Lecture 9 Introduction to Pandas

Pandas--*Python Data Analysis Library* (https://pandas.pydata.org/) provides the high-performance, easy-to-use data structures and data analysis tools in Python, which is very useful in Data Science. In our lectures, we only focust on the elementary usages (https://pandas.pydata.org/pandas-docs/stable/user_guide/10min.html).

In [ ]:
```
import pandas as pd
import numpy as np
```

In [ ]:
```
pd.__version__
```

In [ ]:
```
dir(pd)
```

Important Concepts: `Series` and `DataFrame`

In short, `Series` represents one variable (attributes) of the datasets, while `DataFrame` represents the whole tabular data (it also supports multi-index or tensor cases -- we will not discuss these cases here).

`Series` is Numpy 1d array-like, additionally featuring for "index" which denotes the sample name, which is also similar to Python built-in dictionary type.

In [ ]:
```
s1 = pd.Series([2, 4, 6])
```

In [ ]:
```
type(s1)
```

In [ ]:
```
s1.index
```

In [ ]:
```
s2 = pd.Series([2, 4, 6],index = ['a','b','c'])
```

In [ ]:
```
s2
```

In [ ]:
```
s2_num = s2.values # change to Numpy -- can be view instead of copy if the elements are all numbers
s2_num
```

In [ ]:
```
np.shares_memory(s2_num,s2)
```

In [ ]:
```
s2_num_copy = s2.to_numpy(copy = True) # more recommended in new version of Pandas -- can specify view/copy
np.shares_memory(s2_num_copy,s2)
```

Selection by position

In [ ]:
```
s2[0:2]
```

Selection by index (label)

In [ ]:
```
s2['a']
s2[['a','b']]
```

`Series` and Python Dictionary

```
In [ ]: population_dict = {'California': 38332521,
                           'Texas': 26448193,
                           'New York': 19651127,
                           'Florida': 19552860,
                           'Illinois': 12882135} # this is the built-in python dictionary
        population = pd.Series(population_dict) # initialize Series with dictionary
        population
```

```
In [ ]: population_dict['Texas'] # key and value
```

```
In [ ]: area_dict = {'California': 423967, 'Texas': 695662, 'New York': 141297,
                     'Florida': 170312, 'Illinois': 149995}
        area = pd.Series(area_dict)
        area
```

Create the pandas `DataFrame` from `Series`. Note that in Pandas, the row/column of `DataFrame` are termed as `index` and `columns`.

```
In [ ]: states = pd.DataFrame({'population': population,
                                   'area': area}) # variable names
        states
```

```
In [ ]: type(states)
```

```
In [ ]: states.index
```

```
In [ ]: states.columns
```

```
In [ ]: states['area']
```

```
In [ ]: states.area
```

```
In [ ]: type(states['area'])
```

```
In [ ]: random = pd.DataFrame(np.random.rand(3, 2),columns=['foo', 'bar'],index=['a', 'b',
        'c'])
        random
```

```
In [ ]: random.T
```

Creating DataFrame from Files

```
In [ ]: house_price = pd.read_csv('kc_house_data.csv')
        house_price
```

```
In [ ]: house_price.shape # dimension of the data
```

```
In [ ]: house_price.info() # basic dataset information
```

```
In [ ]: house_price.head(3) # show the head lines
```

```
In [ ]: house_price.sample(5) # show the random samples
```

```
In [ ]: house_price.describe() # descriptive statistics
In [ ]: head = house_price.head()
        head.to_csv('head.csv')
```

```
In [ ]: head.sort_values(by='price')
```

```
In [ ]: help(head.sort_values)
```

```
In [ ]: head.to_numpy()
```

```
In [ ]: help(head.to_numpy)
```

Selection

Selection by label (`.loc`) or by position (`.iloc`)

First recall the basic slicing for Series

```
In [ ]: s2
```

```
In [ ]: s2[0:2] # by position
```

```
In [ ]: s2['a':'c'] # by label
```

```
In [ ]: s2.index
```

However, confusions may occur if the "labels" are very similar to "position"

```
In [ ]: s3= pd.Series(['a','b','c','d','e'])
        s3
```

```
In [ ]: s3.index
```

```
In [ ]: s3[0:2] #slicing -- this is confusing, although it is still by position
```

That's why pandas use `.loc` and `.iloc` to strictly distinguish by label or by position.

```
In [ ]: s3.loc[0:2] # by label
```

```
In [ ]: s3.iloc[0:2] # by position
```

The same applies to DataFrame.

```
In [ ]: head
```

```
In [ ]: head.iloc[:3,:2]
```

```
In [ ]: head.loc[:3,:'date' ]
```

*Note: in the latest version of Pandas, the mixing selection .ix is **deprecated** -- note this when reading the Data Science Handbook!*

```
In [ ]: help(head.loc)
```

```
In [ ]: help(head.iloc)
```

```
In [ ]: head.loc[0,'price']
        head.at[0,'price'] # .at can only access to one value
```

```
In [ ]: help(head.at)
```

More Comments on Slicing and Indexing in DataFrame

Slicing picks rows, while indexing picks columns -- this can be confusing, and that's why `.iloc` and `.loc` are more strict.

```
In [ ]: head['date'] #same with head.date
```

```
In [ ]: head[['date','price']]
```

```
In [ ]: head[['date']]
```

```
In [ ]: head[0:2] #slicing
```

```
In [ ]: head['date':'price'] # this is wrong
```

```
In [ ]: head[:,'date':'price']# this is also wrong!
```

```
In [ ]: head[:,['date','price']] # this is also wrong!!
```

```
In [ ]: head[1:3][['date','price']] # to do slicing and indexing "simultaneously", you have to do them separately!
```

```
In [ ]: head.loc[:,'date':'price'] # no problem for slicing in .loc
```

```
In [ ]: head.loc[:,['date','price']] # fancy indexing is also supported in .loc
```

```
In [ ]: states
```

```
In [ ]: states['California':'Texas']
```

```
In [ ]: states['population']
```

```
In [ ]: states['California':'Texas','population'] # this is wrong
```

```
In [ ]: states.loc['California':'Texas','population']
```

```
In [ ]: states.loc['California':'Texas']
```

Boolean Selection

```
In [ ]: ind = states.area>200000
        ind
```

```
In [ ]: states[ind]
```

```
In [ ]: states[ind,'area'] # this is wrong!
```

In [ ]:

```
states[ind]['area']
```

In [ ]:

```
states.loc[states.area>200000,'population'] # equivalently, states.loc[ind,'population']
```

In [ ]:

```
states.iloc[ind.to_numpy(),1] # in iloc, the boolen should be the Numpy array
```

In [ ]:

```
random
```

In [ ]:

```
random[random['foo']>0.6]
```

In [ ]:

```
house_price
```

Sometimes it's very useful to use the `isin` method to filter samples.

In [ ]:

```
house_price[house_price.loc[:,'bedrooms'].isin([2,4])]
```

In [ ]:

```
house_price[house_price['bedrooms'].isin([2,4])] # the same with column index
```

In [ ]:

```
house_price[(house_price['bedrooms']==2)|(house_price['bedrooms']==4)] #equivalent way
```

Basic Manipulation

- Rename

In [ ]:

```
states
```

In [ ]:

```
states_new = states.rename(columns = {"population":"Population","area":"Area"},index = {"New York":"NewYork"}) # return a new one -- if don't want to, specify inplace = True
states_new
```

In [ ]:

```
help(states.rename)
```

- Append/Drop

In [ ]:

```
states
```

In [ ]:

```
states['density'] = states['population']/states['area']
states
```

In [ ]:

```
new_row = pd.DataFrame({'population':7614893, 'area':184827},index = ['Washington'])
new_row
```

In [ ]:

```
states_new = states.append(new_row)
states_new
```

In [ ]:

```
states_new.drop(index = "Washington",columns = "density",inplace = True)
states_new
```

- Concatenation

`pd.concat()` is a function while `.append()` is a method

```
In [ ]: states_new1 = pd.concat([states,new_row])
        states_new1
```

```
In [ ]: states_new
```

```
In [ ]: pd.concat([states_new,states_new1.loc[:"Illinois","density"]],axis = 1)
```

```
In [ ]: help(pd.concat)
```

- Merge: "Concat by Value"

```
In [ ]: df1 = pd.DataFrame({'employee': ['Bob', 'Jake', 'Lisa', 'Sue'],
                            'group': ['Accounting', 'Engineering', 'Engineering', 'HR']})
        df2 = pd.DataFrame({'employee': ['Lisa', 'Bob', 'Jake', 'Sue'],
                            'hire_date': [2004, 2008, 2012, 2014]})
```

```
In [ ]: df1
```

```
In [ ]: df2
```

```
In [ ]: pd.concat([df1,df2])
```

```
In [ ]: pd.concat([df1,df2],axis=1)
```

```
In [ ]: pd.merge(df1,df2)
```

```
In [ ]: df3 = pd.merge(df1,df2,on="employee")
        df3
```

```
In [ ]: df4 = pd.DataFrame({'group': ['Accounting', 'Engineering', 'HR'],
                            'supervisor': ['Carly', 'Guido', 'Steve']})
        df4
```

```
In [ ]: pd.merge(df3,df4)
```