Practice 2: Data exploration

Practice 2:
2.1. Pandas

Pandas Features

- Easy handling of missing data (represented as NaN) in floating point as well as non-floating point data
- Size mutability: columns can be inserted and deleted from DataFrame and higher dimensional objects
- Automatic and explicit data alignment: objects can be explicitly aligned to a set of labels, or the user can simply ignore the labels and let Series, DataFrame, etc. automatically align the data for you in computations
- Powerful, flexible group by functionality to perform split-apply-combine operations on data sets, for both aggregating and transforming data
- Make it easy to convert ragged, differently-indexed data in other Python and NumPy data structures into DataFrame objects
- Intelligent label-based slicing, fancy indexing, and subsetting of large data sets
- Intuitive merging and joining data sets
- Flexible reshaping and pivoting of data sets
- Hierarchical labeling of axes (possible to have multiple labels per tick)
- Robust IO tools for loading data from flat files (CSV and delimited), Excel files, databases, and saving / loading data from the ultrafast HDF5 format
- Time series-specific functionality: date range generation and frequency conversion, moving window statistics, moving window linear regressions, date shifting and lagging, etc.

Tutorials

- Many good tutorials
  - https://pandas.pydata.org/pandas-docs/stable/getting_started/10min.html
  - https://www.datacamp.com/community/tutorials/pandas-tutorial-dataframe-python

Data structures

- Series: Container of scalars
  - 1D labeled homogeneously-typed array
- DataFrame: Container of series
  - General 2D labeled, size-mutable tabular structure with potentially heterogeneously-typed column

Import

- Import numpy and pandas

```
import numpy as np
import pandas as pd
```

Series

- Series is a one-dimensional labeled array capable of holding any data type (integers, strings, floating point numbers, Python objects, etc.). The axis labels are collectively referred to as the index. The basic method to create a Series is to call:

```
>>> s = pd.Series(data, index=index)
```

- Here, data can be many different things:
  - a Python dict
  - an ndarray
  - a scalar value (like 5)
- The passed index is a list of axis labels. Thus, this separates into a few cases depending on what data is.

Series from ndarray

- If data is an ndarray, index must be the same length as data. If no index is passed, one will be created having values [0, ..., len(data) – 1].

```
>>> s = pd.Series(np.random.randn(5), index=['a', 'b', 'c', 'd', 'e'])
>>> s
a    1.519050
b   -0.849656
c   -0.405133
d    3.707403
e   -0.367354
dtype: float64
>>> s.index
Index(['a', 'b', 'c', 'd', 'e'], dtype='object')
>>> s = pd.Series(np.random.randn(5))
>>> s
0    1.627234
1    1.343015
2   -1.189584
3   -0.066417
4    1.314388
dtype: float64
```

- Non-unique index values are supported

Series from dict

- Series can be instantiated from dicts:

```
>>> d = {'b':1, 'a':0, 'c':2}
>>> pd.Series(d)
b    1
a    0
c    2
dtype: int64
```

Series from scalar

- If data is a scalar value, an index must be provided. The value will be repeated to match the length of index.

```
>>> pd.Series(5., index=['a', 'b', 'c', 'd', 'e'])
a    5.0
b    5.0
c    5.0
d    5.0
e    5.0
dtype: float64
```

Series is ndarray-like

- Series acts very similarly to a ndarray, and is a valid argument to most NumPy functions. However, operations such as slicing will also slice the index.

```
>>> s
0   -0.752727
1    0.344025
2    1.300523
3    0.748218
4    0.647253
dtype: float64
>>> np.exp(s)
0    0.471080
1    1.410615
2    3.671215
3    2.113230
4    1.910286
dtype: float64
```

```
>>> s[0]
-0.752727132804256
>>> s[:3]
0   -0.752727
1    0.344025
2    1.300523
dtype: float64
>>> s[s > s.median()]
2    1.300523
3    0.748218
dtype: float64
>>> s[[4,3,1]]
4    0.647253
3    0.748218
1    0.344025
dtype: float64
```

Example of series creation

- Creating a Series by passing a list of values, letting pandas create a default integer index:

```
>>> s = pd.Series([1, 3, 5, np.nan, 6, 8])
>>> s
0    1.0
1    3.0
2    5.0
3    NaN
4    6.0
5    8.0
dtype: float64
```

DataFrame

- DataFrame is a 2-dimensional labeled data structure with columns of potentially different types. You can think of it like a spreadsheet or SQL table, or a dict of Series objects. It is generally the most commonly used pandas object. Like Series, DataFrame accepts many different kinds of input:
  - Dict of 1D ndarrays, lists, dicts, or Series
  - 2-D numpy.ndarray
  - Structured or record ndarray
  - A Series
  - Another DataFrame
- Along with the data, you can optionally pass index (row labels) and columns (column labels) arguments. If you pass an index and/or columns, you are guaranteeing the index and/or columns of the resulting DataFrame. Thus, a dict of Series plus a specific index will discard all data not matching up to the passed index.
- If axis labels are not passed, they will be constructed from the input data based on common sense rules.

DataFrame from ndarray

- Creating a DataFrame by passing a NumPy array, with a datetime index and labeled columns:

```
>>> dates = pd.date_range('20130101', periods=6)
>>> dates
DatetimeIndex(['2013-01-01', '2013-01-02', '2013-01-03', '2013-01-
04','2013-01-05', '2013-01-06'],
              dtype='datetime64[ns]', freq='D')
>>> df = pd.DataFrame(np.random.randn(6, 4), index=dates,
columns=list("ABCD"))
>>> df
                   A         B         C         D
2013-01-01  0.072782 -1.068376  1.636667 -1.564652
2013-01-02  2.769694  1.626064 -0.757751  0.882998
2013-01-03 -0.392577 -1.021717  0.494420  0.767871
2013-01-04 -0.107602  0.103907 -0.649417  0.351641
2013-01-05 -0.175708 -1.299035 -0.502587  0.175272
2013-01-06 -0.351864 -0.274365 -0.440517 -0.973947
```

DataFrame from dict

- Creating a DataFrame by passing a dict of objects that can be converted to series-like.

```
>>> df2 = pd.DataFrame({'A': 1.,
...                     'B': pd.Timestamp('20130102'),
...                     'C': pd.Series(1, index=list(range(4)), dtype='float32'),
...                     'D': np.array([3] * 4, dtype='int32'),
...                     'E': pd.Categorical(["test", "train", "test", "train"]),
...                     'F': 'foo'})
>>>
>>> df2
     A          B    C  D      E    F
0  1.0 2013-01-02  1.0  3   test  foo
1  1.0 2013-01-02  1.0  3  train  foo
2  1.0 2013-01-02  1.0  3   test  foo
3  1.0 2013-01-02  1.0  3  train  foo
```

DataFrame from dict

- The columns of the resulting DataFrame have different dtypes.

```
>>> df2.dtypes
A             float64
B      datetime64[ns]
C             float32
D               int32
E            category
F              object
dtype: object
```

Viewing data

- View the top and bottom rows of the frame:

```
>>> df.head()
                   A         B         C         D
2013-01-01 -0.517246  0.703026 -2.005974 -0.683947
2013-01-02 -0.893703 -1.598953  0.263607  0.882999
2013-01-03 -0.879086  0.820931 -1.006811  0.730761
2013-01-04 -1.522220  0.335845 -0.268703 -0.701878
2013-01-05  0.698671  1.561876  0.329810 -0.301600
>>> df.tail(3)
                   A         B         C         D
2013-01-04 -1.522220  0.335845 -0.268703 -0.701878
2013-01-05  0.698671  1.561876  0.329810 -0.301600
2013-01-06 -1.223615 -0.777588  0.774213 -0.072565
```

- Display index, columns

```
>>> df.index
DatetimeIndex(['2013-01-01', '2013-01-02', '2013-01-03', '2013-01-04',
               '2013-01-05', '2013-01-06'],
              dtype='datetime64[ns]', freq='D')
>>> df.columns
Index(['A', 'B', 'C', 'D'], dtype='object')
```

Convert to ndarray

- DataFrame.to_numpy() gives a NumPy representation of the underlying data.
- Note that this can be an expensive operation when your DataFrame has columns with different data types, which comes down to a fundamental difference between pandas and NumPy:
    - NumPy arrays have one dtype for the entire array, while pandas DataFrames have one dtype per column.
- When you call DataFrame.to_numpy(), pandas will find the NumPy dtype that can hold all of the dtypes in the DataFrame.
- This may end up being object, which requires casting every value to a Python object.

DataFrame.to_numpy()

- For df, our DataFrame of all floating-point values, DataFrame.to_numpy() is fast and doesn't require copying data.

```
>>> df.to_numpy()
array([[-0.51724627,  0.70302631, -2.00597398, -0.68394667],
       [-0.89370328, -1.59895347,  0.26360718,  0.88299859],
       [-0.87908632,  0.82093119, -1.00681094,  0.73076059],
       [-1.52222034,  0.33584531, -0.26870329, -0.70187839],
       [ 0.6986712 ,  1.56187628,  0.32981012, -0.30159961],
       [-1.22361491, -0.77758782,  0.77421303, -0.07256494]])
```

- **Note:** DataFrame.to_numpy() does not include the index or column labels in the output.

Summary of data

- describe() shows a quick statistic summary of your data:

```
>>> df.describe()
              A         B         C         D
count  6.000000  6.000000  6.000000  6.000000
mean  -0.722867  0.174190 -0.318976 -0.024372
std    0.775417  1.157555  1.027326  0.688008
min   -1.522220 -1.598953 -2.005974 -0.701878
25%   -1.141137 -0.499230 -0.822284 -0.588360
50%   -0.886395  0.519436 -0.002548 -0.187082
75%   -0.607706  0.791455  0.313259  0.529929
max    0.698671  1.561876  0.774213  0.882999
```

Tramposing data

- Transpose: DataFrame.T

```
>>> df.T
   2013-01-01  2013-01-02  2013-01-03  2013-01-04  2013-01-05  2013-01-06
A   -0.517246   -0.893703   -0.879086   -1.522220    0.698671   -1.223615
B    0.703026   -1.598953    0.820931    0.335845    1.561876   -0.777588
C   -2.005974    0.263607   -1.006811   -0.268703    0.329810    0.774213
D   -0.683947    0.882999    0.730761   -0.701878   -0.301600   -0.072565
```

Sorting

- Sorting by an axis

```
>>> df.sort_index(axis=1, ascending=False)
                   D         C         B         A
2013-01-01 -0.683947 -2.005974  0.703026 -0.517246
2013-01-02  0.882999  0.263607 -1.598953 -0.893703
2013-01-03  0.730761 -1.006811  0.820931 -0.879086
2013-01-04 -0.701878 -0.268703  0.335845 -1.522220
2013-01-05 -0.301600  0.329810  1.561876  0.698671
2013-01-06 -0.072565  0.774213 -0.777588 -1.223615
```

- Sorting by a value

```
>>> df.sort_values(by='B')
                   A         B         C         D
2013-01-02 -0.893703 -1.598953  0.263607  0.882999
2013-01-06 -1.223615 -0.777588  0.774213 -0.072565
2013-01-04 -1.522220  0.335845 -0.268703 -0.701878
2013-01-01 -0.517246  0.703026 -2.005974 -0.683947
2013-01-03 -0.879086  0.820931 -1.006811  0.730761
2013-01-05  0.698671  1.561876  0.329810 -0.301600
```

Selection

- **Note:** While standard Python/Numpy expressions for selecting and setting are intuitive and come in handy for interactive work, for production code, we recommend the optimized pandas data access methods, .at, .iat, .loc and .iloc.
- Three basic modes:
  - []
  - .loc:
  - .iloc

.loc

- .loc is primarily label based, but may also be used with a boolean array. .loc will raise KeyError when the items are not found. Allowed inputs are:
  - A single label, e.g. 5 or 'a' (Note that 5 is interpreted as a label of the index. This use is not an integer position along the index.).
  - A list or array of labels ['a', 'b', 'c'].
  - A slice object with labels 'a':'f' (Note that contrary to usual python slices, both the start and the stop are included, when present in the index! See Slicing with labels and Endpoints are inclusive.)
  - A boolean array
  - A callable function with one argument (the calling Series or DataFrame) and that returns valid output for indexing (one of the above).

.iloc

- .iloc is primarily integer position based (from 0 to length-1 of the axis), but may also be used with a boolean array. .iloc will raise IndexError if a requested indexer is out-of-bounds, except slice indexers which allow out-of-bounds indexing. (this conforms with Python/NumPy slice semantics). Allowed inputs are:
  - An integer e.g. 5.
  - A list or array of integers [4, 3, 0].
  - A slice object with ints 1:7.
  - A boolean array.
  - A callable function with one argument (the calling Series or DataFrame) and that returns valid output for indexing (one of the above).

Basics

- The primary function of indexing with [] is selecting out lower-dimensional slices.

Object Type	Selection	Return Value Type
Series	`series[label]`	scalar value
DataFrame	`frame[colname]`	`Series` corresponding to colname

- Example:

```
>>> dates = pd.date_range('1/1/2000', periods=8)
>>> df = pd.DataFrame(np.random.randn(8, 4), index=dates, columns=['A', 'B', 'C', 'D'])
>>> df
                   A         B         C         D
2000-01-01 -0.799199 -1.594965  0.724898 -2.719031
2000-01-02  0.969193  0.950294  0.177314  0.107394
2000-01-03  1.667980 -0.593788 -0.270656  0.628732
2000-01-04  0.026901  0.525567 -0.575381 -0.453548
2000-01-05 -0.033356  1.465294 -1.050563  0.376754
2000-01-06 -0.423718 -0.917791  1.398832  0.255521
2000-01-07 -0.887049  0.208985  1.398034 -0.585589
2000-01-08  0.737691 -1.304846  1.013431  0.195951
```

Basics

- We have the most basic indexing using []:

```
>>> s = df['A']
>>> s
2000-01-01   -0.373138
2000-01-02    0.934918
2000-01-03   -1.571071
2000-01-04    2.369529
2000-01-05   -0.769576
2000-01-06    0.724399
2000-01-07   -0.391091
2000-01-08    0.458264
Freq: D, Name: A, dtype:
float64

>>> s[dates[5]]
0.724399478182
```

Basics

- You can pass a list of columns to [] to select columns in that order. Multiple columns can also be set in this manner.

```
>>> df[['B', 'A']]
                   B         A
2000-01-01 -1.594965 -0.799199
2000-01-02  0.950294  0.969193
2000-01-03 -0.593788  1.667980
2000-01-04  0.525567  0.026901
2000-01-05  1.465294 -0.033356
2000-01-06 -0.917791 -0.423718
2000-01-07  0.208985 -0.887049
2000-01-08 -1.304846  0.737691
>>> df[['B', 'A']] = df[['A', 'B']]
>>> df
                   A         B         C         D
2000-01-01 -1.594965 -0.799199  0.724898 -2.719031
2000-01-02  0.950294  0.969193  0.177314  0.107394
2000-01-03 -0.593788  1.667980 -0.270656  0.628732
2000-01-04  0.525567  0.026901 -0.575381 -0.453548
2000-01-05  1.465294 -0.033356 -1.050563  0.376754
2000-01-06 -0.917791 -0.423718  1.398832  0.255521
2000-01-07  0.208985 -0.887049  1.398034 -0.585589
2000-01-08 -1.304846  0.737691  1.013431  0.195951
```

Basics

- You may find this useful for applying a transform (in-place) to a subset of the columns.

```
>>> df['B'] = df['B']*3
>>> df
                   A         B         C         D
2000-01-01 -1.594965 -2.397598  0.724898 -2.719031
2000-01-02  0.950294  2.907578  0.177314  0.107394
2000-01-03 -0.593788  5.003939 -0.270656  0.628732
2000-01-04  0.525567  0.080703 -0.575381 -0.453548
2000-01-05  1.465294 -0.100069 -1.050563  0.376754
2000-01-06 -0.917791 -1.271155  1.398832  0.255521
2000-01-07  0.208985 -2.661148  1.398034 -0.585589
2000-01-08 -1.304846  2.213073  1.013431  0.195951
```

- **This will not work with .loc or .iloc**

Attribute access

- You may access an index on a Series or column on a DataFrame directly as an attribute:

```
>>> df['A']
2000-01-01   -1.594965
2000-01-02    0.950294
2000-01-03   -0.593788
2000-01-04    0.525567
2000-01-05    1.465294
2000-01-06   -0.917791
2000-01-07    0.208985
2000-01-08   -1.304846
Freq: D, Name: A, dtype:
float64
```

```
>>> df.A
2000-01-01   -1.594965
2000-01-02    0.950294
2000-01-03   -0.593788
2000-01-04    0.525567
2000-01-05    1.465294
2000-01-06   -0.917791
2000-01-07    0.208985
2000-01-08   -1.304846
Freq: D, Name: A, dtype:
float64
```

Series: Slicing ranges with []

- The best way is with .iloc
- With Series, the syntax works exactly as with an ndarray, returning a slice of the values and the corresponding labels

```
>>> s[:5]
2000-01-01   -0.373138
2000-01-02    0.934918
2000-01-03   -1.571071
2000-01-04    2.369529
2000-01-05   -0.769576
Freq: D, Name: A, dtype: float64
>>> s[::2]
2000-01-01   -0.373138
2000-01-03   -1.571071
2000-01-05   -0.769576
2000-01-07   -0.391091
Freq: 2D, Name: A, dtype: float64
```

```
>>> s[::-1]
2000-01-08    0.458264
2000-01-07   -0.391091
2000-01-06    0.724399
2000-01-05   -0.769576
2000-01-04    2.369529
2000-01-03   -1.571071
2000-01-02    0.934918
2000-01-01   -0.373138
Freq: -1D, Name: A, dtype: float64
```

DataFrame: Slicing ranges with []

- With DataFrame, slicing inside of [] slices the **rows**. This is provided largely as a convenience since it is such a common operation.

```
>>> df[:3]
                   A         B         C         D
2000-01-01 -1.594965 -2.397598  0.724898 -2.719031
2000-01-02  0.950294  2.907578  0.177314  0.107394
2000-01-03 -0.593788  5.003939 -0.270656  0.628732
>>> df[::-1]
                   A         B         C         D
2000-01-08 -1.304846  2.213073  1.013431  0.195951
2000-01-07  0.208985 -2.661148  1.398034 -0.585589
2000-01-06 -0.917791 -1.271155  1.398832  0.255521
2000-01-05  1.465294 -0.100069 -1.050563  0.376754
2000-01-04  0.525567  0.080703 -0.575381 -0.453548
2000-01-03 -0.593788  5.003939 -0.270656  0.628732
2000-01-02  0.950294  2.907578  0.177314  0.107394
2000-01-01 -1.594965 -2.397598  0.724898 -2.719031
```

.loc: Selection by label

- pandas provides a suite of methods in order to have purely label based indexing. This is a strict inclusion based protocol. Every label asked for must be in the index, or a KeyError will be raised. When slicing, both the start bound AND the stop bound are included, if present in the index. Integers are valid labels, but they refer to the label and not the position.
- The .loc attribute is the primary access method. The following are valid inputs:
  - A single label, e.g. 5 or 'a' (Note that 5 is interpreted as a label of the index. This use is not an integer position along the index.).
  - A list or array of labels ['a', 'b', 'c'].
  - A slice object with labels 'a':'f' (Note that contrary to usual python slices, both the start and the stop are included, when present in the index! See Slicing with labels.
  - A boolean array.
  - A callable (The callable must be a function with one argument (the calling Series or DataFrame) that returns valid output for indexing).

With Series

```
>>> s1 = pd.Series(np.random.randn(6), index=list('abcdef'))
>>> s1
a   -0.576626
b    0.106531
c   -1.779471
d    0.836553
e    1.938956
f    0.197919
dtype: float64
>>> s1.loc['c':]
c   -1.779471
d    0.836553
e    1.938956
f    0.197919
dtype: float64
>>> s1.loc['c']
-1.7794711037017745
```

With DataFrame

```
>>> df1 = pd.DataFrame(np.random.randn(6, 4), index=list('abcdef'),
columns=list('ABCD'))
>>> df1
          A         B         C         D
a -0.386263 -0.219816  1.325287  0.566064
b  0.739423  0.771479 -1.763353  0.843234
c  0.185100 -0.317535 -0.083179 -1.699460
d -0.321289  0.258096 -1.083641 -0.244333
e  0.082513 -1.097029  0.518817  0.118281
f -0.241266 -1.224311  0.946267 -1.570834
>>> df1.loc[['a', 'b', 'd'], :]
          A         B         C         D
a -0.386263 -0.219816  1.325287  0.566064
b  0.739423  0.771479 -1.763353  0.843234
d -0.321289  0.258096 -1.083641 -0.244333
```

With DataFrame

- Label slices

```
>>> df1.loc['d':, 'A':'C']
          A         B         C
d -0.321289  0.258096 -1.083641
e  0.082513 -1.097029  0.518817
f -0.241266 -1.224311  0.946267
```

- For getting a cross section using a label (equivalent to df.xs('a')):

```
>>> df1.loc['a']
A   -0.386263
B   -0.219816
C    1.325287
D    0.566064
Name: a, dtype: float64
```

With DataFrame

- For getting values with a boolean arrays

```
>>> df1.loc['a'] > 0
A    False
B    False
C     True
D     True
Name: a, dtype: bool
>>> df1.loc[:,
df1.loc['a'] > 0]
          C         D
a  1.325287  0.566064
b -1.763353  0.843234
c -0.083179 -1.699460
d -1.083641 -0.244333
e  0.518817  0.118281
f  0.946267 -1.570834
```

With DataFrame

- For getting a value explicitly (equivalent to deprecated df.get_value('a','A')):

```
>>> df1.loc['a','A']
-0.38626320463534375
>>> df1.at['a','A']
-0.38626320463534375
```

Slicing with labels

- If both the start and the stop labels are present in the index, then elements located between the two (including them) are returned:

```
>>> s = pd.Series(list('abcde'), index=[0, 3, 2, 5, 4])
>>> s.loc[3:5]
3    b
2    c
5    d
dtype: object
```

Slicing with labels

- If at least one of the two is absent, but the index is sorted, and can be compared against start and stop labels, then slicing will still work as expected, by selecting labels which rank between the two:

```
>>> s.sort_index()
0    a
2    c
3    b
4    e
5    d
dtype: object
>>>
s.sort_index().loc[1:6]
2    c
3    b
4    e
5    d
dtype: object
```

.iloc: Selection by position

- Pandas provides a suite of methods in order to get purely integer based indexing. The semantics follow closely Python and NumPy slicing. These are 0-based indexing. When slicing, the start bound is included, while the upper bound is excluded. Trying to use a non-integer, even a valid label will raise an IndexError.
- The .iloc attribute is the primary access method. The following are valid inputs:
  - An integer e.g. 5.
  - A list or array of integers [4, 3, 0].
  - A slice object with ints 1:7.
  - A boolean array.
  - A callable, see Selection By Callable.

With Series

```
>>> s1 = pd.Series(np.random.randn(5), index=list(range(0, 10, 2)))
>>> s1
0    0.497642
2    0.658404
4   -0.882054
6    0.760452
8   -0.195293
dtype: float64
>>> s1.iloc[:3]
0    0.497642
2    0.658404
4   -0.882054
dtype: float64
>>> s1.iloc[3]
0.7604517699461874
```

With DataFrame

```
>>> df1
          A         B         C         D
a -0.386263 -0.219816  1.325287  0.566064
b  0.739423  0.771479 -1.763353  0.843234
c  0.185100 -0.317535 -0.083179 -1.699460
d -0.321289  0.258096 -1.083641 -0.244333
e  0.082513 -1.097029  0.518817  0.118281
f -0.241266 -1.224311  0.946267 -1.570834
>>> df1.iloc[:3]
          A         B         C         D
a -0.386263 -0.219816  1.325287  0.566064
b  0.739423  0.771479 -1.763353  0.843234
c  0.185100 -0.317535 -0.083179 -1.699460
>>> df1.iloc[1:5, 2:4]
          C         D
b -1.763353  0.843234
c -0.083179 -1.699460
d -1.083641 -0.244333
e  0.518817  0.118281
```

With DataFrame

- Select via integer slicing:

```
>>> df1.iloc[:3]
          A         B         C         D
a -0.386263 -0.219816  1.325287  0.566064
b  0.739423  0.771479 -1.763353  0.843234
c  0.185100 -0.317535 -0.083179 -1.699460
>>> df1.iloc[1:5, 2:4]
          C         D
b -1.763353  0.843234
c -0.083179 -1.699460
d -1.083641 -0.244333
e  0.518817  0.118281
```

With DataFrame

- Select via integer list:

```
>>> df1.iloc[[1, 3, 5], [1, 3]]
          B         D
b  0.771479  0.843234
d  0.258096 -0.244333
f -1.224311 -1.570834
>>> df1.iloc[1:3, :]
          A         B         C         D
b  0.739423  0.771479 -1.763353  0.843234
c  0.185100 -0.317535 -0.083179 -1.699460
>>> df1.iloc[:, 1:3]
          B         C
a -0.219816  1.325287
b  0.771479 -1.763353
c -0.317535 -0.083179
d  0.258096 -1.083641
e -1.097029  0.518817
f -1.224311  0.946267
>>> df1.iloc[1, 1]
0.7714790673524865
```

Other useful options

- Selection by Callable
- Selecting random samples
- Boolean indexing
- Selection with where()
- Selection with mask()
- Selection with query()
- Selection with lookup()
- Set and reset index

Missing data

- Pandas primarily uses the value np.nan to represent missing data. It is by default not included in computations.
- To drop any rows that have missing data.

```
>>> df1 = df.reindex(index=dates[0:4],
columns=list(df.columns) + ['E'])
>>> df1.loc[dates[0]:dates[1], 'E'] = 1
>>> df1
                   A         B         C         D    E
2013-01-01 -1.869249 -1.753779 -2.770687 -0.554581  1.0
2013-01-02  1.041522  0.296168  0.105374  1.347235  1.0
2013-01-03 -1.728628  0.171945 -1.122000 -0.692276  NaN
2013-01-04  0.260960  0.619587  0.448398  0.383778  NaN
>>> df1.dropna(how='any')
                   A         B         C         D    E
2013-01-01 -1.869249 -1.753779 -2.770687 -0.554581  1.0
2013-01-02  1.041522  0.296168  0.105374  1.347235  1.0
```

Missing data

- Filling missing data

```
>>> df1.fillna(value=5)
                   A         B         C         D    E
2013-01-01 -1.869249 -1.753779 -2.770687 -0.554581  1.0
2013-01-02  1.041522  0.296168  0.105374  1.347235  1.0
2013-01-03 -1.728628  0.171945 -1.122000 -0.692276  5.0
2013-01-04  0.260960  0.619587  0.448398  0.383778  5.0
```

- To get the boolean mask where values are nan

```
>>> pd.isna(df1)
                A      B      C      D      E
2013-01-01  False  False  False  False  False
2013-01-02  False  False  False  False  False
2013-01-03  False  False  False  False   True
2013-01-04  False  False  False  False   True
```

Operations

- Stats
  - Binary and other operations
- Histograms

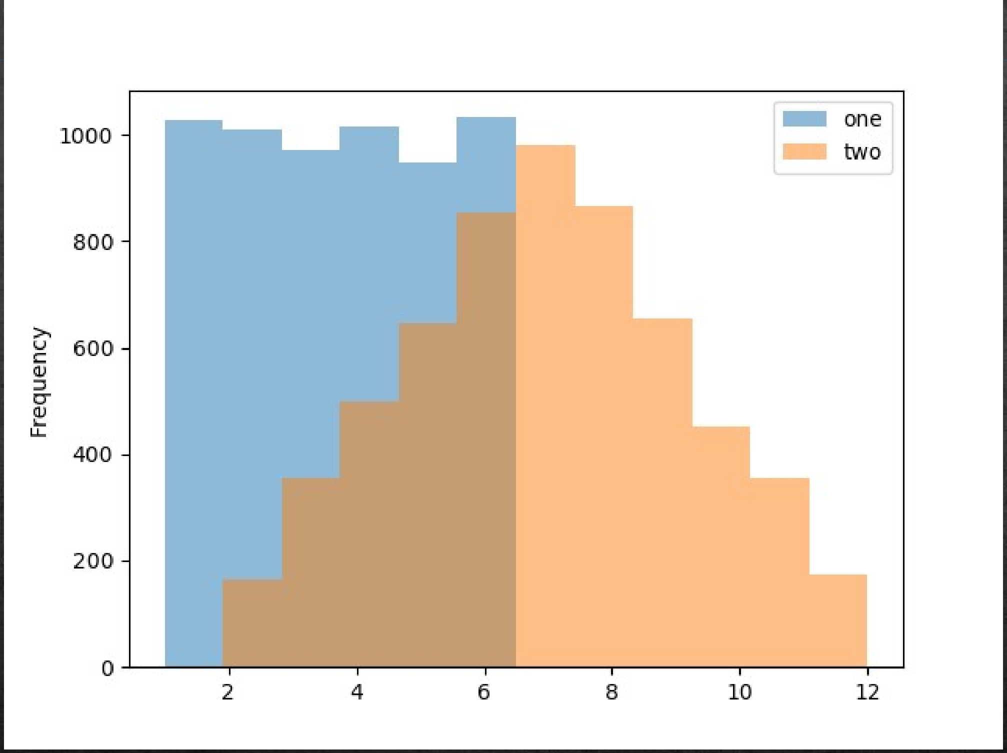
```
>>> s = pd.Series(np.random.randint(0, 7, size=10))
>>> s
0    0
1    2
2    4
3    6
4    6
5    1
6    2
7    0
8    5
9    2
dtype: int64
>>> s.value_counts()
2    3
6    2
0    2
5    1
4    1
1    1
dtype: int64
```

Merging, grouping and reshaping

- Concatenating pandas objects together with concat()
- SQL style merges with merge()
- Append rows to a DataFrame with append()
- By "group by" we are referring to a process involving one or more of the following steps:
  - Splitting the data into groups based on some criteria
  - Applying a function to each group independently
  - Combining the results into a data structure
  - See grouping section of manual
- See the sections on Hierarchical Indexing and Reshaping.

Plot histograms

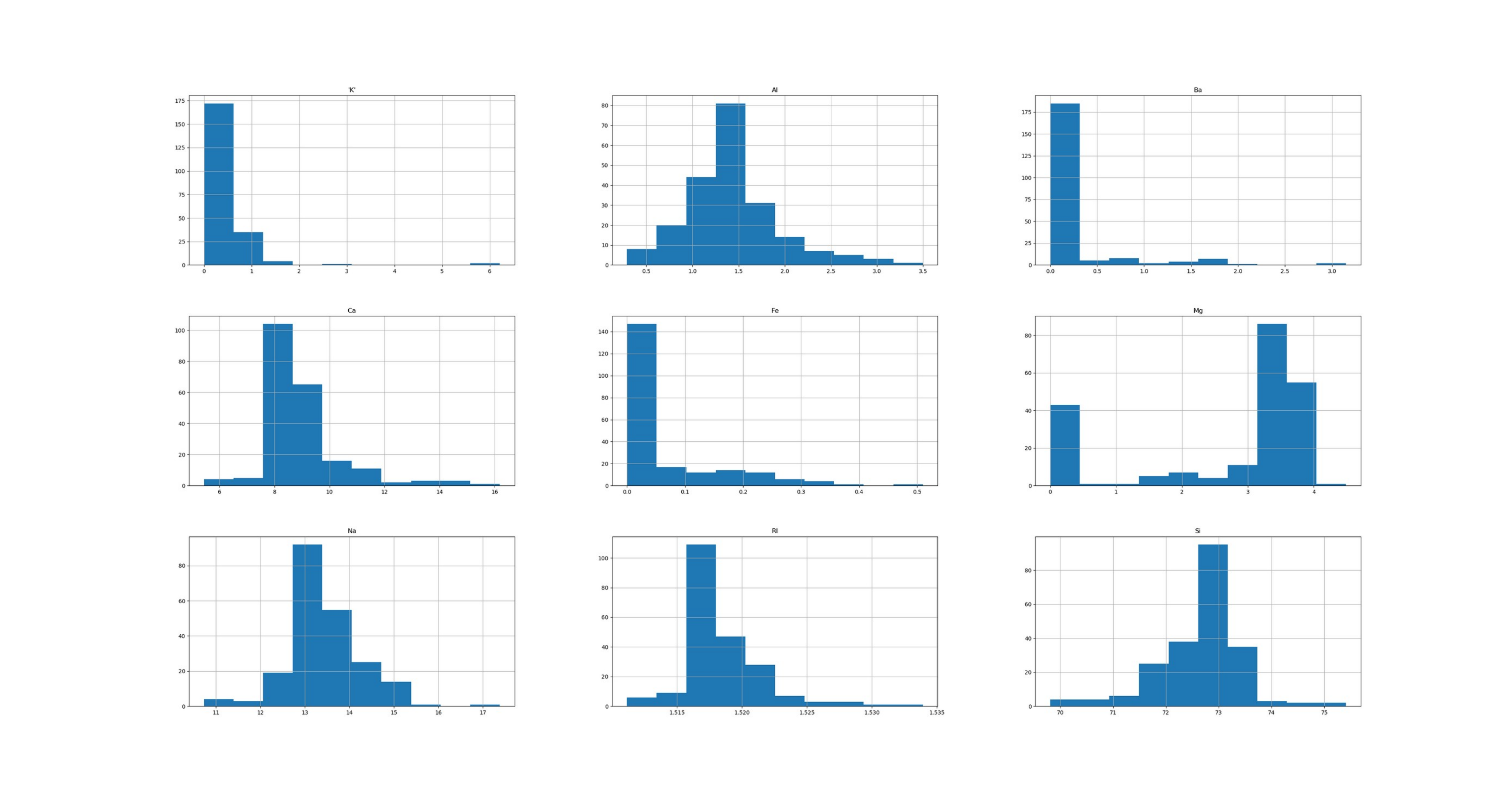
```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> import pandas as pd
>>> df = pd.DataFrame(np.random.randint(1, 7, 6000), columns = ['one'])
>>> df['two'] = df['one'] + np.random.randint(1, 7, 6000)
>>> df.plot.hist(bins=12, alpha=0.5)
<matplotlib.axes._subplots.AxesSubplot object at 0x7f7f948300d0>
>>> plt.show()
```



Plot histograms

- All attributes at the same time

```
>>> df.hist()
>>> plt.show()
```



Plot histograms

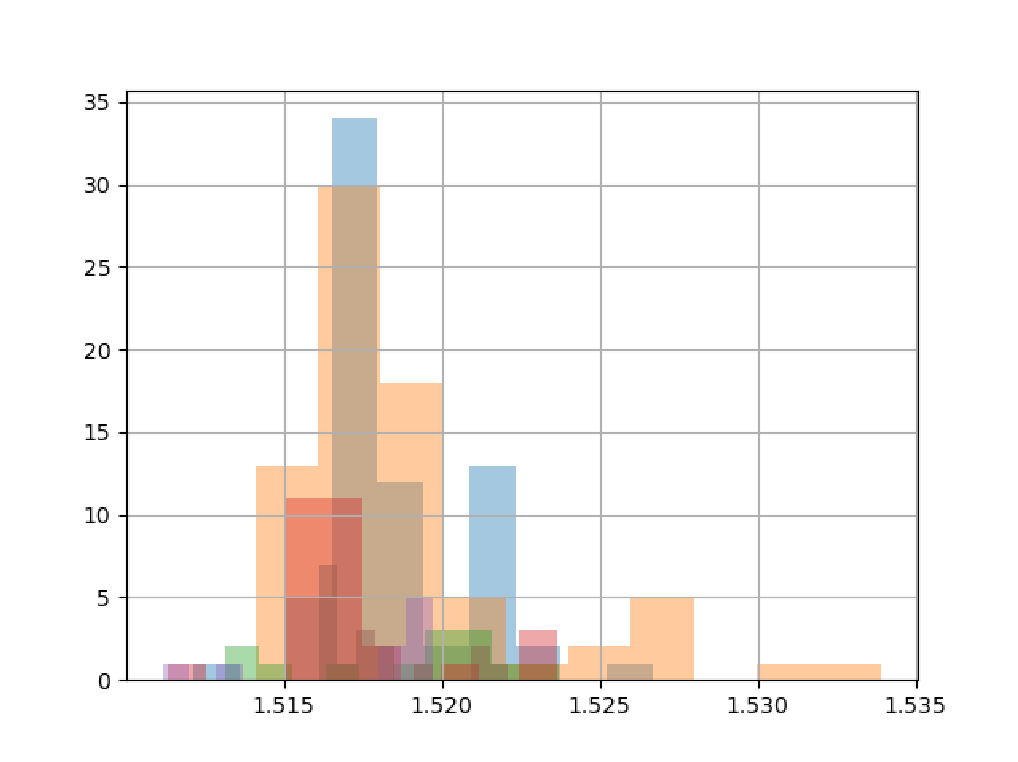
- Using class labels
- Glass dataset

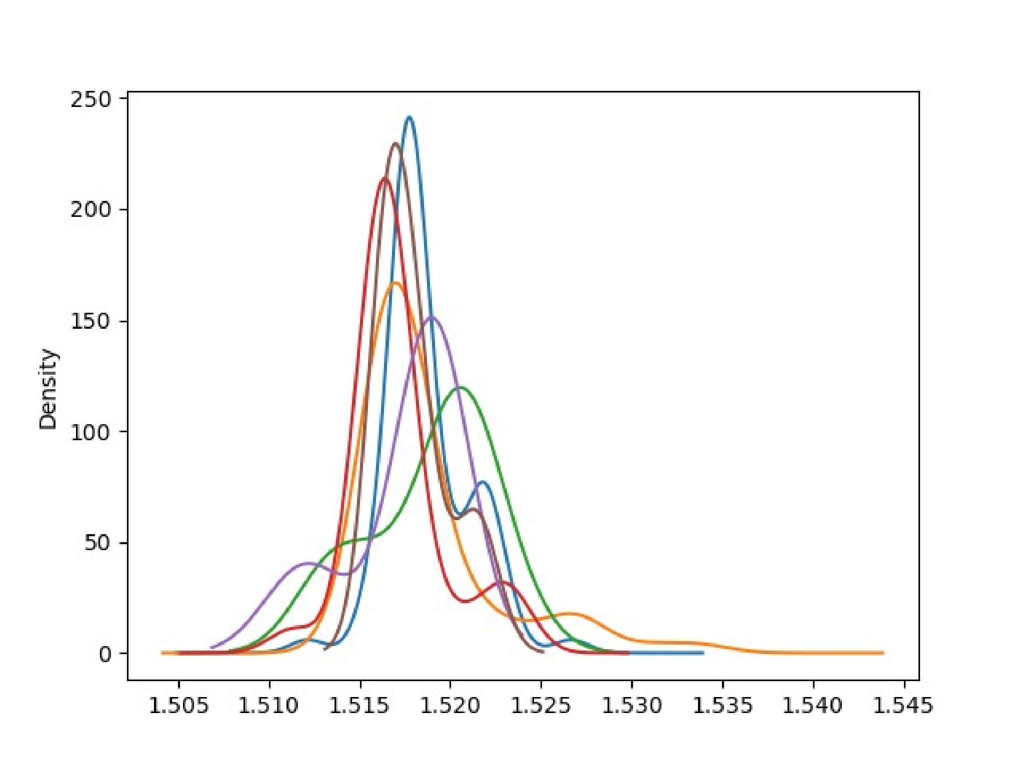
```
>>> df.groupby('Type').RI.hist(alpha=0.4)
Type
b'build wind float'          AxesSubplot(0.125,0.11;0.775x0.77)
b'build wind non-float'      AxesSubplot(0.125,0.11;0.775x0.77)
b'containers'                AxesSubplot(0.125,0.11;0.775x0.77)
b'headlamps'                 AxesSubplot(0.125,0.11;0.775x0.77)
b'tableware'                 AxesSubplot(0.125,0.11;0.775x0.77)
b'vehic wind float'          AxesSubplot(0.125,0.11;0.775x0.77)
Name: RI, dtype: object
>>> plt.show()

# Distributions
>>> df.groupby('Type').RI.plot(kind='kde')
```

Plot histograms

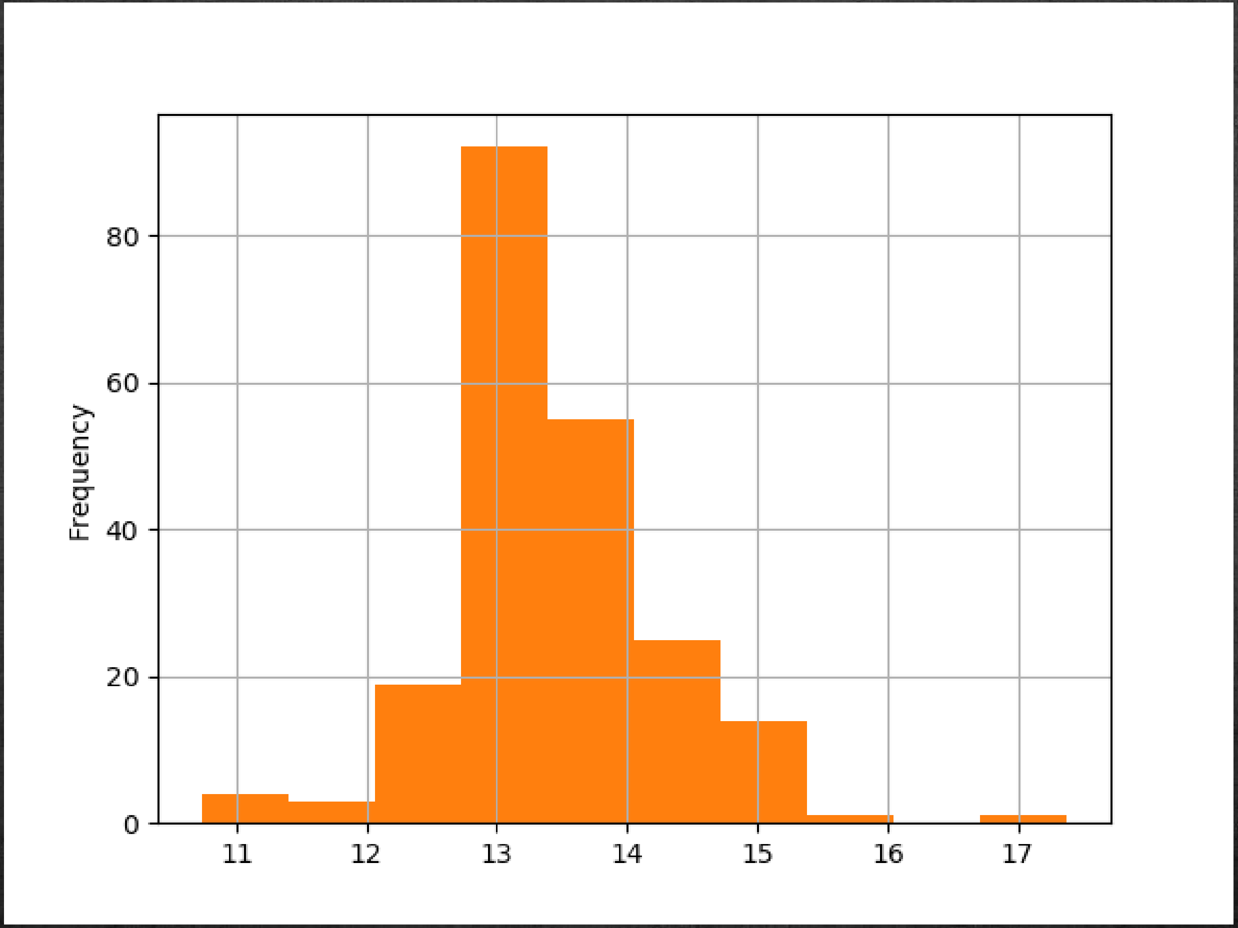
- Histogram and KDE





Plot series

```
>>> df['Na'].hist()
```

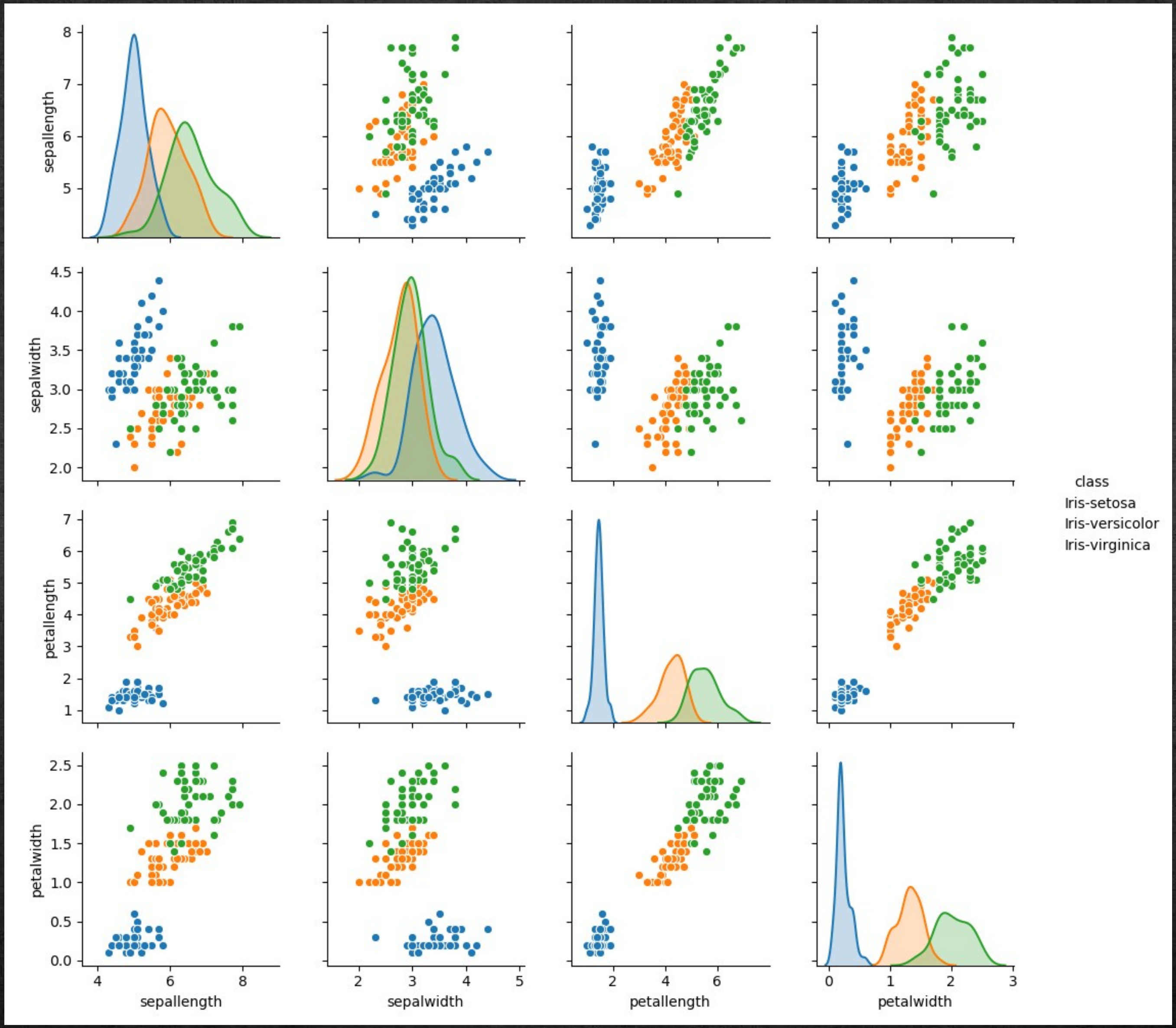


Scatter plot

- Using seaborn (only python3)
- Seaborn is a high level plotting library

```
from scipy.io import arff
import seaborn as sns
import pandas as pd
>>> data = arff.loadarff('iris.arff')
>>> df = pd.DataFrame(data[0])
>>> sns.pairplot(data=df, hue='class')
<seaborn.axisgrid.PairGrid object at 0x7f7f8d48e4d0>
>>> plt.show()
```

Scatter plot

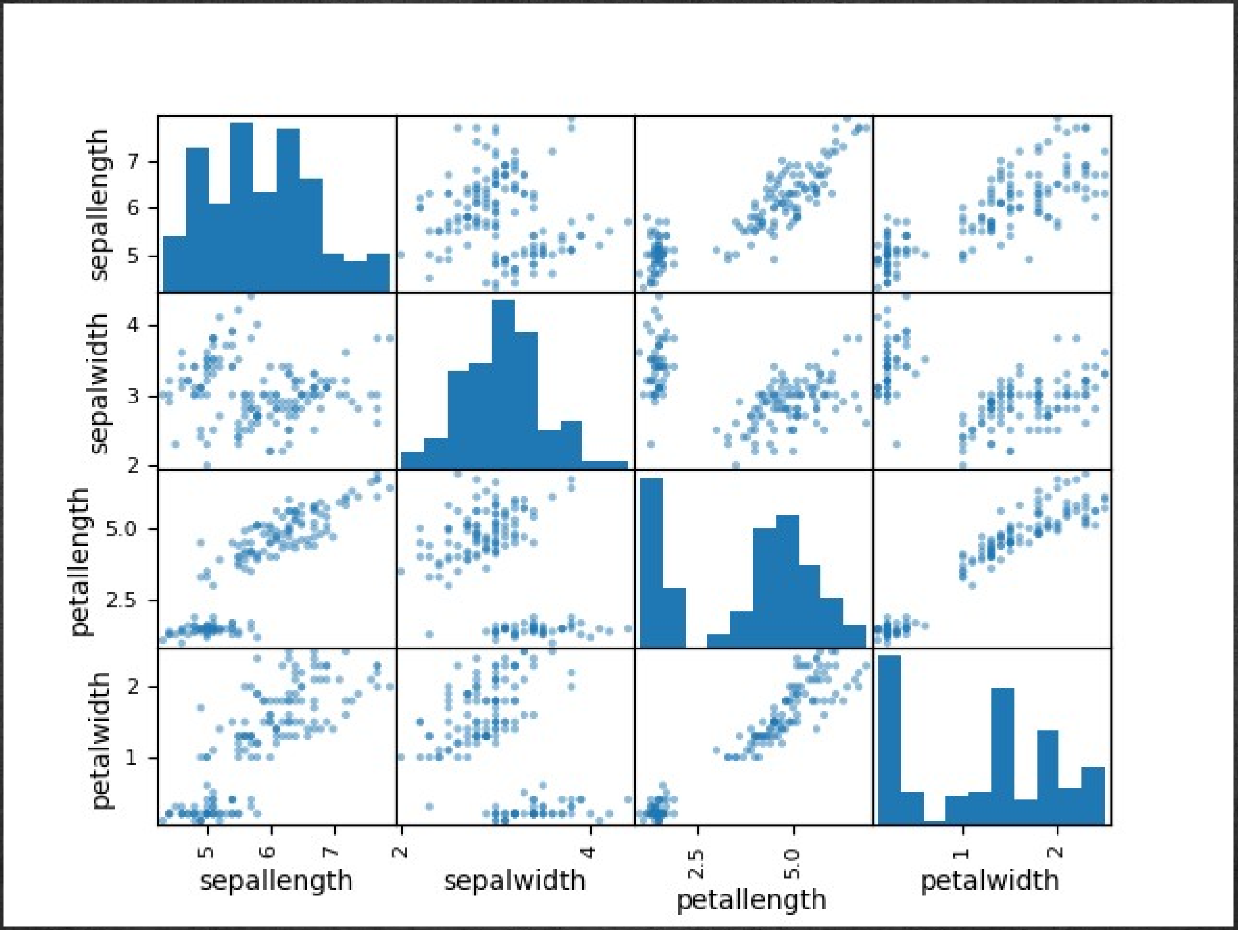


Scatter plot

- Using Pandas
  - It does not use class labels

```
>>> from scipy.io import arff
>>> import pandas as pd
>>> data = arff.loadarff('iris.arff')
>>> df = pd.DataFrame(data[0])
>>> pd.plotting.scatter_matrix(df)
>>> import matplotlib.pyplot as plt
>>> plt.show()
```

Scatter plot

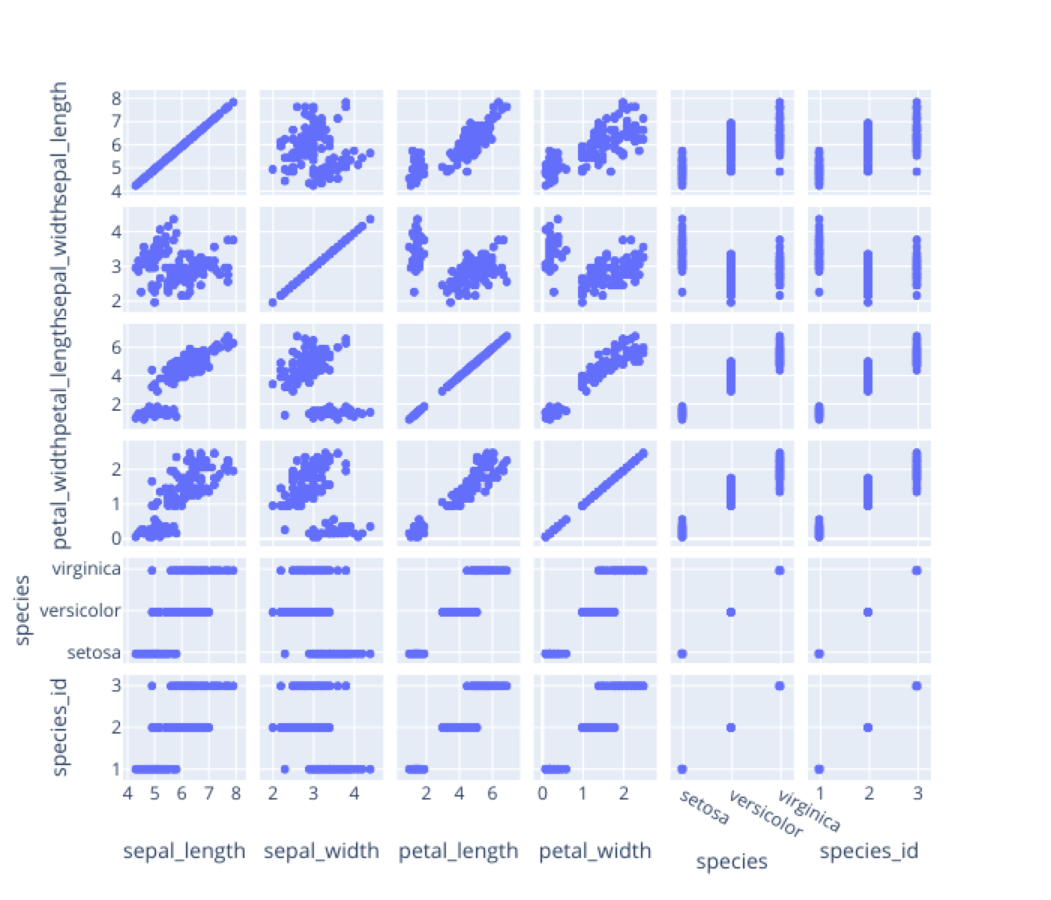


Scatter plot

- Using plotly

```
>>> import plotly.express as px
>>> iris = px.data.iris()
>>> fig = px.scatter_matrix(iris)
>>> fig.show()
```

Scatter plot

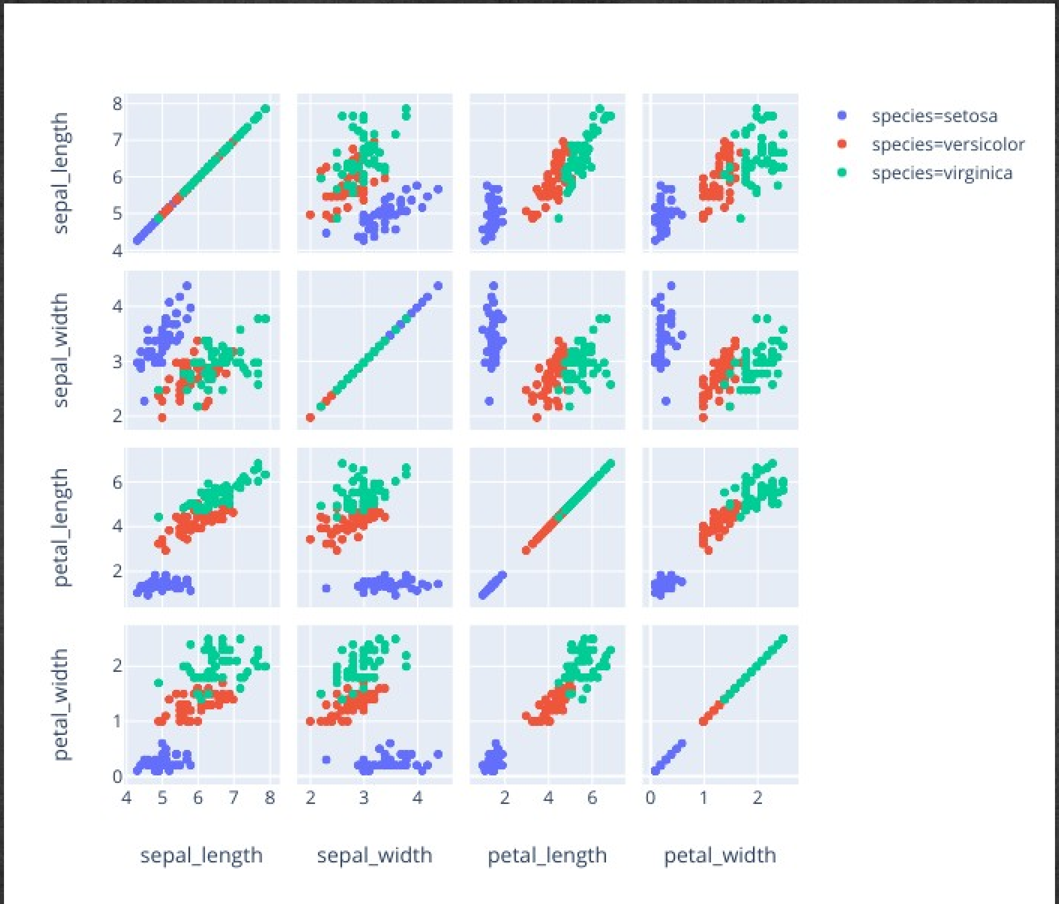


Scatter plot

- Using Plotly and categories

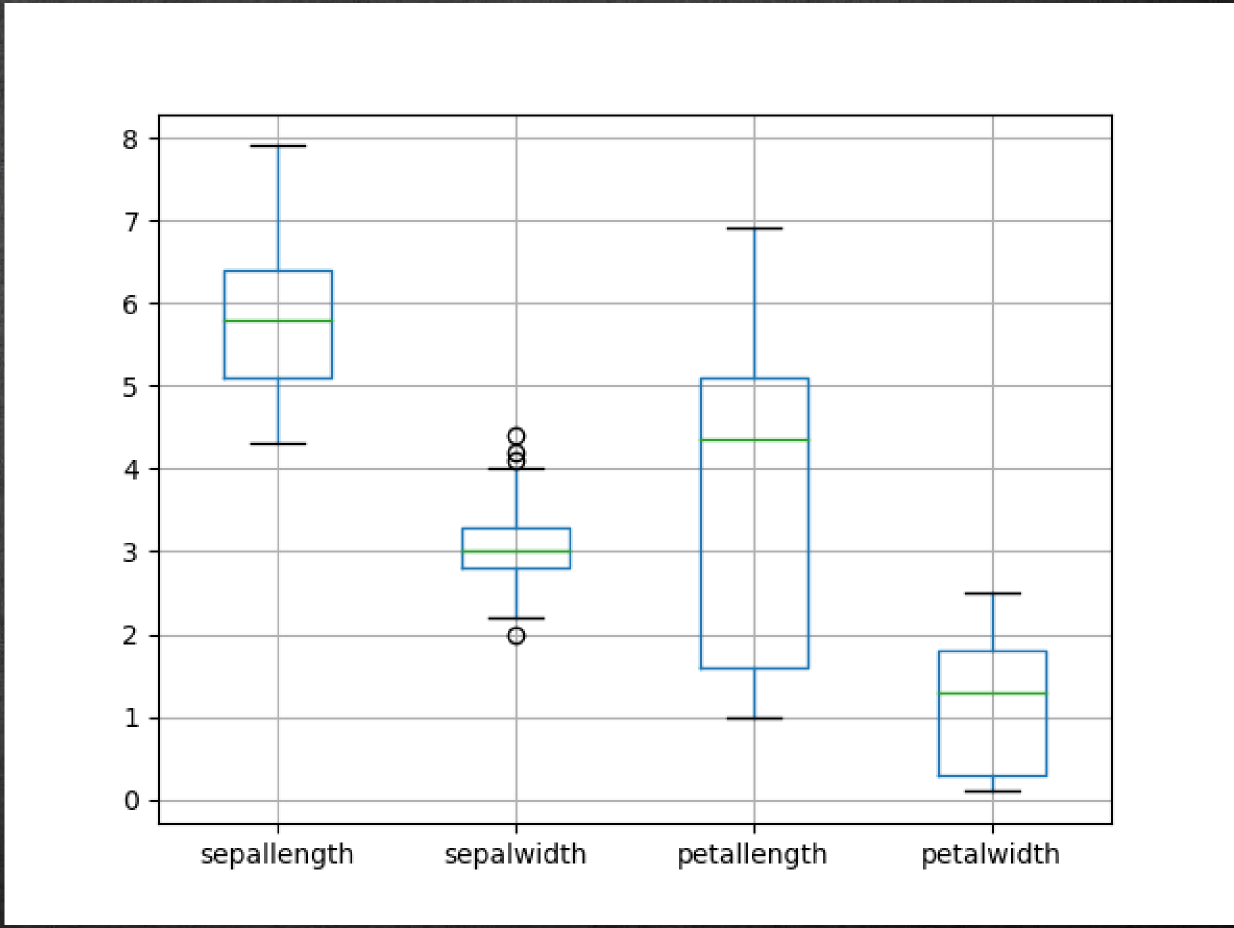
```
>>> import plotly.express as px
>>> iris = px.data.iris()
>>> fig = px.scatter_matrix(iris,
...     dimensions=["sepal_width", "sepal_length", "petal_width", "petal_length"],
...     color="species")
>>> fig.show()
```

Scatter plot



Box plots

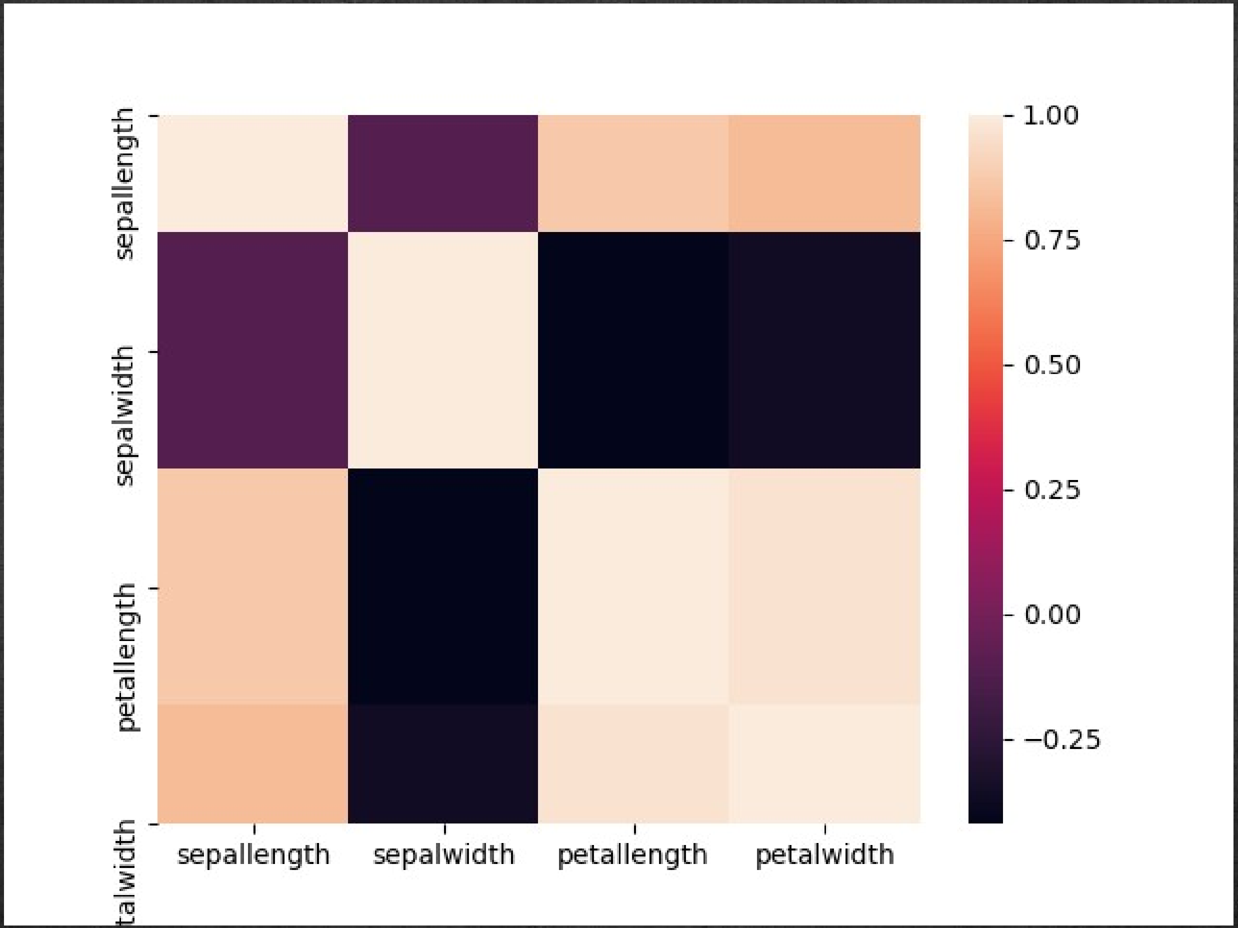
```
>>> import matplotlib.pyplot as plt
>>> from scipy.io import arff
>>> import pandas as pd
>>> data = arff.loadarff('iris.arff')
>>> df = pd.DataFrame(data[0])
>>> df.boxplot()
>>> plt.show()
```



Correlation matrix plots

```
>>> import seaborn as sns
>>> import pandas as pd
>>> from scipy.io import arff
>>> import matplotlib.pyplot as plt
>>> data = arff.loadarff('iris.arff')
>>> df = pd.DataFrame(data[0])
>>> corr = df.corr()
>>> sns.heatmap(corr, xticklabels=corr.columns,yticklabels=corr.columns)
qt5ct: using qt5ct plugin
<matplotlib.axes._subplots.AxesSubplot object at 0x7f6c60fc8550>
>>> plt.show()
```

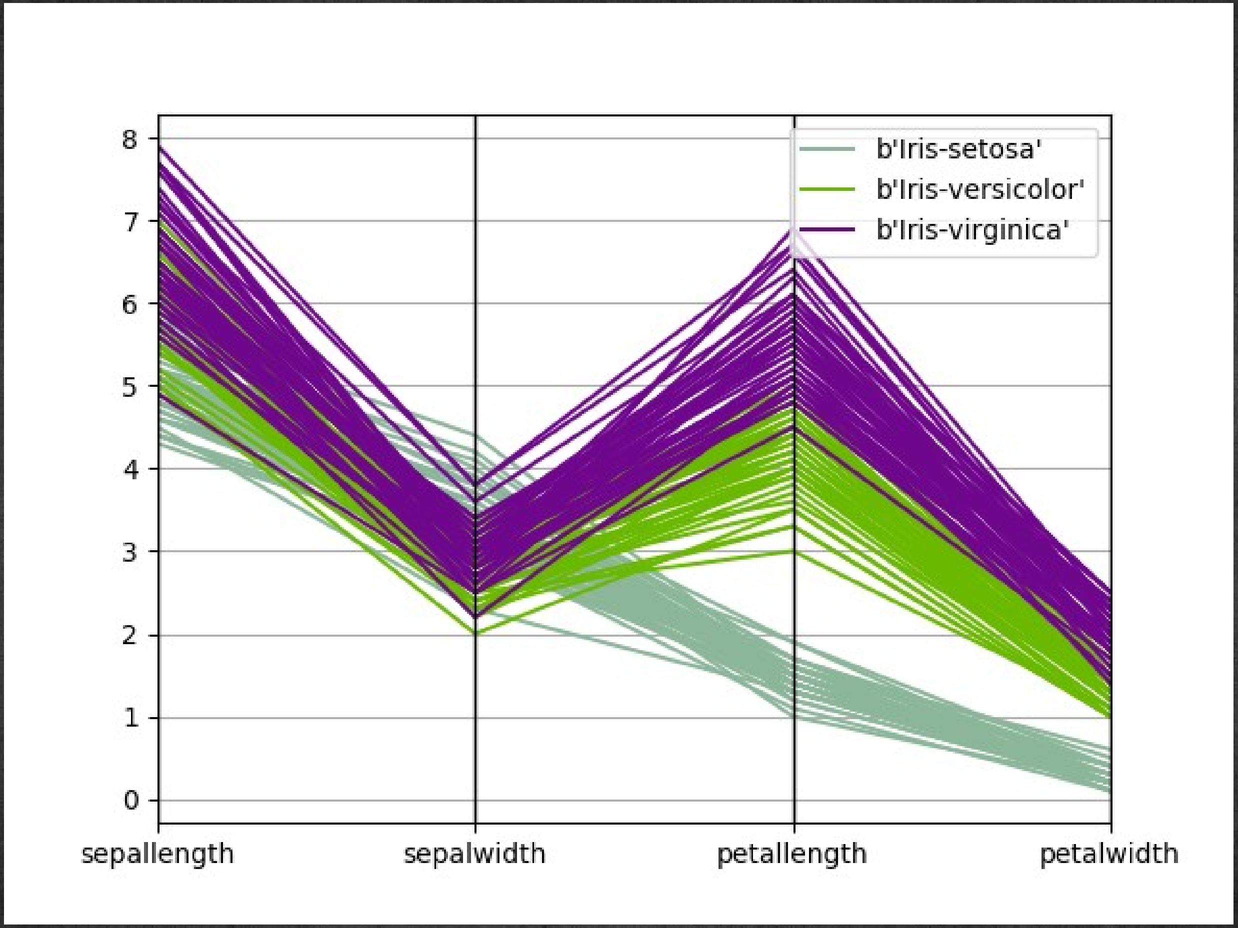
Correlation matrix plots



Parallel coordinates plots

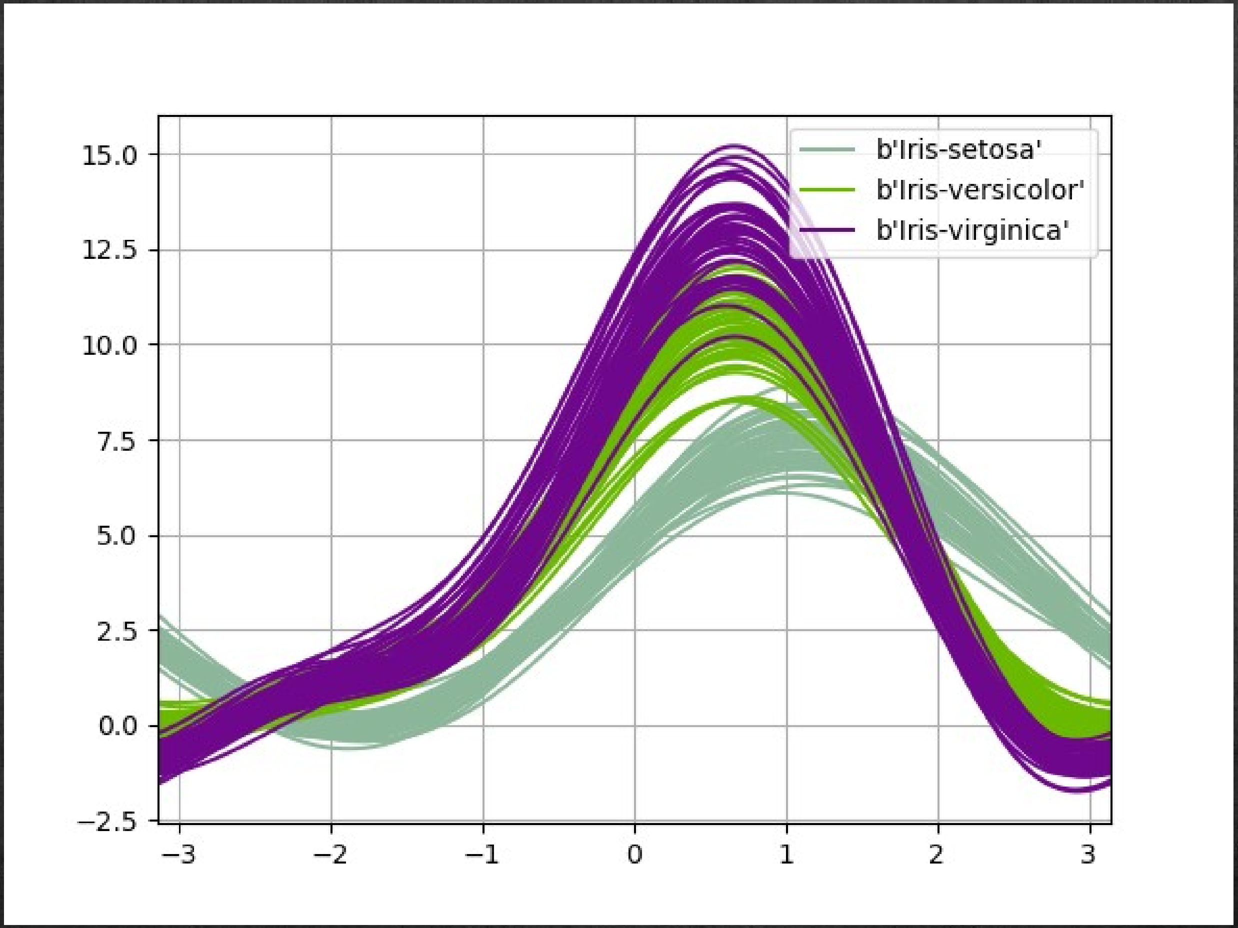
```
>>> import pandas as pd
>>> from scipy.io import arff
>>> import matplotlib.pyplot as plt
>>> data = arff.loadarff('iris.arff')
>>> df = pd.DataFrame(data[0])
>>> pd.plotting.parallel_coordinates(df, 'class')
<matplotlib.axes._subplots.AxesSubplot object at 0x7f6c6062c790>
>>> plt.show()
```

Parallel coordinates



Andrews curves

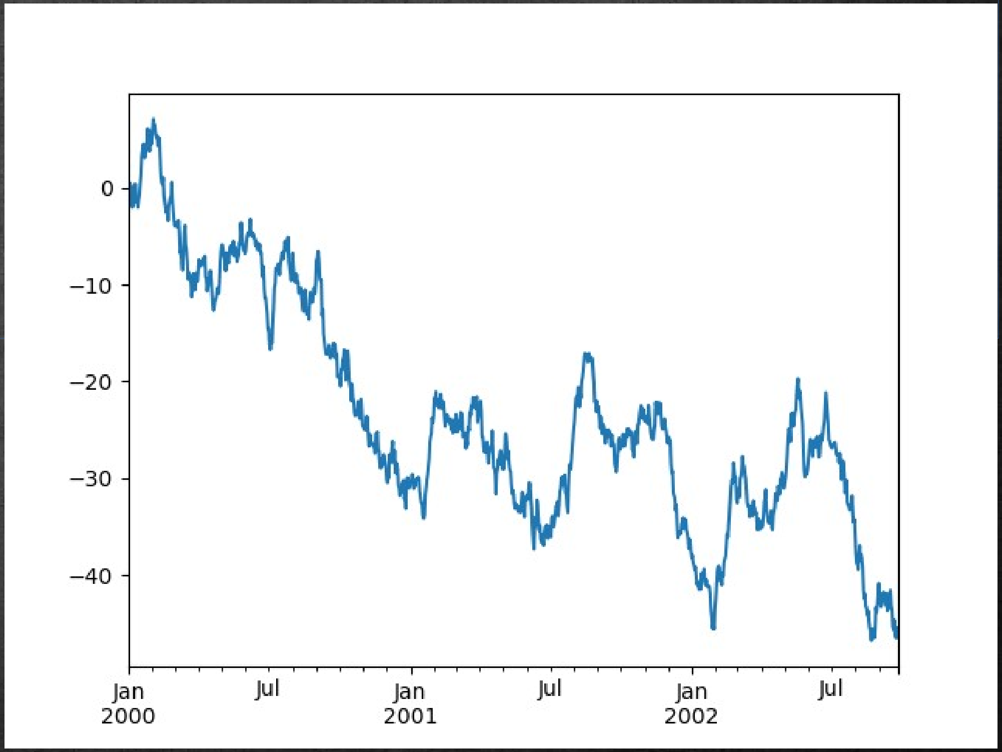
```
pd.plotting.andrews_curves(df, 'class')
```



Plotting

- Using Matplotlib

```
>>> import numpy as np
>>> import pandas as pd
>>> import matplotlib.pyplot as mlp
>>> ts = pd.Series(np.random.randn(1000), index=pd.date_range('1/1/2000',
periods=1000))
>>> ts = ts.cumsum()
>>> ts.plot()
<matplotlib.axes._subplots.AxesSubplot
  object at 0x7f546db5f590>
>>> plt.show()
```



Plotting

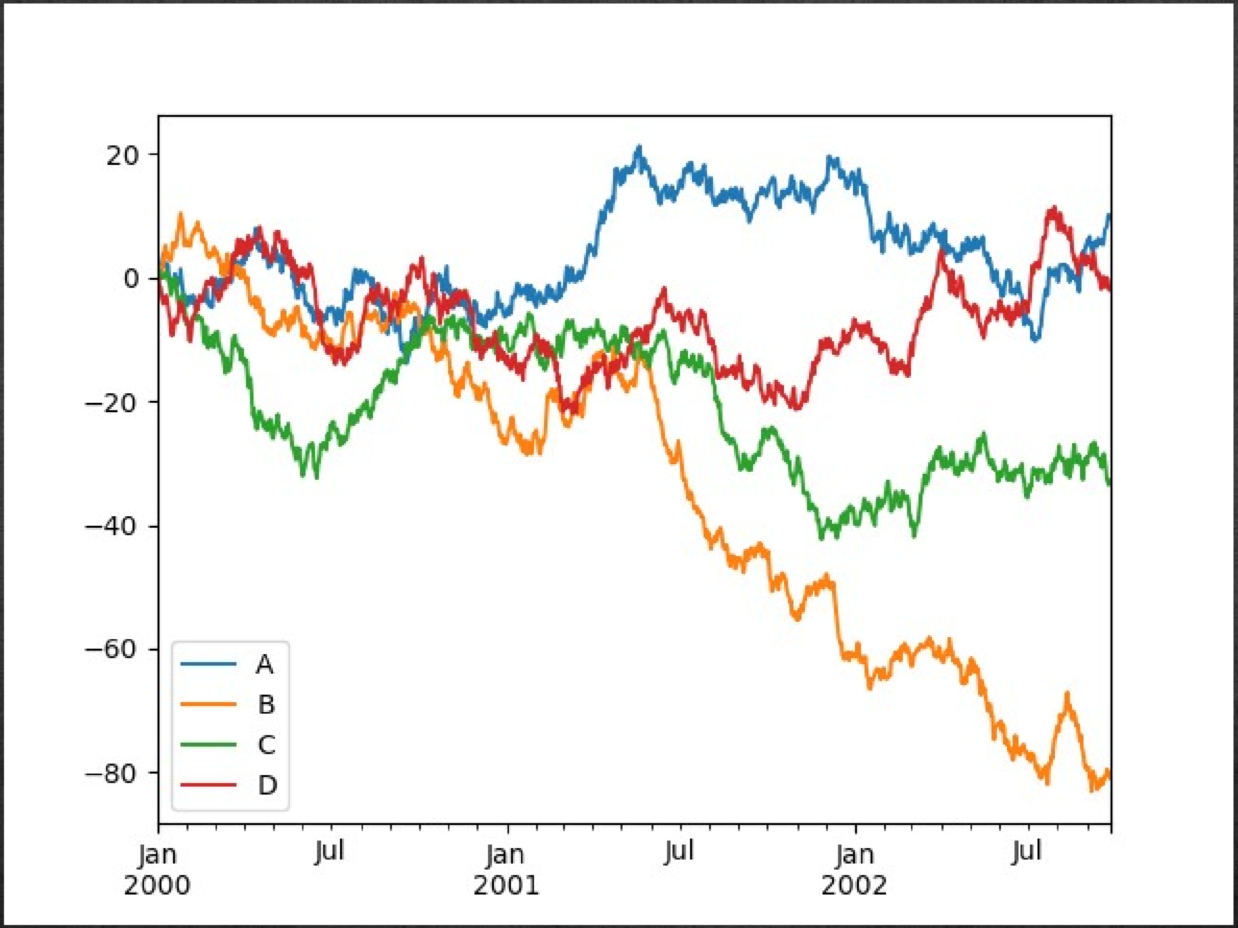
- On a DataFrame, the plot() method is a convenience to plot all of the columns with labels

```
>>> df = pd.DataFrame(np.random.randn(1000, 4), index=ts.index, columns=['A', 'B', 'C', 'D'])
>>> df = df.cumsum()
>>> df
                    A          B          C         D
2000-01-01   0.838655   0.300472  -0.306635 -0.355511
2000-01-02   1.782371   0.002430   0.874933  0.094413
2000-01-03   0.862820   0.062380   1.103303 -1.468325
2000-01-04   1.105240   0.632801   0.717499 -1.514284
2000-01-05   1.852683   1.703786   0.019095 -2.489315
...               ...        ...        ...       ...
2002-09-22   8.167895 -79.576736 -32.644755  0.057446
2002-09-23   9.197353 -79.706547 -32.712248 -0.265785
2002-09-24  10.218521 -80.077821 -33.530171 -1.410210
2002-09-25   9.888908 -80.680651 -33.197733 -2.044380
2002-09-26   9.718245 -81.017158 -32.848111 -1.713411

[1000 rows x 4 columns]
>>> plt.figure()
<Figure size 640x480 with 0 Axes>
>>> df.plot()
<matplotlib.axes._subplots.AxesSubplot object at 0x7f54678ea310>
>>> plt.legend(loc='best')
<matplotlib.legend.Legend object at 0x7f5467c9f250>
>>> plt.show()
```

Plotting

- plt.show()



Reading data from files: CSV

- Writting

```
>>> df.to_csv('foo.csv')
```

- Reading

```
>>> pd.read_csv('foo.csv')
     Unnamed: 0          A          B          C         D
0    2000-01-01   0.838655   0.300472  -0.306635 -0.355511
1    2000-01-02   1.782371   0.002430   0.874933  0.094413
2    2000-01-03   0.862820   0.062380   1.103303 -1.468325
3    2000-01-04   1.105240   0.632801   0.717499 -1.514284
4    2000-01-05   1.852683   1.703786   0.019095 -2.489315
..          ...        ...        ...        ...       ...
995  2002-09-22   8.167895 -79.576736 -32.644755  0.057446
996  2002-09-23   9.197353 -79.706547 -32.712248 -0.265785
997  2002-09-24  10.218521 -80.077821 -33.530171 -1.410210
998  2002-09-25   9.888908 -80.680651 -33.197733 -2.044380
999  2002-09-26   9.718245 -81.017158 -32.848111 -1.713411

[1000 rows x 5 columns]
```

Reading and writting: Excel

- Excel files

```
>>> df.to_excel('foo.xlsx')
>>> pd.read_excel('foo.xlsx', 'Sheet1', index_col=None, na_values=['NA'])
    Unnamed: 0          A          B          C         D
0   2000-01-01   0.838655   0.300472  -0.306635 -0.355511
1   2000-01-02   1.782371   0.002430   0.874933  0.094413
2   2000-01-03   0.862820   0.062380   1.103303 -1.468325
3   2000-01-04   1.105240   0.632801   0.717499 -1.514284
4   2000-01-05   1.852683   1.703786   0.019095 -2.489315
..         ...        ...        ...        ...       ...
995 2002-09-22   8.167895 -79.576736 -32.644755  0.057446
996 2002-09-23   9.197353 -79.706547 -32.712248 -0.265785
997 2002-09-24  10.218521 -80.077821 -33.530171 -1.410210
998 2002-09-25   9.888908 -80.680651 -33.197733 -2.044380
999 2002-09-26   9.718245 -81.017158 -32.848111 -1.713411

[1000 rows x 5 columns]
```

ARFF files

- Using scipy

```
>>> from scipy.io import arff
>>> data = arff.loadarff('iris.arff')
>>> type(data)
<class 'tuple'>
>>> type(data[0])
<class 'numpy.ndarray'>
>>> type(data[1])
<class 'scipy.io.arff.arffread.MetaData'>
>>> df = pd.DataFrame(data[0])
>>> df.head()
   sepallength  sepalwidth  petallength  petalwidth            class
0          5.1         3.5          1.4         0.2  b'Iris-setosa'
1          4.9         3.0          1.4         0.2  b'Iris-setosa'
2          4.7         3.2          1.3         0.2  b'Iris-setosa'
3          4.6         3.1          1.5         0.2  b'Iris-setosa'
4          5.0         3.6          1.4         0.2  b'Iris-setosa'
```