MatchIT: Python for Data Science
Lecture: Pandas

Lecture 2: Content

- Data Manipulation with Pandas
- Pandas Objects
- Indexing and Data Selection on Pandas Objects
- Operating on Data
  - Universal Functions and Index Alignment
  - Handling Missing Data
  - Aggregation
  - Groupby
    - Filter
    - Transform
    - Apply
  - Pivot Tables

Data Manipulation with Pandas

- Pandas is a newer package built on NumPy

- Why Pandas?
  - NumPy's ndarray data structure contains essential features needed to store and manipulate data in numerical computing tasks - BUT
  - Numpy has some limitations that Pandas tries to solve, such as
    - Greater flexibility to attach labels to data
    - Simplifies task of working with missing data
    - Operations such as groupings and pivots

- Pandas Series and DataFrame objects are built on NumPy array structure to provide

Pandas Objects

- Pandas objects are an enhanced versions of NumPy structured arrays
  - Rows and columns are identified with labels rather than simple integer indices.
  - Provide useful tools, methods, and functionality on top of the basic data structures
  - Understanding the basic data structures is essential in taking full advantage of Pandas package

Pandas Data Structures

- Three fundamental Pandas Data structures
  - Series
  - DataFrame
  - Index

Pandas Data Structures: Series Object

- Can be created from a list or array

```
from  scipy import *
from  matplotlib.pyplot import *

import numpy as np
import pandas as pd

data= pd.Series([0.25, 0.5, 0.75, 1.0])
data
```

- The series output, data, combines both a sequence and a sequence of indices.

```
In [2]: data
Out[2]:
0    0.25
1    0.50
2    0.75
3    1.00
dtype: float64
```

Pandas Data Structures: Series Object

- We can access easily data values:

```
In [3]: data.values
Out[3]: array([0.25, 0.5 , 0.75, 1.  ])
```

- As well as indices:

```
In [4]: data.index
Out[4]: RangeIndex(start=0, stop=4, step=1)
```

- As with NumPy array, data can be access via index

```
In [5]: data[1]
Out[5]: 0.5

In [6]: data[1:3]
Out[6]:
1    0.50
2    0.75
dtype: float64
```

- Series object similar to one-dimensional NumPy array, but Series object offers more..

Pandas Data Structures: Series Object

- Series object has explicitly defined index associated with the values:

```
In [10]: data= pd.Series([0.25, 0.5, 0.75, 1.0], index=['a', 'b','c','d'])

In [11]: data
Out[11]:
a    0.25
b    0.50
c    0.75
d    1.00
dtype: float64

In [12]: data['b']
Out[12]: 0.5
```

- Series can be thought of as specialized Python dictionary (key, value), but with more efficient implementation

Pandas Data Structures: Series Object

- Series can be created from a Python dictionary

```
In [23]: results_dict={'passed': 12,
    ...: 'passed with distinction': 5,
    ...: 'failed': 2}

In [24]: results_ser=pd.Series(results_dict)

In [25]: results_ser
Out[25]:
passed                     12
passed with distinction     5
failed                      2
dtype: int64
```

Pandas Data Structures: Data Frame Object

- DataFrame object can also be considered as generalization of NumPy array or specialization of a Python dictionary

  - DataFrame as a generalized NumPy array
    - Analog to *two-dimensional* array with *flexible row indices* and *flexible column names*

Two-dimensional array = Ordered Sequence of aligned one dimension columns

DataFrame = Ordered Sequence of aligned Series objects

aligned - share the same index*

Pandas Data Structures: Data Frame Object

- Example:

```
In [38]: results_ser
Out[38]:
passed                     12
passed with distinction     5
failed                      2
dtype: int64

In [39]: grades_dict={'passed' : 3, 'passed with distinction' : 4,'failed': 2}

In [40]: grading_results=pd.DataFrame({'results': results_ser,'grades': grades_dict})

In [41]: grading_results
Out[41]:
                         results  grades
failed                         2       2
passed                        12       3
passed with distinction        5       4

In [42]: grading_results.index
Out[42]: Index(['failed', 'passed', 'passed with distinction'], dtype='object')

In [43]: grading_results.columns
Out[43]: Index(['results', 'grades'], dtype='object')
```

Pandas Data Structures: Data Frame Object

- How to construct DataFrame Object
  - From a single Series object
  - From a list of dictionaries
  - From a dictionary of Series objects
  - From a two-dimensional NumPy array
  - From a NumPy structured array

Pandas Data Structures: Data Frame Object

Construction Examples

- From a single Series object:

```
In [45]: pd.DataFrame(results_ser, columns=['results'])
Out[45]:
                          results
passed                         12
passed with distinction         5
failed                          2
```

- From a list of dictionary

```
In [50]: data = [{'a': i, 'b': 2*i, 'c': i + 2*i}
    ...:         for i in range(3)]
    ...: pd.DataFrame(data)
Out[50]:
   a  b  c
0  0  0  0
1  1  2  3
2  2  4  6
```

Pandas Data Structures: Data Frame Object

Construction Examples

- From a dictionary of Series objects

```
In [57]: grades_dict
Out[57]: {'passed': 3, 'passed with distinction': 4, 'failed': 2}

In [58]: grades_ser=pd.Series(grades_dict)

In [59]: pd.DataFrame({'results':results_ser,'grades':grades_ser})
Out[59]:
                         results  grades
passed                        12       3
passed with distinction        5       4
failed                         2       2
```

- From two-dimensional NumPy array

```
In [61]: pd.DataFrame(np.random.rand(3,2),
    ...:              columns=['foo','bar'],
    ...:              index=['a','b','c'])
Out[61]:
        foo       bar
a  0.640737  0.344832
b  0.257366  0.020849
c  0.687164  0.142364
```

Pandas Data Structures: Data Frame Object

Construction Examples

- From NumPy structured array:

```
In [68]: A=np.zeros(3, dtype=[('A', 'i8'), ('B','f8')])

In [69]: A
Out[69]: array([(0, 0.), (0, 0.), (0, 0.)], dtype=[('A', '<i8'), ('B', '<f8')])

In [70]: pd.DataFrame(A)
Out[70]:
   A    B
0  0  0.0
1  0  0.0
2  0  0.0
```

Pandas Data Structures: Index Object

- Series and DataFrame objects contain an explicit index for data access and data modification
  - Index : *Immutable array* or *ordered set (or multiset)*

```
In [76]: X=np.arange(2,16,3)

In [77]: ind=pd.Index(X)

In [78]: ind[1]
Out[78]: 5

In [79]: X
Out[79]: array([ 2,  5,  8, 11, 14])

In [80]:
```

Pandas Data Structures: Index Object

- Index as immutable array having attributes familiar from NumPy arrays:

```
In [83]: X
Out[83]: array([ 2,  5,  8, 11, 14])

In [84]: ind[0]
Out[84]: 2

In [85]: ind[::2]
Out[85]: Int64Index([2, 8, 14], dtype='int64')

In [86]: print(ind.size, ind.shape, ind.ndim, ind.dtype)
5 (5,) 1 int64
```

- *Index objects are immutable- can not be modified like:*

```
In [87]: ind[1]=0
Traceback (most recent call last):

  File "<ipython-input-87-b10b243764e2>", line 1, in <module>
    ind[1]=0

  File "/anaconda3/lib/python3.7/site-packages/pandas/core/indexes/base.py", line 3938, in __setitem__
    raise TypeError("Index does not support mutable operations")

TypeError: Index does not support mutable operations
```

Pandas Data Structures: Index Object

- Indexes support operations typical of data sets: unions, intersections, differences, and other combinations

```
In [90]: indA=pd.Index([1,3,5,7,9])

In [91]: indB=pd.Index([2,3,5,7,11])

In [92]: indA & indB
Out[92]: Int64Index([3, 5, 7], dtype='int64')

In [93]: indA | indB
Out[93]: Int64Index([1, 2, 3, 5, 7, 9, 11], dtype='int64')

In [94]: indA ^ indB
Out[94]: Int64Index([1, 2, 9, 11], dtype='int64')
```

Pandas: Data Indexing and Selection

- Review: Remember methods and tools to access, set and modify values in NumPy arrays:

Indexing

```
In [27]: A=np.arange(10,40)

In [28]: A
Out[28]:
array([10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26,
       27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39])

In [29]: A=A.reshape(5,6)

In [30]: A
Out[30]:
array([[10, 11, 12, 13, 14, 15],
       [16, 17, 18, 19, 20, 21],
       [22, 23, 24, 25, 26, 27],
       [28, 29, 30, 31, 32, 33],
       [34, 35, 36, 37, 38, 39]])
```

Slicing

```
In [31]: A=A[:,1:5]

In [32]: A
Out[32]:
array([[11, 12, 13, 14],
       [17, 18, 19, 20],
       [23, 24, 25, 26],
       [29, 30, 31, 32],
       [35, 36, 37, 38]])
```

Masking

```
In [33]: A=A>23

In [34]: A
Out[34]:
array([[False, False, False, False],
       [False, False, False, False],
       [False,  True,  True,  True],
       [ True,  True,  True,  True],
       [ True,  True,  True,  True]])
```

Pandas: Data Indexing and Selection

- Review: Remember methods and tools to access, set and modify values in NumPy arrays:

```
In [51]: y=np.arange(35).reshape(5,7)

In [52]: y
Out[52]:
array([[ 0,  1,  2,  3,  4,  5,  6],
       [ 7,  8,  9, 10, 11, 12, 13],
       [14, 15, 16, 17, 18, 19, 20],
       [21, 22, 23, 24, 25, 26, 27],
       [28, 29, 30, 31, 32, 33, 34]])

In [53]: y[1:5:2,::3]
Out[53]:
array([[ 7, 10, 13],
       [21, 24, 27]])

In [54]:
```

Fancy indexing

Pandas: Data Indexing and Selection

- Data Selection in Series

```
In [63]: data=pd.Series([0.25, 0.5, 0.75, 1.0],
    ...: index=['a', 'b', 'c', 'd'])

In [64]: data
Out[64]:
a    0.25
b    0.50
c    0.75
d    1.00
dtype: float64

In [65]: data['b']
Out[65]: 0.5
```

Accessing an element

```
In [66]: 'a' in data
Out[66]: True
```

Examine key or index

```
In [67]: data.keys()
Out[67]: Index(['a', 'b', 'c', 'd'], dtype='object')

In [68]: list(data.items())
Out[68]: [('a', 0.25), ('b', 0.5), ('c', 0.75), ('d', 1.0)]

In [69]: data['e'] = 1.25
```

Adding new value

```
In [70]: data
Out[70]:
a    0.25
b    0.50
c    0.75
d    1.00
e    1.25
dtype: float64
```

Pandas: Data Indexing and Selection

- Data Selection in Series
  - Series builds on dictionary-like interface
  - Series provide array-style item selection as NumPy arrays: slicing, masking, fancy indexing...

```
In [72]: data['a':'c']
Out[72]:
a    0.25
b    0.50
c    0.75
dtype: float64
```

Slicing by explicit index

```
In [73]: data[0:2]
Out[73]:
a    0.25
b    0.50
dtype: float64
```

Slicing by implicit integer index

```
In [74]: data[(data > 0.3) & ( data < 0.8)]
Out[74]:
b    0.50
c    0.75
dtype: float64
```

Masking

```
In [75]: data[['a','e']]
Out[75]:
a    0.25
e    1.25
dtype: float64
```

Indexing

Pandas: Data Indexing and Selection

loc, iloc, and ix

```
In [77]: data =pd.Series(['a', 'b','c'], index=[1,3,5])

In [78]: data
Out[78]:
1    a
3    b
5    c
dtype: object

In [79]: data[1]
Out[79]: 'a'
```

Explicit index

```
In [80]: data[1:3]
Out[80]:
3    b
5    c
dtype: object
```

Implicit integer index

```
In [81]: data.loc[1:3]
Out[81]:
1    a
3    b
dtype: object
```

loc-always references explicit index

```
In [82]: data.iloc[1:3]
Out[82]:
3    b
5    c
dtype: object
```

iloc-always references implicit index

Pandas: Data Indexing and Selection

Data Selection in DataFrame as Dictionary

- Review Dataframe :
  Can be viewed as a two dimensional structured array or dictionary of Series structures sharing the same index.

```
In [84]: area = pd.Series({'California': 423967, 'Texas': 695662,
    ...:                   'New York': 141297, 'Florida': 170312,
    ...:                   'Illinois': 149995})
    ...: pop = pd.Series({'California': 38332521, 'Texas': 26448193,
    ...:                  'New York': 19651127, 'Florida': 19552860,
    ...:                  'Illinois': 12882135})
    ...: data = pd.DataFrame({'area':area, 'pop':pop})
    ...: data
Out[84]:
              area       pop
California  423967  38332521
Texas       695662  26448193
New York    141297  19651127
Florida     170312  19552860
Illinois    149995  12882135

In [85]: data['area']
Out[85]:
California    423967
Texas         695662
New York      141297
Florida       170312
Illinois      149995
Name: area, dtype: int64

In [86]: data.area
Out[86]:
California    423967
Texas         695662
New York      141297
Florida       170312
Illinois      149995
Name: area, dtype: int64

In [87]: data.area is data['area']
Out[87]: True

In [88]: data.pop is data['pop']
Out[88]: False
```

Dataframe as dictionary

Pandas: Data Indexing and Selection

```
In [90]: data
Out[90]:
              area       pop
California  423967  38332521
Texas       695662  26448193
New York    141297  19651127
Florida     170312  19552860
Illinois    149995  12882135

In [91]: data.iloc[:3,:2]
Out[91]:
              area       pop
California  423967  38332521
Texas       695662  26448193
New York    141297  19651127

In [92]: data.iloc[:3,:]
Out[92]:
              area       pop
California  423967  38332521
Texas       695662  26448193
New York    141297  19651127

In [93]: data.loc[:'Illinois',:'pop']
Out[93]:
              area       pop
California  423967  38332521
Texas       695662  26448193
New York    141297  19651127
Florida     170312  19552860
Illinois    149995  12882135

In [94]: data.loc[:'New York',:]
Out[94]:
              area       pop
California  423967  38332521
Texas       695662  26448193
New York    141297  19651127
```

Data selection in dataframe

Pandas: Data Indexing and Selection

```
In [100]: data['density']=data['pop']/data['area']

In [101]: data
Out[101]:
              area       pop     density
California  423967  38332521   90.413926
Texas       695662  26448193   38.018740
New York    141297  19651127  139.076746
Florida     170312  19552860  114.806121
Illinois    149995  12882135   85.883763
```

Dataframe as two-dimensional array

```
In [102]: data.values
Out[102]:
array([[4.23967000e+05, 3.83325210e+07, 9.04139261e+01],
       [6.95662000e+05, 2.64481930e+07, 3.80187404e+01],
       [1.41297000e+05, 1.96511270e+07, 1.39076746e+02],
       [1.70312000e+05, 1.95528600e+07, 1.14806121e+02],
       [1.49995000e+05, 1.28821350e+07, 8.58837628e+01]])

In [103]: data.T
Out[103]:
           California         Texas      New York       Florida      Illinois
area     4.239670e+05  6.956620e+05  1.412970e+05  1.703120e+05  1.499950e+05
pop      3.833252e+07  2.644819e+07  1.965113e+07  1.955286e+07  1.288214e+07
density  9.041393e+01  3.801874e+01  1.390767e+02  1.148061e+02  8.588376e+01

In [104]: data.values[0]
Out[104]: array([4.23967000e+05, 3.83325210e+07, 9.04139261e+01])

In [105]: data['area']
Out[105]:
California    423967
Texas         695662
New York      141297
Florida       170312
Illinois      149995
Name: area, dtype: int64
```

Pandas: Data Indexing and Selection

Dataframe as two-dimensional array

```
In [107]: data
Out[107]:
              area       pop     density
California  423967  38332521   90.413926
Texas       695662  26448193   38.018740
New York    141297  19651127  139.076746
Florida     170312  19552860  114.806121
Illinois    149995  12882135   85.883763

In [108]: data.iloc[:3,:2]
Out[108]:
              area       pop
California  423967  38332521
Texas       695662  26448193
New York    141297  19651127

In [109]: data.loc[:'Illinois',:'pop']
Out[109]:
              area       pop
California  423967  38332521
Texas       695662  26448193
New York    141297  19651127
Florida     170312  19552860
Illinois    149995  12882135

In [110]: data.loc[data.density>100, ['pop','density']]
Out[110]:
               pop     density
New York  19651127  139.076746
Florida   19552860  114.806121
```

Pandas: Operating on Data in Pandas

- Panda inherits from NumPy functionality that performs efficient element-wise operations for example addition, subtraction, multiplication as well as trigonometric functions, exponential and logarithmic functions, etc.

- Universal Functions, which are used for computation on NumPy Arrays is essential for performing operations. Panda automatically aligns indexes when when passing the objects to the functions

```
In [97]: rng=np.random.RandomState(42)

In [98]: ser=pd.Series(rng.randint(0,10,3))

In [99]: ser
Out[99]:
0    6
1    3
2    7
dtype: int64

In [100]: df=pd.DataFrame(rng.randint(0,10,(3,4)),
     ...: columns=['A','B','C','D'])

In [101]: df
Out[101]:
   A  B  C  D
0  4  6  9  2
1  6  7  4  3
2  7  7  2  5

In [102]: np.exp(ser)
Out[102]:
0     403.428793
1      20.085537
2    1096.633158
dtype: float64

In [103]: np.sin(df * np.pi / 4)
Out[103]:
              A         B             C         D
0  1.224647e-16 -1.000000  7.071068e-01  1.000000
1 -1.000000e+00 -0.707107  1.224647e-16  0.707107
2 -7.071068e-01 -0.707107  1.000000e+00 -0.707107
```

Pandas: Operating on Data in Pandas UFuncs and Index Alignment in Series

```
In [107]: area = pd.Series({'Alaska': 1723337, 'Texas': 695662,
     ...:                   'California': 423967}, name='area')
     ...: population = pd.Series({'California': 38332521, 'Texas': 26448193,
     ...:                         'New York': 19651127}, name='population')

In [108]: population/area
Out[108]:
Alaska              NaN
California    90.413926
New York            NaN
Texas         38.018740
dtype: float64

In [109]: area.index
Out[109]: Index(['Alaska', 'Texas', 'California'], dtype='object')

In [110]: population.index
Out[110]: Index(['California', 'Texas', 'New York'], dtype='object')

In [111]: area.index | population.index
Out[111]: Index(['Alaska', 'California', 'New York', 'Texas'], dtype='object')

In [113]: area.divide(population, fill_value=0)
Out[113]:
Alaska              inf
California     0.011060
New York       0.000000
Texas          0.026303
dtype: float64

In [114]: population.divide(area, fill_value=0)
Out[114]:
Alaska         0.000000
California    90.413926
New York            inf
Texas         38.018740
dtype: float64
```

Pandas: Operating on Data in Pandas UFuncs and Index Alignment in DataFrame

```
In [116]: A = pd.DataFrame(rng.randint(0, 20, (2, 2)),
     ...:                  columns=list('AB'))
     ...: A
Out[116]:
    A   B
0   6  18
1  10  10

In [117]: B = pd.DataFrame(rng.randint(0, 10, (3, 3)),
     ...:                  columns=list('BAC'))
     ...: B
Out[117]:
   B  A  C
0  7  4  3
1  7  7  2
2  5  4  1

In [118]: A+B
Out[118]:
      A     B   C
0  10.0  25.0 NaN
1  17.0  17.0 NaN
2   NaN   NaN NaN

In [119]: A.add(B,fill_value=0)
Out[119]:
      A     B    C
0  10.0  25.0  3.0
1  17.0  17.0  2.0
2   4.0   5.0  1.0
```

Python Operator	Pandas Method(s)
`+`	`add()`
`-`	`sub()`, `subtract()`
`*`	`mul()`, `multiply()`
`/`	`truediv()`, `div()`, `divide()`
`//`	`floordiv()`
`%`	`mod()`
`**`	`pow()`

Pandas: Handling Missing Data

- Real world data used in analysis is rarely clean and homogeneous.

- Python provides two ways to represent missing data:
  1) Using **None** Object . Not suited for aggregate functions, sum() or min() across an array: produces and error

  2) NaN (not a number), is a special floating –point value recognized by all systems that used the standard IEEE floating-point representaton

Pandas: Handling Missing Data with NaN

```
In [130]: vals2 = np.array([1, np.nan, 3, 4])

In [131]: vals2.dtype
Out[131]: dtype('float64')

In [132]: 1 + np.nan
Out[132]: nan

In [133]: 0 * np.nan
Out[133]: nan

In [134]: vals2.sum(), vals2.min(), vals2.max()
Out[134]: (nan, nan, nan)

In [135]: np.nansum(vals2), np.nanmin(vals2), np.nanmax(vals2)
Out[135]: (8.0, 1.0, 4.0)
```

Pandas: Handling Missing Data with NaN

1) Detecting null values:

```
In [139]: data = pd.Series([1, np.nan, 'hello', None])

In [140]: data.isnull()
Out[140]:
0    False
1     True
2    False
3     True
dtype: bool

In [141]: data[data.notnull()]
Out[141]:
0        1
2    hello
dtype: object

In [142]: data.dropna()
Out[142]:
0        1
2    hello
dtype: object
```

Pandas: Handling Missing Data with NaN

1) Detecting null values:

```
In [139]: data = pd.Series([1, np.nan, 'hello', None])

In [140]: data.isnull()
Out[140]:
0    False
1     True
2    False
3     True
dtype: bool

In [141]: data[data.notnull()]
Out[141]:
0        1
2    hello
dtype: object

In [142]: data.dropna()
Out[142]:
0        1
2    hello
dtype: object
```

Pandas: Aggregation

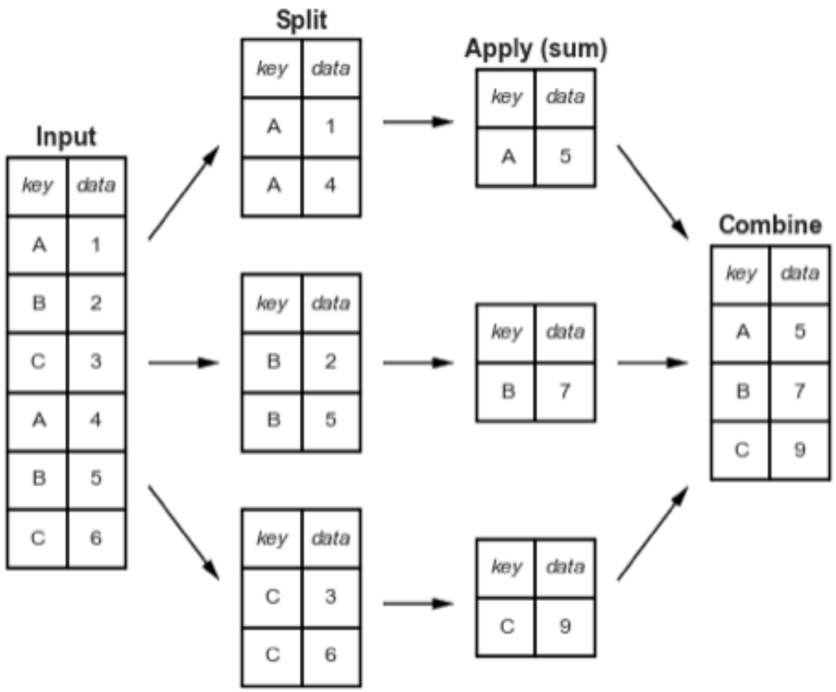
```
In [150]: import seaborn as sns

In [151]: planets=sns.load_dataset('planets')

In [152]: planets.shape
Out[152]: (1035, 6)

In [153]: planets.head
Out[153]:
<bound method NDFrame.head of                       method  number  orbital_period    mass  distance  year
0     Radial Velocity       1      269.300000    7.100      77.40  2006
1     Radial Velocity       1      874.774000    2.210      56.95  2008
2     Radial Velocity       1      763.000000    2.600      19.84  2011
In [155]: planets.dropna().describe()
Out[155]:
          number  orbital_period        mass    distance         year
count  498.00000      498.000000  498.000000  498.000000   498.000000
mean     1.73494      835.778671    2.509320   52.068213  2007.377510
std      1.17572     1469.128259    3.636274   46.596041     4.167284
min      1.00000        1.328300    0.003600    1.350000  1989.000000
25%      1.00000       38.272250    0.212500   24.497500  2005.000000
50%      1.00000      357.000000    1.245000   39.940000  2009.000000
75%      2.00000      999.600000    2.867500   59.332500  2011.000000
max      6.00000    17337.500000   25.000000  354.000000  2014.000000
```

Data available through seaborn package

Understand data with describe()

Pandas GroupBy: Split, Apply, Combine

- groupby operation allows us to aggregate on any label or index



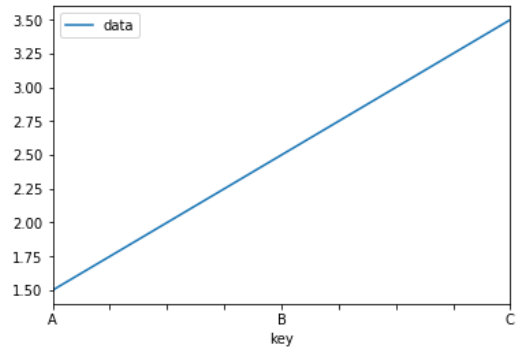
Pandas GroupBy: Split, Apply, Combine

- groupby function on dataframe returns DataFrameGroupBy object.
- to calculate mean, we call mean() function on the DataFrameGroupBy object
- to plot, add call to plot() function

Example:

```
In [165]: df
Out[165]:
  key  data
0   A     0
1   B     1
2   C     2
3   A     3
4   B     4
5   C     5

In [166]: groupByKey=df.groupby('key')

In [167]: groupByKey.mean()
Out[167]:
     data
key
A     1.5
B     2.5
C     3.5
```

```
In [168]: groupByKey.mean().plot()
Out[168]: <matplotlib.axes._subplots.AxesSubplot at 0x1a1f61e828>
```



Pandas GroupBy: Split, Apply, Combine

- Exercise (Stand up and strach): dataFrame: MatchIT studetns

Split, Apply, Combine:

	Name	Country	Age	Weight	Height
0	Olof Jonson	Sweden	30	70	170
1	......				

```
students_df.groupby(‘Country’)
students_df.groupby(‘Country’).shape()
students_df.groupby(‘Country’)[‘Age’].max()
students_df.groupby(‘Country’)[‘Age’].min()
```

Pandas GroupBy: Split, Apply, Combine

```
In [183]: import seaborn as sns
     ...: planets = sns.load_dataset('planets')
     ...: planets.shape
     ...: Out[170]: (1035, 6)

In [184]: planets.head()
Out[184]:
            method  number  orbital_period   mass  distance  year
0  Radial Velocity       1         269.300   7.10     77.40  2006
1  Radial Velocity       1         874.774   2.21     56.95  2008
2  Radial Velocity       1         763.000   2.60     19.84  2011
3  Radial Velocity       1         326.030  19.40    110.62  2007
4  Radial Velocity       1         516.220  10.50    119.47  2009

In [185]: groupByMethod = planets.groupby('method')

In [186]: groupByMethod['year'].median()
Out[186]:
method
Astrometry                       2011.5
Eclipse Timing Variations        2010.0
Imaging                          2009.0
Microlensing                     2010.0
Orbital Brightness Modulation    2011.0
Pulsar Timing                    1994.0
Pulsation Timing Variations      2007.0
Radial Velocity                  2009.0
Transit                          2012.0
Transit Timing Variations        2012.5
Name: year, dtype: float64
```

Column Indexing

GroupByDataFrame Object call to median() method

Pandas GroupBy: Split, Apply, Combine

Iteration over groups:

```
In [246]: for (method, group) in planets.groupby('method'):
     ...:     print("\n {0:20s} \n {1}".format(method, group.describe()))
     ...:

 Astrometry
        number  orbital_period  mass   distance        year
count      2.0        2.000000   0.0   2.000000     2.00000
mean       1.0      631.180000   NaN  17.875000  2011.50000
std        0.0      544.217663   NaN   4.094148     2.12132
min        1.0      246.360000   NaN  14.980000  2010.00000
25%        1.0      438.770000   NaN  16.427500  2010.75000
50%        1.0      631.180000   NaN  17.875000  2011.50000
75%        1.0      823.590000   NaN  19.322500  2012.25000
max        1.0     1016.000000   NaN  20.770000  2013.00000

 Eclipse Timing Variations
          number  orbital_period      mass    distance         year
count   9.000000        9.000000  2.000000    4.000000     9.000000
mean    1.666667     4751.644444  5.125000  315.360000  2010.000000
std     0.500000     2499.130945  1.308148  213.203907     1.414214
min     1.000000     1916.250000  4.200000  130.720000  2008.000000
25%     1.000000     2900.000000  4.662500  130.720000  2009.000000
50%     2.000000     4343.500000  5.125000  315.360000  2010.000000
75%     2.000000     5767.000000  5.587500  500.000000  2011.000000
max     2.000000    10220.000000  6.050000  500.000000  2012.000000

 Imaging
           number  orbital_period  mass    distance         year
count   38.000000       12.000000   0.0   32.000000    38.000000
mean     1.315789   118247.737500   NaN   67.715937  2009.131579
std      0.933035   213978.177277   NaN   53.736817     2.781901
min      1.000000     4639.150000   NaN    7.690000  2004.000000
25%      1.000000     8343.900000   NaN   22.145000  2008.000000
50%      1.000000    27500.000000   NaN   40.395000  2009.000000
75%      1.000000    94250.000000   NaN  132.697500  2011.000000
max      4.000000   730000.000000   NaN  165.000000  2013.000000
```

Pandas GroupBy: Aggregation

Aggregation returns a reduced version of data

```
  key  data1  data2
0   A      0      5
1   B      1      0
2   C      2      3
3   A      3      3
4   B      4      7
5   C      5      9
```

String Function list

```
In [197]: df.groupby('key').aggregate(['min', np.median, max])
Out[197]:
    data1            data2
      min median max   min median max
key
A       0    1.5   3     3    4.0   5
B       1    2.5   4     0    3.5   7
C       2    3.5   5     3    6.0   9
```

Pandas GroupBy: Aggregation

Aggregation, Continued:

```
In [198]: df.groupby('key').aggregate({'data1': 'min',
     ...:                               'data2': 'max'})
Out[198]:
     data1  data2
key
A        0      5
B        1      7
C        2      9
```

Pandas GroupBy: Filter

Allows us to filter out data that we desire(or drop data we do not desire) based on group properties.

```
In [205]: df
Out[205]:
  key  data1  data2
0   A      0      5
1   B      1      0
2   C      2      3
3   A      3      3
4   B      4      7
5   C      5      9

In [206]: df.groupby('key').std()
Out[206]:
       data1     data2
key
A    2.12132  1.414214
B    2.12132  4.949747
C    2.12132  4.242641
```

Standard deviation greater than 4 for B and C keys.

```
In [207]: def filter_func(x):
   ...:     return x['data2'].std() > 4
   ...:
```

Filter function: Display data from column 'data2' where standard deviation greater than 4.

```
In [208]: df.groupby('key').filter(filter_func)
Out[208]:
  key  data1  data2
1   B      1      0
2   C      2      3
4   B      4      7
5   C      5      9
```

Therefore result only includes rows of keys whose standard deviation for column we grouped on – 'key' is greater than 4

Pandas GroupBy: Transformation

- Transform passes each column for each group as Series to the custom function.
- The custom function passed to transform must return a sequence (a one dimensional Series, array or list) the same length as the group.

```
In [214]: df
Out[214]:
  key  data1  data2
0   A      0      5
1   B      1      0
2   C      2      3
3   A      3      3
4   B      4      7
5   C      5      9

In [215]: df.groupby('key').transform(lambda x: x - x)
Out[215]:
   data1  data2
0      0      0
1      0      0
2      0      0
3      0      0
4      0      0
5      0      0

In [216]: df.groupby('key').transform(lambda x: x - x.mean())
Out[216]:
   data1  data2
0   -1.5    1.0
1   -1.5   -3.5
2   -1.5   -3.0
3    1.5   -1.0
4    1.5    3.5
5    1.5    3.0
```

Pandas GroupBy: Apply

Apply implicitly passes all the columns for each group as a **DataFrame** to the custom function – note transform passes each column for each group as a **Series** to the custom function

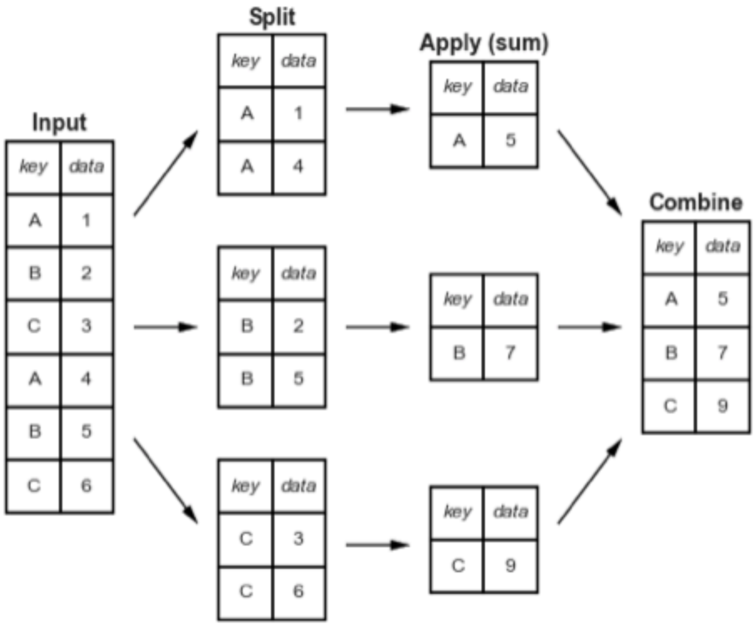
```
In [227]: df
Out[227]:
  key  data1  data2
0   A      0      5
1   B      1      0
2   C      2      3
3   A      3      3
4   B      4      7
5   C      5      9

In [228]: def norm_by_data2(x):
     ...:     # x is a DataFrame of group values
     ...:     x['data1'] /= x['data2'].sum()
     ...:     return x
     ...:

In [229]: df.groupby('key').apply(norm_by_data2)
Out[229]:
  key     data1  data2
0   A  0.000000      5
1   B  0.142857      0
2   C  0.166667      3
3   A  0.375000      3
4   B  0.571429      7
5   C  0.416667      9
```

Normalize column data1 by the sum of data2 column

Would transform instead of apply work here? Why or why not?

Pandas: Pivot Tables

Pivot table is a multidimensional version of GroupBy aggregtion.
Remember split and combine:



In Pivot Table the Split and the Combine are NOT APPLIED ONLY to ONE DIMENSION INDEX – but across two dimensional grid

Pandas: Pivot Tables

Titanic dataset: Available from seaborn library

- We can apply groupby function to e.g. examine fare(ticket) prices bought by males and females by first grouping on ‘sex’ column and than on the returned DataFrameGroupBy object calling function describe() on the fare column.

```
In [166]: titanic.head()
Out[166]:
   survived  pclass     sex   age  ...  deck  embark_town  alive  alone
0         0       3    male  22.0  ...   NaN  Southampton     no  False
1         1       1  female  38.0  ...     C    Cherbourg    yes  False
2         1       3  female  26.0  ...   NaN  Southampton    yes   True
3         1       1  female  35.0  ...     C  Southampton    yes  False
4         0       3    male  35.0  ...   NaN  Southampton     no   True

[5 rows x 15 columns]

In [167]: titanic.groupby('sex')[['fare']].head()
Out[167]:
      fare
0   7.2500
1  71.2833
2   7.9250
3  53.1000
4   8.0500
5   8.4583
6  51.8625
7  21.0750
8  11.1333
9  30.0708

In [168]: titanic.groupby('sex')[['fare']].describe()
Out[168]:
         fare
        count       mean        std   min        25%   50%    75%       max
sex
female  314.0  44.479818  57.997698  6.75  12.071875  23.0  55.00  512.3292
male    577.0  25.523893  43.138263  0.00   7.895800  10.5  26.55  512.3292
```

Pandas: Pivot Tables

The information obtained on the prices of the fare tickets paid by male and female passengers was interesting, but what if we would like to look at the same date but now also with information on the fare class (first, second, third)

```
In [170]: titanic.groupby('sex')[['fare']].mean()
Out[170]:
             fare
sex
female  44.479818
male    25.523893

In [171]: titanic.pivot_table('fare', index='sex', columns='class')
Out[171]:
class        First     Second      Third
sex
female  106.125798  21.970121  16.118810
male     67.226127  19.741782  12.661633
```

Pandas: TODO

- To Do Before Lab2:

  - Read **<u>the following sections</u>** of the Chapter 3:
    - Introduction Pandas Objects
    - Data Indexing and Selection
    - Operating on Data in Pandas
    - Handling Missing Data
    - Aggregation and Grouping
    - Pivot Tables -