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 easy 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 biggest difference is that pandas is designed for working with tabular or heterogeneous data. NumPy, by contrast, is best suited for working with homogeneous 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 800 distinct contributors, who've been helping build the project as they've used it to solve their day-to-day data problems.

Throughout the rest of the book, I use the following import convention for pandas:

```
In [1]: 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 [2]: 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, easy-to-use basis for most applications.

Series

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

```
In [11]: obj = pd.Series([4, 7, -5, 3])

In [12]: obj
Out[12]:
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 values and index attributes, respectively:

```
In [13]: obj.values
Out[13]: array([ 4,  7, -5,  3])

In [14]: obj.index  # like range(4)
Out[14]: RangeIndex(start=0, stop=4, step=1)
```

Often it will be desirable to create a Series with an index identifying each data point with a label:

```
In [15]: obj2 = pd.Series([4, 7, -5, 3], index=['d', 'b', 'a', 'c'])

In [16]: obj2
Out[16]:
d    4
b    7
a   -5
c    3
dtype: int64

In [17]: obj2.index
Out[17]: 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 [18]: obj2['a']
Out[18]: -5

In [19]: obj2['d'] = 6

In [20]: obj2[['c', 'a', 'd']]
Out[20]:
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 [21]: obj2[obj2 > 0]
Out[21]:
d    6
b    7
c    3
dtype: int64

In [22]: obj2 * 2
Out[22]:
d    12
b    14
a   -10
c     6
dtype: int64

In [23]: np.exp(obj2)
Out[23]:
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 dict, as it is a mapping of index values to data values. It can be used in many contexts where you might use a dict:

```
In [24]: 'b' in obj2
Out[24]: True
```

```
In [25]: 'e' in obj2
Out[25]: False
```

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

```
In [26]: sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}

In [27]: obj3 = pd.Series(sdata)

In [28]: obj3
Out[28]:
Ohio      35000
Oregon    16000
Texas     71000
Utah       5000
dtype: int64
```

When you are only passing a dict, the index in the resulting Series will have the dict's keys in sorted order. You can override this by passing the dict keys in the order you want them to appear in the resulting Series:

```
In [29]: states = ['California', 'Ohio', 'Oregon', 'Texas']

In [30]: obj4 = pd.Series(sdata, index=states)

In [31]: obj4
Out[31]:
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" or "NA" interchangeably to refer to missing data. The `isnull` and `notnull` functions in pandas should be used to detect missing data:

```
In [32]: pd.isnull(obj4)
Out[32]:
California     True
Ohio          False
Oregon        False
Texas         False
dtype: bool

In [33]: pd.notnull(obj4)
Out[33]:
```

```
California    False
Ohio           True
Oregon         True
Texas          True
dtype: bool
```

Series also has these as instance methods:

```
In [34]: obj4.isnull()
Out[34]:
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 [35]: obj3
Out[35]:
Ohio      35000
Oregon    16000
Texas     71000
Utah       5000
dtype: int64

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

In [37]: obj3 + obj4
Out[37]:
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 key areas of pandas functionality:

```
In [38]: obj4.name = 'population'

In [39]: obj4.index.name = 'state'

In [40]: obj4
Out[40]:
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 [41]: obj
Out[41]:
0    4
1    7
2   -5
3    3
dtype: int64

In [42]: obj.index = ['Bob', 'Steve', 'Jeff', 'Ryan']

In [43]: obj
Out[43]:
Bob      4
Steve    7
Jeff    -5
Ryan     3
dtype: int64
```

DataFrame

A DataFrame represents a rectangular table of data and contains an ordered 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 dict of Series all sharing the same index. Under the hood, the data is stored as one or more two-dimensional blocks rather than a list, dict, or some other collection of one-dimensional arrays. The exact details of DataFrame's internals are outside the scope of this book.

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 dict 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 in sorted order:

```
In [45]: frame
Out[45]:
   pop   state  year
0  1.5    Ohio  2000
1  1.7    Ohio  2001
2  3.6    Ohio  2002
3  2.4  Nevada  2001
4  2.9  Nevada  2002
5  3.2  Nevada  2003
```

If you are using the Jupyter notebook, pandas DataFrame objects will be displayed as a more browser-friendly HTML table.

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

```
In [46]: frame.head()
Out[46]:
   pop   state  year
0  1.5    Ohio  2000
1  1.7    Ohio  2001
2  3.6    Ohio  2002
3  2.4  Nevada  2001
4  2.9  Nevada  2002
```

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

```
In [47]: pd.DataFrame(data, columns=['year', 'state', 'pop'])
Out[47]:
   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 dict, it will appear with missing values in the result:

```
In [48]: frame2 = pd.DataFrame(data, columns=['year', 'state', 'pop', 'debt'],
   ....:                       index=['one', 'two', 'three', 'four',
   ....:                              'five', 'six'])
```

```
In [49]: frame2
Out[49]:
       year   state  pop debt
one    2000    Ohio  1.5  NaN
two    2001    Ohio  1.7  NaN
three  2002    Ohio  3.6  NaN
four   2001  Nevada  2.4  NaN
five   2002  Nevada  2.9  NaN
six    2003  Nevada  3.2  NaN

In [50]: frame2.columns
Out[50]: Index(['year', 'state', 'pop', 'debt'], dtype='object')
```

A column in a DataFrame can be retrieved as a Series either by dict-like notation or by attribute:

```
In [51]: frame2['state']
Out[51]:
one        Ohio
two        Ohio
three      Ohio
four     Nevada
five     Nevada
six      Nevada
Name: state, dtype: object

In [52]: frame2.year
Out[52]:
one      2000
two      2001
three    2002
four     2001
five     2002
six      2003
Name: year, dtype: int64
```

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

`frame2[column]` works for any column name, but `frame2.column` only works when the column name is a valid Python variable name.

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 `loc` attribute (much more on this later):

```
In [53]: frame2.loc['three']
Out[53]:
year     2002
state    Ohio
pop       3.6
debt      NaN
Name: three, 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 [54]: frame2['debt'] = 16.5

In [55]: frame2
Out[55]:
       year   state  pop  debt
one    2000    Ohio  1.5  16.5
two    2001    Ohio  1.7  16.5
three  2002    Ohio  3.6  16.5
four   2001  Nevada  2.4  16.5
five   2002  Nevada  2.9  16.5
six    2003  Nevada  3.2  16.5

In [56]: frame2['debt'] = np.arange(6.)

In [57]: frame2
Out[57]:
       year   state  pop  debt
one    2000    Ohio  1.5   0.0
two    2001    Ohio  1.7   1.0
three  2002    Ohio  3.6   2.0
four   2001  Nevada  2.4   3.0
five   2002  Nevada  2.9   4.0
six    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 holes:

```
In [58]: val = pd.Series([-1.2, -1.5, -1.7], index=['two', 'four', 'five'])

In [59]: frame2['debt'] = val

In [60]: frame2
Out[60]:
       year   state  pop  debt
one    2000    Ohio  1.5   NaN
two    2001    Ohio  1.7  -1.2
three  2002    Ohio  3.6   NaN
four   2001  Nevada  2.4  -1.5
five   2002  Nevada  2.9  -1.7
six    2003  Nevada  3.2   NaN
```

Assigning a column that doesn't exist will create a new column. The `del` keyword will delete columns as with a dict.

As an example of `del`, I first add a new column of boolean values where the `state` column equals `'Ohio'`:

```
In [61]: frame2['eastern'] = frame2.state == 'Ohio'

In [62]: frame2
Out[62]:
       year   state  pop  debt  eastern
one    2000    Ohio  1.5   NaN     True
two    2001    Ohio  1.7  -1.2     True
three  2002    Ohio  3.6   NaN     True
four   2001  Nevada  2.4  -1.5    False
five   2002  Nevada  2.9  -1.7    False
six    2003  Nevada  3.2   NaN    False
```

New columns cannot be created with the `frame2.eastern` syntax.

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

```
In [63]: del frame2['eastern']

In [64]: frame2.columns
Out[64]: 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 dict of dicts:

```
In [65]: pop = {'Nevada': {2001: 2.4, 2002: 2.9},
   ....:        'Ohio': {2000: 1.5, 2001: 1.7, 2002: 3.6}}
```

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

```
In [66]: frame3 = pd.DataFrame(pop)

In [67]: frame3
Out[67]:
      Nevada  Ohio
2000     NaN   1.5
```

```
2001     2.4   1.7
2002     2.9   3.6
```

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

```
In [68]: frame3.T
Out[68]:
        2000  2001  2002
Nevada   NaN   2.4   2.9
Ohio     1.5   1.7   3.6
```

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

```
In [69]: pd.DataFrame(pop, index=[2001, 2002, 2003])
Out[69]:
      Nevada  Ohio
2001     2.4   1.7
2002     2.9   3.6
2003     NaN   NaN
```

Dicts of Series are treated in much the same way:

```
In [70]: pdata = {'Ohio': frame3['Ohio'][:-1],
   ....:          'Nevada': frame3['Nevada'][:2]}

In [71]: pd.DataFrame(pdata)
Out[71]:
      Nevada  Ohio
2000     NaN   1.5
2001     2.4   1.7
```

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

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

```
In [72]: frame3.index.name = 'year'; frame3.columns.name = 'state'

In [73]: frame3
Out[73]:
state  Nevada  Ohio
year
2000      NaN   1.5
2001      2.4   1.7
2002      2.9   3.6
```

As with Series, the `values` attribute returns the data contained in the DataFrame as a two-dimensional ndarray:

```
In [74]: frame3.values
Out[74]:
array([[ nan,  1.5],
       [ 2.4,  1.7],
       [ 2.9,  3.6]])
```

If the DataFrame's columns are different dtypes, the dtype of the values array will be chosen to accommodate all of the columns:

```
In [75]: frame2.values
Out[75]:
array([[2000, 'Ohio', 1.5, nan],
       [2001, 'Ohio', 1.7, -1.2],
       [2002, 'Ohio', 3.6, nan],
       [2001, 'Nevada', 2.4, -1.5],
       [2002, 'Nevada', 2.9, -1.7],
       [2003, 'Nevada', 3.2, nan]], dtype=object)
```

Table 5-1. Possible data inputs to DataFrame constructor

Type	Notes
2D ndarray	A matrix of data, passing optional row and column labels
dict 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 "dict of arrays" case
dict 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
dict of dicts	Each inner dict becomes a column; keys are unioned to form the row index as in the "dict of Series" case
List of dicts or Series	Each item becomes a row in the DataFrame; union of dict 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 become NA/missing in the DataFrame result

Index Objects

pandas's Index objects are responsible for holding the axis labels 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 [76]: obj = pd.Series(range(3), index=['a', 'b', 'c'])

In [77]: index = obj.index

In [78]: index
Out[78]: Index(['a', 'b', 'c'], dtype='object')

In [79]: index[1:]
Out[79]: 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 [80]: labels = pd.Index(np.arange(3))

In [81]: labels
Out[81]: Int64Index([0, 1, 2], dtype='int64')

In [82]: obj2 = pd.Series([1.5, -2.5, 0], index=labels)

In [83]: obj2
Out[83]:
0    1.5
1   -2.5
2    0.0
dtype: float64

In [84]: obj2.index is labels
Out[84]: True
```

Some users will not often take advantage of the capabilities provided by indexes, 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 [85]: frame3
Out[85]:
state  Nevada  Ohio
year
2000      NaN   1.5
2001      2.4   1.7
2002      2.9   3.6

In [86]: frame3.columns
Out[86]: Index(['Nevada', 'Ohio'], dtype='object', name='state')

In [87]: 'Ohio' in frame3.columns
Out[87]: True

In [88]: 2003 in frame3.index
Out[88]: False
```

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

```
In [89]: dup_labels = pd.Index(['foo', 'foo', 'bar', 'bar'])

In [90]: dup_labels
Out[90]: 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	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`
`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 the most important features, leaving the less common (i.e., more esoteric) things for you to explore on your own.

Reindexing

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

```
In [91]: obj = pd.Series([4.5, 7.2, -5.3, 3.6], index=['d', 'b', 'a', 'c'])

In [92]: obj
Out[92]: 
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 [93]: obj2 = obj.reindex(['a', 'b', 'c', 'd', 'e'])

In [94]: obj2
Out[94]:
a   -5.3
b    7.2
c    3.6
d    4.5
e    NaN
dtype: float64
```

For ordered data like time series, it may be desirable 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 [95]: obj3 = pd.Series(['blue', 'purple', 'yellow'], index=[0, 2, 4])

In [96]: obj3
Out[96]:
0      blue
2    purple
4    yellow
dtype: object

In [97]: obj3.reindex(range(6), method='ffill')
Out[97]:
0      blue
1      blue
2    purple
3    purple
4    yellow
5    yellow
dtype: object
```

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

```
In [98]: frame = pd.DataFrame(np.arange(9).reshape((3, 3)),
   ....:                      index=['a', 'c', 'd'],
   ....:                      columns=['Ohio', 'Texas', 'California'])

In [99]: frame
Out[99]:
   Ohio  Texas  California
a     0      1           2
c     3      4           5
d     6      7           8

In [100]: frame2 = frame.reindex(['a', 'b', 'c', 'd'])
In [101]: frame2
Out[101]:
   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 [102]: states = ['Texas', 'Utah', 'California']

In [103]: frame.reindex(columns=states)
Out[103]:
   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`.

As we'll explore in more detail, you can reindex more succinctly by label-indexing with `loc`, and many users prefer to use it exclusively:

```
In [104]: frame.loc[['a', 'b', 'c', 'd'], states]
Out[104]:
   Texas  Utah  California
a    1.0   NaN         2.0
b    NaN   NaN         NaN
c    4.0   NaN         5.0
d    7.0   NaN         8.0
```

Table 5-3. reindex function arguments

Argument	Description
`index`	New sequence to use as index. Can be Index instance or any other sequence-like Python data structure. An Index will be used exactly as is without any copying.
`method`	Interpolation (fill) method; `'ffill'` fills forward, while `'bfill'` fills backward.
`fill_value`	Substitute value to use when introducing missing data by reindexing.
`limit`	When forward- or backfilling, maximum size gap (in number of elements) to fill.
`tolerance`	When forward- or backfilling, 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 new index is equivalent to old index; if `False`, do not copy the data when the indexes are equivalent.

Dropping Entries from an Axis

Dropping one or more entries from an axis is easy if you already have an index array or list without those entries. 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 [105]: obj = pd.Series(np.arange(5.), index=['a', 'b', 'c', 'd', 'e'])

In [106]: obj
Out[106]:
a    0.0
b    1.0
c    2.0
d    3.0
e    4.0
dtype: float64

In [107]: new_obj = obj.drop('c')

In [108]: new_obj
Out[108]:
a    0.0
b    1.0
d    3.0
e    4.0
dtype: float64

In [109]: obj.drop(['d', 'c'])
Out[109]:
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 [110]: data = pd.DataFrame(np.arange(16).reshape((4, 4)),
   .....:                     index=['Ohio', 'Colorado', 'Utah', 'New York'],
   .....:                     columns=['one', 'two', 'three', 'four'])

In [111]: data
Out[111]:
          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 [112]: data.drop(['Colorado', 'Ohio'])
Out[112]:
          one  two  three  four
Utah        8    9     10    11
New York   12   13     14    15
```

You can drop values from the columns by passing `axis=1` or `axis='columns'`:

```
In [113]: data.drop('two', axis=1)
Out[113]:
          one  three  four
Ohio        0      2     3
Colorado    4      6     7
Utah        8     10    11
New York   12     14    15

In [114]: data.drop(['two', 'four'], axis='columns')
Out[114]:
          one  three
Ohio        0      2
Colorado    4      6
Utah        8     10
New York   12     14
```

Many functions, like `drop`, which modify the size or shape of a Series or DataFrame, can manipulate an object *in-place* without returning a new object:

```
In [115]: obj.drop('c', inplace=True)

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

Be careful with the `inplace`, as it destroys any data that is dropped.

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 [117]: obj = pd.Series(np.arange(4.), index=['a', 'b', 'c', 'd'])

In [118]: obj
Out[118]:
a    0.0
b    1.0
c    2.0
d    3.0
dtype: float64

In [119]: obj['b']
Out[119]: 1.0
```

```
In [120]: obj[1]
Out[120]: 1.0

In [121]: obj[2:4]
Out[121]:
c    2.0
d    3.0
dtype: float64

In [122]: obj[['b', 'a', 'd']]
Out[122]:
b    1.0
a    0.0
d    3.0
dtype: float64

In [123]: obj[[1, 3]]
Out[123]:
b    1.0
d    3.0
dtype: float64

In [124]: obj[obj < 2]
Out[124]:
a    0.0
b    1.0
dtype: float64
```

Slicing with labels behaves differently than normal Python slicing in that the endpoint is inclusive:

```
In [125]: obj['b':'c']
Out[125]:
b    1.0
c    2.0
dtype: float64
```

Setting using these methods modifies the corresponding section of the Series:

```
In [126]: obj['b':'c'] = 5

In [127]: obj
Out[127]:
a    0.0
b    5.0
c    5.0
d    3.0
dtype: float64
```

Indexing into a DataFrame is for retrieving one or more columns either with a single value or sequence:

```
In [128]: data = pd.DataFrame(np.arange(16).reshape((4, 4)),
   .....:                     index=['Ohio', 'Colorado', 'Utah', 'New York'],
   .....:                     columns=['one', 'two', 'three', 'four'])

In [129]: data
Out[129]:
          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 [130]: data['two']
Out[130]:
Ohio         1
Colorado     5
Utah         9
New York    13
Name: two, dtype: int64

In [131]: data[['three', 'one']]
Out[131]:
          three  one
Ohio          2    0
Colorado      6    4
Utah         10    8
New York     14   12
```

Indexing like this has a few special cases. First, slicing or selecting data with a boolean array:

```
In [132]: data[:2]
Out[132]:
          one  two  three  four
Ohio        0    1      2     3
Colorado    4    5      6     7

In [133]: data[data['three'] > 5]
Out[133]:
          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 in indexing with a boolean DataFrame, such as one produced by a scalar comparison:

```
In [134]: data < 5
Out[134]:
            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

In [135]: data[data < 5] = 0

In [136]: data
Out[136]:
          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
```

This makes DataFrame syntactically more like a two-dimensional NumPy array in this particular case.

Selection with loc and iloc

For DataFrame label-indexing on the rows, I introduce the special indexing operators `loc` and `iloc`. They enable you to select a subset of the rows and columns from a DataFrame with NumPy-like notation using either axis labels (`loc`) or integers (`iloc`).

As a preliminary example, let's select a single row and multiple columns by label:

```
In [137]: data.loc['Colorado', ['two', 'three']]
Out[137]:
two      5
three    6
Name: Colorado, dtype: int64
```

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

```
In [138]: data.iloc[2, [3, 0, 1]]
Out[138]:
four    11
one      8
two      9
Name: Utah, dtype: int64

In [139]: data.iloc[2]
Out[139]:
one       8
two       9
three    10
four     11
Name: Utah, dtype: int64
```

```
In [140]: data.iloc[[1, 2], [3, 0, 1]]
Out[140]:
          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 [141]: data.loc[:'Utah', 'two']
Out[141]:
Ohio        0
Colorado    5
Utah        9
Name: two, dtype: int64

In [142]: data.iloc[:, :3][data.three > 5]
Out[142]:
          one  two  three
Colorado    0    5      6
Utah        8    9     10
New York   12   13     14
```

So 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'll see later, there are a number of additional options for working with hierarchical indexes.

When originally designing pandas, I felt that having to type `frame[:, col]` to select a column was too verbose (and error-prone), since column selection is one of the most common operations. I made the design trade-off to push all of the fancy indexing behavior (both labels and integers) into the `ix` operator. In practice, this led to many edge cases in data with integer axis labels, so the pandas team decided to create the `loc` and `iloc` operators to deal with strictly label-based and integer-based indexing, respectively.

The `ix` indexing operator still exists, but it is deprecated. I do not recommend using it.

Table 5-4. Indexing options with DataFrame

Type	Notes
`df[val]`	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[val]`	Selects single row or subset of rows from the DataFrame by label
`df.loc[:, val]`	Selects single column or subset of columns by label
`df.loc[val1, val2]`	Select both rows and columns by label
`df.iloc[where]`	Selects single row or subset of rows from the DataFrame by integer position

Type	Notes
`df.iloc[:, where]`	Selects single column or subset of columns by integer position
`df.iloc[where_i, where_j]`	Select both rows and columns by integer position
`df.at[label_i, label_j]`	Select a single scalar value by row and column label
`df.iat[i, j]`	Select a single scalar value by row and column position (integers)
`reindex` method	Select either rows or columns by labels
`get_value`, `set_value` methods	Select single value by row and column label

Integer Indexes

Working with pandas objects indexed by integers is something that often trips up new users due to some differences with indexing semantics on built-in Python data structures like lists and tuples. For example, you might not expect the following code to generate an error:

```
ser = pd.Series(np.arange(3.))
ser
ser[-1]
```

In this case, pandas could "fall back" on integer indexing, but it's difficult to do this in general without introducing subtle bugs. Here we have an index containing 0, 1, 2, but inferring what the user wants (label-based indexing or position-based) is difficult:

```
In [144]: ser
Out[144]:
0    0.0
1    1.0
2    2.0
dtype: float64
```

On the other hand, with a non-integer index, there is no potential for ambiguity:

```
In [145]: ser2 = pd.Series(np.arange(3.), index=['a', 'b', 'c'])

In [146]: ser2[-1]
Out[146]: 2.0
```

To keep things consistent, if you have an axis index containing integers, data selection will always be label-oriented. For more precise handling, use `loc` (for labels) or `iloc` (for integers):

```
In [147]: ser[:1]
Out[147]:
0    0.0
dtype: float64

In [148]: ser.loc[:1]
Out[148]:
0    0.0
1    1.0
```

```
dtype: float64

In [149]: ser.iloc[:1]
Out[149]:
0    0.0
dtype: float64
```

Arithmetic and Data Alignment

An important pandas feature for some applications is the behavior of arithmetic between objects with different indexes. When you are adding together objects, if any index pairs are not the same, the respective index in the result will be the union of the index pairs. For users with database experience, this is similar to an automatic outer join on the index labels. Let's look at an example:

```
In [150]: s1 = pd.Series([7.3, -2.5, 3.4, 1.5], index=['a', 'c', 'd', 'e'])

In [151]: s2 = pd.Series([-2.1, 3.6, -1.5, 4, 3.1],
   .....:                index=['a', 'c', 'e', 'f', 'g'])

In [152]: s1
Out[152]:
a    7.3
c   -2.5
d    3.4
e    1.5
dtype: float64

In [153]: s2
Out[153]:
a   -2.1
c    3.6
e   -1.5
f    4.0
g    3.1
dtype: float64
```

Adding these together yields:

```
In [154]: s1 + s2
Out[154]:
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 the rows and the columns:

```
In [155]: df1 = pd.DataFrame(np.arange(9.).reshape((3, 3)), columns=list('bcd'),
   .....:                    index=['Ohio', 'Texas', 'Colorado'])

In [156]: df2 = pd.DataFrame(np.arange(12.).reshape((4, 3)), columns=list('bde'),
   .....:                    index=['Utah', 'Ohio', 'Texas', 'Oregon'])

In [157]: df1
Out[157]:
            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 [158]: df2
Out[158]:
          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 together returns a DataFrame whose index and columns are the unions of the ones in each DataFrame:

```
In [159]: df1 + df2
Out[159]:
            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 all missing in the result. The same holds for the rows whose labels 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 [160]: df1 = pd.DataFrame({'A': [1, 2]})

In [161]: df2 = pd.DataFrame({'B': [3, 4]})

In [162]: df1
Out[162]:
   A
0  1
1  2

In [163]: df2
```

```
Out[163]:
   B
0  3
1  4

In [164]: df1 - df2
Out[164]:
    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:

```
In [165]: df1 = pd.DataFrame(np.arange(12.).reshape((3, 4)),
   .....:                    columns=list('abcd'))

In [166]: df2 = pd.DataFrame(np.arange(20.).reshape((4, 5)),
   .....:                    columns=list('abcde'))

In [167]: df2.loc[1, 'b'] = np.nan

In [168]: df1
Out[168]:
     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 [169]: df2
Out[169]:
      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 together results in NA values in the locations that don't overlap:

```
In [170]: df1 + df2
Out[170]:
      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`:

```
In [171]: df1.add(df2, fill_value=0)
Out[171]:
      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 of them has a counterpart, starting with the letter `r`, that has arguments flipped. So these two statements are equivalent:

```
In [172]: 1 / df1
Out[172]:
          a         b         c         d
0       inf  1.000000  0.500000  0.333333
1  0.250000  0.200000  0.166667  0.142857
2  0.125000  0.111111  0.100000  0.090909

In [173]: df1.rdiv(1)
Out[173]:
          a         b         c         d
0       inf  1.000000  0.500000  0.333333
1  0.250000  0.200000  0.166667  0.142857
2  0.125000  0.111111  0.100000  0.090909
```

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

```
In [174]: df1.reindex(columns=df2.columns, fill_value=0)
Out[174]:
     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 [175]: arr = np.arange(12.).reshape((3, 4))

In [176]: arr
Out[176]:
array([[  0.,   1.,   2.,   3.],
       [  4.,   5.,   6.,   7.],
       [  8.,   9.,  10.,  11.]])

In [177]: arr[0]
Out[177]: array([ 0.,  1.,  2.,  3.])

In [178]: arr - arr[0]
Out[178]:
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 [179]: frame = pd.DataFrame(np.arange(12.).reshape((4, 3)),
   .....:                      columns=list('bde'),
   .....:                      index=['Utah', 'Ohio', 'Texas', 'Oregon'])

In [180]: series = frame.iloc[0]

In [181]: frame
Out[181]:
          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 [182]: series
Out[182]:
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 DataFrame's columns, broadcasting down the rows:

```
In [183]: frame - series
Out[183]:
          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 [184]: series2 = pd.Series(range(3), index=['b', 'e', 'f'])

In [185]: frame + series2
Out[185]:
          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. For example:

```
In [186]: series3 = frame['d']

In [187]: frame
Out[187]:
          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 [188]: series3
Out[188]:
Utah       1.0
Ohio       4.0
Texas      7.0
Oregon    10.0
Name: d, dtype: float64

In [189]: frame.sub(series3, axis='index')
Out[189]:
          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 number that you pass is the *axis to match on*. In this case we mean to match on the DataFrame's row index (`axis='index'` or `axis=0`) and broadcast across.

Function Application and Mapping

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

```
In [190]: frame = pd.DataFrame(np.random.randn(4, 3), columns=list('bde'),
   .....:                      index=['Utah', 'Ohio', 'Texas', 'Oregon'])
```

```
In [191]: frame
Out[191]: 
               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 [192]: np.abs(frame)
Out[192]: 
               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 [193]: f = lambda x: x.max() - x.min()

In [194]: frame.apply(f)
Out[194]: 
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:

```
In [195]: frame.apply(f, axis='columns')
Out[195]: 
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 [196]: def f(x):
   .....:     return pd.Series([x.min(), x.max()], index=['min', 'max'])

In [197]: frame.apply(f)
```

```
Out[197]:
            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 [198]: format = lambda x: '%.2f' % x

In [199]: frame.applymap(format)
Out[199]:
            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 [200]: frame['e'].map(format)
Out[200]:
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 index, use the sort_index method, which returns a new, sorted object:

```
In [201]: obj = pd.Series(range(4), index=['d', 'a', 'b', 'c'])

In [202]: obj.sort_index()
Out[202]:
a    1
b    2
c    3
d    0
dtype: int64
```

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

```
In [203]: frame = pd.DataFrame(np.arange(8).reshape((2, 4)),
   .....:                      index=['three', 'one'],
   .....:                      columns=['d', 'a', 'b', 'c'])

In [204]: frame.sort_index()
```

```
Out[204]:
       d  a  b  c
one    4  5  6  7
three  0  1  2  3

In [205]: frame.sort_index(axis=1)
Out[205]:
       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 [206]: frame.sort_index(axis=1, ascending=False)
Out[206]:
       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 [207]: obj = pd.Series([4, 7, -3, 2])

In [208]: obj.sort_values()
Out[208]:
2   -3
3    2
0    4
1    7
dtype: int64
```

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

```
In [209]: obj = pd.Series([4, np.nan, 7, np.nan, -3, 2])

In [210]: obj.sort_values()
Out[210]:
4   -3.0
5    2.0
0    4.0
2    7.0
1    NaN
3    NaN
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 the by option of `sort_values`:

```
In [211]: frame = pd.DataFrame({'b': [4, 7, -3, 2], 'a': [0, 1, 0, 1]})

In [212]: frame
Out[212]:
   a  b
```

```
0  0  4
1  1  7
2  0 -3
3  1  2

In [213]: frame.sort_values(by='b')
Out[213]:
   a  b
2  0 -3
3  1  2
0  0  4
1  1  7
```

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

```
In [214]: frame.sort_values(by=['a', 'b'])
Out[214]:
   a  b
2  0 -3
0  0  4
3  1  2
1  1  7
```

Ranking assigns ranks from one through the number of valid data points in an array. 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 [215]: obj = pd.Series([7, -5, 7, 4, 2, 0, 4])

In [216]: obj.rank()
Out[216]:
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 [217]: obj.rank(method='first')
Out[217]:
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:

```
# Assign tie values the maximum rank in the group
In [218]: obj.rank(ascending=False, method='max')
Out[218]:
0    2.0
1    7.0
2    2.0
3    4.0
4    5.0
5    6.0
6    4.0
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 [219]: frame = pd.DataFrame({'b': [4.3, 7, -3, 2], 'a': [0, 1, 0, 1],
   .....:                       'c': [-2, 5, 8, -2.5]})

In [220]: frame
Out[220]:
   a    b    c
0  0  4.3 -2.0
1  1  7.0  5.0
2  0 -3.0  8.0
3  1  2.0 -2.5

In [221]: frame.rank(axis='columns')
Out[221]:
     a    b    c
0  2.0  3.0  1.0
1  1.0  3.0  2.0
2  2.0  1.0  3.0
3  2.0  3.0  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 in between groups rather than the number of equal elements in a group

Axis Indexes with Duplicate Labels

Up until now all of the examples we've looked at have had 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 [222]: obj = pd.Series(range(5), index=['a', 'a', 'b', 'b', 'c'])

In [223]: obj
Out[223]:
a    0
a    1
b    2
b    3
c    4
dtype: int64
```

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

```
In [224]: obj.index.is_unique
Out[224]: 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 [225]: obj['a']
Out[225]:
a    0
a    1
dtype: int64

In [226]: obj['c']
Out[226]: 4
```

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

The same logic extends to indexing rows in a DataFrame:

```
In [227]: df = pd.DataFrame(np.random.randn(4, 3), index=['a', 'a', 'b', 'b'])

In [228]: df
Out[228]:
          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

In [229]: df.loc['b']
Out[229]:
          0         1         2
```

```
b  1.669025 -0.438570 -0.539741
b  0.476985  3.248944 -1.021228
```

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 [230]: 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 [231]: df
Out[231]:
    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 [232]: df.sum()
Out[232]:
one    9.25
two   -5.80
dtype: float64
```

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

```
In [233]: df.sum(axis='columns')
Out[233]:
a    1.40
b    2.60
c     NaN
d   -0.55
dtype: float64
```

NA values are excluded unless the entire slice (row or column in this case) is NA. This can be disabled with the `skipna` option:

```
In [234]: df.mean(axis='columns', skipna=False)
Out[234]:
a      NaN
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; 0 for DataFrame's rows and 1 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 [235]: df.idxmax()
Out[235]:
one    b
two    d
dtype: object
```

Other methods are *accumulations*:

```
In [236]: df.cumsum()
Out[236]:
    one  two
a  1.40  NaN
b  8.50 -4.5
c   NaN  NaN
d  9.25 -5.8
```

Another type of method is neither a reduction nor an accumulation. `describe` is one such example, producing multiple summary statistics in one shot:

```
In [237]: df.describe()
Out[237]:
            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 non-numeric data, `describe` produces alternative summary statistics:

```
In [238]: obj = pd.Series(['a', 'a', 'b', 'c'] * 4)

In [239]: obj.describe()
Out[239]:
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 for Series or each DataFrame column
`min, max`	Compute minimum and maximum values
`argmin, argmax`	Compute index locations (integers) at which minimum or maximum value obtained, respectively
`idxmin, idxmax`	Compute index labels at which minimum or maximum value obtained, respectively
`quantile`	Compute sample quantile ranging from 0 to 1
`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 obtained from Yahoo! Finance using the add-on `pandas-datareader` package. If you don't have it installed already, it can be obtained via conda or pip:

```
conda install pandas-datareader
```

I use the `pandas_datareader` module to download some data for a few stock tickers:

```
import pandas_datareader.data as web
all_data = {ticker: web.get_data_yahoo(ticker)
            for ticker in ['AAPL', 'IBM', 'MSFT', 'GOOG']}
```

```
price = pd.DataFrame({ticker: data['Adj Close']
                     for ticker, data in all_data.items()})
volume = pd.DataFrame({ticker: data['Volume']
                      for ticker, data in all_data.items()})
```

It's possible by the time you are reading this that Yahoo! Finance no longer exists since Yahoo! was acquired by Verizon in 2017. Refer to the pandas-datareader documentation online for the latest functionality.

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

```
In [242]: returns = price.pct_change()

In [243]: returns.tail()
Out[243]:
                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 [244]: returns['MSFT'].corr(returns['IBM'])
Out[244]: 0.49976361144151144

In [245]: returns['MSFT'].cov(returns['IBM'])
Out[245]: 8.8706554797035462e-05
```

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

```
In [246]: returns.MSFT.corr(returns.IBM)
Out[246]: 0.49976361144151144
```

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

```
In [247]: returns.corr()
Out[247]:
          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 [248]: returns.cov()
Out[248]:
          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 pairwise 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 [249]: returns.corrwith(returns.IBM)
Out[249]:
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 [250]: returns.corrwith(volume)
Out[250]:
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 [251]: 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 [252]: uniques = obj.unique()

In [253]: uniques
Out[253]: array(['c', 'a', 'd', 'b'], dtype=object)
```

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

```
In [254]: obj.value_counts()
Out[254]:
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 any array or sequence:

```
In [255]: pd.value_counts(obj.values, sort=False)
Out[255]:
a    3
b    2
c    3
d    1
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 [256]: obj
Out[256]:
0    c
1    a
2    d
3    a
4    a
5    b
6    b
7    c
8    c
dtype: object

In [257]: mask = obj.isin(['b', 'c'])

In [258]: mask
Out[258]:
0     True
1    False
2    False
3    False
4    False
5     True
6     True
7     True
8     True
dtype: bool

In [259]: obj[mask]
Out[259]:
```

```
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 non-distinct values into another array of distinct values:

```
In [260]: to_match = pd.Series(['c', 'a', 'b', 'b', 'c', 'a'])

In [261]: unique_vals = pd.Series(['c', 'b', 'a'])

In [262]: pd.Index(unique_vals).get_indexer(to_match)
Out[262]: 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 boolean array indicating whether each Series value is contained in the passed sequence of values
`match`	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 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 [263]: data = pd.DataFrame({'Qu1': [1, 3, 4, 3, 4],
   .....:                      'Qu2': [2, 3, 1, 2, 3],
   .....:                      'Qu3': [1, 5, 2, 4, 4]})

In [264]: data
Out[264]:
   Qu1  Qu2  Qu3
0    1    2    1
1    3    3    5
2    4    1    2
3    3    2    4
4    4    3    4
```

Passing `pandas.value_counts` to this DataFrame's `apply` function gives:

```
In [265]: result = data.apply(pd.value_counts).fillna(0)

In [266]: result
Out[266]:
   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.

5.4 Conclusion

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

CHAPTER 6

Data Loading, Storage, and File Formats

Accessing data is a necessary first step for using most of the tools in this book. I'm going to be focused 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 falls 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, though `read_csv` and `read_table` are likely the ones you'll use the most.

Table 6-1. Parsing functions in pandas

Function	Description
`read_csv`	Load delimited data from a file, URL, or file-like object; use comma as default delimiter
`read_table`	Load delimited data from a file, URL, or file-like object; use tab (`'\t'`) as default delimiter
`read_fwf`	Read data in fixed-width column format (i.e., no delimiters)
`read_clipboard`	Version of `read_table` 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
`read_msgpack`	Read pandas data encoded using the MessagePack binary format
`read_pickle`	Read an arbitrary object stored in Python pickle format

Function	Description
`read_sas`	Read a SAS dataset stored in one of the SAS system's custom storage formats
`read_sql`	Read the results of a SQL query (using SQLAlchemy) as a pandas DataFrame
`read_stata`	Read a dataset from Stata file format
`read_feather`	Read the Feather binary file format

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, the user, or not at all.

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

Datetime parsing
: Includes 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
: 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 `read_csv`) have grown very complex in their options over time. It's normal to feel overwhelmed by the number of different parameters (`read_csv` has over 50 as of this writing). The online pandas documentation has many examples about how each of them 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, like `pandas.read_csv`, 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, Feather, and msgpack, have the data types stored in the format.

Handling dates and other custom types can require extra effort. Let's start with a small comma-separated (CSV) text file:

```
In [8]: !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.

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

```
In [9]: df = pd.read_csv('examples/ex1.csv')

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

We could also have used `read_table` and specified the delimiter:

```
In [11]: pd.read_table('examples/ex1.csv', sep=',')
Out[11]:
   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 [12]: !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 [13]: pd.read_csv('examples/ex2.csv', header=None)
Out[13]:
   0   1   2   3      4
0  1   2   3   4  hello
1  5   6   7   8  world
2  9  10  11  12    foo

In [14]: pd.read_csv('examples/ex2.csv', names=['a', 'b', 'c', 'd', 'message'])
Out[14]:
   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 [15]: names = ['a', 'b', 'c', 'd', 'message']

In [16]: pd.read_csv('examples/ex2.csv', names=names, index_col='message')
Out[16]:
         a   b   c   d
message
hello    1   2   3   4
world    5   6   7   8
foo      9  10  11  12
```

In the event that you want to form a hierarchical index from multiple columns, pass a list of column numbers or names:

```
In [17]: !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 [18]: parsed = pd.read_csv('examples/csv_mindex.csv',
   ....:                      index_col=['key1', 'key2'])

In [19]: parsed
Out[19]:
           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 [20]: list(open('examples/ex3.txt'))
Out[20]:
['            A         B         C\n',
 'aaa -0.264438 -1.026059 -0.619500\n',
 'bbb  0.927272  0.302904 -0.032399\n',
 'ccc -0.264273 -0.386314 -0.217601\n',
 'ddd -0.871858 -0.348382  1.100491\n']
```

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 `read_table`. This can be expressed by the regular expression `\s+`, so we have then:

```
In [21]: result = pd.read_table('examples/ex3.txt', sep='\s+')

In [22]: result
Out[22]: 
            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, `read_table` infers that the first column should be the DataFrame's index in this special case.

The parser 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 [23]: !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 [24]: pd.read_csv('examples/ex4.csv', skiprows=[0, 2, 3])
Out[24]: 
   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 parsing process. Missing data is usually either not present (empty string) or marked by some *sentinel* value. By default, pandas uses a set of commonly occurring sentinels, such as `NA` and `NULL`:

```
In [25]: !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 [26]: result = pd.read_csv('examples/ex5.csv')
In [27]: result
Out[27]:
  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

In [28]: pd.isnull(result)
Out[28]:
   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 can take either a list or set of strings to consider missing values:

```
In [29]: result = pd.read_csv('examples/ex5.csv', na_values=['NULL'])

In [30]: result
Out[30]:
  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
```

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

```
In [31]: sentinels = {'message': ['foo', 'NA'], 'something': ['two']}

In [32]: pd.read_csv('examples/ex5.csv', na_values=sentinels)
Out[32]:
  something  a   b     c   d message
0       one  1   2   3.0   4     NaN
1       NaN  5   6   NaN   8   world
2     three  9  10  11.0  12     NaN
```

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

Table 6-2. Some read_csv/read_table 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, combine with `header=None`

Argument	Description
`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.
`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 name 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`	Dict containing column number of 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.
`iterator`	Return a `TextParser` object for reading file piecemeal.
`chunksize`	For iteration, size of file chunks.
`skip_footer`	Number of lines to ignore at end of file.
`verbose`	Print various parser output information, like the number of missing values placed in non-numeric columns.
`encoding`	Text encoding for Unicode (e.g., `'utf-8'` for UTF-8 encoded text).
`squeeze`	If the parsed data only contains one column, return a Series.
`thousands`	Separator for thousands (e.g., `','` or `'.'`).

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 only want to read in 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 [33]: pd.options.display.max_rows = 10
```

Now we have:

```
In [34]: result = pd.read_csv('examples/ex6.csv')

In [35]: result
Out[35]:
           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]
```

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

```
In [36]: pd.read_csv('examples/ex6.csv', nrows=5)
Out[36]:
        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 [37]: chunker = pd.read_csv('examples/ex6.csv', chunksize=1000)

In [38]: chunker
Out[38]: <pandas.io.parsers.TextFileReader at 0x7f6b1e2672e8>
```

The `TextParser` object returned by `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([])
for piece in chunker:
    tot = tot.add(piece['key'].value_counts(), fill_value=0)

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

We have then:

```
In [40]: tot[:10]
Out[40]:
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
```

TextParser 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 [41]: data = pd.read_csv('examples/ex5.csv')

In [42]: data
Out[42]:
  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 [43]: data.to_csv('examples/out.csv')

In [44]: !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):

```
In [45]: import sys

In [46]: 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 [47]: 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 [48]: 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 [49]: data.to_csv(sys.stdout, index=False, columns=['a', 'b', 'c'])
a,b,c
1,2,3.0
5,6,
9,10,11.0
```

Series also has a `to_csv` method:

```
In [50]: dates = pd.date_range('1/1/2000', periods=7)

In [51]: ts = pd.Series(np.arange(7), index=dates)

In [52]: ts.to_csv('examples/tseries.csv')

In [53]: !cat examples/tseries.csv
2000-01-01,0
2000-01-02,1
2000-01-03,2
2000-01-04,3
2000-01-05,4
2000-01-06,5
2000-01-07,6
```

Working with Delimited Formats

It's possible to load most forms of tabular data from disk using functions like `pandas.read_table`. 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 `read_table`. To illustrate the basic tools, consider a small CSV file:

```
In [54]: !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`:

```
import csv
f = open('examples/ex7.csv')

reader = csv.reader(f)
```

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

```
In [56]: for line in reader:
   ....:     print(line)
['a', 'b', 'c']
['1', '2', '3']
['1', '2', '3']
```

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

```
In [57]: 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 [58]: header, values = lines[0], lines[1:]
```

Then we can create a dictionary of data columns using a dictionary comprehension and the expression `zip(*values)`, which transposes rows to columns:

```
In [59]: data_dict = {h: v for h, v in zip(header, zip(*values))}

In [60]: data_dict
Out[60]: {'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 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 can 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 `'"'`.

Argument	Description
`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 string's `split` method or the regular expression method `re.split`.

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",
 "places_lived": ["United States", "Spain", "Germany"],
 "pet": null,
 "siblings": [{"name": "Scott", "age": 30, "pets": ["Zeus", "Zuko"]},
              {"name": "Katie", "age": 38,
               "pets": ["Sixes", "Stache", "Cisco"]}]
}
"""
```

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 (dicts), 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 [62]: import json

In [63]: result = json.loads(obj)

In [64]: result
Out[64]:
{'name': 'Wes',
 'pet': None,
 'places_lived': ['United States', 'Spain', 'Germany'],
 'siblings': [{'age': 30, 'name': 'Scott', 'pets': ['Zeus', 'Zuko']},
  {'age': 38, 'name': 'Katie', 'pets': ['Sixes', 'Stache', 'Cisco']}]}
```

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

```
In [65]: asjson = json.dumps(result)
```

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 dicts (which were previously JSON objects) to the DataFrame constructor and select a subset of the data fields:

```
In [66]: siblings = pd.DataFrame(result['siblings'], columns=['name', 'age'])

In [67]: siblings
Out[67]:
    name  age
0  Scott   30
1  Katie   38
```

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

```
In [68]: !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 [69]: data = pd.read_json('examples/example.json')

In [70]: data
Out[70]:
   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 7.

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

```
In [71]: print(data.to_json())
{"a":{"0":1,"1":4,"2":7},"b":{"0":2,"1":5,"2":8},"c":{"0":3,"1":6,"2":9}}

In [72]: print(data.to_json(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, `read_html`, which uses libraries like lxml and Beautiful Soup 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 United States FDIC government agency showing bank failures.[1] First, you must install some additional libraries used by `read_html`:

```
conda install lxml
pip install beautifulsoup4 html5lib
```

If you are not using conda, `pip install lxml` will likely 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 [73]: tables = pd.read_html('examples/fdic_failed_bank_list.html')

In [74]: len(tables)
Out[74]: 1

In [75]: failures = tables[0]

In [76]: failures.head()
Out[76]: 
                      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   
```

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

```
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
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 [77]: close_timestamps = pd.to_datetime(failures['Closing Date'])

In [78]: close_timestamps.dt.year.value_counts()
Out[78]:
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 (eXtensible Markup Language) 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.

The New York Metropolitan Transportation Authority (MTA) publishes a number of data series about its bus and train services. 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`:

```
from lxml import objectify

path = 'examples/mta_perf/Performance_MNR.xml'
parsed = objectify.parse(open(path))
root = parsed.getroot()
```

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

```
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 dicts into a DataFrame:

```
In [81]: perf = pd.DataFrame(data)

In [82]: perf.head()
Out[82]:
Empty DataFrame
```

```
Columns: []
Index: []
```

XML data can get much more complicated than this example. Each tag can have metadata, too. Consider an HTML link tag, which is also valid XML:

```
from io import StringIO
tag = '<a href="http://www.google.com">Google</a>'
root = objectify.parse(StringIO(tag)).getroot()
```

You can now access any of the fields (like `href`) in the tag or the link text:

```
In [84]: root
Out[84]: <Element a at 0x7f6b15817748>

In [85]: root.get('href')
Out[85]: 'http://www.google.com'

In [86]: root.text
Out[86]: 'Google'
```

6.2 Binary Data Formats

One of the easiest ways to store data (also known as *serialization*) efficiently in binary format is using Python's built-in `pickle` serialization. pandas objects all have a `to_pickle` method that writes the data to disk in pickle format:

```
In [87]: frame = pd.read_csv('examples/ex1.csv')

In [88]: frame
Out[88]:
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo

In [89]: frame.to_pickle('examples/frame_pickle')
```

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 [90]: pd.read_pickle('examples/frame_pickle')
Out[90]:
   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 only recommended 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. We have 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 two more binary data formats: HDF5 and MessagePack. I will give some HDF5 examples in the next section, but I encourage you to explore different file formats to see how fast they are and how well they work for your analysis. Some other storage formats for pandas or NumPy data include:

bcolz
: A compressable column-oriented binary format based on the Blosc compression library.

Feather
: A cross-language column-oriented file format I designed with the R programming community's Hadley Wickham. Feather uses the Apache Arrow columnar memory format.

Using HDF5 Format

HDF5 is a well-regarded 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 very large datasets that don't fit into memory, as you can efficiently read and write small sections of much larger arrays.

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 object. The `HDFStore` class works like a dict and handles the low-level details:

```
In [92]: frame = pd.DataFrame({'a': np.random.randn(100)})

In [93]: store = pd.HDFStore('mydata.h5')

In [94]: store['obj1'] = frame

In [95]: store['obj1_col'] = frame['a']

In [96]: store
```

```
Out[96]:
<class 'pandas.io.pytables.HDFStore'>
File path: mydata.h5
/obj1                frame        (shape->[100,1])

/obj1_col            series       (shape->[100])

/obj2                frame_table  (typ->appendable,nrows->100,ncols->1,indexers->
[index])
/obj3                frame_table  (typ->appendable,nrows->100,ncols->1,indexers->
[index])
```

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

```
In [97]: store['obj1']
Out[97]:
           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 latter is generally slower, but it supports query operations using a special syntax:

```
In [98]: store.put('obj2', frame, format='table')

In [99]: store.select('obj2', where=['index >= 10 and index <= 15'])
Out[99]:
           a
10  1.007189
11 -1.296221
12  0.274992
13  0.228913
14  1.352917
15  0.886429

In [100]: 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 [101]: frame.to_hdf('mydata.h5', 'obj3', format='table')
In [102]: pd.read_hdf('mydata.h5', 'obj3', where=['index < 5'])
Out[102]:
          a
0 -0.204708
1  0.478943
2 -0.519439
3 -0.555730
4  1.965781
```

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. Python for Parquet and other such storage formats is still developing, so I do not write about them in this book.

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.

Reading Microsoft Excel Files

pandas also supports reading tabular data stored in Excel 2003 (and higher) files using either the `ExcelFile` class or `pandas.read_excel` function. Internally these tools use the add-on packages `xlrd` and `openpyxl` to read XLS and XLSX files, respectively. You may need to install these manually with pip or conda.

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

```
In [104]: xlsx = pd.ExcelFile('examples/ex1.xlsx')
```

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

```
In [105]: pd.read_excel(xlsx, 'Sheet1')
Out[105]:
   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 `ExcelFile`, but you can also simply pass the filename to `pandas.read_excel`:

```
In [106]: frame = pd.read_excel('examples/ex1.xlsx', 'Sheet1')

In [107]: frame
Out[107]:
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
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 pandas objects' `to_excel` method:

```
In [108]: writer = pd.ExcelWriter('examples/ex2.xlsx')

In [109]: frame.to_excel(writer, 'Sheet1')

In [110]: writer.save()
```

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

```
In [111]: frame.to_excel('examples/ex2.xlsx')
```

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 easy-to-use method that I recommend is the `requests` package.

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 [113]: import requests

In [114]: url = 'https://api.github.com/repos/pandas-dev/pandas/issues'

In [115]: resp = requests.get(url)

In [116]: resp
Out[116]: <Response [200]>
```

The Response object's `json` method will return a dictionary containing JSON parsed into native Python objects:

```
In [117]: data = resp.json()

In [118]: data[0]['title']
Out[118]: 'Period does not round down for frequencies less that 1 hour'
```

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 DataFrame and extract fields of interest:

```
In [119]: issues = pd.DataFrame(data, columns=['number', 'title',
   .....:                                          'labels', 'state'])

In [120]: issues
Out[120]:
    number                                            title  \
0    17666  Period does not round down for frequencies les...
1    17665            DOC: improve docstring of function where
2    17664                COMPAT: skip 32-bit test on int repr
3    17662                           implement Delegator class
4    17654  BUG: Fix series rename called with str alterin...
..     ...                                              ...
25   17603  BUG: Correctly localize naive datetime strings...
26   17599                     core.dtypes.generic --> cython
27   17596   Merge cdate_range functionality into bdate_range
28   17587  Time Grouper bug fix when applied for list gro...
29   17583  BUG: fix tz-aware DatetimeIndex + TimedeltaInd...
                                               labels state
0                                                  []  open
1   [{'id': 134699, 'url': 'https://api.github.com...  open
2   [{'id': 563047854, 'url': 'https://api.github....  open
3                                                  []  open
4   [{'id': 76811, 'url': 'https://api.github.com/...  open
..                                                ...   ...
25  [{'id': 76811, 'url': 'https://api.github.com/...  open
26  [{'id': 49094459, 'url': 'https://api.github.c...  open
27  [{'id': 35818298, 'url': 'https://api.github.c...  open
28  [{'id': 233160, 'url': 'https://api.github.com...  open
29  [{'id': 76811, 'url': 'https://api.github.com/...  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 easy analysis.

6.4 Interacting with Databases

In a business setting, most 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.

Loading data from SQL into a DataFrame is fairly straightforward, and pandas has some functions to simplify the process. As an example, I'll create a SQLite database using Python's built-in `sqlite3` driver:

```
In [121]: import sqlite3

In [122]: query = """
   .....: CREATE TABLE test
   .....: (a VARCHAR(20), b VARCHAR(20),
   .....:  c REAL,        d INTEGER
   .....: );"""

In [123]: con = sqlite3.connect('mydata.sqlite')

In [124]: con.execute(query)
Out[124]: <sqlite3.Cursor at 0x7f6b12a50f10>

In [125]: con.commit()
```

Then, insert a few rows of data:

```
In [126]: data = [('Atlanta', 'Georgia', 1.25, 6),
   .....:         ('Tallahassee', 'Florida', 2.6, 3),
   .....:         ('Sacramento', 'California', 1.7, 5)]

In [127]: stmt = "INSERT INTO test VALUES(?, ?, ?, ?)"

In [128]: con.executemany(stmt, data)
Out[128]: <sqlite3.Cursor at 0x7f6b15c66ce0>

In [129]: con.commit()
```

Most Python SQL drivers (PyODBC, psycopg2, MySQLdb, pymssql, etc.) return a list of tuples when selecting data from a table:

```
In [130]: cursor = con.execute('select * from test')

In [131]: rows = cursor.fetchall()

In [132]: rows
Out[132]: 
[('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:

```
In [133]: cursor.description
Out[133]: 
(('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 [134]: pd.DataFrame(rows, columns=[x[0] for x in cursor.description])
Out[134]: 
             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. Here, we'll connect to the same SQLite database with SQLAlchemy and read data from the table created before:

```
In [135]: import sqlalchemy as sqla

In [136]: db = sqla.create_engine('sqlite:///mydata.sqlite')

In [137]: pd.read_sql('select * from test', db)
Out[137]:
             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 has been driven by the needs of real-world applications.

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

7.1 Handling Missing Data

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

The way that missing data is represented in pandas objects is somewhat imperfect, but it is functional for a lot of users. For numeric data, pandas uses the floating-point value `NaN` (Not a Number) to represent missing data. We call this a *sentinel value* that can be easily detected:

```
In [10]: string_data = pd.Series(['aardvark', 'artichoke', np.nan, 'avocado'])

In [11]: string_data
Out[11]:
0     aardvark
1    artichoke
2          NaN
3      avocado
dtype: object

In [12]: string_data.isnull()
Out[12]:
0    False
1    False
2     True
3    False
dtype: bool
```

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

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

```
In [13]: string_data[0] = None

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

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

Table 7-1. NA handling methods

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

Filtering Out Missing Data

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

```
In [15]: from numpy import nan as NA

In [16]: data = pd.Series([1, NA, 3.5, NA, 7])

In [17]: data.dropna()
Out[17]: 
0    1.0
2    3.5
4    7.0
dtype: float64
```

This is equivalent to:

```
In [18]: data[data.notnull()]
Out[18]: 
0    1.0
2    3.5
4    7.0
dtype: float64
```

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

```
In [19]: data = pd.DataFrame([[1., 6.5, 3.], [1., NA, NA],
   ....:                      [NA, NA, NA], [NA, 6.5, 3.]])

In [20]: cleaned = data.dropna()

In [21]: data
Out[21]: 
     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 [22]: cleaned
Out[22]: 
     0    1    2
0  1.0  6.5  3.0
```

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

```
In [23]: data.dropna(how='all')
Out[23]: 
     0    1    2
```

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

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

```
In [24]: data[4] = NA

In [25]: data
Out[25]:
     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 [26]: data.dropna(axis=1, how='all')
Out[26]:
     0    1    2
0  1.0  6.5  3.0
1  1.0  NaN  NaN
2  NaN  NaN  NaN
3  NaN  6.5  3.0
```

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

```
In [27]: df = pd.DataFrame(np.random.randn(7, 3))

In [28]: df.iloc[:4, 1] = NA

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

In [30]: df
Out[30]:
          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 [31]: df.dropna()
Out[31]:
          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 [32]: df.dropna(thresh=2)
```

```
Out[32]:
          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 [33]: df.fillna(0)
Out[33]:
          0         1         2
0 -0.204708  0.000000  0.000000
1 -0.555730  0.000000  0.000000
2  0.092908  0.000000  0.769023
3  1.246435  0.000000 -1.296221
4  0.274992  0.228913  1.352917
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741
```

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

```
In [34]: df.fillna({1: 0.5, 2: 0})
Out[34]:
          0         1         2
0 -0.204708  0.500000  0.000000
1 -0.555730  0.500000  0.000000
2  0.092908  0.500000  0.769023
3  1.246435  0.500000 -1.296221
4  0.274992  0.228913  1.352917
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741
```

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

```
In [35]: _ = df.fillna(0, inplace=True)

In [36]: df
Out[36]:
          0         1         2
0 -0.204708  0.000000  0.000000
1 -0.555730  0.000000  0.000000
2  0.092908  0.000000  0.769023
3  1.246435  0.000000 -1.296221
4  0.274992  0.228913  1.352917
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741
```

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

```
In [37]: df = pd.DataFrame(np.random.randn(6, 3))

In [38]: df.iloc[2:, 1] = NA

In [39]: df.iloc[4:, 2] = NA

In [40]: df
Out[40]:
          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 [41]: df.fillna(method='ffill')
Out[41]:
          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 [42]: df.fillna(method='ffill', limit=2)
Out[42]:
          0         1         2
0  0.476985  3.248944 -1.021228
1 -0.577087  0.124121  0.302614
2  0.523772  0.124121  1.343810
3 -0.713544  0.124121 -2.370232
4 -1.860761       NaN -2.370232
5 -1.265934       NaN -2.370232
```

With `fillna` you can do lots of other things with a little creativity. For example, you might pass the mean or median value of a Series:

```
In [43]: data = pd.Series([1., NA, 3.5, NA, 7])

In [44]: data.fillna(data.mean())
Out[44]:
0    1.000000
1    3.833333
2    3.500000
3    3.833333
4    7.000000
dtype: float64
```

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

Table 7-2. fillna function arguments

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

7.2 Data Transformation

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

Removing Duplicates

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

```
In [45]: data = pd.DataFrame({'k1': ['one', 'two'] * 3 + ['two'],
   ....:                      'k2': [1, 1, 2, 3, 3, 4, 4]})

In [46]: data
Out[46]: 
    k1  k2
0  one   1
1  two   1
2  one   2
3  two   3
4  one   3
5  two   4
6  two   4
```

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

```
In [47]: data.duplicated()
Out[47]: 
0    False
1    False
2    False
3    False
4    False
5    False
6     True
dtype: bool
```

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

```
In [48]: data.drop_duplicates()
Out[48]:
    k1  k2
0  one   1
1  two   1
2  one   2
3  two   3
4  one   3
5  two   4
```

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

```
In [49]: data['v1'] = range(7)

In [50]: data.drop_duplicates(['k1'])
Out[50]:
    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 [51]: data.drop_duplicates(['k1', 'k2'], keep='last')
Out[51]:
    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 [52]: 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 [53]: data
Out[53]:
          food  ounces
0        bacon     4.0
1  pulled pork     3.0
2        bacon    12.0
```

```
3     Pastrami     6.0
4  corned beef     7.5
5        Bacon     8.0
6     pastrami     3.0
7    honey ham     5.0
8     nova lox     6.0
```

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

```
meat_to_animal = {
  'bacon': 'pig',
  'pulled pork': 'pig',
  'pastrami': 'cow',
  'corned beef': 'cow',
  'honey ham': 'pig',
  'nova lox': 'salmon'
}
```

The `map` method on a Series accepts a function or dict-like object containing a mapping, but here we have a small problem in that some of the meats are capitalized and others are not. Thus, we need to convert each value to lowercase using the `str.lower` Series method:

```
In [55]: lowercased = data['food'].str.lower()

In [56]: lowercased
Out[56]:
0          bacon
1    pulled pork
2          bacon
3       pastrami
4    corned beef
5          bacon
6       pastrami
7      honey ham
8       nova lox
Name: food, dtype: object

In [57]: data['animal'] = lowercased.map(meat_to_animal)

In [58]: data
Out[58]:
          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 [59]: data['food'].map(lambda x: meat_to_animal[x.lower()])
Out[59]:
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 [60]: data = pd.Series([1., -999., 2., -999., -1000., 3.])

In [61]: data
Out[61]:
0       1.0
1    -999.0
2       2.0
3    -999.0
4   -1000.0
5       3.0
dtype: float64
```

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

```
In [62]: data.replace(-999, np.nan)
Out[62]:
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 [63]: data.replace([-999, -1000], np.nan)
Out[63]:
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 [64]: data.replace([-999, -1000], [np.nan, 0])
Out[64]:
0    1.0
1    NaN
2    2.0
3    NaN
4    0.0
5    3.0
dtype: float64
```

The argument passed can also be a dict:

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

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

Renaming Axis Indexes

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

```
In [66]: 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 [67]: transform = lambda x: x[:4].upper()

In [68]: data.index.map(transform)
Out[68]: Index(['OHIO', 'COLO', 'NEW '], dtype='object')
```

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

```
In [69]: data.index = data.index.map(transform)

In [70]: data
Out[70]:
      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 [71]: data.rename(index=str.title, columns=str.upper)
Out[71]:
      ONE  TWO  THREE  FOUR
Ohio    0    1      2     3
Colo    4    5      6     7
New     8    9     10    11
```

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

```
In [72]: data.rename(index={'OHIO': 'INDIANA'},
   ....:             columns={'three': 'peekaboo'})
Out[72]:
         one  two  peekaboo  four
INDIANA    0    1         2     3
COLO       4    5         6     7
NEW        8    9        10    11
```

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

```
In [73]: data.rename(index={'OHIO': 'INDIANA'}, inplace=True)

In [74]: data
Out[74]:
         one  two  three  four
INDIANA    0    1      2     3
```

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

Discretization and Binning

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

```
In [75]: ages = [20, 22, 25, 27, 21, 23, 37, 31, 61, 45, 41, 32]
```

Let's divide these into bins of 18 to 25, 26 to 35, 36 to 60, and finally 61 and older. To do so, you have to use `cut`, a function in pandas:

```
In [76]: bins = [18, 25, 35, 60, 100]

In [77]: cats = pd.cut(ages, bins)

In [78]: cats
Out[78]:
[(18, 25], (18, 25], (18, 25], (25, 35], (18, 25], ..., (25, 35], (60, 100], (35,
 60], (35, 60], (25, 35]]
Length: 12
Categories (4, interval[int64]): [(18, 25] < (25, 35] < (35, 60] < (60, 100]]
```

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

```
In [79]: cats.codes
Out[79]: array([0, 0, 0, 1, 0, 0, 2, 1, 3, 2, 2, 1], dtype=int8)

In [80]: cats.categories
Out[80]:
IntervalIndex([(18, 25], (25, 35], (35, 60], (60, 100]]
              closed='right',
              dtype='interval[int64]')

In [81]: pd.value_counts(cats)
Out[81]:
(18, 25]     5
(35, 60]     3
(25, 35]     3
(60, 100]    1
dtype: int64
```

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

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

```
In [82]: pd.cut(ages, [18, 26, 36, 61, 100], right=False)
Out[82]:
[[18, 26), [18, 26), [18, 26), [26, 36), [18, 26), ..., [26, 36), [61, 100), [36,
 61), [36, 61), [26, 36)]
Length: 12
Categories (4, interval[int64]): [[18, 26) < [26, 36) < [36, 61) < [61, 100)]
```

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

```
In [83]: group_names = ['Youth', 'YoungAdult', 'MiddleAged', 'Senior']

In [84]: pd.cut(ages, bins, labels=group_names)
Out[84]:
[Youth, Youth, Youth, YoungAdult, Youth, ..., YoungAdult, Senior, MiddleAged, Mid
dleAged, YoungAdult]
Length: 12
Categories (4, object): [Youth < YoungAdult < MiddleAged < Senior]
```

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

```
In [85]: data = np.random.rand(20)

In [86]: pd.cut(data, 4, precision=2)
Out[86]:
[(0.34, 0.55], (0.34, 0.55], (0.76, 0.97], (0.76, 0.97], (0.34, 0.55], ..., (0.34
, 0.55], (0.34, 0.55], (0.55, 0.76], (0.34, 0.55], (0.12, 0.34]]
Length: 20
Categories (4, interval[float64]): [(0.12, 0.34] < (0.34, 0.55] < (0.55, 0.76] <
(0.76, 0.97]]
```

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

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

```
In [87]: data = np.random.randn(1000)  # Normally distributed

In [88]: cats = pd.qcut(data, 4)  # Cut into quartiles

In [89]: cats
Out[89]:
[(-0.0265, 0.62], (0.62, 3.928], (-0.68, -0.0265], (0.62, 3.928], (-0.0265, 0.62]
, ..., (-0.68, -0.0265], (-0.68, -0.0265], (-2.95, -0.68], (0.62, 3.928], (-0.68,
 -0.0265]]
Length: 1000
Categories (4, interval[float64]): [(-2.95, -0.68] < (-0.68, -0.0265] < (-0.0265,
 0.62] <
                                    (0.62, 3.928]]
In [90]: pd.value_counts(cats)
Out[90]:
(0.62, 3.928]       250
(-0.0265, 0.62]     250
(-0.68, -0.0265]    250
(-2.95, -0.68]      250
dtype: int64
```

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

```
In [91]: pd.qcut(data, [0, 0.1, 0.5, 0.9, 1.])
Out[91]:
[(-0.0265, 1.286], (-0.0265, 1.286], (-1.187, -0.0265], (-0.0265, 1.286], (-0.026
5, 1.286], ..., (-1.187, -0.0265], (-1.187, -0.0265], (-2.95, -1.187], (-0.0265,
1.286], (-1.187, -0.0265]]
Length: 1000
Categories (4, interval[float64]): [(-2.95, -1.187] < (-1.187, -0.0265] < (-0.026
5, 1.286] <
                                    (1.286, 3.928]]
```

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

Detecting and Filtering Outliers

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

```
In [92]: data = pd.DataFrame(np.random.randn(1000, 4))

In [93]: data.describe()
Out[93]:
                 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 [94]: col = data[2]

In [95]: col[np.abs(col) > 3]
Out[95]:
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 [96]: data[(np.abs(data) > 3).any(1)]
Out[96]:
            0         1         2         3
41   0.457246 -0.025907 -3.399312 -0.974657
60   1.951312  3.260383  0.963301  1.201206
136  0.508391 -0.196713 -3.745356 -1.520113
235 -0.242459 -3.056990  1.918403 -0.578828
258  0.682841  0.326045  0.425384 -3.428254
322  1.179227 -3.184377  1.369891 -1.074833
544 -3.548824  1.553205 -2.186301  1.277104
635 -0.578093  0.193299  1.397822  3.366626
782 -0.207434  3.525865  0.283070  0.544635
803 -3.645860  0.255475 -0.549574 -1.907459
```

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

```
In [97]: data[np.abs(data) > 3] = np.sign(data) * 3

In [98]: data.describe()
Out[98]:
                 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 [99]: np.sign(data).head()
Out[99]:
     0    1    2    3
0 -1.0  1.0 -1.0  1.0
1  1.0 -1.0  1.0 -1.0
2  1.0  1.0  1.0 -1.0
3 -1.0 -1.0  1.0 -1.0
4 -1.0  1.0 -1.0 -1.0
```

Permutation and Random Sampling

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

```
In [100]: df = pd.DataFrame(np.arange(5 * 4).reshape((5, 4)))

In [101]: sampler = np.random.permutation(5)

In [102]: sampler
Out[102]: array([3, 1, 4, 2, 0])
```

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

```
In [103]: df
Out[103]:
    0   1   2   3
0   0   1   2   3
1   4   5   6   7
2   8   9  10  11
3  12  13  14  15
4  16  17  18  19

In [104]: df.take(sampler)
Out[104]:
    0   1   2   3
3  12  13  14  15
1   4   5   6   7
4  16  17  18  19
2   8   9  10  11
0   0   1   2   3
```

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

```
In [105]: df.sample(n=3)
Out[105]:
    0   1   2   3
3  12  13  14  15
4  16  17  18  19
2   8   9  10  11
```

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

```
In [106]: choices = pd.Series([5, 7, -1, 6, 4])

In [107]: draws = choices.sample(n=10, replace=True)

In [108]: draws
Out[108]:
4    4
1    7
4    4
2   -1
0    5
3    6
1    7
```

```
4    4
0    5
4    4
dtype: int64
```

Computing Indicator/Dummy Variables

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

```
In [109]: df = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'b'],
   .....:                    'data1': range(6)})

In [110]: pd.get_dummies(df['key'])
Out[110]:
   a  b  c
0  0  1  0
1  0  1  0
2  1  0  0
3  0  0  1
4  1  0  0
5  0  1  0
```

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

```
In [111]: dummies = pd.get_dummies(df['key'], prefix='key')

In [112]: df_with_dummy = df[['data1']].join(dummies)

In [113]: df_with_dummy
Out[113]:
   data1  key_a  key_b  key_c
0      0      0      1      0
1      1      0      1      0
2      2      1      0      0
3      3      0      0      1
4      4      1      0      0
5      5      0      1      0
```

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

```
In [114]: mnames = ['movie_id', 'title', 'genres']

In [115]: movies = pd.read_table('datasets/movielens/movies.dat', sep='::',
   .....:                        header=None, names=mnames)

In [116]: movies[:10]
Out[116]:
   movie_id                               title                        genres
0         1                    Toy Story (1995)   Animation|Children's|Comedy
1         2                      Jumanji (1995)  Adventure|Children's|Fantasy
2         3             Grumpier Old Men (1995)                Comedy|Romance
3         4            Waiting to Exhale (1995)                  Comedy|Drama
4         5  Father of the Bride Part II (1995)                        Comedy
5         6                         Heat (1995)         Action|Crime|Thriller
6         7                      Sabrina (1995)                Comedy|Romance
7         8                 Tom and Huck (1995)          Adventure|Children's
8         9                 Sudden Death (1995)                        Action
9        10                    GoldenEye (1995)     Action|Adventure|Thriller
```

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

```
In [117]: all_genres = []

In [118]: for x in movies.genres:
   .....:     all_genres.extend(x.split('|'))

In [119]: genres = pd.unique(all_genres)
```

Now we have:

```
In [120]: genres
Out[120]:
array(['Animation', "Children's", 'Comedy', 'Adventure', 'Fantasy',
       'Romance', 'Drama', 'Action', 'Crime', 'Thriller', 'Horror',
       'Sci-Fi', 'Documentary', 'War', 'Musical', 'Mystery', 'Film-Noir',
       'Western'], dtype=object)
```

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

```
In [121]: zero_matrix = np.zeros((len(movies), len(genres)))

In [122]: dummies = pd.DataFrame(zero_matrix, columns=genres)
```

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

```
In [123]: gen = movies.genres[0]

In [124]: gen.split('|')
Out[124]: ['Animation', "Children's", 'Comedy']

In [125]: dummies.columns.get_indexer(gen.split('|'))
Out[125]: array([0, 1, 2])
```

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

```
In [126]: for i, gen in enumerate(movies.genres):
   .....:     indices = dummies.columns.get_indexer(gen.split('|'))
   .....:     dummies.iloc[i, indices] = 1
   .....:
```

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

```
In [127]: movies_windic = movies.join(dummies.add_prefix('Genre_'))

In [128]: movies_windic.iloc[0]
Out[128]:
movie_id                                       1
title                           Toy Story (1995)
genres               Animation|Children's|Comedy
Genre_Animation                                1
Genre_Children's                               1
Genre_Comedy                                   1
Genre_Adventure                                0
Genre_Fantasy                                  0
Genre_Romance                                  0
Genre_Drama                                    0
                                ...
Genre_Crime                                    0
Genre_Thriller                                 0
Genre_Horror                                   0
Genre_Sci-Fi                                   0
Genre_Documentary                              0
Genre_War                                      0
Genre_Musical                                  0
Genre_Mystery                                  0
Genre_Film-Noir                                0
Genre_Western                                  0
Name: 0, Length: 21, dtype: object
```

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

A useful recipe for statistical applications is to combine `get_dummies` with a discretization function like `cut`:

```
In [129]: np.random.seed(12345)

In [130]: values = np.random.rand(10)

In [131]: values
Out[131]:
array([ 0.9296,  0.3164,  0.1839,  0.2046,  0.5677,  0.5955,  0.9645,
        0.6532,  0.7489,  0.6536])

In [132]: bins = [0, 0.2, 0.4, 0.6, 0.8, 1]

In [133]: pd.get_dummies(pd.cut(values, bins))
Out[133]:
   (0.0, 0.2]  (0.2, 0.4]  (0.4, 0.6]  (0.6, 0.8]  (0.8, 1.0]
0           0           0           0           0           1
1           0           1           0           0           0
2           1           0           0           0           0
3           0           1           0           0           0
4           0           0           1           0           0
5           0           0           1           0           0
6           0           0           0           0           1
7           0           0           0           1           0
8           0           0           0           1           0
9           0           0           0           1           0
```

We set the random seed with `numpy.random.seed` to make the example deterministic. We will look again at `pandas.get_dummies` later in the book.

7.3 String Manipulation

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

String Object Methods

In many string munging and scripting applications, built-in string methods are sufficient. As an example, a comma-separated string can be broken into pieces with `split`:

```
In [134]: val = 'a,b,  guido'

In [135]: val.split(',')
Out[135]: ['a', 'b', '  guido']
```

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

```
In [136]: pieces = [x.strip() for x in val.split(',')]

In [137]: pieces
Out[137]: ['a', 'b', 'guido']
```

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

```
In [138]: first, second, third = pieces

In [139]: first + '::' + second + '::' + third
Out[139]: '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 [140]: '::'.join(pieces)
Out[140]: '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 [141]: 'guido' in val
Out[141]: True

In [142]: val.index(',')
Out[142]: 1

In [143]: val.find(':')
Out[143]: -1
```

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

```
In [144]: val.index(':')
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-144-280f8b2856ce> in <module>()
----> 1 val.index(':')
ValueError: substring not found
```

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

```
In [145]: val.count(',')
Out[145]: 2
```

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

```
In [146]: val.replace(',', '::')
Out[146]: 'a::b::  guido'

In [147]: val.replace(',', '')
Out[147]: 'ab  guido'
```

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

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

Table 7-3. Python built-in string methods

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

Regular Expressions

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

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

The `re` module functions fall into three categories: pattern matching, substitution, and splitting. Naturally these are all related; a regex describes a pattern to locate in the text, which can then be used for many purposes. Let's look at a simple example:

suppose we wanted to split a string with a variable number of whitespace characters (tabs, spaces, and newlines). The regex describing one or more whitespace characters is `\s+`:

```
In [148]: import re

In [149]: text = "foo    bar\t baz  \tqux"

In [150]: re.split('\s+', text)
Out[150]: ['foo', 'bar', 'baz', 'qux']
```

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

```
In [151]: regex = re.compile('\s+')

In [152]: regex.split(text)
Out[152]: ['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 [153]: regex.findall(text)
Out[153]: ['    ', '\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 [155]: regex.findall(text)
Out[155]:
['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 [156]: m = regex.search(text)

In [157]: m
Out[157]: <_sre.SRE_Match object; span=(5, 20), match='dave@google.com'>

In [158]: text[m.start():m.end()]
Out[158]: 'dave@google.com'
```

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

```
In [159]: print(regex.match(text))
None
```

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

```
In [160]: 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 [161]: pattern = r'([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\.([A-Z]{2,4})'

In [162]: 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 [163]: m = regex.match('wesm@bright.net')

In [164]: m.groups()
Out[164]: ('wesm', 'bright', 'net')
```

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

```
In [165]: regex.findall(text)
Out[165]:
```

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

There is much more to regular expressions in Python, most of which is outside the book's scope. Table 7-4 provides a brief summary.

Table 7-4. Regular expression methods

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

Vectorized String Functions in pandas

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

```
In [167]: data = {'Dave': 'dave@google.com', 'Steve': 'steve@gmail.com',
   .....:         'Rob': 'rob@gmail.com', 'Wes': np.nan}

In [168]: data = pd.Series(data)

In [169]: data
Out[169]:
Dave     dave@google.com
Rob        rob@gmail.com
Steve    steve@gmail.com
Wes                  NaN
dtype: object
```

```
In [170]: data.isnull()
Out[170]:
Dave     False
Rob      False
Steve    False
Wes       True
dtype: bool
```

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

```
In [171]: data.str.contains('gmail')
Out[171]:
Dave     False
Rob       True
Steve     True
Wes        NaN
dtype: object
```

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

```
In [172]: pattern
Out[172]: '([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\\.([A-Z]{2,4})'

In [173]: data.str.findall(pattern, flags=re.IGNORECASE)
Out[173]:
Dave     [(dave, google, com)]
Rob        [(rob, gmail, com)]
Steve    [(steve, 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 [174]: matches = data.str.match(pattern, flags=re.IGNORECASE)

In [175]: matches
Out[175]:
Dave     True
Rob      True
Steve    True
Wes       NaN
dtype: object
```

To access elements in the embedded lists, we can pass an index to either of these functions:

```
In [176]: matches.str.get(1)
Out[176]:
```

```
Dave     NaN
Rob      NaN
Steve    NaN
Wes      NaN
dtype: float64

In [177]: matches.str[0]
Out[177]:
Dave     NaN
Rob      NaN
Steve    NaN
Wes      NaN
dtype: float64
```

You can similarly slice strings using this syntax:

```
In [178]: data.str[:5]
Out[178]:
Dave      dave@
Rob       rob@g
Steve     steve
Wes         NaN
dtype: object
```

See Table 7-5 for more pandas string methods.

Table 7-5. Partial listing of vectorized string methods

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

Method	Description
`match`	Use `re.match` with the passed regular expression on each element, returning matched groups as list
`pad`	Add whitespace to left, right, or both sides of strings
`center`	Equivalent to `pad(side='both')`
`repeat`	Duplicate values (e.g., `s.str.repeat(3)` is equivalent to `x * 3` for each string)
`replace`	Replace occurrences of pattern/regex with some other string
`slice`	Slice each string in the Series
`split`	Split strings on delimiter or regular expression
`strip`	Trim whitespace from both sides, including newlines
`rstrip`	Trim whitespace on right side
`lstrip`	Trim whitespace on left side

7.4 Conclusion

Effective data preparation can significantly improve productive 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 easy 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 14.

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. Somewhat abstractly, 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 [9]: data = pd.Series(np.random.randn(9),
   ...:                  index=[['a', 'a', 'a', 'b', 'b', 'c', 'c', 'd', 'd'],
   ...:                         [1, 2, 3, 1, 3, 1, 2, 2, 3]])

In [10]: data
Out[10]:
a  1   -0.204708
   2    0.478943
   3   -0.519439
b  1   -0.555730
   3    1.965781
c  1    1.393406
   2    0.092908
d  2    0.281746
```

```
      3    0.769023
   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 [11]: data.index
Out[11]:
MultiIndex(levels=[['a', 'b', 'c', 'd'], [1, 2, 3]],
           labels=[[0, 0, 0, 1, 1, 2, 2, 3, 3], [0, 1, 2, 0, 2, 0, 1, 1, 2]])
```

With a hierarchically indexed object, so-called *partial* indexing is possible, enabling you to concisely select subsets of the data:

```
In [12]: data['b']
Out[12]:
1   -0.555730
3    1.965781
dtype: float64

In [13]: data['b':'c']
Out[13]:
b  1   -0.555730
   3    1.965781
c  1    1.393406
   2    0.092908
dtype: float64

In [14]: data.loc[['b', 'd']]
Out[14]:
b  1   -0.555730
   3    1.965781
d  2    0.281746
   3    0.769023
dtype: float64
```

Selection is even possible from an "inner" level:

```
In [15]: data.loc[:, 2]
Out[15]:
a    0.478943
c    0.092908
d    0.281746
dtype: float64
```

Hierarchical indexing plays an important role in reshaping data and group-based operations like forming a pivot table. For example, you could rearrange the data into a DataFrame using its `unstack` method:

```
In [16]: data.unstack()
Out[16]:
          1         2         3
a -0.204708  0.478943 -0.519439
b -0.555730       NaN  1.965781
```

```
c  1.393406  0.092908       NaN
d       NaN  0.281746  0.769023
```

The inverse operation of `unstack` is stack:

```
In [17]: data.unstack().stack()
Out[17]:
a  1   -0.204708
   2    0.478943
   3   -0.519439
b  1   -0.555730
   3    1.965781
c  1    1.393406
   2    0.092908
d  2    0.281746
   3    0.769023
dtype: float64
```

`stack` and `unstack` will be explored in more detail later in this chapter.

With a DataFrame, either axis can have a hierarchical index:

```
In [18]: 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 [19]: frame
Out[19]:
     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 [20]: frame.index.names = ['key1', 'key2']

In [21]: frame.columns.names = ['state', 'color']

In [22]: frame
Out[22]:
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
```

Be careful to distinguish the index names 'state' and 'color' from the row labels.

With partial column indexing you can similarly select groups of columns:

```
In [23]: frame['Ohio']
Out[23]:
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 be created like this:

```
MultiIndex.from_arrays([['Ohio', 'Ohio', 'Colorado'], ['Green', 'Red', 'Green']],
                       names=['state', 'color'])
```

Reordering and Sorting Levels

At times you will need to rearrange the order of the levels on an axis or sort the data by the values in one specific level. The swaplevel takes two level numbers or names and returns a new object with the levels interchanged (but the data is otherwise unaltered):

```
In [24]: frame.swaplevel('key1', 'key2')
Out[24]:
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, on the other hand, sorts the data using only the values in a single level. When swapping levels, it's not uncommon to also use sort_index so that the result is lexicographically sorted by the indicated level:

```
In [25]: frame.sort_index(level=1)
Out[25]:
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 [26]: frame.swaplevel(0, 1).sort_index(level=0)
Out[26]:
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 [27]: frame.sum(level='key2')
Out[27]:
state  Ohio     Colorado
color Green Red    Green
key2
1         6   8       10
2        12  14       16

In [28]: frame.sum(level='color', axis=1)
Out[28]:
color      Green  Red
key1 key2
a    1         2    1
     2         8    4
b    1        14    7
     2        20   10
```

Under the hood, this utilizes pandas's groupby machinery, which will be discussed in more detail later in the book.

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 [29]: 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 [30]: frame
Out[30]:
   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 [31]: frame2 = frame.set_index(['c', 'd'])

In [32]: frame2
Out[32]:
       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:

```
In [33]: frame.set_index(['c', 'd'], drop=False)
Out[33]:
       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 [34]: frame2.reset_index()
Out[34]:
     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 together in a number of ways:

- `pandas.merge` connects 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` concatenates or "stacks" together objects along an axis.
- The `combine_first` instance method enables splicing 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 central to relational databases (e.g., SQL-based). The `merge` function in pandas is the main entry point for using these algorithms on your data.

Let's start with a simple example:

```
In [35]: df1 = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'a', 'b'],
   ....:                     'data1': range(7)})

In [36]: df2 = pd.DataFrame({'key': ['a', 'b', 'd'],
   ....:                     'data2': range(3)})

In [37]: df1
Out[37]:
   data1 key
0      0   b
1      1   b
2      2   a
3      3   c
4      4   a
5      5   a
```

```
6      6   b

In [38]: df2
Out[38]:
   data2 key
0      0   a
1      1   b
2      2   d
```

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 `merge` with these objects we obtain:

```
In [39]: pd.merge(df1, df2)
Out[39]:
   data1 key  data2
0      0   b      1
1      1   b      1
2      6   b      1
3      2   a      0
4      4   a      0
5      5   a      0
```

Note that I didn't specify which column to join on. If that information is not specified, `merge` uses the overlapping column names as the keys. It's a good practice to specify explicitly, though:

```
In [40]: pd.merge(df1, df2, on='key')
Out[40]:
   data1 key  data2
0      0   b      1
1      1   b      1
2      6   b      1
3      2   a      0
4      4   a      0
5      5   a      0
```

If the column names are different in each object, you can specify them separately:

```
In [41]: df3 = pd.DataFrame({'lkey': ['b', 'b', 'a', 'c', 'a', 'a', 'b'],
   ....:                     'data1': range(7)})

In [42]: df4 = pd.DataFrame({'rkey': ['a', 'b', 'd'],
   ....:                     'data2': range(3)})

In [43]: pd.merge(df3, df4, left_on='lkey', right_on='rkey')
Out[43]:
   data1 lkey  data2 rkey
0      0    b      1    b
1      1    b      1    b
2      6    b      1    b
3      2    a      0    a
```

```
4       4    a       0    a
5       5    a       0    a
```

You may notice that the `'c'` and `'d'` values and associated data are missing from the result. By default `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 [44]: pd.merge(df1, df2, how='outer')
Out[44]:
   data1 key  data2
0    0.0   b    1.0
1    1.0   b    1.0
2    6.0   b    1.0
3    2.0   a    0.0
4    4.0   a    0.0
5    5.0   a    0.0
6    3.0   c    NaN
7    NaN   d    2.0
```

See Table 8-1 for a summary of the options for `how`.

Table 8-1. Different join types with how argument

Option	Behavior
`'inner'`	Use only the key combinations observed in both tables
`'left'`	Use all key combinations found in the left table
`'right'`	Use all key combinations found in the right table
`'output'`	Use all key combinations observed in both tables together

Many-to-many merges have well-defined, though not necessarily intuitive, behavior. Here's an example:

```
In [45]: df1 = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'b'],
   ....:                     'data1': range(6)})

In [46]: df2 = pd.DataFrame({'key': ['a', 'b', 'a', 'b', 'd'],
   ....:                     'data2': range(5)})

In [47]: df1
Out[47]:
   data1 key
0      0   b
1      1   b
2      2   a
3      3   c
4      4   a
5      5   b
```

```
In [48]: df2
Out[48]:
   data2 key
0      0   a
1      1   b
2      2   a
3      3   b
4      4   d

In [49]: pd.merge(df1, df2, on='key', how='left')
Out[49]:
    data1 key  data2
0       0   b    1.0
1       0   b    3.0
2       1   b    1.0
3       1   b    3.0
4       2   a    0.0
5       2   a    2.0
6       3   c    NaN
7       4   a    0.0
8       4   a    2.0
9       5   b    1.0
10      5   b    3.0
```

Many-to-many joins form the Cartesian product of the rows. 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 only affects the distinct key values appearing in the result:

```
In [50]: pd.merge(df1, df2, how='inner')
Out[50]:
   data1 key  data2
0      0   b      1
1      0   b      3
2      1   b      1
3      1   b      3
4      5   b      1
5      5   b      3
6      2   a      0
7      2   a      2
8      4   a      0
9      4   a      2
```

To merge with multiple keys, pass a list of column names:

```
In [51]: left = pd.DataFrame({'key1': ['foo', 'foo', 'bar'],
   ....:                      'key2': ['one', 'two', 'one'],
   ....:                      'lval': [1, 2, 3]})

In [52]: right = pd.DataFrame({'key1': ['foo', 'foo', 'bar', 'bar'],
   ....:                       'key2': ['one', 'one', 'one', 'two'],
   ....:                       'rval': [4, 5, 6, 7]})

In [53]: pd.merge(left, right, on=['key1', 'key2'], how='outer')
```

```
Out[53]:
  key1 key2  lval  rval
0  foo  one   1.0   4.0
1  foo  one   1.0   5.0
2  foo  two   2.0   NaN
3  bar  one   3.0   6.0
4  bar  two   NaN   7.0
```

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 (even though it's not actually implemented that way).

When you're joining columns-on-columns, the indexes on the passed DataFrame objects are discarded.

A last issue to consider in merge operations is the treatment of overlapping column names. While you can address the overlap manually (see the earlier section on renaming axis labels), `merge` has a `suffixes` option for specifying strings to append to overlapping names in the left and right DataFrame objects:

```
In [54]: pd.merge(left, right, on='key1')
Out[54]:
  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

In [55]: pd.merge(left, right, on='key1', suffixes=('_left', '_right'))
Out[55]:
  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 `merge`. Joining using the DataFrame's row index is the subject of the next section.

Table 8-2. merge function arguments

Argument	Description
`left`	DataFrame to be merged on the left side.
`right`	DataFrame to be merged on the right side.
`how`	One of `'inner'`, `'outer'`, `'left'`, or `'right'`; defaults to `'inner'`.
`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.
`right_on`	Analogous to `left_on` for `left` 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; `True` by default (disable to get better performance in some cases on large datasets).
`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.
`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. 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 [56]: left1 = pd.DataFrame({'key': ['a', 'b', 'a', 'a', 'b', 'c'],
   ....:                       'value': range(6)})

In [57]: right1 = pd.DataFrame({'group_val': [3.5, 7]}, index=['a', 'b'])

In [58]: left1
Out[58]:
  key  value
0   a      0
1   b      1
2   a      2
3   a      3
4   b      4
5   c      5

In [59]: right1
Out[59]:
   group_val
a        3.5
b        7.0
```

```
In [60]: pd.merge(left1, right1, left_on='key', right_index=True)
Out[60]:
  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
```

Since the default merge method is to intersect the join keys, you can instead form the union of them with an outer join:

```
In [61]: pd.merge(left1, right1, left_on='key', right_index=True, how='outer')
Out[61]:
  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 implicitly a multiple-key merge:

```
In [62]: lefth = pd.DataFrame({'key1': ['Ohio', 'Ohio', 'Ohio',
   ....:                                'Nevada', 'Nevada'],
   ....:                       'key2': [2000, 2001, 2002, 2001, 2002],
   ....:                       'data': np.arange(5.)})

In [63]: righth = pd.DataFrame(np.arange(12).reshape((6, 2)),
   ....:                       index=[['Nevada', 'Nevada', 'Ohio', 'Ohio',
   ....:                               'Ohio', 'Ohio'],
   ....:                              [2001, 2000, 2000, 2000, 2001, 2002]],
   ....:                       columns=['event1', 'event2'])

In [64]: lefth
Out[64]:
   data    key1  key2
0   0.0    Ohio  2000
1   1.0    Ohio  2001
2   2.0    Ohio  2002
3   3.0  Nevada  2001
4   4.0  Nevada  2002

In [65]: righth
Out[65]:
             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 [66]: pd.merge(lefth, righth, left_on=['key1', 'key2'], right_index=True)
Out[66]:
   data    key1  key2  event1  event2
0   0.0    Ohio  2000       4       5
0   0.0    Ohio  2000       6       7
1   1.0    Ohio  2001       8       9
2   2.0    Ohio  2002      10      11
3   3.0  Nevada  2001       0       1

In [67]: pd.merge(lefth, righth, left_on=['key1', 'key2'],
   ....:          right_index=True, how='outer')
Out[67]:
   data    key1  key2  event1  event2
0   0.0    Ohio  2000     4.0     5.0
0   0.0    Ohio  2000     6.0     7.0
1   1.0    Ohio  2001     8.0     9.0
2   2.0    Ohio  2002    10.0    11.0
3   3.0  Nevada  2001     0.0     1.0
4   4.0  Nevada  2002     NaN     NaN
4   NaN  Nevada  2000     2.0     3.0
```

Using the indexes of both sides of the merge is also possible:

```
In [68]: left2 = pd.DataFrame([[1., 2.], [3., 4.], [5., 6.]],
   ....:                      index=['a', 'c', 'e'],
   ....:                      columns=['Ohio', 'Nevada'])

In [69]: right2 = pd.DataFrame([[7., 8.], [9., 10.], [11., 12.], [13, 14]],
   ....:                       index=['b', 'c', 'd', 'e'],
   ....:                       columns=['Missouri', 'Alabama'])

In [70]: left2
Out[70]:
   Ohio  Nevada
a   1.0     2.0
c   3.0     4.0
e   5.0     6.0

In [71]: right2
Out[71]:
   Missouri  Alabama
b       7.0      8.0
c       9.0     10.0
d      11.0     12.0
e      13.0     14.0

In [72]: pd.merge(left2, right2, how='outer', left_index=True, right_index=True)
```

```
Out[72]:
   Ohio  Nevada  Missouri  Alabama
a   1.0     2.0       NaN      NaN
b   NaN     NaN       7.0      8.0
c   3.0     4.0       9.0     10.0
d   NaN     NaN      11.0     12.0
e   5.0     6.0      13.0     14.0
```

DataFrame has a convenient `join` instance for merging by index. It can also be used to combine together many DataFrame objects having the same or similar indexes but non-overlapping columns. In the prior example, we could have written:

```
In [73]: left2.join(right2, how='outer')
Out[73]:
   Ohio  Nevada  Missouri  Alabama
a   1.0     2.0       NaN      NaN
b   NaN     NaN       7.0      8.0
c   3.0     4.0       9.0     10.0
d   NaN     NaN      11.0     12.0
e   5.0     6.0      13.0     14.0
```

In part for legacy reasons (i.e., much earlier versions of pandas), DataFrame's `join` method performs a left join on the join keys, exactly preserving the left frame's row index. It also supports joining the index of the passed DataFrame on one of the columns of the calling DataFrame:

```
In [74]: left1.join(right1, on='key')
Out[74]:
  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
```

Lastly, for simple index-on-index merges, you can pass a list of DataFrames to `join` as an alternative to using the more general `concat` function described in the next section:

```
In [75]: another = pd.DataFrame([[7., 8.], [9., 10.], [11., 12.], [16., 17.]],
   ....:                        index=['a', 'c', 'e', 'f'],
   ....:                        columns=['New York', 'Oregon'])

In [76]: another
Out[76]:
   New York  Oregon
a       7.0     8.0
c       9.0    10.0
e      11.0    12.0
f      16.0    17.0
```

```
In [77]: left2.join([right2, another])
Out[77]:
   Ohio  Nevada  Missouri  Alabama  New York  Oregon
a   1.0     2.0       NaN      NaN       7.0     8.0
c   3.0     4.0       9.0     10.0       9.0    10.0
e   5.0     6.0      13.0     14.0      11.0    12.0

In [78]: left2.join([right2, another], how='outer')
Out[78]:
   Ohio  Nevada  Missouri  Alabama  New York  Oregon
a   1.0     2.0       NaN      NaN       7.0     8.0
b   NaN     NaN       7.0      8.0       NaN     NaN
c   3.0     4.0       9.0     10.0       9.0    10.0
d   NaN     NaN      11.0     12.0       NaN     NaN
e   5.0     6.0      13.0     14.0      11.0    12.0
f   NaN     NaN       NaN      NaN      16.0    17.0
```

Concatenating Along an Axis

Another kind of data combination operation is referred to interchangeably as concatenation, binding, or stacking. NumPy's `concatenate` function can do this with NumPy arrays:

```
In [79]: arr = np.arange(12).reshape((3, 4))

In [80]: arr
Out[80]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])

In [81]: np.concatenate([arr, arr], axis=1)
Out[81]:
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 things to think about:

- If the objects are indexed differently on the other axes, should we combine the distinct elements in these axes or use only the shared values (the intersection)?
- Do the concatenated chunks of data need to be identifiable 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 concerns. I'll give a number of examples to illustrate how it works. Suppose we have three Series with no index overlap:

```
In [82]: s1 = pd.Series([0, 1], index=['a', 'b'])

In [83]: s2 = pd.Series([2, 3, 4], index=['c', 'd', 'e'])

In [84]: s3 = pd.Series([5, 6], index=['f', 'g'])
```

Calling concat with these objects in a list glues together the values and indexes:

```
In [85]: pd.concat([s1, s2, s3])
Out[85]:
a    0
b    1
c    2
d    3
e    4
f    5
g    6
dtype: int64
```

By default concat works along axis=0, producing another Series. If you pass axis=1, the result will instead be a DataFrame (axis=1 is the columns):

```
In [86]: pd.concat([s1, s2, s3], axis=1)
Out[86]:
     0    1    2
a  0.0  NaN  NaN
b  1.0  NaN  NaN
c  NaN  2.0  NaN
d  NaN  3.0  NaN
e  NaN  4.0  NaN
f  NaN  NaN  5.0
g  NaN  NaN  6.0
```

In this case there is no overlap on the other axis, which as you can see is the sorted union (the 'outer' join) of the indexes. You can instead intersect them by passing join='inner':

```
In [87]: s4 = pd.concat([s1, s3])

In [88]: s4
Out[88]:
a    0
b    1
f    5
g    6
dtype: int64

In [89]: pd.concat([s1, s4], axis=1)
Out[89]:
     0  1
a  0.0  0
b  1.0  1
f  NaN  5
g  NaN  6

In [90]: pd.concat([s1, s4], axis=1, join='inner')
Out[90]:
   0  1
a  0  0
b  1  1
```

In this last example, the 'f' and 'g' labels disappeared because of the join='inner' option.

You can even specify the axes to be used on the other axes with join_axes:

```
In [91]: pd.concat([s1, s4], axis=1, join_axes=[['a', 'c', 'b', 'e']])
Out[91]:
     0    1
a  0.0  0.0
c  NaN  NaN
b  1.0  1.0
e  NaN  NaN
```

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 [92]: result = pd.concat([s1, s1, s3], keys=['one', 'two', 'three'])

In [93]: result
Out[93]:
one    a    0
       b    1
two    a    0
       b    1
three  f    5
       g    6
dtype: int64

In [94]: result.unstack()
Out[94]:
         a    b    f    g
one    0.0  1.0  NaN  NaN
two    0.0  1.0  NaN  NaN
three  NaN  NaN  5.0  6.0
```

In the case of combining Series along axis=1, the keys become the DataFrame column headers:

```
In [95]: pd.concat([s1, s2, s3], axis=1, keys=['one', 'two', 'three'])
Out[95]:
   one  two  three
a  0.0  NaN    NaN
b  1.0  NaN    NaN
c  NaN  2.0    NaN
d  NaN  3.0    NaN
e  NaN  4.0    NaN
f  NaN  NaN    5.0
g  NaN  NaN    6.0
```

The same logic extends to DataFrame objects:

```
In [96]: df1 = pd.DataFrame(np.arange(6).reshape(3, 2), index=['a', 'b', 'c'],
   ....:                    columns=['one', 'two'])

In [97]: df2 = pd.DataFrame(5 + np.arange(4).reshape(2, 2), index=['a', 'c'],
   ....:                    columns=['three', 'four'])

In [98]: df1
Out[98]:
   one  two
a    0    1
b    2    3
c    4    5

In [99]: df2
Out[99]:
   three  four
a      5     6
c      7     8

In [100]: pd.concat([df1, df2], axis=1, keys=['level1', 'level2'])
Out[100]:
  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
```

If you pass a dict of objects instead of a list, the dict's keys will be used for the keys option:

```
In [101]: pd.concat({'level1': df1, 'level2': df2}, axis=1)
Out[101]:
  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 [102]: pd.concat([df1, df2], axis=1, keys=['level1', 'level2'],
   .....:           names=['upper', 'lower'])
Out[102]:
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 [103]: df1 = pd.DataFrame(np.random.randn(3, 4), columns=['a', 'b', 'c', 'd'])

In [104]: df2 = pd.DataFrame(np.random.randn(2, 3), columns=['b', 'd', 'a'])

In [105]: df1
Out[105]:
          a         b         c         d
0  1.246435  1.007189 -1.296221  0.274992
1  0.228913  1.352917  0.886429 -2.001637
2 -0.371843  1.669025 -0.438570 -0.539741

In [106]: df2
Out[106]:
          b         d         a
0  0.476985  3.248944 -1.021228
1 -0.577087  0.124121  0.302614
```

In this case, you can pass `ignore_index=True`:

```
In [107]: pd.concat([df1, df2], ignore_index=True)
Out[107]:
          a         b         c         d
0  1.246435  1.007189 -1.296221  0.274992
1  0.228913  1.352917  0.886429 -2.001637
2 -0.371843  1.669025 -0.438570 -0.539741
3 -1.021228  0.476985       NaN  3.248944
4  0.302614 -0.577087       NaN  0.124121
```

Table 8-3. concat function arguments

Argument	Description
`objs`	List or dict of pandas objects to be concatenated; this is the only required argument
`axis`	Axis to concatenate along; defaults to 0 (along rows)
`join`	Either `'inner'` or `'outer'` (`'outer'` by default); whether to intersection (inner) or union (outer) together indexes along the other axes
`join_axes`	Specific indexes to use for the other *n*–1 axes instead of performing union/intersection logic
`keys`	Values to associate with objects being concatenated, forming a hierarchical index along the concatenation axis; can either be a list or array of arbitrary values, an array of tuples, or a list of arrays (if multiple-level arrays passed in `levels`)

Argument	Description
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 exception if so; by default (False) allows duplicates
ignore_index	Do not preserve indexes along concatenation axis, instead producing 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 whose indexes overlap in full or part. As a motivating example, consider NumPy's `where` function, which performs the array-oriented equivalent of an if-else expression:

```
In [108]: a = pd.Series([np.nan, 2.5, np.nan, 3.5, 4.5, np.nan],
   .....:               index=['f', 'e', 'd', 'c', 'b', 'a'])

In [109]: b = pd.Series(np.arange(len(a), dtype=np.float64),
   .....:               index=['f', 'e', 'd', 'c', 'b', 'a'])

In [110]: b[-1] = np.nan

In [111]: a
Out[111]:
f    NaN
e    2.5
d    NaN
c    3.5
b    4.5
a    NaN
dtype: float64

In [112]: b
Out[112]:
f    0.0
e    1.0
d    2.0
c    3.0
b    4.0
a    NaN
dtype: float64

In [113]: np.where(pd.isnull(a), b, a)
Out[113]: array([ 0. ,  2.5,  2. ,  3.5,  4.5,  nan])
```

Series has a `combine_first` method, which performs the equivalent of this operation along with pandas's usual data alignment logic:

```
In [114]: b[:-2].combine_first(a[2:])
Out[114]:
a    NaN
b    4.5
c    3.0
d    2.0
e    1.0
f    0.0
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 [115]: df1 = pd.DataFrame({'a': [1., np.nan, 5., np.nan],
   .....:                     'b': [np.nan, 2., np.nan, 6.],
   .....:                     'c': range(2, 18, 4)})

In [116]: df2 = pd.DataFrame({'a': [5., 4., np.nan, 3., 7.],
   .....:                     'b': [np.nan, 3., 4., 6., 8.]})

In [117]: df1
Out[117]:
     a    b   c
0  1.0  NaN   2
1  NaN  2.0   6
2  5.0  NaN  10
3  NaN  6.0  14

In [118]: df2
Out[118]:
     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 [119]: df1.combine_first(df2)
Out[119]:
     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
```

8.3 Reshaping and Pivoting

There are a number of basic operations for rearranging tabular data. These are alternatingly 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 [120]: 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 [121]: data
Out[121]:
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 [122]: result = data.stack()

In [123]: result
Out[123]:
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 [124]: result.unstack()
Out[124]:
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 [125]: result.unstack(0)
Out[125]:
state   Ohio  Colorado
number
one        0         3
two        1         4
three      2         5

In [126]: result.unstack('state')
Out[126]:
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 of the subgroups:

```
In [127]: s1 = pd.Series([0, 1, 2, 3], index=['a', 'b', 'c', 'd'])

In [128]: s2 = pd.Series([4, 5, 6], index=['c', 'd', 'e'])

In [129]: data2 = pd.concat([s1, s2], keys=['one', 'two'])

In [130]: data2
Out[130]:
one  a    0
     b    1
     c    2
     d    3
two  c    4
     d    5
     e    6
dtype: int64

In [131]: data2.unstack()
Out[131]:
       a    b    c    d    e
one  0.0  1.0  2.0  3.0  NaN
two  NaN  NaN  4.0  5.0  6.0
```

Stacking filters out missing data by default, so the operation is more easily invertible:

```
In [132]: data2.unstack()
Out[132]:
       a    b    c    d    e
one  0.0  1.0  2.0  3.0  NaN
two  NaN  NaN  4.0  5.0  6.0
```

```
In [133]: data2.unstack().stack()
Out[133]:
one  a    0.0
     b    1.0
     c    2.0
     d    3.0
two  c    4.0
     d    5.0
     e    6.0
dtype: float64

In [134]: data2.unstack().stack(dropna=False)
Out[134]:
one  a    0.0
     b    1.0
     c    2.0
     d    3.0
     e    NaN
two  a    NaN
     b    NaN
     c    4.0
     d    5.0
     e    6.0
dtype: float64
```

When you unstack in a DataFrame, the level unstacked becomes the lowest level in the result:

```
In [135]: df = pd.DataFrame({'left': result, 'right': result + 5},
   .....:                   columns=pd.Index(['left', 'right'], name='side'))

In [136]: df
Out[136]:
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 [137]: df.unstack('state')
Out[137]:
side   left          right
state  Ohio Colorado  Ohio Colorado
number
one       0        3     5        8
two       1        4     6        9
three     2        5     7       10
```

When calling `stack`, we can indicate the name of the axis to stack:

```
In [138]: df.unstack('state').stack('side')
Out[138]:
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 is in so-called *long* or *stacked* format. Let's load some example data and do a small amount of time series wrangling and other data cleaning:

```
In [139]: data = pd.read_csv('examples/macrodata.csv')

In [140]: data.head()
Out[140]:
     year  quarter   realgdp  realcons  realinv  realgovt  realdpi    cpi  \
0  1959.0      1.0  2710.349    1707.4  286.898   470.045   1886.9  28.98
1  1959.0      2.0  2778.801    1733.7  310.859   481.301   1919.7  29.15
2  1959.0      3.0  2775.488    1751.8  289.226   491.260   1916.4  29.35
3  1959.0      4.0  2785.204    1753.7  299.356   484.052   1931.3  29.37
4  1960.0      1.0  2847.699    1770.5  331.722   462.199   1955.5  29.54
      m1  tbilrate  unemp      pop  infl  realint
0  139.7      2.82    5.8  177.146  0.00     0.00
1  141.7      3.08    5.1  177.830  2.34     0.74
2  140.5      3.82    5.3  178.657  2.74     1.09
3  140.0      4.33    5.6  179.386  0.27     4.06
4  139.6      3.50    5.2  180.007  2.31     1.19

In [141]: periods = pd.PeriodIndex(year=data.year, quarter=data.quarter,
   .....:                          name='date')

In [142]: columns = pd.Index(['realgdp', 'infl', 'unemp'], name='item')

In [143]: data = data.reindex(columns=columns)

In [144]: data.index = periods.to_timestamp('D', 'end')

In [145]: ldata = data.stack().reset_index().rename(columns={0: 'value'})
```

We will look at `PeriodIndex` a bit more closely in Chapter 11. In short, it combines the `year` and `quarter` columns to create a kind of time interval type.

Now, `ldata` looks like:

```
In [146]: ldata[:10]
Out[146]:
```

```
         date     item     value
0 1959-03-31  realgdp  2710.349
1 1959-03-31     infl     0.000
2 1959-03-31    unemp     5.800
3 1959-06-30  realgdp  2778.801
4 1959-06-30     infl     2.340
5 1959-06-30    unemp     5.100
6 1959-09-30  realgdp  2775.488
7 1959-09-30     infl     2.740
8 1959-09-30    unemp     5.300
9 1959-12-31  realgdp  2785.204
```

This is the so-called *long* format for multiple time series, or other observational data with two or more keys (here, our keys are date and item). Each row in the table represents a single observation.

Data is frequently stored this way in relational databases like MySQL, 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 [147]: pivoted = ldata.pivot('date', 'item', 'value')

In [148]: pivoted
Out[148]:
item        infl    realgdp  unemp
date
1959-03-31  0.00   2710.349    5.8
1959-06-30  2.34   2778.801    5.1
1959-09-30  2.74   2775.488    5.3
1959-12-31  0.27   2785.204    5.6
1960-03-31  2.31   2847.699    5.2
1960-06-30  0.14   2834.390    5.2
1960-09-30  2.70   2839.022    5.6
1960-12-31  1.21   2802.616    6.3
1961-03-31 -0.40   2819.264    6.8
1961-06-30  1.47   2872.005    7.0
...          ...        ...    ...
2007-06-30  2.75  13203.977    4.5
2007-09-30  3.45  13321.109    4.7
2007-12-31  6.38  13391.249    4.8
2008-03-31  2.82  13366.865    4.9
2008-06-30  8.53  13415.266    5.4
2008-09-30 -3.16  13324.600    6.0
2008-12-31 -8.79  13141.920    6.9
2009-03-31  0.94  12925.410    8.1
2009-06-30  3.37  12901.504    9.2
```

```
2009-09-30  3.56  12990.341    9.6
[203 rows x 3 columns]
```

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 [149]: ldata['value2'] = np.random.randn(len(ldata))

In [150]: ldata[:10]
Out[150]:
        date     item     value    value2
0 1959-03-31  realgdp  2710.349  0.523772
1 1959-03-31     infl     0.000  0.000940
2 1959-03-31    unemp     5.800  1.343810
3 1959-06-30  realgdp  2778.801 -0.713544
4 1959-06-30     infl     2.340 -0.831154
5 1959-06-30    unemp     5.100 -2.370232
6 1959-09-30  realgdp  2775.488 -1.860761
7 1959-09-30     infl     2.740 -0.860757
8 1959-09-30    unemp     5.300  0.560145
9 1959-12-31  realgdp  2785.204 -1.265934
```

By omitting the last argument, you obtain a DataFrame with hierarchical columns:

```
In [151]: pivoted = ldata.pivot('date', 'item')

In [152]: pivoted[:5]
Out[152]:
           value                    value2
item        infl   realgdp unemp      infl   realgdp     unemp
date
1959-03-31  0.00  2710.349   5.8  0.000940  0.523772  1.343810
1959-06-30  2.34  2778.801   5.1 -0.831154 -0.713544 -2.370232
1959-09-30  2.74  2775.488   5.3 -0.860757 -1.860761  0.560145
1959-12-31  0.27  2785.204   5.6  0.119827 -1.265934 -1.063512
1960-03-31  2.31  2847.699   5.2 -2.359419  0.332883 -0.199543

In [153]: pivoted['value'][:5]
Out[153]:
item        infl   realgdp  unemp
date
1959-03-31  0.00  2710.349    5.8
1959-06-30  2.34  2778.801    5.1
1959-09-30  2.74  2775.488    5.3
1959-12-31  0.27  2785.204    5.6
1960-03-31  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 [154]: unstacked = ldata.set_index(['date', 'item']).unstack('item')

In [155]: unstacked[:7]
Out[155]:
           value                    value2
item        infl   realgdp unemp      infl   realgdp     unemp
date
1959-03-31  0.00  2710.349   5.8  0.000940  0.523772  1.343810
1959-06-30  2.34  2778.801   5.1 -0.831154 -0.713544 -2.370232
1959-09-30  2.74  2775.488   5.3 -0.860757 -1.860761  0.560145
1959-12-31  0.27  2785.204   5.6  0.119827 -1.265934 -1.063512
1960-03-31  2.31  2847.699   5.2 -2.359419  0.332883 -0.199543
1960-06-30  0.14  2834.390   5.2 -0.970736 -1.541996 -1.307030
1960-09-30  2.70  2839.022   5.6  0.377984  0.286350 -0.753887
```

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 [157]: df = pd.DataFrame({'key': ['foo', 'bar', 'baz'],
   .....:                    'A': [1, 2, 3],
   .....:                    'B': [4, 5, 6],
   .....:                    'C': [7, 8, 9]})

In [158]: df
Out[158]:
   A  B  C  key
0  1  4  7  foo
1  2  5  8  bar
2  3  6  9  baz
```

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 [159]: melted = pd.melt(df, ['key'])

In [160]: melted
Out[160]:
   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 [161]: reshaped = melted.pivot('key', 'variable', 'value')

In [162]: reshaped
Out[162]:
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 [163]: reshaped.reset_index()
Out[163]:
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 [164]: pd.melt(df, id_vars=['key'], value_vars=['A', 'B'])
Out[164]:
   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 [165]: pd.melt(df, value_vars=['A', 'B', 'C'])
Out[165]:
  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 [166]: pd.melt(df, value_vars=['key', 'A', 'B'])
Out[166]:
  variable value
0      key   foo
1      key   bar
```

```
    key   baz
      A     1
      A     2
      A     3
      B     4
      B     5
      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 pandas later in the book when we discuss more advanced analytics.