CHAPTER 5

Getting Started with pandas

pandas will be a major tool of interest throughout much of the rest of the book. It contains data structures and data manipulation tools designed to make data cleaning and analysis fast and convenient in Python. pandas is often used in tandem with numerical computing tools like NumPy and SciPy, analytical libraries like statsmodels and scikit-learn, and data visualization libraries like matplotlib. pandas adopts significant parts of NumPy's idiomatic style of array-based computing, especially array-based functions and a preference for data processing without `for` loops.

While pandas adopts many coding idioms from NumPy, the biggestabout difference is that pandas is designed for working with tabular or heterogeneous data. NumPy, by contrast, is best suited for working with homogeneously typed numerical array data.

Since becoming an open source project in 2010, pandas has matured into a quite large library that's applicable in a broad set of real-world use cases. The developer community has grown to over 2,500 distinct contributors, who've been helping build the project as they used it to solve their day-to-day data problems. The vibrant pandas developer and user communities have been a key part of its success.

> Many people don't know that I haven't been actively involved in day-to-day pandas development since 2013; it has been an entirely community-managed project since then. Be sure to pass on your thanks to the core development and all the contributors for their hard work!

Throughout the rest of the book, I use the following import conventions for NumPy and pandas:

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

In [2]: import pandas as pd
```

Thus, whenever you see `pd.` in code, it's referring to pandas. You may also find it easier to import Series and DataFrame into the local namespace since they are so frequently used:

```
In [3]: from pandas import Series, DataFrame
```

5.1 Introduction to pandas Data Structures

To get started with pandas, you will need to get comfortable with its two workhorse data structures: *Series* and *DataFrame*. While they are not a universal solution for every problem, they provide a solid foundation for a wide variety of data tasks.

Series

A Series is a one-dimensional array-like object containing a sequence of values (of similar types to NumPy types) of the same type and an associated array of data labels, called its *index*. The simplest Series is formed from only an array of data:

```
In [14]: obj = pd.Series([4, 7, -5, 3])

In [15]: obj
Out[15]:
0    4
1    7
2   -5
3    3
dtype: int64
```

The string representation of a Series displayed interactively shows the index on the left and the values on the right. Since we did not specify an index for the data, a default one consisting of the integers `0` through `N - 1` (where `N` is the length of the data) is created. You can get the array representation and index object of the Series via its `array` and `index` attributes, respectively:

```
In [16]: obj.array
Out[16]:
<PandasArray>
[4, 7, -5, 3]
Length: 4, dtype: int64

In [17]: obj.index
Out[17]: RangeIndex(start=0, stop=4, step=1)
```

The result of the `.array` attribute is a `PandasArray` which usually wraps a NumPy array but can also contain special extension array types which will be discussed more in Section 7.3, "Extension Data Types," on page 224.

Often, you'll want to create a Series with an index identifying each data point with a label:

```
In [18]: obj2 = pd.Series([4, 7, -5, 3], index=["d", "b", "a", "c"])

In [19]: obj2
Out[19]:
d    4
b    7
a   -5
c    3
dtype: int64

In [20]: obj2.index
Out[20]: Index(['d', 'b', 'a', 'c'], dtype='object')
```

Compared with NumPy arrays, you can use labels in the index when selecting single values or a set of values:

```
In [21]: obj2["a"]
Out[21]: -5

In [22]: obj2["d"] = 6

In [23]: obj2[["c", "a", "d"]]
Out[23]:
c    3
a   -5
d    6
dtype: int64
```

Here `["c", "a", "d"]` is interpreted as a list of indices, even though it contains strings instead of integers.

Using NumPy functions or NumPy-like operations, such as filtering with a Boolean array, scalar multiplication, or applying math functions, will preserve the index-value link:

```
In [24]: obj2[obj2 > 0]
Out[24]:
d    6
b    7
c    3
dtype: int64

In [25]: obj2 * 2
Out[25]:
d    12
```

```
b    14
a   -10
c     6
dtype: int64

In [26]: import numpy as np

In [27]: np.exp(obj2)
Out[27]:
d     403.428793
b    1096.633158
a       0.006738
c      20.085537
dtype: float64
```

Another way to think about a Series is as a fixed-length, ordered dictionary, as it is a mapping of index values to data values. It can be used in many contexts where you might use a dictionary:

```
In [28]: "b" in obj2
Out[28]: True

In [29]: "e" in obj2
Out[29]: False
```

Should you have data contained in a Python dictionary, you can create a Series from it by passing the dictionary:

```
In [30]: sdata = {"Ohio": 35000, "Texas": 71000, "Oregon": 16000, "Utah": 5000}

In [31]: obj3 = pd.Series(sdata)

In [32]: obj3
Out[32]:
Ohio      35000
Texas     71000
Oregon    16000
Utah       5000
dtype: int64
```

A Series can be converted back to a dictionary with its `to_dict` method:

```
In [33]: obj3.to_dict()
Out[33]: {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}
```

When you are only passing a dictionary, the index in the resulting Series will respect the order of the keys according to the dictionary's `keys` method, which depends on the key insertion order. You can override this by passing an index with the dictionary keys in the order you want them to appear in the resulting Series:

```
In [34]: states = ["California", "Ohio", "Oregon", "Texas"]

In [35]: obj4 = pd.Series(sdata, index=states)
```

```
In [36]: obj4
Out[36]:
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
dtype: float64
```

Here, three values found in `sdata` were placed in the appropriate locations, but since no value for `"California"` was found, it appears as `NaN` (Not a Number), which is considered in pandas to mark missing or *NA* values. Since `"Utah"` was not included in `states`, it is excluded from the resulting object.

I will use the terms "missing," "NA," or "null" interchangeably to refer to missing data. The `isna` and `notna` functions in pandas should be used to detect missing data:

```
In [37]: pd.isna(obj4)
Out[37]:
California     True
Ohio          False
Oregon        False
Texas         False
dtype: bool

In [38]: pd.notna(obj4)
Out[38]:
California    False
Ohio           True
Oregon         True
Texas          True
dtype: bool
```

Series also has these as instance methods:

```
In [39]: obj4.isna()
Out[39]:
California     True
Ohio          False
Oregon        False
Texas         False
dtype: bool
```

I discuss working with missing data in more detail in Chapter 7.

A useful Series feature for many applications is that it automatically aligns by index label in arithmetic operations:

```
In [40]: obj3
Out[40]:
Ohio      35000
Texas     71000
Oregon    16000
```

```
Utah          5000
dtype: int64

In [41]: obj4
Out[41]:
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
dtype: float64

In [42]: obj3 + obj4
Out[42]:
California         NaN
Ohio           70000.0
Oregon         32000.0
Texas         142000.0
Utah               NaN
dtype: float64
```

Data alignment features will be addressed in more detail later. If you have experience with databases, you can think about this as being similar to a join operation.

Both the Series object itself and its index have a `name` attribute, which integrates with other areas of pandas functionality:

```
In [43]: obj4.name = "population"

In [44]: obj4.index.name = "state"

In [45]: obj4
Out[45]:
state
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
Name: population, dtype: float64
```

A Series's index can be altered in place by assignment:

```
In [46]: obj
Out[46]:
0    4
1    7
2   -5
3    3
dtype: int64

In [47]: obj.index = ["Bob", "Steve", "Jeff", "Ryan"]

In [48]: obj
Out[48]:
```

```
Bob      4
Steve    7
Jeff    -5
Ryan     3
dtype: int64
```

DataFrame

A DataFrame represents a rectangular table of data and contains an ordered, named collection of columns, each of which can be a different value type (numeric, string, Boolean, etc.). The DataFrame has both a row and column index; it can be thought of as a dictionary of Series all sharing the same index.

> While a DataFrame is physically two-dimensional, you can use it to represent higher dimensional data in a tabular format using hierarchical indexing, a subject we will discuss in Chapter 8 and an ingredient in some of the more advanced data-handling features in pandas.

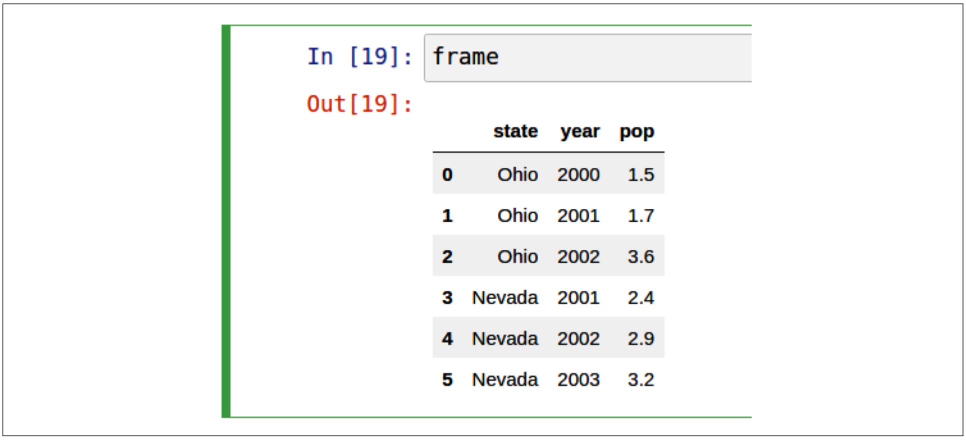
There are many ways to construct a DataFrame, though one of the most common is from a dictionary of equal-length lists or NumPy arrays:

```
data = {"state": ["Ohio", "Ohio", "Ohio", "Nevada", "Nevada", "Nevada"],
        "year": [2000, 2001, 2002, 2001, 2002, 2003],
        "pop": [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]}
frame = pd.DataFrame(data)
```

The resulting DataFrame will have its index assigned automatically, as with Series, and the columns are placed according to the order of the keys in `data` (which depends on their insertion order in the dictionary):

```
In [50]: frame
Out[50]:
    state  year  pop
0    Ohio  2000  1.5
1    Ohio  2001  1.7
2    Ohio  2002  3.6
3  Nevada  2001  2.4
4  Nevada  2002  2.9
5  Nevada  2003  3.2
```

> If you are using the Jupyter notebook, pandas DataFrame objects will be displayed as a more browser-friendly HTML table. See Figure 5-1 for an example.



	state	year	pop
0	Ohio	2000	1.5
1	Ohio	2001	1.7
2	Ohio	2002	3.6
3	Nevada	2001	2.4
4	Nevada	2002	2.9
5	Nevada	2003	3.2

Figure 5-1. How pandas DataFrame objects look in Jupyter

For large DataFrames, the `head` method selects only the first five rows:

```
In [51]: frame.head()
Out[51]:
    state  year  pop
0    Ohio  2000  1.5
1    Ohio  2001  1.7
2    Ohio  2002  3.6
3  Nevada  2001  2.4
4  Nevada  2002  2.9
```

Similarly, `tail` returns the last five rows:

```
In [52]: frame.tail()
Out[52]:
    state  year  pop
1    Ohio  2001  1.7
2    Ohio  2002  3.6
3  Nevada  2001  2.4
4  Nevada  2002  2.9
5  Nevada  2003  3.2
```

If you specify a sequence of columns, the DataFrame's columns will be arranged in that order:

```
In [53]: pd.DataFrame(data, columns=["year", "state", "pop"])
Out[53]:
   year   state  pop
0  2000    Ohio  1.5
1  2001    Ohio  1.7
2  2002    Ohio  3.6
3  2001  Nevada  2.4
4  2002  Nevada  2.9
5  2003  Nevada  3.2
```

If you pass a column that isn't contained in the dictionary, it will appear with missing values in the result:

```
In [54]: frame2 = pd.DataFrame(data, columns=["year", "state", "pop", "debt"])

In [55]: frame2
Out[55]:
   year   state  pop debt
0  2000    Ohio  1.5  NaN
1  2001    Ohio  1.7  NaN
2  2002    Ohio  3.6  NaN
3  2001  Nevada  2.4  NaN
4  2002  Nevada  2.9  NaN
5  2003  Nevada  3.2  NaN

In [56]: frame2.columns
Out[56]: Index(['year', 'state', 'pop', 'debt'], dtype='object')
```

A column in a DataFrame can be retrieved as a Series either by dictionary-like notation or by using the dot attribute notation:

```
In [57]: frame2["state"]
Out[57]:
0      Ohio
1      Ohio
2      Ohio
3    Nevada
4    Nevada
5    Nevada
Name: state, dtype: object

In [58]: frame2.year
Out[58]:
0    2000
1    2001
2    2002
3    2001
4    2002
5    2003
Name: year, dtype: int64
```

Attribute-like access (e.g., `frame2.year`) and tab completion of column names in IPython are provided as a convenience.

`frame2[column]` works for any column name, but `frame2.column` works only when the column name is a valid Python variable name and does not conflict with any of the method names in DataFrame. For example, if a column's name contains whitespace or symbols other than underscores, it cannot be accessed with the dot attribute method.

Note that the returned Series have the same index as the DataFrame, and their `name` attribute has been appropriately set.

Rows can also be retrieved by position or name with the special `iloc` and `loc` attributes (more on this later in "Selection on DataFrame with loc and iloc" on page 147):

```
In [59]: frame2.loc[1]
Out[59]:
year     2001
state    Ohio
pop       1.7
debt      NaN
Name: 1, dtype: object

In [60]: frame2.iloc[2]
Out[60]:
year     2002
state    Ohio
pop       3.6
debt      NaN
Name: 2, dtype: object
```

Columns can be modified by assignment. For example, the empty `debt` column could be assigned a scalar value or an array of values:

```
In [61]: frame2["debt"] = 16.5

In [62]: frame2
Out[62]:
   year   state  pop  debt
0  2000    Ohio  1.5  16.5
1  2001    Ohio  1.7  16.5
2  2002    Ohio  3.6  16.5
3  2001  Nevada  2.4  16.5
4  2002  Nevada  2.9  16.5
5  2003  Nevada  3.2  16.5

In [63]: frame2["debt"] = np.arange(6.)

In [64]: frame2
Out[64]:
   year   state  pop  debt
0  2000    Ohio  1.5   0.0
1  2001    Ohio  1.7   1.0
2  2002    Ohio  3.6   2.0
3  2001  Nevada  2.4   3.0
4  2002  Nevada  2.9   4.0
5  2003  Nevada  3.2   5.0
```

When you are assigning lists or arrays to a column, the value's length must match the length of the DataFrame. If you assign a Series, its labels will be realigned exactly to the DataFrame's index, inserting missing values in any index values not present:

```
In [65]: val = pd.Series([-1.2, -1.5, -1.7], index=["two", "four", "five"])

In [66]: frame2["debt"] = val

In [67]: frame2
Out[67]:
   year   state  pop  debt
0  2000    Ohio  1.5   NaN
1  2001    Ohio  1.7   NaN
2  2002    Ohio  3.6   NaN
3  2001  Nevada  2.4   NaN
4  2002  Nevada  2.9   NaN
5  2003  Nevada  3.2   NaN
```

Assigning a column that doesn't exist will create a new column.

The `del` keyword will delete columns like with a dictionary. As an example, I first add a new column of Boolean values where the `state` column equals `"Ohio"`:

```
In [68]: frame2["eastern"] = frame2["state"] == "Ohio"

In [69]: frame2
Out[69]:
   year   state  pop  debt  eastern
0  2000    Ohio  1.5   NaN     True
1  2001    Ohio  1.7   NaN     True
2  2002    Ohio  3.6   NaN     True
3  2001  Nevada  2.4   NaN    False
4  2002  Nevada  2.9   NaN    False
5  2003  Nevada  3.2   NaN    False
```

New columns cannot be created with the `frame2.eastern` dot attribute notation.

The `del` method can then be used to remove this column:

```
In [70]: del frame2["eastern"]

In [71]: frame2.columns
Out[71]: Index(['year', 'state', 'pop', 'debt'], dtype='object')
```

The column returned from indexing a DataFrame is a *view* on the underlying data, not a copy. Thus, any in-place modifications to the Series will be reflected in the DataFrame. The column can be explicitly copied with the Series's `copy` method.

Another common form of data is a nested dictionary of dictionaries:

```
In [72]: populations = {"Ohio": {2000: 1.5, 2001: 1.7, 2002: 3.6},
   ....:                "Nevada": {2001: 2.4, 2002: 2.9}}
```

If the nested dictionary is passed to the DataFrame, pandas will interpret the outer dictionary keys as the columns, and the inner keys as the row indices:

```
In [73]: frame3 = pd.DataFrame(populations)

In [74]: frame3
Out[74]:
      Ohio  Nevada
2000   1.5     NaN
2001   1.7     2.4
2002   3.6     2.9
```

You can transpose the DataFrame (swap rows and columns) with similar syntax to a NumPy array:

```
In [75]: frame3.T
Out[75]:
        2000  2001  2002
Ohio     1.5   1.7   3.6
Nevada   NaN   2.4   2.9
```

Note that transposing discards the column data types if the columns do not all have the same data type, so transposing and then transposing back may lose the previous type information. The columns become arrays of pure Python objects in this case.

The keys in the inner dictionaries are combined to form the index in the result. This isn't true if an explicit index is specified:

```
In [76]: pd.DataFrame(populations, index=[2001, 2002, 2003])
Out[76]:
      Ohio  Nevada
2001   1.7     2.4
2002   3.6     2.9
2003   NaN     NaN
```

Dictionaries of Series are treated in much the same way:

```
In [77]: pdata = {"Ohio": frame3["Ohio"][:-1],
   ....:          "Nevada": frame3["Nevada"][:2]}
```

```
In [78]: pd.DataFrame(pdata)
Out[78]:
      Ohio  Nevada
2000   1.5     NaN
2001   1.7     2.4
```

For a list of many of the things you can pass to the DataFrame constructor, see Table 5-1.

Table 5-1. Possible data inputs to the DataFrame constructor

Type	Notes
2D ndarray	A matrix of data, passing optional row and column labels
Dictionary of arrays, lists, or tuples	Each sequence becomes a column in the DataFrame; all sequences must be the same length
NumPy structured/record array	Treated as the "dictionary of arrays" case
Dictionary of Series	Each value becomes a column; indexes from each Series are unioned together to form the result's row index if no explicit index is passed
Dictionary of dictionaries	Each inner dictionary becomes a column; keys are unioned to form the row index as in the "dictionary of Series" case
List of dictionaries or Series	Each item becomes a row in the DataFrame; unions of dictionary keys or Series indexes become the DataFrame's column labels
List of lists or tuples	Treated as the "2D ndarray" case
Another DataFrame	The DataFrame's indexes are used unless different ones are passed
NumPy MaskedArray	Like the "2D ndarray" case except masked values are missing in the DataFrame result

If a DataFrame's `index` and `columns` have their `name` attributes set, these will also be displayed:

```
In [79]: frame3.index.name = "year"

In [80]: frame3.columns.name = "state"

In [81]: frame3
Out[81]:
state  Ohio  Nevada
year
2000    1.5     NaN
2001    1.7     2.4
2002    3.6     2.9
```

Unlike Series, DataFrame does not have a `name` attribute. DataFrame's `to_numpy` method returns the data contained in the DataFrame as a two-dimensional ndarray:

```
In [82]: frame3.to_numpy()
Out[82]:
array([[1.5, nan],
```

```
       [1.7, 2.4],
       [3.6, 2.9]])
```

If the DataFrame's columns are different data types, the data type of the returned array will be chosen to accommodate all of the columns:

```
In [83]: frame2.to_numpy()
Out[83]:
array([[2000, 'Ohio', 1.5, nan],
       [2001, 'Ohio', 1.7, nan],
       [2002, 'Ohio', 3.6, nan],
       [2001, 'Nevada', 2.4, nan],
       [2002, 'Nevada', 2.9, nan],
       [2003, 'Nevada', 3.2, nan]], dtype=object)
```

Index Objects

pandas's Index objects are responsible for holding the axis labels (including a DataFrame's column names) and other metadata (like the axis name or names). Any array or other sequence of labels you use when constructing a Series or DataFrame is internally converted to an Index:

```
In [84]: obj = pd.Series(np.arange(3), index=["a", "b", "c"])

In [85]: index = obj.index

In [86]: index
Out[86]: Index(['a', 'b', 'c'], dtype='object')

In [87]: index[1:]
Out[87]: Index(['b', 'c'], dtype='object')
```

Index objects are immutable and thus can't be modified by the user:

```
index[1] = "d"  # TypeError
```

Immutability makes it safer to share Index objects among data structures:

```
In [88]: labels = pd.Index(np.arange(3))

In [89]: labels
Out[89]: Int64Index([0, 1, 2], dtype='int64')

In [90]: obj2 = pd.Series([1.5, -2.5, 0], index=labels)

In [91]: obj2
Out[91]:
0    1.5
1   -2.5
2    0.0
dtype: float64
```

```
In [92]: obj2.index is labels
Out[92]: True
```

Some users will not often take advantage of the capabilities provided by an Index, but because some operations will yield results containing indexed data, it's important to understand how they work.

In addition to being array-like, an Index also behaves like a fixed-size set:

```
In [93]: frame3
Out[93]:
state  Ohio  Nevada
year
2000    1.5     NaN
2001    1.7     2.4
2002    3.6     2.9

In [94]: frame3.columns
Out[94]: Index(['Ohio', 'Nevada'], dtype='object', name='state')

In [95]: "Ohio" in frame3.columns
Out[95]: True

In [96]: 2003 in frame3.index
Out[96]: False
```

Unlike Python sets, a pandas Index can contain duplicate labels:

```
In [97]: pd.Index(["foo", "foo", "bar", "bar"])
Out[97]: Index(['foo', 'foo', 'bar', 'bar'], dtype='object')
```

Selections with duplicate labels will select all occurrences of that label.

Each Index has a number of methods and properties for set logic, which answer other common questions about the data it contains. Some useful ones are summarized in Table 5-2.

Table 5-2. Some Index methods and properties

Method/Property	Description
`append()`	Concatenate with additional Index objects, producing a new Index
`difference()`	Compute set difference as an Index
`intersection()`	Compute set intersection
`union()`	Compute set union
`isin()`	Compute Boolean array indicating whether each value is contained in the passed collection
`delete()`	Compute new Index with element at Index `i` deleted
`drop()`	Compute new Index by deleting passed values
`insert()`	Compute new Index by inserting element at Index `i`

Method/Property	Description
`is_monotonic`	Returns `True` if each element is greater than or equal to the previous element
`is_unique`	Returns `True` if the Index has no duplicate values
`unique()`	Compute the array of unique values in the Index

5.2 Essential Functionality

This section will walk you through the fundamental mechanics of interacting with the data contained in a Series or DataFrame. In the chapters to come, we will delve more deeply into data analysis and manipulation topics using pandas. This book is not intended to serve as exhaustive documentation for the pandas library; instead, we'll focus on familiarizing you with heavily used features, leaving the less common (i.e., more esoteric) things for you to learn more about by reading the online pandas documentation.

Reindexing

An important method on pandas objects is `reindex`, which means to create a new object with the values rearranged to align with the new index. Consider an example:

```
In [98]: obj = pd.Series([4.5, 7.2, -5.3, 3.6], index=["d", "b", "a", "c"])

In [99]: obj
Out[99]:
d    4.5
b    7.2
a   -5.3
c    3.6
dtype: float64
```

Calling `reindex` on this Series rearranges the data according to the new index, introducing missing values if any index values were not already present:

```
In [100]: obj2 = obj.reindex(["a", "b", "c", "d", "e"])

In [101]: obj2
Out[101]:
a   -5.3
b    7.2
c    3.6
d    4.5
e    NaN
dtype: float64
```

For ordered data like time series, you may want to do some interpolation or filling of values when reindexing. The `method` option allows us to do this, using a method such as `ffill`, which forward-fills the values:

```
In [102]: obj3 = pd.Series(["blue", "purple", "yellow"], index=[0, 2, 4])

In [103]: obj3
Out[103]:
0      blue
2    purple
4    yellow
dtype: object

In [104]: obj3.reindex(np.arange(6), method="ffill")
Out[104]:
0      blue
1      blue
2    purple
3    purple
4    yellow
5    yellow
dtype: object
```

With DataFrame, `reindex` can alter the (row) index, columns, or both. When passed only a sequence, it reindexes the rows in the result:

```
In [105]: frame = pd.DataFrame(np.arange(9).reshape((3, 3)),
   .....:                      index=["a", "c", "d"],
   .....:                      columns=["Ohio", "Texas", "California"])

In [106]: frame
Out[106]:
   Ohio  Texas  California
a     0      1           2
c     3      4           5
d     6      7           8

In [107]: frame2 = frame.reindex(index=["a", "b", "c", "d"])

In [108]: frame2
Out[108]:
   Ohio  Texas  California
a   0.0    1.0         2.0
b   NaN    NaN         NaN
c   3.0    4.0         5.0
d   6.0    7.0         8.0
```

The columns can be reindexed with the `columns` keyword:

```
In [109]: states = ["Texas", "Utah", "California"]

In [110]: frame.reindex(columns=states)
Out[110]:
   Texas  Utah  California
a      1   NaN           2
c      4   NaN           5
d      7   NaN           8
```

Because "Ohio" was not in states, the data for that column is dropped from the result.

Another way to reindex a particular axis is to pass the new axis labels as a positional argument and then specify the axis to reindex with the axis keyword:

```
In [111]: frame.reindex(states, axis="columns")
Out[111]:
   Texas  Utah  California
a      1   NaN           2
c      4   NaN           5
d      7   NaN           8
```

See Table 5-3 for more about the arguments to reindex.

Table 5-3. reindex function arguments

Argument	Description
labels	New sequence to use as an index. Can be Index instance or any other sequence-like Python data structure. An Index will be used exactly as is without any copying.
index	Use the passed sequence as the new index labels.
columns	Use the passed sequence as the new column labels.
axis	The axis to reindex, whether "index" (rows) or "columns". The default is "index". You can alternately do reindex(index=new_labels) or reindex(columns=new_labels).
method	Interpolation (fill) method; "ffill" fills forward, while "bfill" fills backward.
fill_value	Substitute value to use when introducing missing data by reindexing. Use fill_value="missing" (the default behavior) when you want absent labels to have null values in the result.
limit	When forward filling or backfilling, the maximum size gap (in number of elements) to fill.
tolerance	When forward filling or backfilling, the maximum size gap (in absolute numeric distance) to fill for inexact matches.
level	Match simple Index on level of MultiIndex; otherwise select subset of.
copy	If True, always copy underlying data even if the new index is equivalent to the old index; if False, do not copy the data when the indexes are equivalent.

As we'll explore later in "Selection on DataFrame with loc and iloc" on page 147, you can also reindex by using the loc operator, and many users prefer to always do it this way. This works only if all of the new index labels already exist in the DataFrame (whereas reindex will insert missing data for new labels):

```
In [112]: frame.loc[["a", "d", "c"], ["California", "Texas"]]
Out[112]:
   California  Texas
a           2      1
d           8      7
c           5      4
```

Dropping Entries from an Axis

Dropping one or more entries from an axis is simple if you already have an index array or list without those entries, since you can use the `reindex` method or `.loc`-based indexing. As that can require a bit of munging and set logic, the `drop` method will return a new object with the indicated value or values deleted from an axis:

```
In [113]: obj = pd.Series(np.arange(5.), index=["a", "b", "c", "d", "e"])

In [114]: obj
Out[114]:
a    0.0
b    1.0
c    2.0
d    3.0
e    4.0
dtype: float64

In [115]: new_obj = obj.drop("c")

In [116]: new_obj
Out[116]:
a    0.0
b    1.0
d    3.0
e    4.0
dtype: float64

In [117]: obj.drop(["d", "c"])
Out[117]:
a    0.0
b    1.0
e    4.0
dtype: float64
```

With DataFrame, index values can be deleted from either axis. To illustrate this, we first create an example DataFrame:

```
In [118]: data = pd.DataFrame(np.arange(16).reshape((4, 4)),
   .....:                     index=["Ohio", "Colorado", "Utah", "New York"],
   .....:                     columns=["one", "two", "three", "four"])

In [119]: data
Out[119]:
          one  two  three  four
Ohio        0    1      2     3
Colorado    4    5      6     7
Utah        8    9     10    11
New York   12   13     14    15
```

Calling drop with a sequence of labels will drop values from the row labels (axis 0):

```
In [120]: data.drop(index=["Colorado", "Ohio"])
Out[120]:
          one  two  three  four
Utah        8    9     10    11
New York   12   13     14    15
```

To drop labels from the columns, instead use the columns keyword:

```
In [121]: data.drop(columns=["two"])
Out[121]:
          one  three  four
Ohio        0      2     3
Colorado    4      6     7
Utah        8     10    11
New York   12     14    15
```

You can also drop values from the columns by passing axis=1 (which is like NumPy) or axis="columns":

```
In [122]: data.drop("two", axis=1)
Out[122]:
          one  three  four
Ohio        0      2     3
Colorado    4      6     7
Utah        8     10    11
New York   12     14    15

In [123]: data.drop(["two", "four"], axis="columns")
Out[123]:
          one  three
Ohio        0      2
Colorado    4      6
Utah        8     10
New York   12     14
```

Indexing, Selection, and Filtering

Series indexing (obj[...]) works analogously to NumPy array indexing, except you can use the Series's index values instead of only integers. Here are some examples of this:

```
In [124]: obj = pd.Series(np.arange(4.), index=["a", "b", "c", "d"])

In [125]: obj
Out[125]:
a    0.0
b    1.0
c    2.0
d    3.0
dtype: float64
```

```
In [126]: obj["b"]
Out[126]: 1.0

In [127]: obj[1]
Out[127]: 1.0

In [128]: obj[2:4]
Out[128]:
c    2.0
d    3.0
dtype: float64

In [129]: obj[["b", "a", "d"]]
Out[129]:
b    1.0
a    0.0
d    3.0
dtype: float64

In [130]: obj[[1, 3]]
Out[130]:
b    1.0
d    3.0
dtype: float64

In [131]: obj[obj < 2]
Out[131]:
a    0.0
b    1.0
dtype: float64
```

While you can select data by label this way, the preferred way to select index values is with the special `loc` operator:

```
In [132]: obj.loc[["b", "a", "d"]]
Out[132]:
b    1.0
a    0.0
d    3.0
dtype: float64
```

The reason to prefer `loc` is because of the different treatment of integers when indexing with `[]`. Regular `[]`-based indexing will treat integers as labels if the index contains integers, so the behavior differs depending on the data type of the index. For example:

```
In [133]: obj1 = pd.Series([1, 2, 3], index=[2, 0, 1])

In [134]: obj2 = pd.Series([1, 2, 3], index=["a", "b", "c"])

In [135]: obj1
Out[135]:
```

```
2    1
0    2
1    3
dtype: int64

In [136]: obj2
Out[136]:
a    1
b    2
c    3
dtype: int64

In [137]: obj1[[0, 1, 2]]
Out[137]:
0    2
1    3
2    1
dtype: int64

In [138]: obj2[[0, 1, 2]]
Out[138]:
a    1
b    2
c    3
dtype: int64
```

When using `loc`, the expression `obj.loc[[0, 1, 2]]` will fail when the index does not contain integers:

```
In [134]: obj2.loc[[0, 1]]
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
/tmp/ipykernel_804589/4185657903.py in <module>
----> 1 obj2.loc[[0, 1]]

^ LONG EXCEPTION ABBREVIATED ^

KeyError: "None of [Int64Index([0, 1], dtype="int64")] are in the [index]"
```

Since `loc` operator indexes exclusively with labels, there is also an `iloc` operator that indexes exclusively with integers to work consistently whether or not the index contains integers:

```
In [139]: obj1.iloc[[0, 1, 2]]
Out[139]:
2    1
0    2
1    3
dtype: int64

In [140]: obj2.iloc[[0, 1, 2]]
Out[140]:
a    1
```

```
b    2
c    3
dtype: int64
```

You can also slice with labels, but it works differently from normal Python slicing in that the endpoint is inclusive:

```
In [141]: obj2.loc["b":"c"]
Out[141]:
b    2
c    3
dtype: int64
```

Assigning values using these methods modifies the corresponding section of the Series:

```
In [142]: obj2.loc["b":"c"] = 5

In [143]: obj2
Out[143]:
a    1
b    5
c    5
dtype: int64
```

It can be a common newbie error to try to call `loc` or `iloc` like functions rather than "indexing into" them with square brackets. The square bracket notation is used to enable slice operations and to allow for indexing on multiple axes with DataFrame objects.

Indexing into a DataFrame retrieves one or more columns either with a single value or sequence:

```
In [144]: data = pd.DataFrame(np.arange(16).reshape((4, 4)),
   .....:                     index=["Ohio", "Colorado", "Utah", "New York"],
   .....:                     columns=["one", "two", "three", "four"])

In [145]: data
Out[145]:
          one  two  three  four
Ohio        0    1      2     3
Colorado    4    5      6     7
Utah        8    9     10    11
New York   12   13     14    15

In [146]: data["two"]
Out[146]:
Ohio         1
Colorado     5
Utah         9
```

```
New York    13
Name: two, dtype: int64

In [147]: data[["three", "one"]]
Out[147]:
          three  one
Ohio          2    0
Colorado      6    4
Utah         10    8
New York     14   12
```

Indexing like this has a few special cases. The first is slicing or selecting data with a Boolean array:

```
In [148]: data[:2]
Out[148]:
          one  two  three  four
Ohio        0    1      2     3
Colorado    4    5      6     7

In [149]: data[data["three"] > 5]
Out[149]:
          one  two  three  four
Colorado    4    5      6     7
Utah        8    9     10    11
New York   12   13     14    15
```

The row selection syntax `data[:2]` is provided as a convenience. Passing a single element or a list to the `[]` operator selects columns.

Another use case is indexing with a Boolean DataFrame, such as one produced by a scalar comparison. Consider a DataFrame with all Boolean values produced by comparing with a scalar value:

```
In [150]: data < 5
Out[150]:
            one    two  three   four
Ohio       True   True   True   True
Colorado   True  False  False  False
Utah      False  False  False  False
New York  False  False  False  False
```

We can use this DataFrame to assign the value 0 to each location with the value `True`, like so:

```
In [151]: data[data < 5] = 0

In [152]: data
Out[152]:
          one  two  three  four
Ohio        0    0      0     0
Colorado    0    5      6     7
```

```
Utah        8    9     10    11
New York   12   13     14    15
```

Selection on DataFrame with loc and iloc

Like Series, DataFrame has special attributes `loc` and `iloc` for label-based and integer-based indexing, respectively. Since DataFrame is two-dimensional, you can select a subset of the rows and columns with NumPy-like notation using either axis labels (`loc`) or integers (`iloc`).

As a first example, let's select a single row by label:

```
In [153]: data
Out[153]:
          one  two  three  four
Ohio        0    0      0     0
Colorado    0    5      6     7
Utah        8    9     10    11
New York   12   13     14    15

In [154]: data.loc["Colorado"]
Out[154]:
one      0
two      5
three    6
four     7
Name: Colorado, dtype: int64
```

The result of selecting a single row is a Series with an index that contains the DataFrame's column labels. To select multiple roles, creating a new DataFrame, pass a sequence of labels:

```
In [155]: data.loc[["Colorado", "New York"]]
Out[155]:
          one  two  three  four
Colorado    0    5      6     7
New York   12   13     14    15
```

You can combine both row and column selection in `loc` by separating the selections with a comma:

```
In [156]: data.loc["Colorado", ["two", "three"]]
Out[156]:
two      5
three    6
Name: Colorado, dtype: int64
```

We'll then perform some similar selections with integers using `iloc`:

```
In [157]: data.iloc[2]
Out[157]:
one      8
two      9
```

```
three    10
four     11
Name: Utah, dtype: int64

In [158]: data.iloc[[2, 1]]
Out[158]:
          one  two  three  four
Utah        8    9     10    11
Colorado    0    5      6     7

In [159]: data.iloc[2, [3, 0, 1]]
Out[159]:
four    11
one      8
two      9
Name: Utah, dtype: int64

In [160]: data.iloc[[1, 2], [3, 0, 1]]
Out[160]:
          four  one  two
Colorado     7    0    5
Utah        11    8    9
```

Both indexing functions work with slices in addition to single labels or lists of labels:

```
In [161]: data.loc[:"Utah", "two"]
Out[161]:
Ohio        0
Colorado    5
Utah        9
Name: two, dtype: int64

In [162]: data.iloc[:, :3][data.three > 5]
Out[162]:
          one  two  three
Colorado    0    5      6
Utah        8    9     10
New York   12   13     14
```

Boolean arrays can be used with `loc` but not `iloc`:

```
In [163]: data.loc[data.three >= 2]
Out[163]:
          one  two  three  four
Colorado    0    5      6     7
Utah        8    9     10    11
New York   12   13     14    15
```

There are many ways to select and rearrange the data contained in a pandas object. For DataFrame, Table 5-4 provides a short summary of many of them. As you will see later, there are a number of additional options for working with hierarchical indexes.

Table 5-4. Indexing options with DataFrame

Type	Notes
`df[column]`	Select single column or sequence of columns from the DataFrame; special case conveniences: Boolean array (filter rows), slice (slice rows), or Boolean DataFrame (set values based on some criterion)
`df.loc[rows]`	Select single row or subset of rows from the DataFrame by label
`df.loc[:, cols]`	Select single column or subset of columns by label
`df.loc[rows, cols]`	Select both row(s) and column(s) by label
`df.iloc[rows]`	Select single row or subset of rows from the DataFrame by integer position
`df.iloc[:, cols]`	Select single column or subset of columns by integer position
`df.iloc[rows, cols]`	Select both row(s) and column(s) by integer position
`df.at[row, col]`	Select a single scalar value by row and column label
`df.iat[row, col]`	Select a single scalar value by row and column position (integers)
`reindex` method	Select either rows or columns by labels

Integer indexing pitfalls

Working with pandas objects indexed by integers can be a stumbling block for new users since they work differently from built-in Python data structures like lists and tuples. For example, you might not expect the following code to generate an error:

```
In [164]: ser = pd.Series(np.arange(3.))

In [165]: ser
Out[165]:
0    0.0
1    1.0
2    2.0
dtype: float64

In [166]: ser[-1]
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
/miniconda/envs/book-env/lib/python3.10/site-packages/pandas/core/indexes/range.p
y in get_loc(self, key, method, tolerance)
    384                 try:
--> 385                     return self._range.index(new_key)
    386                 except ValueError as err:
ValueError: -1 is not in range
The above exception was the direct cause of the following exception:
KeyError                                  Traceback (most recent call last)
<ipython-input-166-44969a759c20> in <module>
----> 1 ser[-1]
/miniconda/envs/book-env/lib/python3.10/site-packages/pandas/core/series.py in __
getitem__(self, key)
    956
    957         elif key_is_scalar:
--> 958             return self._get_value(key)
```

```
    959
    960         if is_hashable(key):
/miniconda/envs/book-env/lib/python3.10/site-packages/pandas/core/series.py in _g
et_value(self, label, takeable)
   1067
   1068         # Similar to Index.get_value, but we do not fall back to position
al
-> 1069         loc = self.index.get_loc(label)
   1070         return self.index._get_values_for_loc(self, loc, label)
   1071
/miniconda/envs/book-env/lib/python3.10/site-packages/pandas/core/indexes/range.p
y in get_loc(self, key, method, tolerance)
    385                 return self._range.index(new_key)
    386             except ValueError as err:
--> 387                 raise KeyError(key) from err
    388         self._check_indexing_error(key)
    389         raise KeyError(key)
KeyError: -1
```

In this case, pandas could "fall back" on integer indexing, but it is difficult to do this in general without introducing subtle bugs into the user code. Here we have an index containing `0`, `1`, and `2`, but pandas does not want to guess what the user wants (label-based indexing or position-based):

```
In [167]: ser
Out[167]:
0    0.0
1    1.0
2    2.0
dtype: float64
```

On the other hand, with a noninteger index, there is no such ambiguity:

```
In [168]: ser2 = pd.Series(np.arange(3.), index=["a", "b", "c"])

In [169]: ser2[-1]
Out[169]: 2.0
```

If you have an axis index containing integers, data selection will always be label oriented. As I said above, if you use `loc` (for labels) or `iloc` (for integers) you will get exactly what you want:

```
In [170]: ser.iloc[-1]
Out[170]: 2.0
```

On the other hand, slicing with integers is always integer oriented:

```
In [171]: ser[:2]
Out[171]:
0    0.0
1    1.0
dtype: float64
```

As a result of these pitfalls, it is best to always prefer indexing with `loc` and `iloc` to avoid ambiguity.

Pitfalls with chained indexing

In the previous section we looked at how you can do flexible selections on a DataFrame using `loc` and `iloc`. These indexing attributes can also be used to modify DataFrame objects in place, but doing so requires some care.

For example, in the example DataFrame above, we can assign to a column or row by label or integer position:

```
In [172]: data.loc[:, "one"] = 1

In [173]: data
Out[173]:
          one  two  three  four
Ohio        1    0      0     0
Colorado    1    5      6     7
Utah        1    9     10    11
New York    1   13     14    15

In [174]: data.iloc[2] = 5

In [175]: data
Out[175]:
          one  two  three  four
Ohio        1    0      0     0
Colorado    1    5      6     7
Utah        5    5      5     5
New York    1   13     14    15

In [176]: data.loc[data["four"] > 5] = 3

In [177]: data
Out[177]:
          one  two  three  four
Ohio        1    0      0     0
Colorado    3    3      3     3
Utah        5    5      5     5
New York    3    3      3     3
```

A common gotcha for new pandas users is to chain selections when assigning, like this:

```
In [177]: data.loc[data.three == 5]["three"] = 6
<ipython-input-11-0ed1cf2155d5>:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead
```

Depending on the data contents, this may print a special `SettingWithCopyWarning`, which warns you that you are trying to modify a temporary value (the nonempty result of `data.loc[data.three == 5]`) instead of the original DataFrame `data`, which might be what you were intending. Here, `data` was unmodified:

```
In [179]: data
Out[179]:
          one  two  three  four
Ohio        1    0      0     0
Colorado    3    3      3     3
Utah        5    5      5     5
New York    3    3      3     3
```

In these scenarios, the fix is to rewrite the chained assignment to use a single `loc` operation:

```
In [180]: data.loc[data.three == 5, "three"] = 6

In [181]: data
Out[181]:
          one  two  three  four
Ohio        1    0      0     0
Colorado    3    3      3     3
Utah        5    5      6     5
New York    3    3      3     3
```

A good rule of thumb is to avoid chained indexing when doing assignments. There are other cases where pandas will generate `SettingWithCopyWarning` that have to do with chained indexing. I refer you to this topic in the online pandas documentation.

Arithmetic and Data Alignment

pandas can make it much simpler to work with objects that have different indexes. For example, when you add objects, if any index pairs are not the same, the respective index in the result will be the union of the index pairs. Let's look at an example:

```
In [182]: s1 = pd.Series([7.3, -2.5, 3.4, 1.5], index=["a", "c", "d", "e"])

In [183]: s2 = pd.Series([-2.1, 3.6, -1.5, 4, 3.1],
   .....:                index=["a", "c", "e", "f", "g"])

In [184]: s1
Out[184]:
a    7.3
c   -2.5
d    3.4
e    1.5
dtype: float64

In [185]: s2
Out[185]:
a   -2.1
c    3.6
e   -1.5
```

```
f    4.0
g    3.1
dtype: float64
```

Adding these yields:

```
In [186]: s1 + s2
Out[186]:
a    5.2
c    1.1
d    NaN
e    0.0
f    NaN
g    NaN
dtype: float64
```

The internal data alignment introduces missing values in the label locations that don't overlap. Missing values will then propagate in further arithmetic computations.

In the case of DataFrame, alignment is performed on both rows and columns:

```
In [187]: df1 = pd.DataFrame(np.arange(9.).reshape((3, 3)), columns=list("bcd"),
   .....:                    index=["Ohio", "Texas", "Colorado"])

In [188]: df2 = pd.DataFrame(np.arange(12.).reshape((4, 3)), columns=list("bde"),
   .....:                    index=["Utah", "Ohio", "Texas", "Oregon"])

In [189]: df1
Out[189]:
            b    c    d
Ohio      0.0  1.0  2.0
Texas     3.0  4.0  5.0
Colorado  6.0  7.0  8.0

In [190]: df2
Out[190]:
          b     d     e
Utah    0.0   1.0   2.0
Ohio    3.0   4.0   5.0
Texas   6.0   7.0   8.0
Oregon  9.0  10.0  11.0
```

Adding these returns a DataFrame with index and columns that are the unions of the ones in each DataFrame:

```
In [191]: df1 + df2
Out[191]:
            b   c     d   e
Colorado  NaN NaN   NaN NaN
Ohio      3.0 NaN   6.0 NaN
Oregon    NaN NaN   NaN NaN
Texas     9.0 NaN  12.0 NaN
Utah      NaN NaN   NaN NaN
```

Since the "c" and "e" columns are not found in both DataFrame objects, they appear as missing in the result. The same holds for the rows with labels that are not common to both objects.

If you add DataFrame objects with no column or row labels in common, the result will contain all nulls:

```
In [192]: df1 = pd.DataFrame({"A": [1, 2]})

In [193]: df2 = pd.DataFrame({"B": [3, 4]})

In [194]: df1
Out[194]:
   A
0  1
1  2

In [195]: df2
Out[195]:
   B
0  3
1  4

In [196]: df1 + df2
Out[196]:
    A   B
0 NaN NaN
1 NaN NaN
```

Arithmetic methods with fill values

In arithmetic operations between differently indexed objects, you might want to fill with a special value, like 0, when an axis label is found in one object but not the other. Here is an example where we set a particular value to NA (null) by assigning `np.nan` to it:

```
In [197]: df1 = pd.DataFrame(np.arange(12.).reshape((3, 4)),
   .....:                    columns=list("abcd"))

In [198]: df2 = pd.DataFrame(np.arange(20.).reshape((4, 5)),
   .....:                    columns=list("abcde"))

In [199]: df2.loc[1, "b"] = np.nan

In [200]: df1
Out[200]:
     a    b     c     d
0  0.0  1.0   2.0   3.0
1  4.0  5.0   6.0   7.0
2  8.0  9.0  10.0  11.0
```

```
In [201]: df2
Out[201]:
      a     b     c     d     e
0   0.0   1.0   2.0   3.0   4.0
1   5.0   NaN   7.0   8.0   9.0
2  10.0  11.0  12.0  13.0  14.0
3  15.0  16.0  17.0  18.0  19.0
```

Adding these results in missing values in the locations that don't overlap:

```
In [202]: df1 + df2
Out[202]:
      a     b     c     d   e
0   0.0   2.0   4.0   6.0 NaN
1   9.0   NaN  13.0  15.0 NaN
2  18.0  20.0  22.0  24.0 NaN
3   NaN   NaN   NaN   NaN NaN
```

Using the `add` method on `df1`, I pass `df2` and an argument to `fill_value`, which substitutes the passed value for any missing values in the operation:

```
In [203]: df1.add(df2, fill_value=0)
Out[203]:
      a     b     c     d     e
0   0.0   2.0   4.0   6.0   4.0
1   9.0   5.0  13.0  15.0   9.0
2  18.0  20.0  22.0  24.0  14.0
3  15.0  16.0  17.0  18.0  19.0
```

See Table 5-5 for a listing of Series and DataFrame methods for arithmetic. Each has a counterpart, starting with the letter `r`, that has arguments reversed. So these two statements are equivalent:

```
In [204]: 1 / df1
Out[204]:
       a         b         c         d
0    inf  1.000000  0.500000  0.333333
1  0.250  0.200000  0.166667  0.142857
2  0.125  0.111111  0.100000  0.090909

In [205]: df1.rdiv(1)
Out[205]:
       a         b         c         d
0    inf  1.000000  0.500000  0.333333
1  0.250  0.200000  0.166667  0.142857
2  0.125  0.111111  0.100000  0.090909
```

Relatedly, when reindexing a Series or DataFrame, you can also specify a different fill value:

```
In [206]: df1.reindex(columns=df2.columns, fill_value=0)
Out[206]:
     a    b     c     d  e
0  0.0  1.0   2.0   3.0  0
```

```
1  4.0  5.0   6.0   7.0  0
2  8.0  9.0  10.0  11.0  0
```

Table 5-5. Flexible arithmetic methods

Method	Description
`add, radd`	Methods for addition (+)
`sub, rsub`	Methods for subtraction (-)
`div, rdiv`	Methods for division (/)
`floordiv, rfloordiv`	Methods for floor division (//)
`mul, rmul`	Methods for multiplication (*)
`pow, rpow`	Methods for exponentiation (**)

Operations between DataFrame and Series

As with NumPy arrays of different dimensions, arithmetic between DataFrame and Series is also defined. First, as a motivating example, consider the difference between a two-dimensional array and one of its rows:

```
In [207]: arr = np.arange(12.).reshape((3, 4))

In [208]: arr
Out[208]:
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  6.,  7.],
       [ 8.,  9., 10., 11.]])

In [209]: arr[0]
Out[209]: array([0., 1., 2., 3.])

In [210]: arr - arr[0]
Out[210]:
array([[0., 0., 0., 0.],
       [4., 4., 4., 4.],
       [8., 8., 8., 8.]])
```

When we subtract `arr[0]` from `arr`, the subtraction is performed once for each row. This is referred to as *broadcasting* and is explained in more detail as it relates to general NumPy arrays in Appendix A. Operations between a DataFrame and a Series are similar:

```
In [211]: frame = pd.DataFrame(np.arange(12.).reshape((4, 3)),
   .....:                      columns=list("bde"),
   .....:                      index=["Utah", "Ohio", "Texas", "Oregon"])

In [212]: series = frame.iloc[0]

In [213]: frame
Out[213]:
          b     d     e
```

```
Utah    0.0   1.0   2.0
Ohio    3.0   4.0   5.0
Texas   6.0   7.0   8.0
Oregon  9.0  10.0  11.0

In [214]: series
Out[214]:
b    0.0
d    1.0
e    2.0
Name: Utah, dtype: float64
```

By default, arithmetic between DataFrame and Series matches the index of the Series on the columns of the DataFrame, broadcasting down the rows:

```
In [215]: frame - series
Out[215]:
          b    d    e
Utah    0.0  0.0  0.0
Ohio    3.0  3.0  3.0
Texas   6.0  6.0  6.0
Oregon  9.0  9.0  9.0
```

If an index value is not found in either the DataFrame's columns or the Series's index, the objects will be reindexed to form the union:

```
In [216]: series2 = pd.Series(np.arange(3), index=["b", "e", "f"])

In [217]: series2
Out[217]:
b    0
e    1
f    2
dtype: int64

In [218]: frame + series2
Out[218]:
          b   d     e   f
Utah    0.0 NaN   3.0 NaN
Ohio    3.0 NaN   6.0 NaN
Texas   6.0 NaN   9.0 NaN
Oregon  9.0 NaN  12.0 NaN
```

If you want to instead broadcast over the columns, matching on the rows, you have to use one of the arithmetic methods and specify to match over the index. For example:

```
In [219]: series3 = frame["d"]

In [220]: frame
Out[220]:
          b     d     e
Utah    0.0   1.0   2.0
Ohio    3.0   4.0   5.0
```

```
Texas   6.0   7.0   8.0
Oregon  9.0  10.0  11.0

In [221]: series3
Out[221]:
Utah       1.0
Ohio       4.0
Texas      7.0
Oregon    10.0
Name: d, dtype: float64

In [222]: frame.sub(series3, axis="index")
Out[222]:
          b    d    e
Utah   -1.0  0.0  1.0
Ohio   -1.0  0.0  1.0
Texas  -1.0  0.0  1.0
Oregon -1.0  0.0  1.0
```

The axis that you pass is the *axis to match on*. In this case we mean to match on the DataFrame's row index (`axis="index"`) and broadcast across the columns.

Function Application and Mapping

NumPy ufuncs (element-wise array methods) also work with pandas objects:

```
In [223]: frame = pd.DataFrame(np.random.standard_normal((4, 3)),
   .....:                      columns=list("bde"),
   .....:                      index=["Utah", "Ohio", "Texas", "Oregon"])

In [224]: frame
Out[224]:
               b         d         e
Utah   -0.204708  0.478943 -0.519439
Ohio   -0.555730  1.965781  1.393406
Texas   0.092908  0.281746  0.769023
Oregon  1.246435  1.007189 -1.296221

In [225]: np.abs(frame)
Out[225]:
               b         d         e
Utah    0.204708  0.478943  0.519439
Ohio    0.555730  1.965781  1.393406
Texas   0.092908  0.281746  0.769023
Oregon  1.246435  1.007189  1.296221
```

Another frequent operation is applying a function on one-dimensional arrays to each column or row. DataFrame's `apply` method does exactly this:

```
In [226]: def f1(x):
   .....:     return x.max() - x.min()
```

```
In [227]: frame.apply(f1)
Out[227]:
b    1.802165
d    1.684034
e    2.689627
dtype: float64
```

Here the function f, which computes the difference between the maximum and minimum of a Series, is invoked once on each column in `frame`. The result is a Series having the columns of `frame` as its index.

If you pass `axis="columns"` to `apply`, the function will be invoked once per row instead. A helpful way to think about this is as "apply across the columns":

```
In [228]: frame.apply(f1, axis="columns")
Out[228]:
Utah      0.998382
Ohio      2.521511
Texas     0.676115
Oregon    2.542656
dtype: float64
```

Many of the most common array statistics (like `sum` and `mean`) are DataFrame methods, so using `apply` is not necessary.

The function passed to `apply` need not return a scalar value; it can also return a Series with multiple values:

```
In [229]: def f2(x):
   .....:     return pd.Series([x.min(), x.max()], index=["min", "max"])

In [230]: frame.apply(f2)
Out[230]:
            b         d         e
min -0.555730  0.281746 -1.296221
max  1.246435  1.965781  1.393406
```

Element-wise Python functions can be used, too. Suppose you wanted to compute a formatted string from each floating-point value in `frame`. You can do this with `applymap`:

```
In [231]: def my_format(x):
   .....:     return f"{x:.2f}"

In [232]: frame.applymap(my_format)
Out[232]:
            b     d      e
Utah    -0.20  0.48  -0.52
Ohio    -0.56  1.97   1.39
Texas    0.09  0.28   0.77
Oregon   1.25  1.01  -1.30
```

The reason for the name `applymap` is that Series has a `map` method for applying an element-wise function:

```
In [233]: frame["e"].map(my_format)
Out[233]:
Utah      -0.52
Ohio       1.39
Texas      0.77
Oregon    -1.30
Name: e, dtype: object
```

Sorting and Ranking

Sorting a dataset by some criterion is another important built-in operation. To sort lexicographically by row or column label, use the `sort_index` method, which returns a new, sorted object:

```
In [234]: obj = pd.Series(np.arange(4), index=["d", "a", "b", "c"])

In [235]: obj
Out[235]:
d    0
a    1
b    2
c    3
dtype: int64

In [236]: obj.sort_index()
Out[236]:
a    1
b    2
c    3
d    0
dtype: int64
```

With a DataFrame, you can sort by index on either axis:

```
In [237]: frame = pd.DataFrame(np.arange(8).reshape((2, 4)),
   .....:                      index=["three", "one"],
   .....:                      columns=["d", "a", "b", "c"])

In [238]: frame
Out[238]:
       d  a  b  c
three  0  1  2  3
one    4  5  6  7

In [239]: frame.sort_index()
Out[239]:
       d  a  b  c
one    4  5  6  7
three  0  1  2  3
```

```
In [240]: frame.sort_index(axis="columns")
Out[240]:
       a  b  c  d
three  1  2  3  0
one    5  6  7  4
```

The data is sorted in ascending order by default but can be sorted in descending order, too:

```
In [241]: frame.sort_index(axis="columns", ascending=False)
Out[241]:
       d  c  b  a
three  0  3  2  1
one    4  7  6  5
```

To sort a Series by its values, use its `sort_values` method:

```
In [242]: obj = pd.Series([4, 7, -3, 2])

In [243]: obj.sort_values()
Out[243]:
2   -3
3    2
0    4
1    7
dtype: int64
```

Any missing values are sorted to the end of the Series by default:

```
In [244]: obj = pd.Series([4, np.nan, 7, np.nan, -3, 2])

In [245]: obj.sort_values()
Out[245]:
4   -3.0
5    2.0
0    4.0
2    7.0
1    NaN
3    NaN
dtype: float64
```

Missing values can be sorted to the start instead by using the `na_position` option:

```
In [246]: obj.sort_values(na_position="first")
Out[246]:
1    NaN
3    NaN
4   -3.0
5    2.0
0    4.0
2    7.0
dtype: float64
```

When sorting a DataFrame, you can use the data in one or more columns as the sort keys. To do so, pass one or more column names to `sort_values`:

```
In [247]: frame = pd.DataFrame({"b": [4, 7, -3, 2], "a": [0, 1, 0, 1]})

In [248]: frame
Out[248]:
   b  a
0  4  0
1  7  1
2 -3  0
3  2  1

In [249]: frame.sort_values("b")
Out[249]:
   b  a
2 -3  0
3  2  1
0  4  0
1  7  1
```

To sort by multiple columns, pass a list of names:

```
In [250]: frame.sort_values(["a", "b"])
Out[250]:
   b  a
2 -3  0
0  4  0
3  2  1
1  7  1
```

Ranking assigns ranks from one through the number of valid data points in an array, starting from the lowest value. The `rank` methods for Series and DataFrame are the place to look; by default, `rank` breaks ties by assigning each group the mean rank:

```
In [251]: obj = pd.Series([7, -5, 7, 4, 2, 0, 4])

In [252]: obj.rank()
Out[252]:
0    6.5
1    1.0
2    6.5
3    4.5
4    3.0
5    2.0
6    4.5
dtype: float64
```

Ranks can also be assigned according to the order in which they're observed in the data:

```
In [253]: obj.rank(method="first")
Out[253]:
0    6.0
1    1.0
2    7.0
3    4.0
4    3.0
5    2.0
6    5.0
dtype: float64
```

Here, instead of using the average rank 6.5 for the entries 0 and 2, they instead have been set to 6 and 7 because label 0 precedes label 2 in the data.

You can rank in descending order, too:

```
In [254]: obj.rank(ascending=False)
Out[254]:
0    1.5
1    7.0
2    1.5
3    3.5
4    5.0
5    6.0
6    3.5
dtype: float64
```

See Table 5-6 for a list of tie-breaking methods available.

DataFrame can compute ranks over the rows or the columns:

```
In [255]: frame = pd.DataFrame({"b": [4.3, 7, -3, 2], "a": [0, 1, 0, 1],
   .....:                       "c": [-2, 5, 8, -2.5]})

In [256]: frame
Out[256]:
     b  a    c
0  4.3  0 -2.0
1  7.0  1  5.0
2 -3.0  0  8.0
3  2.0  1 -2.5

In [257]: frame.rank(axis="columns")
Out[257]:
     b    a    c
0  3.0  2.0  1.0
1  3.0  1.0  2.0
2  1.0  2.0  3.0
3  3.0  2.0  1.0
```

Table 5-6. Tie-breaking methods with rank

Method	Description
`"average"`	Default: assign the average rank to each entry in the equal group
`"min"`	Use the minimum rank for the whole group
`"max"`	Use the maximum rank for the whole group
`"first"`	Assign ranks in the order the values appear in the data
`"dense"`	Like `method="min"`, but ranks always increase by 1 between groups rather than the number of equal elements in a group

Axis Indexes with Duplicate Labels

Up until now almost all of the examples we have looked at have unique axis labels (index values). While many pandas functions (like `reindex`) require that the labels be unique, it's not mandatory. Let's consider a small Series with duplicate indices:

```
In [258]: obj = pd.Series(np.arange(5), index=["a", "a", "b", "b", "c"])

In [259]: obj
Out[259]:
a    0
a    1
b    2
b    3
c    4
dtype: int64
```

The `is_unique` property of the index can tell you whether or not its labels are unique:

```
In [260]: obj.index.is_unique
Out[260]: False
```

Data selection is one of the main things that behaves differently with duplicates. Indexing a label with multiple entries returns a Series, while single entries return a scalar value:

```
In [261]: obj["a"]
Out[261]:
a    0
a    1
dtype: int64

In [262]: obj["c"]
Out[262]: 4
```

This can make your code more complicated, as the output type from indexing can vary based on whether or not a label is repeated.

The same logic extends to indexing rows (or columns) in a DataFrame:

```
In [263]: df = pd.DataFrame(np.random.standard_normal((5, 3)),
   .....:                   index=["a", "a", "b", "b", "c"])

In [264]: df
Out[264]: 
          0         1         2
a  0.274992  0.228913  1.352917
a  0.886429 -2.001637 -0.371843
b  1.669025 -0.438570 -0.539741
b  0.476985  3.248944 -1.021228
c -0.577087  0.124121  0.302614

In [265]: df.loc["b"]
Out[265]: 
          0         1         2
b  1.669025 -0.438570 -0.539741
b  0.476985  3.248944 -1.021228

In [266]: df.loc["c"]
Out[266]: 
0   -0.577087
1    0.124121
2    0.302614
Name: c, dtype: float64
```

5.3 Summarizing and Computing Descriptive Statistics

pandas objects are equipped with a set of common mathematical and statistical methods. Most of these fall into the category of *reductions* or *summary statistics*, methods that extract a single value (like the sum or mean) from a Series, or a Series of values from the rows or columns of a DataFrame. Compared with the similar methods found on NumPy arrays, they have built-in handling for missing data. Consider a small DataFrame:

```
In [267]: df = pd.DataFrame([[1.4, np.nan], [7.1, -4.5],
   .....:                    [np.nan, np.nan], [0.75, -1.3]],
   .....:                   index=["a", "b", "c", "d"],
   .....:                   columns=["one", "two"])

In [268]: df
Out[268]: 
    one  two
a  1.40  NaN
b  7.10 -4.5
c   NaN  NaN
d  0.75 -1.3
```

Calling DataFrame's `sum` method returns a Series containing column sums:

```
In [269]: df.sum()
Out[269]:
one    9.25
two   -5.80
dtype: float64
```

Passing `axis="columns"` or `axis=1` sums across the columns instead:

```
In [270]: df.sum(axis="columns")
Out[270]:
a    1.40
b    2.60
c    0.00
d   -0.55
dtype: float64
```

When an entire row or column contains all NA values, the sum is 0, whereas if any value is not NA, then the result is NA. This can be disabled with the `skipna` option, in which case any NA value in a row or column names the corresponding result NA:

```
In [271]: df.sum(axis="index", skipna=False)
Out[271]:
one   NaN
two   NaN
dtype: float64

In [272]: df.sum(axis="columns", skipna=False)
Out[272]:
a     NaN
b    2.60
c     NaN
d   -0.55
dtype: float64
```

Some aggregations, like `mean`, require at least one non-NA value to yield a value result, so here we have:

```
In [273]: df.mean(axis="columns")
Out[273]:
a    1.400
b    1.300
c      NaN
d   -0.275
dtype: float64
```

See Table 5-7 for a list of common options for each reduction method.

Table 5-7. Options for reduction methods

Method	Description
`axis`	Axis to reduce over; "index" for DataFrame's rows and "columns" for columns
`skipna`	Exclude missing values; `True` by default
`level`	Reduce grouped by level if the axis is hierarchically indexed (MultiIndex)

Some methods, like `idxmin` and `idxmax`, return indirect statistics, like the index value where the minimum or maximum values are attained:

```
In [274]: df.idxmax()
Out[274]:
one    b
two    d
dtype: object
```

Other methods are *accumulations*:

```
In [275]: df.cumsum()
Out[275]:
    one  two
a  1.40  NaN
b  8.50 -4.5
c   NaN  NaN
d  9.25 -5.8
```

Some methods are neither reductions nor accumulations. `describe` is one such example, producing multiple summary statistics in one shot:

```
In [276]: df.describe()
Out[276]:
            one       two
count  3.000000  2.000000
mean   3.083333 -2.900000
std    3.493685  2.262742
min    0.750000 -4.500000
25%    1.075000 -3.700000
50%    1.400000 -2.900000
75%    4.250000 -2.100000
max    7.100000 -1.300000
```

On nonnumeric data, `describe` produces alternative summary statistics:

```
In [277]: obj = pd.Series(["a", "a", "b", "c"] * 4)

In [278]: obj.describe()
Out[278]:
count     16
unique     3
top        a
freq       8
dtype: object
```

See Table 5-8 for a full list of summary statistics and related methods.

Table 5-8. Descriptive and summary statistics

Method	Description
`count`	Number of non-NA values
`describe`	Compute set of summary statistics
`min, max`	Compute minimum and maximum values
`argmin, argmax`	Compute index locations (integers) at which minimum or maximum value is obtained, respectively; not available on DataFrame objects
`idxmin, idxmax`	Compute index labels at which minimum or maximum value is obtained, respectively
`quantile`	Compute sample quantile ranging from 0 to 1 (default: 0.5)
`sum`	Sum of values
`mean`	Mean of values
`median`	Arithmetic median (50% quantile) of values
`mad`	Mean absolute deviation from mean value
`prod`	Product of all values
`var`	Sample variance of values
`std`	Sample standard deviation of values
`skew`	Sample skewness (third moment) of values
`kurt`	Sample kurtosis (fourth moment) of values
`cumsum`	Cumulative sum of values
`cummin, cummax`	Cumulative minimum or maximum of values, respectively
`cumprod`	Cumulative product of values
`diff`	Compute first arithmetic difference (useful for time series)
`pct_change`	Compute percent changes

Correlation and Covariance

Some summary statistics, like correlation and covariance, are computed from pairs of arguments. Let's consider some DataFrames of stock prices and volumes originally obtained from Yahoo! Finance and available in binary Python pickle files you can find in the accompanying datasets for the book:

```
In [279]: price = pd.read_pickle("examples/yahoo_price.pkl")

In [280]: volume = pd.read_pickle("examples/yahoo_volume.pkl")
```

I now compute percent changes of the prices, a time series operation that will be explored further in Chapter 11:

```
In [281]: returns = price.pct_change()

In [282]: returns.tail()
Out[282]: 
                AAPL      GOOG       IBM      MSFT
```

```
Date
2016-10-17 -0.000680  0.001837  0.002072 -0.003483
2016-10-18 -0.000681  0.019616 -0.026168  0.007690
2016-10-19 -0.002979  0.007846  0.003583 -0.002255
2016-10-20 -0.000512 -0.005652  0.001719 -0.004867
2016-10-21 -0.003930  0.003011 -0.012474  0.042096
```

The `corr` method of Series computes the correlation of the overlapping, non-NA, aligned-by-index values in two Series. Relatedly, `cov` computes the covariance:

```
In [283]: returns["MSFT"].corr(returns["IBM"])
Out[283]: 0.49976361144151144

In [284]: returns["MSFT"].cov(returns["IBM"])
Out[284]: 8.870655479703546e-05
```

Since `MSFT` is a valid Python variable name, we can also select these columns using more concise syntax:

```
In [285]: returns["MSFT"].corr(returns["IBM"])
Out[285]: 0.49976361144151144
```

DataFrame's `corr` and `cov` methods, on the other hand, return a full correlation or covariance matrix as a DataFrame, respectively:

```
In [286]: returns.corr()
Out[286]:
          AAPL      GOOG       IBM      MSFT
AAPL  1.000000  0.407919  0.386817  0.389695
GOOG  0.407919  1.000000  0.405099  0.465919
IBM   0.386817  0.405099  1.000000  0.499764
MSFT  0.389695  0.465919  0.499764  1.000000

In [287]: returns.cov()
Out[287]:
          AAPL      GOOG       IBM      MSFT
AAPL  0.000277  0.000107  0.000078  0.000095
GOOG  0.000107  0.000251  0.000078  0.000108
IBM   0.000078  0.000078  0.000146  0.000089
MSFT  0.000095  0.000108  0.000089  0.000215
```

Using DataFrame's `corrwith` method, you can compute pair-wise correlations between a DataFrame's columns or rows with another Series or DataFrame. Passing a Series returns a Series with the correlation value computed for each column:

```
In [288]: returns.corrwith(returns["IBM"])
Out[288]:
AAPL    0.386817
GOOG    0.405099
IBM     1.000000
MSFT    0.499764
dtype: float64
```

Passing a DataFrame computes the correlations of matching column names. Here, I compute correlations of percent changes with volume:

```
In [289]: returns.corrwith(volume)
Out[289]:
AAPL   -0.075565
GOOG   -0.007067
IBM    -0.204849
MSFT   -0.092950
dtype: float64
```

Passing `axis="columns"` does things row-by-row instead. In all cases, the data points are aligned by label before the correlation is computed.

Unique Values, Value Counts, and Membership

Another class of related methods extracts information about the values contained in a one-dimensional Series. To illustrate these, consider this example:

```
In [290]: obj = pd.Series(["c", "a", "d", "a", "a", "b", "b", "c", "c"])
```

The first function is `unique`, which gives you an array of the unique values in a Series:

```
In [291]: uniques = obj.unique()

In [292]: uniques
Out[292]: array(['c', 'a', 'd', 'b'], dtype=object)
```

The unique values are not necessarily returned in the order in which they first appear, and not in sorted order, but they could be sorted after the fact if needed (`uniques.sort()`). Relatedly, `value_counts` computes a Series containing value frequencies:

```
In [293]: obj.value_counts()
Out[293]:
c    3
a    3
b    2
d    1
dtype: int64
```

The Series is sorted by value in descending order as a convenience. `value_counts` is also available as a top-level pandas method that can be used with NumPy arrays or other Python sequences:

```
In [294]: pd.value_counts(obj.to_numpy(), sort=False)
Out[294]:
c    3
a    3
d    1
b    2
dtype: int64
```

`isin` performs a vectorized set membership check and can be useful in filtering a dataset down to a subset of values in a Series or column in a DataFrame:

```
In [295]: obj
Out[295]:
0    c
1    a
2    d
3    a
4    a
5    b
6    b
7    c
8    c
dtype: object

In [296]: mask = obj.isin(["b", "c"])

In [297]: mask
Out[297]:
0     True
1    False
2    False
3    False
4    False
5     True
6     True
7     True
8     True
dtype: bool

In [298]: obj[mask]
Out[298]:
0    c
5    b
6    b
7    c
8    c
dtype: object
```

Related to `isin` is the `Index.get_indexer` method, which gives you an index array from an array of possibly nondistinct values into another array of distinct values:

```
In [299]: to_match = pd.Series(["c", "a", "b", "b", "c", "a"])

In [300]: unique_vals = pd.Series(["c", "b", "a"])

In [301]: indices = pd.Index(unique_vals).get_indexer(to_match)

In [302]: indices
Out[302]: array([0, 2, 1, 1, 0, 2])
```

See Table 5-9 for a reference on these methods.

Table 5-9. Unique, value counts, and set membership methods

Method	Description
`isin`	Compute a Boolean array indicating whether each Series or DataFrame value is contained in the passed sequence of values
`get_indexer`	Compute integer indices for each value in an array into another array of distinct values; helpful for data alignment and join-type operations
`unique`	Compute an array of unique values in a Series, returned in the order observed
`value_counts`	Return a Series containing unique values as its index and frequencies as its values, ordered count in descending order

In some cases, you may want to compute a histogram on multiple related columns in a DataFrame. Here's an example:

```
In [303]: data = pd.DataFrame({"Qu1": [1, 3, 4, 3, 4],
   .....:                      "Qu2": [2, 3, 1, 2, 3],
   .....:                      "Qu3": [1, 5, 2, 4, 4]})

In [304]: data
Out[304]:
   Qu1  Qu2  Qu3
0    1    2    1
1    3    3    5
2    4    1    2
3    3    2    4
4    4    3    4
```

We can compute the value counts for a single column, like so:

```
In [305]: data["Qu1"].value_counts().sort_index()
Out[305]:
1    1
3    2
4    2
Name: Qu1, dtype: int64
```

To compute this for all columns, pass `pandas.value_counts` to the DataFrame's `apply` method:

```
In [306]: result = data.apply(pd.value_counts).fillna(0)

In [307]: result
Out[307]:
   Qu1  Qu2  Qu3
1  1.0  1.0  1.0
2  0.0  2.0  1.0
3  2.0  2.0  0.0
4  2.0  0.0  2.0
5  0.0  0.0  1.0
```

Here, the row labels in the result are the distinct values occurring in all of the columns. The values are the respective counts of these values in each column.

There is also a `DataFrame.value_counts` method, but it computes counts considering each row of the DataFrame as a tuple to determine the number of occurrences of each distinct row:

```
In [308]: data = pd.DataFrame({"a": [1, 1, 1, 2, 2], "b": [0, 0, 1, 0, 0]})

In [309]: data
Out[309]:
   a  b
0  1  0
1  1  0
2  1  1
3  2  0
4  2  0

In [310]: data.value_counts()
Out[310]:
a  b
1  0    2
2  0    2
1  1    1
dtype: int64
```

In this case, the result has an index representing the distinct rows as a hierarchical index, a topic we will explore in greater detail in Chapter 8.

5.4 Conclusion

In the next chapter, we will discuss tools for reading (or *loading*) and writing datasets with pandas. After that, we will dig deeper into data cleaning, wrangling, analysis, and visualization tools using pandas.

CHAPTER 6

Data Loading, Storage, and File Formats

Reading data and making it accessible (often called *data loading*) is a necessary first step for using most of the tools in this book. The term *parsing* is also sometimes used to describe loading text data and interpreting it as tables and different data types. I'm going to focus on data input and output using pandas, though there are numerous tools in other libraries to help with reading and writing data in various formats.

Input and output typically fall into a few main categories: reading text files and other more efficient on-disk formats, loading data from databases, and interacting with network sources like web APIs.

6.1 Reading and Writing Data in Text Format

pandas features a number of functions for reading tabular data as a DataFrame object. Table 6-1 summarizes some of them; `pandas.read_csv` is one of the most frequently used in this book. We will look at binary data formats later in Section 6.2, "Binary Data Formats," on page 193.

Table 6-1. Text and binary data loading functions in pandas

Function	Description
`read_csv`	Load delimited data from a file, URL, or file-like object; use comma as default delimiter
`read_fwf`	Read data in fixed-width column format (i.e., no delimiters)
`read_clipboard`	Variation of `read_csv` that reads data from the clipboard; useful for converting tables from web pages
`read_excel`	Read tabular data from an Excel XLS or XLSX file
`read_hdf`	Read HDF5 files written by pandas
`read_html`	Read all tables found in the given HTML document
`read_json`	Read data from a JSON (JavaScript Object Notation) string representation, file, URL, or file-like object

Function	Description
`read_feather`	Read the Feather binary file format
`read_orc`	Read the Apache ORC binary file format
`read_parquet`	Read the Apache Parquet binary file format
`read_pickle`	Read an object stored by pandas using the Python pickle format
`read_sas`	Read a SAS dataset stored in one of the SAS system's custom storage formats
`read_spss`	Read a data file created by SPSS
`read_sql`	Read the results of a SQL query (using SQLAlchemy)
`read_sql_table`	Read a whole SQL table (using SQLAlchemy); equivalent to using a query that selects everything in that table using `read_sql`
`read_stata`	Read a dataset from Stata file format
`read_xml`	Read a table of data from an XML file

I'll give an overview of the mechanics of these functions, which are meant to convert text data into a DataFrame. The optional arguments for these functions may fall into a few categories:

Indexing
: Can treat one or more columns as the returned DataFrame, and whether to get column names from the file, arguments you provide, or not at all.

Type inference and data conversion
: Includes the user-defined value conversions and custom list of missing value markers.

Date and time parsing
: Includes a combining capability, including combining date and time information spread over multiple columns into a single column in the result.

Iterating
: Support for iterating over chunks of very large files.

Unclean data issues
: Includes skipping rows or a footer, comments, or other minor things like numeric data with thousands separated by commas.

Because of how messy data in the real world can be, some of the data loading functions (especially `pandas.read_csv`) have accumulated a long list of optional arguments over time. It's normal to feel overwhelmed by the number of different parameters (`pandas.read_csv` has around 50). The online pandas documentation has many examples about how each of these works, so if you're struggling to read a particular file, there might be a similar enough example to help you find the right parameters.

Some of these functions perform *type inference*, because the column data types are not part of the data format. That means you don't necessarily have to specify which columns are numeric, integer, Boolean, or string. Other data formats, like HDF5, ORC, and Parquet, have the data type information embedded in the format.

Handling dates and other custom types can require extra effort.

Let's start with a small comma-separated values (CSV) text file:

```
In [10]: !cat examples/ex1.csv
a,b,c,d,message
1,2,3,4,hello
5,6,7,8,world
9,10,11,12,foo
```

Here I used the Unix `cat` shell command to print the raw contents of the file to the screen. If you're on Windows, you can use `type` instead of `cat` to achieve the same effect within a Windows terminal (or command line).

Since this is comma-delimited, we can then use `pandas.read_csv` to read it into a DataFrame:

```
In [11]: df = pd.read_csv("examples/ex1.csv")

In [12]: df
Out[12]: 
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo
```

A file will not always have a header row. Consider this file:

```
In [13]: !cat examples/ex2.csv
1,2,3,4,hello
5,6,7,8,world
9,10,11,12,foo
```

To read this file, you have a couple of options. You can allow pandas to assign default column names, or you can specify names yourself:

```
In [14]: pd.read_csv("examples/ex2.csv", header=None)
Out[14]: 
   0   1   2   3      4
0  1   2   3   4  hello
1  5   6   7   8  world
2  9  10  11  12    foo

In [15]: pd.read_csv("examples/ex2.csv", names=["a", "b", "c", "d", "message"])
Out[15]: 
```

```
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo
```

Suppose you wanted the `message` column to be the index of the returned DataFrame. You can either indicate you want the column at index 4 or named `"message"` using the `index_col` argument:

```
In [16]: names = ["a", "b", "c", "d", "message"]

In [17]: pd.read_csv("examples/ex2.csv", names=names, index_col="message")
Out[17]: 
         a   b   c   d
message               
hello    1   2   3   4
world    5   6   7   8
foo      9  10  11  12
```

If you want to form a hierarchical index (discussed in Section 8.1, "Hierarchical Indexing," on page 247) from multiple columns, pass a list of column numbers or names:

```
In [18]: !cat examples/csv_mindex.csv
key1,key2,value1,value2
one,a,1,2
one,b,3,4
one,c,5,6
one,d,7,8
two,a,9,10
two,b,11,12
two,c,13,14
two,d,15,16

In [19]: parsed = pd.read_csv("examples/csv_mindex.csv",
   ....:                      index_col=["key1", "key2"])

In [20]: parsed
Out[20]: 
           value1  value2
key1 key2                
one  a          1       2
     b          3       4
     c          5       6
     d          7       8
two  a          9      10
     b         11      12
     c         13      14
     d         15      16
```

In some cases, a table might not have a fixed delimiter, using whitespace or some other pattern to separate fields. Consider a text file that looks like this:

```
In [21]: !cat examples/ex3.txt
A         B         C
aaa -0.264438 -1.026059 -0.619500
bbb  0.927272  0.302904 -0.032399
ccc -0.264273 -0.386314 -0.217601
ddd -0.871858 -0.348382  1.100491
```

While you could do some munging by hand, the fields here are separated by a variable amount of whitespace. In these cases, you can pass a regular expression as a delimiter for `pandas.read_csv`. This can be expressed by the regular expression `\s+`, so we have then:

```
In [22]: result = pd.read_csv("examples/ex3.txt", sep="\s+")

In [23]: result
Out[23]: 
            A         B         C
aaa -0.264438 -1.026059 -0.619500
bbb  0.927272  0.302904 -0.032399
ccc -0.264273 -0.386314 -0.217601
ddd -0.871858 -0.348382  1.100491
```

Because there was one fewer column name than the number of data rows, `pandas.read_csv` infers that the first column should be the DataFrame's index in this special case.

The file parsing functions have many additional arguments to help you handle the wide variety of exception file formats that occur (see a partial listing in Table 6-2). For example, you can skip the first, third, and fourth rows of a file with `skiprows`:

```
In [24]: !cat examples/ex4.csv
# hey!
a,b,c,d,message
# just wanted to make things more difficult for you
# who reads CSV files with computers, anyway?
1,2,3,4,hello
5,6,7,8,world
9,10,11,12,foo

In [25]: pd.read_csv("examples/ex4.csv", skiprows=[0, 2, 3])
Out[25]: 
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo
```

Handling missing values is an important and frequently nuanced part of the file reading process. Missing data is usually either not present (empty string) or marked by some *sentinel* (placeholder) value. By default, pandas uses a set of commonly occurring sentinels, such as NA and NULL:

```
In [26]: !cat examples/ex5.csv
something,a,b,c,d,message
one,1,2,3,4,NA
two,5,6,,8,world
three,9,10,11,12,foo
In [27]: result = pd.read_csv("examples/ex5.csv")

In [28]: result
Out[28]:
  something  a   b     c   d message
0       one  1   2   3.0   4     NaN
1       two  5   6   NaN   8   world
2     three  9  10  11.0  12     foo
```

Recall that pandas outputs missing values as NaN, so we have two null or missing values in result:

```
In [29]: pd.isna(result)
Out[29]:
   something      a      b      c      d  message
0      False  False  False  False  False     True
1      False  False  False   True  False    False
2      False  False  False  False  False    False
```

The na_values option accepts a sequence of strings to add to the default list of strings recognized as missing:

```
In [30]: result = pd.read_csv("examples/ex5.csv", na_values=["NULL"])

In [31]: result
Out[31]:
  something  a   b     c   d message
0       one  1   2   3.0   4     NaN
1       two  5   6   NaN   8   world
2     three  9  10  11.0  12     foo
```

pandas.read_csv has a list of many default NA value representations, but these defaults can be disabled with the keep_default_na option:

```
In [32]: result2 = pd.read_csv("examples/ex5.csv", keep_default_na=False)

In [33]: result2
Out[33]:
  something  a   b   c   d message
0       one  1   2   3   4      NA
1       two  5   6       8   world
2     three  9  10  11  12     foo

In [34]: result2.isna()
Out[34]:
```

```
   something      a      b      c      d  message
0      False  False  False  False  False    False
1      False  False  False  False  False    False
2      False  False  False  False  False    False

In [35]: result3 = pd.read_csv("examples/ex5.csv", keep_default_na=False,
   ....:                       na_values=["NA"])

In [36]: result3
Out[36]:
  something  a   b   c   d message
0       one  1   2   3   4     NaN
1       two  5   6       8   world
2     three  9  10  11  12     foo

In [37]: result3.isna()
Out[37]:
   something      a      b      c      d  message
0      False  False  False  False  False     True
1      False  False  False  False  False    False
2      False  False  False  False  False    False
```

Different NA sentinels can be specified for each column in a dictionary:

```
In [38]: sentinels = {"message": ["foo", "NA"], "something": ["two"]}

In [39]: pd.read_csv("examples/ex5.csv", na_values=sentinels,
   ....:             keep_default_na=False)
Out[39]:
  something  a   b   c   d message
0       one  1   2   3   4     NaN
1       NaN  5   6       8   world
2     three  9  10  11  12     NaN
```

Table 6-2 lists some frequently used options in `pandas.read_csv`.

Table 6-2. Some `pandas.read_csv` function arguments

Argument	Description
`path`	String indicating filesystem location, URL, or file-like object.
`sep` or `delimiter`	Character sequence or regular expression to use to split fields in each row.
`header`	Row number to use as column names; defaults to 0 (first row), but should be `None` if there is no header row.
`index_col`	Column numbers or names to use as the row index in the result; can be a single name/number or a list of them for a hierarchical index.
`names`	List of column names for result.
`skiprows`	Number of rows at beginning of file to ignore or list of row numbers (starting from 0) to skip.
`na_values`	Sequence of values to replace with NA. They are added to the default list unless `keep_default_na=False` is passed.
`keep_default_na`	Whether to use the default NA value list or not (`True` by default).

Argument	Description
`comment`	Character(s) to split comments off the end of lines.
`parse_dates`	Attempt to parse data to `datetime`; `False` by default. If `True`, will attempt to parse all columns. Otherwise, can specify a list of column numbers or names to parse. If element of list is tuple or list, will combine multiple columns together and parse to date (e.g., if date/time split across two columns).
`keep_date_col`	If joining columns to parse date, keep the joined columns; `False` by default.
`converters`	Dictionary containing column number or name mapping to functions (e.g., `{"foo": f}` would apply the function `f` to all values in the `"foo"` column).
`dayfirst`	When parsing potentially ambiguous dates, treat as international format (e.g., 7/6/2012 -> June 7, 2012); `False` by default.
`date_parser`	Function to use to parse dates.
`nrows`	Number of rows to read from beginning of file (not counting the header).
`iterator`	Return a `TextFileReader` object for reading the file piecemeal. This object can also be used with the `with` statement.
`chunksize`	For iteration, size of file chunks.
`skip_footer`	Number of lines to ignore at end of file.
`verbose`	Print various parsing information, like the time spent in each stage of the file conversion and memory use information.
`encoding`	Text encoding (e.g., `"utf-8` for UTF-8 encoded text). Defaults to `"utf-8"` if `None`.
`squeeze`	If the parsed data contains only one column, return a Series.
`thousands`	Separator for thousands (e.g., `","` or `"."`); default is `None`.
`decimal`	Decimal separator in numbers (e.g., `"."` or `","`); default is `"."`.
`engine`	CSV parsing and conversion engine to use; can be one of `"c"`, `"python"`, or `"pyarrow"`. The default is `"c"`, though the newer `"pyarrow"` engine can parse some files much faster. The `"python"` engine is slower but supports some features that the other engines do not.

Reading Text Files in Pieces

When processing very large files or figuring out the right set of arguments to correctly process a large file, you may want to read only a small piece of a file or iterate through smaller chunks of the file.

Before we look at a large file, we make the pandas display settings more compact:

```
In [40]: pd.options.display.max_rows = 10
```

Now we have:

```
In [41]: result = pd.read_csv("examples/ex6.csv")

In [42]: result
Out[42]:
          one       two     three      four key
0    0.467976 -0.038649 -0.295344 -1.824726   L
1   -0.358893  1.404453  0.704965 -0.200638   B
2   -0.501840  0.659254 -0.421691 -0.057688   G
```

```
3     0.204886  1.074134  1.388361 -0.982404   R
4     0.354628 -0.133116  0.283763 -0.837063   Q
...        ...       ...       ...       ...  ..
9995  2.311896 -0.417070 -1.409599 -0.515821   L
9996 -0.479893 -0.650419  0.745152 -0.646038   E
9997  0.523331  0.787112  0.486066  1.093156   K
9998 -0.362559  0.598894 -1.843201  0.887292   G
9999 -0.096376 -1.012999 -0.657431 -0.573315   0
[10000 rows x 5 columns]
```

The elipsis marks `...` indicate that rows in the middle of the DataFrame have been omitted.

If you want to read only a small number of rows (avoiding reading the entire file), specify that with `nrows`:

```
In [43]: pd.read_csv("examples/ex6.csv", nrows=5)
Out[43]:
        one       two     three      four key
0  0.467976 -0.038649 -0.295344 -1.824726   L
1 -0.358893  1.404453  0.704965 -0.200638   B
2 -0.501840  0.659254 -0.421691 -0.057688   G
3  0.204886  1.074134  1.388361 -0.982404   R
4  0.354628 -0.133116  0.283763 -0.837063   Q
```

To read a file in pieces, specify a `chunksize` as a number of rows:

```
In [44]: chunker = pd.read_csv("examples/ex6.csv", chunksize=1000)

In [45]: type(chunker)
Out[45]: pandas.io.parsers.readers.TextFileReader
```

The `TextFileReader` object returned by `pandas.read_csv` allows you to iterate over the parts of the file according to the `chunksize`. For example, we can iterate over `ex6.csv`, aggregating the value counts in the `"key"` column, like so:

```
chunker = pd.read_csv("examples/ex6.csv", chunksize=1000)

tot = pd.Series([], dtype='int64')
for piece in chunker:
    tot = tot.add(piece["key"].value_counts(), fill_value=0)

tot = tot.sort_values(ascending=False)
```

We have then:

```
In [47]: tot[:10]
Out[47]:
E    368.0
X    364.0
L    346.0
O    343.0
Q    340.0
M    338.0
```

```
J    337.0
F    335.0
K    334.0
H    330.0
dtype: float64
```

`TextFileReader` is also equipped with a `get_chunk` method that enables you to read pieces of an arbitrary size.

Writing Data to Text Format

Data can also be exported to a delimited format. Let's consider one of the CSV files read before:

```
In [48]: data = pd.read_csv("examples/ex5.csv")

In [49]: data
Out[49]:
  something  a   b     c   d message
0       one  1   2   3.0   4     NaN
1       two  5   6   NaN   8   world
2     three  9  10  11.0  12     foo
```

Using DataFrame's `to_csv` method, we can write the data out to a comma-separated file:

```
In [50]: data.to_csv("examples/out.csv")

In [51]: !cat examples/out.csv
,something,a,b,c,d,message
0,one,1,2,3.0,4,
1,two,5,6,,8,world
2,three,9,10,11.0,12,foo
```

Other delimiters can be used, of course (writing to `sys.stdout` so it prints the text result to the console rather than a file):

```
In [52]: import sys

In [53]: data.to_csv(sys.stdout, sep="|")
|something|a|b|c|d|message
0|one|1|2|3.0|4|
1|two|5|6||8|world
2|three|9|10|11.0|12|foo
```

Missing values appear as empty strings in the output. You might want to denote them by some other sentinel value:

```
In [54]: data.to_csv(sys.stdout, na_rep="NULL")
,something,a,b,c,d,message
0,one,1,2,3.0,4,NULL
1,two,5,6,NULL,8,world
2,three,9,10,11.0,12,foo
```

With no other options specified, both the row and column labels are written. Both of these can be disabled:

```
In [55]: data.to_csv(sys.stdout, index=False, header=False)
one,1,2,3.0,4,
two,5,6,,8,world
three,9,10,11.0,12,foo
```

You can also write only a subset of the columns, and in an order of your choosing:

```
In [56]: data.to_csv(sys.stdout, index=False, columns=["a", "b", "c"])
a,b,c
1,2,3.0
5,6,
9,10,11.0
```

Working with Other Delimited Formats

It's possible to load most forms of tabular data from disk using functions like `pandas.read_csv`. In some cases, however, some manual processing may be necessary. It's not uncommon to receive a file with one or more malformed lines that trip up `pandas.read_csv`. To illustrate the basic tools, consider a small CSV file:

```
In [57]: !cat examples/ex7.csv
"a","b","c"
"1","2","3"
"1","2","3"
```

For any file with a single-character delimiter, you can use Python's built-in `csv` module. To use it, pass any open file or file-like object to `csv.reader`:

```
In [58]: import csv

In [59]: f = open("examples/ex7.csv")

In [60]: reader = csv.reader(f)
```

Iterating through the reader like a file yields lists of values with any quote characters removed:

```
In [61]: for line in reader:
   ....:     print(line)
['a', 'b', 'c']
['1', '2', '3']
['1', '2', '3']

In [62]: f.close()
```

From there, it's up to you to do the wrangling necessary to put the data in the form that you need. Let's take this step by step. First, we read the file into a list of lines:

```
In [63]: with open("examples/ex7.csv") as f:
   ....:     lines = list(csv.reader(f))
```

Then we split the lines into the header line and the data lines:

```
In [64]: header, values = lines[0], lines[1:]
```

Then we can create a dictionary of data columns using a dictionary comprehension and the expression `zip(*values)` (beware that this will use a lot of memory on large files), which transposes rows to columns:

```
In [65]: data_dict = {h: v for h, v in zip(header, zip(*values))}

In [66]: data_dict
Out[66]: {'a': ('1', '1'), 'b': ('2', '2'), 'c': ('3', '3')}
```

CSV files come in many different flavors. To define a new format with a different delimiter, string quoting convention, or line terminator, we could define a simple subclass of `csv.Dialect`:

```
class my_dialect(csv.Dialect):
    lineterminator = "\n"
    delimiter = ";"
    quotechar = '"'
    quoting = csv.QUOTE_MINIMAL

reader = csv.reader(f, dialect=my_dialect)
```

We could also give individual CSV dialect parameters as keywords to `csv.reader` without having to define a subclass:

```
reader = csv.reader(f, delimiter="|")
```

The possible options (attributes of `csv.Dialect`) and what they do can be found in Table 6-3.

Table 6-3. CSV `dialect` options

Argument	Description
`delimiter`	One-character string to separate fields; defaults to `","`.
`lineterminator`	Line terminator for writing; defaults to `"\r\n"`. Reader ignores this and recognizes cross-platform line terminators.
`quotechar`	Quote character for fields with special characters (like a delimiter); default is `'"'`.
`quoting`	Quoting convention. Options include `csv.QUOTE_ALL` (quote all fields), `csv.QUOTE_MINIMAL` (only fields with special characters like the delimiter), `csv.QUOTE_NONNUMERIC`, and `csv.QUOTE_NONE` (no quoting). See Python's documentation for full details. Defaults to `QUOTE_MINIMAL`.
`skipinitialspace`	Ignore whitespace after each delimiter; default is `False`.
`doublequote`	How to handle quoting character inside a field; if `True`, it is doubled (see online documentation for full detail and behavior).
`escapechar`	String to escape the delimiter if `quoting` is set to `csv.QUOTE_NONE`; disabled by default.

> For files with more complicated or fixed multicharacter delimiters, you will not be able to use the `csv` module. In those cases, you'll have to do the line splitting and other cleanup using the string's `split` method or the regular expression method `re.split`. Thankfully, `pandas.read_csv` is capable of doing almost anything you need if you pass the necessary options, so you only rarely will have to parse files by hand.

To *write* delimited files manually, you can use `csv.writer`. It accepts an open, writable file object and the same dialect and format options as `csv.reader`:

```
with open("mydata.csv", "w") as f:
    writer = csv.writer(f, dialect=my_dialect)
    writer.writerow(("one", "two", "three"))
    writer.writerow(("1", "2", "3"))
    writer.writerow(("4", "5", "6"))
    writer.writerow(("7", "8", "9"))
```

JSON Data

JSON (short for JavaScript Object Notation) has become one of the standard formats for sending data by HTTP request between web browsers and other applications. It is a much more free-form data format than a tabular text form like CSV. Here is an example:

```
obj = """
{"name": "Wes",
 "cities_lived": ["Akron", "Nashville", "New York", "San Francisco"],
 "pet": null,
 "siblings": [{"name": "Scott", "age": 34, "hobbies": ["guitars", "soccer"]},
              {"name": "Katie", "age": 42, "hobbies": ["diving", "art"]}]
}
"""
```

JSON is very nearly valid Python code with the exception of its null value `null` and some other nuances (such as disallowing trailing commas at the end of lists). The basic types are objects (dictionaries), arrays (lists), strings, numbers, Booleans, and nulls. All of the keys in an object must be strings. There are several Python libraries for reading and writing JSON data. I'll use `json` here, as it is built into the Python standard library. To convert a JSON string to Python form, use `json.loads`:

```
In [68]: import json

In [69]: result = json.loads(obj)

In [70]: result
Out[70]:
{'name': 'Wes',
 'cities_lived': ['Akron', 'Nashville', 'New York', 'San Francisco'],
```

```
 'pet': None,
 'siblings': [{'name': 'Scott',
   'age': 34,
   'hobbies': ['guitars', 'soccer']},
  {'name': 'Katie', 'age': 42, 'hobbies': ['diving', 'art']}]}
```

`json.dumps`, on the other hand, converts a Python object back to JSON:

```
In [71]: asjson = json.dumps(result)

In [72]: asjson
Out[72]: '{"name": "Wes", "cities_lived": ["Akron", "Nashville", "New York", "San
 Francisco"], "pet": null, "siblings": [{"name": "Scott", "age": 34, "hobbies": [
"guitars", "soccer"]}, {"name": "Katie", "age": 42, "hobbies": ["diving", "art"]}
]}'
```

How you convert a JSON object or list of objects to a DataFrame or some other data structure for analysis will be up to you. Conveniently, you can pass a list of dictionaries (which were previously JSON objects) to the DataFrame constructor and select a subset of the data fields:

```
In [73]: siblings = pd.DataFrame(result["siblings"], columns=["name", "age"])

In [74]: siblings
Out[74]:
    name  age
0  Scott   34
1  Katie   42
```

The `pandas.read_json` can automatically convert JSON datasets in specific arrangements into a Series or DataFrame. For example:

```
In [75]: !cat examples/example.json
[{"a": 1, "b": 2, "c": 3},
 {"a": 4, "b": 5, "c": 6},
 {"a": 7, "b": 8, "c": 9}]
```

The default options for `pandas.read_json` assume that each object in the JSON array is a row in the table:

```
In [76]: data = pd.read_json("examples/example.json")

In [77]: data
Out[77]:
   a  b  c
0  1  2  3
1  4  5  6
2  7  8  9
```

For an extended example of reading and manipulating JSON data (including nested records), see the USDA food database example in Chapter 13.

If you need to export data from pandas to JSON, one way is to use the `to_json` methods on Series and DataFrame:

```
In [78]: data.to_json(sys.stdout)
{"a":{"0":1,"1":4,"2":7},"b":{"0":2,"1":5,"2":8},"c":{"0":3,"1":6,"2":9}}
In [79]: data.to_json(sys.stdout, orient="records")
[{"a":1,"b":2,"c":3},{"a":4,"b":5,"c":6},{"a":7,"b":8,"c":9}]
```

XML and HTML: Web Scraping

Python has many libraries for reading and writing data in the ubiquitous HTML and XML formats. Examples include lxml, Beautiful Soup, and html5lib. While lxml is comparatively much faster in general, the other libraries can better handle malformed HTML or XML files.

pandas has a built-in function, `pandas.read_html`, which uses all of these libraries to automatically parse tables out of HTML files as DataFrame objects. To show how this works, I downloaded an HTML file (used in the pandas documentation) from the US FDIC showing bank failures.[1] First, you must install some additional libraries used by `read_html`:

```
conda install lxml beautifulsoup4 html5lib
```

If you are not using conda, `pip install lxml` should also work.

The `pandas.read_html` function has a number of options, but by default it searches for and attempts to parse all tabular data contained within `<table>` tags. The result is a list of DataFrame objects:

```
In [80]: tables = pd.read_html("examples/fdic_failed_bank_list.html")

In [81]: len(tables)
Out[81]: 1

In [82]: failures = tables[0]

In [83]: failures.head()
Out[83]: 
                      Bank Name             City  ST   CERT  \
0                   Allied Bank         Mulberry  AR     91   
1  The Woodbury Banking Company         Woodbury  GA  11297   
2        First CornerStone Bank  King of Prussia  PA  35312   
3            Trust Company Bank          Memphis  TN   9956   
4    North Milwaukee State Bank        Milwaukee  WI  20364   
                Acquiring Institution        Closing Date       Updated Date  
0                         Today's Bank  September 23, 2016  November 17, 2016  
1                          United Bank     August 19, 2016  November 17, 2016  
```

1 For the full list, see *https://www.fdic.gov/bank/individual/failed/banklist.html*.

```
2  First-Citizens Bank & Trust Company        May 6, 2016  September 6, 2016
3           The Bank of Fayette County      April 29, 2016  September 6, 2016
4  First-Citizens Bank & Trust Company      March 11, 2016      June 16, 2016
```

Because `failures` has many columns, pandas inserts a line break character `\`.

As you will learn in later chapters, from here we could proceed to do some data cleaning and analysis, like computing the number of bank failures by year:

```
In [84]: close_timestamps = pd.to_datetime(failures["Closing Date"])

In [85]: close_timestamps.dt.year.value_counts()
Out[85]:
2010    157
2009    140
2011     92
2012     51
2008     25
       ...
2004      4
2001      4
2007      3
2003      3
2000      2
Name: Closing Date, Length: 15, dtype: int64
```

Parsing XML with lxml.objectify

XML is another common structured data format supporting hierarchical, nested data with metadata. The book you are currently reading was actually created from a series of large XML documents.

Earlier, I showed the `pandas.read_html` function, which uses either lxml or Beautiful Soup under the hood to parse data from HTML. XML and HTML are structurally similar, but XML is more general. Here, I will show an example of how to use lxml to parse data from a more general XML format.

For many years, the New York Metropolitan Transportation Authority (MTA) published a number of data series about its bus and train services in XML format. Here we'll look at the performance data, which is contained in a set of XML files. Each train or bus service has a different file (like *Performance_MNR.xml* for the Metro-North Railroad) containing monthly data as a series of XML records that look like this:

```
<INDICATOR>
  <INDICATOR_SEQ>373889</INDICATOR_SEQ>
  <PARENT_SEQ></PARENT_SEQ>
  <AGENCY_NAME>Metro-North Railroad</AGENCY_NAME>
  <INDICATOR_NAME>Escalator Availability</INDICATOR_NAME>
  <DESCRIPTION>Percent of the time that escalators are operational
  systemwide. The availability rate is based on physical observations performed
  the morning of regular business days only. This is a new indicator the agency
```

```
began reporting in 2009.</DESCRIPTION>
  <PERIOD_YEAR>2011</PERIOD_YEAR>
  <PERIOD_MONTH>12</PERIOD_MONTH>
  <CATEGORY>Service Indicators</CATEGORY>
  <FREQUENCY>M</FREQUENCY>
  <DESIRED_CHANGE>U</DESIRED_CHANGE>
  <INDICATOR_UNIT>%</INDICATOR_UNIT>
  <DECIMAL_PLACES>1</DECIMAL_PLACES>
  <YTD_TARGET>97.00</YTD_TARGET>
  <YTD_ACTUAL></YTD_ACTUAL>
  <MONTHLY_TARGET>97.00</MONTHLY_TARGET>
  <MONTHLY_ACTUAL></MONTHLY_ACTUAL>
</INDICATOR>
```

Using `lxml.objectify`, we parse the file and get a reference to the root node of the XML file with `getroot`:

```
In [86]: from lxml import objectify

In [87]: path = "datasets/mta_perf/Performance_MNR.xml"

In [88]: with open(path) as f:
   ....:     parsed = objectify.parse(f)

In [89]: root = parsed.getroot()
```

`root.INDICATOR` returns a generator yielding each `<INDICATOR>` XML element. For each record, we can populate a dictionary of tag names (like `YTD_ACTUAL`) to data values (excluding a few tags) by running the following code:

```
data = []

skip_fields = ["PARENT_SEQ", "INDICATOR_SEQ",
               "DESIRED_CHANGE", "DECIMAL_PLACES"]

for elt in root.INDICATOR:
    el_data = {}
    for child in elt.getchildren():
        if child.tag in skip_fields:
            continue
        el_data[child.tag] = child.pyval
    data.append(el_data)
```

Lastly, convert this list of dictionaries into a DataFrame:

```
In [91]: perf = pd.DataFrame(data)

In [92]: perf.head()
Out[92]:
            AGENCY_NAME                        INDICATOR_NAME  \
0  Metro-North Railroad  On-Time Performance (West of Hudson)
1  Metro-North Railroad  On-Time Performance (West of Hudson)
2  Metro-North Railroad  On-Time Performance (West of Hudson)
```

```
3  Metro-North Railroad  On-Time Performance (West of Hudson)
4  Metro-North Railroad  On-Time Performance (West of Hudson)
                                                         DESCRIPTION  \
0  Percent of commuter trains that arrive at their destinations within 5 m...
1  Percent of commuter trains that arrive at their destinations within 5 m...
2  Percent of commuter trains that arrive at their destinations within 5 m...
3  Percent of commuter trains that arrive at their destinations within 5 m...
4  Percent of commuter trains that arrive at their destinations within 5 m...
   PERIOD_YEAR  PERIOD_MONTH            CATEGORY FREQUENCY INDICATOR_UNIT  \
0         2008             1  Service Indicators         M              %
1         2008             2  Service Indicators         M              %
2         2008             3  Service Indicators         M              %
3         2008             4  Service Indicators         M              %
4         2008             5  Service Indicators         M              %
  YTD_TARGET YTD_ACTUAL MONTHLY_TARGET MONTHLY_ACTUAL
0       95.0       96.9           95.0           96.9
1       95.0       96.0           95.0           95.0
2       95.0       96.3           95.0           96.9
3       95.0       96.8           95.0           98.3
4       95.0       96.6           95.0           95.8
```

pandas's `pandas.read_xml` function turns this process into a one-line expression:

```
In [93]: perf2 = pd.read_xml(path)

In [94]: perf2.head()
Out[94]:
   INDICATOR_SEQ  PARENT_SEQ           AGENCY_NAME  \
0          28445         NaN  Metro-North Railroad
1          28445         NaN  Metro-North Railroad
2          28445         NaN  Metro-North Railroad
3          28445         NaN  Metro-North Railroad
4          28445         NaN  Metro-North Railroad
                        INDICATOR_NAME  \
0  On-Time Performance (West of Hudson)
1  On-Time Performance (West of Hudson)
2  On-Time Performance (West of Hudson)
3  On-Time Performance (West of Hudson)
4  On-Time Performance (West of Hudson)
                                                         DESCRIPTION  \
0  Percent of commuter trains that arrive at their destinations within 5 m...
1  Percent of commuter trains that arrive at their destinations within 5 m...
2  Percent of commuter trains that arrive at their destinations within 5 m...
3  Percent of commuter trains that arrive at their destinations within 5 m...
4  Percent of commuter trains that arrive at their destinations within 5 m...
   PERIOD_YEAR  PERIOD_MONTH            CATEGORY FREQUENCY DESIRED_CHANGE  \
0         2008             1  Service Indicators         M              U
1         2008             2  Service Indicators         M              U
2         2008             3  Service Indicators         M              U
3         2008             4  Service Indicators         M              U
4         2008             5  Service Indicators         M              U
  INDICATOR_UNIT  DECIMAL_PLACES YTD_TARGET YTD_ACTUAL MONTHLY_TARGET  \
0              %               1      95.00      96.90          95.00
```

```
1             %                1      95.00      96.00          95.00
2             %                1      95.00      96.30          95.00
3             %                1      95.00      96.80          95.00
4             %                1      95.00      96.60          95.00
  MONTHLY_ACTUAL
0          96.90
1          95.00
2          96.90
3          98.30
4          95.80
```

For more complex XML documents, refer to the docstring for `pandas.read_xml` which describes how to do selections and filters to extract a particular table of interest.

6.2 Binary Data Formats

One simple way to store (or *serialize*) data in binary format is using Python's built-in `pickle` module. pandas objects all have a `to_pickle` method that writes the data to disk in pickle format:

```
In [95]: frame = pd.read_csv("examples/ex1.csv")

In [96]: frame
Out[96]:
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo

In [97]: frame.to_pickle("examples/frame_pickle")
```

Pickle files are in general readable only in Python. You can read any "pickled" object stored in a file by using the built-in `pickle` directly, or even more conveniently using `pandas.read_pickle`:

```
In [98]: pd.read_pickle("examples/frame_pickle")
Out[98]:
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo
```

`pickle` is recommended only as a short-term storage format. The problem is that it is hard to guarantee that the format will be stable over time; an object pickled today may not unpickle with a later version of a library. pandas has tried to maintain backward compatibility when possible, but at some point in the future it may be necessary to "break" the pickle format.

pandas has built-in support for several other open source binary data formats, such as HDF5, ORC, and Apache Parquet. For example, if you install the pyarrow package (conda install pyarrow), then you can read Parquet files with pandas.read_parquet:

```
In [100]: fec = pd.read_parquet('datasets/fec/fec.parquet')
```

I will give some HDF5 examples in "Using HDF5 Format" on page 195. I encourage you to explore different file formats to see how fast they are and how well they work for your analysis.

Reading Microsoft Excel Files

pandas also supports reading tabular data stored in Excel 2003 (and higher) files using either the pandas.ExcelFile class or pandas.read_excel function. Internally, these tools use the add-on packages xlrd and openpyxl to read old-style XLS and newer XLSX files, respectively. These must be installed separately from pandas using pip or conda:

```
conda install openpyxl xlrd
```

To use pandas.ExcelFile, create an instance by passing a path to an xls or xlsx file:

```
In [101]: xlsx = pd.ExcelFile("examples/ex1.xlsx")
```

This object can show you the list of available sheet names in the file:

```
In [102]: xlsx.sheet_names
Out[102]: ['Sheet1']
```

Data stored in a sheet can then be read into DataFrame with parse:

```
In [103]: xlsx.parse(sheet_name="Sheet1")
Out[103]:
   Unnamed: 0  a   b   c   d message
0           0  1   2   3   4   hello
1           1  5   6   7   8   world
2           2  9  10  11  12     foo
```

This Excel table has an index column, so we can indicate that with the index_col argument:

```
In [104]: xlsx.parse(sheet_name="Sheet1", index_col=0)
Out[104]:
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo
```

If you are reading multiple sheets in a file, then it is faster to create the pandas.ExcelFile, but you can also simply pass the filename to pandas.read_excel:

```
In [105]: frame = pd.read_excel("examples/ex1.xlsx", sheet_name="Sheet1")

In [106]: frame
Out[106]:
   Unnamed: 0  a   b   c   d message
0           0  1   2   3   4   hello
1           1  5   6   7   8   world
2           2  9  10  11  12     foo
```

To write pandas data to Excel format, you must first create an `ExcelWriter`, then write data to it using the pandas object's `to_excel` method:

```
In [107]: writer = pd.ExcelWriter("examples/ex2.xlsx")

In [108]: frame.to_excel(writer, "Sheet1")

In [109]: writer.save()
```

You can also pass a file path to `to_excel` and avoid the `ExcelWriter`:

```
In [110]: frame.to_excel("examples/ex2.xlsx")
```

Using HDF5 Format

HDF5 is a respected file format intended for storing large quantities of scientific array data. It is available as a C library, and it has interfaces available in many other languages, including Java, Julia, MATLAB, and Python. The "HDF" in HDF5 stands for *hierarchical data format*. Each HDF5 file can store multiple datasets and supporting metadata. Compared with simpler formats, HDF5 supports on-the-fly compression with a variety of compression modes, enabling data with repeated patterns to be stored more efficiently. HDF5 can be a good choice for working with datasets that don't fit into memory, as you can efficiently read and write small sections of much larger arrays.

To get started with HDF5 and pandas, you must first install PyTables by installing the `tables` package with conda:

```
conda install pytables
```

> Note that the PyTables package is called "tables" in PyPI, so if you install with pip you will have to run `pip install tables`.

While it's possible to directly access HDF5 files using either the PyTables or h5py libraries, pandas provides a high-level interface that simplifies storing Series and DataFrame objects. The `HDFStore` class works like a dictionary and handles the low-level details:

```
In [113]: frame = pd.DataFrame({"a": np.random.standard_normal(100)})

In [114]: store = pd.HDFStore("examples/mydata.h5")

In [115]: store["obj1"] = frame

In [116]: store["obj1_col"] = frame["a"]

In [117]: store
Out[117]:
<class 'pandas.io.pytables.HDFStore'>
File path: examples/mydata.h5
```

Objects contained in the HDF5 file can then be retrieved with the same dictionary-like API:

```
In [118]: store["obj1"]
Out[118]:
           a
0  -0.204708
1   0.478943
2  -0.519439
3  -0.555730
4   1.965781
..       ...
95  0.795253
96  0.118110
97 -0.748532
98  0.584970
99  0.152677
[100 rows x 1 columns]
```

`HDFStore` supports two storage schemas, `"fixed"` and `"table"` (the default is `"fixed"`). The latter is generally slower, but it supports query operations using a special syntax:

```
In [119]: store.put("obj2", frame, format="table")

In [120]: store.select("obj2", where=["index >= 10 and index <= 15"])
Out[120]:
           a
10  1.007189
11 -1.296221
12  0.274992
13  0.228913
14  1.352917
15  0.886429

In [121]: store.close()
```

The `put` is an explicit version of the `store["obj2"] = frame` method but allows us to set other options like the storage format.

The pandas.read_hdf function gives you a shortcut to these tools:

```
In [122]: frame.to_hdf("examples/mydata.h5", "obj3", format="table")

In [123]: pd.read_hdf("examples/mydata.h5", "obj3", where=["index < 5"])
Out[123]: 
          a
0 -0.204708
1  0.478943
2 -0.519439
3 -0.555730
4  1.965781
```

If you'd like, you can delete the HDF5 file you created, like so:

```
In [124]: import os

In [125]: os.remove("examples/mydata.h5")
```

If you are processing data that is stored on remote servers, like Amazon S3 or HDFS, using a different binary format designed for distributed storage like Apache Parquet may be more suitable.

If you work with large quantities of data locally, I would encourage you to explore PyTables and h5py to see how they can suit your needs. Since many data analysis problems are I/O-bound (rather than CPU-bound), using a tool like HDF5 can massively accelerate your applications.

HDF5 is *not* a database. It is best suited for write-once, read-many datasets. While data can be added to a file at any time, if multiple writers do so simultaneously, the file can become corrupted.

6.3 Interacting with Web APIs

Many websites have public APIs providing data feeds via JSON or some other format. There are a number of ways to access these APIs from Python; one method that I recommend is the requests package, which can be installed with pip or conda:

```
conda install requests
```

To find the last 30 GitHub issues for pandas on GitHub, we can make a GET HTTP request using the add-on requests library:

```
In [126]: import requests

In [127]: url = "https://api.github.com/repos/pandas-dev/pandas/issues"
```

```
In [128]: resp = requests.get(url)

In [129]: resp.raise_for_status()

In [130]: resp
Out[130]: <Response [200]>
```

It's a good practice to always call `raise_for_status` after using `requests.get` to check for HTTP errors.

The response object's `json` method will return a Python object containing the parsed JSON data as a dictionary or list (depending on what JSON is returned):

```
In [131]: data = resp.json()

In [132]: data[0]["title"]
Out[132]: 'REF: make copy keyword non-stateful'
```

Since the results retrieved are based on real-time data, what you see when you run this code will almost definitely be different.

Each element in `data` is a dictionary containing all of the data found on a GitHub issue page (except for the comments). We can pass `data` directly to `pandas.DataFrame` and extract fields of interest:

```
In [133]: issues = pd.DataFrame(data, columns=["number", "title",
   .....:                                      "labels", "state"])

In [134]: issues
Out[134]:
    number  \
0    48062
1    48061
2    48060
3    48059
4    48058
..     ...
25   48032
26   48030
27   48028
28   48027
29   48026
                                                                 title  \
0                                  REF: make copy keyword non-stateful
1                                                STYLE: upgrade flake8
2   DOC: "Creating a Python environment" in "Creating a development environ...
3                                     REGR: Avoid overflow with groupby sum
4   REGR: fix reset_index (Index.insert) regression with custom Index subcl...
..                                                                     ...
25                     BUG: Union of multi index with EA types can lose EA dtype
26                                                  ENH: Add rolling.prod()
```

```
27                                            CLN: Refactor groupby's _make_wrapper
28                                              ENH: Support masks in groupby prod
29                                                DEP: Add pip to environment.yml
                                                                         labels  \
0                                                                            []
1   [{'id': 106935113, 'node_id': 'MDU6TGFiZWwxMDY5MzUxMTM=', 'url': 'https...
2   [{'id': 134699, 'node_id': 'MDU6TGFiZWwxMzQ2OTk=', 'url': 'https://api....
3   [{'id': 233160, 'node_id': 'MDU6TGFiZWwyMzMxNjA=', 'url': 'https://api....
4   [{'id': 32815646, 'node_id': 'MDU6TGFiZWwzMjgxNTY0Ng==', 'url': 'https:...
..                                                                          ...
25  [{'id': 76811, 'node_id': 'MDU6TGFiZWw3NjgxMQ==', 'url': 'https://api.g...
26  [{'id': 76812, 'node_id': 'MDU6TGFiZWw3NjgxMg==', 'url': 'https://api.g...
27  [{'id': 233160, 'node_id': 'MDU6TGFiZWwyMzMxNjA=', 'url': 'https://api....
28  [{'id': 233160, 'node_id': 'MDU6TGFiZWwyMzMxNjA=', 'url': 'https://api....
29  [{'id': 76811, 'node_id': 'MDU6TGFiZWw3NjgxMQ==', 'url': 'https://api.g...
   state
0   open
1   open
2   open
3   open
4   open
..   ...
25  open
26  open
27  open
28  open
29  open
[30 rows x 4 columns]
```

With a bit of elbow grease, you can create some higher-level interfaces to common web APIs that return DataFrame objects for more convenient analysis.

6.4 Interacting with Databases

In a business setting, a lot of data may not be stored in text or Excel files. SQL-based relational databases (such as SQL Server, PostgreSQL, and MySQL) are in wide use, and many alternative databases have become quite popular. The choice of database is usually dependent on the performance, data integrity, and scalability needs of an application.

pandas has some functions to simplify loading the results of a SQL query into a DataFrame. As an example, I'll create a SQLite3 database using Python's built-in `sqlite3` driver:

```
In [135]: import sqlite3

In [136]: query = """
   .....: CREATE TABLE test
   .....: (a VARCHAR(20), b VARCHAR(20),
   .....:  c REAL,        d INTEGER
```

```
   .....: );"""

In [137]: con = sqlite3.connect("mydata.sqlite")

In [138]: con.execute(query)
Out[138]: <sqlite3.Cursor at 0x7fdfd73b69c0>

In [139]: con.commit()
```

Then, insert a few rows of data:

```
In [140]: data = [("Atlanta", "Georgia", 1.25, 6),
   .....:         ("Tallahassee", "Florida", 2.6, 3),
   .....:         ("Sacramento", "California", 1.7, 5)]

In [141]: stmt = "INSERT INTO test VALUES(?, ?, ?, ?)"

In [142]: con.executemany(stmt, data)
Out[142]: <sqlite3.Cursor at 0x7fdfd73a00c0>

In [143]: con.commit()
```

Most Python SQL drivers return a list of tuples when selecting data from a table:

```
In [144]: cursor = con.execute("SELECT * FROM test")

In [145]: rows = cursor.fetchall()

In [146]: rows
Out[146]: 
[('Atlanta', 'Georgia', 1.25, 6),
 ('Tallahassee', 'Florida', 2.6, 3),
 ('Sacramento', 'California', 1.7, 5)]
```

You can pass the list of tuples to the DataFrame constructor, but you also need the column names, contained in the cursor's `description` attribute. Note that for SQLite3, the cursor `description` only provides column names (the other fields, which are part of Python's Database API specification, are `None`), but for some other database drivers, more column information is provided:

```
In [147]: cursor.description
Out[147]: 
(('a', None, None, None, None, None, None),
 ('b', None, None, None, None, None, None),
 ('c', None, None, None, None, None, None),
 ('d', None, None, None, None, None, None))

In [148]: pd.DataFrame(rows, columns=[x[0] for x in cursor.description])
Out[148]: 
             a           b     c  d
0      Atlanta     Georgia  1.25  6
1  Tallahassee     Florida  2.60  3
2   Sacramento  California  1.70  5
```

This is quite a bit of munging that you'd rather not repeat each time you query the database. The SQLAlchemy project is a popular Python SQL toolkit that abstracts away many of the common differences between SQL databases. pandas has a `read_sql` function that enables you to read data easily from a general SQLAlchemy connection. You can install SQLAlchemy with conda like so:

```
conda install sqlalchemy
```

Now, we'll connect to the same SQLite database with SQLAlchemy and read data from the table created before:

```
In [149]: import sqlalchemy as sqla

In [150]: db = sqla.create_engine("sqlite:///mydata.sqlite")

In [151]: pd.read_sql("SELECT * FROM test", db)
Out[151]:
             a           b     c  d
0      Atlanta     Georgia  1.25  6
1  Tallahassee     Florida  2.60  3
2   Sacramento  California  1.70  5
```

6.5 Conclusion

Getting access to data is frequently the first step in the data analysis process. We have looked at a number of useful tools in this chapter that should help you get started. In the upcoming chapters we will dig deeper into data wrangling, data visualization, time series analysis, and other topics.

CHAPTER 7
Data Cleaning and Preparation

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

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

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

7.1 Handling Missing Data

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

The way that missing data is represented in pandas objects is somewhat imperfect, but it is sufficient for most real-world use. For data with `float64` dtype, pandas uses the floating-point value `NaN` (Not a Number) to represent missing data.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Table 7-1. NA handling object methods

Method	Description
`dropna`	Filter axis labels based on whether values for each label have missing data, with varying thresholds for how much missing data to tolerate.
`fillna`	Fill in missing data with some value or using an interpolation method such as `"ffill"` or `"bfill"`.
`isna`	Return Boolean values indicating which values are missing/NA.
`notna`	Negation of `isna`, returns `True` for non-NA values and `False` for NA values.

Filtering Out Missing Data

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

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

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

This is the same thing as doing:

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

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

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

In [27]: data
Out[27]: 
     0    1    2
0  1.0  6.5  3.0
1  1.0  NaN  NaN
2  NaN  NaN  NaN
3  NaN  6.5  3.0

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

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

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

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

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

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

In [31]: data
Out[31]: 
     0    1    2   4
0  1.0  6.5  3.0 NaN
1  1.0  NaN  NaN NaN
2  NaN  NaN  NaN NaN
3  NaN  6.5  3.0 NaN

In [32]: data.dropna(axis="columns", how="all")
Out[32]: 
     0    1    2
0  1.0  6.5  3.0
1  1.0  NaN  NaN
2  NaN  NaN  NaN
3  NaN  6.5  3.0
```

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

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

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

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

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

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

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

Filling In Missing Data

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Table 7-2. `fillna` function arguments

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

7.2 Data Transformation

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

Removing Duplicates

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

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

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

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

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

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

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

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

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

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

In [55]: data.drop_duplicates(subset=["k1"])
Out[55]:
    k1  k2  v1
0  one   1   0
1  two   1   1
```

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

```
In [56]: data.drop_duplicates(["k1", "k2"], keep="last")
Out[56]:
    k1  k2  v1
0  one   1   0
1  two   1   1
2  one   2   2
3  two   3   3
4  one   3   4
6  two   4   6
```

Transforming Data Using a Function or Mapping

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

```
In [57]: data = pd.DataFrame({"food": ["bacon", "pulled pork", "bacon",
   ....:                               "pastrami", "corned beef", "bacon",
   ....:                               "pastrami", "honey ham", "nova lox"],
   ....:                      "ounces": [4, 3, 12, 6, 7.5, 8, 3, 5, 6]})

In [58]: data
Out[58]:
          food  ounces
0        bacon     4.0
1  pulled pork     3.0
2        bacon    12.0
3     pastrami     6.0
4  corned beef     7.5
5        bacon     8.0
6     pastrami     3.0
7    honey ham     5.0
8     nova lox     6.0
```

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

```
meat_to_animal = {
  "bacon": "pig",
  "pulled pork": "pig",
  "pastrami": "cow",
  "corned beef": "cow",
  "honey ham": "pig",
  "nova lox": "salmon"
}
```

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

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

In [61]: data
Out[61]:
          food  ounces  animal
0        bacon     4.0     pig
1  pulled pork     3.0     pig
2        bacon    12.0     pig
3     pastrami     6.0     cow
4  corned beef     7.5     cow
5        bacon     8.0     pig
6     pastrami     3.0     cow
7    honey ham     5.0     pig
8     nova lox     6.0  salmon
```

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

```
In [62]: def get_animal(x):
   ....:     return meat_to_animal[x]

In [63]: data["food"].map(get_animal)
Out[63]:
0       pig
1       pig
2       pig
3       cow
4       cow
5       pig
6       cow
7       pig
8    salmon
Name: food, dtype: object
```

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

Replacing Values

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

```
In [64]: data = pd.Series([1., -999., 2., -999., -1000., 3.])

In [65]: data
Out[65]:
0       1.0
```

```
1    -999.0
2       2.0
3    -999.0
4   -1000.0
5       3.0
dtype: float64
```

The `-999` values might be sentinel values for missing data. To replace these with NA values that pandas understands, we can use `replace`, producing a new Series:

```
In [66]: data.replace(-999, np.nan)
Out[66]:
0       1.0
1       NaN
2       2.0
3       NaN
4   -1000.0
5       3.0
dtype: float64
```

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

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

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

```
In [68]: data.replace([-999, -1000], [np.nan, 0])
Out[68]:
0    1.0
1    NaN
2    2.0
3    NaN
4    0.0
5    3.0
dtype: float64
```

The argument passed can also be a dictionary:

```
In [69]: data.replace({-999: np.nan, -1000: 0})
Out[69]:
0    1.0
1    NaN
2    2.0
3    NaN
4    0.0
```

```
5    3.0
dtype: float64
```

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

Renaming Axis Indexes

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

```
In [70]: data = pd.DataFrame(np.arange(12).reshape((3, 4)),
   ....:                     index=["Ohio", "Colorado", "New York"],
   ....:                     columns=["one", "two", "three", "four"])
```

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

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

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

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

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

In [74]: data
Out[74]: 
      one  two  three  four
OHIO    0    1      2     3
COLO    4    5      6     7
NEW     8    9     10    11
```

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

```
In [75]: data.rename(index=str.title, columns=str.upper)
Out[75]: 
      ONE  TWO  THREE  FOUR
Ohio    0    1      2     3
Colo    4    5      6     7
New     8    9     10    11
```

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

```
In [76]: data.rename(index={"OHIO": "INDIANA"},
   ....:             columns={"three": "peekaboo"})
Out[76]: 
         one  two  peekaboo  four
INDIANA    0    1         2     3
COLO       4    5         6     7
NEW        8    9        10    11
```

`rename` saves you from the chore of copying the DataFrame manually and assigning new values to its `index` and `columns` attributes.

Discretization and Binning

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

```
In [77]: ages = [20, 22, 25, 27, 21, 23, 37, 31, 61, 45, 41, 32]
```

Let's divide these into bins of 18 to 25, 26 to 35, 36 to 60, and finally 61 and older. To do so, you have to use `pandas.cut`:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
In [88]: data = np.random.uniform(size=20)

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

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

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

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

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

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

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

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

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

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

Detecting and Filtering Outliers

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

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

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

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

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

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

In [98]: col[col.abs() > 3]
Out[98]:
41    -3.399312
136   -3.745356
Name: 2, dtype: float64
```

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

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

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

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

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

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

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

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

```
In [102]: np.sign(data).head()
Out[102]:
     0    1    2    3
0 -1.0  1.0 -1.0  1.0
1  1.0 -1.0  1.0 -1.0
2  1.0  1.0  1.0 -1.0
3 -1.0 -1.0  1.0 -1.0
4 -1.0  1.0 -1.0 -1.0
```

Permutation and Random Sampling

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

```
In [103]: df = pd.DataFrame(np.arange(5 * 7).reshape((5, 7)))

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

In [114]: choices.sample(n=10, replace=True)
Out[114]: 
2   -1
0    5
3    6
1    7
4    4
0    5
4    4
0    5
4    4
4    4
dtype: int64
```

Computing Indicator/Dummy Variables

Another type of transformation for statistical modeling or machine learning applications is converting a categorical variable into a *dummy* or *indicator* matrix. If a column in a DataFrame has `k` distinct values, you would derive a matrix or DataFrame with `k` columns containing all 1s and 0s. pandas has a `pandas.get_dummies` function for doing this, though you could also devise one yourself. Let's consider an example DataFrame:

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

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

In [117]: pd.get_dummies(df["key"])
Out[117]: 
   a  b  c
0  0  1  0
1  0  1  0
2  1  0  0
3  0  0  1
4  1  0  0
5  0  1  0
```

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

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

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

In [120]: df_with_dummy
Out[120]: 
   data1  key_a  key_b  key_c
0      0      0      1      0
1      1      0      1      0
2      2      1      0      0
3      3      0      0      1
4      4      1      0      0
5      5      0      1      0
```

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

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

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

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

In [123]: movies[:10]
Out[123]:
   movie_id                               title                        genres
0         1                    Toy Story (1995)   Animation|Children's|Comedy
1         2                      Jumanji (1995)  Adventure|Children's|Fantasy
2         3             Grumpier Old Men (1995)                Comedy|Romance
3         4            Waiting to Exhale (1995)                  Comedy|Drama
4         5  Father of the Bride Part II (1995)                        Comedy
5         6                         Heat (1995)         Action|Crime|Thriller
6         7                      Sabrina (1995)                Comedy|Romance
7         8                 Tom and Huck (1995)          Adventure|Children's
8         9                 Sudden Death (1995)                        Action
9        10                    GoldenEye (1995)     Action|Adventure|Thriller
```

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

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

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

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

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

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

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

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

A useful recipe for statistical applications is to combine `pandas.get_dummies` with a discretization function like `pandas.cut`:

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

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

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

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

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

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

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

7.3 Extension Data Types

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Table 7-3. pandas extension data types

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

7.4 String Manipulation

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

Python Built-In String Object Methods

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

```
In [151]: val = "a,b,  guido"

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

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

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

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

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.

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']
```

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," on page 224.

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" on page 242.

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

```
In [219]: dict(enumerate(c.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 469 ms, sys: 106 ms, total: 574 ms
Wall time: 577 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()
840 ms +- 10.9 ms per loop (mean +- std. dev. of 7 runs, 1 loop each)

In [250]: %timeit categories.value_counts()
30.1 ms +- 549 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.

CHAPTER 8
Data Wrangling: Join, Combine, and Reshape

In many applications, data may be spread across a number of files or databases, or be arranged in a form that is not convenient to analyze. This chapter focuses on tools to help combine, join, and rearrange data.

First, I introduce the concept of *hierarchical indexing* in pandas, which is used extensively in some of these operations. I then dig into the particular data manipulations. You can see various applied usages of these tools in Chapter 13.

8.1 Hierarchical Indexing

Hierarchical indexing is an important feature of pandas that enables you to have multiple (two or more) index *levels* on an axis. Another way of thinking about it is that it provides a way for you to work with higher dimensional data in a lower dimensional form. Let's start with a simple example: create a Series with a list of lists (or arrays) as the index:

```
In [11]: data = pd.Series(np.random.uniform(size=9),
   ....:                  index=[["a", "a", "a", "b", "b", "c", "c", "d", "d"],
   ....:                         [1, 2, 3, 1, 3, 1, 2, 2, 3]])

In [12]: data
Out[12]:
a  1    0.929616
   2    0.316376
   3    0.183919
b  1    0.204560
   3    0.567725
c  1    0.595545
   2    0.964515
```

```
d  2    0.653177
   3    0.748907
dtype: float64
```

What you're seeing is a prettified view of a Series with a `MultiIndex` as its index. The "gaps" in the index display mean "use the label directly above":

```
In [13]: data.index
Out[13]:
MultiIndex([('a', 1),
            ('a', 2),
            ('a', 3),
            ('b', 1),
            ('b', 3),
            ('c', 1),
            ('c', 2),
            ('d', 2),
            ('d', 3)],
           )
```

With a hierarchically indexed object, so-called *partial* indexing is possible, enabling you to concisely select subsets of the data:

```
In [14]: data["b"]
Out[14]:
1    0.204560
3    0.567725
dtype: float64

In [15]: data["b":"c"]
Out[15]:
b  1    0.204560
   3    0.567725
c  1    0.595545
   2    0.964515
dtype: float64

In [16]: data.loc[["b", "d"]]
Out[16]:
b  1    0.204560
   3    0.567725
d  2    0.653177
   3    0.748907
dtype: float64
```

Selection is even possible from an "inner" level. Here I select all of the values having the value `2` from the second index level:

```
In [17]: data.loc[:, 2]
Out[17]:
a    0.316376
c    0.964515
```

```
d    0.653177
dtype: float64
```

Hierarchical indexing plays an important role in reshaping data and in group-based operations like forming a pivot table. For example, you can rearrange this data into a DataFrame using its `unstack` method:

```
In [18]: data.unstack()
Out[18]:
          1         2         3
a  0.929616  0.316376  0.183919
b  0.204560       NaN  0.567725
c  0.595545  0.964515       NaN
d       NaN  0.653177  0.748907
```

The inverse operation of `unstack` is `stack`:

```
In [19]: data.unstack().stack()
Out[19]:
a  1    0.929616
   2    0.316376
   3    0.183919
b  1    0.204560
   3    0.567725
c  1    0.595545
   2    0.964515
d  2    0.653177
   3    0.748907
dtype: float64
```

`stack` and `unstack` will be explored in more detail later in Section 8.3, "Reshaping and Pivoting," on page 270.

With a DataFrame, either axis can have a hierarchical index:

```
In [20]: frame = pd.DataFrame(np.arange(12).reshape((4, 3)),
   ....:                      index=[["a", "a", "b", "b"], [1, 2, 1, 2]],
   ....:                      columns=[["Ohio", "Ohio", "Colorado"],
   ....:                               ["Green", "Red", "Green"]])

In [21]: frame
Out[21]:
     Ohio     Colorado
    Green Red    Green
a 1     0   1        2
  2     3   4        5
b 1     6   7        8
  2     9  10       11
```

The hierarchical levels can have names (as strings or any Python objects). If so, these will show up in the console output:

```
In [22]: frame.index.names = ["key1", "key2"]
```

```
In [23]: frame.columns.names = ["state", "color"]

In [24]: frame
Out[24]: 
state      Ohio     Colorado
color     Green Red    Green
key1 key2                   
a    1        0   1        2
     2        3   4        5
b    1        6   7        8
     2        9  10       11
```

These names supersede the `name` attribute, which is used only with single-level indexes.

> Be careful to note that the index names `"state"` and `"color"` are not part of the row labels (the `frame.index` values).

You can see how many levels an index has by accessing its `nlevels` attribute:

```
In [25]: frame.index.nlevels
Out[25]: 2
```

With partial column indexing you can similarly select groups of columns:

```
In [26]: frame["Ohio"]
Out[26]: 
color      Green  Red
key1 key2            
a    1         0    1
     2         3    4
b    1         6    7
     2         9   10
```

A `MultiIndex` can be created by itself and then reused; the columns in the preceding DataFrame with level names could also be created like this:

```
pd.MultiIndex.from_arrays([["Ohio", "Ohio", "Colorado"],
                          ["Green", "Red", "Green"]],
                          names=["state", "color"])
```

Reordering and Sorting Levels

At times you may need to rearrange the order of the levels on an axis or sort the data by the values in one specific level. The `swaplevel` method takes two level numbers or names and returns a new object with the levels interchanged (but the data is otherwise unaltered):

```
In [27]: frame.swaplevel("key1", "key2")
Out[27]: 
state      Ohio     Colorado
color     Green Red    Green
key2 key1                   
1    a        0   1        2
2    a        3   4        5
1    b        6   7        8
2    b        9  10       11
```

`sort_index` by default sorts the data lexicographically using all the index levels, but you can choose to use only a single level or a subset of levels to sort by passing the `level` argument. For example:

```
In [28]: frame.sort_index(level=1)
Out[28]: 
state      Ohio     Colorado
color     Green Red    Green
key1 key2                   
a    1        0   1        2
b    1        6   7        8
a    2        3   4        5
b    2        9  10       11

In [29]: frame.swaplevel(0, 1).sort_index(level=0)
Out[29]: 
state      Ohio     Colorado
color     Green Red    Green
key2 key1                   
1    a        0   1        2
     b        6   7        8
2    a        3   4        5
     b        9  10       11
```

Data selection performance is much better on hierarchically indexed objects if the index is lexicographically sorted starting with the outermost level—that is, the result of calling `sort_index(level=0)` or `sort_index()`.

Summary Statistics by Level

Many descriptive and summary statistics on DataFrame and Series have a `level` option in which you can specify the level you want to aggregate by on a particular axis. Consider the above DataFrame; we can aggregate by level on either the rows or columns, like so:

```
In [30]: frame.groupby(level="key2").sum()
Out[30]: 
state  Ohio     Colorado
color Green Red    Green
```

```
key2
1          6   8        10
2         12  14        16

In [31]: frame.groupby(level="color", axis="columns").sum()
Out[31]: 
color      Green  Red
key1 key2            
a    1         2    1
     2         8    4
b    1        14    7
     2        20   10
```

We will discuss groupby in much more detail later in Chapter 10.

Indexing with a DataFrame's columns

It's not unusual to want to use one or more columns from a DataFrame as the row index; alternatively, you may wish to move the row index into the DataFrame's columns. Here's an example DataFrame:

```
In [32]: frame = pd.DataFrame({"a": range(7), "b": range(7, 0, -1),
   ....:                       "c": ["one", "one", "one", "two", "two",
   ....:                             "two", "two"],
   ....:                       "d": [0, 1, 2, 0, 1, 2, 3]})

In [33]: frame
Out[33]: 
   a  b    c  d
0  0  7  one  0
1  1  6  one  1
2  2  5  one  2
3  3  4  two  0
4  4  3  two  1
5  5  2  two  2
6  6  1  two  3
```

DataFrame's set_index function will create a new DataFrame using one or more of its columns as the index:

```
In [34]: frame2 = frame.set_index(["c", "d"])

In [35]: frame2
Out[35]: 
       a  b
c   d      
one 0  0  7
    1  1  6
    2  2  5
two 0  3  4
    1  4  3
```

```
2  5  2
3  6  1
```

By default, the columns are removed from the DataFrame, though you can leave them in by passing `drop=False` to `set_index`:

```
In [36]: frame.set_index(["c", "d"], drop=False)
Out[36]:
       a  b    c  d
c   d
one 0  0  7  one  0
    1  1  6  one  1
    2  2  5  one  2
two 0  3  4  two  0
    1  4  3  two  1
    2  5  2  two  2
    3  6  1  two  3
```

`reset_index`, on the other hand, does the opposite of `set_index`; the hierarchical index levels are moved into the columns:

```
In [37]: frame2.reset_index()
Out[37]:
     c  d  a  b
0  one  0  0  7
1  one  1  1  6
2  one  2  2  5
3  two  0  3  4
4  two  1  4  3
5  two  2  5  2
6  two  3  6  1
```

8.2 Combining and Merging Datasets

Data contained in pandas objects can be combined in a number of ways:

`pandas.merge`
: Connect rows in DataFrames based on one or more keys. This will be familiar to users of SQL or other relational databases, as it implements database *join* operations.

`pandas.concat`
: Concatenate or "stack" objects together along an axis.

`combine_first`
: Splice together overlapping data to fill in missing values in one object with values from another.

I will address each of these and give a number of examples. They'll be utilized in examples throughout the rest of the book.

Database-Style DataFrame Joins

Merge or *join* operations combine datasets by linking rows using one or more *keys*. These operations are particularly important in relational databases (e.g., SQL-based). The `pandas.merge` function in pandas is the main entry point for using these algorithms on your data.

Let's start with a simple example:

```
In [38]: df1 = pd.DataFrame({"key": ["b", "b", "a", "c", "a", "a", "b"],
   ....:                     "data1": pd.Series(range(7), dtype="Int64")})

In [39]: df2 = pd.DataFrame({"key": ["a", "b", "d"],
   ....:                     "data2": pd.Series(range(3), dtype="Int64")})

In [40]: df1
Out[40]: 
  key  data1
0   b      0
1   b      1
2   a      2
3   c      3
4   a      4
5   a      5
6   b      6

In [41]: df2
Out[41]: 
  key  data2
0   a      0
1   b      1
2   d      2
```

Here I am using pandas's `Int64` extension type for nullable integers, discussed in Section 7.3, "Extension Data Types," on page 224.

This is an example of a *many-to-one* join; the data in `df1` has multiple rows labeled `a` and `b`, whereas `df2` has only one row for each value in the `key` column. Calling `pandas.merge` with these objects, we obtain:

```
In [42]: pd.merge(df1, df2)
Out[42]: 
  key  data1  data2
0   b      0      1
1   b      1      1
2   b      6      1
3   a      2      0
4   a      4      0
5   a      5      0
```

Note that I didn't specify which column to join on. If that information is not specified, `pandas.merge` uses the overlapping column names as the keys. It's a good practice to specify explicitly, though:

```
In [43]: pd.merge(df1, df2, on="key")
Out[43]:
  key  data1  data2
0   b      0      1
1   b      1      1
2   b      6      1
3   a      2      0
4   a      4      0
5   a      5      0
```

In general, the order of column output in `pandas.merge` operations is unspecified.

If the column names are different in each object, you can specify them separately:

```
In [44]: df3 = pd.DataFrame({"lkey": ["b", "b", "a", "c", "a", "a", "b"],
   ....:                     "data1": pd.Series(range(7), dtype="Int64")})

In [45]: df4 = pd.DataFrame({"rkey": ["a", "b", "d"],
   ....:                     "data2": pd.Series(range(3), dtype="Int64")})

In [46]: pd.merge(df3, df4, left_on="lkey", right_on="rkey")
Out[46]:
  lkey  data1 rkey  data2
0    b      0    b      1
1    b      1    b      1
2    b      6    b      1
3    a      2    a      0
4    a      4    a      0
5    a      5    a      0
```

You may notice that the `"c"` and `"d"` values and associated data are missing from the result. By default, `pandas.merge` does an `"inner"` join; the keys in the result are the intersection, or the common set found in both tables. Other possible options are `"left"`, `"right"`, and `"outer"`. The outer join takes the union of the keys, combining the effect of applying both left and right joins:

```
In [47]: pd.merge(df1, df2, how="outer")
Out[47]:
  key  data1  data2
0   b      0      1
1   b      1      1
2   b      6      1
3   a      2      0
4   a      4      0
5   a      5      0
6   c      3   <NA>
7   d   <NA>      2
```

```
In [48]: pd.merge(df3, df4, left_on="lkey", right_on="rkey", how="outer")
Out[48]: 
  lkey  data1 rkey  data2
0    b      0    b      1
1    b      1    b      1
2    b      6    b      1
3    a      2    a      0
4    a      4    a      0
5    a      5    a      0
6    c      3  NaN   <NA>
7  NaN   <NA>    d      2
```

In an outer join, rows from the left or right DataFrame objects that do not match on keys in the other DataFrame will appear with NA values in the other DataFrame's columns for the nonmatching rows.

See Table 8-1 for a summary of the options for `how`.

Table 8-1. Different join types with the `how` argument

Option	Behavior
`how="inner"`	Use only the key combinations observed in both tables
`how="left"`	Use all key combinations found in the left table
`how="right"`	Use all key combinations found in the right table
`how="outer"`	Use all key combinations observed in both tables together

Many-to-many merges form the Cartesian product of the matching keys. Here's an example:

```
In [49]: df1 = pd.DataFrame({"key": ["b", "b", "a", "c", "a", "b"],
   ....:                     "data1": pd.Series(range(6), dtype="Int64")})

In [50]: df2 = pd.DataFrame({"key": ["a", "b", "a", "b", "d"],
   ....:                     "data2": pd.Series(range(5), dtype="Int64")})

In [51]: df1
Out[51]: 
  key  data1
0   b      0
1   b      1
2   a      2
3   c      3
4   a      4
5   b      5

In [52]: df2
Out[52]: 
  key  data2
0   a      0
1   b      1
2   a      2
```

```
3   b       3
4   d       4

In [53]: pd.merge(df1, df2, on="key", how="left")
Out[53]:
   key  data1  data2
0    b      0      1
1    b      0      3
2    b      1      1
3    b      1      3
4    a      2      0
5    a      2      2
6    c      3   <NA>
7    a      4      0
8    a      4      2
9    b      5      1
10   b      5      3
```

Since there were three "b" rows in the left DataFrame and two in the right one, there are six "b" rows in the result. The join method passed to the `how` keyword argument affects only the distinct key values appearing in the result:

```
In [54]: pd.merge(df1, df2, how="inner")
Out[54]:
  key  data1  data2
0   b      0      1
1   b      0      3
2   b      1      1
3   b      1      3
4   b      5      1
5   b      5      3
6   a      2      0
7   a      2      2
8   a      4      0
9   a      4      2
```

To merge with multiple keys, pass a list of column names:

```
In [55]: left = pd.DataFrame({"key1": ["foo", "foo", "bar"],
   ....:                      "key2": ["one", "two", "one"],
   ....:                      "lval": pd.Series([1, 2, 3], dtype='Int64')})

In [56]: right = pd.DataFrame({"key1": ["foo", "foo", "bar", "bar"],
   ....:                       "key2": ["one", "one", "one", "two"],
   ....:                       "rval": pd.Series([4, 5, 6, 7], dtype='Int64')})

In [57]: pd.merge(left, right, on=["key1", "key2"], how="outer")
Out[57]:
  key1 key2  lval  rval
0  foo  one     1     4
1  foo  one     1     5
2  foo  two     2  <NA>
```

```
3  bar  one     3     6
4  bar  two  <NA>     7
```

To determine which key combinations will appear in the result depending on the choice of merge method, think of the multiple keys as forming an array of tuples to be used as a single join key.

> When you're joining columns on columns, the indexes on the passed DataFrame objects are discarded. If you need to preserve the index values, you can use `reset_index` to append the index to the columns.

A last issue to consider in merge operations is the treatment of overlapping column names. For example:

```
In [58]: pd.merge(left, right, on="key1")
Out[58]:
  key1 key2_x  lval key2_y  rval
0  foo    one     1    one     4
1  foo    one     1    one     5
2  foo    two     2    one     4
3  foo    two     2    one     5
4  bar    one     3    one     6
5  bar    one     3    two     7
```

While you can address the overlap manually (see the section "Renaming Axis Indexes" on page 214 for renaming axis labels), `pandas.merge` has a `suffixes` option for specifying strings to append to overlapping names in the left and right DataFrame objects:

```
In [59]: pd.merge(left, right, on="key1", suffixes=("_left", "_right"))
Out[59]:
  key1 key2_left  lval key2_right  rval
0  foo       one     1        one     4
1  foo       one     1        one     5
2  foo       two     2        one     4
3  foo       two     2        one     5
4  bar       one     3        one     6
5  bar       one     3        two     7
```

See Table 8-2 for an argument reference on `pandas.merge`. The next section covers joining using the DataFrame's row index.

Table 8-2. `pandas.merge` function arguments

Argument	Description
`left`	DataFrame to be merged on the left side.
`right`	DataFrame to be merged on the right side.
`how`	Type of join to apply: one of `"inner"`, `"outer"`, `"left"`, or `"right"`; defaults to `"inner"`.

Argument	Description
on	Column names to join on. Must be found in both DataFrame objects. If not specified and no other join keys given, will use the intersection of the column names in `left` and `right` as the join keys.
left_on	Columns in `left` DataFrame to use as join keys. Can be a single column name or a list of column names.
right_on	Analogous to `left_on` for `right` DataFrame.
left_index	Use row index in `left` as its join key (or keys, if a `MultiIndex`).
right_index	Analogous to `left_index`.
sort	Sort merged data lexicographically by join keys; `False` by default.
suffixes	Tuple of string values to append to column names in case of overlap; defaults to `("_x", "_y")` (e.g., if `"data"` in both DataFrame objects, would appear as `"data_x"` and `"data_y"` in result).
copy	If `False`, avoid copying data into resulting data structure in some exceptional cases; by default always copies.
validate	Verifies if the merge is of the specified type, whether one-to-one, one-to-many, or many-to-many. See the docstring for full details on the options.
indicator	Adds a special column `_merge` that indicates the source of each row; values will be `"left_only"`, `"right_only"`, or `"both"` based on the origin of the joined data in each row.

Merging on Index

In some cases, the merge key(s) in a DataFrame will be found in its index (row labels). In this case, you can pass `left_index=True` or `right_index=True` (or both) to indicate that the index should be used as the merge key:

```
In [60]: left1 = pd.DataFrame({"key": ["a", "b", "a", "a", "b", "c"],
   ....:                       "value": pd.Series(range(6), dtype="Int64")})

In [61]: right1 = pd.DataFrame({"group_val": [3.5, 7]}, index=["a", "b"])

In [62]: left1
Out[62]:
  key  value
0   a      0
1   b      1
2   a      2
3   a      3
4   b      4
5   c      5

In [63]: right1
Out[63]:
   group_val
a        3.5
b        7.0

In [64]: pd.merge(left1, right1, left_on="key", right_index=True)
Out[64]:
  key  value  group_val
0   a      0        3.5
```

```
2   a       2          3.5
3   a       3          3.5
1   b       1          7.0
4   b       4          7.0
```

> If you look carefully here, you will see that the index values for `left1` have been preserved, whereas in other examples above, the indexes of the input DataFrame objects are dropped. Because the index of `right1` is unique, this "many-to-one" merge (with the default `how="inner"` method) can preserve the index values from `left1` that correspond to rows in the output.

Since the default merge method is to intersect the join keys, you can instead form the union of them with an outer join:

```
In [65]: pd.merge(left1, right1, left_on="key", right_index=True, how="outer")
Out[65]:
  key  value  group_val
0   a      0        3.5
2   a      2        3.5
3   a      3        3.5
1   b      1        7.0
4   b      4        7.0
5   c      5        NaN
```

With hierarchically indexed data, things are more complicated, as joining on index is equivalent to a multiple-key merge:

```
In [66]: lefth = pd.DataFrame({"key1": ["Ohio", "Ohio", "Ohio",
   ....:                                "Nevada", "Nevada"],
   ....:                       "key2": [2000, 2001, 2002, 2001, 2002],
   ....:                       "data": pd.Series(range(5), dtype="Int64")})

In [67]: righth_index = pd.MultiIndex.from_arrays(
   ....:     [
   ....:         ["Nevada", "Nevada", "Ohio", "Ohio", "Ohio", "Ohio"],
   ....:         [2001, 2000, 2000, 2000, 2001, 2002]
   ....:     ]
   ....: )

In [68]: righth = pd.DataFrame({"event1": pd.Series([0, 2, 4, 6, 8, 10], dtype="I
nt64",
   ....:                                            index=righth_index),
   ....:                        "event2": pd.Series([1, 3, 5, 7, 9, 11], dtype="I
nt64",
   ....:                                            index=righth_index)})

In [69]: lefth
Out[69]:
     key1  key2  data
0    Ohio  2000     0
```

```
1     Ohio  2001     1
2     Ohio  2002     2
3   Nevada  2001     3
4   Nevada  2002     4

In [70]: righth
Out[70]:
             event1  event2
Nevada 2001       0       1
       2000       2       3
Ohio   2000       4       5
       2000       6       7
       2001       8       9
       2002      10      11
```

In this case, you have to indicate multiple columns to merge on as a list (note the handling of duplicate index values with `how="outer"`):

```
In [71]: pd.merge(lefth, righth, left_on=["key1", "key2"], right_index=True)
Out[71]:
     key1  key2  data  event1  event2
0    Ohio  2000     0       4       5
0    Ohio  2000     0       6       7
1    Ohio  2001     1       8       9
2    Ohio  2002     2      10      11
3  Nevada  2001     3       0       1

In [72]: pd.merge(lefth, righth, left_on=["key1", "key2"],
   ....:          right_index=True, how="outer")
Out[72]:
     key1  key2  data  event1  event2
0    Ohio  2000     0       4       5
0    Ohio  2000     0       6       7
1    Ohio  2001     1       8       9
2    Ohio  2002     2      10      11
3  Nevada  2001     3       0       1
4  Nevada  2002     4    <NA>    <NA>
4  Nevada  2000  <NA>       2       3
```

Using the indexes of both sides of the merge is also possible:

```
In [73]: left2 = pd.DataFrame([[1., 2.], [3., 4.], [5., 6.]],
   ....:                      index=["a", "c", "e"],
   ....:                      columns=["Ohio", "Nevada"]).astype("Int64")

In [74]: right2 = pd.DataFrame([[7., 8.], [9., 10.], [11., 12.], [13, 14]],
   ....:                       index=["b", "c", "d", "e"],
   ....:                       columns=["Missouri", "Alabama"]).astype("Int64")

In [75]: left2
Out[75]:
   Ohio  Nevada
a     1       2
```

```
c     3       4
e     5       6

In [76]: right2
Out[76]:
   Missouri  Alabama
b         7        8
c         9       10
d        11       12
e        13       14

In [77]: pd.merge(left2, right2, how="outer", left_index=True, right_index=True)
Out[77]:
   Ohio  Nevada  Missouri  Alabama
a     1       2      <NA>     <NA>
b  <NA>    <NA>         7        8
c     3       4         9       10
d  <NA>    <NA>        11       12
e     5       6        13       14
```

DataFrame has a `join` instance method to simplify merging by index. It can also be used to combine many DataFrame objects having the same or similar indexes but nonoverlapping columns. In the prior example, we could have written:

```
In [78]: left2.join(right2, how="outer")
Out[78]:
   Ohio  Nevada  Missouri  Alabama
a     1       2      <NA>     <NA>
b  <NA>    <NA>         7        8
c     3       4         9       10
d  <NA>    <NA>        11       12
e     5       6        13       14
```

Compared with `pandas.merge`, DataFrame's `join` method performs a left join on the join keys by default. It also supports joining the index of the passed DataFrame on one of the columns of the calling DataFrame:

```
In [79]: left1.join(right1, on="key")
Out[79]:
  key  value  group_val
0   a      0        3.5
1   b      1        7.0
2   a      2        3.5
3   a      3        3.5
4   b      4        7.0
5   c      5        NaN
```

You can think of this method as joining data "into" the object whose `join` method was called.

Lastly, for simple index-on-index merges, you can pass a list of DataFrames to `join` as an alternative to using the more general `pandas.concat` function described in the next section:

```
In [80]: another = pd.DataFrame([[7., 8.], [9., 10.], [11., 12.], [16., 17.]],
   ....:                        index=["a", "c", "e", "f"],
   ....:                        columns=["New York", "Oregon"])

In [81]: another
Out[81]:
   New York  Oregon
a       7.0     8.0
c       9.0    10.0
e      11.0    12.0
f      16.0    17.0

In [82]: left2.join([right2, another])
Out[82]:
   Ohio  Nevada  Missouri  Alabama  New York  Oregon
a     1       2      <NA>     <NA>       7.0     8.0
c     3       4         9       10       9.0    10.0
e     5       6        13       14      11.0    12.0

In [83]: left2.join([right2, another], how="outer")
Out[83]:
   Ohio  Nevada  Missouri  Alabama  New York  Oregon
a     1       2      <NA>     <NA>       7.0     8.0
c     3       4         9       10       9.0    10.0
e     5       6        13       14      11.0    12.0
b  <NA>    <NA>         7        8       NaN     NaN
d  <NA>    <NA>        11       12       NaN     NaN
f  <NA>    <NA>      <NA>     <NA>      16.0    17.0
```

Concatenating Along an Axis

Another kind of data combination operation is referred to interchangeably as *concatenation* or *stacking*. NumPy's `concatenate` function can do this with NumPy arrays:

```
In [84]: arr = np.arange(12).reshape((3, 4))

In [85]: arr
Out[85]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])

In [86]: np.concatenate([arr, arr], axis=1)
Out[86]:
array([[ 0,  1,  2,  3,  0,  1,  2,  3],
       [ 4,  5,  6,  7,  4,  5,  6,  7],
       [ 8,  9, 10, 11,  8,  9, 10, 11]])
```

In the context of pandas objects such as Series and DataFrame, having labeled axes enable you to further generalize array concatenation. In particular, you have a number of additional concerns:

- If the objects are indexed differently on the other axes, should we combine the distinct elements in these axes or use only the values in common?
- Do the concatenated chunks of data need to be identifiable as such in the resulting object?
- Does the "concatenation axis" contain data that needs to be preserved? In many cases, the default integer labels in a DataFrame are best discarded during concatenation.

The `concat` function in pandas provides a consistent way to address each of these questions. I'll give a number of examples to illustrate how it works. Suppose we have three Series with no index overlap:

```
In [87]: s1 = pd.Series([0, 1], index=["a", "b"], dtype="Int64")

In [88]: s2 = pd.Series([2, 3, 4], index=["c", "d", "e"], dtype="Int64")

In [89]: s3 = pd.Series([5, 6], index=["f", "g"], dtype="Int64")
```

Calling `pandas.concat` with these objects in a list glues together the values and indexes:

```
In [90]: s1
Out[90]:
a    0
b    1
dtype: Int64

In [91]: s2
Out[91]:
c    2
d    3
e    4
dtype: Int64

In [92]: s3
Out[92]:
f    5
g    6
dtype: Int64

In [93]: pd.concat([s1, s2, s3])
Out[93]:
a    0
b    1
c    2
```

```
d    3
e    4
f    5
g    6
dtype: Int64
```

By default, `pandas.concat` works along `axis="index"`, producing another Series. If you pass `axis="columns"`, the result will instead be a DataFrame:

```
In [94]: pd.concat([s1, s2, s3], axis="columns")
Out[94]:
      0     1     2
a     0  <NA>  <NA>
b     1  <NA>  <NA>
c  <NA>     2  <NA>
d  <NA>     3  <NA>
e  <NA>     4  <NA>
f  <NA>  <NA>     5
g  <NA>  <NA>     6
```

In this case there is no overlap on the other axis, which as you can see is the union (the `"outer"` join) of the indexes. You can instead intersect them by passing `join="inner"`:

```
In [95]: s4 = pd.concat([s1, s3])

In [96]: s4
Out[96]:
a    0
b    1
f    5
g    6
dtype: Int64

In [97]: pd.concat([s1, s4], axis="columns")
Out[97]:
      0  1
a     0  0
b     1  1
f  <NA>  5
g  <NA>  6

In [98]: pd.concat([s1, s4], axis="columns", join="inner")
Out[98]:
   0  1
a  0  0
b  1  1
```

In this last example, the `"f"` and `"g"` labels disappeared because of the `join="inner"` option.

A potential issue is that the concatenated pieces are not identifiable in the result. Suppose instead you wanted to create a hierarchical index on the concatenation axis. To do this, use the `keys` argument:

```
In [99]: result = pd.concat([s1, s1, s3], keys=["one", "two", "three"])

In [100]: result
Out[100]:
one    a    0
       b    1
two    a    0
       b    1
three  f    5
       g    6
dtype: Int64

In [101]: result.unstack()
Out[101]:
          a     b     f     g
one       0     1  <NA>  <NA>
two       0     1  <NA>  <NA>
three  <NA>  <NA>     5     6
```

In the case of combining Series along `axis="columns"`, the `keys` become the DataFrame column headers:

```
In [102]: pd.concat([s1, s2, s3], axis="columns", keys=["one", "two", "three"])
Out[102]:
    one   two  three
a     0  <NA>   <NA>
b     1  <NA>   <NA>
c  <NA>     2   <NA>
d  <NA>     3   <NA>
e  <NA>     4   <NA>
f  <NA>  <NA>      5
g  <NA>  <NA>      6
```

The same logic extends to DataFrame objects:

```
In [103]: df1 = pd.DataFrame(np.arange(6).reshape(3, 2), index=["a", "b", "c"],
   .....:                    columns=["one", "two"])

In [104]: df2 = pd.DataFrame(5 + np.arange(4).reshape(2, 2), index=["a", "c"],
   .....:                    columns=["three", "four"])

In [105]: df1
Out[105]:
   one  two
a    0    1
b    2    3
c    4    5

In [106]: df2
```

```
Out[106]:
   three  four
a      5     6
c      7     8

In [107]: pd.concat([df1, df2], axis="columns", keys=["level1", "level2"])
Out[107]:
  level1     level2
     one two  three four
a      0   1    5.0  6.0
b      2   3    NaN  NaN
c      4   5    7.0  8.0
```

Here the keys argument is used to create a hierarchical index where the first level can be used to identify each of the concatenated DataFrame objects.

If you pass a dictionary of objects instead of a list, the dictionary's keys will be used for the keys option:

```
In [108]: pd.concat({"level1": df1, "level2": df2}, axis="columns")
Out[108]:
  level1     level2
     one two  three four
a      0   1    5.0  6.0
b      2   3    NaN  NaN
c      4   5    7.0  8.0
```

There are additional arguments governing how the hierarchical index is created (see Table 8-3). For example, we can name the created axis levels with the names argument:

```
In [109]: pd.concat([df1, df2], axis="columns", keys=["level1", "level2"],
   .....:           names=["upper", "lower"])
Out[109]:
upper level1     level2
lower    one two  three four
a          0   1    5.0  6.0
b          2   3    NaN  NaN
c          4   5    7.0  8.0
```

A last consideration concerns DataFrames in which the row index does not contain any relevant data:

```
In [110]: df1 = pd.DataFrame(np.random.standard_normal((3, 4)),
   .....:                    columns=["a", "b", "c", "d"])

In [111]: df2 = pd.DataFrame(np.random.standard_normal((2, 3)),
   .....:                    columns=["b", "d", "a"])

In [112]: df1
Out[112]:
          a         b         c         d
0  1.248804  0.774191 -0.319657 -0.624964
```

```
1  1.078814  0.544647  0.855588  1.343268
2 -0.267175  1.793095 -0.652929 -1.886837

In [113]: df2
Out[113]:
          b         d         a
0  1.059626  0.644448 -0.007799
1 -0.449204  2.448963  0.667226
```

In this case, you can pass `ignore_index=True`, which discards the indexes from each DataFrame and concatenates the data in the columns only, assigning a new default index:

```
In [114]: pd.concat([df1, df2], ignore_index=True)
Out[114]:
          a         b         c         d
0  1.248804  0.774191 -0.319657 -0.624964
1  1.078814  0.544647  0.855588  1.343268
2 -0.267175  1.793095 -0.652929 -1.886837
3 -0.007799  1.059626       NaN  0.644448
4  0.667226 -0.449204       NaN  2.448963
```

Table 8-3 describes the `pandas.concat` function arguments.

Table 8-3. `pandas.concat` function arguments

Argument	Description
`objs`	List or dictionary of pandas objects to be concatenated; this is the only required argument
`axis`	Axis to concatenate along; defaults to concatenating along rows (`axis="index"`)
`join`	Either `"inner"` or `"outer"` (`"outer"` by default); whether to intersect (inner) or union (outer) indexes along the other axes
`keys`	Values to associate with objects being concatenated, forming a hierarchical index along the concatenation axis; can be a list or array of arbitrary values, an array of tuples, or a list of arrays (if multiple-level arrays passed in `levels`)
`levels`	Specific indexes to use as hierarchical index level or levels if keys passed
`names`	Names for created hierarchical levels if `keys` and/or `levels` passed
`verify_integrity`	Check new axis in concatenated object for duplicates and raise an exception if so; by default (`False`) allows duplicates
`ignore_index`	Do not preserve indexes along concatenation `axis`, instead produce a new `range(total_length)` index

Combining Data with Overlap

There is another data combination situation that can't be expressed as either a merge or concatenation operation. You may have two datasets with indexes that overlap in full or in part. As a motivating example, consider NumPy's `where` function, which performs the array-oriented equivalent of an if-else expression:

```
In [115]: a = pd.Series([np.nan, 2.5, 0.0, 3.5, 4.5, np.nan],
   .....:               index=["f", "e", "d", "c", "b", "a"])

In [116]: b = pd.Series([0., np.nan, 2., np.nan, np.nan, 5.],
   .....:               index=["a", "b", "c", "d", "e", "f"])

In [117]: a
Out[117]:
f    NaN
e    2.5
d    0.0
c    3.5
b    4.5
a    NaN
dtype: float64

In [118]: b
Out[118]:
a    0.0
b    NaN
c    2.0
d    NaN
e    NaN
f    5.0
dtype: float64

In [119]: np.where(pd.isna(a), b, a)
Out[119]: array([0. , 2.5, 0. , 3.5, 4.5, 5. ])
```

Here, whenever values in `a` are null, values from `b` are selected, otherwise the non-null values from `a` are selected. Using `numpy.where` does not check whether the index labels are aligned or not (and does not even require the objects to be the same length), so if you want to line up values by index, use the Series `combine_first` method:

```
In [120]: a.combine_first(b)
Out[120]:
a    0.0
b    4.5
c    3.5
d    0.0
e    2.5
f    5.0
dtype: float64
```

With DataFrames, `combine_first` does the same thing column by column, so you can think of it as "patching" missing data in the calling object with data from the object you pass:

```
In [121]: df1 = pd.DataFrame({"a": [1., np.nan, 5., np.nan],
   .....:                     "b": [np.nan, 2., np.nan, 6.],
   .....:                     "c": range(2, 18, 4)})
```

```
In [122]: df2 = pd.DataFrame({"a": [5., 4., np.nan, 3., 7.],
   .....:                     "b": [np.nan, 3., 4., 6., 8.]})

In [123]: df1
Out[123]:
     a    b   c
0  1.0  NaN   2
1  NaN  2.0   6
2  5.0  NaN  10
3  NaN  6.0  14

In [124]: df2
Out[124]:
     a    b
0  5.0  NaN
1  4.0  3.0
2  NaN  4.0
3  3.0  6.0
4  7.0  8.0

In [125]: df1.combine_first(df2)
Out[125]:
     a    b     c
0  1.0  NaN   2.0
1  4.0  2.0   6.0
2  5.0  4.0  10.0
3  3.0  6.0  14.0
4  7.0  8.0   NaN
```

The output of `combine_first` with DataFrame objects will have the union of all the column names.

8.3 Reshaping and Pivoting

There are a number of basic operations for rearranging tabular data. These are referred to as *reshape* or *pivot* operations.

Reshaping with Hierarchical Indexing

Hierarchical indexing provides a consistent way to rearrange data in a DataFrame. There are two primary actions:

`stack`
: This "rotates" or pivots from the columns in the data to the rows.

`unstack`
: This pivots from the rows into the columns.

I'll illustrate these operations through a series of examples. Consider a small DataFrame with string arrays as row and column indexes:

```
In [126]: data = pd.DataFrame(np.arange(6).reshape((2, 3)),
   .....:                     index=pd.Index(["Ohio", "Colorado"], name="state"),
   .....:                     columns=pd.Index(["one", "two", "three"],
   .....:                     name="number"))

In [127]: data
Out[127]:
number    one  two  three
state
Ohio        0    1      2
Colorado    3    4      5
```

Using the `stack` method on this data pivots the columns into the rows, producing a Series:

```
In [128]: result = data.stack()

In [129]: result
Out[129]:
state     number
Ohio      one       0
          two       1
          three     2
Colorado  one       3
          two       4
          three     5
dtype: int64
```

From a hierarchically indexed Series, you can rearrange the data back into a DataFrame with `unstack`:

```
In [130]: result.unstack()
Out[130]:
number    one  two  three
state
Ohio        0    1      2
Colorado    3    4      5
```

By default, the innermost level is unstacked (same with `stack`). You can unstack a different level by passing a level number or name:

```
In [131]: result.unstack(level=0)
Out[131]:
state   Ohio  Colorado
number
one        0         3
two        1         4
three      2         5

In [132]: result.unstack(level="state")
```

```
Out[132]:
state   Ohio  Colorado
number
one        0         3
two        1         4
three      2         5
```

Unstacking might introduce missing data if all of the values in the level aren't found in each subgroup:

```
In [133]: s1 = pd.Series([0, 1, 2, 3], index=["a", "b", "c", "d"], dtype="Int64")

In [134]: s2 = pd.Series([4, 5, 6], index=["c", "d", "e"], dtype="Int64")

In [135]: data2 = pd.concat([s1, s2], keys=["one", "two"])

In [136]: data2
Out[136]:
one  a    0
     b    1
     c    2
     d    3
two  c    4
     d    5
     e    6
dtype: Int64
```

Stacking filters out missing data by default, so the operation is more easily invertible:

```
In [137]: data2.unstack()
Out[137]:
        a     b  c  d     e
one     0     1  2  3  <NA>
two  <NA>  <NA>  4  5     6

In [138]: data2.unstack().stack()
Out[138]:
one  a    0
     b    1
     c    2
     d    3
two  c    4
     d    5
     e    6
dtype: Int64

In [139]: data2.unstack().stack(dropna=False)
Out[139]:
one  a       0
     b       1
     c       2
     d       3
     e    <NA>
```

```
two   a     <NA>
      b     <NA>
      c        4
      d        5
      e        6
dtype: Int64
```

When you unstack in a DataFrame, the level unstacked becomes the lowest level in the result:

```
In [140]: df = pd.DataFrame({"left": result, "right": result + 5},
   .....:                   columns=pd.Index(["left", "right"], name="side"))

In [141]: df
Out[141]:
side             left  right
state    number
Ohio     one        0      5
         two        1      6
         three      2      7
Colorado one        3      8
         two        4      9
         three      5     10

In [142]: df.unstack(level="state")
Out[142]:
side   left          right
state  Ohio Colorado  Ohio Colorado
number
one       0        3     5        8
two       1        4     6        9
three     2        5     7       10
```

As with `unstack`, when calling `stack` we can indicate the name of the axis to stack:

```
In [143]: df.unstack(level="state").stack(level="side")
Out[143]:
state         Colorado  Ohio
number side
one    left          3     0
       right         8     5
two    left          4     1
       right         9     6
three  left          5     2
       right        10     7
```

Pivoting “Long” to “Wide” Format

A common way to store multiple time series in databases and CSV files is what is sometimes called *long* or *stacked* format. In this format, individual values are represented by a single row in a table rather than multiple values per row.

Let's load some example data and do a small amount of time series wrangling and other data cleaning:

```
In [144]: data = pd.read_csv("examples/macrodata.csv")

In [145]: data = data.loc[:, ["year", "quarter", "realgdp", "infl", "unemp"]]

In [146]: data.head()
Out[146]:
   year  quarter   realgdp  infl  unemp
0  1959        1  2710.349  0.00    5.8
1  1959        2  2778.801  2.34    5.1
2  1959        3  2775.488  2.74    5.3
3  1959        4  2785.204  0.27    5.6
4  1960        1  2847.699  2.31    5.2
```

First, I use `pandas.PeriodIndex` (which represents time intervals rather than points in time), discussed in more detail in Chapter 11, to combine the `year` and `quarter` columns to set the index to consist of `datetime` values at the end of each quarter:

```
In [147]: periods = pd.PeriodIndex(year=data.pop("year"),
   .....:                          quarter=data.pop("quarter"),
   .....:                          name="date")

In [148]: periods
Out[148]:
PeriodIndex(['1959Q1', '1959Q2', '1959Q3', '1959Q4', '1960Q1', '1960Q2',
             '1960Q3', '1960Q4', '1961Q1', '1961Q2',
             ...
             '2007Q2', '2007Q3', '2007Q4', '2008Q1', '2008Q2', '2008Q3',
             '2008Q4', '2009Q1', '2009Q2', '2009Q3'],
            dtype='period[Q-DEC]', name='date', length=203)

In [149]: data.index = periods.to_timestamp("D")

In [150]: data.head()
Out[150]:
             realgdp  infl  unemp
date
1959-01-01  2710.349  0.00    5.8
1959-04-01  2778.801  2.34    5.1
1959-07-01  2775.488  2.74    5.3
1959-10-01  2785.204  0.27    5.6
1960-01-01  2847.699  2.31    5.2
```

Here I used the `pop` method on the DataFrame, which returns a column while deleting it from the DataFrame at the same time.

Then, I select a subset of columns and give the `columns` index the name `"item"`:

```
In [151]: data = data.reindex(columns=["realgdp", "infl", "unemp"])

In [152]: data.columns.name = "item"
```

```
In [153]: data.head()
Out[153]:
item         realgdp  infl  unemp
date
1959-01-01  2710.349  0.00    5.8
1959-04-01  2778.801  2.34    5.1
1959-07-01  2775.488  2.74    5.3
1959-10-01  2785.204  0.27    5.6
1960-01-01  2847.699  2.31    5.2
```

Lastly, I reshape with `stack`, turn the new index levels into columns with `reset_index`, and finally give the column containing the data values the name `"value"`:

```
In [154]: long_data = (data.stack()
   .....:              .reset_index()
   .....:              .rename(columns={0: "value"}))
```

Now, `ldata` looks like:

```
In [155]: long_data[:10]
Out[155]:
        date     item     value
0 1959-01-01  realgdp  2710.349
1 1959-01-01     infl     0.000
2 1959-01-01    unemp     5.800
3 1959-04-01  realgdp  2778.801
4 1959-04-01     infl     2.340
5 1959-04-01    unemp     5.100
6 1959-07-01  realgdp  2775.488
7 1959-07-01     infl     2.740
8 1959-07-01    unemp     5.300
9 1959-10-01  realgdp  2785.204
```

In this so-called *long* format for multiple time series, each row in the table represents a single observation.

Data is frequently stored this way in relational SQL databases, as a fixed schema (column names and data types) allows the number of distinct values in the `item` column to change as data is added to the table. In the previous example, `date` and `item` would usually be the primary keys (in relational database parlance), offering both relational integrity and easier joins. In some cases, the data may be more difficult to work with in this format; you might prefer to have a DataFrame containing one column per distinct `item` value indexed by timestamps in the `date` column. DataFrame's `pivot` method performs exactly this transformation:

```
In [156]: pivoted = long_data.pivot(index="date", columns="item",
   .....:                           values="value")

In [157]: pivoted.head()
Out[157]:
```

```
item          infl   realgdp  unemp
date
1959-01-01  0.00  2710.349    5.8
1959-04-01  2.34  2778.801    5.1
1959-07-01  2.74  2775.488    5.3
1959-10-01  0.27  2785.204    5.6
1960-01-01  2.31  2847.699    5.2
```

The first two values passed are the columns to be used, respectively, as the row and column index, then finally an optional value column to fill the DataFrame. Suppose you had two value columns that you wanted to reshape simultaneously:

```
In [158]: long_data["value2"] = np.random.standard_normal(len(long_data))

In [159]: long_data[:10]
Out[159]:
        date     item     value    value2
0 1959-01-01  realgdp  2710.349  0.802926
1 1959-01-01     infl     0.000  0.575721
2 1959-01-01    unemp     5.800  1.381918
3 1959-04-01  realgdp  2778.801  0.000992
4 1959-04-01     infl     2.340 -0.143492
5 1959-04-01    unemp     5.100 -0.206282
6 1959-07-01  realgdp  2775.488 -0.222392
7 1959-07-01     infl     2.740 -1.682403
8 1959-07-01    unemp     5.300  1.811659
9 1959-10-01  realgdp  2785.204 -0.351305
```

By omitting the last argument, you obtain a DataFrame with hierarchical columns:

```
In [160]: pivoted = long_data.pivot(index="date", columns="item")

In [161]: pivoted.head()
Out[161]:
           value                    value2
item        infl   realgdp unemp      infl   realgdp     unemp
date
1959-01-01  0.00  2710.349   5.8  0.575721  0.802926  1.381918
1959-04-01  2.34  2778.801   5.1 -0.143492  0.000992 -0.206282
1959-07-01  2.74  2775.488   5.3 -1.682403 -0.222392  1.811659
1959-10-01  0.27  2785.204   5.6  0.128317 -0.351305 -1.313554
1960-01-01  2.31  2847.699   5.2 -0.615939  0.498327  0.174072

In [162]: pivoted["value"].head()
Out[162]:
item          infl   realgdp  unemp
date
1959-01-01  0.00  2710.349    5.8
1959-04-01  2.34  2778.801    5.1
1959-07-01  2.74  2775.488    5.3
1959-10-01  0.27  2785.204    5.6
1960-01-01  2.31  2847.699    5.2
```

Note that pivot is equivalent to creating a hierarchical index using set_index followed by a call to unstack:

```
In [163]: unstacked = long_data.set_index(["date", "item"]).unstack(level="item")

In [164]: unstacked.head()
Out[164]:
           value                    value2
item        infl   realgdp unemp      infl   realgdp     unemp
date
1959-01-01  0.00  2710.349   5.8  0.575721  0.802926  1.381918
1959-04-01  2.34  2778.801   5.1 -0.143492  0.000992 -0.206282
1959-07-01  2.74  2775.488   5.3 -1.682403 -0.222392  1.811659
1959-10-01  0.27  2785.204   5.6  0.128317 -0.351305 -1.313554
1960-01-01  2.31  2847.699   5.2 -0.615939  0.498327  0.174072
```

Pivoting "Wide" to "Long" Format

An inverse operation to pivot for DataFrames is pandas.melt. Rather than transforming one column into many in a new DataFrame, it merges multiple columns into one, producing a DataFrame that is longer than the input. Let's look at an example:

```
In [166]: df = pd.DataFrame({"key": ["foo", "bar", "baz"],
   .....:                    "A": [1, 2, 3],
   .....:                    "B": [4, 5, 6],
   .....:                    "C": [7, 8, 9]})

In [167]: df
Out[167]:
   key  A  B  C
0  foo  1  4  7
1  bar  2  5  8
2  baz  3  6  9
```

The "key" column may be a group indicator, and the other columns are data values. When using pandas.melt, we must indicate which columns (if any) are group indicators. Let's use "key" as the only group indicator here:

```
In [168]: melted = pd.melt(df, id_vars="key")

In [169]: melted
Out[169]:
   key variable  value
0  foo        A      1
1  bar        A      2
2  baz        A      3
3  foo        B      4
4  bar        B      5
5  baz        B      6
6  foo        C      7
7  bar        C      8
8  baz        C      9
```

Using `pivot`, we can reshape back to the original layout:

```
In [170]: reshaped = melted.pivot(index="key", columns="variable",
   .....:                         values="value")

In [171]: reshaped
Out[171]:
variable  A  B  C
key
bar       2  5  8
baz       3  6  9
foo       1  4  7
```

Since the result of `pivot` creates an index from the column used as the row labels, we may want to use `reset_index` to move the data back into a column:

```
In [172]: reshaped.reset_index()
Out[172]:
variable  key  A  B  C
0         bar  2  5  8
1         baz  3  6  9
2         foo  1  4  7
```

You can also specify a subset of columns to use as `value` columns:

```
In [173]: pd.melt(df, id_vars="key", value_vars=["A", "B"])
Out[173]:
   key variable  value
0  foo        A      1
1  bar        A      2
2  baz        A      3
3  foo        B      4
4  bar        B      5
5  baz        B      6
```

`pandas.melt` can be used without any group identifiers, too:

```
In [174]: pd.melt(df, value_vars=["A", "B", "C"])
Out[174]:
  variable  value
0        A      1
1        A      2
2        A      3
3        B      4
4        B      5
5        B      6
6        C      7
7        C      8
8        C      9

In [175]: pd.melt(df, value_vars=["key", "A", "B"])
Out[175]:
  variable value
0      key   foo
```

```
1       key   bar
2       key   baz
3         A     1
4         A     2
5         A     3
6         B     4
7         B     5
8         B     6
```

8.4 Conclusion

Now that you have some pandas basics for data import, cleaning, and reorganization under your belt, we are ready to move on to data visualization with matplotlib. We will return to explore other areas of pandas later in the book when we discuss more advanced analytics.

CHAPTER 9

Plotting and Visualization

Making informative visualizations (sometimes called *plots*) is one of the most important tasks in data analysis. It may be a part of the exploratory process—for example, to help identify outliers or needed data transformations, or as a way of generating ideas for models. For others, building an interactive visualization for the web may be the end goal. Python has many add-on libraries for making static or dynamic visualizations, but I'll be mainly focused on matplotlib and libraries that build on top of it.

matplotlib is a desktop plotting package designed for creating plots and figures suitable for publication. The project was started by John Hunter in 2002 to enable a MATLAB-like plotting interface in Python. The matplotlib and IPython communities have collaborated to simplify interactive plotting from the IPython shell (and now, Jupyter notebook). matplotlib supports various GUI backends on all operating systems and can export visualizations to all of the common vector and raster graphics formats (PDF, SVG, JPG, PNG, BMP, GIF, etc.). With the exception of a few diagrams, nearly all of the graphics in this book were produced using matplotlib.

Over time, matplotlib has spawned a number of add-on toolkits for data visualization that use matplotlib for their underlying plotting. One of these is seaborn, which we explore later in this chapter.

The simplest way to follow the code examples in the chapter is to output plots in the Jupyter notebook. To set this up, execute the following statement in a Jupyter notebook:

```
%matplotlib inline
```