Getting Started with pandas

March 6, 2018

0.1 Introduction

pandas will be a major tool of interest throughout much of the rest of the book. It contains data structures and data manipulation tools designed to make data cleaning and analysis fast and easy in Python. pandas is often used in tandem with numerical computing tools like NumPy and SciPy, analytical libraries like statsmodels and scikit-learn, and data visualization libraries like matplotlib. pandas adopts significant parts of NumPy's idiomatic style of array-based computing, especially array-based functions and a preference for data processing without for loops. While pandas adopts many coding idioms from NumPy, the biggest difference is that **pandas is designed for working with tabular or heterogeneous data. NumPy, by contrast, is best suited for working with homogeneous numerical array data**. Since becoming an open source project in 2010, pandas has matured into a quite large library that's applicable in a broad set of real-world use cases. The developer community has grown to over 800 distinct contributors, who've been helping build the project as they've used it to solve their day-to-day data problems

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

import pandas as pd

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

from pandas import Series, DataFrame

0.2 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.

1) Series: A Series is a one-dimensional array-like object containing a sequence of values (of similar types to NumPy types) and an associated array of data labels, called its index. The default one consisting of the integers 0 through N - 1. The simplest Series is formed from only an array of data

2) You can use labels in the index when selecting single values or a set of values:

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

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

```
In [12]: import pandas as pd
         obj = pd.Series([4, 7, -5, 3])
         obj
         obj.values
         obj.index
         obj2 = pd.Series([4, 7, -5, 3], index=['d', 'b', 'a', 'c'])
         obj2
         obj2.values
         obj2.values
         obj2['a']
         obj2['d'] = 6

         obj2[['c', 'a', 'd']]
         obj2[obj2 > 0]
         obj2 * 2
         import numpy as np
         np.exp(obj2)

         'b' in obj2
```

```
Out[12]: True
```

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

```
In [14]: sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}
         obj3 = pd.Series(sdata)
         obj3
```

```
Out[14]: Ohio      35000
         Oregon    16000
         Texas     71000
         Utah       5000
         dtype: int64
```

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

```
In [16]: states = ['California', 'Ohio', 'Oregon', 'Texas']
         obj4 = pd.Series(sdata, index=states)
         obj4
```

```
Out[16]: California        NaN
         Ohio          35000.0
         Oregon        16000.0
         Texas         71000.0
         dtype: float64
```

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

I will use the terms "missing" or "NA" interchangeably to refer to missing data. The isnull and notnull functions in pandas should be used to detect missing data

```
In [20]: pd.isnull(obj4)
         pd.notnull(obj4)
         obj4.isnull()
```

```
Out[20]: California     True
         Ohio          False
         Oregon        False
         Texas         False
         dtype: bool
```

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

```
In [21]: obj3
         obj4
         obj3+obj4
```

```
Out[21]: California         NaN
         Ohio           70000.0
         Oregon         32000.0
         Texas         142000.0
         Utah               NaN
         dtype: float64
```

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

```
In [23]: obj4.name = 'population'
         obj4.index.name = 'state'
         obj4
```

```
Out[23]: state
         California        NaN
         Ohio          35000.0
         Oregon        16000.0
         Texas         71000.0
         Name: population, dtype: float64
```

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

```
In [25]: obj
         obj.index = ['Bob', 'Steve', 'Jeff', 'Ryan']
         obj
```

```
Out[25]: Bob      4
         Steve    7
         Jeff    -5
         Ryan     3
         dtype: int64
```

0.3 DataFrame

A DataFrame represents a rectangular table of data and contains an ordered collection of columns, each of which can be a different value type (numeric, string, boolean, etc.). The DataFrame has both a row and column index; it can be thought of as a dict of Series all sharing the same index. Under the hood, the data is stored as one or more two-dimensional blocks rather than a list, dict, or some other collection of one-dimensional arrays. The exact details of DataFrame's internals are outside the scope of this book.

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

```
In [28]: data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada', 'Nevada'],
         'year': [2000, 2001, 2002, 2001, 2002, 2003],
         'pop': [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]}
         frame = pd.DataFrame(data)
```

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

```
In [29]: frame.head()

Out[29]:    pop   state  year
         0  1.5    Ohio  2000
         1  1.7    Ohio  2001
         2  3.6    Ohio  2002
         3  2.4  Nevada  2001
         4  2.9  Nevada  2002
```

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

```
In [30]: pd.DataFrame(data, columns=['year', 'state', 'pop'])

Out[30]:    year   state  pop
         0  2000    Ohio  1.5
         1  2001    Ohio  1.7
         2  2002    Ohio  3.6
         3  2001  Nevada  2.4
         4  2002  Nevada  2.9
         5  2003  Nevada  3.2
```

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

```
In [32]: frame2 = pd.DataFrame(data, columns=['year', 'state', 'pop', 'debt'],
             index=['one', 'two', 'three', 'four',
          'five', 'six'])
         frame2
```

```
Out[32]:        year   state  pop debt
         one    2000    Ohio  1.5  NaN
         two    2001    Ohio  1.7  NaN
         three  2002    Ohio  3.6  NaN
         four   2001  Nevada  2.4  NaN
         five   2002  Nevada  2.9  NaN
         six    2003  Nevada  3.2  NaN
```

A column in a DataFrame can be retrieved as a Series either by dict-like notation or by attribute. Attribute-like access (e.g., frame2.year) and tab completion of column names in IPython is provided as a convenience. frame2[column] works for any column name, but frame2.column only works when the column name is a valid Python variable name.

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

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

```
Out[33]: one      2000
         two      2001
         three    2002
         four     2001
         five     2002
         six      2003
         Name: year, dtype: int64
```

Rows can also be retrieved by position or name with the special loc attribute (much more on this later):

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

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

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

```
In [36]: frame2['debt'] = 16.5
         frame2
         frame2['debt'] = np.arange(6.)
         frame2
```

```
Out[36]:         year   state  pop  debt
         one     2000    Ohio  1.5   0.0
         two     2001    Ohio  1.7   1.0
         three   2002    Ohio  3.6   2.0
         four    2001  Nevada  2.4   3.0
         five    2002  Nevada  2.9   4.0
         six     2003  Nevada  3.2   5.0
```

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

```
In [37]: val = pd.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
         six     2003  Nevada  3.2   NaN
```

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

```
In [38]: val2 = pd.Series([1, 2, 3, 5], index=['two', 'four', 'five', 'six'])
         frame2['new'] = val2
         frame2
```

```
Out[38]:         year   state  pop  debt  new
         one     2000    Ohio  1.5   NaN  NaN
         two     2001    Ohio  1.7  -1.2  1.0
         three   2002    Ohio  3.6   NaN  NaN
         four    2001  Nevada  2.4  -1.5  2.0
         five    2002  Nevada  2.9  -1.7  3.0
         six     2003  Nevada  3.2   NaN  5.0
```

```
In [40]: frame2['eastern'] = frame2.state == 'Ohio'
         frame2
         frame2.columns
         del frame2['eastern']
         frame2.columns
```

```
Out[40]: Index(['year', 'state', 'pop', 'debt', 'new'], dtype='object')
```

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

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

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

```
In [41]: pop = {'Nevada': {2001: 2.4, 2002: 2.9},
           'Ohio': {2000: 1.5, 2001: 1.7, 2002: 3.6}}
         frame3 = pd.DataFrame(pop)
         frame3
```

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

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

```
In [42]: frame3.T
```

```
Out[42]:         2000  2001  2002
         Nevada   NaN   2.4   2.9
         Ohio     1.5   1.7   3.6
```

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

```
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
```

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

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

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

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

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

```
Out[46]: array([[2000, 'Ohio', 1.5, nan, nan],
                [2001, 'Ohio', 1.7, -1.2, 1.0],
                [2002, 'Ohio', 3.6, nan, nan],
                [2001, 'Nevada', 2.4, -1.5, 2.0],
                [2002, 'Nevada', 2.9, -1.7, 3.0],
                [2003, 'Nevada', 3.2, nan, 5.0]], dtype=object)
```

0.4 Index Objects

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

```
In [47]: obj = pd.Series(range(3), index=['a', 'b', 'c'])
         index = obj.index
         index[1:]
```

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

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

```
In [48]: index[1] = 'd' # TypeError
```

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

    TypeError                                 Traceback (most recent call last)

    <ipython-input-48-d429372094ee> in <module>()
----> 1 index[1] = 'd' # TypeError


    ~\AppData\Local\Continuum\anaconda3\lib\site-packages\pandas\core\indexes\base.py in __s
   1668
   1669     def __setitem__(self, key, value):
-> 1670         raise TypeError("Index does not support mutable operations")
   1671
   1672     def __getitem__(self, key):


    TypeError: Index does not support mutable operations
```

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

```
In [51]: labels = pd.Index(np.arange(3))
         labels
         obj2 = pd.Series([1.5, -2.5, 0], index=labels)
         obj2
         obj2.index is labels

         frame3.columns
         'Ohio' in frame3.columns
         2003 in frame3.index
         frame3
```

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

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

```
In [52]: dup_labels = pd.Index(['foo', 'foo', 'bar', 'bar'])
```

0.5 Essential Functionality

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

Reindexing An important method on pandas objects is reindex, which means to create a new object with the data conformed to a new index

```
In [53]: obj = pd.Series([4.5, 7.2, -5.3, 3.6], index=['d', 'b', 'a', 'c'])
         obj
         obj2 = obj.reindex(['a', 'b', 'c', 'd', 'e'])
         obj2

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

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

```
In [54]: obj3 = pd.Series(['blue', 'purple', 'yellow'], index=[0, 2, 4])
         obj3
         obj3.reindex(range(6), method='ffill')

Out[54]: 0      blue
         1      blue
         2    purple
         3    purple
         4    yellow
         5    yellow
         dtype: object
```

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

```
In [55]: frame = pd.DataFrame(np.arange(9).reshape((3, 3)),
             index=['a', 'c', 'd'],
             columns=['Ohio', 'Texas', 'California'])
         frame
```

```
Out[55]:    Ohio  Texas  California
         a     0      1           2
         c     3      4           5
         d     6      7           8
```

```
In [59]: frame2 = frame.reindex(['a', 'b', 'c', 'd'])
         frame2
         states = ['Texas', 'Utah', 'California']
         #The columns can be reindexed with the columns keyword
         frame.reindex(columns=states)
         frame.loc[['a', 'b', 'c', 'd'], states]
         frame
```

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

0.6 Dropping Entries from an Axis

Dropping one or more entries from an axis is easy if you already have an index array or list without those entries. As that can require a bit of munging and set logic, the drop method will return a new object with the indicated value or values deleted from an axis

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

```
Out[60]: a    0.0
         b    1.0
         d    3.0
         e    4.0
         dtype: float64
```

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

```
Out[61]: a    0.0
         b    1.0
         e    4.0
         dtype: float64
```

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

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

2) You can drop values from the columns by passing axis=1 or axis='columns'

```
In [65]: data = pd.DataFrame(np.arange(16).reshape((4, 4)),
             index=['Ohio', 'Colorado', 'Utah', 'New York'],
             columns=['one', 'two', 'three', 'four'])
         data

         data.drop(['Colorado', 'Ohio'])
         data.drop('two', axis=1)

         data.drop(['two', 'four'], axis='columns')
```

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

0.7 Indexing, Selection, and Filtering

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

```
In [68]: obj = pd.Series(np.arange(4.), index=['a', 'b', 'c', 'd'])
         obj
         obj['b']
         obj[1]
         obj[2:4]
         obj[['b', 'a', 'd']]
         obj[[1, 3]]
         obj[obj < 2]
```

```
Out[68]: a    0.0
         b    1.0
         dtype: float64
```

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

```
In [72]: obj['b':'c']
         obj['b':'c'] = 5
         data
```

```
Out[72]:           one  two  three  four
         Ohio        0    1      2     3
         Colorado    4    5      6     7
```

```
In [73]: data = pd.DataFrame(np.arange(16).reshape((4, 4)),
             index=['Ohio', 'Colorado', 'Utah', 'New York'],
             columns=['one', 'two', 'three', 'four'])
         data['two']
         data[['three', 'one']]
         data[:2]
         data[data['three'] > 5]
```

```
Out[73]:            one  two  three  four
         Colorado    4    5      6     7
         Utah        8    9     10    11
         New York   12   13     14    15
```

The row selection syntax data[:2] is provided as a convenience. Passing a single element or a list to the [] operator selects columns. Another use case is in indexing with a boolean DataFrame, such as one produced by a scalar comparison:

```
In [74]: data < 5
         data[data < 5] = 0
```

```
Out[74]:              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
```

0.8 Selection with loc and iloc

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

```
In [77]: data.loc['Colorado', ['two', 'three']]
         data.iloc[2, [3, 0, 1]]
         data.iloc[2]
         data.iloc[[1, 2], [3, 0, 1]]
         data.loc[:'Utah', 'two']
         data.iloc[:, :3][data.three > 5]
```

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