

Getting Started with pandas

pandas will be the primary library of interest throughout much of the rest of the book. It contains high-level data structures and manipulation tools designed to make data analysis fast and easy in Python. pandas is built on top of NumPy and makes it easy to use in NumPy-centric applications.

As a bit of background, I started building pandas in early 2008 during my tenure at AQR, a quantitative investment management firm. At the time, I had a distinct set of requirements that were not well-addressed by any single tool at my disposal:

- Data structures with labeled axes supporting automatic or explicit data alignment. This prevents common errors resulting from misaligned data and working with differently-indexed data coming from different sources.
- Integrated time series functionality.
- The same data structures handle both time series data and non-time series data.
- Arithmetic operations and reductions (like summing across an axis) would pass on the metadata (axis labels).
- Flexible handling of missing data.
- Merge and other relational operations found in popular database databases (SQL-based, for example).

I wanted to be able to do all of these things in one place, preferably in a language well-suited to general purpose software development. Python was a good candidate language for this, but at that time there was not an integrated set of data structures and tools providing this functionality.

Over the last four years, pandas has matured into a quite large library capable of solving a much broader set of data handling problems than I ever anticipated, but it has expanded in its scope without compromising the simplicity and ease-of-use that I desired from the very beginning. I hope that after reading this book, you will find it to be just as much of an indispensable tool as I do.

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

```
In [1]: from pandas import Series, DataFrame
```

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

Thus, whenever you see `pd.` in code, it's referring to `pandas`. `Series` and `DataFrame` are used so much that I find it easier to import them into the local namespace.

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 an array of data (of any NumPy data type) and an associated array of data labels, called its *index*. The simplest `Series` is formed from only an array of data:

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

```
In [5]: obj
```

```
Out[5]:
```

```
0    4
1    7
2   -5
3    3
```

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 [6]: obj.values
```

```
Out[6]: array([ 4,  7, -5,  3])
```

```
In [7]: obj.index
```

```
Out[7]: Int64Index([0, 1, 2, 3])
```

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

```
In [8]: obj2 = Series([4, 7, -5, 3], index=['d', 'b', 'a', 'c'])
```

```
In [9]: obj2
```

```
Out[9]:
```

```
d    4
b    7
a   -5
c    3
```

```
In [10]: obj2.index
Out[10]: Index([d, b, a, c], dtype=object)
```

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

```
In [11]: obj2['a']
Out[11]: -5

In [12]: obj2['d'] = 6

In [13]: obj2[['c', 'a', 'd']]
Out[13]:
c    3
a   -5
d    6
```

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

```
In [14]: obj2
Out[14]:
d    6
b    7
a   -5
c    3

In [15]: obj2[obj2 > 0]
Out[15]:
d    6
b    7
c    3

In [16]: obj2 * 2
Out[16]:
d   12
b   14
a  -10
c    6

In [17]: np.exp(obj2)
Out[17]:
d    403.428793
b   1096.633158
a     0.006738
c    20.085537
```

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 substituted into many functions that expect a dict:

```
In [18]: 'b' in obj2
Out[18]: True

In [19]: 'e' in obj2
Out[19]: False
```

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

```
In [20]: sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}

In [21]: obj3 = Series(sdata)

In [22]: obj3
Out[22]:
Ohio    35000
Oregon   16000
```

```
Texas    71000
Utah      5000
```

When only passing a dict, the index in the resulting Series will have the dict's keys in sorted order.

```
In [23]: states = ['California', 'Ohio', 'Oregon', 'Texas']
```

```
In [24]: obj4 = Series(sdata, index=states)
```

```
In [25]: obj4
Out[25]:
California    NaN
Ohio          35000
Oregon        16000
Texas         71000
```

In this case, 3 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. I will use the terms “missing” or “NA” to refer to missing data. The `isnull` and `notnull` functions in pandas should be used to detect missing data:

```
In [26]: pd.isnull(obj4)      In [27]: pd.notnull(obj4)
Out[26]:
California    True
Ohio          False
Oregon        False
Texas         False
Out[27]:
California    False
Ohio          True
Oregon        True
Texas         True
```

Series also has these as instance methods:

```
In [28]: obj4.isnull()
Out[28]:
California    True
Ohio          False
Oregon        False
Texas         False
```

I discuss working with missing data in more detail later in this chapter.

A critical Series feature for many applications is that it automatically aligns differently-indexed data in arithmetic operations:

```
In [29]: obj3      In [30]: obj4
Out[29]:
Ohio    35000
Oregon  16000
Texas   71000
Utah     5000
Out[30]:
California    NaN
Ohio          35000
Oregon        16000
Texas         71000
```

```
In [31]: obj3 + obj4
Out[31]:
California    NaN
Ohio          70000
Oregon        32000
```

```
Texas      142000
Utah        NaN
```

Data alignment features are addressed as a separate topic.

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

```
In [32]: obj4.name = 'population'

In [33]: obj4.index.name = 'state'

In [34]: obj4
Out[34]:
state
California      NaN
Ohio            35000
Oregon          16000
Texas           71000
Name: population
```

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

```
In [35]: obj.index = ['Bob', 'Steve', 'Jeff', 'Ryan']

In [36]: obj
Out[36]:
Bob      4
Steve    7
Jeff    -5
Ryan     3
```

DataFrame

A DataFrame represents a tabular, spreadsheet-like data structure containing 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 (one for all sharing the same index). Compared with other such DataFrame-like structures you may have used before (like R's `data.frame`), row-oriented and column-oriented operations in DataFrame are treated roughly symmetrically. 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 far outside the scope of this book.



While DataFrame stores the data internally in a two-dimensional format, you can easily represent much higher-dimensional data in a tabular format using hierarchical indexing, a subject of a later section and a key ingredient in many of the more advanced data-handling features in pandas.

There are numerous 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'],
        'year': [2000, 2001, 2002, 2001, 2002],
        'pop': [1.5, 1.7, 3.6, 2.4, 2.9]}
frame = DataFrame(data)
```

The resulting DataFrame will have its index assigned automatically as with Series, and the columns are placed in sorted order:

```
In [38]: frame
Out[38]:
```

	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 exactly what you pass:

```
In [39]: DataFrame(data, columns=['year', 'state', 'pop'])
Out[39]:
```

	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

As with Series, if you pass a column that isn't contained in data, it will appear with NA values in the result:

```
In [40]: frame2 = DataFrame(data, columns=['year', 'state', 'pop', 'debt'],
.....:                        index=['one', 'two', 'three', 'four', 'five'])
```

```
In [41]: frame2
Out[41]:
```

	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

```
In [42]: frame2.columns
Out[42]: 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 [43]: frame2['state']
Out[43]:
```

	Ohio
one	

```
In [44]: frame2.year
Out[44]:
```

	2000
one	

two	Ohio	two	2001
three	Ohio	three	2002
four	Nevada	four	2001
five	Nevada	five	2002
Name: state		Name: year	

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 by a couple of methods, such as the `ix` indexing field (much more on this later):

```
In [45]: frame2.ix['three']
Out[45]:
year      2002
state     Ohio
pop        3.6
debt       NaN
Name: three
```

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

```
In [46]: frame2['debt'] = 16.5
```

```
In [47]: frame2
Out[47]:
   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
```

```
In [48]: frame2['debt'] = np.arange(5.)
```

```
In [49]: frame2
Out[49]:
   year  state  pop  debt
one  2000   Ohio  1.5    0
two   2001   Ohio  1.7    1
three 2002   Ohio  3.6    2
four   2001  Nevada  2.4    3
five   2002  Nevada  2.9    4
```

When assigning lists or arrays to a column, the value's length must match the length of the DataFrame. If you assign a Series, it will be instead conformed exactly to the DataFrame's index, inserting missing values in any holes:

```
In [50]: val = Series([-1.2, -1.5, -1.7], index=['two', 'four', 'five'])
```

```
In [51]: frame2['debt'] = val
```

```
In [52]: frame2
Out[52]:
   year  state  pop  debt
one  2000   Ohio  1.5    0
two   2001   Ohio  1.7    1
three 2002   Ohio  3.6    2
four   2001  Nevada  2.4    3
five   2002  Nevada  2.9    4
```

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

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

```
In [53]: frame2['eastern'] = frame2.state == 'Ohio'
```

```
In [54]: frame2
```

```
Out[54]:
```

	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

```
In [55]: del frame2['eastern']
```

```
In [56]: frame2.columns
```

```
Out[56]: Index([year, state, pop, debt], dtype=object)
```



The column returned when 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 using the Series's `copy` method.

Another common form of data is a nested dict of dicts format:

```
In [57]: pop = {'Nevada': {2001: 2.4, 2002: 2.9},
.....:         'Ohio': {2000: 1.5, 2001: 1.7, 2002: 3.6}}
```

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

```
In [58]: frame3 = DataFrame(pop)
```

```
In [59]: frame3
```

```
Out[59]:
```

	Nevada	Ohio
2000	NaN	1.5
2001	2.4	1.7
2002	2.9	3.6

Of course you can always transpose the result:

```
In [60]: frame3.T
```

```
Out[60]:
```

	2000	2001	2002
Nevada	NaN	2.4	2.9
Ohio	1.5	1.7	3.6

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

```
In [61]: DataFrame(pop, index=[2001, 2002, 2003])
Out[61]:
```

	Nevada	Ohio
2001	2.4	1.7
2002	2.9	3.6
2003	NaN	NaN

Dicts of Series are treated much in the same way:

```
In [62]: pdata = {'Ohio': frame3['Ohio'][::-1],
.....:           'Nevada': frame3['Nevada'][:2]}

In [63]: DataFrame(pdata)
Out[63]:
```

	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 [64]: frame3.index.name = 'year'; frame3.columns.name = 'state'

In [65]: frame3
Out[65]:
```

state	Nevada	Ohio
year		
2000	NaN	1.5
2001	2.4	1.7
2002	2.9	3.6

Like Series, the `values` attribute returns the data contained in the DataFrame as a 2D ndarray:

```
In [66]: frame3.values
Out[66]:
```

```
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 accomodate all of the columns:

```
In [67]: frame2.values
Out[67]:
```

```
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]], 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 used when constructing a Series or DataFrame is internally converted to an Index:

```
In [68]: obj = Series(range(3), index=['a', 'b', 'c'])
```

```
In [69]: index = obj.index
```

```
In [70]: index
```

```
Out[70]: Index([a, b, c], dtype=object)
```

```
In [71]: index[1:]
```

```
Out[71]: Index([b, c], dtype=object)
```

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

```
In [72]: index[1] = 'd'
```

```
-----
Exception                                Traceback (most recent call last)
<ipython-input-72-676fdeb26a68> in <module>()
----> 1 index[1] = 'd'
/Users/wesm/code/pandas/pandas/core/index.pyc in __setitem__(self, key, value)
    302     def __setitem__(self, key, value):
    303         """Disable the setting of values."""
--> 304         raise Exception(str(self.__class__) + ' object is immutable')
    305
    306     def __getitem__(self, key):
Exception: <class 'pandas.core.index.Index'> object is immutable
```

Immutability is important so that Index objects can be safely shared among data structures:

```
In [73]: index = pd.Index(np.arange(3))

In [74]: obj2 = Series([1.5, -2.5, 0], index=index)

In [75]: obj2.index is index
Out[75]: True
```

[Table 5-2](#) has a list of built-in Index classes in the library. With some development effort, Index can even be subclassed to implement specialized axis indexing functionality.



Many users will not need to know much about Index objects, but they're nonetheless an important part of pandas's data model.

Table 5-2. Main Index objects in pandas

Class	Description
Index	The most general Index object, representing axis labels in a NumPy array of Python objects.
Int64Index	Specialized Index for integer values.
MultiIndex	“Hierarchical” index object representing multiple levels of indexing on a single axis. Can be thought of as similar to an array of tuples.
DatetimeIndex	Stores nanosecond timestamps (represented using NumPy's <code>datetime64</code> dtype).
PeriodIndex	Specialized Index for Period data (timespans).

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

```
In [76]: frame3
Out[76]:
state  Nevada  Ohio
year
2000      NaN   1.5
2001      2.4   1.7
2002      2.9   3.6

In [77]: 'Ohio' in frame3.columns
Out[77]: True

In [78]: 2003 in frame3.index
Out[78]: False
```

Each Index has a number of methods and properties for set logic and answering other common questions about the data it contains. These are summarized in [Table 5-3](#).

Table 5-3. Index methods and properties

Method	Description
<code>append</code>	Concatenate with additional Index objects, producing a new Index
<code>diff</code>	Compute set difference as an Index
<code>intersection</code>	Compute set intersection
<code>union</code>	Compute set union
<code>isin</code>	Compute boolean array indicating whether each value is contained in the passed collection
<code>delete</code>	Compute new Index with element at index <code>i</code> deleted
<code>drop</code>	Compute new index by deleting passed values
<code>insert</code>	Compute new Index by inserting element at index <code>i</code>
<code>is_monotonic</code>	Returns <code>True</code> if each element is greater than or equal to the previous element
<code>is_unique</code>	Returns <code>True</code> if the Index has no duplicate values
<code>unique</code>	Compute the array of unique values in the Index

Essential Functionality

In this section, I'll walk you through the fundamental mechanics of interacting with the data contained in a `Series` or `DataFrame`. Upcoming chapters 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; I instead focus on the most important features, leaving the less common (that is, more esoteric) things for you to explore on your own.

Reindexing

A critical method on `pandas` objects is `reindex`, which means to create a new object with the data *conformed* to a new index. Consider a simple example from above:

```
In [79]: obj = Series([4.5, 7.2, -5.3, 3.6], index=['d', 'b', 'a', 'c'])

In [80]: obj
Out[80]:
d    4.5
b    7.2
a   -5.3
c    3.6
```

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 [81]: obj2 = obj.reindex(['a', 'b', 'c', 'd', 'e'])

In [82]: obj2
Out[82]:
a   -5.3
```

```

b    7.2
c    3.6
d    4.5
e    NaN

In [83]: obj.reindex(['a', 'b', 'c', 'd', 'e'], fill_value=0)
Out[83]:
a    -5.3
b     7.2
c     3.6
d     4.5
e     0.0

```

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 [84]: obj3 = Series(['blue', 'purple', 'yellow'], index=[0, 2, 4])

In [85]: obj3.reindex(range(6), method='ffill')
Out[85]:
0    blue
1    blue
2   purple
3   purple
4   yellow
5   yellow

```

Table 5-4 lists available `method` options. At this time, interpolation more sophisticated than forward- and backfilling would need to be applied after the fact.

Table 5-4. *reindex method (interpolation) options*

Argument	Description
<code>ffill</code> or <code>pad</code>	Fill (or carry) values forward
<code>bfill</code> or <code>backfill</code>	Fill (or carry) values backward

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

```

In [86]: frame = DataFrame(np.arange(9).reshape((3, 3)), index=['a', 'c', 'd'],
.....:                    columns=['Ohio', 'Texas', 'California'])

In [87]: frame
Out[87]:
   Ohio  Texas  California
a      0      1           2
c      3      4           5
d      6      7           8

In [88]: frame2 = frame.reindex(['a', 'b', 'c', 'd'])

In [89]: frame2
Out[89]:

```

	Ohio	Texas	California
a	0	1	2
b	NaN	NaN	NaN
c	3	4	5
d	6	7	8

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

```
In [90]: states = ['Texas', 'Utah', 'California']
```

```
In [91]: frame.reindex(columns=states)
```

```
Out[91]:
```

	Texas	Utah	California
a	1	NaN	2
c	4	NaN	5
d	7	NaN	8

Both can be reindexed in one shot, though interpolation will only apply row-wise (axis 0):

```
In [92]: frame.reindex(index=['a', 'b', 'c', 'd'], method='ffill',
.....:                  columns=states)
```

```
Out[92]:
```

	Texas	Utah	California
a	1	NaN	2
b	1	NaN	2
c	4	NaN	5
d	7	NaN	8

As you'll see soon, reindexing can be done more succinctly by label-indexing with `ix`:

```
In [93]: frame.ix[['a', 'b', 'c', 'd'], states]
```

```
Out[93]:
```

	Texas	Utah	California
a	1	NaN	2
b	NaN	NaN	NaN
c	4	NaN	5
d	7	NaN	8

Table 5-5. *reindex* function arguments

Argument	Description
<code>index</code>	New sequence to use as index. Can be <code>Index</code> instance or any other sequence-like Python data structure. An <code>Index</code> will be used exactly as is without any copying
<code>method</code>	Interpolation (fill) method, see Table 5-4 for options.
<code>fill_value</code>	Substitute value to use when introducing missing data by reindexing
<code>limit</code>	When forward- or backfilling, maximum size gap to fill
<code>level</code>	Match simple <code>Index</code> on level of <code>Multindex</code> , otherwise select subset of
<code>copy</code>	Do not copy underlying data if new index is equivalent to old index. <code>True</code> by default (i.e. always copy data).

Dropping entries from an axis

Dropping one or more entries from an axis is easy if you 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 [94]: obj = Series(np.arange(5.), index=['a', 'b', 'c', 'd', 'e'])
```

```
In [95]: new_obj = obj.drop('c')
```

```
In [96]: new_obj
```

```
Out[96]:
```

```
a    0
b    1
d    3
e    4
```

```
In [97]: obj.drop(['d', 'c'])
```

```
Out[97]:
```

```
a    0
b    1
e    4
```

With `DataFrame`, index values can be deleted from either axis:

```
In [98]: data = DataFrame(np.arange(16).reshape((4, 4)),
.....:                    index=['Ohio', 'Colorado', 'Utah', 'New York'],
.....:                    columns=['one', 'two', 'three', 'four'])
```

```
In [99]: data.drop(['Colorado', 'Ohio'])
```

```
Out[99]:
```

	one	two	three	four
Utah	8	9	10	11
New York	12	13	14	15

```
In [100]: data.drop('two', axis=1)
```

```
Out[100]:
```

	one	three	four
Ohio	0	2	3
Colorado	4	6	7
Utah	8	10	11
New York	12	14	15

```
In [101]: data.drop(['two', 'four'], axis=1)
```

```
Out[101]:
```

	one	three
Ohio	0	2
Colorado	4	6
Utah	8	10
New York	12	14

Indexing, selection, and filtering

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

```
In [102]: obj = Series(np.arange(4.), index=['a', 'b', 'c', 'd'])
```

```
In [103]: obj['b']
```

```
Out[103]: 1.0
```

```
In [104]: obj[1]
```

```
Out[104]: 1.0
```

```
In [105]: obj[2:4]
```

```
Out[105]:
```

```
In [106]: obj[['b', 'a', 'd']]
```

```
Out[106]:
```

```

c    2
d    3

b    1
a    0
d    3

In [107]: obj[[1, 3]]
Out[107]:
b    1
d    3

In [108]: obj[obj < 2]
Out[108]:
a    0
b    1

```

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

```

In [109]: obj['b':'c']
Out[109]:
b    1
c    2

```

Setting using these methods works just as you would expect:

```

In [110]: obj['b':'c'] = 5

In [111]: obj
Out[111]:
a    0
b    5
c    5
d    3

```

As you've seen above, indexing into a DataFrame is for retrieving one or more columns either with a single value or sequence:

```

In [112]: data = DataFrame(np.arange(16).reshape((4, 4)),
.....:                    index=['Ohio', 'Colorado', 'Utah', 'New York'],
.....:                    columns=['one', 'two', 'three', 'four'])

In [113]: data
Out[113]:
      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 [114]: data['two']
Out[114]:
Ohio      1
Colorado   5
Utah      9
New York  13
Name: two

In [115]: data[['three', 'one']]
Out[115]:
      three  one
Ohio       2    0
Colorado   6    4
Utah      10    8
New York  14   12

```

Indexing like this has a few special cases. First selecting rows by slicing or a boolean array:

```

In [116]: data[:2]
Out[116]:
      one  two  three  four
Ohio     0   1     2     3
Colorado  4   5     6     7

In [117]: data[data['three'] > 5]
Out[117]:
      one  two  three  four
Utah     8   9    10    11
New York 12  13    14    15

```


Ohio	0	1	2	3	Colorado	4	5	6	7
Colorado	4	5	6	7	Utah	8	9	10	11
					New York	12	13	14	15

This might seem inconsistent to some readers, but this syntax arose out of practicality and nothing more. Another use case is in indexing with a boolean DataFrame, such as one produced by a scalar comparison:

```
In [118]: data < 5
Out[118]:
```

	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 [119]: data[data < 5] = 0
```

```
In [120]: data
Out[120]:
```

	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 is intended to make DataFrame syntactically more like an ndarray in this case.

For DataFrame label-indexing on the rows, I introduce the special indexing field `ix`. It enables you to select a subset of the rows and columns from a DataFrame with NumPy-like notation plus axis labels. As I mentioned earlier, this is also a less verbose way to do reindexing:

```
In [121]: data.ix['Colorado', ['two', 'three']]
Out[121]:
```

two	5
three	6

Name: Colorado

```
In [122]: data.ix[['Colorado', 'Utah'], [3, 0, 1]]
Out[122]:
```

	four	one	two
Colorado	7	0	5
Utah	11	8	9

In [123]: data.ix[2]	In [124]: data.ix[:'Utah', 'two']		
Out[123]:	Out[124]:		
one	8	Ohio	0
two	9	Colorado	5
three	10	Utah	9
four	11		
Name: Utah		Name: two	

```
In [125]: data.ix[data.three > 5, :3]
Out[125]:
```

	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, there is a short summary of many of them in [Table 5-6](#). You have a number of additional options when working with hierarchical indexes as you'll later see.



When 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. Thus I made the design trade-off to push all of the rich label-indexing into `ix`.

Table 5-6. Indexing options with DataFrame

Type	Notes
<code>obj[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>obj.ix[val]</code>	Selects single row of subset of rows from the DataFrame.
<code>obj.ix[:, val]</code>	Selects single column of subset of columns.
<code>obj.ix[val1, val2]</code>	Select both rows and columns.
<code>reindex</code> method	Conform one or more axes to new indexes.
<code>xs</code> method	Select single row or column as a Series by label.
<code>icol</code> , <code>irow</code> methods	Select single column or row, respectively, as a Series by integer location.
<code>get_value</code> , <code>set_value</code> methods	Select single value by row and column label.

Arithmetic and data alignment

One of the most important pandas features is the behavior of arithmetic between objects with different indexes. When 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. Let's look at a simple example:

```
In [126]: s1 = Series([7.3, -2.5, 3.4, 1.5], index=['a', 'c', 'd', 'e'])
```

```
In [127]: s2 = Series([-2.1, 3.6, -1.5, 4, 3.1], index=['a', 'c', 'e', 'f', 'g'])
```

```
In [128]: s1      In [129]: s2
Out[128]: Out[129]:
a    7.3      a   -2.1
c   -2.5      c    3.6
d    3.4      e   -1.5
```

```
e    1.5          f    4.0
          g    3.1
```

Adding these together yields:

```
In [130]: s1 + s2
Out[130]:
a    5.2
c    1.1
d    NaN
e    0.0
f    NaN
g    NaN
```

The internal data alignment introduces NA values in the indices that don't overlap. Missing values propagate in arithmetic computations.

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

```
In [131]: df1 = DataFrame(np.arange(9.).reshape((3, 3)), columns=list('bcd'),
.....:                    index=['Ohio', 'Texas', 'Colorado'])

In [132]: df2 = DataFrame(np.arange(12.).reshape((4, 3)), columns=list('bde'),
.....:                    index=['Utah', 'Ohio', 'Texas', 'Oregon'])

In [133]: df1
Out[133]:
   b  c  d
Ohio  0  1  2
Texas  3  4  5
Colorado 6  7  8

In [134]: df2
Out[134]:
   b  d  e
Utah  0  1  2
Ohio  3  4  5
Texas  6  7  8
Oregon 9 10 11
```

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

```
In [135]: df1 + df2
Out[135]:
   b  c  d  e
Colorado NaN NaN NaN NaN
Ohio      3 NaN  6 NaN
Oregon   NaN NaN NaN NaN
Texas     9 NaN 12 NaN
Utah     NaN NaN 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 [136]: df1 = DataFrame(np.arange(12.).reshape((3, 4)), columns=list('abcd'))

In [137]: df2 = DataFrame(np.arange(20.).reshape((4, 5)), columns=list('abcde'))

In [138]: df1
Out[138]:
   a  b  c  d
a  b  c  d

In [139]: df2
Out[139]:
   a  b  c  d  e
```

0	0	1	2	3		0	0	1	2	3	4
1	4	5	6	7		1	5	6	7	8	9
2	8	9	10	11		2	10	11	12	13	14
						3	15	16	17	18	19

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

```
In [140]: df1 + df2
Out[140]:
```

	a	b	c	d	e
0	0	2	4	6	NaN
1	9	11	13	15	NaN
2	18	20	22	24	NaN
3	NaN	NaN	NaN	NaN	NaN

Using the add method on df1, I pass df2 and an argument to fill_value:

```
In [141]: df1.add(df2, fill_value=0)
Out[141]:
```

	a	b	c	d	e
0	0	2	4	6	4
1	9	11	13	15	9
2	18	20	22	24	14
3	15	16	17	18	19

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

```
In [142]: df1.reindex(columns=df2.columns, fill_value=0)
Out[142]:
```

	a	b	c	d	e
0	0	1	2	3	0
1	4	5	6	7	0
2	8	9	10	11	0

Table 5-7. Flexible arithmetic methods

Method	Description
add	Method for addition (+)
sub	Method for subtraction (-)
div	Method for division (/)
mul	Method for multiplication (*)

Operations between DataFrame and Series

As with NumPy arrays, arithmetic between DataFrame and Series is well-defined. First, as a motivating example, consider the difference between a 2D array and one of its rows:

```
In [143]: arr = np.arange(12.).reshape((3, 4))

In [144]: arr
Out[144]:
```

```
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  6.,  7.]])
```

```

[ 8.,  9., 10., 11.])

In [145]: arr[0]
Out[145]: array([ 0.,  1.,  2.,  3.])

In [146]: arr - arr[0]
Out[146]:
array([[ 0.,  0.,  0.,  0.],
       [ 4.,  4.,  4.,  4.],
       [ 8.,  8.,  8.,  8.]])

```

This is referred to as *broadcasting* and is explained in more detail in [Chapter 12](#). Operations between a DataFrame and a Series are similar:

```

In [147]: frame = DataFrame(np.arange(12.).reshape((4, 3)), columns=list('bde'),
.....:                    index=['Utah', 'Ohio', 'Texas', 'Oregon'])

In [148]: series = frame.ix[0]

In [149]: frame
Out[149]:
   b  d  e
Utah  0  1  2
Ohio  3  4  5
Texas  6  7  8
Oregon 9 10 11

In [150]: series
Out[150]:
b    0
d    1
e    2
Name: Utah

```

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

```

In [151]: frame - series
Out[151]:
   b  d  e
Utah  0  0  0
Ohio  3  3  3
Texas  6  6  6
Oregon 9  9  9

```

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 [152]: series2 = Series(range(3), index=['b', 'e', 'f'])

In [153]: frame + series2
Out[153]:
   b  d  e  f
Utah  0 NaN  3 NaN
Ohio  3 NaN  6 NaN
Texas  6 NaN  9 NaN
Oregon 9 NaN 12 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 [154]: series3 = frame['d']

In [155]: frame
Out[155]:
   b  d  e
Utah  0  1  2
Ohio  3  4  5
Texas  6  7  8
Oregon 9 10 11

In [156]: series3
Out[156]:
d
Utah    1
Ohio    4
Texas    7
Oregon 10

```

```

Out[155]:          b  d  e
Utah      0  1  2
Ohio      3  4  5
Texas     6  7  8
Oregon    9 10 11
Name: d

In [157]: frame.sub(series3, axis=0)
Out[157]:
          b  d  e
Utah     -1  0  1
Ohio     -1  0  1
Texas    -1  0  1
Oregon   -1  0  1

```

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 and broadcast across.

Function application and mapping

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

```

In [158]: frame = DataFrame(np.random.randn(4, 3), columns=list('bde'),
.....:                      index=['Utah', 'Ohio', 'Texas', 'Oregon'])

In [159]: frame
Out[159]:
          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 [160]: np.abs(frame)
Out[160]:
          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 1D arrays to each column or row. DataFrame's `apply` method does exactly this:

```

In [161]: f = lambda x: x.max() - x.min()

In [162]: frame.apply(f)
Out[162]:
b    1.802165
d    1.684034
e    2.689627

In [163]: frame.apply(f, axis=1)
Out[163]:
Utah    0.998382
Ohio    2.521511
Texas    0.676115
Oregon   2.542656

```

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 [164]: def f(x):
.....:     return Series([x.min(), x.max()], index=['min', 'max'])

In [165]: frame.apply(f)

```

```
Out[165]:
      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 [166]: format = lambda x: '%.2f' % x

In [167]: frame.applymap(format)
Out[167]:
      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 [168]: frame['e'].map(format)
Out[168]:
Utah      -0.52
Ohio       1.39
Texas       0.77
Oregon    -1.30
Name: e
```

Sorting and ranking

Sorting a data set 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 [169]: obj = Series(range(4), index=['d', 'a', 'b', 'c'])

In [170]: obj.sort_index()
Out[170]:
a      1
b      2
c      3
d      0
```

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

```
In [171]: frame = DataFrame(np.arange(8).reshape((2, 4)), index=['three', 'one'],
.....:                      columns=['d', 'a', 'b', 'c'])

In [172]: frame.sort_index()
Out[172]:
      d  a  b  c
one   4  5  6  7
three 0  1  2  3

In [173]: frame.sort_index(axis=1)
Out[173]:
      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 [174]: frame.sort_index(axis=1, ascending=False)
Out[174]:
```

	d	c	b	a
three	0	3	2	1
one	4	7	6	5

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

```
In [175]: obj = Series([4, 7, -3, 2])

In [176]: obj.order()
Out[176]:
```

2	-3
3	2
0	4
1	7

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

```
In [177]: obj = Series([4, np.nan, 7, np.nan, -3, 2])

In [178]: obj.order()
Out[178]:
```

4	-3
5	2
0	4
2	7
1	NaN
3	NaN

On DataFrame, you may want to sort by the values in one or more columns. To do so, pass one or more column names to the `by` option:

```
In [179]: frame = DataFrame({'b': [4, 7, -3, 2], 'a': [0, 1, 0, 1]})

In [180]: frame
Out[180]:
```

	a	b
0	0	4
1	1	7
2	0	-3
3	1	2

```
In [181]: frame.sort_index(by='b')
Out[181]:
```

	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 [182]: frame.sort_index(by=['a', 'b'])
Out[182]:
```

	a	b
2	0	-3
0	0	4
3	1	2
1	1	7

Ranking is closely related to sorting, assigning ranks from one through the number of valid data points in an array. It is similar to the indirect sort indices produced by `numpy.argsort`, except that ties are broken according to a rule. 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 [183]: obj = Series([7, -5, 7, 4, 2, 0, 4])
```

```
In [184]: obj.rank()
```

```
Out[184]:
```

```
0    6.5
1     1.0
2     6.5
3     4.5
4     3.0
5     2.0
6     4.5
```

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

```
In [185]: obj.rank(method='first')
```

```
Out[185]:
```

```
0    6
1    1
2    7
3    4
4    3
5    2
6    5
```

Naturally, you can rank in descending order, too:

```
In [186]: obj.rank(ascending=False, method='max')
```

```
Out[186]:
```

```
0    2
1    7
2    2
3    4
4    5
5    6
6    4
```

See [Table 5-8](#) for a list of tie-breaking methods available. `DataFrame` can compute ranks over the rows or the columns:

```
In [187]: frame = DataFrame({'b': [4.3, 7, -3, 2], 'a': [0, 1, 0, 1],
.....:                      'c': [-2, 5, 8, -2.5]})
```

```
In [188]: frame
```

```
Out[188]:
```

```
   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 [189]: frame.rank(axis=1)
```

```
Out[189]:
```

```
   a  b  c
0  2  3  1
1  1  3  2
2  2  1  3
3  2  3  1
```

Table 5-8. 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.

Axis indexes with duplicate values

Up until now all of the examples I've showed you 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 [190]: obj = Series(range(5), index=['a', 'a', 'b', 'b', 'c'])

In [191]: obj
Out[191]:
a    0
a    1
b    2
b    3
c    4
```

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

```
In [192]: obj.index.is_unique
Out[192]: False
```

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

```
In [193]: obj['a']
Out[193]:
a    0
a    1

In [194]: obj['c']
Out[194]: 4
```

The same logic extends to indexing rows in a DataFrame:

```
In [195]: df = DataFrame(np.random.randn(4, 3), index=['a', 'a', 'b', 'b'])

In [196]: df
Out[196]:
      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 [197]: df.ix['b']
Out[197]:
      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
```

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 equivalent methods of vanilla NumPy arrays, they are all built from the ground up to exclude missing data. Consider a small DataFrame:

```
In [198]: df = 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 [199]: df
Out[199]:
   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 [200]: df.sum()
Out[200]:
one    9.25
two   -5.80
```

Passing `axis=1` sums over the rows instead:

```
In [201]: df.sum(axis=1)
Out[201]:
a    1.40
b    2.60
c     NaN
d   -0.55
```

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

```
In [202]: df.mean(axis=1, skipna=False)
Out[202]:
a     NaN
b    1.300
c     NaN
d   -0.275
```

See [Table 5-9](#) for a list of common options for each reduction method options.

Table 5-9. 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 [203]: df.idxmax()
Out[203]:
one      b
two      d
```

Other methods are *accumulations*:

```
In [204]: df.cumsum()
Out[204]:
      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 [205]: df.describe()
Out[205]:
      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 alternate summary statistics:

```
In [206]: obj = Series(['a', 'a', 'b', 'c'] * 4)

In [207]: obj.describe()
Out[207]:
count      16
unique       3
top         a
freq        8
```

See [Table 5-10](#) for a full list of summary statistics and related methods.

Table 5-10. 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 values 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
var	Sample variance of values
std	Sample standard deviation of values
skew	Sample skewness (3rd moment) of values
kurt	Sample kurtosis (4th moment) of values
cumsum	Cumulative sum of values
cummin, cummax	Cumulative minimum or maximum of values, respectively
cumprod	Cumulative product of values
diff	Compute 1st 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:

```
import pandas.io.data as web

all_data = {}
for ticker in ['AAPL', 'IBM', 'MSFT', 'GOOG']:
    all_data[ticker] = web.get_data_yahoo(ticker, '1/1/2000', '1/1/2010')

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

I now compute percent changes of the prices:

```
In [209]: returns = price.pct_change()

In [210]: returns.tail()
```

```
Out[210]:
```

	AAPL	GOOG	IBM	MSFT
Date				
2009-12-24	0.034339	0.011117	0.004420	0.002747
2009-12-28	0.012294	0.007098	0.013282	0.005479
2009-12-29	-0.011861	-0.005571	-0.003474	0.006812
2009-12-30	0.012147	0.005376	0.005468	-0.013532
2009-12-31	-0.004300	-0.004416	-0.012609	-0.015432

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 [211]: returns.MSFT.corr(returns.IBM)
Out[211]: 0.49609291822168838
```

```
In [212]: returns.MSFT.cov(returns.IBM)
Out[212]: 0.00021600332437329015
```

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

```
In [213]: returns.corr()
Out[213]:
```

	AAPL	GOOG	IBM	MSFT
AAPL	1.000000	0.470660	0.410648	0.424550
GOOG	0.470660	1.000000	0.390692	0.443334
IBM	0.410648	0.390692	1.000000	0.496093
MSFT	0.424550	0.443334	0.496093	1.000000

```
In [214]: returns.cov()
Out[214]:
```

	AAPL	GOOG	IBM	MSFT
AAPL	0.001028	0.000303	0.000252	0.000309
GOOG	0.000303	0.000580	0.000142	0.000205
IBM	0.000252	0.000142	0.000367	0.000216
MSFT	0.000309	0.000205	0.000216	0.000516

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 [215]: returns.corrwith(returns.IBM)
Out[215]:
```

AAPL	0.410648
GOOG	0.390692
IBM	1.000000
MSFT	0.496093

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

```
In [216]: returns.corrwith(volume)
Out[216]:
```

AAPL	-0.057461
GOOG	0.062644

```
IBM      -0.007900
MSFT     -0.014175
```

Passing `axis=1` does things row-wise instead. In all cases, the data points are aligned by label before computing the correlation.

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 [217]: obj = 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 [218]: uniques = obj.unique()

In [219]: uniques
Out[219]: 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 [220]: obj.value_counts()
Out[220]:
c      3
a      3
b      2
d      1
```

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 [221]: pd.value_counts(obj.values, sort=False)
Out[221]:
a      3
b      2
c      3
d      1
```

Lastly, `isin` is responsible for vectorized set membership and can be very useful in filtering a data set down to a subset of values in a Series or column in a DataFrame:

```
In [222]: mask = obj.isin(['b', 'c'])

In [223]: mask
Out[223]:
0      True
1     False
2     False
3     False
4     False
5      True
6      True

In [224]: obj[mask]
Out[224]:
0      c
5      b
6      b
7      c
8      c
```

```
7    True
8    True
```

See [Table 5-11](#) for a reference on these methods.

Table 5-11. Unique, value counts, and binning methods

Method	Description
<code>isin</code>	Compute boolean array indicating whether each Series value is contained in the passed sequence of values.
<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 [225]: data = DataFrame({'Qu1': [1, 3, 4, 3, 4],
.....:                    'Qu2': [2, 3, 1, 2, 3],
.....:                    'Qu3': [1, 5, 2, 4, 4]})
```

```
In [226]: data
Out[226]:
   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 [227]: result = data.apply(pd.value_counts).fillna(0)

In [228]: result
Out[228]:
   Qu1  Qu2  Qu3
1     1     1     1
2     0     2     1
3     2     2     0
4     2     0     2
5     0     0     1
```

Handling Missing Data

Missing data is common in most data analysis applications. One of the goals in designing pandas was to make working with missing data as painless as possible. For example, all of the descriptive statistics on pandas objects exclude missing data as you've seen earlier in the chapter.

pandas uses the floating point value **NaN** (Not a Number) to represent missing data in both floating as well as in non-floating point arrays. It is just used as a *sentinel* that can be easily detected:

```
In [229]: string_data = Series(['aardvark', 'artichoke', np.nan, 'avocado'])

In [230]: string_data
Out[230]:
0    aardvark
1    artichoke
2         NaN
3     avocado

In [231]: string_data.isnull()
Out[231]:
0    False
1    False
2     True
3    False
```

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

```
In [232]: string_data[0] = None

In [233]: string_data.isnull()
Out[233]:
0     True
1    False
2     True
3    False
```

I do not claim that pandas’s NA representation is optimal, but it is simple and reasonably consistent. It’s the best solution, with good all-around performance characteristics and a simple API, that I could concoct in the absence of a true NA data type or bit pattern in NumPy’s data types. Ongoing development work in NumPy may change this in the future.

Table 5-12. 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 like-type object containing boolean values indicating which values are missing / NA.
notnull	Negation of isnull.

Filtering Out Missing Data

You have a number of options for filtering out missing data. While doing it by hand is always an option, **dropna** can be very helpful. On a Series, it returns the Series with only the non-null data and index values:

```
In [234]: from numpy import nan as NA

In [235]: data = Series([1, NA, 3.5, NA, 7])

In [236]: data.dropna()
Out[236]:
```

```
0    1.0
2    3.5
4    7.0
```

Naturally, you could have computed this yourself by boolean indexing:

```
In [237]: data[data.notnull()]
Out[237]:
0    1.0
2    3.5
4    7.0
```

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

```
In [238]: data = DataFrame([[1., 6.5, 3.], [1., NA, NA],
.....:                      [NA, NA, NA], [NA, 6.5, 3.]])

In [239]: cleaned = data.dropna()

In [240]: data
Out[240]:
   0    1    2
0  1  6.5    3
1  1  NaN  NaN
2 NaN  NaN  NaN
3 NaN  6.5    3

In [241]: cleaned
Out[241]:
   0    1    2
0  1  6.5    3
```

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

```
In [242]: data.dropna(how='all')
Out[242]:
   0    1    2
0  1  6.5    3
1  1  NaN  NaN
3 NaN  6.5    3
```

Dropping columns in the same way is only a matter of passing `axis=1`:

```
In [243]: data[4] = NA

In [244]: data
Out[244]:
   0    1    2    4
0  1  6.5    3 NaN
1  1  NaN  NaN NaN
2 NaN  NaN  NaN NaN
3 NaN  6.5    3 NaN

In [245]: data.dropna(axis=1, how='all')
Out[245]:
   0    1    2
0  1  6.5    3
1  1  NaN  NaN
2 NaN  NaN  NaN
3 NaN  6.5    3
```

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 [246]: df = DataFrame(np.random.randn(7, 3))

In [247]: df.ix[:4, 1] = NA; df.ix[:2, 2] = NA
```

```

In [248]: df
Out[248]:
   0         1         2
0 -0.577087    NaN    NaN
1  0.523772    NaN    NaN
2 -0.713544    NaN    NaN
3 -1.860761    NaN  0.560145
4 -1.265934    NaN -1.063512
5  0.332883 -2.359419 -0.199543
6 -1.541996 -0.970736 -1.307030

In [249]: df.dropna(thresh=3)
Out[249]:
   0         1         2
5  0.332883 -2.359419 -0.199543
6 -1.541996 -0.970736 -1.307030

```

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 [250]: df.fillna(0)
Out[250]:
   0         1         2
0 -0.577087  0.000000  0.000000
1  0.523772  0.000000  0.000000
2 -0.713544  0.000000  0.000000
3 -1.860761  0.000000  0.560145
4 -1.265934  0.000000 -1.063512
5  0.332883 -2.359419 -0.199543
6 -1.541996 -0.970736 -1.307030

```

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

```

In [251]: df.fillna({'1': 0.5, '3': -1})
Out[251]:
   0         1         2
0 -0.577087  0.500000    NaN
1  0.523772  0.500000    NaN
2 -0.713544  0.500000    NaN
3 -1.860761  0.500000  0.560145
4 -1.265934  0.500000 -1.063512
5  0.332883 -2.359419 -0.199543
6 -1.541996 -0.970736 -1.307030

```

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

```

# always returns a reference to the filled object
In [252]: _ = df.fillna(0, inplace=True)

In [253]: df
Out[253]:
   0         1         2
0 -0.577087  0.000000  0.000000
1  0.523772  0.000000  0.000000
2 -0.713544  0.000000  0.000000
3 -1.860761  0.000000  0.560145

```

```

4 -1.265934  0.000000 -1.063512
5  0.332883 -2.359419 -0.199543
6 -1.541996 -0.970736 -1.307030

```

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

```
In [254]: df = DataFrame(np.random.randn(6, 3))
```

```
In [255]: df.ix[2:, 1] = NA; df.ix[4:, 2] = NA
```

```
In [256]: df
```

```

Out[256]:
   0         1         2
0  0.286350  0.377984 -0.753887
1  0.331286  1.349742  0.069877
2  0.246674      NaN  1.004812
3  1.327195      NaN -1.549106
4  0.022185      NaN      NaN
5  0.862580      NaN      NaN

```

```
In [257]: df.fillna(method='ffill')
```

```

Out[257]:
   0         1         2
0  0.286350  0.377984 -0.753887
1  0.331286  1.349742  0.069877
2  0.246674  1.349742  1.004812
3  1.327195  1.349742 -1.549106
4  0.022185  1.349742 -1.549106
5  0.862580  1.349742 -1.549106

```

```
In [258]: df.fillna(method='ffill', limit=2)
```

```

Out[258]:
   0         1         2
0  0.286350  0.377984 -0.753887
1  0.331286  1.349742  0.069877
2  0.246674  1.349742  1.004812
3  1.327195  1.349742 -1.549106
4  0.022185      NaN -1.549106
5  0.862580      NaN -1.549106

```

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 [259]: data = Series([1., NA, 3.5, NA, 7])
```

```
In [260]: data.fillna(data.mean())
```

```

Out[260]:
0    1.000000
1    3.833333
2    3.500000
3    3.833333
4    7.000000

```

See [Table 5-13](#) for a reference on `fillna`.

Table 5-13. *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