categories, setting any removed values to null
remove_unused_categories	Remove any category values that do not appear in the data
rename_categories	Replace categories with indicated set of new category names; cannot change the number of categories
reorder_categories	Behaves like rename_categories, but can also change the result to have ordered categories
set_categories	Replace the categories with the indicated set of new categories; can add or remove categories

Creating dummy variables for modeling

When you're using statistics or machine learning tools, you'll often transform categorical data into *dummy variables*, also known as *one-hot* encoding. This involves creating a DataFrame with a column for each distinct category; these columns contain 1s for occurrences of a given category and 0 otherwise.

Consider the previous example:

```
In [264]: cat_s = pd.Series(['a', 'b', 'c', 'd'] * 2, dtype='category')
```

As mentioned previously in this chapter, the pandas.get_dummies function converts this one-dimensional categorical data into a DataFrame containing the dummy variable:

```
In [265]: pd.get_dummies(cat_s)
Out[265]:
   a  b  c  d
0  1  0  0  0
1  0  1  0  0
2  0  0  1  0
3  0  0  0  1
4  1  0  0  0
5  0  1  0  0
6  0  0  1  0
7  0  0  0  1
```

7.6 Conclusion

Effective data preparation can significantly improve productivity by enabling you to spend more time analyzing data and less time getting it ready for analysis. We have explored a number of tools in this chapter, but the coverage here is by no means comprehensive. In the next chapter, we will explore pandas's joining and grouping functionality.