Applied Statistical Methods

Data Wrangling with Pandas

Xuemao Zhang
East Stroudsburg University

February 6, 2023

Outline

- Sorting
- Missing and duplicate data
- Renaming
- Transformation or Adding new columns
- Reshaping data
- Merging data
- Aggregating data

Sorting

- We will often find the need to sort our data frame by the values of one or many columns.
- To accomplish this, we can use the `sort_values()` method.
  - `inplace`, default False; If True, perform operation in-place

```
import pandas as pd
mtcars=pd.read_csv("../data/mtcars.csv")
mtcars.sort_values(by='mpg',ascending=False,inplace=True)
mtcars.head()
#by default ascending=True
```

```
##          Unnamed: 0   mpg  cyl  disp   hp  ...   qsec  vs  am  gear  carb
## 19  Toyota Corolla  33.9    4  71.1   65  ...  19.90   1   1     4     1
## 17        Fiat 128  32.4    4  78.7   66  ...  19.47   1   1     4     1
## 27    Lotus Europa  30.4    4  95.1  113  ...  16.90   1   1     5     2
## 18     Honda Civic  30.4    4  75.7   52  ...  18.52   1   1     4     2
## 25       Fiat X1-9  27.3    4  79.0   66  ...  18.90   1   1     4     1
##
## [5 rows x 12 columns]
```

Sorting

- Sort by two variables

```
import pandas as pd
mtcars.sort_values(by=['mpg','disp'],ascending=[True,False]).head()
#by default ascending=True
```

```
##              Unnamed: 0   mpg  cyl   disp   hp  ...   qsec  vs  am  gear
## 14   Cadillac Fleetwood  10.4    8  472.0  205  ...  17.98   0   0     3
## 15  Lincoln Continental  10.4    8  460.0  215  ...  17.82   0   0     3
## 23           Camaro Z28  13.3    8  350.0  245  ...  15.41   0   0     3
## 6            Duster 360  14.3    8  360.0  245  ...  15.84   0   0     3
## 16    Chrysler Imperial  14.7    8  440.0  230  ...  17.42   0   0     3
##
## [5 rows x 12 columns]
```

Sorting

- Pandas also provides an additional way to look at a subset of the sorted values; we can use `nlargest()` to grab the n rows with the largest values according to specific criteria and `nsmallest()` to grab the n smallest rows, without the need to sort the data beforehand

```
mtcars.nlargest(n=6, columns='mpg')
```

```
##          Unnamed: 0   mpg  cyl   disp   hp  ...   qsec  vs  am  gear  car
## 19  Toyota Corolla  33.9    4   71.1   65  ...  19.90   1   1     4
## 17        Fiat 128  32.4    4   78.7   66  ...  19.47   1   1     4
## 27    Lotus Europa  30.4    4   95.1  113  ...  16.90   1   1     5
## 18     Honda Civic  30.4    4   75.7   52  ...  18.52   1   1     4
## 25       Fiat X1-9  27.3    4   79.0   66  ...  18.90   1   1     4
## 26   Porsche 914-2  26.0    4  120.3   91  ...  16.70   0   1     5
##
## [6 rows x 12 columns]
```

```
mtcars.nsmallest(n=6, columns='mpg')
```

```
##              Unnamed: 0   mpg  cyl   disp   hp  ...   qsec  vs  am  gear
## 15  Lincoln Continental  10.4    8  460.0  215  ...  17.82   0   0     3
## 14   Cadillac Fleetwood  10.4    8  472.0  205  ...  17.98   0   0     3
## 23           Camaro Z28  13.3    8  350.0  245  ...  15.41   0   0     3
```

Missing and duplicate data

- Types of `mising` values
  - Pandas use **NaN** (Not a Number) for missing or non-numeric values in a float or integer column.
  - `NA`: general missing/empty value which is a convention in R.
  - `inf` and `-inf` : Infinity, happens when you divide a positive number (or negative number) by 0.
- We can use the info() method to see if we have any missing values and check that our columns have the expected data types

```
quiz=pd.read_csv("../data/quiz1.txt",header=0, delimiter='\t')
quiz.info()
```

```
## <class 'pandas.core.frame.DataFrame'>
## RangeIndex: 40 entries, 0 to 39
## Data columns (total 7 columns):
##  #   Column  Non-Null Count  Dtype
## ---  ------  --------------  -----
##  0   ID      40 non-null     int64
##  1   Q1      38 non-null     float64
##  2   Q2      38 non-null     float64
##  3   Q3      39 non-null     float64
##  4   Q4      39 non-null     float64
```

Missing and duplicate data

- `DataFrame.isna()`: Return a boolean same-sized object indicating if the values are NA
  - `DataFrame.isnull()` is an alias for `DataFrame.isna()`.

```
quiz.isna()
#quiz.isnull()
```

```
##        ID     Q1     Q2     Q3     Q4     Q5     Q6
## 0   False  False  False  False  False  False  False
## 1   False  False  False  False  False  False  False
## 2   False  False  False  False  False  False  False
## 3   False  False  False  False  False  False   True
## 4   False  False  False  False  False   True   True
## 5   False   True  False  False  False  False   True
## 6   False  False  False  False  False  False  False
## 7   False  False  False  False  False  False  False
## 8   False  False  False  False  False  False  False
## 9   False  False  False  False  False  False  False
## 10  False  False   True  False  False  False   True
## 11  False  False  False  False  False  False  False
## 12  False  False  False  False  False  False  False
## 13  False  False  False  False  False  False  False
```

Missing and duplicate data

- Use `pandas.DataFrame.any()` check if `isna()` and `isnull()` are equivalent.
  - https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.any.html

```
(quiz.isna()!= quiz.isnull()).any()
#type(quiz.isna()== quiz.isnull())
```

```
## ID    False
## Q1    False
## Q2    False
## Q3    False
## Q4    False
## Q5    False
## Q6    False
## dtype: bool
```

Missing and duplicate data

```
quiz.Q6.isna()
#quiz.isnull()
```

```
## 0     False
## 1     False
## 2     False
## 3      True
## 4      True
## 5      True
## 6     False
## 7     False
## 8     False
## 9     False
## 10     True
## 11    False
## 12    False
## 13    False
## 14    False
## 15    False
## 16    False
## 17    False
## 18    False
```

Missing and duplicate data

- `numpy.isnan` can test element-wise for NaN and return result as a boolean array.
- https://numpy.org/doc/stable/reference/generated/numpy.isnan.html

```
import numpy as np
np.isnan(quiz.Q6)
```

```
## 0     False
## 1     False
## 2     False
## 3      True
## 4      True
## 5      True
## 6     False
## 7     False
## 8     False
## 9     False
## 10     True
## 11    False
## 12    False
## 13    False
## 14    False
## 15    False
```

Missing and duplicate data

- `DataFrame.dropna()`: Remove all rows with missing values.

```
quiz.dropna()
```

```
##      ID    Q1    Q2    Q3    Q4    Q5    Q6
## 0     1   8.0   9.0  10.0   9.5  10.0   8.0
## 1     2   8.0   8.0   8.0  10.0   9.0   8.0
## 2     3  10.0   7.0  10.0  10.0  10.0   8.0
## 6     7  10.0   5.0   9.0   8.0  10.0   7.0
## 7     8  10.0  10.0  10.0   9.0  10.0   7.0
## 8     9  10.0  10.0   6.0   7.0   8.0  10.0
## 9    10   8.0  10.0  10.0   4.0  10.0   7.0
## 11   12   6.0   8.0   5.0   8.0  10.0   8.0
## 12   13  10.0  10.0   8.0  10.0  10.0   9.0
## 13   14  10.0  10.0  10.0  10.0   9.0   8.0
## 14   15  10.0  10.0  10.0  10.0  10.0  10.0
## 15   16  10.0   8.0   9.0   6.0   9.0   6.0
## 17   18  10.0  10.0  10.0   7.0   6.0   9.0
## 18   19  10.0   7.0   7.0   7.0  10.0   8.0
## 19   20  10.0  10.0  10.0  10.0  10.0   8.0
## 20   21  10.0   9.0  10.0   8.5  10.0   5.0
## 21   22  10.0  10.0  10.0   7.0  10.0   7.0
```

Missing and duplicate data

- `DataFrame.fillna()`: Fill NA/NaN values using the specified method.
  - Check https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.fillna.html see methods filling missing values

```
quiz.fillna(-999)
```

```
##      ID     Q1     Q2     Q3     Q4     Q5     Q6
## 0     1    8.0    9.0   10.0    9.5   10.0    8.0
## 1     2    8.0    8.0    8.0   10.0    9.0    8.0
## 2     3   10.0    7.0   10.0   10.0   10.0    8.0
## 3     4    6.0    5.0    9.0    8.0    5.0 -999.0
## 4     5   10.0    6.0    8.0    6.0 -999.0 -999.0
## 5     6 -999.0    9.0   10.0   10.0   10.0 -999.0
## 6     7   10.0    5.0    9.0    8.0   10.0    7.0
## 7     8   10.0   10.0   10.0    9.0   10.0    7.0
## 8     9   10.0   10.0    6.0    7.0    8.0   10.0
## 9    10    8.0   10.0   10.0    4.0   10.0    7.0
## 10   11   10.0 -999.0    4.0    7.0    9.0 -999.0
## 11   12    6.0    8.0    5.0    8.0   10.0    8.0
## 12   13   10.0   10.0    8.0   10.0   10.0    9.0
## 13   14   10.0   10.0   10.0   10.0    9.0    8.0
```

Missing and duplicate data

- Duplicate rows may be found in a DataFrame for any number of reasons
- The DataFrame method `duplicated()` returns a boolean Series indicating whether each row is a duplicate or not

```
data = pd.DataFrame({'k1': ['one'] * 3 + ['two'] * 4,
'k2': [1, 1, 2, 3, 3, 4, 4]})
data.duplicated()
```

```
## 0    False
## 1     True
## 2    False
## 3    False
## 4     True
## 5    False
## 6     True
## dtype: bool
```

Missing and duplicate data

- `drop_duplicates`: Return DataFrame with duplicate rows removed.

```
data.drop_duplicates()
```

```
##     k1  k2
## 0  one   1
## 2  one   2
## 3  two   3
## 5  two   4
```

Renaming

- The DataFrame class has a `rename()` method that takes a dictionary mapping the old column name to the new column name.
  - `inplace=True` updates the raw data

```
data.rename(columns={'k1': 'Var1', 'k2': 'Var2'}, inplace=True)
data.head()
```

```
##   Var1  Var2
## 0  one     1
## 1  one     1
## 2  one     2
## 3  two     3
## 4  two     3
```

Renaming

- `str.upper`: Convert strings in the Series/Index to uppercase

```
mtcars.rename(columns=str.upper, inplace=True)
mtcars.head()
```

```
##          UNNAMED: 0   MPG  CYL  DISP   HP  ...   QSEC  VS  AM  GEAR  CARB
## 19  Toyota Corolla  33.9    4  71.1   65  ...  19.90   1   1     4     1
## 17        Fiat 128  32.4    4  78.7   66  ...  19.47   1   1     4     1
## 27    Lotus Europa  30.4    4  95.1  113  ...  16.90   1   1     5     2
## 18     Honda Civic  30.4    4  75.7   52  ...  18.52   1   1     4     2
## 25       Fiat X1-9  27.3    4  79.0   66  ...  18.90   1   1     4     1
##
## [5 rows x 12 columns]
```

Renaming

- `str.lower`: Convert strings in the Series/Index to lowercase

```
mtcars.rename(columns=str.lower, inplace=True)
mtcars.head()
```

```
##          unnamed: 0   mpg  cyl  disp   hp  ...   qsec  vs  am  gear  carb
## 19  Toyota Corolla  33.9    4  71.1   65  ...  19.90   1   1     4     1
## 17        Fiat 128  32.4    4  78.7   66  ...  19.47   1   1     4     1
## 27    Lotus Europa  30.4    4  95.1  113  ...  16.90   1   1     5     2
## 18     Honda Civic  30.4    4  75.7   52  ...  18.52   1   1     4     2
## 25       Fiat X1-9  27.3    4  79.0   66  ...  18.90   1   1     4     1
##
## [5 rows x 12 columns]
```

Transformation: Data type converting

- Type conversion: We sometimes want to convert `dtypes` attribute of columns
- Method `pd.DataFrame.astype` can convert a pandas object to a specified `dtype`.

```
mtcars['cyl'] = mtcars['cyl'].astype(str)
mtcars.dtypes
```

```
## unnamed: 0     object
## mpg           float64
## cyl            object
## disp          float64
## hp              int64
## drat          float64
## wt            float64
## qsec          float64
## vs              int64
## am              int64
## gear            int64
## carb            int64
## dtype: object
```

Transformation: Data type converting

- We can use the `pd.to_numeric()` function to convert it back into numeric

```
mtcars.loc[:,'cyl'] = pd.to_numeric(mtcars.cyl)
```

```
## <string>:1: DeprecationWarning: In a future version, `df.iloc[:, i] = ne
```

```
mtcars.dtypes
```

```
## unnamed: 0     object
## mpg           float64
## cyl             int64
## disp          float64
## hp              int64
## drat          float64
## wt            float64
## qsec          float64
## vs              int64
## am              int64
## gear            int64
## carb            int64
## dtype: object
```

Transformation: Data type converting

- Function `pd.to_numeric()` converts a string to numeric
- Function `pd.to_datetime` converts a string into a datetime
  - https://pandas.pydata.org/docs/reference/api/pandas.to_datetime.html
- Function `pd.to_timedelta()` converts a string to timedelta.
  - https://pandas.pydata.org/docs/reference/api/pandas.to_timedelta.html

Transformation or Adding new columns

- For many data sets, we may wish to perform some transformation based on the values in a column in a DataFrame. For example,

```
mtcars['wt2']=mtcars['wt']**2
mtcars.columns
```

```
## Index(['unnamed: 0', 'mpg', 'cyl', 'disp', 'hp', 'drat', 'wt', 'qsec', '
##        'am', 'gear', 'carb', 'wt2'],
##       dtype='object')
```

```
mtcars[['wt','wt2']].head()
```

```
##         wt       wt2
## 19   1.835  3.367225
## 17   2.200  4.840000
## 27   1.513  2.289169
## 18   1.615  2.608225
## 25   1.935  3.744225
```

Transformation or Adding new columns

- Sometimes, we may want to convert Index to a column
- Method `DataFrame.reset_index()` can be used. When we reset the index, the old index is added as a column, and a new sequential index is used
  - https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.reset_index.html

```
data = {'Product': ['Computer','Printer','Monitor','Desk','Phone'],
          'Price': [1200,250,400,700,350]
        }

df = pd.DataFrame(data, columns = ['Product','Price'],
index = ['Item_1','Item_2','Item_3','Item_4','Item_5'])

df.reset_index(inplace=True)
df = df.rename(columns = {'index':'item'})
df.head()
```

```
##       item   Product  Price
## 0   Item_1  Computer   1200
## 1   Item_2   Printer    250
## 2   Item_3   Monitor    400
## 3   Item_4      Desk    700
```

Transformation or Adding new columns

Transformation with a Function or Mapping:

- Consider the following hypothetical data collected about some kinds of meat:

```
data = pd.DataFrame({'food': ['bacon', 'pulled pork', 'bacon', 'Pastrami',
'corned beef', 'Bacon', 'pastrami', 'honey ham','nova lox'],
'ounces': [4, 3, 12, 6, 7.5, 8, 3, 5, 6]})
data
```

```
##           food  ounces
## 0        bacon     4.0
## 1  pulled pork     3.0
## 2        bacon    12.0
## 3     Pastrami     6.0
## 4  corned beef     7.5
## 5        Bacon     8.0
## 6     pastrami     3.0
## 7    honey ham     5.0
## 8     nova lox     6.0
```

Transformation or Adding new columns

Transformation with a Function or Mapping:

- Suppose you wanted to add a column indicating the type of animal that each food came from. Let's write down a **mapping** of each distinct meat type to the kind of animal.
- The mapping is a dictionary

```
meat_to_animal = {
'bacon': 'pig',
'pulled pork': 'pig',
'pastrami': 'cow',
'corned beef': 'cow',
'honey ham': 'pig',
'nova lox': 'salmon'
}
```

Transformation or Adding new columns

Transformation with a Function or Mapping:

- Some of the meats above are capitalized and others are not. Thus we need to convert the values of column `food` to lowercase first.
  - Method `str.lower()` converts strings in the Series/Index to lowercase.
- The `pd.Series.map()` method maps values of Series according to an input mapping or function.
  - `map` accepts a `dict` or a Series.

```
data['animal']=data['food'].str.lower().map(meat_to_animal)
#data['animal']=data['food'].map(str.lower).map(meat_to_animal)
#new column 'animal'
data
```

```
##            food  ounces  animal
## 0         bacon     4.0     pig
## 1  pulled pork     3.0     pig
## 2         bacon    12.0     pig
## 3     Pastrami     6.0     cow
## 4  corned beef     7.5     cow
## 5         Bacon     8.0     pig
## 6     pastrami     3.0     cow
## 7    honey ham     5.0     pig
```

Transformation or Adding new columns

Transformation with a Function or Mapping:

- Using `map` is a convenient way to perform element-wise transformations and other data cleaning-related operations.
- We could also have passed a function that does all the work

```
data['animal2']=data['food'].map(lambda x: meat_to_animal[x.lower()])
data
```

```
##            food  ounces  animal animal2
## 0         bacon     4.0     pig     pig
## 1  pulled pork     3.0     pig     pig
## 2         bacon    12.0     pig     pig
## 3     Pastrami     6.0     cow     cow
## 4  corned beef     7.5     cow     cow
## 5         Bacon     8.0     pig     pig
## 6     pastrami     3.0     cow     cow
## 7    honey ham     5.0     pig     pig
## 8     nova lox     6.0  salmon  salmon
```

Transformation or Adding new columns

- Suppose we add a new column based on a condition

```
mtcars['disp_cat'] = ['Low' if x <=200
else 'Hign' for x in mtcars['disp']]
mtcars[['disp', 'disp_cat']].head(10)
```

```
##       disp disp_cat
## 19    71.1      Low
## 17    78.7      Low
## 27    95.1      Low
## 18    75.7      Low
## 25    79.0      Low
## 26   120.3      Low
## 7    146.7      Low
## 2    108.0      Low
## 8    140.8      Low
## 20   120.1      Low
```

Transformation or Adding new columns

Using function DataFrame.apply():

- Use function `DataFrame.apply()` to create variables with conditions
  - https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.apply.html

```
def set_disp(row):
    if row['disp'] <= 200:
        return("Low")
    elif (row['disp'] > 200) & (row['disp'] <= 400):
        return("Medium")
    else:
        return("High")

mtcars['disp_cat2'] = mtcars.apply(set_disp, axis=1)
```

Transformation or Adding new columns

DataFrame.assign() method can be used:

- Use function `DataFrame.assign()` and `DataFrame.apply()` function to creating variables with conditions
  - `DataFrame.assign()` assign new columns to a DataFrame. https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.assign.html
  - https://sparkbyexamples.com/pandas/pandas-create-conditional-column-in-dataframe/

```
mtcars = mtcars.assign(disp_cat3=mtcars.apply(set_disp, axis=1))
```

Transformation or Adding new columns

```
mtcars[['disp', 'disp_cat','disp_cat2','disp_cat3']].head(20)
```

```
##      disp disp_cat disp_cat2 disp_cat3
## 19   71.1      Low       Low       Low
## 17   78.7      Low       Low       Low
## 27   95.1      Low       Low       Low
## 18   75.7      Low       Low       Low
## 25   79.0      Low       Low       Low
## 26  120.3      Low       Low       Low
## 7   146.7      Low       Low       Low
## 2   108.0      Low       Low       Low
## 8   140.8      Low       Low       Low
## 20  120.1      Low       Low       Low
## 31  121.0      Low       Low       Low
## 3   258.0     Hign    Medium    Medium
## 1   160.0      Low       Low       Low
## 0   160.0      Low       Low       Low
## 29  145.0      Low       Low       Low
## 9   167.6      Low       Low       Low
## 24  400.0     Hign    Medium    Medium
## 4   360.0     Hign    Medium    Medium
## 5   225.0     Hign    Medium    Medium
```

Transformation or Adding new columns

- It is equivalent to the following

```
mtcars['disp_cat4'] = ['Low' if x <=200
else 'Medium' if (x>200) & (x<=400)
else 'High' for x in mtcars['disp']]
any(mtcars['disp_cat4']!=mtcars['disp_cat3'])
```

```
## False
```

```
mtcars[['disp', 'disp_cat','disp_cat2','disp_cat3','disp_cat4']].tail(10)
```

```
##      disp disp_cat disp_cat2 disp_cat3 disp_cat4
## 28  351.0     Hign    Medium    Medium    Medium
## 21  318.0     Hign    Medium    Medium    Medium
## 13  275.8     Hign    Medium    Medium    Medium
## 22  304.0     Hign    Medium    Medium    Medium
## 30  301.0     Hign    Medium    Medium    Medium
## 16  440.0     Hign      High      High      High
## 6   360.0     Hign    Medium    Medium    Medium
## 23  350.0     Hign    Medium    Medium    Medium
## 15  460.0     Hign      High      High      High
## 14  472.0     Hign      High      High      High
```

Transformation or Adding new columns

- Or use the lambda function

```
def set2_disp(x):
    if x <= 200:
        return("Low")
    elif (x > 200) & (x <= 400):
        return("Medium")
    else:
        return("High")

mtcars['disp_cat5'] = mtcars['disp'].apply(lambda x: set2_disp(x))
#mtcars['disp_cat5'] = mtcars['disp'].apply(set2_disp)
any(mtcars['disp_cat5']!=mtcars['disp_cat3'])
```

```
## False
```

Transformation: cut method

- pd.cut() method can be used to convert continuous variable to a categorical variable by sorting data values into bins.
  - https://pandas.pydata.org/docs/reference/api/pandas.cut.html

```
mtcars=mtcars.assign(mpg_cat=pd.cut(mtcars['mpg'], bins=4))
mtcars[['mpg','mpg_cat']].head(10)
```

```
##       mpg          mpg_cat
## 19   33.9   (28.025, 33.9]
## 17   32.4   (28.025, 33.9]
## 27   30.4   (28.025, 33.9]
## 18   30.4   (28.025, 33.9]
## 25   27.3  (22.15, 28.025]
## 26   26.0  (22.15, 28.025]
## 7    24.4  (22.15, 28.025]
## 2    22.8  (22.15, 28.025]
## 8    22.8  (22.15, 28.025]
## 20   21.5  (16.275, 22.15]
```

Reshaping data

- Long data and wide data
  - Data is wide or long is **with respect** to certain variables.

variables

	date	TMAX	TMIN	TOBS
0	2018-10-01	21.1	8.9	13.9
1	2018-10-02	23.9	13.9	17.2
2	2018-10-03	25.0	15.6	16.1
3	2018-10-04	22.8	11.7	11.7
4	2018-10-05	23.3	11.7	18.9
5	2018-10-06	20.0	13.3	16.1

observations

repeated values for **date** column

variable names

variable values

	date	datatype	value
0	2018-10-01	TMAX	21.1
1	2018-10-01	TMIN	8.9
2	2018-10-01	TOBS	13.9
3	2018-10-02	TMAX	23.9
4	2018-10-02	TMIN	13.9
5	2018-10-02	TOBS	17.2

Reshaping data

- Long data

```
import pandas as pd
long_df = pd.read_csv('../data/long_data.csv', parse_dates=['date'])
long_df.head(10)
```

```
##   attributes datatype       date            station  value
## 0    ,,H,0700     TMAX 2018-10-01  GHCND:USC00280907   21.1
## 1    ,,H,0700     TMIN 2018-10-01  GHCND:USC00280907    8.9
## 2    ,,H,0700     TOBS 2018-10-01  GHCND:USC00280907   13.9
## 3    ,,H,0700     TMAX 2018-10-02  GHCND:USC00280907   23.9
## 4    ,,H,0700     TMIN 2018-10-02  GHCND:USC00280907   13.9
## 5    ,,H,0700     TOBS 2018-10-02  GHCND:USC00280907   17.2
## 6    ,,H,0700     TMAX 2018-10-03  GHCND:USC00280907   25.0
## 7    ,,H,0700     TMIN 2018-10-03  GHCND:USC00280907   15.6
## 8    ,,H,0700     TOBS 2018-10-03  GHCND:USC00280907   16.1
## 9    ,,H,0700     TMAX 2018-10-04  GHCND:USC00280907   22.8
```

Reshaping data

- long-to-wide: **spread rows into columns**
- See `df.pivot()`: https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.pivot.html

```
wide_df=long_df.pivot(columns='datatype', values='value', index='date')
wide_df=wide_df.reset_index()
wide_df.head()
```

```
## datatype        date  TMAX  TMIN  TOBS
## 0         2018-10-01  21.1   8.9  13.9
## 1         2018-10-02  23.9  13.9  17.2
## 2         2018-10-03  25.0  15.6  16.1
## 3         2018-10-04  22.8  11.7  11.7
## 4         2018-10-05  23.3  11.7  18.9
```

Reshaping data

- wide-to-long: **Gather columns into rows**
- See `df.melt()`: https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.melt.html

```
longdf=wide_df.melt(id_vars='date', value_vars=['TMAX', 'TMIN', 'TOBS'])
longdf.head()
```

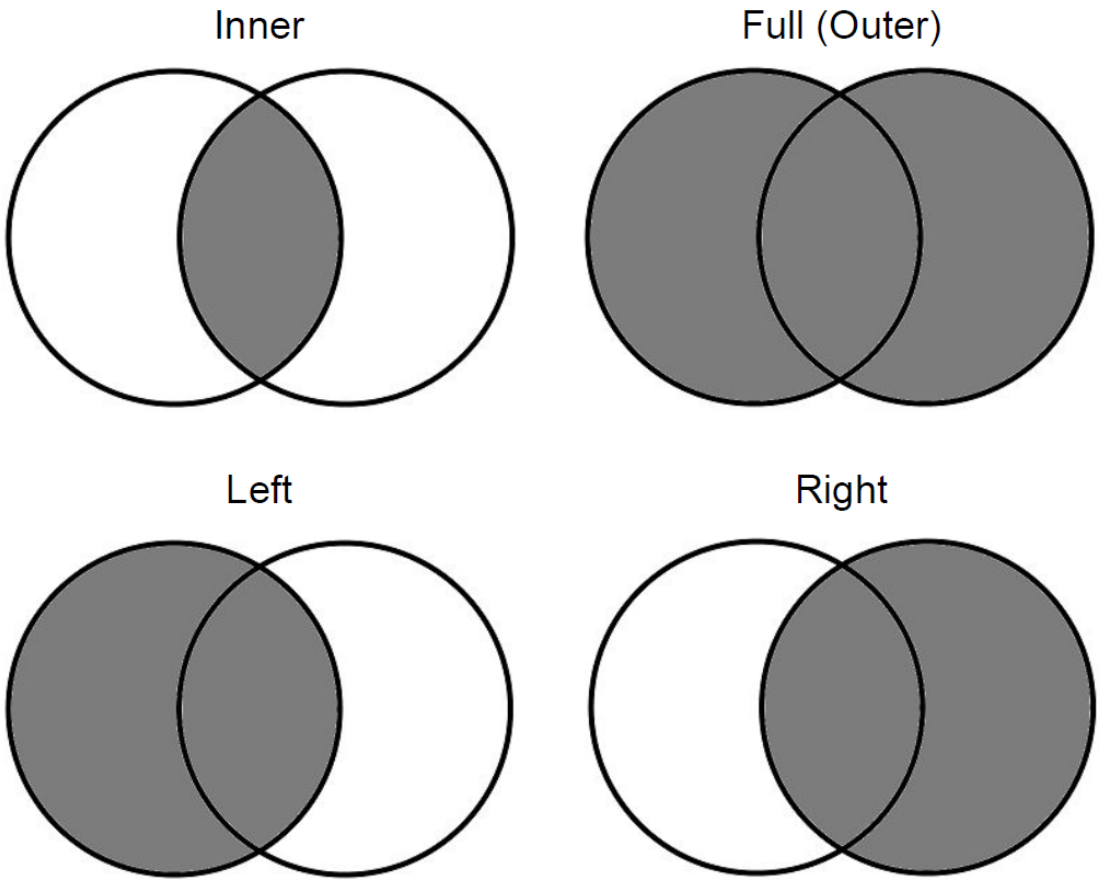
```
##         date datatype  value
## 0 2018-10-01     TMAX   21.1
## 1 2018-10-02     TMAX   23.9
## 2 2018-10-03     TMAX   25.0
## 3 2018-10-04     TMAX   22.8
## 4 2018-10-05     TMAX   23.3
```

Merging data

- When referring to databases, merging is traditionally called a join. There are four types of joins: full (outer), left, right, and inner
- The difference between these functions is what happens when there is a row in one data frame without a corresponding row in the other data frame.
  - `inner join` discards such rows.
  - `full join` always keeps them, filling in missing data with NA.
  - `left join` always keeps rows from the first data frame
  - `right join` always keeps rows from the second data frame

Merging data

- `DataFrame.merge()`: https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.merge.html
- In the following, the darker regions represent the data we are left with after performing the join:



Merging data

```
#suppose country and year determines the ID
data1= pd.DataFrame({'country':["A","A","B","B","C","C"],
  'year': [2017,2018,2017,2017,2017,2018],
  'x1': [1,2,3,4,5,6]})
data2=pd.DataFrame({'country':["A","A","B","B","C"],
  'year': [2017,2018,2017,2018,2018],
  'x2': [7,8,9,10,11]})
```

Merging data

```
print(data1)
```

```
##   country  year  x1
## 0       A  2017   1
## 1       A  2018   2
## 2       B  2017   3
## 3       B  2017   4
## 4       C  2017   5
## 5       C  2018   6
```

```
print(data2)
```

```
##   country  year  x2
## 0       A  2017   7
## 1       A  2018   8
## 2       B  2017   9
## 3       B  2018  10
## 4       C  2018  11
```

Merging data: inner join

- inner join animation
- `inner join` returns matching rows only.

```
pd.merge(data1,data2, how='inner', on=["country","year"])
```

```
##   country  year  x1  x2
## 0       A  2017   1   7
## 1       A  2018   2   8
## 2       B  2017   3   9
## 3       B  2017   4   9
## 4       C  2018   6  11
```

Merging data: full join

full join animation

- `full join` returns all rows. So NA/NaN could be introduced.

```
pd.merge(data1,data2, how='outer', on=["country","year"])
```

```
##   country  year   x1    x2
## 0       A  2017  1.0   7.0
## 1       A  2018  2.0   8.0
## 2       B  2017  3.0   9.0
## 3       B  2017  4.0   9.0
## 4       C  2017  5.0   NaN
## 5       C  2018  6.0  11.0
## 6       B  2018  NaN  10.0
```

Merging data: left join

left_join animation

- `left join` always keeps rows from the first data frame

```
pd.merge(data1,data2, how='left', on=["country","year"])
```

```
##   country  year  x1    x2
## 0       A  2017   1   7.0
## 1       A  2018   2   8.0
## 2       B  2017   3   9.0
## 3       B  2017   4   9.0
## 4       C  2017   5   NaN
## 5       C  2018   6  11.0
```

Merging data: right join

right_join animation

- `right join` always keeps rows from the second data frame

```
rj1=pd.merge(data1,data2, how='right', on=["country","year"])
print(rj1)
```

```
##   country  year   x1  x2
## 0       A  2017  1.0   7
## 1       A  2018  2.0   8
## 2       B  2017  3.0   9
## 3       B  2017  4.0   9
## 4       B  2018  NaN  10
## 5       C  2018  6.0  11
```

Merging data: right join

- The order of the data sets matters

```
rj2=pd.merge(data2,data1, how='right', on=["country","year"])
print(rj2)
```

```
##   country  year    x2  x1
## 0       A  2017   7.0   1
## 1       A  2018   8.0   2
## 2       B  2017   9.0   3
## 3       B  2017   9.0   4
## 4       C  2017   NaN   5
## 5       C  2018  11.0   6
```

Merging data: right join

- `DataFrame.equals()` tests whether two objects contain the same elements.
  - https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.equals.html

```
rj2.equals(rj1)
```

```
## False
```

Aggregating data

- pandas provides a large set of summary functions that operate on different kinds of pandas objects (DataFrame columns, Series, GroupBy, Expanding and Rolling (see below)) and produce single values for each of the groups . When applied to a DataFrame, the result is returned as a pandas Series for each column.
  - https://pandas.pydata.org/pandas-docs/stable/user_guide/basics.html#descriptive-statistics
- Examples:

`sum()`
Sum values of each object.

`count()`
Count non-NA/null values of each object.

`median()`
Median value of each object.

`quantile([0.25,0.75])`
Quantiles of each object.

`apply(function)`
Apply function to each object.

`min()`
Minimum value in each object.

`max()`
Maximum value in each object.

`mean()`
Mean value of each object.

`var()`
Variance of each object.

`std()`
Standard deviation of each object.

Aggregating data

- All such methods have a `skipna` option signaling whether to exclude missing data (True by default):

```
mtcars=pd.read_csv("../data/mtcars.csv")
mtcars['mpg'].mean(skipna=True)
```

```
## 20.090625000000003
```

```
mtcars['mpg'].median()
```

```
## 19.2
```

```
mtcars['mpg'].min()
```

```
## 10.4
```

```
mtcars['mpg'].max()
```

```
## 33.9
```

```
mtcars['mpg'].quantile([0.25,0.75])
```

```
## 0.25    15.425
## 0.75    22.800
## Name: mpg, dtype: float64
```

Aggregating data

- axis=0: apply function to each column. default 0.

```
mtcars['mpg'].var()
```

```
## 36.32410282258064
```

```
mtcars['mpg'].std()
```

```
## 6.026948052089104
```

```
import numpy as np
mtcars.drop(['Unnamed: 0'],axis=1, inplace=True)
mtcars.apply(np.mean, axis=0)
```

```
## mpg      20.090625
## cyl       6.187500
## disp    230.721875
## hp      146.687500
## drat      3.596563
## wt        3.217250
## qsec     17.848750
## vs        0.437500
## am        0.406250
## gear      3.687500
```

Aggregating data

```
mtcars.describe()
```

```
##                mpg        cyl         disp  ...          am       gear      c
## count   32.000000  32.000000   32.000000  ...  32.000000  32.000000  32.0
## mean    20.090625   6.187500  230.721875  ...   0.406250   3.687500   2.8
## std      6.026948   1.785922  123.938694  ...   0.498991   0.737804   1.6
## min     10.400000   4.000000   71.100000  ...   0.000000   3.000000   1.0
## 25%     15.425000   4.000000  120.825000  ...   0.000000   3.000000   2.0
## 50%     19.200000   6.000000  196.300000  ...   0.000000   4.000000   2.0
## 75%     22.800000   8.000000  326.000000  ...   1.000000   4.000000   4.0
## max     33.900000   8.000000  472.000000  ...   1.000000   5.000000   8.0
##
## [8 rows x 11 columns]
```

Aggregating data

- We can get several summary statistics of a column using method `pandas.DataFrame.agg()`: Aggregate using one or more operations over the specified axis.
  - https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.agg.html
- Using `numpy` functions:

```
mtcars['mpg'].agg([np.mean, np.median, np.std])
```

```
## mean      20.090625
## median    19.200000
## std        6.026948
## Name: mpg, dtype: float64
```

- Using pandas methods:

```
mtcars['mpg'].agg(['mean', 'median', 'std'])
```

```
## mean      20.090625
## median    19.200000
## std        6.026948
## Name: mpg, dtype: float64
```

Aggregating data: One-way Frequency Table

- Recall that method `DataFrame.value_counts` can produce a one-way frequency table

```
mtcars['cyl'].unique()
```

```
## array([6, 4, 8], dtype=int64)
```

```
mtcars.sort_values('cyl', ascending=True, inplace=True)
mtcars['cyl'].value_counts(sort=False)
# sort, default True
```

```
## 4    11
## 6     7
## 8    14
## Name: cyl, dtype: int64
```

Aggregating data: Two-way Frequency Table

- Function `pd.crosstab()` computes a simple cross tabulation of two (or more) factors.
  - https://pandas.pydata.org/docs/reference/api/pandas.crosstab.html

```
pd.crosstab(index=mtcars['cyl'],columns='count') #one-way table
```

```
## col_0  count
## cyl
## 4         11
## 6          7
## 8         14
```

```
pd.crosstab(index=mtcars['cyl'],columns=mtcars['am'],margins=True)
```

```
## am    0   1  All
## cyl
## 4     3   8   11
## 6     4   3    7
## 8    12   2   14
## All  19  13   32
```

Aggregating data: Two-way Frequency Table

Three-way Frequency Table:

- Function `pd.crosstab()` computes a simple cross tabulation of two (or more) factors.
  - https://pandas.pydata.org/docs/reference/api/pandas.crosstab.html

```
pd.crosstab(index=[mtcars['am'], mtcars['vs']],columns=mtcars['cyl'])
# use Hierarchical / Multi-level indexing create 3-way table
```

```
## cyl    4  6   8
## am vs
## 0  0   0  0  12
##    1   3  4   0
## 1  0   1  3   2
##    1   7  0   0
```

Aggregating data: Two-way Frequency Table

- We can find the marginal proportions of the table using argument `normalize=True`.

```
pd.crosstab(index=mtcars['cyl'],columns='count',normalize=True)
```

```
## col_0    count
## cyl
## 4      0.34375
## 6      0.21875
## 8      0.43750
```

```
pd.crosstab(index=mtcars['cyl'],columns=mtcars['am'],
margins=True,normalize=True)
```

```
## am          0        1      All
## cyl
## 4     0.09375  0.25000  0.34375
## 6     0.12500  0.09375  0.21875
## 8     0.37500  0.06250  0.43750
## All   0.59375  0.40625  1.00000
```

Aggregating data: Two-way Frequency Table

- We can find the marginal proportions of the table using argument `normalize=True`.

```
pd.crosstab(index=[mtcars['am'], mtcars['vs']],
columns=mtcars['cyl'],margins=True,normalize=True)
```

```
## cyl            4        6       8      All
## am  vs
## 0   0    0.00000  0.00000  0.3750  0.37500
##     1    0.09375  0.12500  0.0000  0.21875
## 1   0    0.03125  0.09375  0.0625  0.18750
##     1    0.21875  0.00000  0.0000  0.21875
## All      0.34375  0.21875  0.4375  1.00000
```

Aggregating data

- A lot of times, we want to split a data into subsets based on a categorical variable, computes summary statistics for each, and returns the result in a convenient form.
- This can be done by pandas's `groupby()` method.
  - https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.groupby.html
- One-way Frequency table
  - `pandas.Series.to_frame` Converts Series to DataFrame.

```
table1=mtcars.groupby(['cyl']).mpg.count()
table1.reset_index()
```

```
##    cyl  mpg
## 0    4   11
## 1    6    7
## 2    8   14
```

```
table1=table1.to_frame()
table1.rename(columns={'mpg': 'freq'}, inplace=True)
table1.head()
```

```
##      freq
## cyl
## 4      11
```

Aggregating data

- Method `pandas.DataFrame.agg()`: Aggregate using one or more operations over the specified axis.
  - https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.agg.html

```
#Aggregate functions over column(s) and rename the index of the resulting
table1_1=mtcars.groupby(['cyl']).agg(freq=("mpg",'count'))
table1_1.head()
```

```
##      freq
## cyl
## 4      11
## 6       7
## 8      14
```

Aggregating data

```
mtcars.groupby(['cyl']).mpg.mean()
# mtcars.groupby(['cyl'])['mpg'].mean()
```

```
## cyl
## 4    26.663636
## 6    19.742857
## 8    15.100000
## Name: mpg, dtype: float64
```

```
mtcars.groupby(['cyl']).agg(mean_mpg=("mpg",'mean'))
```

```
##        mean_mpg
## cyl
## 4    26.663636
## 6    19.742857
## 8    15.100000
```

```
mtcars.groupby(['cyl']).agg(mean_mpg=("mpg",np.mean))
```

```
##        mean_mpg
## cyl
## 4    26.663636
## 6    19.742857
## 8    15.100000
```

Aggregating data

- We can group by several categorical variables
- Two-way frequency table

```
table2=mtcars.groupby(['cyl', 'am']).mpg.count()
table2.reset_index()
```

```
##    cyl  am  mpg
## 0    4   0    3
## 1    4   1    8
## 2    6   0    4
## 3    6   1    3
## 4    8   0   12
## 5    8   1    2
```

```
table2=table2.to_frame()
table2.rename(columns={'mpg': 'freq'}, inplace=True)
table2.head()
```

```
##          freq
## cyl am
## 4   0       3
##     1       8
## 6   0       4
```

Aggregating data

```
#Aggregate functions over column(s) and rename the index of the resulting
table2_1=mtcars.groupby(['cyl', 'am']).agg(freq=("mpg",'count'))
table2_1.head()
```

```
##          freq
## cyl am
## 4   0       3
##     1       8
## 6   0       4
##     1       3
## 8   0      12
```

Aggregating data

- We can group by several categorical variables

```
mtcars.groupby(['cyl', 'am']).mpg.mean()
```

```
## cyl  am
## 4    0     22.900000
##      1     28.075000
## 6    0     19.125000
##      1     20.566667
## 8    0     15.050000
##      1     15.400000
## Name: mpg, dtype: float64
```

```
mtcars.groupby(['cyl','am']).agg(mean_mpg=("mpg",'mean'))
#mtcars.groupby(['cyl','am']).agg(mean_mpg=("mpg",np.mean))
```

```
##          mean_mpg
## cyl am
## 4   0   22.900000
##     1   28.075000
## 6   0   19.125000
##     1   20.566667
## 8   0   15.050000
```

Aggregating data

- Calculate both mean and standard deviation

```
import numpy as np
mtcars.groupby(['cyl']).mpg.apply(lambda x: [np.mean(x), np.std(x)])
```

```
## cyl
## 4     [26.66363636363636, 4.2999519505291 57]
## 6    [19.74285714285714, 1.3457415829806494]
## 8                  [15.1, 2.4669240534954 22]
## Name: mpg, dtype: object
```

```
mtcars.groupby(['cyl']).mpg.apply(lambda x: [np.min(x), np.max(x)])
```

```
## cyl
## 4    [21.4, 33.9]
## 6    [17.8, 21.4]
## 8    [10.4, 19.2]
## Name: mpg, dtype: object
```

Aggregating data

```
mtcars.groupby(['cyl']).mpg.idxmin()
```

```
## cyl
## 4    31
## 6    10
## 8    14
## Name: mpg, dtype: int64
```

```
mtcars.groupby(['cyl']).mpg.idxmax()
```

```
## cyl
## 4    19
## 6     3
## 8    24
## Name: mpg, dtype: int64
```

```
mtcars.groupby(['cyl']).mpg.apply(lambda x: [np.min(x), np.max(x)])
# mtcars.loc[31,'mpg']
```

```
## cyl
## 4    [21.4, 33.9]
## 6    [17.8, 21.4]
## 8    [10.4, 19.2]
## Name: mpg, dtype: object
```

Aggregating data

- Or we use the lambda function

```
mtcars.groupby(['cyl']).mpg.apply(lambda x: [np.min(x), np.max(x)])
```

```
## cyl
## 4    [21.4, 33.9]
## 6    [17.8, 21.4]
## 8    [10.4, 19.2]
## Name: mpg, dtype: object
```

```
mtcars.groupby(['cyl']).mpg.apply(lambda x: [x.idxmin(),x.idxmax()])
#get_minimum = lambda x: x.idxmin()
#get_maximum = lambda x: x.idxmax()
#mtcars.groupby(['cyl']).mpg.apply(lambda x: [get_minimum(x),get_maximum(x)])
```

```
## cyl
## 4    [31, 19]
## 6     [10, 3]
## 8    [14, 24]
## Name: mpg, dtype: object
```

Aggregating data

- The summary statistics can be added as new columns using `pandas.DataFrame.transform()`
  - http://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.transform.html

```
mtcars['mean_mpg']=mtcars.groupby(['cyl'])['mpg'].transform('mean')
mtcars[['mpg','cyl','mean_mpg']]
```

```
##       mpg  cyl   mean_mpg
## 31   21.4    4  26.663636
## 2    22.8    4  26.663636
## 27   30.4    4  26.663636
## 26   26.0    4  26.663636
## 25   27.3    4  26.663636
## 20   21.5    4  26.663636
## 7    24.4    4  26.663636
## 8    22.8    4  26.663636
## 19   33.9    4  26.663636
## 18   30.4    4  26.663636
## 17   32.4    4  26.663636
## 29   19.7    6  19.742857
## 0    21.0    6  19.742857
```

License