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 has 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 functional for a lot of users. For numeric data, pandas uses the floating-point value `NaN` (Not a Number) to represent missing data. We call this a *sentinel value* that can be easily detected:

```
In [12]: string_data = pd.Series(['aardvark', 'artichoke', np.nan, 'avocado'])

In [13]: string_data
Out[13]:
0     aardvark
1    artichoke
2          NaN
3      avocado
dtype: object

In [14]: string_data.isnull()
Out[14]:
0    False
1    False
2     True
3    False
dtype: bool
```

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

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

```
In [15]: string_data[0] = None

In [16]: string_data.isnull()
Out[16]:
0     True
1    False
2     True
3    False
dtype: bool
```

There is work ongoing in the pandas project to improve the internal details of how missing data is handled, but the user API functions, like `pandas.isnull`, abstract away many of the annoying details. See Table 7-1 for a list of some functions related to missing data handling.

Table 7-1. NA handling methods

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

Filtering Out Missing Data

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

```
In [17]: from numpy import nan as NA

In [18]: data = pd.Series([1, NA, 3.5, NA, 7])

In [19]: data.dropna()
Out[19]:
0    1.0
2    3.5
4    7.0
dtype: float64
```

This is equivalent to:

```
In [20]: data[data.notnull()]
Out[20]:
0    1.0
2    3.5
4    7.0
dtype: float64
```

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

```
In [21]: data = pd.DataFrame([[1., 6.5, 3.], [1., NA, NA],
   ....:                      [NA, NA, NA], [NA, 6.5, 3.]])

In [22]: cleaned = data.dropna()

In [23]: data
Out[23]:
     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 [24]: cleaned
Out[24]:
     0    1    2
0  1.0  6.5  3.0
```

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

```
In [25]: data.dropna(how='all')
Out[25]:
     0    1    2
```

```
0  1.0  6.5  3.0
1  1.0  NaN  NaN
3  NaN  6.5  3.0
```

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

```
In [26]: data[4] = NA

In [27]: data
Out[27]:
     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 [28]: data.dropna(axis=1, how='all')
Out[28]:
     0    1    2
0  1.0  6.5  3.0
1  1.0  NaN  NaN
2  NaN  NaN  NaN
3  NaN  6.5  3.0
```

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

```
In [29]: df = pd.DataFrame(np.random.randn(7, 3))

In [30]: df.iloc[:4, 1] = NA

In [31]: df.iloc[:2, 2] = NA

In [32]: df
Out[32]:
          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 [33]: df.dropna()
Out[33]:
          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 [34]: df.dropna(thresh=2)
```

```
Out[34]:
          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 [35]: df.fillna(0)
Out[35]:
          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 dict, you can use a different fill value for each column:

```
In [36]: df.fillna({1: 0.5, 2: 0})
Out[36]:
          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
```

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

```
In [37]: _ = df.fillna(0, inplace=True)

In [38]: df
Out[38]:
          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
```

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

```
In [39]: df = pd.DataFrame(np.random.randn(6, 3))

In [40]: df.iloc[2:, 1] = NA

In [41]: df.iloc[4:, 2] = NA

In [42]: df
Out[42]:
          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 [43]: df.fillna(method='ffill')
Out[43]:
          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 [44]: df.fillna(method='ffill', limit=2)
Out[44]:
          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 with a little creativity. For example, you might pass the mean or median value of a Series:

```
In [45]: data = pd.Series([1., NA, 3.5, NA, 7])

In [46]: data.fillna(data.mean())
Out[46]:
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`.

Table 7-2. fillna function arguments

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

7.2 Data Transformation

So far in this chapter we've been concerned with rearranging 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 [47]: data = pd.DataFrame({'k1': ['one', 'two'] * 3 + ['two'],
   ....:                      'k2': [1, 1, 2, 3, 3, 4, 4]})

In [48]: data
Out[48]:
    k1  k2
0  one   1
1  two   1
2  one   2
3  two   3
4  one   3
5  two   4
6  two   4
```

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

```
In [49]: data.duplicated()
Out[49]:
0    False
1    False
2    False
3    False
4    False
5    False
6     True
dtype: bool
```

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

```
In [50]: data.drop_duplicates()
Out[50]:
    k1  k2
0  one   1
1  two   1
2  one   2
3  two   3
4  one   3
5  two   4
```

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

```
In [51]: data['v1'] = range(7)

In [52]: data.drop_duplicates(['k1'])
Out[52]:
    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 [53]: data.drop_duplicates(['k1', 'k2'], keep='last')
Out[53]:
    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 [54]: 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 [55]: data
Out[55]:
          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 accepts a function or dict-like object containing a mapping, but here we have a small problem in that some of the meats are capitalized and others are not. Thus, we need to convert each value to lowercase using the `str.lower` Series method:

```
In [57]: lowercased = data['food'].str.lower()

In [58]: lowercased
Out[58]:
0          bacon
1    pulled pork
2          bacon
3       pastrami
4    corned beef
5          bacon
6       pastrami
7      honey ham
8       nova lox
Name: food, dtype: object

In [59]: data['animal'] = lowercased.map(meat_to_animal)

In [60]: data
Out[60]:
          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 [61]: data['food'].map(lambda x: meat_to_animal[x.lower()])
Out[61]:
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 [62]: data = pd.Series([1., -999., 2., -999., -1000., 3.])

In [63]: data
Out[63]:
0       1.0
1    -999.0
2       2.0
3    -999.0
4   -1000.0
5       3.0
dtype: float64
```

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

```
In [64]: data.replace(-999, np.nan)
Out[64]:
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 [65]: data.replace([-999, -1000], np.nan)
Out[65]:
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 [66]: data.replace([-999, -1000], [np.nan, 0])
Out[66]:
0    1.0
1    NaN
2    2.0
3    NaN
4    0.0
5    3.0
dtype: float64
```

The argument passed can also be a dict:

```
In [67]: data.replace({-999: np.nan, -1000: 0})
Out[67]:
0    1.0
1    NaN
2    2.0
3    NaN
4    0.0
5    3.0
dtype: float64
```

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

Renaming Axis Indexes

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

```
In [68]: 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 [69]: transform = lambda x: x[:4].upper()

In [70]: data.index.map(transform)
Out[70]: Index(['OHIO', 'COLO', 'NEW '], dtype='object')
```

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

```
In [71]: data.index = data.index.map(transform)

In [72]: data
Out[72]:
      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 [73]: data.rename(index=str.title, columns=str.upper)
Out[73]:
      ONE  TWO  THREE  FOUR
Ohio    0    1      2     3
Colo    4    5      6     7
New     8    9     10    11
```

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

```
In [74]: data.rename(index={'OHIO': 'INDIANA'},
   ....:             columns={'three': 'peekaboo'})
Out[74]:
         one  two  peekaboo  four
INDIANA    0    1         2     3
COLO       4    5         6     7
NEW        8    9        10    11
```

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

```
In [75]: data.rename(index={'OHIO': 'INDIANA'}, inplace=True)

In [76]: data
Out[76]:
         one  two  three  four
INDIANA    0    1      2     3
```

```
COLO       4    5      6     7
NEW        8    9     10    11
```

Discretization and Binning

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

```
In [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 `cut`, a function in pandas:

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

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

In [80]: cats
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]): [(18, 25] < (25, 35] < (35, 60] < (60, 100]]
```

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

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

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

In [83]: pd.value_counts(cats)
Out[83]:
(18, 25]     5
(35, 60]     3
(25, 35]     3
(60, 100]    1
dtype: int64
```

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

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

```
In [84]: pd.cut(ages, [18, 26, 36, 61, 100], right=False)
Out[84]:
[[18, 26), [18, 26), [18, 26), [26, 36), [18, 26), ..., [26, 36), [61, 100), [36,
 61), [36, 61), [26, 36)]
Length: 12
Categories (4, interval[int64]): [[18, 26) < [26, 36) < [36, 61) < [61, 100)]
```

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

```
In [85]: group_names = ['Youth', 'YoungAdult', 'MiddleAged', 'Senior']

In [86]: pd.cut(ages, bins, labels=group_names)
Out[86]:
[Youth, Youth, Youth, YoungAdult, Youth, ..., YoungAdult, Senior, MiddleAged, Mid
dleAged, YoungAdult]
Length: 12
Categories (4, object): [Youth < YoungAdult < MiddleAged < Senior]
```

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

```
In [87]: data = np.random.rand(20)

In [88]: pd.cut(data, 4, precision=2)
Out[88]:
[(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]): [(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, `qcut`, bins the data based on sample quantiles. Depending on the distribution of the data, using `cut` will not usually result in each bin having the same number of data points. Since `qcut` uses sample quantiles instead, by definition you will obtain roughly equal-size bins:

```
In [89]: data = np.random.randn(1000)  # Normally distributed

In [90]: cats = pd.qcut(data, 4)  # Cut into quartiles

In [91]: cats
Out[91]:
[(-0.0265, 0.62], (0.62, 3.928], (-0.68, -0.0265], (0.62, 3.928], (-0.0265, 0.62]
, ..., (-0.68, -0.0265], (-0.68, -0.0265], (-2.9499999999999997, -0.68], (0.62, 3
.928], (-0.68, -0.0265]]
Length: 1000
Categories (4, interval[float64]): [(-2.9499999999999997, -0.68] < (-0.68, -0.026
5] < (-0.0265, 0.62] <
                                    (0.62, 3.928]]
```

```
In [92]: pd.value_counts(cats)
Out[92]:
(0.62, 3.928]                   250
(-0.0265, 0.62]                 250
(-0.68, -0.0265]                250
(-2.9499999999999997, -0.68]    250
dtype: int64
```

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

```
In [93]: pd.qcut(data, [0, 0.1, 0.5, 0.9, 1.])
Out[93]:
[(-0.0265, 1.286], (-0.0265, 1.286], (-1.187, -0.0265], (-0.0265, 1.286], (-0.026
5, 1.286], ..., (-1.187, -0.0265], (-1.187, -0.0265], (-2.9499999999999997, -1.18
7], (-0.0265, 1.286], (-1.187, -0.0265]]
Length: 1000
Categories (4, interval[float64]): [(-2.9499999999999997, -1.187] < (-1.187, -0.0
265] < (-0.0265, 1.286] <
                                    (1.286, 3.928]]
```

We'll return to `cut` and `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 [94]: data = pd.DataFrame(np.random.randn(1000, 4))

In [95]: data.describe()
Out[95]:
                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 [96]: col = data[2]

In [97]: col[np.abs(col) > 3]
Out[97]:
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 [98]: data[(np.abs(data) > 3).any(1)]
Out[98]:
            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
```

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

```
In [99]: data[np.abs(data) > 3] = np.sign(data) * 3

In [100]: data.describe()
Out[100]:
                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 [101]: np.sign(data).head()
Out[101]:
     0    1    2    3
0 -1.0  1.0 -1.0  1.0
1  1.0 -1.0  1.0 -1.0
2  1.0  1.0  1.0 -1.0
3 -1.0 -1.0  1.0 -1.0
4 -1.0  1.0 -1.0 -1.0
```

Permutation and Random Sampling

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

```
In [102]: df = pd.DataFrame(np.arange(5 * 4).reshape((5, 4)))

In [103]: sampler = np.random.permutation(5)

In [104]: sampler
Out[104]: array([3, 1, 4, 2, 0])
```

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

```
In [105]: df
Out[105]:
    0   1   2   3
0   0   1   2   3
1   4   5   6   7
2   8   9  10  11
3  12  13  14  15
4  16  17  18  19

In [106]: df.take(sampler)
Out[106]:
    0   1   2   3
3  12  13  14  15
1   4   5   6   7
4  16  17  18  19
2   8   9  10  11
0   0   1   2   3
```

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

```
In [107]: df.sample(n=3)
Out[107]:
    0   1   2   3
3  12  13  14  15
4  16  17  18  19
2   8   9  10  11
```

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

```
In [108]: choices = pd.Series([5, 7, -1, 6, 4])

In [109]: draws = choices.sample(n=10, replace=True)

In [110]: draws
Out[110]:
4    4
1    7
4    4
2   -1
0    5
3    6
1    7
```

```
4    4
0    5
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 `get_dummies` function for doing this, though devising one yourself is not difficult. Let's consider an example DataFrame:

```
In [111]: df = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'b'],
   .....:                    'data1': range(6)})

In [112]: pd.get_dummies(df['key'])
Out[112]:
   a  b  c
0  0  1  0
1  0  1  0
2  1  0  0
3  0  0  1
4  1  0  0
5  0  1  0
```

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

```
In [113]: dummies = pd.get_dummies(df['key'], prefix='key')

In [114]: df_with_dummy = df[['data1']].join(dummies)

In [115]: df_with_dummy
Out[115]:
   data1  key_a  key_b  key_c
0      0      0      1      0
1      1      0      1      0
2      2      1      0      0
3      3      0      0      1
4      4      1      0      0
5      5      0      1      0
```

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

```
In [116]: mnames = ['movie_id', 'title', 'genres']

In [117]: movies = pd.read_table('datasets/movielens/movies.dat', sep='::',
   .....:                        header=None, names=mnames)

In [118]: movies[:10]
Out[118]:
   movie_id                               title                        genres
0         1                    Toy Story (1995)   Animation|Children's|Comedy
1         2                      Jumanji (1995)  Adventure|Children's|Fantasy
2         3             Grumpier Old Men (1995)                Comedy|Romance
3         4            Waiting to Exhale (1995)                  Comedy|Drama
4         5  Father of the Bride Part II (1995)                        Comedy
5         6                         Heat (1995)         Action|Crime|Thriller
6         7                      Sabrina (1995)                Comedy|Romance
7         8             Tom and Huck (1995)          Adventure|Children's
8         9             Sudden Death (1995)                        Action
9        10                    GoldenEye (1995)     Action|Adventure|Thriller
```

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

```
In [119]: all_genres = []

In [120]: for x in movies.genres:
   .....:     all_genres.extend(x.split('|'))

In [121]: genres = pd.unique(all_genres)
```

Now we have:

```
In [122]: genres
Out[122]:
array(['Animation', "Children's", 'Comedy', 'Adventure', 'Fantasy',
       'Romance', 'Drama', 'Action', 'Crime', 'Thriller', 'Horror',
       'Sci-Fi', 'Documentary', 'War', 'Musical', 'Mystery', 'Film-Noir',
       'Western'], dtype=object)
```

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

```
In [123]: zero_matrix = np.zeros((len(movies), len(genres)))

In [124]: dummies = pd.DataFrame(zero_matrix, columns=genres)
```

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

```
In [125]: gen = movies.genres[0]

In [126]: gen.split('|')
Out[126]: ['Animation', "Children's", 'Comedy']

In [127]: dummies.columns.get_indexer(gen.split('|'))
Out[127]: array([0, 1, 2])
```

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

```
In [128]: for i, gen in enumerate(movies.genres):
   .....:     indices = dummies.columns.get_indexer(gen.split('|'))
   .....:     dummies.iloc[i, indices] = 1
   .....:
```

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

```
In [129]: movies_windic = movies.join(dummies.add_prefix('Genre_'))

In [130]: movies_windic.iloc[0]
Out[130]:
movie_id                                    1
title                           Toy Story (1995)
genres               Animation|Children's|Comedy
Genre_Animation                             1
Genre_Children's                            1
Genre_Comedy                                1
Genre_Adventure                             0
Genre_Fantasy                               0
Genre_Romance                               0
Genre_Drama                                 0
                                  ...
Genre_Crime                                 0
Genre_Thriller                              0
Genre_Horror                                0
Genre_Sci-Fi                                0
Genre_Documentary                           0
Genre_War                                   0
Genre_Musical                               0
Genre_Mystery                               0
Genre_Film-Noir                             0
Genre_Western                               0
Name: 0, Length: 21, 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 `get_dummies` with a discretization function like `cut`:

```
In [131]: np.random.seed(12345)

In [132]: values = np.random.rand(10)

In [133]: values
Out[133]:
array([0.9296, 0.3164, 0.1839, 0.2046, 0.5677, 0.5955, 0.9645, 0.6532,
       0.7489, 0.6536])

In [134]: bins = [0, 0.2, 0.4, 0.6, 0.8, 1]

In [135]: pd.get_dummies(pd.cut(values, bins))
Out[135]:
   (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 set the random seed with `numpy.random.seed` to make the example deterministic. We will look again at `pandas.get_dummies` later in the book.

7.3 String Manipulation

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

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 [136]: val = 'a,b,  guido'

In [137]: val.split(',')
Out[137]: ['a', 'b', '  guido']
```

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

```
In [138]: pieces = [x.strip() for x in val.split(',')]

In [139]: pieces
Out[139]: ['a', 'b', 'guido']
```

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

```
In [140]: first, second, third = pieces

In [141]: first + '::' + second + '::' + third
Out[141]: '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 [142]: '::'.join(pieces)
Out[142]: '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 [143]: 'guido' in val
Out[143]: True

In [144]: val.index(',')
Out[144]: 1

In [145]: val.find(':')
Out[145]: -1
```

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

```
In [146]: val.index(':')
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-146-280f8b2856ce> in <module>()
----> 1 val.index(':')
ValueError: substring not found
```

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

```
In [147]: val.count(',')
Out[147]: 2
```

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

```
In [148]: val.replace(',', '::')
Out[148]: 'a::b::  guido'

In [149]: val.replace(',', '')
Out[149]: 'ab  guido'
```

See Table 7-3 for a listing of some of Python's string methods.

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

Table 7-3. Python built-in string methods

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

Regular Expressions

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

The 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 [150]: import re

In [151]: text = "foo    bar\t baz  \tqux"

In [152]: re.split('\s+', text)
Out[152]: ['foo', 'bar', 'baz', 'qux']
```

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

```
In [153]: regex = re.compile('\s+')

In [154]: regex.split(text)
Out[154]: ['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 [155]: regex.findall(text)
Out[155]: ['    ', '\t ', '  \t']
```

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

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

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

```
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 [157]: regex.findall(text)
Out[157]:
['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 [158]: m = regex.search(text)

In [159]: m
Out[159]: <_sre.SRE_Match object; span=(5, 20), match='dave@google.com'>

In [160]: text[m.start():m.end()]
Out[160]: 'dave@google.com'
```

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

```
In [161]: print(regex.match(text))
None
```

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

```
In [162]: 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 [163]: pattern = r'([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\.([A-Z]{2,4})'

In [164]: 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 [165]: m = regex.match('wesm@bright.net')

In [166]: m.groups()
Out[166]: ('wesm', 'bright', 'net')
```

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

```
In [167]: regex.findall(text)
Out[167]:
```

```
[('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 [168]: 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 7-4 provides a brief summary.

Table 7-4. Regular expression methods

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

Vectorized String Functions in pandas

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

```
In [169]: data = {'Dave': 'dave@google.com', 'Steve': 'steve@gmail.com',
   .....:         'Rob': 'rob@gmail.com', 'Wes': np.nan}

In [170]: data = pd.Series(data)

In [171]: data
Out[171]:
Dave     dave@google.com
Steve    steve@gmail.com
Rob        rob@gmail.com
Wes                  NaN
dtype: object
```

```
In [172]: data.isnull()
Out[172]:
Dave     False
Steve    False
Rob      False
Wes       True
dtype: bool
```

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

```
In [173]: data.str.contains('gmail')
Out[173]:
Dave     False
Steve     True
Rob       True
Wes        NaN
dtype: object
```

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

```
In [174]: pattern = r'([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\.([A-Z]{2,4})'

In [175]: data.str.findall(pattern, flags=re.IGNORECASE)
Out[175]:
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 [176]: matches = data.str.findall(pattern, flags=re.IGNORECASE).str[0]

In [177]: matches
Out[177]:
Dave     (dave, google, com)
Steve    (steve, gmail, com)
Rob        (rob, gmail, com)
Wes                      NaN
dtype: object

In [178]: matches.str.get(1)
Out[178]:
Dave     google
Steve     gmail
Rob       gmail
```

```
Wes         NaN
dtype: object
```

You can similarly slice strings using this syntax:

```
In [179]: data.str[:5]
Out[179]:
Dave     dave@
Steve    steve
Rob      rob@g
Wes        NaN
dtype: object
```

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

```
In [180]: data.str.extract(pattern, flags=re.IGNORECASE)
Out[180]:
           0       1    2
Dave    dave  google  com
Steve  steve   gmail  com
Rob      rob   gmail  com
Wes      NaN     NaN  NaN
```

See Table 7-5 for more pandas string methods.

Table 7-5. Partial listing of vectorized string methods

Method	Description
`cat`	Concatenate strings element-wise with optional delimiter
`contains`	Return boolean array if each string contains pattern/regex
`count`	Count occurrences of pattern
`extract`	Use a regular expression with groups to extract one or more strings from a Series of strings; the result will be a DataFrame with one column per group
`endswith`	Equivalent to `x.endswith(pattern)` for each element
`startswith`	Equivalent to `x.startswith(pattern)` for each element
`findall`	Compute list of all occurrences of pattern/regex for each string
`get`	Index into each element (retrieve *i*-th element)
`isalnum`	Equivalent to built-in `str.alnum`
`isalpha`	Equivalent to built-in `str.isalpha`
`isdecimal`	Equivalent to built-in `str.isdecimal`
`isdigit`	Equivalent to built-in `str.isdigit`
`islower`	Equivalent to built-in `str.islower`
`isnumeric`	Equivalent to built-in `str.isnumeric`
`isupper`	Equivalent to built-in `str.isupper`
`join`	Join strings in each element of the Series with passed separator
`len`	Compute length of each string
`lower, upper`	Convert cases; equivalent to `x.lower()` or `x.upper()` for each element

Method	Description
`match`	Use `re.match` with the passed regular expression on each element, returning `True` or `False` whether it matches.
`extract`	Extract captured group element (if any) by index from each string
`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

7.4 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.