

# 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 can 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>i</i> deleted
drop	Compute new Index by deleting passed values
insert	Compute new Index by inserting element at index <i>i</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 fill-in 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
<code>index</code>	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.
<code>method</code>	Interpolation (fill) method; 'ffill' fills forward, while 'bfill' fills backward.
<code>fill_value</code>	Substitute value to use when introducing missing data by reindexing.
<code>limit</code>	When forward- or backfilling, maximum size gap (in number of elements) to fill.
<code>tolerance</code>	When forward- or backfilling, maximum size gap (in absolute numeric distance) to fill for inexact matches.
<code>level</code>	Match simple Index on level of MultiIndex; otherwise select subset of.
<code>copy</code>	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 end-point 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
<code>df[val]</code>	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)
<code>df.loc[val]</code>	Selects single row or subset of rows from the DataFrame by label
<code>df.loc[:, val]</code>	Selects single column or subset of columns by label
<code>df.loc[val1, val2]</code>	Select both rows and columns by label
<code>df.iloc[where]</code>	Selects single row or subset of rows from the DataFrame by integer position

Type	Notes
<code>df.iloc[:, where]</code>	Selects single column or subset of columns by integer position
<code>df.iloc[where_i, where_j]</code>	Select both rows and columns by integer position
<code>df.at[label_i, label_j]</code>	Select a single scalar value by row and column label
<code>df.iat[i, j]</code>	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
<code>add</code> , <code>radd</code>	Methods for addition (+)
<code>sub</code> , <code>rsub</code>	Methods for subtraction (-)
<code>div</code> , <code>rdiv</code>	Methods for division (/)
<code>floordiv</code> , <code>rfloordiv</code>	Methods for floor division (//)
<code>mul</code> , <code>rmul</code>	Methods for multiplication (*)
<code>pow</code> , <code>rpow</code>	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	d	e
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]:
```

	b	d	e
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 `apply` `map`:

```
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 <code>method='min'</code> , 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

```
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
<code>isin</code>	Compute boolean array indicating whether each Series value is contained in the passed sequence of values
<code>match</code>	Compute integer indices for each value in an array into another array of distinct values; helpful for data alignment and join-type operations
<code>unique</code>	Compute array of unique values in a Series, returned in the order observed
<code>value_counts</code>	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.