Pandas

Pandas provides high-level data structures and functions designed to make working with structured or tabular data fast, easy, and expressive. Since its emergence in 2010, it has helped enable Python to be a powerful and productive data analysis environment. The primary objects in pandas that will be used in this book are the DataFrame, a tabular, column-oriented data structure with both row and column labels, and the Series, a one-dimensional labeled array object.

Pandas blends the high-performance, array-computing ideas of NumPy with the flex- ible data manipulation capabilities of spreadsheets and relational databases (such as SQL). It provides sophisticated indexing functionality to make it easy to reshape, slice and dice, perform aggregations, and select subsets of data.

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.

Introduction to pandas Data Structures

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

Series

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

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

In [2]: obj = pd.Series([4, 7, -5, 3])

In [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], dtype=int64)

In [5]: obj.index

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 [12]: obj2[obj2 > 0]
Out[12]: d    6
         b    7
         c    3
         dtype: int64

In [13]: obj2 * 2
Out[13]: d    12
         b    14
         a   -10
         c     6
         dtype: int64

In [15]: import numpy as np
         np.exp(obj2)
Out[15]: 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 map- ping of index values to data values. It can be used in many contexts where you might use a dict:

```
In [16]: 'b' in obj2
Out[16]: True
```

```
In [17]: 'e' in obj2
Out[17]: False
```

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

```
In [18]: sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}

In [19]: obj3 = pd.Series(sdata)

In [20]: obj3

Out[20]: 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. You can override this by passing the dict keys in the order you want them to appear in the resulting Series:

```
In [21]: states = ['California', 'Ohio', 'Oregon', 'Texas']

In [22]: obj4 = pd.Series(sdata, index=states)

In [23]: obj4

Out[23]: 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.

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

In [25]:
```
pd.notnull(obj4)
```
Out[25]:
```
California    False
Ohio           True
Oregon         True
Texas          True
dtype: bool
```

Series also has these as instance methods:

In [26]:
```
obj4.isnull()
```
Out[26]:
```
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 [27]:
```
obj3
```
Out[27]:
```
Ohio      35000
Texas     71000
Oregon    16000
Utah       5000
dtype: int64
```

In [28]:
```
obj4
```
Out[28]:
```
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
dtype: float64
```

In [29]:
```
obj3 + obj4
```
Out[29]:
```
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 [30]:
```
obj4.name = 'population'
```
In [31]:
```
obj4.index.name = 'state'
```
In [32]:
```
obj4
```
Out[32]:
```
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 [33]: obj
Out[33]: 0    4
         1    7
         2   -5
         3    3
         dtype: int64
```

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

```
In [36]: obj
Out[36]: Bob      4
         Steve    7
         Jeff    -5
         Ryan     3
         dtype: int64
```

DataFrame

A DataFrame represents a rectangular table of data and contains an ordered collec- tion 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.

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 [35]: 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 [36]: frame
```

Out[36]:

	state	year	pop
0	Ohio	2000	1.5
1	Ohio	2001	1.7
2	Ohio	2002	3.6
3	Nevada	2001	2.4
4	Nevada	2002	2.9
5	Nevada	2003	3.2

For large DataFrames, the head method selects only the first five rows:

```
In [37]: frame.head()
```

Out[37]:

	state	year	pop
0	Ohio	2000	1.5
1	Ohio	2001	1.7
2	Ohio	2002	3.6
3	Nevada	2001	2.4
4	Nevada	2002	2.9

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

```
In [38]: pd.DataFrame(data, columns=['year', 'state', 'pop'])
```

Out[38]:

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

```
In [40]: frame2
```

Out[40]:

	year	state	pop	debt
one	2000	Ohio	1.5	NaN
two	2001	Ohio	1.7	NaN
three	2002	Ohio	3.6	NaN
four	2001	Nevada	2.4	NaN
five	2002	Nevada	2.9	NaN
six	2003	Nevada	3.2	NaN

```
In [41]: frame2.columns

Out[41]: Index(['year', 'state', 'pop', 'debt'], dtype='object')
```

A column in a DataFrame can be retrieved as a Series either by dict-like notation or by attribute:

```
In [42]: frame2['state']

Out[42]: one        Ohio
         two        Ohio
         three      Ohio
         four     Nevada
         five     Nevada
         six      Nevada
         Name: state, dtype: object
```

```
In [43]: frame2.year

Out[43]: one      2000
         two      2001
         three    2002
         four     2001
         five     2002
         six      2003
         Name: year, dtype: int64
```

Note that the returned Series have the same index as the DataFrame, and their name attribute has been appropriately set. Rows can also be retrieved by position or name with the special loc attribute.

```
In [44]: frame2.loc['three']
Out[44]: 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 [45]: frame2['debt'] = 16.5
```

```
In [46]: frame2
Out[46]:
```

	year	state	pop	debt
one	2000	Ohio	1.5	16.5
two	2001	Ohio	1.7	16.5
three	2002	Ohio	3.6	16.5
four	2001	Nevada	2.4	16.5
five	2002	Nevada	2.9	16.5
six	2003	Nevada	3.2	16.5

```
In [47]: frame2['debt'] = np.arange(6.)
```

```
In [48]: frame2
Out[48]:
```

	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 [49]: val = pd.Series([-1.2, -1.5, -1.7], index=['two', 'four', 'five'])

In [50]: frame2['debt'] = val

In [51]: frame2
```

Out[51]:

	year	state	pop	debt
one	2000	Ohio	1.5	NaN
two	2001	Ohio	1.7	-1.2
three	2002	Ohio	3.6	NaN
four	2001	Nevada	2.4	-1.5
five	2002	Nevada	2.9	-1.7
six	2003	Nevada	3.2	NaN

Assigning a column that doesn't exist will create a new column. The del keyword will delete columns as with a dict. As an example of del, I first add a new column of boolean values where the state column equals 'Ohio':

```
In [52]: frame2['eastern'] = frame2.state == 'Ohio'

In [53]: frame2
```

Out[53]:

	year	state	pop	debt	eastern
one	2000	Ohio	1.5	NaN	True
two	2001	Ohio	1.7	-1.2	True
three	2002	Ohio	3.6	NaN	True
four	2001	Nevada	2.4	-1.5	False
five	2002	Nevada	2.9	-1.7	False
six	2003	Nevada	3.2	NaN	False

Note: New columns cannot be created with the frame2.eastern syntax.

The del method can then be used to remove this column:

```
In [54]: del frame2['eastern']

In [55]: frame2.columns
Out[55]: Index(['year', 'state', 'pop', 'debt'], dtype='object')
```

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

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

```
In [56]: pop = {'Nevada': {2001: 2.4, 2002: 2.9},'Ohio': {2000: 1.5, 2001: 1.7, 2002: 3.6}}
```

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

```
In [57]: frame3 = pd.DataFrame(pop)
```

```
In [58]: frame3
```

Out[58]:

	Nevada	Ohio
2001	2.4	1.7
2002	2.9	3.6
2000	NaN	1.5

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

```
In [59]: frame3.T
```

Out[59]:

	2001	2002	2000
Nevada	2.4	2.9	NaN
Ohio	1.7	3.6	1.5

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

```
In [60]: pd.DataFrame(pop, index=[2001, 2002, 2003])
```

Out[60]:

	Nevada	Ohio
2001	2.4	1.7
2002	2.9	3.6
2003	NaN	NaN

Dicts of Series are treated in much the same way:

In [61]:
```
pdata = {'Ohio': frame3['Ohio'][:-1],'Nevada': frame3['Nevada'][:2]}
```

In [62]:
```
pd.DataFrame(pdata)
```

Out[62]:

	Ohio	Nevada
2001	1.7	2.4
2002	3.6	2.9

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

In [63]:
```
frame3.index.name = 'year'; frame3.columns.name = 'state'
```

In [64]:
```
frame3
```

Out[64]:

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

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

In [65]:
```
frame3.values
```

Out[65]:
```
array([[2.4, 1.7],
       [2.9, 3.6],
       [nan, 1.5]])
```

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

In [66]:
```
frame2.values
```

Out[66]:
```
array([[2000, 'Ohio', 1.5, nan],
       [2001, 'Ohio', 1.7, -1.2],
       [2002, 'Ohio', 3.6, nan],
       [2001, 'Nevada', 2.4, -1.5],
       [2002, 'Nevada', 2.9, -1.7],
       [2003, 'Nevada', 3.2, nan]], dtype=object)
```

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 [67]: obj = pd.Series(range(3), index=['a', 'b', 'c'])

In [68]: index = obj.index

In [69]: index
Out[69]: Index(['a', 'b', 'c'], dtype='object')

In [70]: index[1:]
Out[70]: Index(['b', 'c'], dtype='object')
```

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

Learnvista Pvt Ltd.
2nd Floor, 147, 5th Main Rd, Rajiv Gandhi Nagar HSR Sector 7,Near Salarpuria Serenity, Bengaluru, Karnataka 560102
Mob:- +91 779568798, Email:- contacts@learnbay.co

Essential Functionality

Reindexing:

An important method on pandas objects is reindex, which means to create a new object with the data conformed to a new index. Consider an example:

```
In [71]: obj = pd.Series([4.5, 7.2, -5.3, 3.6], index=['d', 'b', 'a', 'c'])

In [72]: obj

Out[72]: d    4.5
         b    7.2
         a   -5.3
         c    3.6
         dtype: float64
```

Calling reindex on this Series rearranges the data according to the new index, intro- ducing missing values if any index values were not already present:

```
In [73]: obj2 = obj.reindex(['a', 'b', 'c', 'd', 'e'])

In [74]: obj2

Out[74]: a   -5.3
         b    7.2
         c    3.6
         d    4.5
         e    NaN
         dtype: float64
```

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

```
In [75]: obj3 = pd.Series(['blue', 'purple', 'yellow'], index=[0, 2, 4])

In [76]: obj3

Out[76]: 0      blue
         2    purple
         4    yellow
         dtype: object
```

```
In [77]: obj3.reindex(range(6), method='ffill')
```

```
Out[77]: 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 [78]: frame = pd.DataFrame(np.arange(9).reshape((3, 3)),index=['a', 'c', 'd'],columns=['Ohio', 'Texas', 'California'])
```

```
In [79]: frame
```

Out[79]:

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

```
In [80]: frame2 = frame.reindex(['a', 'b', 'c', 'd'])
```

```
In [81]: frame2
```

Out[81]:

	Ohio	Texas	California
a	0.0	1.0	2.0
b	NaN	NaN	NaN
c	3.0	4.0	5.0
d	6.0	7.0	8.0

The columns can be reindexed with the columns keyword:

```
In [82]: states = ['Texas', 'Utah', 'California']
```

```
In [83]: frame.reindex(columns=states)
```

Out[83]:

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

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 [84]: obj = pd.Series(np.arange(5.), index=['a', 'b', 'c', 'd', 'e'])

In [85]: obj

Out[85]: a    0.0
         b    1.0
         c    2.0
         d    3.0
         e    4.0
         dtype: float64

In [86]: new_obj = obj.drop('c')

In [87]: new_obj

Out[87]: a    0.0
         b    1.0
         d    3.0
         e    4.0
         dtype: float64

In [88]: obj.drop(['d', 'c'])

Out[88]: a    0.0
         b    1.0
         e    4.0
         dtype: float64
```

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

```
In [91]: data = pd.DataFrame(np.arange(16).reshape((4, 4)),index=['Ohio', 'Colorado', 'Utah', 'New York'],
                             columns=['one', 'two', 'three', 'four'])

In [92]: data

Out[92]:
```

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

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

```
In [93]: data.drop(['Colorado', 'Ohio'])
```

Out[93]:

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

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

```
In [94]: data.drop('two', axis=1)
```

Out[94]:

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

```
In [95]: data.drop(['two', 'four'], axis='columns')
```

Out[95]:

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

Many functions, like drop, which modify the size or shape of a Series or DataFrame, can manipulate an object in-place without returning a new object:

```
In [96]: obj.drop('c', inplace=True)

In [97]: obj

Out[97]: a    0.0
         b    1.0
         d    3.0
         e    4.0
         dtype: float64
```

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 [98]: obj = pd.Series(np.arange(4.), index=['a', 'b', 'c', 'd'])

In [99]: obj

Out[99]: a    0.0
         b    1.0
         c    2.0
         d    3.0
         dtype: float64

In [100]: obj['b']

Out[100]: 1.0

In [101]: obj[2:4]

Out[101]: c    2.0
          d    3.0
          dtype: float64

In [102]: obj[['b', 'a', 'd']]

Out[102]: b    1.0
          a    0.0
          d    3.0
          dtype: float64
```

Slicing with labels behaves differently than normal Python slicing in that the end- point is inclusive:

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

Out[103]: b    1.0
          c    2.0
          dtype: float64
```

Setting using these methods modifies the corresponding section of the Series:

```
In [104]: obj['b':'c'] = 5

In [105]: obj

Out[105]: a    0.0
          b    5.0
          c    5.0
          d    3.0
          dtype: float64
```

Integer Indexes:

Working with pandas objects indexed by integers is something that often trips up new users due to some differences with indexing semantics on built-in Python data structures like lists and tuples. For example, you might not expect the following code to generate an error:

```
ser = pd.Series(np.arange(3.))
ser
ser[-1]
```

In this case, pandas could “fall back” on integer indexing, but it’s difficult to do this in general without introducing subtle bugs. Here we have an index containing 0, 1, 2, but inferring what the user wants (label-based indexing or position-based) is difficult:

```
In [107]: ser

Out[107]: 0    0.0
          1    1.0
          2    2.0
          dtype: float64
```

On the other hand, with a non-integer index, there is no potential for ambiguity:

```
In [108]: ser2 = pd.Series(np.arange(3.), index=['a', 'b', 'c'])
          ser2[-1]

Out[108]: 2.0
```

To keep things consistent, if you have an axis index containing integers, data selection will always be label-oriented. For more precise handling, use loc (for labels) or iloc (for integers):

```
In [109]: ser[:1]

Out[109]: 0    0.0
          dtype: float64

In [110]: ser.loc[:1]

Out[110]: 0    0.0
          1    1.0
          dtype: float64

In [111]: ser.iloc[:1]

Out[111]: 0    0.0
          dtype: float64
```

Flexible arithmetic methods:

Method	Description

add, radd	Methods for addition (+)
sub, rsub	Methods for subtraction (-)
div, rdiv	Methods for division (/)
floordiv, rfloordiv	Methods for floor division (//)
mul, rmul	Methods for multiplication (*)
pow, rpow	Methods for exponentiation (**)

Function Application and Mapping:

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

```
In [112]: frame = pd.DataFrame(np.random.randn(4, 3), columns=list('bde'),index=['Utah', 'Ohio', 'Texas', 'Oregon'])

In [113]: frame
Out[113]:
```

	b	d	e
Utah	-1.518081	1.283954	-0.827006
Ohio	0.381688	-1.676275	1.311967
Texas	0.915429	-0.552984	0.194851
Oregon	0.901716	-0.407801	-1.448699

```
In [114]: np.abs(frame)
Out[114]:
```

	b	d	e
Utah	1.518081	1.283954	0.827006
Ohio	0.381688	1.676275	1.311967
Texas	0.915429	0.552984	0.194851
Oregon	0.901716	0.407801	1.448699

Another frequent operation is applying a function on one-dimensional arrays to each column or row. DataFrame's apply method does exactly this:

```
In [115]: f = lambda x: x.max() - x.min()

In [116]: frame.apply(f)
Out[116]: b    2.433511
          d    2.960228
          e    2.760666
          dtype: float64
```

Sorting and Ranking:

Sorting a dataset by some criterion is another important built-in operation. To sort lexicographically by row or column index, use the sort_index method, which returns a new, sorted object:

```
In [117]: obj = pd.Series(range(4), index=['d', 'a', 'b', 'c'])
          obj.sort_index()

Out[117]: a    1
          b    2
          c    3
          d    0
          dtype: int64
```

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

```
In [118]: frame = pd.DataFrame(np.arange(8).reshape((2, 4)),index=['three', 'one'],
                               columns=['d', 'a', 'b', 'c'])
          frame.sort_index()
```

Out[118]:

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

```
In [119]: frame.sort_index(axis=1)
```

Out[119]:

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

The data is sorted in ascending order by default, but can be sorted in descending order, too:

```
In [120]: frame.sort_index(axis=1, ascending=False)
```

Out[120]:

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

To sort a Series by its values, use its sort_values method:

```
In [121]: obj = pd.Series([4, 7, -3, 2])

In [122]: obj.sort_values()

Out[122]: 2   -3
          3    2
          0    4
          1    7
          dtype: int64
```

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

```
In [123]: obj = pd.Series([4, np.nan, 7, np.nan, -3, 2])

In [124]: obj.sort_values()

Out[124]: 4   -3.0
          5    2.0
          0    4.0
          2    7.0
          1    NaN
          3    NaN
          dtype: float64
```

Ranking assigns ranks from one through the number of valid data points in an array. The rank methods for Series and DataFrame are the place to look; by default rank breaks ties by assigning each group the mean rank:

```
In [125]: obj = pd.Series([7, -5, 7, 4, 2, 0, 4])
          obj.rank()

Out[125]: 0    6.5
          1    1.0
          2    6.5
          3    4.5
          4    3.0
          5    2.0
          6    4.5
          dtype: float64
```

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

```
In [126]: obj.rank(method='first')

Out[126]: 0    6.0
          1    1.0
          2    7.0
          3    4.0
          4    3.0
          5    2.0
          6    5.0
          dtype: float64
```

Here, instead of using the average rank 6.5 for the entries 0 and 2, they instead have been set to 6 and 7 because label 0 precedes label 2 in the data. You can rank in descending order, too:

```
In [127]: obj.rank(ascending=False, method='max')

Out[127]: 0    2.0
          1    7.0
          2    2.0
          3    4.0
          4    5.0
          5    6.0
          6    4.0
          dtype: float64
```

Method	Description
'average'	Default: assign the average rank to each entry in the equal group
'min'	Use the minimum rank for the whole group
'max'	Use the maximum rank for the whole group
'first'	Assign ranks in the order the values appear in the data
'dense'	Like method='min', but ranks always increase by 1 in between groups rather than the number of equal elements in a group

Axis Indexes with Duplicate Labels:

Up until now all of the examples we've looked at have had unique axis labels (index values). While many pandas functions (like reindex) require that the labels be unique, it's not mandatory. Let's consider a small Series with duplicate indices:

```
In [128]: obj = pd.Series(range(5), index=['a', 'a', 'b', 'b', 'c'])
          obj

Out[128]: a    0
          a    1
          b    2
          b    3
          c    4
          dtype: int64
```

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

```
In [129]: obj.index.is_unique

Out[129]: False
```

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

```
In [130]: obj['a']

Out[130]: a    0
          a    1
          dtype: int64
```

This can make your code more complicated, as the output type from indexing can vary based on whether a label is repeated or not. The same logic extends to indexing rows in a DataFrame:

```
In [131]: df = pd.DataFrame(np.random.randn(4, 3), index=['a', 'a', 'b', 'b'])

In [132]: df

Out[132]:
```

	0	1	2
a	0.163226	-0.367548	1.409098
a	1.033743	-0.695531	0.396414
b	-0.033969	1.305551	-0.896194
b	-0.999036	0.112585	-0.814612

In this article we have learnt about pandas,series,data frames,index objects,essential functionalities,reindexing,selection,filtering,sorting, ranking etc.

Why only talk about knowledge when you can work it? Join our program to get trained and work on live industry projects and get certified by start-ups. Use your skills on projects and increase your practical knowledge.