ISOM 2600 Business Analytics

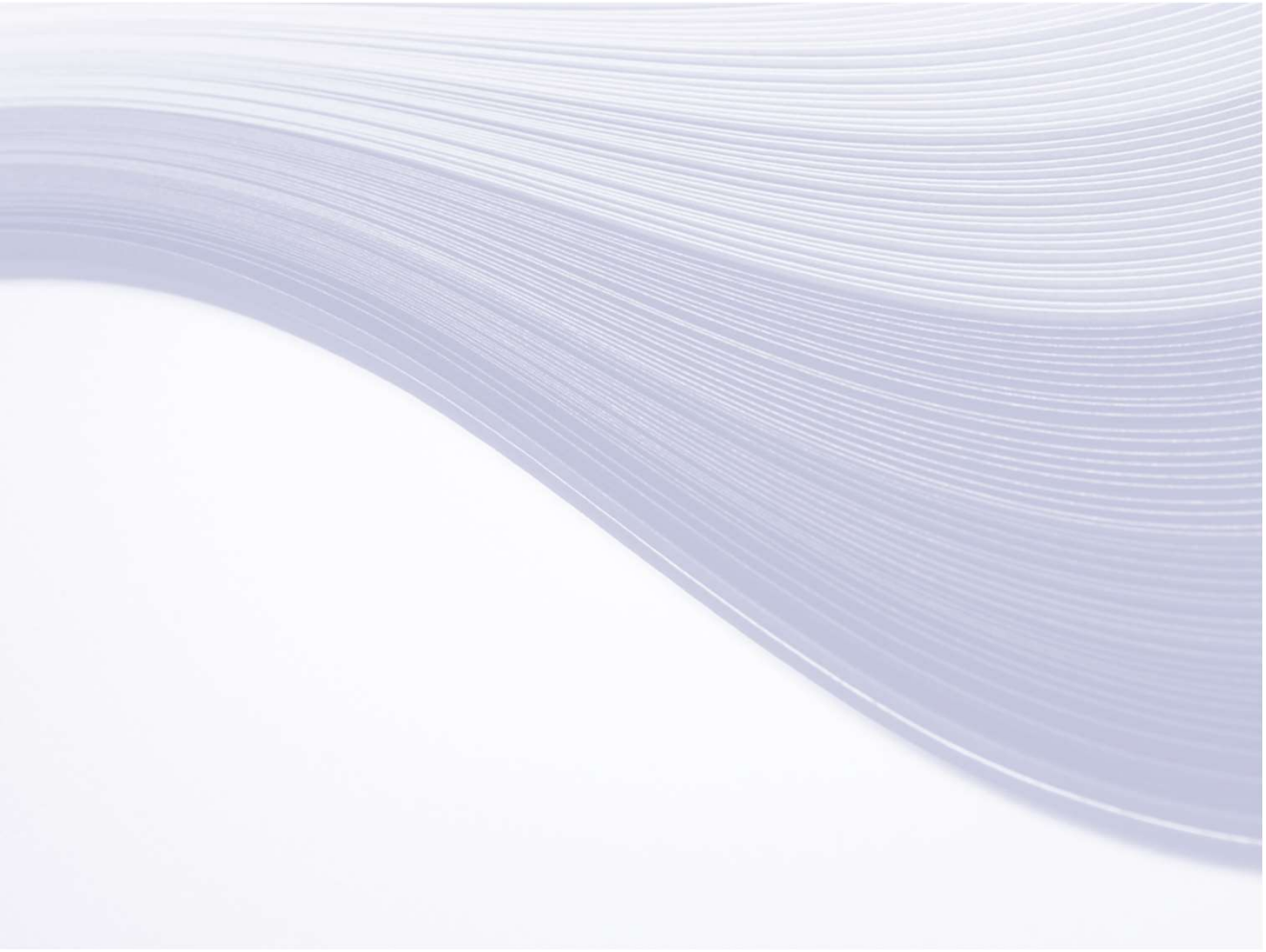
TOPIC 2: PANDAS AND DATA PREPROCESSING

XUHU WAN

HKUST

JANUARY 15, 2022

Contents



Pandas

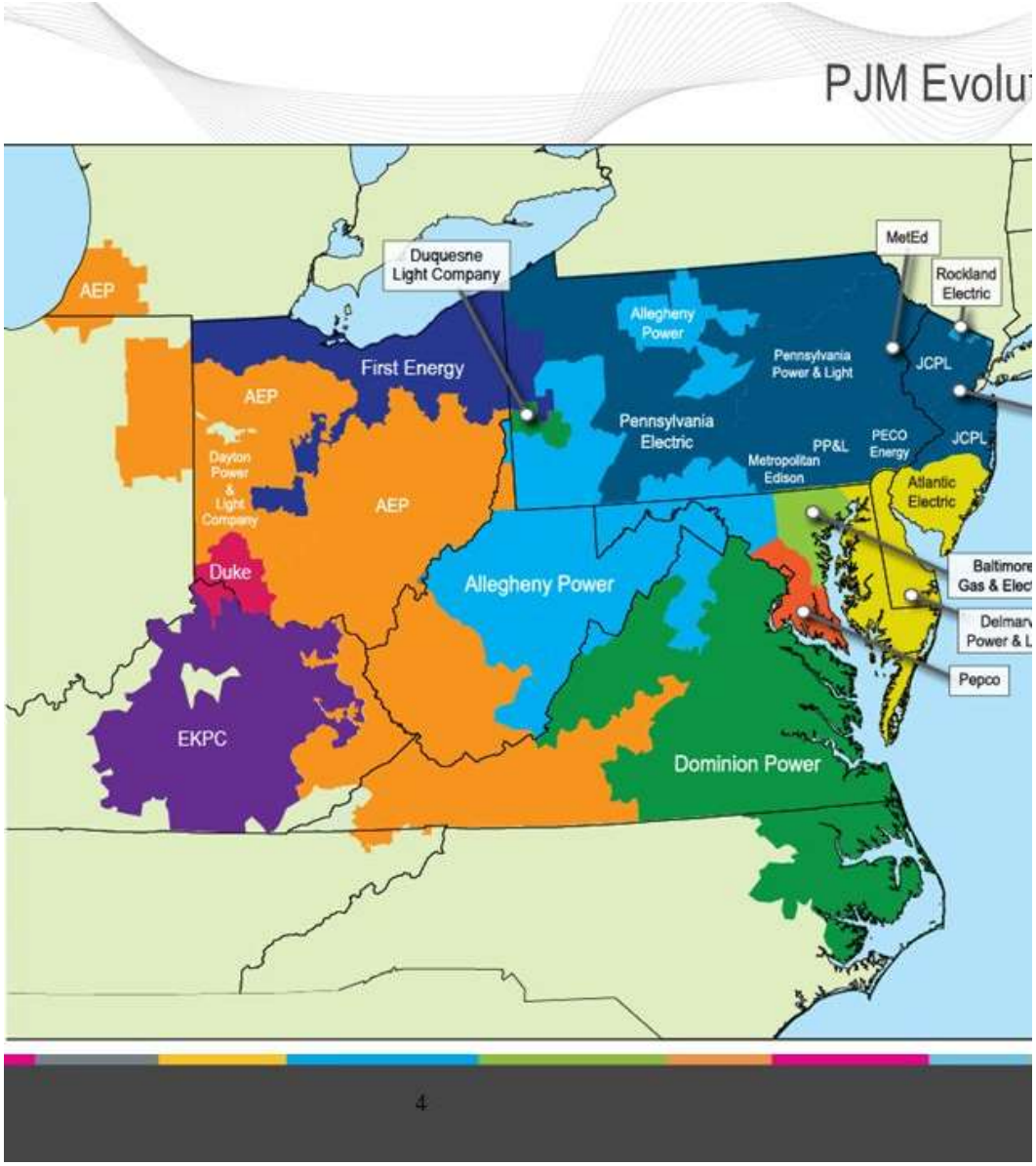
csv Data File

Information is separated with a comma. The comma is not the only type of delimiter, however. Some files are delimited by a tab (TSV) or even a semicolon. The main reason why CSVs are a preferred data format when collaborating and sharing data is because any program can open this kind of data structure. It can even be opened in a text editor.

Index: date and hour

Columns: Hourly power consumption in Different regions of PJM LLC.

PJM Interconnection LLC (PJM) is a regional transmission organization (RTO) in the United States. It is part of the Eastern Interconnection grid operating an electric transmission system serving all or parts of Delaware, Illinois, Indiana, Kentucky, Maryland, Michigan, New Jersey, North Carolina, Ohio, Pennsylvania, Tennessee, Virginia, West Virginia, and the District of Columbia.



Pandas DataFrame

Pandas is a library for data analysis. It gives Python the ability to work with spreadsheet-like data for fast data loading, manipulating, aligning, and merging, among other functions. In the first lecture , we will focus on DataFrame only and learn to read data file and do column-wise operation.

```
import pandas as pd
```

Import pandas and read data

```
df = pd.read_csv(data_path,index_col=0)
df.index=pd.to_datetime(df.index) # Orgi
```

Column

Index

```
df.tail()
```

Datetime	AEP	COMED	DAYTON	DEOK	DOM	DUQ	EKPC	FE	NI	PJME	PJMW	PJM_Load
2018-01-01 20:00:00	21089.0	13858.0	2732.0	4426.0	18418.0	1962.0	2866.0	9378.0	NaN	44284.0	8401.0	NaN
2018-01-01 21:00:00	20999.0	13758.0	2724.0	4419.0	18567.0	1940.0	2846.0	9255.0	NaN	43751.0	8373.0	NaN
2018-01-01 22:00:00	20820.0	13627.0	2664.0	4355.0	18307.0	1891.0	2883.0	9044.0	NaN	42402.0	8238.0	NaN
2018-01-01 23:00:00	20415.0	13336.0	2614.0	4224.0	17814.0	1820.0	2880.0	8676.0	NaN	40164.0	7958.0	NaN
2018-01-02 00:00:00	19993.0	12816.0	2552.0	4100.0	17428.0	1721.0	2846.0	8393.0	NaN	38608.0	7691.0	NaN

Series and DataFrame

Pandas introduces two data types to Python: Series and DataFrame. The DataFrame represents your entire spreadsheet or rectangular data, whereas the Series is a single column of the DataFrame. A Pandas DataFrame can also be thought of as a dictionary or collection of Series objects . The DataFrame has Columns and Index.

```
df=pd.read_csv("data/pjm.csv",index_col=0)
```

A Series is a column of DataFrame without column name

```
type(df)
```

```
pandas.core.frame.DataFrame
```

```
type(df["PJME"])
```

```
pandas.core.series.Series
```

Columns and Index

Index

```
df.index
```

```
Index(['1998-12-31 01:00:00', '1998-12-31 02:00:00', '1998-12-31 03:00:00',
       '1998-12-31 04:00:00', '1998-12-31 05:00:00', '1998-12-31 06:00:00',
       '1998-12-31 07:00:00', '1998-12-31 08:00:00', '1998-12-31 09:00:00',
       '1998-12-31 10:00:00',
       ...
       '2018-01-01 15:00:00', '2018-01-01 16:00:00', '2018-01-01 17:00:00',
       '2018-01-01 18:00:00', '2018-01-01 19:00:00', '2018-01-01 20:00:00',
       '2018-01-01 21:00:00', '2018-01-01 22:00:00', '2018-01-01 23:00:00',
       '2018-01-02 00:00:00'],
      dtype='object', name='Datetime', length=178262)
```

Column names

```
df.columns
```

```
Index(['AEP', 'COMED', 'DAYTON', 'DEOK', 'DOM', 'DUQ', 'EKPC', 'FE', 'NI',
       'PJME', 'PJMW', 'PJM_Load'],
      dtype='object')
```

Dimension of data

```
1  df.shape
```

```
(178262, 12)
```

```
1  len(df)
```

```
178262
```

Convert string to Datetime

You read time series data from csv file and index is in the format of year-month-days. However it is not in the type of datetime and their type is string. We need to transform the index using pd.to_datetime

```
1  str1="2017-01-12"
2  pd.to_datetime(str1)
```

```
Timestamp('2017-01-12 00:00:00')
```

```
df = pd.read_csv(data_path,index_col=0)
df.index=pd.to_datetime(df.index) # Orgi
```

If the index is converted to datetime type, we can get day, month ,year and time easily

```
df.index[0].year,df.index[0].month,df.index[0].day,df.index[0].weekday()
```

```
(1998, 12, 31, 3)
```

Data Slicing

Data Slicing (Subsetting)

The data is usually large and it is better to check details by displaying or make a copy of a part of the data.

- We can use the head or tail method to look at the first or last the several rows . This is useful to see if the data is loaded correctly and has a sense of the data contents, i.e. index, columns, values etc.
- We can select one or multiple columns to compare different variables
- We can select one or more rows to check data pattern only in certain time window.
- Mixed selections of rows and columns
- Selection of data following certain conditions

Head or Tail

```
df.head(3)
```

Datetime	AEP	COMED	DAYTON	DEOK	DOM	DUQ	EKPC	FE	NI	PJME	PJMW	PJM_Load
1998-12-31 01:00:00	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	29309.0
1998-12-31 02:00:00	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	28236.0
1998-12-31 03:00:00	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	27692.0

```
df.tail(2)
```

Datetime	AEP	COMED	DAYTON	DEOK	DOM	DUQ	EKPC	FE	NI	PJME	PJMW	PJM_Load
2018-01-01 23:00:00	20415.0	13336.0	2614.0	4224.0	17814.0	1820.0	2880.0	8676.0	NaN	40164.0	7958.0	NaN
2018-01-02 00:00:00	19993.0	12816.0	2552.0	4100.0	17428.0	1721.0	2846.0	8393.0	NaN	38608.0	7691.0	NaN

Selection of Columns

SELECT ONE COLUMN

```
df["AEP"]
```

```
Datetime
1998-12-31 01:00:00        NaN
1998-12-31 02:00:00        NaN
1998-12-31 03:00:00        NaN
1998-12-31 04:00:00        NaN
1998-12-31 05:00:00        NaN
                          ...
2018-01-01 20:00:00    21089.0
2018-01-01 21:00:00    20999.0
2018-01-01 22:00:00    20820.0
2018-01-01 23:00:00    20415.0
2018-01-02 00:00:00    19993.0
Name: AEP, Length: 178262, dtype: float64
```

SELECT MULTIPLE COLUMN

```
df[["AEP","COMED"]]
```

Datetime	AEP	COMED
1998-12-31 01:00:00	NaN	NaN
1998-12-31 02:00:00	NaN	NaN
1998-12-31 03:00:00	NaN	NaN
1998-12-31 04:00:00	NaN	NaN
1998-12-31 05:00:00	NaN	NaN
...	...	...
2018-01-01 20:00:00	21089.0	13858.0
2018-01-01 21:00:00	20999.0	13758.0
2018-01-01 22:00:00	20820.0	13627.0
2018-01-01 23:00:00	20415.0	13336.0
2018-01-02 00:00:00	19993.0	12816.0

Selection of Rows

LOC METHOD: ROW NAMES

```
df.loc["2018-01-01 22:00:00"]
```

```
AEP          20820.0
COMED        13627.0
DAYTON        2664.0
DEOK          4355.0
DOM          18307.0
DUQ           1891.0
EKPC          2883.0
FE            9044.0
NI               NaN
PJME         42402.0
PJMW          8238.0
PJM_Load         NaN
Name: 2018-01-01 22:00:00, dtype: float64
```

ILOC METHOD: ROW NUMBERS

```
df.iloc[-3]
```

```
AEP          20820.0
COMED        13627.0
DAYTON        2664.0
DEOK          4355.0
DOM          18307.0
DUQ           1891.0
EKPC          2883.0
FE            9044.0
NI               NaN
PJME         42402.0
PJMW          8238.0
PJM_Load         NaN
Name: 2018-01-01 22:00:00, dtype: float64
```

```
df.loc["2018-01-01 22:00:00":"2018-01-02 00:00:00"]
```

Datetime	AEP	COMED	DAYTON	DEOK	DOM	DUQ	EKPC	FE	NI	PJME	PJMW	PJM_Load
2018-01-01 22:00:00	20820.0	13627.0	2664.0	4355.0	18307.0	1891.0	2883.0	9044.0	NaN	42402.0	8238.0	NaN
2018-01-01 23:00:00	20415.0	13336.0	2614.0	4224.0	17814.0	1820.0	2880.0	8676.0	NaN	40164.0	7958.0	NaN
2018-01-02 00:00:00	19993.0	12816.0	2552.0	4100.0	17428.0	1721.0	2846.0	8393.0	NaN	38608.0	7691.0	NaN

```
df.iloc[-3:]
```

Datetime	AEP	COMED	DAYTON	DEOK	DOM	DUQ	EKPC	FE	NI	PJME	PJMW	PJM_Load
2018-01-01 22:00:00	20820.0	13627.0	2664.0	4355.0	18307.0	1891.0	2883.0	9044.0	NaN	42402.0	8238.0	NaN
2018-01-01 23:00:00	20415.0	13336.0	2614.0	4224.0	17814.0	1820.0	2880.0	8676.0	NaN	40164.0	7958.0	NaN
2018-01-02 00:00:00	19993.0	12816.0	2552.0	4100.0	17428.0	1721.0	2846.0	8393.0	NaN	38608.0	7691.0	NaN

ATTN: Loc method is right inclusive and iloc is right exclusive

Mixed Selection

```
df.loc["2018-01-01 22:00:00":"2018-01-02 00:00:00", ["AEP","COMED","DAYTON"]]
```

Datetime	AEP	COMED	DAYTON
2018-01-01 22:00:00	20820.0	13627.0	2664.0
2018-01-01 23:00:00	20415.0	13336.0	2614.0
2018-01-02 00:00:00	19993.0	12816.0	2552.0

```
df.iloc[-5:,-5:]
```

Datetime	FE	NI	PJME	PJMW	PJM_Load
2018-01-01 20:00:00	9378.0	NaN	44284.0	8401.0	NaN
2018-01-01 21:00:00	9255.0	NaN	43751.0	8373.0	NaN
2018-01-01 22:00:00	9044.0	NaN	42402.0	8238.0	NaN
2018-01-01 23:00:00	8676.0	NaN	40164.0	7958.0	NaN
2018-01-02 00:00:00	8393.0	NaN	38608.0	7691.0	NaN

Split data into train and test data (80% train and 20% test)

If the data is time series, the first 80% rows as train and the most recent 20% as the test data.

```
trainsize=int(0.8*len(df))
testsize=len(df)-trainsize
trainsize,testsize
```

```
(142609, 35653)
```

Slice data using iloc method

```
traindata=df.iloc[:trainsize]
testdata=df.iloc[trainsize:]
```

If the data is cross-sectional (not time series), we need to shuffle data first before selecting using iloc as shown in the left.

```
from sklearn.utils import shuffle
shuffleddata=  shuffle(profitdata)
```

	Unnamed: 0.1	Location	Profit	Income
27	27	Cumberland,MD-WV	201940.0	22719
62	62	LittleRock-NorthLittleRock,AR	152840.0	25392

```
trainsize=int(0.8*len(shuffleddata))
testsize=len(shuffleddata)-trainsize
traindata=shuffleddata.iloc[:trainsize]
testdata=shuffleddata.iloc[trainsize:]
```

Conditional Selection

We also may select part of data which satisfy certain conditions. For example, select all data, such that income is above median

```
profitdata["Income"].median()
```

```
24939.0
```

```
median_income=profitdata["Income"].median()
profitdata[profitdata["Income"]>median_income]
```

Profit	Income	Disposable Income	Birth Rate (per 1,000)
199780.0	28719	22701	13.1
165530.0	25835	19890	15.8
166890.0	33501	27031	18.5

Making Changes to Data

Add Columns

```
df["Month"]=df.index.month
df[["AEP","COMED","Month"]].tail(5)
```

Datetime	AEP	COMED	Month
2018-01-01 20:00:00	21089.0	13858.0	1
2018-01-01 21:00:00	20999.0	13758.0	1
2018-01-01 22:00:00	20820.0	13627.0	1
2018-01-01 23:00:00	20415.0	13336.0	1
2018-01-02 00:00:00	19993.0	12816.0	1

Change Column directly

We make a new column “Season”

```
df["Season"]=[1 if t <4 else 2 if t<7 else 3 if t <10 else 4 for t in df["Month"] ]
```

```
df["Season"].value_counts()
```

```
2    45854
3    44953
4    44147
1    43308
Name: Season, dtype: int64
```

Change the new column directly

```
df["Season"]=["spring" if t==1 else "summer" if t==2 else "fall" if t==3 else "winter" for t in df["Season"]]
df["Season"].value_counts()
```

```
summer    45854
fall      44953
winter    44147
spring    43308
Name: Season, dtype: int64
```

Value_counts() is to list the frequencies of all values.

Dropping Column/Row

Drop Column

```
df.drop(["WeekDay","Month","Season"],axis=1)
```

Datetime	AEP	COMED	DAYTON	DEOK	DOM	DUQ	EKPC	FE	NI	PJME	PJMW	PJM_Load
1998-12-31 01:00:00	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	29309.0
1998-12-31 02:00:00	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	28236.0
1998-12-31 03:00:00	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	27692.0
1998-12-31 04:00:00	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	27596.0
1998-12-31 05:00:00	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	27888.0

Drop Row

```
df.drop(df.index[0:10],axis=0).head()
```

Datetime	AEP	COMED	DAYTON	DEOK	DOM	DUQ	EKPC	FE	NI	PJME	PJMW	PJM_Load	WeekDay	Month	Season
1998-12-31 11:00:00	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	35401.0	3	12	winter
1998-12-31 12:00:00	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	35331.0	3	12	winter
1998-12-31 13:00:00	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	34582.0	3	12	winter
1998-12-31 14:00:00	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	33767.0	3	12	winter

Handling Missing Data

What is NaN Value

- Rarely you will get a data set without any missing values. Pandas displays missing values as NaN. The NaN value in Pandas comes from numpy. Missing values may be used or displayed in a few ways in Python—NaN, NAN, or nan—but they are all equivalent.
- We can get NaN when we load in a data set with missing values, or from the data preprocessing
- NaN value may be useful in data preprocessing.

Count Missing Values

```
df.isnull()
```

Datetime	AEP	COMED	DAYTON	DEOK	DOM	DUQ	EKPC	FE	NI	PJME	PJMW	PJM_Load	WeekDay	Month	Season
1998-12-31 01:00:00	True	True	True	True	True	True	True	True	True	True	True	False	False	False	False
1998-12-31 02:00:00	True	True	True	True	True	True	True	True	True	True	True	False	False	False	False
1998-12-31 03:00:00	True	True	True	True	True	True	True	True	True	True	True	False	False	False	False
1998-12-31 04:00:00	True	True	True	True	True	True	True	True	True	True	True	False	False	False	False
1998-12-31 05:00:00	True	True	True	True	True	True	True	True	True	True	True	False	False	False	False

```
df.isnull().sum()
```

```
AEP          56989
COMED       111765
DAYTON       56987
DEOK        120523
DOM          62073
DUQ          59194
EKPC        132928
FE          115388
NI          119812
PJME         32896
PJMW         35056
PJM_Load    145366
WeekDay          0
Month            0
Season           0
dtype: int64
```

```
df.isnull().sum().sum()
```

```
1048977
```

Replace NaN Values

We can use the fillna() method to recode the missing values to another value. For example, suppose we wanted the missing values to be recoded as a 0 or mode/median of the column.

```
df.fillna(0)
```

Datetime	AEP	COMED	DAYTON	DEOK	DOM	DUQ	EKPC	FE	NI	PJME	PJMW	PJM_Load	V
1998-12-31 01:00:00	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	29309.0	
1998-12-31 02:00:00	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	28236.0	
1998-12-31 03:00:00	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	27692.0	
1998-12-31 04:00:00	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	27596.0	
1998-12-31 05:00:00	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	27888.0	
...	...	...	...	...	...	...	...	...	...	...	...	...	
2018-01-01 20:00:00	21089.0	13858.0	2732.0	4426.0	18418.0	1962.0	2866.0	9378.0	0.0	44284.0	8401.0	0.0	
2018-01-01 21:00:00	20999.0	13758.0	2724.0	4419.0	18567.0	1940.0	2846.0	9255.0	0.0	43751.0	8373.0	0.0	

Fill Missing Values with Fill Forward Method

```
newdf=pd.DataFrame(index=df.index[0:10],columns=["A","B"])
newdf.loc[newdf.index[3]]=3
newdf.loc[newdf.index[4]]=4
newdf
```

Datetime	A	B
1998-12-31 01:00:00	NaN	NaN
1998-12-31 02:00:00	NaN	NaN
1998-12-31 03:00:00	NaN	NaN
1998-12-31 04:00:00	3	3
1998-12-31 05:00:00	4	4
1998-12-31 06:00:00	NaN	NaN
1998-12-31 07:00:00	NaN	NaN
1998-12-31 08:00:00	NaN	NaN
1998-12-31 09:00:00	NaN	NaN
1998-12-31 10:00:00	NaN	NaN

```
newdf.fillna(method="ffill")
```

Datetime	A	B
1998-12-31 01:00:00	NaN	NaN
1998-12-31 02:00:00	NaN	NaN
1998-12-31 03:00:00	NaN	NaN
1998-12-31 04:00:00	3.0	3.0
1998-12-31 05:00:00	4.0	4.0
1998-12-31 06:00:00	4.0	4.0
1998-12-31 07:00:00	4.0	4.0
1998-12-31 08:00:00	4.0	4.0
1998-12-31 09:00:00	4.0	4.0
1998-12-31 10:00:00	4.0	4.0

Fill Missing Values with Fill BackWard

```
newdf=pd.DataFrame(index=df.index[0:10],columns=["A","B"])
newdf.loc[newdf.index[3]]=3
newdf.loc[newdf.index[4]]=4
newdf
```

Datetime	A	B
1998-12-31 01:00:00	NaN	NaN
1998-12-31 02:00:00	NaN	NaN
1998-12-31 03:00:00	NaN	NaN
1998-12-31 04:00:00	3	3
1998-12-31 05:00:00	4	4
1998-12-31 06:00:00	NaN	NaN
1998-12-31 07:00:00	NaN	NaN
1998-12-31 08:00:00	NaN	NaN
1998-12-31 09:00:00	NaN	NaN
1998-12-31 10:00:00	NaN	NaN

```
newdf.fillna(method="bfill")
```

Datetime	A	B
1998-12-31 01:00:00	3.0	3.0
1998-12-31 02:00:00	3.0	3.0
1998-12-31 03:00:00	3.0	3.0
1998-12-31 04:00:00	3.0	3.0
1998-12-31 05:00:00	4.0	4.0
1998-12-31 06:00:00	NaN	NaN
1998-12-31 07:00:00	NaN	NaN
1998-12-31 08:00:00	NaN	NaN
1998-12-31 09:00:00	NaN	NaN
1998-12-31 10:00:00	NaN	NaN

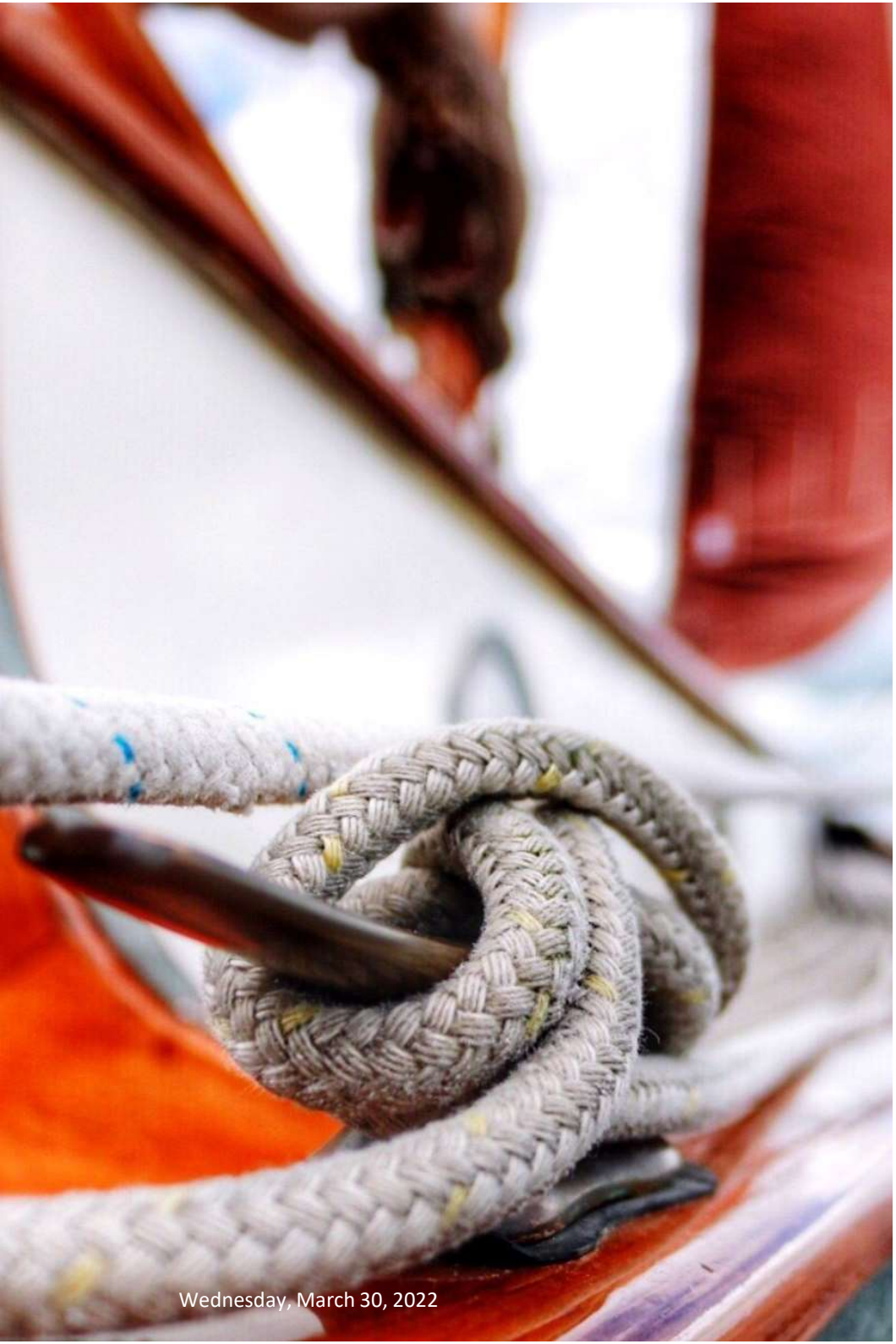
Drop Missing Values

```
newdf=pd.DataFrame(index=df.index[0:10],columns=["A","B"])
newdf.loc[newdf.index[3]]=3
newdf.loc[newdf.index[4]]=4
newdf
```

Datetime	A	B
1998-12-31 01:00:00	NaN	NaN
1998-12-31 02:00:00	NaN	NaN
1998-12-31 03:00:00	NaN	NaN
1998-12-31 04:00:00	3	3
1998-12-31 05:00:00	4	4
1998-12-31 06:00:00	NaN	NaN
1998-12-31 07:00:00	NaN	NaN
1998-12-31 08:00:00	NaN	NaN
1998-12-31 09:00:00	NaN	NaN
1998-12-31 10:00:00	NaN	NaN

```
newdf.dropna()
```

Datetime	A	B
1998-12-31 04:00:00	3	3
1998-12-31 05:00:00	4	4



Standardization of Data

Standardization/Normalization

Max-Min Standardization

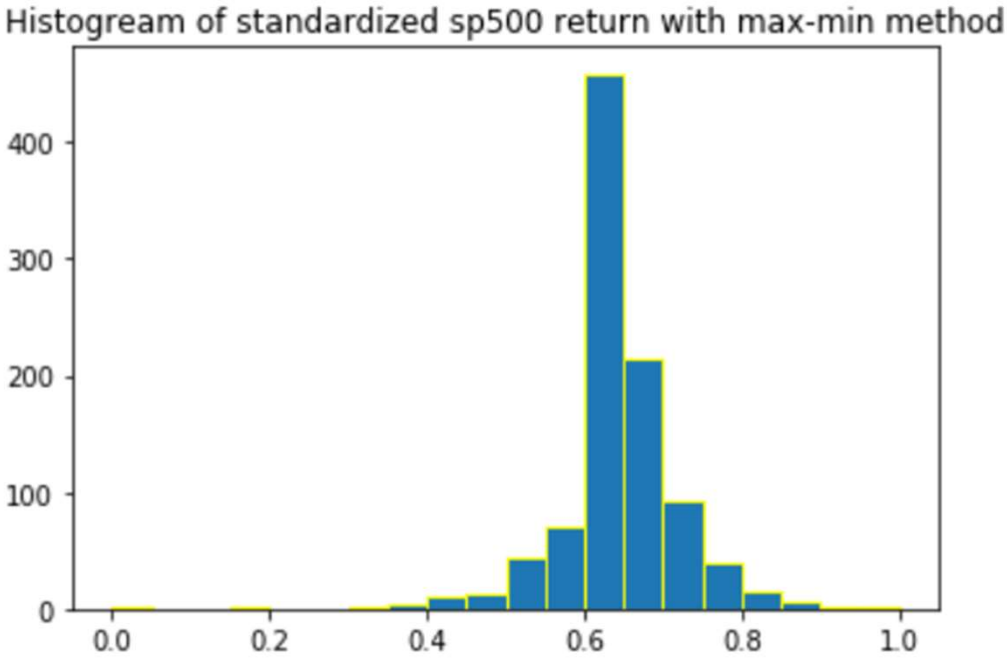
Mean-standard deviation

Max-min Standardization

Standardize data into those with the range [0,1]

$$\frac{x - min}{max - min}$$

	sp500_return	max-min
2013-09-11	0.007346	0.709435
2013-09-12	0.003052	0.670059
2013-09-13	-0.003380	0.611063
2013-09-17	0.008423	0.719317
2013-09-18	0.004218	0.680747
...	...	...
2017-08-29	0.002161	0.661882
2017-08-30	0.000843	0.649795
2017-08-31	0.004615	0.684392
2017-09-01	0.005721	0.694534
2017-09-04	0.000000	0.642066

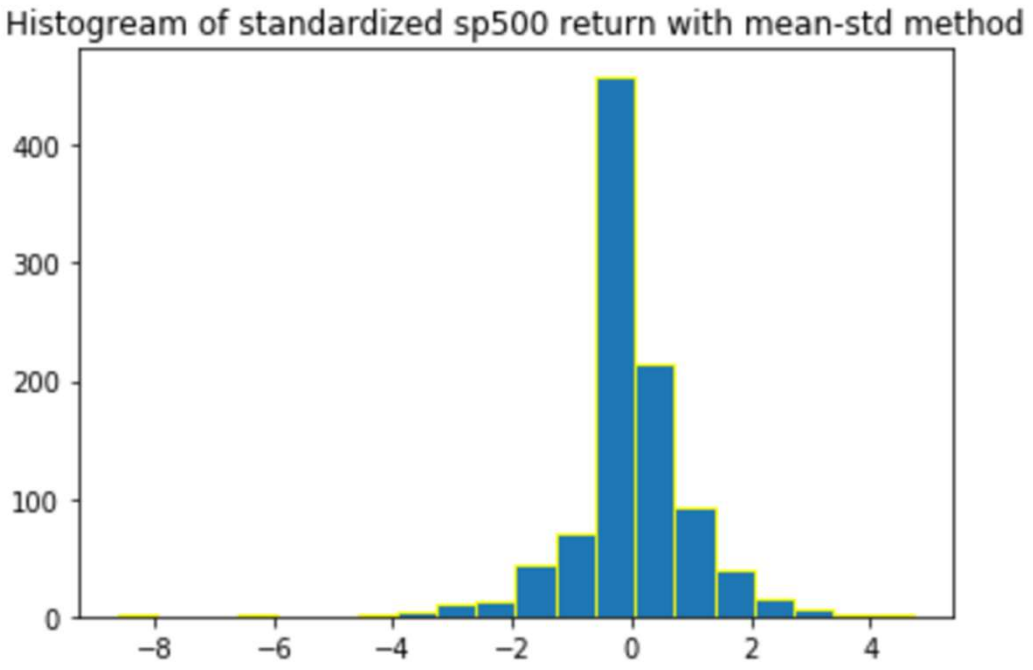


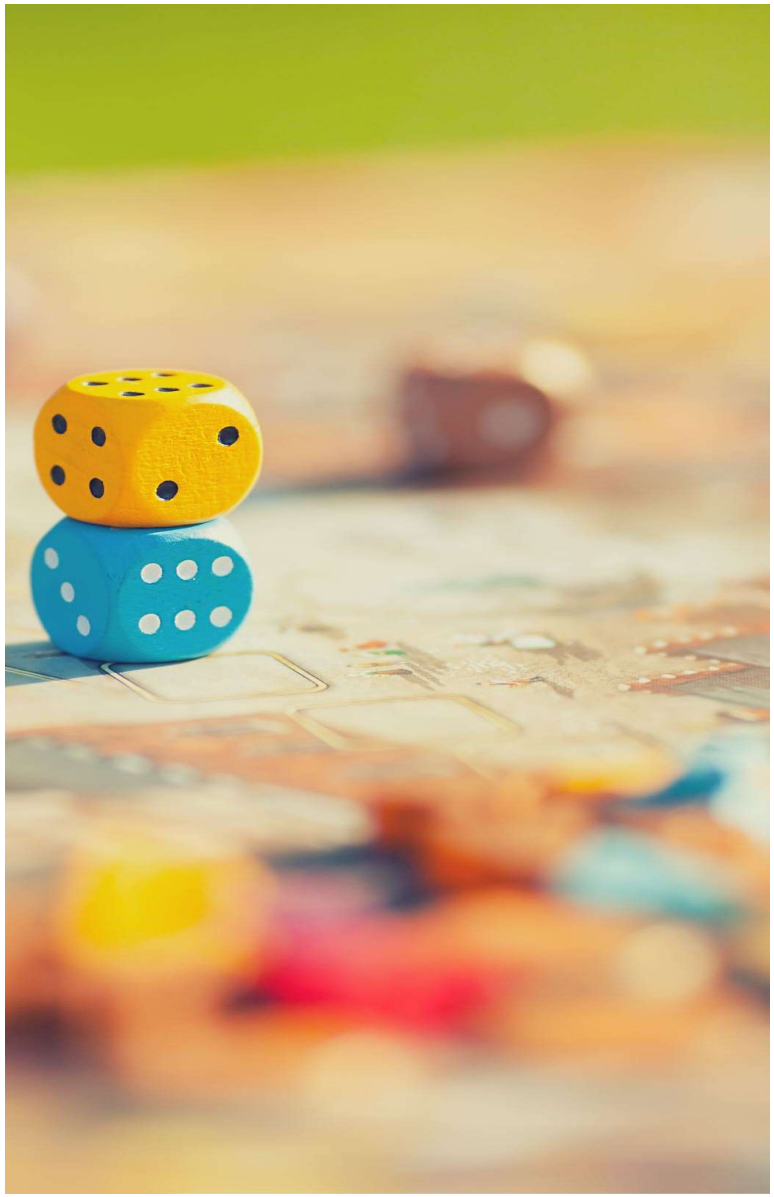
Mean-SD Standardization

The range of standardized data is not bounded but most of data is in between -3 and 3. It is applicable for the bell-shaped data.

$$\frac{x - mean}{std}$$

	sp500_return	mean-std
2013-09-11	0.007346	0.844070
2013-09-12	0.003052	0.319621
2013-09-13	-0.003380	-0.466133
2013-09-17	0.008423	0.975693
2013-09-18	0.004218	0.461980
...	...	...
2017-08-29	0.002161	0.210721
2017-08-30	0.000843	0.049734
2017-08-31	0.004615	0.510524
2017-09-01	0.005721	0.645611
2017-09-04	0.000000	-0.053213





Rolling and Cumulative Methods

Rolling Method

The rolling() function is used to provide rolling window calculations. In financial times, it can be applied to compute all kinds of dynamic statistic, e.g. moving average,.

	Date	Close*	Rolling Close Average
10	Feb 01, 2019	62.44	NaN
9	Feb 04, 2019	62.58	62.510
8	Feb 05, 2019	64.27	63.425
7	Feb 06, 2019	63.44	63.855
6	Feb 07, 2019	61.50	62.470

Compute Rolling Mean

```
stock=pd.DataFrame(np.random.randn(20),index=dates,columns=["daily return"])
stock.head()
```

	daily return
2021-01-05	-0.229811
2021-01-06	-1.604339
2021-01-07	-0.646546
2021-01-08	-1.136787
2021-01-09	0.597907

```
stock["RollMean5"]=stock["daily return"].rolling(5).mean()
stock.head(8)
```

	daily return	RollMean5
2021-01-05	0.569065	NaN
2021-01-06	-0.034536	NaN
2021-01-07	0.236928	NaN
2021-01-08	-0.765561	NaN
2021-01-09	0.806891	0.162558
2021-01-10	-0.368254	-0.024906
2021-01-11	0.591866	0.100374
2021-01-12	-0.303040	-0.007619

Compute Rolling Sum

```
stock=pd.DataFrame(np.random.randn(20),index=dates,columns=["daily return"])
stock.head()
```

	daily return
2021-01-05	-0.229811
2021-01-06	-1.604339
2021-01-07	-0.646546
2021-01-08	-1.136787
2021-01-09	0.597907

```
stock["RollSum5"]=stock["daily return"].rolling(5).sum()
stock.head(8)
```

	daily return	RollSum5
2021-01-05	-0.229811	NaN
2021-01-06	-1.604339	NaN
2021-01-07	-0.646546	NaN
2021-01-08	-1.136787	NaN
2021-01-09	0.597907	-3.019576
2021-01-10	-0.714235	-3.504001
2021-01-11	0.832169	-1.067493
2021-01-12	1.797286	1.376339

Compute Cumulative Sum

```
stock=pd.DataFrame(np.random.randn(20),index=dates,columns=["daily return"])
stock.head()
```

	daily return
2021-01-05	-0.229811
2021-01-06	-1.604339
2021-01-07	-0.646546
2021-01-08	-1.136787
2021-01-09	0.597907

Cumulative sum is different which compute the sum starting from the first row.

```
stock["CumSum"]=stock["daily return"].cumsum()
stock.head(8)
```

	daily return	RollSum5	CumSum
2021-01-05	-0.229811	NaN	-0.229811
2021-01-06	-1.604339	NaN	-1.834150
2021-01-07	-0.646546	NaN	-2.480696
2021-01-08	-1.136787	NaN	-3.617483
2021-01-09	0.597907	-3.019576	-3.019576
2021-01-10	-0.714235	-3.504001	-3.733811
2021-01-11	0.832169	-1.067493	-2.901642
2021-01-12	1.797286	1.376339	-1.104357

Practice

Given apple stock price (5 seconds tick data), Compute rolling(30) average of stock price (close price in every 5 seconds) as the fast signal of price movement and rolling(80) average of stock price as the slow signal of price movement

Plot stock price, slow signal and fast signal in the same plot. Discuss how to trade stock using these two signals?

date	open	high	low	close	volume	average	barCount
2022-02-04 09:30:00	171.73	171.73	171.27	171.51	8932.35	171.519	931
2022-02-04 09:30:05	171.51	171.64	171.28	171.48	701.57	171.486	358
2022-02-04 09:30:10	171.46	171.68	171.45	171.62	548.99	171.602	204
2022-02-04 09:30:15	171.64	171.65	171.26	171.26	692.40	171.492	319
2022-02-04 09:30:20	171.34	171.59	171.33	171.47	384.28	171.479	208