pandas

- pandas 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.

We use the following import convention for pandas

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

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
```

In [3]:
```
import numpy as np
```

Introduction to pandas Data Structures

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

Out[4]:
```
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.

In [5]:
```
obj.values
```

Out[5]:
```
array([ 4,  7, -5,  3], dtype=int64)
```

In [6]:
```
obj.index  # like range(4)
```

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

```
Out[14]: d     403.428793
         b    1096.633158
         a       0.006738
         c      20.085537
         dtype: float64
```

A Series can be thought of 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 [15]: 'b' in obj2
```

```
Out[15]: True
```

In [16]:
```
'e' in obj2
```
Out[16]:
```
False
```

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

In [17]:
```
sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}
```
In [18]:
```
obj3 = pd.Series(sdata)
```
In [19]:
```
obj3
```
Out[19]:
```
Ohio      35000
Texas     71000
Oregon    16000
Utah       5000
dtype: int64
```

You can override the order by passing the dict keys in the order you want them to appear in the resulting Series:

In [20]:
```
states = ['California', 'Ohio', 'Oregon', 'Texas']
```
In [21]:
```
obj4 = pd.Series(sdata, index=states)
```
In [22]:
```
obj4
```
Out[22]:
```
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
dtype: float64
```

- 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.

The `isnull` and `notnull` functions in pandas can be used to detect missing data:

In [23]:
```
pd.isnull(obj4)
```
Out[23]:
```
California     True
Ohio          False
Oregon        False
Texas         False
dtype: bool
```
In [24]:
```
pd.notnull(obj4)
```
Out[24]:
```
California    False
Ohio           True
Oregon         True
Texas          True
dtype: bool
```

Series also has these as instance methods:

In [25]:
```
obj4.isnull()
```
Out[25]:
```
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 [26]: obj3
```

```
Out[26]: Ohio      35000
         Texas     71000
         Oregon    16000
         Utah       5000
         dtype: int64
```

```
In [27]: obj4
```

```
Out[27]: California        NaN
         Ohio          35000.0
         Oregon        16000.0
         Texas         71000.0
         dtype: float64
```

```
In [28]: obj3 + obj4
```

```
Out[28]: California         NaN
         Ohio           70000.0
         Oregon         32000.0
         Texas         142000.0
         Utah               NaN
         dtype: float64
```

Data alignment features can be as a join operation of database.

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

```
In [29]: obj4.name = 'population'
```

```
In [30]: obj4.index.name = 'state'
```

```
In [31]: 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']
```

```
In [34]: obj
```

```
Out[34]: Bob      4
         Steve    7
         Jeff    -5
         Ryan     3
         dtype: int64
```

DataFrame

- A DataFrame represents a **rectangular table of data**.
- Each of the columns 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.
  - While a DataFrame is physically two-dimensional, you can use it to represent higher dimensional data in a tabular format using **hierarchical indexing** (advanced feature of pandas).

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:

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'])
frame2
```
Out[39]:

	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:

In [40]:
```
frame2.columns
```
Out[40]:
```
Index(['year', 'state', 'pop', 'debt'], dtype='object')
```

In [41]:
```
frame2['state']
```
Out[41]:
```
one        Ohio
two        Ohio
three      Ohio
four     Nevada
five     Nevada
six      Nevada
Name: state, dtype: object
```

- Attribute-like access, such as, `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.

In [42]:
```
frame2.year
```
Out[42]:
```
one      2000
two      2001
three    2002
four     2001
five     2002
six      2003
Name: year, dtype: int64
```

In [43]:
```
frame2
```

Out[43]:

	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

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
frame2
```

Out[45]:

	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 [46]:
```
frame2['debt'] = np.arange(6.)
frame2
```

Out[46]:

	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 (i.e., NaN) in any holes:

In [47]:
```
val = pd.Series([-1.2, -1.5, -1.7], index=['two', 'four', 'five'])
val
```

Out[47]:
```
two     -1.2
four    -1.5
five    -1.7
dtype: float64
```

In [48]:
```
frame2['debt'] = val
```

In [49]:
```
frame2
```

Out[49]:

	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.

To demonestrate `del`, let us first add a new column of boolean values where the state column equals 'Ohio':

In [50]:
```
frame2['eastern'] = frame2.state == 'Ohio'
```

In [51]:
```
frame2
```

Out[51]:

	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

In [52]:
```
del frame2['eastern']
```

In [53]:
```
frame2.columns
```

Out[53]:
```
Index(['year', 'state', 'pop', 'debt'], dtype='object')
```

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

Out[54]:

	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

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 [55]:
```
pop = {'Nevada': {2001: 2.4, 2002: 2.9},
       'Ohio': {2000: 1.5, 2001: 1.7, 2002: 3.6}}
```

In [56]:
```
frame3 = pd.DataFrame(pop)
frame3
```

Out[56]:

	Nevada	Ohio
2001	2.4	1.7
2002	2.9	3.6
2000	NaN	1.5

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.

You can transpose the DataFrame using `.T` :

In [57]:
```
frame3.T
```

Out[57]:

	2001	2002	2000
Nevada	2.4	2.9	NaN
Ohio	1.7	3.6	1.5

The keys in the inner dicts are combined.

Index can be specified explicitly:

In [58]:
```
pd.DataFrame(pop, index=[2001, 2002, 2003])
```

Out[58]:

	Nevada	Ohio
2001	2.4	1.7
2002	2.9	3.6
2003	NaN	NaN

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

Out[59]:

	Nevada	Ohio
2001	2.4	1.7
2002	2.9	3.6
2000	NaN	1.5

Dicts of Series are treated in much the same way:

In [60]:
```
pdata = {'Ohio': frame3['Ohio'][:-1],
         'Nevada': frame3['Nevada'][:2]}
pd.DataFrame(pdata)
```

Out[60]:

	Ohio	Nevada
2001	1.7	2.4
2002	3.6	2.9

For a complete list of things you can pass the DataFrame constructor, see Table 1.

Table 1: Possible data inputs to DataFrame constructor

Type	Notes
2D ndarray	A matrix of data, passing optional row and column labels
dict of arrays, lists, or tuples	Each sequence becomes a column in the DataFrame; all sequences must be the same length
NumPy structured/record array	Treated as the "dict of arrays" case
dict of Series	Each value becomes a column; indexes from each Series are unioned together to form the result's row index if no explicit index is passed
dict of dicts	Each inner dict becomes a column; keys are unioned to form the row index as in the "dict of Series" case
List of dicts or Series	Each item becomes a row in the DataFrame; union of dict keys or Series indexes become the DataFrame's column labels
List of lists or tuples	Treated as the "2D ndarray" case
Another DataFrame	The DataFrame's indexes are used unless different ones are passed
NumPy MaskedArray	Like the "2D ndarray" case except masked values become NA/missing in the DataFrame result

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

In [61]:
```
frame3
```

Out[61]:

	Nevada	Ohio
2001	2.4	1.7
2002	2.9	3.6
2000	NaN	1.5

In [62]:
```
frame3.index.name = 'year'; frame3.columns.name = 'state'
frame3
```

Out[62]:

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 [63]:
```
frame3.values
```

Out[63]:
```
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 [64]:
```
frame2.values
```

Out[64]:
```
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 [65]:
```
obj = pd.Series(range(3), index=['a', 'b', 'c'])
obj
```

Out[65]:
```
a    0
b    1
c    2
dtype: int64
```

In [66]:
```
index = obj.index
```

In [67]:
```
index
```

Out[67]:
```
Index(['a', 'b', 'c'], dtype='object')
```

In [68]:
```
index[1:]
```

Out[68]:
```
Index(['b', 'c'], dtype='object')
```

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

In [69]:
```
index[1] = 'd'  # TypeError
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-69-d11f5623d88a> in <module>
----> 1 index[1] = 'd'  # TypeError

E:\Anaconda3\lib\site-packages\pandas\core\indexes\base.py in __setitem__(self, key, val
ue)
   4258
   4259     def __setitem__(self, key, value):
-> 4260         raise TypeError("Index does not support mutable operations")
   4261
   4262     def __getitem__(self, key):

TypeError: Index does not support mutable operations
```

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

In [70]:
```
labels = pd.Index(np.arange(3))
labels
```

Out[70]:
```
Int64Index([0, 1, 2], dtype='int64')
```

In [71]:
```
obj2 = pd.Series([1.5, -2.5, 0], index=labels)
```

In [72]:
```
obj2
```

Out[72]:
```
0    1.5
1   -2.5
2    0.0
dtype: float64
```

In [73]:
```
obj2.index is labels
```

Out[73]:
```
True
```

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

In [74]:
```
frame3
```

Out[74]:

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

In [75]:
```
frame3.columns
```

Out[75]:
```
Index(['Nevada', 'Ohio'], dtype='object', name='state')
```

In [76]:
```
'Ohio' in frame3.columns
```

Out[76]:
```
True
```

In [77]:
```
2003 in frame3.index
```

Out[77]:
```
False
```

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

In [78]:
```
dup_labels = pd.Index(['foo', 'foo', 'bar', 'bar'])
```

In [79]:
```
dup_labels
```

Out[79]:
```
Index(['foo', 'foo', 'bar', 'bar'], dtype='object')
```

Selections with duplicate labels will select all occurrences of that label.

Each Index has a number of methods and properties for set logic, which answer other common questions about the data it contains. Some useful ones are summarized in Table 2.

Table 2: Some Index methods and properties.

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

Essential Functionality

This section will walk you through the fundamental mechanics of interacting with the data contained in a Series or DataFrame.

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 [80]:
```
obj = pd.Series([4.5, 7.2, -5.3, 3.6], index=['d', 'b', 'a', 'c'])
obj
```

Out[80]:
```
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, introducing missing values if any index values were not already present:

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

Out[81]:
```
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 [82]:
```
obj3 = pd.Series(['blue', 'purple', 'yellow'], index=[0, 2, 4])
```

In [83]:
```
obj3
```

Out[83]:
```
0      blue
2    purple
4    yellow
dtype: object
```

In [84]:
```
obj3.reindex(range(6), method='ffill')
```

Out[84]:
```
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 [85]:
```
frame = pd.DataFrame(np.arange(9).reshape((3, 3)),
                     index=['a', 'c', 'd'],
                     columns=['Ohio', 'Texas', 'California'])
frame
```

Out[85]:

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

In [86]:
```
frame2 = frame.reindex(['a', 'b', 'c', 'd'])
frame2
```

Out[86]:

	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 [87]:
```
frame
```

Out[87]:

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

In [88]:
```
states = ['Texas', 'Utah', 'California']
frame.reindex(columns=states)
```

Out[88]:

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

See Table 3 for more about the arguments to `reindex` .

Table 3: `reindex` function arguments

Argument	Description
`index`	New sequence to use as index. Can be Index instance or any other sequence-like Python data structure. An Index will be used exactly as is without any copying.
`method`	Interpolation (fill) method; `'ffill'` fills forward, while `'bfill'` fills backward.
`fill_value`	Substitute value to use when introducing missing data by reindexing.
`limit`	When forward- or backfilling, maximum size gap (in number of elements) to fill.
`tolerance`	When forward- or backfilling, maximum size gap (in absolute numeric distance) to fill for inexact matches.
`level`	Match simple Index on level of MultiIndex; otherwise select subset of.
`copy`	If `True`, always copy underlying data even if new index is equivalent to old index; if `False`, do not copy the data when the indexes are equivalent.

You can `reindex` more concisely by label-indexing with `loc` , and many users prefer to use it exclusively:

In [89]:
```
frame.loc[['a', 'b', 'c', 'd'], states]
```

```
E:\Anaconda3\lib\site-packages\pandas\core\indexing.py:1418: FutureWarning:
Passing list-likes to .loc or [] with any missing label will raise
KeyError in the future, you can use .reindex() as an alternative.

See the documentation here:
https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#deprecate-loc-rein
dex-listlike
  return self._getitem_tuple(key)
```

Out[89]:

	Texas	Utah	California
a	1.0	NaN	2.0
b	NaN	NaN	NaN
c	4.0	NaN	5.0
d	7.0	NaN	8.0

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 [90]:
```
obj = pd.Series(np.arange(5.), index=['a', 'b', 'c', 'd', 'e'])
obj
```

Out[90]:
```
a    0.0
b    1.0
c    2.0
d    3.0
e    4.0
dtype: float64
```

In [91]:
```
new_obj = obj.drop('c')
```

In [92]:
```
new_obj
```

Out[92]:
```
a    0.0
b    1.0
d    3.0
e    4.0
dtype: float64
```

In [93]:
```
obj.drop(['d', 'c'])
```

Out[93]:
```
a    0.0
b    1.0
e    4.0
dtype: float64
```

In [94]:
```
obj  # NOTE the output, why? See 'inplace' after a few cells.
```

Out[94]:
```
a    0.0
b    1.0
c    2.0
d    3.0
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 [95]:
```
data = pd.DataFrame(np.arange(16).reshape((4, 4)),
                    index=['Ohio', 'Colorado', 'Utah', 'New York'],
                    columns=['one', 'two', 'three', 'four'])
data
```

Out[95]:

	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 [96]:
```
data.drop(['Colorado', 'Ohio'])
```

Out[96]:

	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 [97]:
```
data.drop('two', axis=1)
```

Out[97]:

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

In [98]:
```
data.drop(['two', 'four'], axis='columns')
```
Out[98]:

	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 `inplace` **without returning a new object**:

In [99]:
```
obj
```
Out[99]:
```
a    0.0
b    1.0
c    2.0
d    3.0
e    4.0
dtype: float64
```

In [100]:
```
obj.drop('c', inplace=True)
```

In [101]:
```
obj  #Be careful with the inplace, as it destroys any data that is dropped.
```
Out[101]:
```
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 [112]:
```
obj = pd.Series(np.arange(4.), index=['a', 'b', 'c', 'd'])
obj
```
Out[112]:
```
a    0.0
b    1.0
c    2.0
d    3.0
dtype: float64
```

In [103]:
```
obj['b']
```
Out[103]:
```
1.0
```

In [104]:
```
obj[1]
```
Out[104]:
```
1.0
```

In [105]:
```
obj[2:4]
```
Out[105]:
```
c    2.0
d    3.0
dtype: float64
```

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

Out[106]:
```
b    1.0
a    0.0
d    3.0
dtype: float64
```

In [107]:
```
obj[[1, 3]]
```

Out[107]:
```
b    1.0
d    3.0
dtype: float64
```

In [108]:
```
obj[obj < 2]
```

Out[108]:
```
a    0.0
b    1.0
dtype: float64
```

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

In [109]:
```
obj
```

Out[109]:
```
a    0.0
b    1.0
c    2.0
d    3.0
dtype: float64
```

In [110]:
```
obj['b':'c']
```

Out[110]:
```
b    1.0
c    2.0
dtype: float64
```

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

In [113]:
```
obj['b':'c'] = 5
obj
```

Out[113]:
```
a    0.0
b    5.0
c    5.0
d    3.0
dtype: float64
```

Indexing into a DataFrame is for retrieving one or more columns either with a single value or sequence:

In [114]:
```
data = pd.DataFrame(np.arange(16).reshape((4, 4)),
                    index=['Ohio', 'Colorado', 'Utah', 'New York'],
                    columns=['one', 'two', 'three', 'four'])
data
```

Out[114]:

	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 [115]: data['two']
```

```
Out[115]: Ohio         1
          Colorado     5
          Utah         9
          New York    13
          Name: two, dtype: int32
```

```
In [116]: data[['three', 'one']]
```

Out[116]:

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

Indexing like this has a few special cases. First, slicing or selecting data with a boolean array:

```
In [117]: data[:2]
```

Out[117]:

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7

```
In [118]: data[data['three'] > 5]
```

Out[118]:

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

```
In [119]: data
```

Out[119]:

	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

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 [120]: data < 5
```

Out[120]:

	one	two	three	four
Ohio	True	True	True	True
Colorado	True	False	False	False
Utah	False	False	False	False
New York	False	False	False	False

```
In [121]: data[data < 5] = 0
```

In [122]:
```
data
```
Out[122]:

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

This makes DataFrame syntactically more like a two-dimensional NumPy array in this particular case.

Selection with loc and iloc

For DataFrame label-indexing on the rows, there are two 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 [123]:
```
data
```
Out[123]:

	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

As a preliminary example, let’s select a single row and multiple columns by label:

In [124]:
```
data.loc['Colorado', ['two', 'three']]
```
Out[124]:
```
two      5
three    6
Name: Colorado, dtype: int32
```

We’ll then perform some similar selections with integers using iloc:

In [125]:
```
data.iloc[2, [3, 0, 1]]
```
Out[125]:
```
four    11
one      8
two      9
Name: Utah, dtype: int32
```

In [126]:
```
data.iloc[2]
```
Out[126]:
```
one       8
two       9
three    10
four     11
Name: Utah, dtype: int32
```

In [127]:
```
data
```
Out[127]:

	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

In [128]:
```
data.iloc[[1, 2], [3, 0, 1]]
```
Out[128]:

	four	one	two
Colorado	7	0	5
Utah	11	8	9

Both indexing functions work with slices in addition to single labels or lists of labels:

In [129]:
```
data.loc[:'Utah', 'two']
```
Out[129]:
```
Ohio        0
Colorado    5
Utah        9
Name: two, dtype: int32
```

In [130]:
```
data
```
Out[130]:

	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

In [131]:
```
data.iloc[:, :3]
```
Out[131]:

	one	two	three
Ohio	0	0	0
Colorado	0	5	6
Utah	8	9	10
New York	12	13	14

In [132]:
```
data.iloc[:, :3][data.three > 5]
```
Out[132]:

	one	two	three
Colorado	0	5	6
Utah	8	9	10
New York	12	13	14

So there are many ways to select and rearrange the data contained in a pandas object. For DataFrame, Table 4 provides a short summary of many of them.

Table 4: Indexing options with DataFrame.

Type	Notes
`df[val]`	Select single column or sequence of columns from the DataFrame; special case conveniences: boolean array (filter rows), slice (slice rows), or boolean DataFrame (set values based on some criterion)
`df.loc[val]`	Selects single row or subset of rows from the DataFrame by label
`df.loc[:, val]`	Selects single column or subset of columns by label
`df.loc[val1, val2]`	Select both rows and columns by label
`df.iloc[where]`	Selects single row or subset of rows from the DataFrame by integer position
`df.iloc[:, where]`	Selects single column or subset of columns by integer position
`df.iloc[where_i, where_j]`	Select both rows and columns by integer position
`df.at[label_i, label_j]`	Select a single scalar value by row and column label
`df.iat[i, j]`	Select a single scalar value by row and column position (integers)
`reindex` method	Select either rows or columns by labels
`get_value, set_value` methods	Select single value by row and column label

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:

In [133]:
```
ser = pd.Series(np.arange(3.))
```

In [134]:
```
ser
```

Out[134]:
```
0    0.0
1    1.0
2    2.0
dtype: float64
```

In [135]:
```
ser[-1]
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-135-44969a759c20> in <module>
----> 1 ser[-1]

E:\Anaconda3\lib\site-packages\pandas\core\series.py in __getitem__(self, key)
   1066         key = com.apply_if_callable(key, self)
   1067         try:
-> 1068             result = self.index.get_value(self, key)
   1069 
   1070             if not is_scalar(result):

E:\Anaconda3\lib\site-packages\pandas\core\indexes\base.py in get_value(self, series, ke
y)
   4728         k = self._convert_scalar_indexer(k, kind="getitem")
   4729         try:
-> 4730             return self._engine.get_value(s, k, tz=getattr(series.dtype, "tz", N
one))
   4731         except KeyError as e1:
   4732             if len(self) > 0 and (self.holds_integer() or self.is_boolean()):

pandas\_libs\index.pyx in pandas._libs.index.IndexEngine.get_value()

pandas\_libs\index.pyx in pandas._libs.index.IndexEngine.get_value()

pandas\_libs\index.pyx in pandas._libs.index.IndexEngine.get_loc()

pandas\_libs\hashtable_class_helper.pxi in pandas._libs.hashtable.Int64HashTable.get_ite
m()

pandas\_libs\hashtable_class_helper.pxi in pandas._libs.hashtable.Int64HashTable.get_ite
m()

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

In [ ]:
```
ser
```

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

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

In [ ]:
```
ser2
```

In [ ]:
```
ser2[-1]
```

In [136]:
```
ser[:1]
```

Out[136]:
```
0    0.0
dtype: float64
```

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 [137]:
```
ser.loc[:2]  #Note: '2' is inclusive since 'loc' is for labels
```

Out[137]:
```
0    0.0
1    1.0
2    2.0
dtype: float64
```

Important Note: that contrary to usual python slices, **both** the start and the stop are included for `loc`

In [138]:
```
ser.iloc[:1]
```

Out[138]:
```
0    0.0
dtype: float64
```

Arithmetic and Data Alignment

An important pandas feature for some applications is the behavior of arithmetic between objects with different indexes. When you are adding together objects, if any index pairs are not the same, the respective index in the result will be the union of the index pairs. For users with database experience, this is similar to an automatic outer join on the index labels. Let's look at an example:

In [139]:
```
s1 = pd.Series([7.3, -2.5, 3.4, 1.5], index=['a', 'c', 'd', 'e'])
s1
```

Out[139]:
```
a    7.3
c   -2.5
d    3.4
e    1.5
dtype: float64
```

In [140]:
```
s2 = pd.Series([-2.1, 3.6, -1.5, 4, 3.1],
               index=['a', 'c', 'e', 'f', 'g'])
s2
```

Out[140]:
```
a   -2.1
c    3.6
e   -1.5
f    4.0
g    3.1
dtype: float64
```

Adding these together yields:

In [141]:
```
s1 + s2
```

Out[141]:
```
a    5.2
c    1.1
d    NaN
e    0.0
f    NaN
g    NaN
dtype: float64
```

The internal data alignment introduces missing values in the label locations that don't overlap. Missing values will then propagate in further arithmetic computations.

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

In [142]:
```
df1 = pd.DataFrame(np.arange(9.).reshape((3, 3)), columns=list('bcd'),
                   index=['Ohio', 'Texas', 'Colorado'])
df1
```

Out[142]:

	b	c	d
Ohio	0.0	1.0	2.0
Texas	3.0	4.0	5.0
Colorado	6.0	7.0	8.0

In [143]:
```
df2 = pd.DataFrame(np.arange(12.).reshape((4, 3)), columns=list('bde'),
                   index=['Utah', 'Ohio', 'Texas', 'Oregon'])
df2
```

Out[143]:

	b	d	e
Utah	0.0	1.0	2.0
Ohio	3.0	4.0	5.0
Texas	6.0	7.0	8.0
Oregon	9.0	10.0	11.0

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

In [144]:
```
df1 + df2
```

Out[144]:

	b	c	d	e
Colorado	NaN	NaN	NaN	NaN
Ohio	3.0	NaN	6.0	NaN
Oregon	NaN	NaN	NaN	NaN
Texas	9.0	NaN	12.0	NaN
Utah	NaN	NaN	NaN	NaN

Since the `'c'` and `'e'` columns are not found in both DataFrame objects, they appear as all missing in the result. The same holds for the rows whose labels are not common to both objects.

If you add DataFrame objects with no column or row labels in common, the result will contain all nulls:

In [145]:
```
df1 = pd.DataFrame({'A': [1, 2]})
df1
```

Out[145]:

	A
0	1
1	2

In [146]:
```
df2 = pd.DataFrame({'B': [3, 4]})
df2
```

Out[146]:

	B
0	3
1	4

In [147]:
```
df1 - df2
```
Out[147]:

	A	B
0	NaN	NaN
1	NaN	NaN

Arithmetic methods with fill values

In arithmetic operations between differently indexed objects, you might want to fill with a special value, like 0, when an axis label is found in one object but not the other:

In [148]:
```
df1 = pd.DataFrame(np.arange(12.).reshape((3, 4)),
                   columns=list('abcd'))
df1
```
Out[148]:

	a	b	c	d
0	0.0	1.0	2.0	3.0
1	4.0	5.0	6.0	7.0
2	8.0	9.0	10.0	11.0

In [149]:
```
df2 = pd.DataFrame(np.arange(20.).reshape((4, 5)),
                   columns=list('abcde'))
df2
```
Out[149]:

	a	b	c	d	e
0	0.0	1.0	2.0	3.0	4.0
1	5.0	6.0	7.0	8.0	9.0
2	10.0	11.0	12.0	13.0	14.0
3	15.0	16.0	17.0	18.0	19.0

In [150]:
```
df2.loc[1, 'b'] = np.nan
```
In [151]:
```
df2
```
Out[151]:

	a	b	c	d	e
0	0.0	1.0	2.0	3.0	4.0
1	5.0	NaN	7.0	8.0	9.0
2	10.0	11.0	12.0	13.0	14.0
3	15.0	16.0	17.0	18.0	19.0

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

In [152]:
```
df1 + df2
```
Out[152]:

	a	b	c	d	e
0	0.0	2.0	4.0	6.0	NaN
1	9.0	NaN	13.0	15.0	NaN
2	18.0	20.0	22.0	24.0	NaN
3	NaN	NaN	NaN	NaN	NaN

Using the `add` method on `df1`, you can pass `df2` and an argument to `fill_value`:

In [153]:
```
df1.add(df2, fill_value=0)
```

Out[153]:

	a	b	c	d	e
0	0.0	2.0	4.0	6.0	4.0
1	9.0	5.0	13.0	15.0	9.0
2	18.0	20.0	22.0	24.0	14.0
3	15.0	16.0	17.0	18.0	19.0

See Table 5 for a listing of Series and DataFrame methods for arithmetic.

Table 5: 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 (**)

Each of them has a counterpart, starting with the letter **r**, that has arguments flipped. So these two statements are equivalent:

In [154]:
```
1 / df1
```

Out[154]:

	a	b	c	d
0	inf	1.000000	0.500000	0.333333
1	0.250	0.200000	0.166667	0.142857
2	0.125	0.111111	0.100000	0.090909

In [155]:
```
df1.rdiv(1)
```

Out[155]:

	a	b	c	d
0	inf	1.000000	0.500000	0.333333
1	0.250	0.200000	0.166667	0.142857
2	0.125	0.111111	0.100000	0.090909

Relatedly, when reindexing a Series or DataFrame, you can also specify a different `fill_value` :

In [156]:
```
df1.reindex(columns=df2.columns, fill_value=777)
```

Out[156]:

	a	b	c	d	e
0	0.0	1.0	2.0	3.0	777
1	4.0	5.0	6.0	7.0	777
2	8.0	9.0	10.0	11.0	777

Operations between DataFrame and Series

As with NumPy arrays of different dimensions, arithmetic between DataFrame and Series is also defined. First, as a motivating example, consider the difference between a two-dimensional array and one of its rows:

In [157]:
```
arr = np.arange(12.).reshape((3, 4))
arr
```
Out[157]:
```
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  6.,  7.],
       [ 8.,  9., 10., 11.]])
```
In [158]:
```
arr[0]
```
Out[158]:
```
array([0., 1., 2., 3.])
```
In [159]:
```
arr - arr[0]
```
Out[159]:
```
array([[0., 0., 0., 0.],
       [4., 4., 4., 4.],
       [8., 8., 8., 8.]])
```

When we subtract `arr[0]` from `arr`, the subtraction is performed once **for each row**. This is referred to as ***broadcasting***.

Operations between a DataFrame and a Series are similar:

In [160]:
```
frame = pd.DataFrame(np.arange(12.).reshape((4, 3)),
                     columns=list('bde'),
                     index=['Utah', 'Ohio', 'Texas', 'Oregon'])
frame
```
Out[160]:

	b	d	e
Utah	0.0	1.0	2.0
Ohio	3.0	4.0	5.0
Texas	6.0	7.0	8.0
Oregon	9.0	10.0	11.0

In [161]:
```
series = frame.iloc[0]
series
```
Out[161]:
```
b    0.0
d    1.0
e    2.0
Name: Utah, dtype: float64
```
In [162]:
```
frame - series # Note: The subtraction is performed once for each row.
```
Out[162]:

	b	d	e
Utah	0.0	0.0	0.0
Ohio	3.0	3.0	3.0
Texas	6.0	6.0	6.0
Oregon	9.0	9.0	9.0

If an index value is not found in either the DataFrame's columns or the Series's index, the objects will be reindexed to form the union:

In [163]:
```
series2 = pd.Series(range(3), index=['b', 'e', 'f'])
series2
```
Out[163]:
```
b    0
e    1
f    2
dtype: int64
```

In [164]:
```
frame
```
Out[164]:

	b	d	e
Utah	0.0	1.0	2.0
Ohio	3.0	4.0	5.0
Texas	6.0	7.0	8.0
Oregon	9.0	10.0	11.0

In [165]:
```
frame + series2
```
Out[165]:

	b	d	e	f
Utah	0.0	NaN	3.0	NaN
Ohio	3.0	NaN	6.0	NaN
Texas	6.0	NaN	9.0	NaN
Oregon	9.0	NaN	12.0	NaN

If you want to instead broadcast over the columns, matching on the rows, you have to use one of the arithmetic methods. For example:

In [166]:
```
series3 = frame['d']
```
In [167]:
```
series3
```
Out[167]:
```
Utah       1.0
Ohio       4.0
Texas      7.0
Oregon    10.0
Name: d, dtype: float64
```
In [168]:
```
frame
```
Out[168]:

	b	d	e
Utah	0.0	1.0	2.0
Ohio	3.0	4.0	5.0
Texas	6.0	7.0	8.0
Oregon	9.0	10.0	11.0

In [169]:
```
frame.sub(series3, axis='index')
```
Out[169]:

	b	d	e
Utah	-1.0	0.0	1.0
Ohio	-1.0	0.0	1.0
Texas	-1.0	0.0	1.0
Oregon	-1.0	0.0	1.0

The axis number that you pass is the axis to match on. In this case we mean to match on the DataFrame's row index (`axis='index'` or `axis=0`) and broadcast across.

Function Application and Mapping

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

In [170]:
```
frame = pd.DataFrame(np.random.randn(4, 3), columns=list('bde'),
                     index=['Utah', 'Ohio', 'Texas', 'Oregon'])
frame
```

Out[170]:

	b	d	e
Utah	-2.449249	0.461031	-0.575932
Ohio	-0.677216	1.296211	-0.338710
Texas	-2.823431	-0.634437	-0.152185
Oregon	0.241051	0.717488	-0.445206

In [171]:
```
np.abs(frame)
```

Out[171]:

	b	d	e
Utah	2.449249	0.461031	0.575932
Ohio	0.677216	1.296211	0.338710
Texas	2.823431	0.634437	0.152185
Oregon	0.241051	0.717488	0.445206

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

In [172]:
```
f = lambda x: x.max() - x.min()
frame.apply(f)
```

Out[172]:
```
b    3.064483
d    1.930648
e    0.423747
dtype: float64
```

Here the function *f*, which computes the difference between the maximum and minimum of a Series, is invoked once on each **column** in `frame`. The result is a Series having the columns of `frame` as its index.

If you pass `axis='columns'` to apply, the function will be invoked once per **row** instead:

In [173]:
```
frame.apply(f, axis='columns')
```

Out[173]:
```
Utah      2.910280
Ohio      1.973427
Texas     2.671246
Oregon    1.162694
dtype: float64
```

Many of the most common array statistics (like `sum` and `mean`) are DataFrame methods, so using apply is not necessary.

The function passed to `apply` need not return a scalar value; it can also return a Series with multiple values:

In [174]:
```
def f(x):
    return pd.Series([x.min(), x.max()], index=['min', 'max'])
frame.apply(f)
```

Out[174]:

	b	d	e
min	-2.823431	-0.634437	-0.575932
max	0.241051	1.296211	-0.152185

Element-wise Python functions can be used, too. Suppose you wanted to compute a formatted string from each floating-point value in frame. You can do this with `applymap`:

In [175]:
```
format = lambda x: '%.2f' %x
frame.applymap(format)
```

Out[175]:

	b	d	e
Utah	-2.45	0.46	-0.58
Ohio	-0.68	1.30	-0.34
Texas	-2.82	-0.63	-0.15
Oregon	0.24	0.72	-0.45

The reason for the name `applymap` is that Series has a `map` method for applying an element-wise function:

In [176]:
```
frame['e'].map(format)
```

Out[176]:
```
Utah      -0.58
Ohio      -0.34
Texas     -0.15
Oregon    -0.45
Name: e, dtype: object
```

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 [177]:
```
obj = pd.Series(range(4), index=['d', 'a', 'b', 'c'])
```

In [178]:
```
obj
```

Out[178]:
```
d    0
a    1
b    2
c    3
dtype: int64
```

In [179]:
```
obj.sort_index()
```

Out[179]:
```
a    1
b    2
c    3
d    0
dtype: int64
```

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

In [180]:
```
frame = pd.DataFrame(np.arange(8).reshape((2, 4)),
                     index=['three', 'one'],
                     columns=['d', 'a', 'b', 'c'])
```

In [181]:
```
frame
```

Out[181]:

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

In [182]:
```
frame.sort_index()
```

Out[182]:

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

In [183]:
```
frame.sort_index(axis=1)
```

Out[183]:

	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 [184]:
```
frame.sort_index(axis=1, ascending=False)
```

Out[184]:

	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 [185]:
```
obj = pd.Series([4, 7, -3, 2])
```

In [186]:
```
obj
```

Out[186]:
```
0    4
1    7
2   -3
3    2
dtype: int64
```

In [187]:
```
obj.sort_values()
```

Out[187]:
```
2   -3
3    2
0    4
1    7
dtype: int64
```

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

In [188]:
```
obj = pd.Series([4, np.nan, 7, np.nan, -3, 2])
```

In [189]:
```
obj
```

Out[189]:
```
0    4.0
1    NaN
2    7.0
3    NaN
4   -3.0
5    2.0
dtype: float64
```

In [190]:
```
obj.sort_values()
```

Out[190]:
```
4   -3.0
5    2.0
0    4.0
2    7.0
1    NaN
3    NaN
dtype: float64
```

When sorting a DataFrame, you can use the data in one or more columns as the sort keys. To do so, pass one or more column names to the `by` option of `sort_values` :

In [191]:
```
frame = pd.DataFrame({'b': [4, 7, -3, 2], 'a': [0, 1, 0, 1]})
frame
```

Out[191]:

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

In [192]:
```
frame.sort_values(by='b')
```

Out[192]:

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

To sort by multiple columns, pass a list of names:

In [193]:
```
frame.sort_values(by=['a', 'b'])
```

Out[193]:

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

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 [194]:
```
obj = pd.Series([7, -5, 7, 4, 2, 0, 4])
```

In [195]:
```
obj
```

Out[195]:
```
0    7
1   -5
2    7
3    4
4    2
5    0
6    4
dtype: int64
```

```
In [196]: obj.rank()
```

```
Out[196]: 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.

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.

```
In [197]: obj.rank(method='first')
```

```
Out[197]: 0    6.0
          1    1.0
          2    7.0
          3    4.0
          4    3.0
          5    2.0
          6    5.0
          dtype: float64
```

You can rank in descending order, too:

```
In [198]: # Assign tie values the maximum rank in the group
          obj.rank(ascending=False, method='max')
```

```
Out[198]: 0    2.0
          1    7.0
          2    2.0
          3    4.0
          4    5.0
          5    6.0
          6    4.0
          dtype: float64
```

See Table 6 for a list of tie-breaking methods available.

Table 6: Tie-breaking methods with rank.

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

DataFrame can compute ranks over the rows or the columns:

In [199]:
```
frame = pd.DataFrame({'b': [4.3, 7, -3, 2], 'a': [0, 1, 0, 1],
                      'c': [-2, 5, 8, -2.5]})
frame
```

Out[199]:

	b	a	c
0	4.3	0	-2.0
1	7.0	1	5.0
2	-3.0	0	8.0
3	2.0	1	-2.5

In [200]:
```
frame.rank(axis='columns')
```

Out[200]:

	b	a	c
0	3.0	2.0	1.0
1	3.0	1.0	2.0
2	1.0	2.0	3.0
3	3.0	2.0	1.0

Axis Indexes with Duplicate Labels

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 [201]:
```
obj = pd.Series(range(5), index=['a', 'a', 'b', 'b', 'c'])
obj
```

Out[201]:
```
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 [202]:
```
obj.index.is_unique
```

Out[202]:
```
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 [203]:
```
obj['a']
```

Out[203]:
```
a    0
a    1
dtype: int64
```

In [204]:
```
obj['c']
```

Out[204]:
```
4
```

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 [205]:
```
df = pd.DataFrame(np.random.randn(4, 3), index=['a', 'a', 'b', 'b'])
```

In [206]:
```
df
```

Out[206]:

	0	1	2
a	0.682936	1.424945	-1.151070
a	0.278771	-0.329596	0.633083
b	-0.130387	1.685845	-1.529052
b	-0.179275	0.278146	-1.284492

In [207]:
```
df.loc['b']
```

Out[207]:

	0	1	2
b	-0.130387	1.685845	-1.529052
b	-0.179275	0.278146	-1.284492

Summarizing and Computing Descriptive Statistics

- pandas objects are equipped with a set of common mathematical and statistical methods.
- Most of these fall into the category of reductions or summary statistics, methods that extract a single value (like the `sum` or `mean`) from a Series or a Series of values from the rows or columns of a DataFrame.
- Compared with the similar methods found on NumPy arrays, they have built-in handling for **missing data**.

Consider a small DataFrame:

In [208]:
```
df = pd.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[208]:

	one	two
a	1.40	NaN
b	7.10	-4.5
c	NaN	NaN
d	0.75	-1.3

Calling DataFrame's sum method returns a Series containing column sums:

In [209]:
```
df.sum()
```

Out[209]:
```
one    9.25
two   -5.80
dtype: float64
```

Passing `axis='columns'` or `axis=1` sums across the columns instead:

In [210]:
```
df.sum(axis='columns')
```

Out[210]:
```
a    1.40
b    2.60
c    0.00
d   -0.55
dtype: float64
```

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

In [211]:
```
df.mean(axis='columns', skipna=False)
```

Out[211]:
```
a        NaN
b      1.300
c        NaN
d     -0.275
dtype: float64
```

See Table 7 for a list of common options for each reduction method.

Table 7: Options for reduction methods.

Method	Description
`axis`	Axis to reduce over; 0 for DataFrame's rows and 1 for columns
`skipna`	Exclude missing values; `True` by default
`level`	Reduce grouped by level if the axis is hierarchically indexed (MultiIndex)

In [212]:
```
df
```

Out[212]:

	one	two
a	1.40	NaN
b	7.10	-4.5
c	NaN	NaN
d	0.75	-1.3

Some methods, like `idxmin` and `idxmax`, return indirect statistics like the index value where the minimum or maximum values are attained:

In [213]:
```
df.idxmax()
```

Out[213]:
```
one      b
two      d
dtype: object
```

Other methods are accumulations:

In [214]:
```
df.cumsum()
```

Out[214]:

	one	two
a	1.40	NaN
b	8.50	-4.5
c	NaN	NaN
d	9.25	-5.8

Another type of method is neither a reduction nor an accumulation. `describe` is one such example, producing multiple summary statistics in one shot:

In [215]:
```
df.describe()
```

Out[215]:

	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

In [216]:
```
obj = pd.Series(['a', 'a', 'b', 'c'] * 4)
```

In [217]:
```
obj
```

Out[217]:
```
0      a
1      a
2      b
3      c
4      a
5      a
6      b
7      c
8      a
9      a
10     b
11     c
12     a
13     a
14     b
15     c
dtype: object
```

In [218]:
```
# For object data (e.g. strings or timestamps), outputs are: count, unique, top, freq
# Here: The 'top' is the most common value. The 'freq' is the most common value's frequen
cy.
obj.describe()
```

Out[218]:
```
count     16
unique     3
top        a
freq       8
dtype: object
```

See Table 8 for a full list of summary statistics and related methods.

Table 8: Descriptive and summary statistics.

Method	Description
count	Number of non-NA values
describe	Compute set of summary statistics for Series or each DataFrame column
min, max	Compute minimum and maximum values
argmin, argmax	Compute index locations (integers) at which minimum or maximum value obtained, respectively
idxmin, idxmax	Compute index labels at which minimum or maximum value obtained, respectively
quantile	Compute sample quantile ranging from 0 to 1
sum	Sum of values
mean	Mean of values
median	Arithmetic median (50% quantile) of values
mad	Mean absolute deviation from mean value
prod	Product of all values
var	Sample variance of values
std	Sample standard deviation of values
skew	Sample skewness (third moment) of values
kurt	Sample kurtosis (fourth moment) of values
cumsum	Cumulative sum of values
cummin, cummax	Cumulative minimum or maximum of values, respectively
cumprod	Cumulative product of values
diff	Compute first arithmetic difference (useful for time series)
pct_change	Compute percent changes

Correlation and Covariance

Some summary statistics, like correlation and covariance, are computed from pairs of arguments.

In [219]:
```python
# Load iris dataset [you may review the code we used in chapter 1]
import pandas as pd
iris=pd.read_csv("iris.arff")

iriscp=iris.copy()
myreplacementlist= {"Iris-setosa":0, "Iris-versicolor":1,"Iris-virginica":2}
iriscp.replace({'class ': myreplacementlist}, inplace=True)

iriscp
```

Out[219]:

	sepal_length	sepal_width	petal_length	petal_width	class
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0
3	4.6	3.1	1.5	0.2	0
4	5.0	3.6	1.4	0.2	0
...	...	...	...	...	...
145	6.7	3.0	5.2	2.3	2
146	6.3	2.5	5.0	1.9	2
147	6.5	3.0	5.2	2.0	2
148	6.2	3.4	5.4	2.3	2
149	5.9	3.0	5.1	1.8	2

150 rows × 5 columns

In [220]:
```python
iriscp.columns
```

Out[220]:
```
Index(['sepal_length', 'sepal_width', 'petal_length', 'petal_width', 'class '], dtype='o
bject')
```

The `corr` method of Series computes the correlation of the overlapping, non-NA, aligned-by-index values in two Series. Relatedly, `cov` computes the covariance:

In [221]:
```python
# Correlation shows whether and how strongly pairs of variables are related.

iriscp['sepal_length'].corr(iriscp['class '])  # correlation between 'sepal_length' and '
class '
```

Out[221]:
```
0.7825612318100815
```

In [222]:
```python
# covariance is a measure of the joint variability of two random variables

iriscp['sepal_length'].cov(iriscp['class '])  # covariance:'sepal_length' versus 'class '
```

Out[222]:
```
0.5308724832214765
```

Since `sepal_length` is a valid Python attribute (but not 'class ' because of the space at the end), we can also select these columns using more concise syntax:

In [223]:
```python
iriscp.sepal_length.corr(iriscp['class '])
```

Out[223]:
```
0.7825612318100815
```

Ans

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

564-, 10-1.

F=1 : ... F_{10}

QDA

LDA

μ_0, μ_1

In [224]:
```
iriscp.corr()
```
Out[224]:

	sepal_length	sepal_width	petal_length	petal_width	class
sepal_length	1.000000	-0.109369	0.871754	0.817954	0.782561
sepal_width	-0.109369	1.000000	-0.420516	-0.356544	-0.419446
petal_length	0.871754	-0.420516	1.000000	0.962757	0.949043
petal_width	0.817954	-0.356544	0.962757	1.000000	0.956464
class	0.782561	-0.419446	0.949043	0.956464	1.000000

90%

10%

In [225]:
```
iriscp.cov()
```
Out[225]:

$\Sigma =$

	sepal_length	sepal_width	petal_length	petal_width	class
sepal_length	0.685694	-0.039268	1.273682	0.516904	0.530872
sepal_width	-0.039268	0.188004	-0.321713	-0.117981	-0.148993
petal_length	1.273682	-0.321713	3.113179	1.296387	1.371812
petal_width	0.516904	-0.117981	1.296387	0.582414	0.597987
class	0.530872	-0.148993	1.371812	0.597987	0.671141

22×22

22

Using DataFrame's `corrwith` method, you can compute pairwise correlations between a DataFrame's columns or rows with another Series or DataFrame. Passing a Series returns a Series with the correlation value computed for each column:

In [226]:
```
iriscp.corrwith(iriscp.sepal_length)
```
Out[226]:
```
sepal_length     1.000000
sepal_width     -0.109369
petal_length     0.871754
petal_width      0.817954
class            0.782561
dtype: float64
```

Unique Values, Value Counts, and Membership

Another class of related methods extracts information about the values contained in a one-dimensional Series. To illustrate these, consider this example:

In [227]:
```
obj = pd.Series(['c', 'a', 'd', 'a', 'a', 'b', 'b', 'c', 'c'])
```

The first function is unique, which gives you an array of the unique values in a Series:

In [228]:
```
uniques = obj.unique()
```

In [229]:
```
uniques
```
Out[229]:
```
array(['c', 'a', 'd', 'b'], dtype=object)
```

The unique values are not necessarily returned in sorted order, but could be sorted after the fact if needed (`uniques.sort()`). Relatedly, `value_counts` computes a Series containing value frequencies:

In [230]:
```
obj.value_counts()
```
Out[230]:
```
c    3
a    3
b    2
d    1
dtype: int64
```

The Series is sorted by value in descending order as a convenience. `value_counts` is also available as a top-level pandas method that can be used with any array or sequence:

In [231]:
```
pd.value_counts(obj.values, sort=False)
```
Out[231]:
```
b    2
a    3
d    1
c    3
dtype: int64
```

`isin` performs a vectorized set membership check and can be useful in filtering a dataset down to a subset of values in a Series or column in a DataFrame:

In [232]:
```
obj
```
Out[232]:
```
0    c
1    a
2    d
3    a
4    a
5    b
6    b
7    c
8    c
dtype: object
```

In [233]:
```
mask = obj.isin(['b', 'c'])
```

In [234]:
```
mask
```
Out[234]:
```
0     True
1    False
2    False
3    False
4    False
5     True
6     True
7     True
8     True
dtype: bool
```

In [235]:
```
obj[mask]
```
Out[235]:
```
0    c
5    b
6    b
7    c
8    c
dtype: object
```

Related to `isin` is the `Index.get_indexer` method, which gives you an index array from an array of possibly non-distinct values into another array of distinct values:

In [236]:
```
to_match = pd.Series(['c', 'a', 'b', 'b', 'c', 'a'])
unique_vals = pd.Series(['c', 'b', 'a'])
pd.Index(unique_vals).get_indexer(to_match)
```

Out[236]:
```
array([0, 2, 1, 1, 0, 2], dtype=int64)
```

See Table 9 for a reference on these methods.

Table 9: Unique, value counts, and set membership methods.

Method	Description
`isin`	Compute boolean array indicating whether each Series value is contained in the passed sequence of values
`get_indexer`	Compute integer indices for each value in an array into another array of distinct values; helpful for data alignment and join-type operations
`unique`	Compute array of unique values in a Series, returned in the order observed
`value_counts`	Return a Series containing unique values as its index and frequencies as its values, ordered count in descending order

In some cases, you may want to compute a histogram on multiple related columns in a DataFrame. Here's an example:

In [237]:
```
data = pd.DataFrame({'Qu1': [1, 3, 4, 3, 4],
                     'Qu2': [2, 3, 1, 2, 3],
                     'Qu3': [1, 5, 2, 4, 4]})
data
```

Out[237]:

	Qu1	Qu2	Qu3
0	1	2	1
1	3	3	5
2	4	1	2
3	3	2	4
4	4	3	4

Passing `pandas.value_counts` to this DataFrame's `apply` function gives:

In [238]:
```
result = data.apply(pd.value_counts).fillna(0)
result
```

Out[238]:

	Qu1	Qu2	Qu3
1	1.0	1.0	1.0
2	0.0	2.0	1.0
3	2.0	2.0	0.0
4	2.0	0.0	2.0
5	0.0	0.0	1.0

Here, the row labels in the result are the distinct values occurring in all of the columns. The values are the respective counts of these values in each column.

References:

[1] Python for Data Analysis Data Wrangling with Pandas, NumPy, and IPython by Wes McKinney, 2nd Edn, O'Reilly 2017.