Getting Started with pandas

Part 1

- You may use the following import convention for pandas:

```
In [1]: import pandas as pd
```

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

```
In [2]: from pandas import Series, DataFrame
```

Introduction to pandas Data Structures

Part 1

- To get started with pandas, you will need to get comfortable with its two workhorse data structures: *Series* and *DataFrame*.
- While they are not a universal solution for every problem, they provide a solid, easy-to-use basis for most applications.

Series

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

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

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

- The string representation of a Series displayed interactively shows the index on the left and the values on the right.
- Since we did not specify an index for the data, a default one consisting of the integers `0` through `N - 1` (where `N` is the length of the data) is created.

- You can get the array representation and index object of the Series via its `values` and `index` attributes, respectively:

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

In [5]: obj.index  # like range(4)
Out[5]: RangeIndex(start=0, stop=4, step=1)
```

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

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

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

In [8]: obj2.index
Out[8]: Index(['d', 'b', 'a', 'c'], dtype='object')
```

- Compared with NumPy arrays, you can use labels in the index when selecting single values or a set of values:

```
In [9]: obj2['a']
Out[9]: -5

In [10]: obj2['d'] = 6

In [11]: obj2[['c', 'a', 'd']]
Out[11]: c    3
         a   -5
         d    6
         dtype: int64
```

- Here `['c', 'a', 'd']` is interpreted as a list of indices, even though it contains strings instead of integers.

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

```
In [16]: obj2[obj2 > 0]
Out[16]: d    6
         b    7
         c    3
         dtype: int64

In [17]: obj2 * 2
Out[17]: d    12
         b    14
         a   -10
         c     6
         dtype: int64

In [18]: np.exp(obj2)
Out[18]: d     403.428793
         b    1096.633158
         a       0.006738
         c      20.085537
         dtype: float64
```

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

```
In [19]: 'b' in obj2
Out[19]: True

In [20]: 'e' in obj2
Out[20]: False
```

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

```
In [21]: sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}
         obj3 = pd.Series(sdata)
         obj3

Out[21]: Ohio      35000
         Texas     71000
         Oregon    16000
         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. (Changed in version `0.23.0`: If data is a dict, argument order is maintained for Python `3.6` and later.)
- You can override this by passing the dict keys in the order you want them to appear in the resulting Series:

```
In [23]: sdata

Out[23]: {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}

In [24]: states = ['California', 'Ohio', 'Oregon', 'Texas']
         obj4 = pd.Series(sdata, index=states)
         obj4

Out[24]: 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 [25]: pd.isnull(obj4)
Out[25]: California     True
         Ohio          False
         Oregon        False
         Texas         False
         dtype: bool

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

- Series also has these as instance methods:

```
In [27]: obj4.isnull()

Out[27]: 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 [28]: obj3

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

In [29]: obj4

Out[29]: California        NaN
         Ohio          35000.0
         Oregon        16000.0
         Texas         71000.0
         dtype: float64

In [30]: obj3 + obj4

Out[30]: 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 [31]: obj4.name = 'population'
         obj4.index.name = 'state'
         obj4

Out[31]: 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 [32]: obj
Out[32]: 0    4
         1    7
         2   -5
         3    3
         dtype: int64

In [33]: obj.index = ['Bob', 'Steve', 'Jeff', 'Ryan']
         obj
Out[33]: Bob      4
         Steve    7
         Jeff    -5
         Ryan     3
         dtype: int64
```