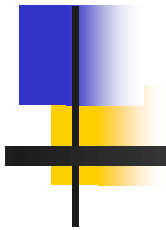
Pandas

IOE 373 Lecture 19



Topics

- Data Input and Output
- Missing Data
- GroupBy
- Merging, Joining and Concatenating

Data Input and Output

- Pandas can read a variety of file types using its pd.read_methods. Let's take a look at the most common data types:
  - CSV Input

```
In [3]: df = pd.read_csv('example')

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

  - CSV Output

Use index=false to avoid passing the dataframe index

```
In [5]: df.to_csv('example',index=False)
```

Data Input and Output

- Excel: Pandas can read and write excel files, keep in mind, this only imports data.
  - Not formulas or images, having images or macros may cause this read_excel method to crash.
  - Excel Input

```
In [10]: pd.read_excel('Excel_Sample.xlsx',sheet_name='Sheet1')
Out[10]:
   Unnamed: 0   a   b   c   d
0           0   0   1   2   3
1           1   4   5   6   7
2           2   8   9  10  11
3           3  12  13  14  15
```

  - Excel Output

```
In [9]: df.to_excel('Excel_Sample.xlsx',sheet_name='Sheet1')
```

Missing Data

- Missing data is common in most data analysis applications.
  - One goal of pandas was to make working with missing data as painless as possible.
  - Pandas uses the floating point value NaN (Not a Number, np.NAN) to represent missing data

```
In [88]: string_data = pd.Series(['aardvark', 'artichoke', np.nan,
'avocado'])

In [89]: string_data
Out[89]:
0     aardvark
1    artichoke
2          NaN
3      avocado
dtype: object
```

```
In [90]: string_data.isnull()
Out[90]:
0    False
1    False
2     True
3    False
dtype: bool
```

Missing Data

- The built-in Python None value is also treated as NA in object arrays:

```
In [91]: string_data[0] = None

In [92]: string_data.isnull()
Out[92]:
0     True
1    False
2     True
3    False
dtype: bool
```

Handling Methods

Argument	Description
`dropna`	Filter axis labels based on whether values for each label have missing data, with varying thresholds for how much missing data to tolerate.
`fillna`	Fill in missing data with some value or using an interpolation method such as `'ffill'` or `'bfill'`.
`isnull`	Return like-type object containing boolean values indicating which values are missing / NA.
`notnull`	Negation of `isnull`.

Filtering out

- You have a number of options for filtering out missing data:
  - **dropna** can be very helpful. On a Series, it returns the Series with only the non-null data and index values:

```
In [95]: from numpy import nan as NA

In [96]: data = pd.Series([1, NA, 3.5, NA, 7])

In [97]: data.dropna()
Out[97]:
0    1.0
2    3.5
4    7.0
dtype: float64
```

Filtering out

- Analogous to Boolean indexing using **notnull():**

```
In [98]: data[data.notnull()]
Out[98]:
0    1.0
2    3.5
4    7.0
dtype: float64
```

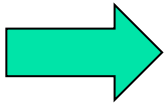
Filtering in Data Frames

- With DataFrame objects, these are a bit more complex. You may want to drop rows or columns which are all NA or just those containing any NAs.
  - **dropna()** by default drops any row containing a missing value:

```
In [99]: data = pd.DataFrame([[1., 6.5, 3.], [1., NA, NA],
    ...:    .....:                [NA, NA, NA], [NA, 6.5, 3.]])

In [100]: cleaned = data.dropna()
```

```
In [101]: data
Out[101]:
     0    1    2
0  1.0  6.5  3.0
1  1.0  NaN  NaN
2  NaN  NaN  NaN
3  NaN  6.5  3.0
```



```
In [102]: cleaned
Out[102]:
     0    1    2
0  1.0  6.5  3.0
```

Filtering

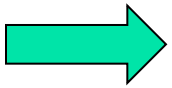
- how='all' will only drop rows that are all NA:

```
In [103]: data.dropna(how='all')
Out[103]:
     0    1    2
0  1.0  6.5  3.0
1  1.0  NaN  NaN
3  NaN  6.5  3.0
```

- Dropping columns in the same way is only a matter of passing axis=1:

```
In [104]: data[4] = NA

In [105]: data
Out[105]:
     0    1    2   4
0  1.0  6.5  3.0 NaN
1  1.0  NaN  NaN NaN
2  NaN  NaN  NaN NaN
3  NaN  6.5  3.0 NaN
```



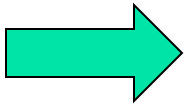
```
In [106]: data.dropna(axis=1, how='all')
Out[106]:
     0    1    2
0  1.0  6.5  3.0
1  1.0  NaN  NaN
2  NaN  NaN  NaN
3  NaN  6.5  3.0
```

Filtering

- Suppose you want to keep only rows containing a certain number of observations. You can indicate this with the **thresh** argument:

```
In [107]: df = pd.DataFrame(np.random.randn(7, 3))

In [108]: df
Out[108]:
          0         1         2
0  0.026430  0.869331  1.301239
1  0.454207 -0.474118  1.667015
2  0.253467 -2.150635  0.331486
3 -0.491507 -1.360017 -1.133550
4 -0.020115  0.140531 -2.560774
5 -0.585818  0.162767  0.351409
6  0.321682  0.842006 -0.284016
```



```
In [111]: df.iloc[:4, 1] = NA; df.iloc[:2, 2] = NA

In [112]: df
Out[112]:
          0         1         2
0  0.026430       NaN       NaN
1  0.454207       NaN       NaN
2  0.253467       NaN  0.331486
3 -0.491507       NaN -1.133550
4 -0.020115  0.140531 -2.560774
5 -0.585818  0.162767  0.351409
6  0.321682  0.842006 -0.284016
```

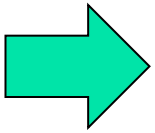
```
In [113]: df.dropna(thresh=3)
Out[113]:
          0         1         2
4 -0.020115  0.140531 -2.560774
5 -0.585818  0.162767  0.351409
6  0.321682  0.842006 -0.284016
```

Thresh: Minimum number of non NaN values

Filling in Data

- What if you may want to fill in the "holes" in there data?
- **fillna** method is the workhorse function to use. Calling fillna with a constant replaces missing values with that value:

```
In [114]: df
Out[114]:
          0         1         2
0  0.026430       NaN       NaN
1  0.454207       NaN       NaN
2  0.253467       NaN  0.331486
3 -0.491507       NaN -1.133550
4 -0.020115  0.140531 -2.560774
5 -0.585818  0.162767  0.351409
6  0.321682  0.842006 -0.284016
```

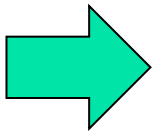


```
In [115]: df.fillna(0)
Out[115]:
          0         1         2
0  0.026430  0.000000  0.000000
1  0.454207  0.000000  0.000000
2  0.253467  0.000000  0.331486
3 -0.491507  0.000000 -1.133550
4 -0.020115  0.140531 -2.560774
5 -0.585818  0.162767  0.351409
6  0.321682  0.842006 -0.284016
```

Filling in Data

- Calling **fillna** with a dictionary you can use a different fill value for each column:

```
In [114]: df
Out[114]:
          0         1         2
0  0.026430       NaN       NaN
1  0.454207       NaN       NaN
2  0.253467       NaN  0.331486
3 -0.491507       NaN -1.133550
4 -0.020115  0.140531 -2.560774
5 -0.585818  0.162767  0.351409
6  0.321682  0.842006 -0.284016
```



```
In [117]: df.fillna({1: 0.5, 2: -1})
Out[117]:
          0         1         2
0  0.026430  0.500000 -1.000000
1  0.454207  0.500000 -1.000000
2  0.253467  0.500000  0.331486
3 -0.491507  0.500000 -1.133550
4 -0.020115  0.140531 -2.560774
5 -0.585818  0.162767  0.351409
6  0.321682  0.842006 -0.284016
```

Filling in Data

- With **fillna** you can also pass the mean or median value of a Series (common method for replacing missing values):

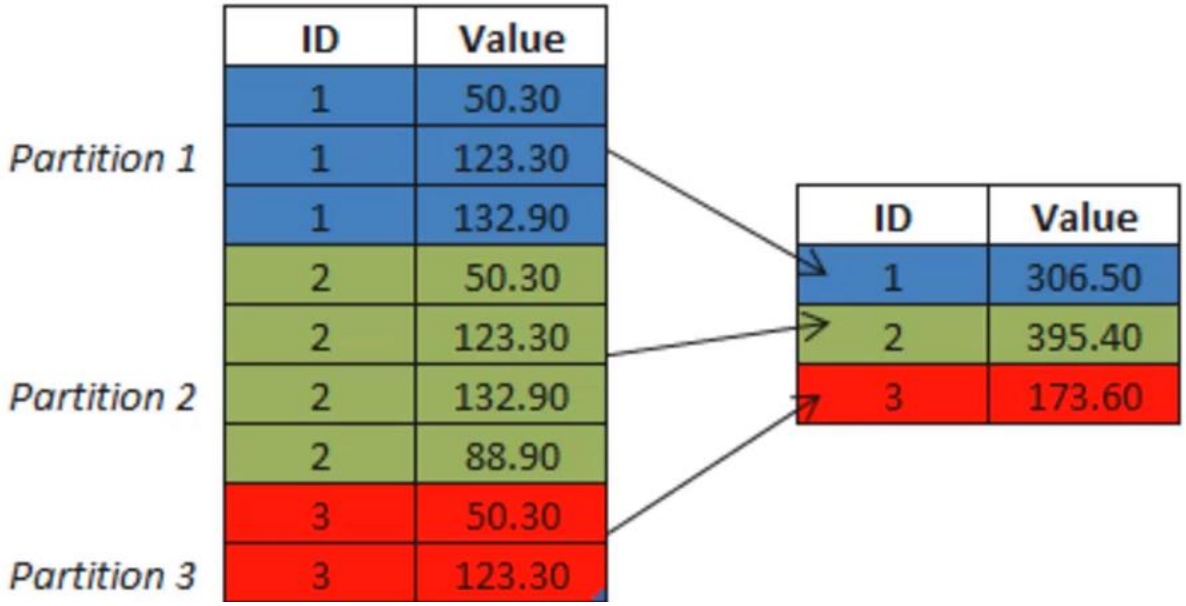
```
In [118]: data = pd.Series([1., NA, 3.5, NA, 7])
```

```
In [119]: data.fillna(data.mean())
Out[119]:
0    1.000000
1    3.833333
2    3.500000
3    3.833333
4    7.000000
dtype: float64
```

```
In [120]: data.fillna(data.median())
Out[120]:
0    1.0
1    3.5
2    3.5
3    3.5
4    7.0
dtype: float64
```

GroupBy

- As we learned in SQL, we can create aggregations in Python as well
- Groupby allows you to group together rows based off of a column and perform an aggregate function on them



Groupby Method

- The groupby method allows you to group rows of data together and call aggregate functions

```
In [122]: data = {'Company':
['GOOG','GOOG','MSFT','MSFT','FB','FB'],
     ...:         'Person':
['Sam','Charlie','Amy','Vanessa','Carl','Sarah'],
     ...:         'Sales':[200,120,340,124,243,350]}

In [123]: df = pd.DataFrame(data)

In [124]: df
Out[124]:
  Company   Person  Sales
0    GOOG      Sam    200
1    GOOG  Charlie    120
2    MSFT      Amy    340
3    MSFT  Vanessa    124
4      FB     Carl    243
5      FB    Sarah    350
```

Groupby Method

- Use **.groupby()** method to group rows together based off of a column name.
  - For example let's group based off of Company. This will create a DataFrameGroupBy object:

```
In [125]: df.groupby('Company')
Out[125]: <pandas.core.groupby.generic.DataFrameGroupBy object at
0x000001927E40B4F0>
```

  - We can save this object as a new variable:

```
In [126]: by_comp = df.groupby("Company")
```

Groupby Method

- With this variable we can call aggregate methods, such as mean():

```
In [127]: by_comp.mean()
Out[127]:
         Sales
Company
FB       296.5
GOOG     160.0
MSFT     232.0
```

```
In [128]: df.groupby('Company').mean()
Out[128]:
         Sales
Company
FB       296.5
GOOG     160.0
MSFT     232.0
```

Groupby – Other Aggregates

- Std deviation, max, min, count

```
In [129]: by_comp.std()
Out[129]:
              Sales
Company
FB        75.660426
GOOG      56.568542
MSFT     152.735065
```

```
In [130]: by_comp.min()
Out[130]:
          Person  Sales
Company
FB           Carl    243
GOOG      Charlie    120
MSFT          Amy    124
```

```
In [131]: by_comp.max()
Out[131]:
          Person  Sales
Company
FB         Sarah    350
GOOG         Sam    200
MSFT     Vanessa    340
```

```
In [132]: by_comp.count()
Out[132]:
         Person  Sales
Company
FB            2      2
GOOG          2      2
MSFT          2      2
```

Or Describe…

```
In [133]: by_comp.describe()
Out[133]:
        Sales
        count   mean         std    min     25%    50%     75%    max
Company
FB        2.0  296.5   75.660426  243.0  269.75  296.5  323.25  350.0
GOOG      2.0  160.0   56.568542  120.0  140.00  160.0  180.00  200.0
MSFT      2.0  232.0  152.735065  124.0  178.00  232.0  286.00  340.0
```

```
In [134]: by_comp.describe().transpose()
Out[134]:
Company              FB        GOOG        MSFT
Sales count    2.000000    2.000000    2.000000
      mean   296.500000  160.000000  232.000000
      std     75.660426   56.568542  152.735065
      min    243.000000  120.000000  124.000000
      25%    269.750000  140.000000  178.000000
      50%    296.500000  160.000000  232.000000
      75%    323.250000  180.000000  286.000000
      max    350.000000  200.000000  340.000000
```

```
In [142]: by_comp.describe().transpose()['GOOG']
Out[142]:
Sales  count      2.000000
       mean     160.000000
       std       56.568542
       min      120.000000
       25%      140.000000
       50%      160.000000
       75%      180.000000
       max      200.000000
Name: GOOG, dtype: float64
```

Merging, Joining and Concatenating

- Three main ways of combining DataFrames together: Merging, Joining and Concatenating.
- Let's start with 3 DataFrames:

```
In [144]: df1 = pd.DataFrame({'A': ['A0', 'A1', 'A2', 'A3'],
     ...:                     'B': ['B0', 'B1', 'B2', 'B3'],
     ...:                     'C': ['C0', 'C1', 'C2', 'C3'],
     ...:                     'D': ['D0', 'D1', 'D2', 'D3']},
     ...:                    index=[0, 1, 2, 3])

In [145]: df2 = pd.DataFrame({'A': ['A4', 'A5', 'A6', 'A7'],
     ...:                     'B': ['B4', 'B5', 'B6', 'B7'],
     ...:                     'C': ['C4', 'C5', 'C6', 'C7'],
     ...:                     'D': ['D4', 'D5', 'D6', 'D7']},
     ...:                     index=[4, 5, 6, 7])

In [146]: df3 = pd.DataFrame({'A': ['A8', 'A9', 'A10', 'A11'],
     ...:                     'B': ['B8', 'B9', 'B10', 'B11'],
     ...:                     'C': ['C8', 'C9', 'C10', 'C11'],
     ...:                     'D': ['D8', 'D9', 'D10', 'D11']},
     ...:                    index=[8, 9, 10, 11])
```

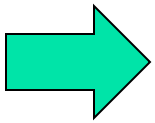
Concatenation

- Concatenation glues together DataFrames.
  - dimensions should match along the axis you are concatenating on.
  - use **pd.concat** and pass in a list of DataFrames to concatenate together:

```
In [7]: df1
Out[7]:
    A   B   C   D
0  A0  B0  C0  D0
1  A1  B1  C1  D1
2  A2  B2  C2  D2
3  A3  B3  C3  D3

In [4]: df2
Out[4]:
    A   B   C   D
4  A4  B4  C4  D4
5  A5  B5  C5  D5
6  A6  B6  C6  D6
7  A7  B7  C7  D7

In [6]: df3
Out[6]:
     A    B    C    D
8   A8   B8   C8   D8
9   A9   B9   C9   D9
10 A10  B10  C10  D10
11 A11  B11  C11  D11
```



```
In [147]: pd.concat([df1,df2,df3])
Out[147]:
      A    B    C    D
0    A0   B0   C0   D0
1    A1   B1   C1   D1
2    A2   B2   C2   D2
3    A3   B3   C3   D3
4    A4   B4   C4   D4
5    A5   B5   C5   D5
6    A6   B6   C6   D6
7    A7   B7   C7   D7
8    A8   B8   C8   D8
9    A9   B9   C9   D9
10  A10  B10  C10  D10
11  A11  B11  C11  D11
```

Merging

- The **merge** function allows you to merge DataFrames together using a similar logic as merging SQL Tables together. Let's start with sample tables:

```
In [150]: left = pd.DataFrame({'key': ['K0', 'K1', 'K2', 'K3'],
     ...:                      'A': ['A0', 'A1', 'A2', 'A3'],
     ...:                      'B': ['B0', 'B1', 'B2', 'B3']})

In [151]: right = pd.DataFrame({'key': ['K0', 'K1', 'K2', 'K3'],
     ...:                       'C': ['C0', 'C1', 'C2', 'C3'],
     ...:                       'D': ['D0', 'D1', 'D2', 'D3']})
```

Merging

```
In [152]: left
Out[152]:
  key   A   B
0  K0  A0  B0
1  K1  A1  B1
2  K2  A2  B2
3  K3  A3  B3
```

```
In [153]: right
Out[153]:
  key   C   D
0  K0  C0  D0
1  K1  C1  D1
2  K2  C2  D2
3  K3  C3  D3
```

```
In [154]: pd.merge(left,right,how='inner',on='key')
Out[154]:
  key   A   B   C   D
0  K0  A0  B0  C0  D0
1  K1  A1  B1  C1  D1
2  K2  A2  B2  C2  D2
3  K3  A3  B3  C3  D3
```

Joining

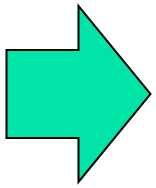
- Joining is a convenient method for combining the columns of two potentially differently-indexed DataFrames into a single result DataFrame.
- Sample dataframes:

```
In [155]: left = pd.DataFrame({'A': ['A0', 'A1', 'A2'],
     ...:                      'B': ['B0', 'B1', 'B2']},
     ...:                       index=['K0', 'K1', 'K2'])

In [156]: right = pd.DataFrame({'C': ['C0', 'C2', 'C3'],
     ...:                       'D': ['D0', 'D2', 'D3']},
     ...:                        index=['K0', 'K2', 'K3'])
```

Joining

```
In [158]: left
Out[158]:
     A   B
K0   A0  B0
K1   A1  B1
K2   A2  B2
```



```
In [159]: right
Out[159]:
     C   D
K0   C0  D0
K2   C2  D2
K3   C3  D3
```

```
In [157]: left.join(right)
Out[157]:
     A   B    C    D
K0   A0  B0   C0   D0
K1   A1  B1  NaN  NaN
K2   A2  B2   C2   D2
```

```
In [160]: left.join(right, how='outer')
Out[160]:
     A    B    C    D
K0   A0   B0   C0   D0
K1   A1   B1  NaN  NaN
K2   A2   B2   C2   D2
K3  NaN  NaN   C3   D3
```

```
In [161]: left.join(right, how='inner')
Out[161]:
     A   B   C   D
K0   A0  B0  C0  D0
K2   A2  B2  C2  D2
```