Getting Started with Pandas

March 31, 2015

1 Pandas in a nutshell

Pandas is the Python library designed specifically for data analysis. The author of *Python for Data Analysis*, Wes McKinney, began developing Pandas in 2008

> while at AQR Capital Management out of need for a performant, flexible tool to perform quantitative analysis on financial data. Before leaving AQR he was able to convince management to allow him to open source the library.
>
> Another AQR employee, Chang She, joined the effort in 2012 as the second major contributor to the library. Right around that time, the library became popular in the Python community, and many more contributors joined the project making it one of the most vital and active data analysis libraries for Python. (Wikipedia

Pandas can be thought of the Python equivalent of Microsoft Excel. It abstracts the notion of the spreadsheet, allowing the user to use powerful and robust analytical tools generally to automate repeated processes.

The twin centerpieces of the Pandas library are the Series and the DataFrame. The Series class is, at its core, a one-dimensional NumPy array, surrounded by additional information, such as its index. The DataFrame is conceptually an *array* of Series classes, each sharing the same index.

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

We will be using Wes McKinney's GitHub notebook as a skeleton. He imports the following libraries for later use:

```
In [2]: from __future__ import division
        from numpy.random import randn
        import numpy as np
        import os
        import matplotlib.pyplot as plt
        np.random.seed(12345)
        plt.rc('figure', figsize=(10, 6))
        from pandas import Series, DataFrame
        import pandas as pd
        np.set_printoptions(precision=4)
```

2 Introduction to pandas data structures

2.1 Series

Consider the following input:

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

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

We have set the variable `obj` to reference a new `Pandas Series`, which we initialized by giving a `Python` list as input. Notice that `Pandas` automatically interprets the input data as type `int64`, which indicates that it is fairly smart! Also, notice that upon printing `obj` we see *two* columns. The first column is the *index* of the `Series` class, which is presently the natural index, `range(4)`. The second column is the input data that we gave initially, which `Pandas` refers to as the *values* of `obj`. You can access these columns individualy by calling `obj.index` and `obj.values`, respectively. For example:

```
In [4]: print obj.index, "\n\n"
        print obj.values
```

```
Int64Index([0, 1, 2, 3], dtype='int64')


[ 4  7 -5  3]
```

2.1.1 Indexing

As previously mentioned, the natural index simply starts at 0 and increments integers to the size of the list of the input values. Alternatively, we can specify the index explicitly when we initialize the `Series`, as in the following:

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

```
Out[5]: d    4
        b    7
        a   -5
        c    3
        dtype: int64
```

What happens when you examine the index now?

```
In [6]: obj2.index
```

```
Out[6]: Index([u'd', u'b', u'a', u'c'], dtype='object')
```

Don't be alarmed by the `u` prefix to each of the index values. In `Python` (as well as in other languages, this simply indicates a Unicode string. Ostensibly, there is no difference between normal strings and Unicode strings.

You can access a particular member of the `Series` data by specifying its index. For example,

```
In [7]: obj2['a']
```

```
Out[7]: -5
```

This allows you to change the value of specific entries in your `Series` data. Additionally, you can call *sub*`Series` by specifying a sublist of the index.

```
In [8]: obj2['d'] = 6
        obj2[['c', 'a', 'd']]
```

```
Out[8]: c    3
        a   -5
        d    6
        dtype: int64
```

A powerful tool in `Pandas` is the ability to concisely access data meeting `Boolean` qualifications. In the case below, `obj2 > 0` is given as the "index," and the output is the sub`Series` of `obj2` for which all entries are positive.

```
In [9]: obj2[obj2 > 0]
```

```
Out[9]: d    6
        b    7
        c    3
        dtype: int64
```

2.1.2 Aside

What is `obj2 > 0` actually?

```
In [10]: obj2 > 0
```

```
Out[10]: d     True
         b     True
         a    False
         c     True
         dtype: bool
```

This is actually a neat property of `Pandas` which is similar to `NumPy`. In `NumPy`, suppose you are given an array:

```
In [11]: arr = np.random.rand(5) * 2.0 - 1.0
```

The actual array itself is given by

```
In [12]: arr
```

```
Out[12]: array([ 0.8592, -0.3672, -0.6322, -0.5909,  0.1355])
```

The `Boolean` array specifying which elements of `arr` are positive is given by

```
In [13]: boolArr = arr > 0.0
         boolArr
```

```
Out[13]: array([ True, False, False, False,  True], dtype=bool)
```

Similarly, you can generate a new `Series` of `Boolean` values by subjecting the original `Series` to a `Boolean` statement, as we did above.

2.1.3 Broadcasting

Like `NumPy`, we can *broadcast* arithmetic operations onto `Series` data. For example,

```
In [14]: obj2 * 2
```

```
Out[14]: d    12
         b    14
         a   -10
         c     6
         dtype: int64
```

returns a `Series` whose values are doubled, and

```
In [15]: np.exp(obj2)
```

```
Out[15]: d     403.428793
         b    1096.633158
         a       0.006738
         c      20.085537
         dtype: float64
```

returns a `Series` whose values have all been subject to the transformation $x \mapsto e^x$. Notice additionally that the `dtype` of `obj2` has automatically been changed from `int64` to `float64`. Again, `Pandas` is being smart!

2.1.4 Querying a `Series`

In `Python`, there is a binary operator called `in`, which takes two "arguments." The left-hand argument is can be any type of data (or object, we won't get into this), while the right-hand argument is some type of iterable object. Then `in` returns `True` if the left-hand argument is an *element* of the right-hand argument. Mathematically, this is equivalent to set membership. For example,

```
In [16]: odds = [i for i in range(20) if i%2 == 1]
         print 3 in odds, "|", 2 in odds
```

```
True | False
```

is equivalent to noting that if

$$\text{Odds} = \{n : 0 \le n < 20 \text{ and } n \text{ is odd}\}$$

we have that

$$3 \in \text{Odds}$$

while

$$2 \notin \text{Odds}.$$

(In fact, we have already seen `in` in action with `Python`'s `for` loop, which has the form

```
for element in iterative_object:
```

indicating that the code should loop through every element `element` that is a member of `iterative_object`.)

You can use `in` with `Pandas` `Series` to test that an element is a member of the index of the `Series`. For example,

```
In [17]: 'b' in obj2
```

```
Out[17]: True
```

```
In [18]: 'e' in obj2
```

```
Out[18]: False
```

2.1.5 Aside

We have talked about the native Python list data type. There is another important native data type in Python, called a dict, which you can learn about more here. Python dict types are similar to association lists in Scheme, in that they require a lookup key in order to access elements.

Crucially, Pandas can create a Series from a dict by interpreting the key for each item as its corresponding index value, which is actually quite natural. In this sense, I find that it is useful to think of the relationship between NumPy and Pandas as akin to the relationship between a list and a dict.

```
In [19]: sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}
         obj3 = Series(sdata)
         obj3

Out[19]: Ohio      35000
         Oregon    16000
         Texas     71000
         Utah       5000
         dtype: int64
```

What happens when you use an existing dataset with a new index, in which there is a new, unfilled index?

```
In [20]: states = ['California', 'Ohio', 'Oregon', 'Texas']
         obj4 = Series(sdata, index=states)
         obj4

Out[20]: California      NaN
         Ohio          35000
         Oregon        16000
         Texas         71000
         dtype: float64
```

In this case, California is a previously-unused index, which has no corresponding value. Thus, Pandas initializes the new Series with the value corresponding to California set to NaN (Python-speak for null).

The isnull method returns a Series of Boolean values whenever the original Series has a null (NaN) value.

```
In [21]: pd.isnull(obj4)

Out[21]: California     True
         Ohio          False
         Oregon        False
         Texas         False
         dtype: bool
```

The notnull method does the exact opposite!

```
In [22]: pd.notnull(obj4)

Out[22]: California    False
         Ohio           True
         Oregon         True
         Texas          True
         dtype: bool
```

The methods isnull and notnull are "static" in the sense that they can be called straight from the pd module or for a specific Series object.

```
In [23]: obj4.isnull()

Out[23]: California     True
         Ohio          False
         Oregon        False
         Texas         False
         dtype: bool
```

Recall the two `Series`, `obj3` and `obj4`:

```
In [24]: print "\tobj3:\n",obj3, "\n\n\tobj4:\n", obj4

obj3:
Ohio      35000
Oregon    16000
Texas     71000
Utah       5000
dtype: int64

        obj4:
California      NaN
Ohio          35000
Oregon        16000
Texas         71000
dtype: float64
```

Arithmetic operations between distinct `Series` objects work conservatively. For data types, `int64 + float64 = float64` to preserve the decimal information. The summed index is the union of the two indices. Consider the following example:

```
In [25]: obj3 + obj4

Out[25]: California       NaN
         Ohio           70000
         Oregon         32000
         Texas         142000
         Utah             NaN
         dtype: float64
```

No entry for `California` exists in `obj3`, while no entry for `Utah` exists in `obj4`. `Pandas` interprets `NaN + x = NaN` for all `x`, so the resultant `Series` sets `NaN` for both `California` and `Utah`.

We can set some metadata for a `Series`, such as the name of the values column and the name of the index column.

```
In [26]: obj4.name = 'population'
         obj4.index.name = 'state'
         obj4

Out[26]: state
         California      NaN
         Ohio          35000
         Oregon        16000
         Texas         71000
         Name: population, dtype: float64
```

You can also completely change the index at any time. This is something we will get into more detail later.

```
In [27]: obj.index = ['Bob', 'Steve', 'Jeff', 'Ryan']
         obj
```

```
Out[27]: Bob      4
         Steve    7
         Jeff    -5
         Ryan     3
         dtype: int64
```

2.2 DataFrame

Like we said before, you can think of a `DataFrame` as an *array* of `Series` objects. Specifically, a `DataFrame` is a two-dimensional array of `Series` objects, all indexed by the same index series. You can also think of a `DataFrame` as a single Microsoft Excel spreadsheet.

One way to initialize a `DataFrame` is by giving a `dict` where each key indicates a `Python` list.

```
In [28]: 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)

         frame
```

```
Out[28]:    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
```

You can reorder the columns in a new `DataFrame` using the following argument:

```
In [29]: DataFrame(data, columns=['year', 'state', 'pop'])
```

```
Out[29]:    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
```

Similarly, the `index` optional argument in `DataFrame` allows you to specify the index list. Additionally, adding a `debt` column with no corresponding data in `data` will initialize a column filled with `NaN` entries.

```
In [30]: frame2 = DataFrame(data, columns=['year', 'state', 'pop', 'debt'],
                            index=['one', 'two', 'three', 'four', 'five'])
         frame2
```

```
Out[30]:        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
```

You can access the columns of a `DataFrame` as follows:

```
In [31]: frame2.columns
```

```
Out[31]: Index([u'year', u'state', u'pop', u'debt'], dtype='object')
```

You can slice a particular column by specifying its column name. Notice how this returns a `Series`.

```
In [32]: frame2['state']
```

```
Out[32]: one        Ohio
         two        Ohio
         three      Ohio
         four     Nevada
         five     Nevada
         Name: state, dtype: object
```

Alternatively, you can slice a column using the following syntax:

```
In [33]: frame2.year
```

```
Out[33]: one      2000
         two      2001
         three    2002
         four     2001
         five     2002
         Name: year, dtype: int64
```

To slice a row, you can specify an index, which will return a `Series` representing the row at the index.

```
In [34]: frame2.ix['three']
```

```
Out[34]: year     2002
         state    Ohio
         pop       3.6
         debt      NaN
         Name: three, dtype: object
```

Broadcasting works in the natural way that you might expect:

```
In [35]: frame2['debt'] = 16.5
         frame2
```

```
Out[35]:        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
```

You can also give a particular column a list or `ndarray`, which will then be distributed across the column.

```
In [36]: frame2['debt'] = np.arange(5.)
         frame2
```

```
Out[36]:        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
```

Finally, you can give a column of a `DataFrame` a `Series`. If you specify a `Series` with an index differing from the main `DataFrame`, then the entries of the `DataFrame` will be set to `NaN`.

```
In [37]: val = Series([-1.2, -1.5, -1.7], index=['two', 'four', 'five'])
         frame2['debt'] = val
         frame2
```

```
Out[37]:        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
```

The point of `Pandas` is there are *numerous* ways to achieve the same effect, depending on whatever is easiest for the task at hand. Here is another way to add a column:

```
In [38]: frame2['eastern'] = frame2.state == 'Ohio'
         frame2
```

```
Out[38]:        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
```

We can also use `Python`'s `del` function to remove a column:

```
In [39]: del frame2['eastern']
         frame2.columns
```

```
Out[39]: Index([u'year', u'state', u'pop', u'debt'], dtype='object')
```

One final way to initialize `DataFrame` objects is with nested `dict` objects.

```
In [40]: pop = {'Nevada': {2001: 2.4, 2002: 2.9},
                'Ohio': {2000: 1.5, 2001: 1.7, 2002: 3.6}}

         frame3 = DataFrame(pop)
         frame3
```

```
Out[40]:       Nevada  Ohio
         2000     NaN   1.5
         2001     2.4   1.7
         2002     2.9   3.6
```

You can transpose a `DataFrame` if it makes more sense to work with the rows and columns flipped.

```
In [41]: frame3.T
```

```
Out[41]:         2000  2001  2002
         Nevada   NaN   2.4   2.9
         Ohio     1.5   1.7   3.6
```

You can do this transpose operation from the outset by manually specifying the index.

```
In [42]: DataFrame(pop, index=[2001, 2002, 2003])
```

```
Out[42]:       Nevada  Ohio
         2001     2.4   1.7
         2002     2.9   3.6
         2003     NaN   NaN
```

DataFrame objects can also be initialized from dicts of Series objects.

```
In [43]: pdata = {'Ohio': frame3['Ohio'][:-1],
                  'Nevada': frame3['Nevada'][:2]}
         DataFrame(pdata)
```

```
Out[43]:       Nevada  Ohio
         2000     NaN   1.5
         2001     2.4   1.7
```

```
In [44]: frame3.index.name = 'year'; frame3.columns.name = 'state'
         frame3
```

```
Out[44]: state  Nevada  Ohio
         year
         2000      NaN   1.5
         2001      2.4   1.7
         2002      2.9   3.6
```

If you need to access the underlying ndarray from any DataFrame, use the DataFrame.values field.

```
In [45]: frame3.values
```

```
Out[45]: array([[ nan,  1.5],
                [ 2.4,  1.7],
                [ 2.9,  3.6]])
```

```
In [46]: frame2.values
```

```
Out[46]: 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)
```

2.3 Index objects

The Index is the "metadata" object for Series and DataFrame objects. We've seen ways of initializing Index objects before, so we will go over some features of these objects.

```
In [47]: obj = Series(range(3), index=['a', 'b', 'c'])
         index = obj.index
         index
```

```
Out[47]: Index([u'a', u'b', u'c'], dtype='object')
```

Index objects can be sliced like arrays.

```
In [48]: index[1:]
```

```
Out[48]: Index([u'b', u'c'], dtype='object')
```

Importantly, Index objects are not mutable, so you can't change their values in the natural way:

```
In [49]: index[1] = 'd'
```

```
    ---------------------------------------------------------------------------

    TypeError                                 Traceback (most recent call last)

    <ipython-input-49-676fdeb26a68> in <module>()
----> 1 index[1] = 'd'


    /home/alethiometryst/anaconda/lib/python2.7/site-packages/pandas/core/index.pyc in __setitem__(se
    894
    895     def __setitem__(self, key, value):
--> 896         raise TypeError("Indexes does not support mutable operations")
    897
    898     def __getitem__(self, key):


    TypeError: Indexes does not support mutable operations
```

You can initialize `Index` objects with `NumPy ndarray` objects.

```
In [50]: index = pd.Index(np.arange(3))
         obj2 = Series([1.5, -2.5, 0], index=index)
         obj2.index is index

Out[50]: True

In [51]: frame3

Out[51]: state  Nevada  Ohio
         year
         2000      NaN   1.5
         2001      2.4   1.7
         2002      2.9   3.6

In [52]: print 'Ohio' in frame3.columns, "|", 2003 in frame3.index

True | False
```

3 Essential functionality

Now that we are familiar with the basic objects in `Pandas`, we will start working with the mechanics of these objects.

3.1 Reindexing

In the previous section we mentioned that `Index` objects are immutable. Here we will address this issue.

```
In [53]: obj = Series([4.5, 7.2, -5.3, 3.6], index=['d', 'b', 'a', 'c'])
         obj
```

```
Out[53]: d    4.5
         b    7.2
         a   -5.3
         c    3.6
         dtype: float64
```

The simplest way to change an `Index` object in an existing `Series` or `DataFrame` is with the `reindex` method.

```
In [54]: obj2 = obj.reindex(['a', 'b', 'c', 'd', 'e'])
         obj2
```

```
Out[54]: a   -5.3
         b    7.2
         c    3.6
         d    4.5
         e    NaN
         dtype: float64
```

In the above example, since `"e"` was not in the original `Index`, the corresponding `Series` value is set to `NaN`. If you want to change the default fill value, `reindex` can take an additional parameter, `fill_value`.

```
In [55]: obj.reindex(['a', 'b', 'c', 'd', 'e'], fill_value=0)
```

```
Out[55]: a   -5.3
         b    7.2
         c    3.6
         d    4.5
         e    0.0
         dtype: float64
```

A different approach uses a `method` parameter that attempts to extrapolate existing data into the new `Index`. One such method is `ffill`, which "step-fills" the existing data forward. Alternatively, `bfill` "step-fills" the data backwards.

```
In [56]: obj3 = Series(['blue', 'purple', 'yellow'], index=[0, 2, 4])
         obj3.reindex(range(6), method='ffill')
```

```
Out[56]: 0      blue
         1      blue
         2    purple
         3    purple
         4    yellow
         5    yellow
         dtype: object
```

The `reindex` method works for `DataFrame` objects as well. For `DataFrame` objects, `reindex` can also specify column reindexing.

```
In [57]: frame = DataFrame(np.arange(9).reshape((3, 3)), index=['a', 'c', 'd'],
                           columns=['Ohio', 'Texas', 'California'])
         frame
```

```
Out[57]:    Ohio  Texas  California
         a     0      1           2
         c     3      4           5
         d     6      7           8
```

```
In [58]: frame.reindex(index=['a', 'b', 'c', 'd'], method='ffill',
                       columns=states)

Out[58]:    California  Ohio  Oregon  Texas
         a           2     0     NaN      1
         b           2     0     NaN      1
         c           5     3     NaN      4
         d           8     6     NaN      7
```

Alternatively, you can use `ix` to achieve the same effect more concisely.

```
In [59]: frame.ix[['a', 'b', 'c', 'd'], states]

Out[59]:    California  Ohio  Oregon  Texas
         a           2     0     NaN      1
         b         NaN   NaN     NaN    NaN
         c           5     3     NaN      4
         d           8     6     NaN      7
```

3.2 Dropping entries from an axis

Suppose you have a `Series` object with data you wish to remove. Using the `drop` method, you can specify an index element to remove.

```
In [60]: obj = Series(np.arange(5.), index=['a', 'b', 'c', 'd', 'e'])
         new_obj = obj.drop('c')
         new_obj

Out[60]: a    0
         b    1
         d    3
         e    4
         dtype: float64
```

You can also drop a list of index elements at once.

```
In [61]: obj.drop(['d', 'c'])

Out[61]: a    0
         b    1
         e    4
         dtype: float64
```

The same works for `DataFrame` objects and the `drop` method.

```
In [62]: data = DataFrame(np.arange(16).reshape((4, 4)),
                          index=['Ohio', 'Colorado', 'Utah', 'New York'],
                          columns=['one', 'two', 'three', 'four'])

In [63]: data.drop(['Colorado', 'Ohio'])

Out[63]:           one  two  three  four
         Utah        8    9     10    11
         New York   12   13     14    15
```

Additionally, `DataFrame.drop()` can remove columns by specifying an `axis` parameter.

```
In [64]: data.drop('two', axis=1)
```

```
Out[64]:           one  three  four
         Ohio        0      2     3
         Colorado    4      6     7
         Utah        8     10    11
         New York   12     14    15
```

```
In [65]: data.drop(['two', 'four'], axis=1)
```

```
Out[65]:           one  three
         Ohio        0      2
         Colorado    4      6
         Utah        8     10
         New York   12     14
```

3.3 Indexing, selection, and filtering

In this section we will explore the various techniques available for slicing `Series` and `DataFrame` objects. One the one hand, we can deal with these objects as `dict` structures, accessing elements by requesting their index keys. On the other hand, we can treat these objects as list structures, accessing elements by the order of the index list.

```
In [66]: obj = Series(np.arange(4.), index=['a', 'b', 'c', 'd'])
         obj['b']
```

```
Out[66]: 1.0
```

```
In [67]: obj[1]
```

```
Out[67]: 1.0
```

This flexibility allows you to incorporate all of the previous array slicing that worked for `NumPy ndarray` objects.

```
In [68]: obj[2:4]
```

```
Out[68]: c    2
         d    3
         dtype: float64
```

Conversely, you can use a list of `dict` keys to achieve the same end.

```
In [69]: obj[['b', 'a', 'd']]
```

```
Out[69]: b    1
         a    0
         d    3
         dtype: float64
```

Here are some alternative slicing techniques for `Series` objects.

```
In [70]: obj[[1, 3]]
```

```
Out[70]: b    1
         d    3
         dtype: float64
```

```
In [71]: obj[obj < 2]
```

```
Out[71]: a    0
         b    1
         dtype: float64
```

```
In [72]: obj['b':'c']
```

```
Out[72]: b    1
         c    2
         dtype: float64
```

You can assign values to sub-objects which then reflect on the original object.

```
In [73]: obj['b':'c'] = 5
         obj
```

```
Out[73]: a    0
         b    5
         c    5
         d    3
         dtype: float64
```

The same capabilities are extended to the DataFrame objects. The added flexibility is that the same indexing techniques also apply to column slicing as well as index slicing.

```
In [74]: data = DataFrame(np.arange(16).reshape((4, 4)),
                          index=['Ohio', 'Colorado', 'Utah', 'New York'],
                          columns=['one', 'two', 'three', 'four'])
         data
```

```
Out[74]:           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 [75]: data['two']
```

```
Out[75]: Ohio         1
         Colorado     5
         Utah         9
         New York    13
         Name: two, dtype: int64
```

```
In [76]: data[['three', 'one']]
```

```
Out[76]:           three  one
         Ohio          2    0
         Colorado      6    4
         Utah         10    8
         New York     14   12
```

The natural slicing will always refer to the index list, not the column list, which is useful to keep in mind. data[:2]

```
In [77]: data[data['three'] > 5]
```

```
Out[77]:           one  two  three  four
        Colorado    4    5      6     7
        Utah        8    9     10    11
        New York   12   13     14    15
```

Recall that you can generate a corresponding Boolean array by subjecting a DataFrame to a boolean statement, such as the following:

```
In [78]: data < 5
```

```
Out[78]:             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
```

You can use Boolean arrays to do simple thresholding to your data. You can isolate entries in your data subject to identical Boolean conditions, and manipulate these specific subsets of the data.

```
In [79]: data[data < 5] = 0
        data
```

```
Out[79]:           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
```

The DataFrame.ix field gives you even more powerful ways to slice your data. In general, slicing works by providing two arguments, an index and a column specification, and it will then return that particular subset.

```
In [80]: data.ix['Colorado', ['two', 'three']]
```

```
Out[80]: two      5
        three    6
        Name: Colorado, dtype: int64
```

You can overload requests by using a list of index or column elements. Additionally, you may reorder the indices or columns in your subset by permuting the order of the specified elements, so long as they exist in the original DataFrame.

```
In [81]: data.ix[['Colorado', 'Utah'], [3, 0, 1]]
```

```
Out[81]:           four  one  two
        Colorado     7    0    5
        Utah        11    8    9
```

The ix approach is very powerful. See if you can work through the mechanics of the next few examples to see just how versatile slicing with ix actually is.

```
In [82]: data.ix[2]
```

```
Out[82]: one       8
        two       9
        three    10
        four     11
        Name: Utah, dtype: int64
```

```
In [83]: data.ix[:'Utah', 'two']
```

```
Out[83]: Ohio        0
         Colorado    5
         Utah        9
         Name: two, dtype: int64
```

```
In [84]: data.ix[data.three > 5, :3]
```

```
Out[84]:           one  two  three
         Colorado    0    5      6
         Utah        8    9     10
         New York   12   13     14
```

3.4 Arithmetic and data alignment

As we mentioned before, we can do arithmetic on `Series` and `DataFrame` objects.

```
In [85]: s1 = Series([7.3, -2.5, 3.4, 1.5], index=['a', 'c', 'd', 'e'])
         s2 = Series([-2.1, 3.6, -1.5, 4, 3.1], index=['a', 'c', 'e', 'f', 'g'])
         print s1, "\n\n"
         print s2
```

```
a    7.3
c   -2.5
d    3.4
e    1.5
dtype: float64



a   -2.1
c    3.6
e   -1.5
f    4.0
g    3.1
dtype: float64
```

Importantly, arithmetic is only performed on elements sharing an index. If either object has an index value that the other does not, the arithmetic operation is undefined, so the resultant object contains an `NaN` element.

```
In [86]: s1 + s2
```

```
Out[86]: a    5.2
         c    1.1
         d    NaN
         e    0.0
         f    NaN
         g    NaN
         dtype: float64
```

The same holds for `DataFrame` arithmetic, except now it requires that both the index and column of each `DataFrame` object is well-defined.

```
In [87]: df1 = DataFrame(np.arange(9.).reshape((3, 3)), columns=list('bcd'),
                         index=['Ohio', 'Texas', 'Colorado'])
```

```
df2 = DataFrame(np.arange(12.).reshape((4, 3)), columns=list('bde'),
                index=['Utah', 'Ohio', 'Texas', 'Oregon'])
print df1, "\n\n"
print df2
```

```
b  c  d
Ohio      0  1  2
Texas     3  4  5
Colorado  6  7  8



        b   d   e
Utah    0   1   2
Ohio    3   4   5
Texas   6   7   8
Oregon  9  10  11
```

```
In [88]: df1 + df2
```

```
Out[88]:           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
```

3.4.1 Arithmetic methods with fill values

Often NaN values are undesirable, as they can cause errors when doing arithmetic operations on the data.

```
In [89]: df1 = DataFrame(np.arange(12.).reshape((3, 4)), columns=list('abcd'))
         df2 = DataFrame(np.arange(20.).reshape((4, 5)), columns=list('abcde'))

         print df1, "\n\n"
         print df2
```

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



    a   b   c   d   e
0   0   1   2   3   4
1   5   6   7   8   9
2  10  11  12  13  14
3  15  16  17  18  19
```

```
In [90]: df1 + df2
```

```
Out[90]:    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
```

This can be avoided by using the built-in `DataFrame.add()` method, which takes as parameters a `DataFrame` object *and* an optional `fill_value` which deals with otherwise `NaN` entries.

```
In [91]: df1.add(df2, fill_value=0)
```

```
Out[91]:     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
```

In fact, most `DataFrame` organization methods take `fill_value` as a parameter to deal with undefined cases, such as `reindex`.

```
In [92]: df1.reindex(columns=df2.columns, fill_value=0)
```

```
Out[92]:    a  b   c   d  e
         0  0  1   2   3  0
         1  4  5   6   7  0
         2  8  9  10  11  0
```

3.5 Operations between DataFrame and Series

Broadcasting `NumPy` arrays is a very useful technique for performing arithmetic operations concisely. And efficiently, actually. This is because while normal `Python` arithmetic is *interpreted*, `NumPy` arithmetic is based on *compiled* `C` code, which is much more efficient in general.

```
In [93]: arr = np.arange(12.).reshape((3, 4))
         arr
```

```
Out[93]: array([[  0.,   1.,   2.,   3.],
                [  4.,   5.,   6.,   7.],
                [  8.,   9.,  10.,  11.]])
```

Normally we think of broadcasting a scalar element onto a one-dimensional array vector. In fact, broadcasting is much more powerful, because you can broadcast an *array* over a bigger array.

```
In [94]: arr[0]
```

```
Out[94]: array([ 0.,  1.,  2.,  3.])
```

```
In [95]: arr - arr[0]
```

```
Out[95]: array([[ 0.,  0.,  0.,  0.],
                [ 4.,  4.,  4.,  4.],
                [ 8.,  8.,  8.,  8.]])
```

`DataFrame` and `Series` objects work along similar lines.

```
In [96]: frame = DataFrame(np.arange(12.).reshape((4, 3)), columns=list('bde'),
                           index=['Utah', 'Ohio', 'Texas', 'Oregon'])
         series = frame.ix[0]
         print frame, "\n\n"
         print series
```

```
b   d   e
Utah     0   1   2
Ohio     3   4   5
Texas    6   7   8
Oregon   9  10  11


b    0
d    1
e    2
Name: Utah, dtype: float64
```

You can broadcast the values in a `Series` over its parent `DataFrame` as you would with `NumPy ndarrays`.

```
In [97]: frame - series
```

```
Out[97]:         b  d  e
         Utah    0  0  0
         Ohio    3  3  3
         Texas   6  6  6
         Oregon  9  9  9
```

Of course, if either a `Series` and `DataFrame` object has index or column values the other does not, the undefined arithmetic simply is sent to `NaN`. (We discussed ways to avoid this issue in the previous sections).

```
In [98]: series2 = Series(range(3), index=['b', 'e', 'f'])
         frame + series2
```

```
Out[98]:         b   d   e   f
         Utah    0 NaN   3 NaN
         Ohio    3 NaN   6 NaN
         Texas   6 NaN   9 NaN
         Oregon  9 NaN  12 NaN
```

```
In [99]: series3 = frame['d']

         print frame, "\n\n"
         print series3
```

```
b   d   e
Utah     0   1   2
Ohio     3   4   5
Texas    6   7   8
Oregon   9  10  11



Utah       1
Ohio       4
Texas      7
Oregon    10
Name: d, dtype: float64
```

Using the built-in `DataFrame` arithmetic operations such as `add` or `sub` gives the option to specify the axis (0: index, 1: columns) over which the arithmetic will take place (again, you can use `fill_value` to avoid potential `NaN` values).

```
In [100]: frame.sub(series3, axis=0)
```

```
Out[100]:         b  d  e
          Utah   -1  0  1
          Ohio   -1  0  1
          Texas  -1  0  1
          Oregon -1  0  1
```

3.6 Function application and mapping

One of the most important capabilities of `Series` and `DataFrame` is the ability to apply function transformations to the data. Every `ufunc` defined by `NumPy` can be applied to a `DataFrame` (or `Series`) object.

```
In [101]: frame = DataFrame(np.random.randn(4, 3), columns=list('bde'),
                            index=['Utah', 'Ohio', 'Texas', 'Oregon'])
          frame
```

```
Out[101]:                b         d         e
          Utah    0.450554  0.092673  1.248133
          Ohio    0.768101  1.248804  0.774191
          Texas  -0.319657 -0.624964  1.078814
          Oregon  0.544647  0.855588  1.343268
```

For example, you can apply a nonnegativity transform by including a built-in `NumPy` absolute value.

```
In [102]: np.abs(frame)
```

```
Out[102]:                b         d         e
          Utah    0.450554  0.092673  1.248133
          Ohio    0.768101  1.248804  0.774191
          Texas   0.319657  0.624964  1.078814
          Oregon  0.544647  0.855588  1.343268
```

You can define and apply custom functions in two fashions. One is by using lambdas to construct anonymous functions:

```
In [103]: frame.apply(lambda x: x.max() - x.min())
```

```
Out[103]: b    1.087758
          d    1.873768
          e    0.569077
          dtype: float64
```

```
In [104]: frame.apply(lambda x: x.max() - x.min(), axis=1)
```

```
Out[104]: Utah      1.155460
          Ohio      0.480703
          Texas     1.703778
          Oregon    0.798621
          dtype: float64
```

Alternatively, you can define your own unary function and simply apply it using the same overall approach.

```
In [105]: def f(x):
              return Series([x.min(), x.max()], index=['min', 'max'])
          frame.apply(f)
```

```
Out[105]:             b         d         e
          min -0.319657 -0.624964  0.774191
          max  0.768101  1.248804  1.343268
```

For presentations and general readability, it is useful to format decimal or date values into condensed forms, and Pandas lets you achieve this by using the applymap method for DataFrame and Series objects. The difference between apply and applymap is rather subtle and often functionally neglibible, but the idea is that apply works on a particular subsets of rows or columns, whereas applymap is element-wise.

```
In [106]: format = lambda x: '%.2f' % x
          frame.applymap(format)

Out[106]:            b      d     e
          Utah     0.45   0.09  1.25
          Ohio     0.77   1.25  0.77
          Texas   -0.32  -0.62  1.08
          Oregon   0.54   0.86  1.34
```

Alternatively, you can use the built-in Python map function.

```
In [107]: frame['e'].map(format)

Out[107]: Utah      1.25
          Ohio      0.77
          Texas     1.08
          Oregon    1.34
          Name: e, dtype: object
```

Exercise: Determine which of apply or map is computationally more efficient.

3.7 Sorting and ranking

One fundamental problem in data analysis, let alone computer science in general, is sorting data. Pandas provides a number of techniques for sorting information in the index, columns, and the actual data itself.

The first technique is sort_index, which is a method for both Series and DataFrame objects. For Series objects, sort_index works as follows:

```
In [108]: obj = Series(range(4), index=['d', 'a', 'b', 'c'])
          obj.sort_index()

Out[108]: a    1
          b    2
          c    3
          d    0
          dtype: int64
```

Since there is only one meaningful index to sort, the labels, sort_index is a very intuitive method. I want to point out that sort_index does not have side-effects; that is, calling sort_index on an object does not actually change the internals of the object itself. Instead, a sorted copy of the original object is produced.

The method sort_index works similarly with DataFrame objects, but now there are two potential axes along which to sort. The default is the index, as we see below:

```
In [109]: frame = DataFrame(np.arange(8).reshape((2, 4)), index=['three', 'one'],
                            columns=['d', 'a', 'b', 'c'])
          frame.sort_index()

Out[109]:        d  a  b  c
          one    4  5  6  7
          three  0  1  2  3
```

By specifying the axis as a parameter, one can choose the columns instead. (Recall that in `Python` everything begins at 0, so the second axis corresponds to axis number 1).

```
In [110]: frame.sort_index(axis=1)

Out[110]:        a  b  c  d
          three  1  2  3  0
          one    5  6  7  4
```

The `sort_index` method also allows you to flip the ordering by specifying the `ascending` parameter.

```
In [111]: frame.sort_index(axis=1, ascending=False)

Out[111]:        d  c  b  a
          three  0  3  2  1
          one    4  7  6  5
```

If you want to sort the elements themselves, as opposed to the index, `Pandas` provides the `order` method for `Series` objects.

```
In [112]: obj = Series([4, 7, -3, 2])
          obj.order()

Out[112]: 2   -3
          3    2
          0    4
          1    7
          dtype: int64
```

By default, `NaN` values are placed at the end upon sorting the `Series`.

```
In [113]: obj = Series([4, np.nan, 7, np.nan, -3, 2])
          obj.order()

Out[113]: 4    -3
          5     2
          0     4
          2     7
          1   NaN
          3   NaN
          dtype: float64
```

For `DataFrame` objects you can specify the index or column you wish to sort. Additionally, if your data set is properly constructed, you can sort by two columns or indices, as the below example exhibits:

```
In [114]: frame = DataFrame({'b': [4, 7, -3, 2], 'a': [0, 1, 0, 1]})

          frame.sort_index(by='b')

Out[114]:    a  b
          2  0 -3
          3  1  2
          0  0  4
          1  1  7

In [115]: frame.sort_index(by=['a', 'b'])

Out[115]:    a  b
          2  0 -3
          0  0  4
          3  1  2
          1  1  7
```

3.8 Axis indexes with duplicate values

It is possible for a `Series` or `DataFrame` object not to have a unique index. For example:

```
In [116]: obj = Series(range(5), index=['a', 'a', 'b', 'b', 'c'])
          obj

Out[116]: a    0
          a    1
          b    2
          b    3
          c    4
          dtype: int64
```

`Pandas` has a field for the index of any object to indicate whether or ot the index is unique (no duplicate indices).

```
In [117]: obj.index.is_unique

Out[117]: False
```

If an index is not unique, then slicing the object for a repeated index returns a sub-object. For example:

```
In [118]: obj['a']

Out[118]: a    0
          a    1
          dtype: int64
```

For unique index items, the default return-type is a scalar:

```
In [119]: obj['c']

Out[119]: 4
```

The same goes for `DataFrame` objects, although they are more complicated.

```
In [120]: df = DataFrame(np.random.randn(4, 3), index=['a', 'a', 'b', 'b'])
          df

Out[120]:           0         1         2
          a -0.267175  1.793095 -0.652929
          a -1.886837  1.059626  0.644448
          b -0.007799 -0.449204  2.448963
          b  0.667226  0.802926  0.575721
```

Remember that you have to slice the index using `ix`, and you will observe the same behavior.

```
In [121]: df.ix['b']

Out[121]:           0         1         2
          b -0.007799 -0.449204  2.448963
          b  0.667226  0.802926  0.575721
```

4 Summarizing and computing descriptive statistics

Oftentimes, you need a quick way to come up with basic summary statistics of data sets. The solution that `Pandas` provides is incredibly robust, especially with regard to `NaN` entries.

```
In [122]: 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'])
          df

Out[122]:     one  two
          a  1.40  NaN
          b  7.10 -4.5
          c   NaN  NaN
          d  0.75 -1.3
```

By default, the `sum` method will skip `NaN` entries for each column in a `DataFrame`.

```
In [123]: df.sum()

Out[123]: one    9.25
          two   -5.80
          dtype: float64
```

For the `DataFrame` object, you can also apply along either the index axis or the column axis. Again, `sum` will skip over `NaN` elements when arriving at a value.

```
In [124]: df.sum(axis=1)

Out[124]: a    1.40
          b    2.60
          c     NaN
          d   -0.55
          dtype: float64
```

If you don't want this behavior, you can always tell the statistics function you are applying not to skip the `NaN` entries. Here is an example using `mean`:

```
In [125]: df.mean(axis=1, skipna=False)

Out[125]: a      NaN
          b    1.300
          c      NaN
          d   -0.275
          dtype: float64
```

Another useful statistic is `idxmax`, which returns the index of the maximum value of a column in a `DataFrame`.

```
In [126]: df.idxmax()

Out[126]: one    b
          two    d
          dtype: object
```

One incredibly useful method is `cumsum`, which has a number of important applications in the analysis of probability distributions and random walks.

```
In [127]: df.cumsum()

Out[127]:     one  two
          a  1.40  NaN
          b  8.50 -4.5
          c   NaN  NaN
          d  9.25 -5.8
```

You can also get a quick overview of all of the summary statistics of a `DataFrame` simply by calling the `describe` method.

```
In [128]: df.describe()

Out[128]:             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
```

Method	Description
`count`	Number of non-`NaN` 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 for minimum and maximum values
`idmin`, `idmax`	Compute index values for minimum and maximum values
`quantile`	Compute sample quantile ranging from 0 to 1
`sum`	Sum of values
`mean`	Mean of values
`median`	Arithmetic median 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 min and max of values
`cumprod`	cumulative product of values
`diff`	Compute 1st arithmetic difference (useful for time series
`pct_change`	Compute percent changes

Descriptive and Summary Statistics `Series` objects also have a `describe` method. The `describe` method outputs statistics based on the `dtype` of the underlying object. In the above example, `df` had a `dtype` of `float64`, so `describe` produced information pertinent to floating-point numerics. In the below example, the `Series` object has a `dtype` of `object`, which results in different summary statistics.

```
In [129]: obj = Series(['a', 'a', 'b', 'c'] * 4)
          obj.describe()

Out[129]: count     16
          unique     3
          top        a
          freq       8
          dtype: object
```

4.1 Correlation and covariance

One common problem in data analysis, especially in the analysis of time series data like historical prices for financial securities, is correlation and covariance analysis. To this end `Pandas` has a number of features to make the analysis simple.

Here is one example, using a built-in data aggregator using Yahoo! Finance in the `Pandas` API. Returns on a stock are defined as the percent change in the stock's closing value from day-to-day.

```
In [130]: import pandas.io.data as web

          all_data = {}
          for ticker in ['AAPL', 'IBM', 'MSFT', 'CSCO']:
              all_data[ticker] = web.get_data_yahoo(ticker)

          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()})

          returns = price.pct_change()
          returns.tail()

Out[130]:                 AAPL      CSCO       IBM      MSFT
          Date
          2015-03-25 -0.026127 -0.019431 -0.023313 -0.033566
          2015-03-26  0.006970 -0.013578  0.008731 -0.006030
          2015-03-27 -0.007968  0.001488 -0.001183 -0.005824
          2015-03-30  0.025314  0.019316  0.014152 -0.000244
          2015-03-31 -0.015352  0.003280 -0.013340 -0.007324
```

When given a `Series` object, the `corr` method computes the scalar correlation between the `Series` and another `Series`.

```
In [131]: returns.MSFT.corr(returns.IBM)

Out[131]: 0.50144660652462925
```

By contrast, `corr` and `cov` returns a correlation and covariance matrix `DataFrame` with filled correlation and covariance values, respectively.

```
In [132]: returns.MSFT.cov(returns.IBM)

Out[132]: 8.3470993744258389e-05
```

```
In [133]: returns.corr()
```

```
Out[133]:           AAPL      CSCO       IBM      MSFT
          AAPL  1.000000  0.335746  0.371383  0.347494
          CSCO  0.335746  1.000000  0.448426  0.464868
          IBM   0.371383  0.448426  1.000000  0.501447
          MSFT  0.347494  0.464868  0.501447  1.000000
```

```
In [134]: returns.cov()
```

```
Out[134]:           AAPL      CSCO       IBM      MSFT
          AAPL  0.000283  0.000098  0.000073  0.000083
          CSCO  0.000098  0.000301  0.000091  0.000114
          IBM   0.000073  0.000091  0.000138  0.000083
          MSFT  0.000083  0.000114  0.000083  0.000200
```

The `corrwith` method computes pairwise correlations and stores the resultant in a `Series`. Note that the correlation between IBM and IBM is 1.

```
In [135]: returns.corrwith(returns.IBM)
```

```
Out[135]: AAPL    0.371383
          CSCO    0.448426
          IBM     1.000000
          MSFT    0.501447
          dtype: float64
```

Passing a `DataFrame` instead computes correlation with like-columns.

```
In [136]: returns.corrwith(volume)
```

```
Out[136]: AAPL   -0.097904
          CSCO   -0.235580
          IBM    -0.184944
          MSFT   -0.120080
          dtype: float64
```

4.2 Unique values, value counts, and membership

Given data with repeats, you can eliminate the excess by using the `unique` method.

```
In [137]: obj = Series(['c', 'a', 'd', 'a', 'a', 'b', 'b', 'c', 'c'])
          uniques = obj.unique()
          uniques
```

```
Out[137]: array(['c', 'a', 'd', 'b'], dtype=object)
```

The `value_counts` returns a `Series` with an index made up of the unique entries in the original `Series`, and the new entries give the total appearances of each value.

```
In [138]: obj.value_counts()
```

```
Out[138]: c    3
          a    3
          b    2
          d    1
          dtype: int64
```

You can perform set-membership operations to, for example, construct masks which you can then apply to your original data.

```
In [139]: mask = obj.isin(['b', 'c']) # This forms a Series object of Boolean values
          obj[mask]

Out[139]: 0    c
          5    b
          6    b
          7    c
          8    c
          dtype: object
```

For `DataFrame` objects, you can apply the `value_counts` method to each subseries, producing a new `DataFrame` of frequency statistics.

```
In [140]: data = DataFrame({'Qu1': [1, 3, 4, 3, 4],
                            'Qu2': [2, 3, 1, 2, 3],
                            'Qu3': [1, 5, 2, 4, 4]})
          data.apply(pd.value_counts).fillna(0)

Out[140]:    Qu1  Qu2  Qu3
          1    1    1    1
          2    0    2    1
          3    2    2    0
          4    2    0    2
          5    0    0    1
```

5 Handling missing data

One of the primary problems with data analysis is the prevalence of missing data. In many cases, arithmetic operations, summary statistics, and other functions require that your data be intact in order to provide meaningful results. `Pandas` gives a number of functions to address the problem of missing data, allowing you to filter it out easily.

Consider this `Series` of `string` values.

```
In [141]: string_data = Series(['aardvark', 'artichoke', np.nan, 'avocado'])
          string_data

Out[141]: 0     aardvark
          1    artichoke
          2          NaN
          3      avocado
          dtype: object
```

The `isnull` method identifies every `NaN` entry. Alternatively,`notnull` will identify every non-`NaN` entry.

```
In [142]: string_data.isnull()

Out[142]: 0    False
          1    False
          2     True
          3    False
          dtype: bool

In [143]: string_data[0] = None
          string_data.notnull()
```

```
Out[143]: 0    False
          1     True
          2    False
          3     True
          dtype: bool
```

5.1 Filtering out missing data

A simple way to remove missing entries from a `Series` object is to use `dropna`.

```
In [144]: from numpy import nan as NA
          data = Series([1, NA, 3.5, NA, 7])
          data.dropna()

Out[144]: 0    1.0
          2    3.5
          4    7.0
          dtype: float64
```

Alternatively, you can use `Boolean Series` and `notnull` to mask the original data.

```
In [145]: data[data.notnull()]

Out[145]: 0    1.0
          2    3.5
          4    7.0
          dtype: float64
```

`DataFrame` objects are trickier. For example, how should `Pandas` handle a mostly-complete row? The correct answer is ambiguous. By default, `dropna` will eliminate *any* row with a `NaN` (we redefined `NaN` to `NA` here) value.

```
In [146]: data = DataFrame([[1., 6.5, 3.], [1., NA, NA],
                            [NA, NA, NA], [NA, 6.5, 3.]])
          cleaned = data.dropna()
          data

Out[146]:     0    1    2
          0   1  6.5    3
          1   1  NaN  NaN
          2 NaN  NaN  NaN
          3 NaN  6.5    3

In [147]: cleaned

Out[147]:    0    1  2
          0  1  6.5  3
```

Alternatively, you can require that a row be eliminated only if it is *completely* empty.

```
In [148]: data.dropna(how='all')

Out[148]:     0    1    2
          0   1  6.5    3
          1   1  NaN  NaN
          3 NaN  6.5    3
```

You can also specify columns for deletion. Again, you can change the deletion requirements as needed.

```
In [149]: data[4] = NA # fill a column entirely with NA
          data.dropna(axis=1, how='all')

Out[149]:      0    1    2
          0    1  6.5    3
          1    1  NaN  NaN
          2  NaN  NaN  NaN
          3  NaN  6.5    3
```

The `dropna` method is very robust. You can also specify a minimum threshold of data in a particular row as a criterion for deletion. In the next example, we threshold at 2 entries per row, allowing rows with one `NaN` value to stay while deleting any more patchy rows.

```
In [150]: df = DataFrame(np.random.randn(7, 3))
          df.ix[:4, 1] = NA; df.ix[:2, 2] = NA
          df

Out[150]:           0         1         2
          0  1.381918       NaN       NaN
          1 -0.206282       NaN       NaN
          2  1.811659       NaN       NaN
          3 -1.313554       NaN -0.615939
          4  0.174072       NaN -0.035408
          5 -1.194063  1.037339 -0.364987
          6 -0.215726  0.763525  0.671308

In [151]: df.dropna(thresh=2)

Out[151]:           0         1         2
          3 -1.313554       NaN -0.615939
          4  0.174072       NaN -0.035408
          5 -1.194063  1.037339 -0.364987
          6 -0.215726  0.763525  0.671308
```

5.2 Filling in missing data

Instead of eliminating missing data outright, `Pandas` lets you fill in the missing values. The simple approach, using `fillna`, is to pass a value that will then replace every `NaN` entry.

```
In [152]: df.fillna(0)

Out[152]:           0         1         2
          0  1.381918  0.000000  0.000000
          1 -0.206282  0.000000  0.000000
          2  1.811659  0.000000  0.000000
          3 -1.313554  0.000000 -0.615939
          4  0.174072  0.000000 -0.035408
          5 -1.194063  1.037339 -0.364987
          6 -0.215726  0.763525  0.671308
```

Alternatively, you can specify different fill values in different columns by giving a `dict` with keys of column names.

```
In [153]: df.fillna({1: 0.5, 2: -1})
```

```
Out[153]:           0         1         2
          0  1.381918  0.500000 -1.000000
          1 -0.206282  0.500000 -1.000000
          2  1.811659  0.500000 -1.000000
          3 -1.313554  0.500000 -0.615939
          4  0.174072  0.500000 -0.035408
          5 -1.194063  1.037339 -0.364987
          6 -0.215726  0.763525  0.671308
```

Using the `inplace` argument, you can overwrite the original `DataFrame` object.

```
In [154]: # always returns a reference to the filled object
          _ = df.fillna(0, inplace=True)
          df

Out[154]:           0         1         2
          0  1.381918  0.000000  0.000000
          1 -0.206282  0.000000  0.000000
          2  1.811659  0.000000  0.000000
          3 -1.313554  0.000000 -0.615939
          4  0.174072  0.000000 -0.035408
          5 -1.194063  1.037339 -0.364987
          6 -0.215726  0.763525  0.671308
```

The other main filling technique is to fill by procedure. `ffill` will copy the previous value in a column into the `NaN` entry.

```
In [155]: df = DataFrame(np.random.randn(6, 3))
          df.ix[2::2, 1] = NA; df.ix[4:, 2] = NA
          print df, "\n\n"
          print df.fillna(method='ffill')

0         1         2
0 -1.810961 -0.246414 -0.205597
1  0.998181  0.625001  0.410271
2  0.063753       NaN  0.289051
3 -2.202882 -0.068072  0.033761
4  1.840764       NaN       NaN
5 -0.024116  1.462648       NaN



          0         1         2
0 -1.810961 -0.246414 -0.205597
1  0.998181  0.625001  0.410271
2  0.063753  0.625001  0.289051
3 -2.202882 -0.068072  0.033761
4  1.840764 -0.068072  0.033761
5 -0.024116  1.462648  0.033761
```

In cases where you don't want this to extend indefinitely, you can limit the fill method to a certain number of `NaN` entries after the last available one.

```
In [156]: df.fillna(method='ffill', limit=1)

Out[156]:           0         1         2
          0 -1.810961 -0.246414 -0.205597
```

```
1  0.998181  0.625001  0.410271
2  0.063753  0.625001  0.289051
3 -2.202882 -0.068072  0.033761
4  1.840764 -0.068072  0.033761
5 -0.024116  1.462648       NaN
```

6 Hierarchical indexing

From "Python for Data Analysis":

> *Hierarchical indexing* is an important feature of pandas enabling you to have multiple (two or more) index *levels* on an axis. Somewhat abstractly, it provides a way for you to work with higher dimensional data in a lower dimensional form. Let's start with a simple example; create a `Series` with a list of lists or arrays as the index:

```
In [157]: data = Series(np.random.randn(10),
                        index=[['a', 'a', 'a', 'b', 'b', 'b', 'c', 'c', 'd', 'd'],
                               [1, 2, 3, 1, 2, 3, 1, 2, 2, 3]])
          data
```

```
Out[157]: a  1    0.221755
             2   -0.838210
             3    1.396553
          b  1   -1.553775
             2   -0.007680
             3    1.335753
          c  1   -1.296638
             2    1.067990
          d  2   -0.743429
             3    0.500286
          dtype: float64
```

Notice how the index `a` corresponds to the sub-indices `1` and `2`, and their corresponding data. The `index` object is thus not a simple list but a series of lists corresponding to the inner sub-indices.

```
In [158]: data.index
```

```
Out[158]: MultiIndex(levels=[[u'a', u'b', u'c', u'd'], [1, 2, 3]],
                     labels=[[0, 0, 0, 1, 1, 1, 2, 2, 3, 3], [0, 1, 2, 0, 1, 2, 0, 1, 1, 2]])
```

Accessing an outer label will give you the sub-`Series` that it corresponds to.

```
In [159]: print data['b'], "\n\n"
          print data['b':'c']
```

```
1   -1.553775
2   -0.007680
3    1.335753
dtype: float64



b  1   -1.553775
   2   -0.007680
   3    1.335753
c  1   -1.296638
   2    1.067990
dtype: float64
```

You can access sub-indices, which returns the Series of all upper indices and their corresponding values.

```
In [160]: data[:, 2]

Out[160]: a   -0.838210
          b   -0.007680
          c    1.067990
          d   -0.743429
          dtype: float64
```

You can use unstack to take the multi-index and place it into a DataFrame object.

```
In [161]: data.unstack()

Out[161]:           1         2         3
          a  0.221755 -0.838210  1.396553
          b -1.553775 -0.007680  1.335753
          c -1.296638  1.067990       NaN
          d       NaN -0.743429  0.500286
```

The inverse of unstack is stack. Observe:

```
In [162]: data.unstack().stack()

Out[162]: a  1    0.221755
             2   -0.838210
             3    1.396553
          b  1   -1.553775
             2   -0.007680
             3    1.335753
          c  1   -1.296638
             2    1.067990
          d  2   -0.743429
             3    0.500286
          dtype: float64
```

Multi-indexing has a similar logic with DataFrame objects, but it becomes more complicated as both the index and the columns can be given a hierarchy:

```
In [163]: frame = DataFrame(np.arange(12).reshape((4, 3)),
                            index=[['a', 'a', 'b', 'b'], [1, 2, 1, 2]],
                            columns=[['Ohio', 'Ohio', 'Colorado'],
                                     ['Green', 'Red', 'Green']])
          frame

Out[163]:      Ohio     Colorado
              Green Red    Green
          a 1     0   1        2
            2     3   4        5
          b 1     6   7        8
            2     9  10       11
```

For clarity, let's rename the labels so we know what level we are looking at.

```
In [164]: frame.index.names = ['key1', 'key2']
          frame.columns.names = ['state', 'color']
          frame
```

```
Out[164]: state      Ohio     Colorado
          color     Green Red    Green
          key1 key2
          a    1        0   1        2
               2        3   4        5
          b    1        6   7        8
               2        9  10       11
```

Now by specifying any column, whether on the top level or any sublevel, you can get the DataFrame of values corresponding to the name.

```
In [165]: frame['Ohio']
```

```
Out[165]: color      Green  Red
          key1 key2
          a    1         0    1
               2         3    4
          b    1         6    7
               2         9   10
```

MultiIndex objects are independent in Pandas, meaning that you can create them without a corresponding DataFrame and reuse them as needed.

```
In [166]: from pandas import MultiIndex
          MultiIndex.from_arrays([['Ohio', 'Ohio', 'Colorado'], ['Green', 'Red', 'Green']],
                                 names=['state', 'color'])
```

```
Out[166]: MultiIndex(levels=[[u'Colorado', u'Ohio'], [u'Green', u'Red']],
                     labels=[[1, 1, 0], [0, 1, 0]],
                     names=[u'state', u'color'])
```

6.1 Reordering and sorting levels

You can always swap indices on the same level. For example, if you want key2 and key1 to switch, you can write

```
In [167]: frame.swaplevel('key1', 'key2')
```

```
Out[167]: state      Ohio     Colorado
          color     Green Red    Green
          key2 key1
          1    a        0   1        2
          2    a        3   4        5
          1    b        6   7        8
          2    b        9  10       11
```

Additionally, you can sort a particular index (in general, you can't sort them all). Specify the index by its order (first is 0, second is 2), and you will see the sort take place:

```
In [168]: frame.sortlevel(1)
```

```
Out[168]: state      Ohio     Colorado
          color     Green Red    Green
          key1 key2
          a    1        0   1        2
          b    1        6   7        8
          a    2        3   4        5
          b    2        9  10       11
```

As with the object-oriented paradigm, you can combine these actions into one statement. For example:

```
In [169]: frame.swaplevel(0, 1).sortlevel(0)

Out[169]: state      Ohio     Colorado
          color     Green Red    Green
          key2 key1
          1    a        0   1        2
               b        6   7        8
          2    a        3   4        5
               b        9  10       11
```

6.2 Summary statistics by level

With hierarchical indexing, you can specify the level and axis with which to compute summary statistics. If one wants to compute the sum of all values in the `key2` index, you get the relevant sub-`DataFrame`.

```
In [170]: frame.sum(level='key2')

Out[170]: state  Ohio     Colorado
          color Green Red    Green
          key2
          1         6   8       10
          2        12  14       16
```

This of course gets extended to the columns as well, which you have grown accustomed to with `DataFrame` methods.

```
In [171]: frame.sum(level='color', axis=1)

Out[171]: color      Green  Red
          key1 key2
          a    1         2    1
               2         8    4
          b    1        14    7
               2        20   10
```

6.3 Using a DataFrame's columns

In the examples above we showed how to `stack` and `unstack` `Series` objects into `DataFrames`. But in general `DataFrame` objects give you a lot of discretion regarding which columns you want to convert into indices.

```
In [172]: frame = DataFrame({'a': range(7), 'b': range(7, 0, -1),
                             'c': ['one', 'one', 'one', 'two', 'two', 'two', 'two'],
                             'd': [0, 1, 2, 0, 1, 2, 3]})
          frame

Out[172]:    a  b    c  d
          0  0  7  one  0
          1  1  6  one  1
          2  2  5  one  2
          3  3  4  two  0
          4  4  3  two  1
          5  5  2  two  2
          6  6  1  two  3
```

You can overload set_index with more than one column to produce a hierarchical index using the values of each respective column.

```
In [173]: frame2 = frame.set_index(['c', 'd'])
          frame2

Out[173]:        a  b
          c   d
          one 0  0  7
              1  1  6
              2  2  5
          two 0  3  4
              1  4  3
              2  5  2
              3  6  1
```

Crucially, the default Pandas behavior is to remove the indexed columns. You can force Pandas to keep the old columns by specifying the drop parameter:

```
In [174]: frame.set_index(['c', 'd'], drop=False)

Out[174]:        a  b    c  d
          c   d
          one 0  0  7  one  0
              1  1  6  one  1
              2  2  5  one  2
          two 0  3  4  two  0
              1  4  3  two  1
              2  5  2  two  2
              3  6  1  two  3

In [175]: frame2.reset_index()

Out[175]:      c  d  a  b
          0  one  0  0  7
          1  one  1  1  6
          2  one  2  2  5
          3  two  0  3  4
          4  two  1  4  3
          5  two  2  5  2
          6  two  3  6  1
```