# ISOM 2600 – Introduction to Business Analytics

## Topic 2: Exploratory Data Analysis
## Readings: Chapter 3

# About this Course

1

## Python Basic

- List
- NumPy
- SciPy
- Plotting

2

## Exploratory Data Analysis (EDA)

- Pandas
- EDA
- Methods for Time Series

3

## Multiple Linear Regression

4

## Clustering

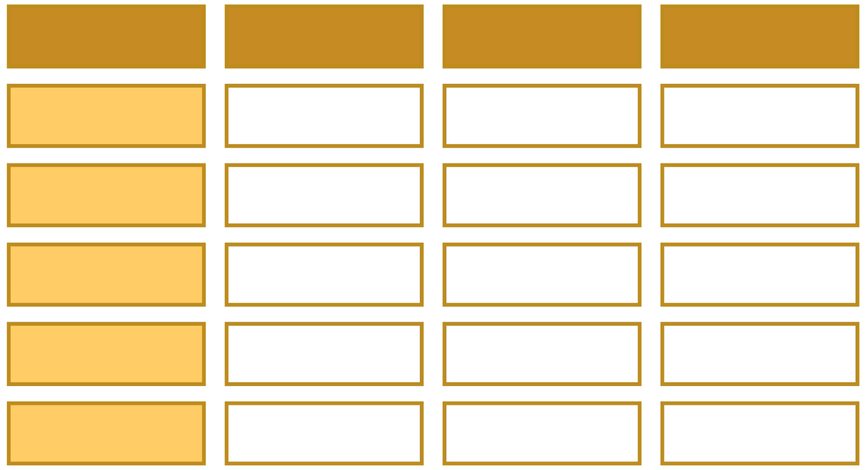
# Pandas

Pandas is a powerful open-source **library** built on top of Python.

It provides easy-to-use **data structures (DataFrame and Series)**, where **rows and columns are identified using labels** rather than just integer indices

Pandas handles **spreadsheet-like data** for **fast data loading, manipulation, alignment, and merging**, among many other functions

```
import pandas as pd
```



80% of data analysis is **EDA**

# Basic EDA

- ❑ Explore and summarize the data
- ❑ Data Visualization
- ❑ Data Standardization/Data Rescaling
- ❑ Handle Missing Data

# Agenda

**Series and DataFrame**

Methods: df.head(), df.tail()
Attributes: df.shape, df.index, df.columns
Series is a column of a DataFrame

**EDA: Visualization**

plot.hist()
plot.box()
plot.line()

**Make Changes to Data**

Add a new column
Update existing column
Drop columns

**EDA: Handle Missing Data**

isnull()
fillna()
Forward imputation
Backward imputation
dropna()

**EDA: Data Summarization**

describe()
mean(), median(), mode()
std(), var() (ddof = 0 or 1)
quantile()
Axes of Dataframe
value_counts()
sort_values()

**Data Slicing**

Column selection
Row selection:
loc() right inclusive
iloc() right exclusive
Mixed selection
Split Data into training and testing set

**EDA: Standardization**

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

**Methods for Time Series**

rolling().mean()
cumsum()

# Data: PJM

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.

**Data index** is the date/hour, **columns** are for different regions within PJM.

```
df = isom2600.data.energy()
df.head()
```

| Datetime | AEP | DAYTON | PJME | PJMW |
|---|---|---|---|---|
| 2004-10-01 01:00:00 | 12379.0 | 1621.0 | 24025.0 | 4628.0 |
| 2004-10-01 02:00:00 | 11935.0 | 1536.0 | 22845.0 | 4520.0 |
| 2004-10-01 03:00:00 | 11692.0 | 1500.0 | 22138.0 | 4431.0 |
| 2004-10-01 04:00:00 | 11597.0 | 1434.0 | 21922.0 | 4383.0 |
| 2004-10-01 05:00:00 | 11681.0 | 1489.0 | 22193.0 | 4460.0 |

**Pandas Data Structure: DataFrame**

# Pandas: DataFrame and Series

| Data Structure | Dimensionality | Format | View |
|---|---|---|---|
| **Series** | 1D | Column | |
| **DataFrame** | 2D | Single Sheet | |
| **Panel** | 3D | Multiple Sheets | |

Extra

**Series**

| | name |
|---|---|
| 0 | Rukshan |
| 1 | Prasadi |
| 2 | Gihan |
| 3 | Hansana |

| | age |
|---|---|
| 0 | 25 |
| 1 | 25 |
| 2 | 26 |
| 3 | 24 |

| | marks |
|---|---|
| 0 | 85 |
| 1 | 90 |
| 2 | 70 |
| 3 | 80 |

**DataFrame**

| | name | age | marks |
|---|---|---|---|
| 0 | Rukshan | 25 | 85 |
| 1 | Prasadi | 25 | 90 |
| 2 | Gihan | 26 | 70 |
| 3 | Hansana | 24 | 80 |

**Panel**

| | name | age | marks |
|---|---|---|---|
| 0 | Rukshan | 25 | 85 |
| 1 | Prasadi | 25 | 90 |
| 2 | Gihan | 26 | 70 |
| 3 | Hansana | 24 | 80 |

# DataFrame

DataFrame is a 2-dimensional data structure which has the **index** and **columns**

| Datetime | AEP | DAYTON | PJME | PJMW |
|---|---|---|---|---|
| 2004-10-01 01:00:00 | 12379.0 | 1621.0 | 24025.0 | 4628.0 |
| 2004-10-01 02:00:00 | 11935.0 | 1536.0 | 22845.0 | 4520.0 |
| 2004-10-01 03:00:00 | 11692.0 | 1500.0 | 22138.0 | 4431.0 |
| 2004-10-01 04:00:00 | 11597.0 | 1434.0 | 21922.0 | 4383.0 |
| 2004-10-01 05:00:00 | 11681.0 | 1489.0 | 22193.0 | 4460.0 |
| ... | ... | ... | ... | ... |
| 2018-08-02 20:00:00 | 17673.0 | 2554.0 | 44057.0 | 6545.0 |
| 2018-08-02 21:00:00 | 17303.0 | 2481.0 | 43256.0 | 6496.0 |
| 2018-08-02 22:00:00 | 17001.0 | 2405.0 | 41552.0 | 6325.0 |
| 2018-08-02 23:00:00 | 15964.0 | 2250.0 | 38500.0 | 5892.0 |
| 2018-08-03 00:00:00 | 14809.0 | 2042.0 | 35486.0 | 5489.0 |

121273 rows × 4 columns

**Row Index**
**Axis 0**

```
df.index
```

**Column**
**Axis 1**

```
df.columns
```

**Shape of the DataFrame**

```
df.shape
```

```
(121273, 4)
```

# Series

A Series in Pandas is **a one-dimensional** labeled array that can hold any data type, such as integers, floats, strings, or even objects

It is similar to a NumPy array, but with the added advantage of having **index labels** for each element, the default index is the counting number

***Note:*** **Series has no column attribute,** only values and index (row labels)

```
aep = df["PJME"] # Select one column
aep
```

| | PJME |
|---|---|
| Datetime | |
| 2004-10-01 01:00:00 | 24025.0 |
| 2004-10-01 02:00:00 | 22845.0 |
| 2004-10-01 03:00:00 | 22138.0 |
| 2004-10-01 04:00:00 | 21922.0 |
| 2004-10-01 05:00:00 | 22193.0 |
| ... | ... |
| 2018-08-02 20:00:00 | 44057.0 |
| 2018-08-02 21:00:00 | 43256.0 |
| 2018-08-02 22:00:00 | 41552.0 |
| 2018-08-02 23:00:00 | 38500.0 |
| 2018-08-03 00:00:00 | 35486.0 |

121273 rows × 1 columns

```
type(aep)
```

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

```
len(aep)
```

```
121273
```

# Methods and Attributes

Pandas library provides a wide range of functionalities, both Series and DataFrame objects have their own set of **methods** and **attributes** that make data manipulation easy

- ❑ **Attributes** are characteristics of the data type
- ❑ **Methods** define the operation and actions

Auto-Completion in Google Colab:

- ❑ Typing a dot (.) after a Pandas object
- ❑ Pressing Ctrl + Space to see the list of available methods

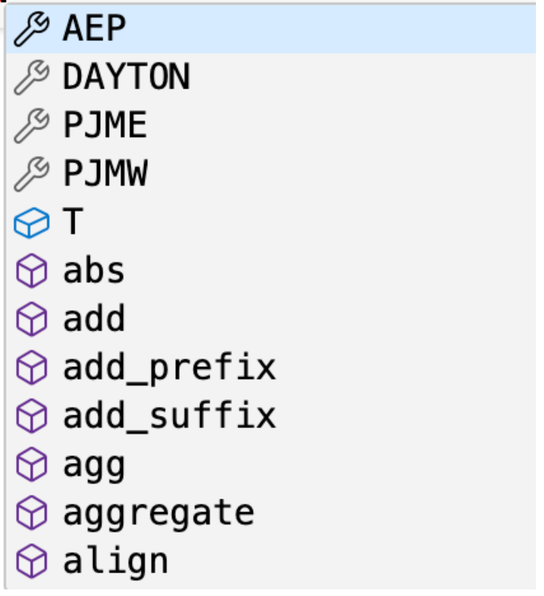
**Attribute**

```
df.index
```

**Method**

```
df.min()
```

```
df.
```



# Example: Head or Tail Methods

Return top n rows of a DataFrame or Series by default $n = 5$

```
df.head()
```

| Datetime | AEP | DAYTON | PJME | PJMW |
|---|---|---|---|---|
| 2004-10-01 01:00:00 | 12379.0 | 1621.0 | 24025.0 | 4628.0 |
| 2004-10-01 02:00:00 | 11935.0 | 1536.0 | 22845.0 | 4520.0 |
| 2004-10-01 03:00:00 | 11692.0 | 1500.0 | 22138.0 | 4431.0 |
| 2004-10-01 04:00:00 | 11597.0 | 1434.0 | 21922.0 | 4383.0 |
| 2004-10-01 05:00:00 | 11681.0 | 1489.0 | 22193.0 | 4460.0 |

Return bottom n rows of a DataFrame or Series, $n = 3$ specified

```
df.tail(3)
```

| Datetime | AEP | DAYTON | PJME | PJMW |
|---|---|---|---|---|
| 2018-08-02 22:00:00 | 17001.0 | 2405.0 | 41552.0 | 6325.0 |
| 2018-08-02 23:00:00 | 15964.0 | 2250.0 | 38500.0 | 5892.0 |
| 2018-08-03 00:00:00 | 14809.0 | 2042.0 | 35486.0 | 5489.0 |

**View Data:** Use the head() or tail() methods to display the first or bottom few rows. It helps verify that the data has loaded correctly and provides an overview of its structure (index, columns, values, etc.)

# Extra: Get Help on a Specific Method

If you need detailed documentation on a method, use: help()

Or use the ? operator in Jupyter/Colab to get the side bar

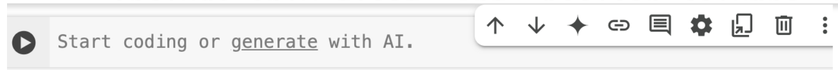
```
help(pd.Series.mean)
```

```
    Parameters
    ----------
    axis : {index (0)}
        Axis for the function to be applied on.
        For `Series` this parameter is unused and defaults to 0.

        For DataFrames, specifying ``axis=None`` will apply the aggregation
        across both axes.

        .. versionadded:: 2.0.0

    skipna : bool, default True
        Exclude NA/null values when computing the result.
    numeric_only : bool, default False
```




```
[16] pd.Series.mean?
```

Help ×

```
Signature: pd.Series.mean(self, axis:
'Axis | None' = 0, skipna: 'bool' =
True, numeric_only: 'bool' = False,
**kwargs)
Docstring:
Return the mean of the values over the
requested axis.

Parameters
----------
axis : {index (0)}
    Axis for the function to be applied
on.
    For `Series` this parameter is
unused and defaults to 0.

    For DataFrames, specifying
``axis=None`` will apply the
aggregation
    across both axes.

    .. versionadded:: 2.0.0
```

# EDA: Data Summarization

The first step of Exploratory Data Analysis (EDA) is to develop a strong understanding of the dataset's structure and the meaning behind each feature. This helps guide cleaning, transformation, and modeling decisions.

- ❑ **Column Overview:** identify what each column represents, check the data types (numerical or categorical)
- ❑ **Explore the distribution of each column/variable/feature:**

  **Position/Center:** mean, median and mode

  **Variation:** range, quantile, interquartile range(IQR) , variance and standard deviation

- ❑ **Skewness and Outliers:** boxplots, histogram **(in EDA: Data Visualization)**

# Example: Summary Statistics

DataFrame with **all numerical data**

```
df.describe()
```

| | AEP | DAYTON | PJME | PJMW |
|---|---|---|---|---|
| **count** | 121273.000000 | 121273.000000 | 121273.000000 | 121273.000000 |
| **mean** | 15499.513717 | 2037.853784 | 32097.205470 | 5578.201059 |
| **std** | 2591.399065 | 393.405597 | 6487.353855 | 991.168471 |
| **min** | 9581.000000 | 982.000000 | 14544.000000 | 2553.000000 |
| **25%** | 13630.000000 | 1749.000000 | 27612.000000 | 4875.000000 |
| **50%** | 15310.000000 | 2009.000000 | 31401.000000 | 5490.000000 |
| **75%** | 17200.000000 | 2279.000000 | 35647.000000 | 6229.000000 |
| **max** | 25695.000000 | 3746.000000 | 62009.000000 | 9594.000000 |

# Example: Summary Statistics

We can also call **statistical methods** for each column (a Series)

- **Mean/Median/Mode**

```
df['AEP'].mean()
```

```
np.float64(15499.513716985644)
```

```
df['AEP'].median()
```

```
15310.0
```

```
df['AEP'].mode()
```

| | AEP |
|---|---|
| 0 | 14941.0 |
| 1 | 15399.0 |

- **Range = max – min**

```
range = aSeries.max() - aSeries.min()
print(range)
```

```
16114.0
```

- **Quantile**

- **IQR = Q3 – Q1**

```
Q1 = aSeries.quantile(0.25)
Q3 = aSeries.quantile(0.75)
IQR = aSeries.quantile(0.75) - aSeries.quantile(0.25)
print(IQR)
```

```
3570.0
```

- **Variance and standard deviation**

```
aSeries.var(ddof = 1)
```

```
6715349.116196988
```

```
aSeries.std(ddof = 1)
```

```
2591.39906540791
```

# Axes of DataFrame

DataFrame's axes refer to the **directions along which operations** are performed, axis = 0 moves down the rows (for each column), axis = 1 moves across the columns (for each row)

```
df.mean(axis = 1)
# mean for each row, sometimes it makes no sense
```

| | 0 |
|---|---|
| **Datetime** | |
| **2004-10-01 01:00:00** | 10663.25 |
| **2004-10-01 02:00:00** | 10209.00 |
| **2004-10-01 03:00:00** | 9940.25 |
| **2004-10-01 04:00:00** | 9834.00 |
| **2004-10-01 05:00:00** | 9955.75 |
| **...** | ... |

```
df.mean(axis = 0)
```

| | 0 |
|---|---|
| **AEP** | 15499.513717 |
| **DAYTON** | 2037.853784 |
| **PJME** | 32097.205470 |
| **PJMW** | 5578.201059 |

```
df.mean()
# by default axis=0, for each column
```

| | 0 |
|---|---|
| **AEP** | 15499.513717 |
| **DAYTON** | 2037.853784 |
| **PJME** | 32097.205470 |
| **PJMW** | 5578.201059 |

**Note: When we do EDA, we do column-wise calculation (axis = 0, for each column) most of the time**

# Case: Titanic Data

```
titanic = isom2600.data.titanic()
titanic.head()
```

| | name | gender | age | class | embarked | country | ticketno | fare | survived |
|---|---|---|---|---|---|---|---|---|---|
| **0** | Abbing, Mr. Anthony | male | 42.0 | 3rd | S | United States | 5547.0 | 7.11 | 0 |
| **1** | Abbott, Mr. Eugene Joseph | male | 13.0 | 3rd | S | United States | 2673.0 | 20.05 | 0 |
| **2** | Abbott, Mr. Rossmore Edward | male | 16.0 | 3rd | S | United States | 2673.0 | 20.05 | 0 |
| **3** | Abbott, Mrs. Rhoda Mary 'Rosa' | female | 39.0 | 3rd | S | England | 2673.0 | 20.05 | 1 |
| **4** | Abelseth, Miss. Karen Marie | female | 16.0 | 3rd | S | Norway | 348125.0 | 7.13 | 1 |

https://www.kaggle.com/datasets/yasserh/titanic-dataset

# Example: Summary Statistics

DataFrame with both numerical and **categorical data**

```
titanic = isom2600.data.titanic()
titanic.describe(include = 'all')
```

| | name | gender | age | class | embarked | country | ticketno | fare | survived |
|---|---|---|---|---|---|---|---|---|---|
| **count** | 2207 | 2207 | 2205.000000 | 2207 | 2207 | 2126 | 1.316000e+03 | 1291.000000 | 2207.000000 |
| **unique** | 2202 | 2 | NaN | 7 | 4 | 48 | NaN | NaN | NaN |
| **top** | Kelly, Mr. James | male | NaN | 3rd | S | England | NaN | NaN | NaN |
| **freq** | 3 | 1718 | NaN | 709 | 1616 | 1125 | NaN | NaN | NaN |
| **mean** | NaN | NaN | 30.436735 | NaN | NaN | NaN | 2.842157e+05 | 33.404760 | 0.322157 |
| **std** | NaN | NaN | 12.159677 | NaN | NaN | NaN | 6.334726e+05 | 52.227592 | 0.467409 |
| **min** | NaN | NaN | 0.166667 | NaN | NaN | NaN | 2.000000e+00 | 3.030500 | 0.000000 |
| **25%** | NaN | NaN | 22.000000 | NaN | NaN | NaN | 1.426225e+04 | 7.180600 | 0.000000 |
| **50%** | NaN | NaN | 29.000000 | NaN | NaN | NaN | 1.114265e+05 | 14.090200 | 0.000000 |
| **75%** | NaN | NaN | 38.000000 | NaN | NaN | NaN | 3.470770e+05 | 31.060750 | 1.000000 |
| **max** | NaN | NaN | 74.000000 | NaN | NaN | NaN | 3.101317e+06 | 512.060700 | 1.000000 |

# Case: Titanic Data

**Question 1: Identify the top 5 countries where the majority of passengers originate**

```
titanic["country"].value_counts(dropna=True).head(5)
# by default, count the freq and sort in a descending order
```

| country | count |
|---|---|
| England | 1125 |
| United States | 264 |
| Ireland | 137 |
| Sweden | 105 |
| Lebanon | 71 |

value_counts() counts the frequency of unique values in a column and sort them in descending order by default (we drop the missing values), head(5) selects the top 5 countries

# Case: Titanic Data

**Question 2: Count the number of female passengers and the number of passengers who survived**

```
titanic['gender'].value_counts()
```

| gender | count |
|---|---|
| male | 1718 |
| female | 489 |

```
titanic["survived"].value_counts()
```

| survived | count |
|---|---|
| 0 | 1496 |
| 1 | 711 |

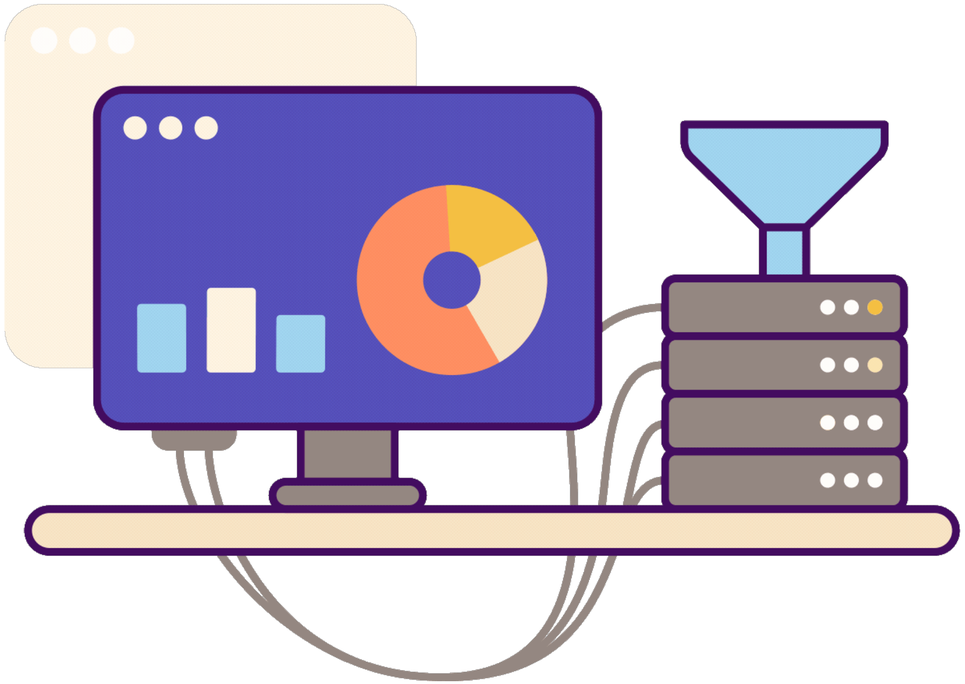
# Case: Titanic Data

## Question 3: find the oldest passenger and his/her information

```
titanic.sort_values(by = "age", ascending=False)
```

| | name | gender | age | class | embarked | country | ticketno | fare | survived |
|---|---|---|---|---|---|---|---|---|---|
| **1176** | Svensson, Mr. Johan | male | 74.000000 | 3rd | S | Sweden | 347060.0 | 7.1506 | 0 |
| **820** | Mitchell, Mr. Henry Michael | male | 72.000000 | 2nd | S | England | 24580.0 | 10.1000 | 0 |
| **53** | Artagaveytia, Mr. Ramon | male | 71.000000 | 1st | C | Argentina | 17609.0 | 49.1001 | 0 |
| **456** | Goldschmidt, Mr. George B. | male | 71.000000 | 1st | C | United States | 17754.0 | 34.1301 | 0 |
| **282** | Crosby, Captain. Edward Gifford | male | 70.000000 | 1st | S | United States | 5735.0 | 71.0000 | 0 |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| **1182** | Tannūs, Master. As'ad | male | 0.416667 | 3rd | C | Lebanon | 2625.0 | 8.1004 | 1 |
| **296** | Danbom, Master. Gilbert Sigvard Emanuel | male | 0.333333 | 3rd | S | Sweden | 347080.0 | 14.0800 | 0 |
| **316** | Dean, Miss. Elizabeth Gladys 'Millvina' | female | 0.166667 | 3rd | S | England | 2315.0 | 20.1106 | 1 |
| **439** | Gheorgheff, Mr. Stanio | male | NaN | 3rd | C | Bulgaria | 349254.0 | 7.1711 | 0 |
| **677** | Kraeff, Mr. Theodor | male | NaN | 3rd | C | Bulgaria | 349253.0 | 7.1711 | 0 |

# EDA: Data Visualization



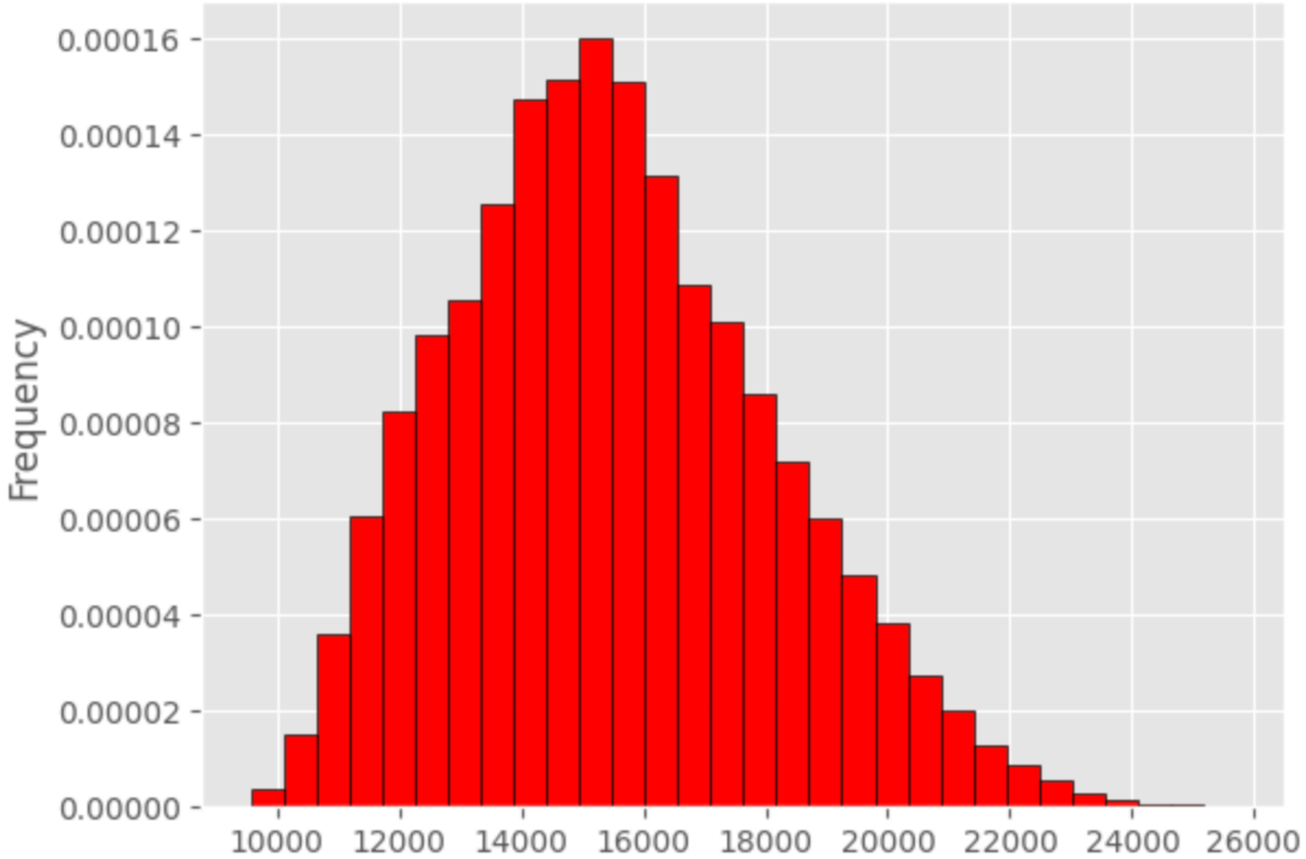
- ❑ Histogram
- ❑ Boxplot
- ❑ Line Plot

# Histogram

**Pandas has built-in plotting capabilities that make it super easy to create quick visualizations (compared with library: matplotlib in Topic 1)**

```
aSeries.plot.hist(bins=30, density=True, edgecolor="black", color="red")
# built-in plot methods to draw graph for the data
```

```
<Axes: ylabel='Frequency'>
```




Call plot.hist() method of Series, it uses matplotlib under the hood

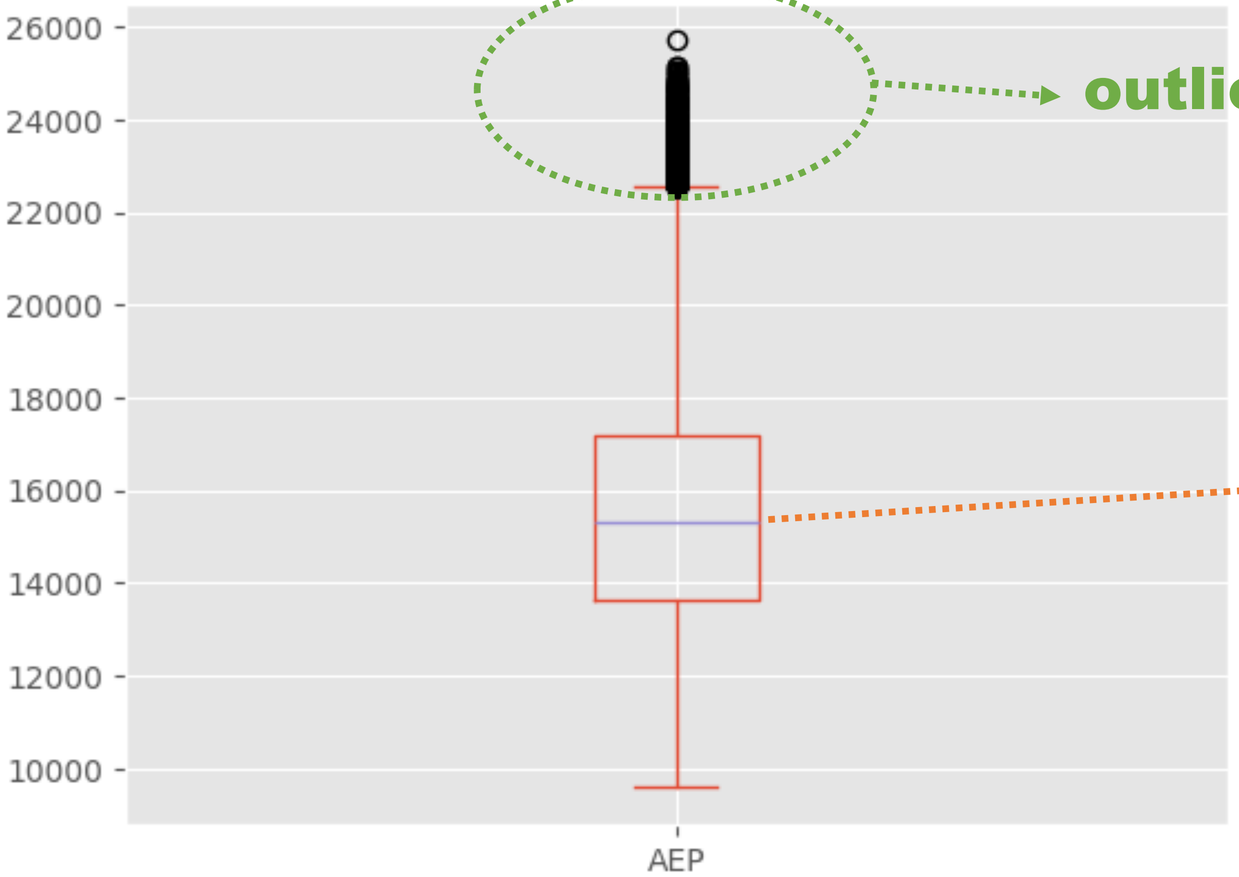
**in matplotlib**

```
plt.hist(aSeries, bins=30, edgecolor="black", color="red")
plt.show() # topic 1
```

# Boxplot

**A boxplot summarizes the distribution of a dataset—especially its spread, center, and potential outliers**

```
aSeries.plot.box()
```




| | AEP |
|---|---|
| count | 121273.000000 |
| mean | 15499.513717 |
| std | 2591.399065 |
| min | 9581.000000 |
| 25% | 13630.000000 |
| 50% | 15310.000000 |
| 75% | 17200.000000 |
| max | 25695.000000 |

# Line Chart

**A line chart is ideal for showing patterns, trends, and changes over time—especially in time series data like sales, temperature, stock prices, etc.**

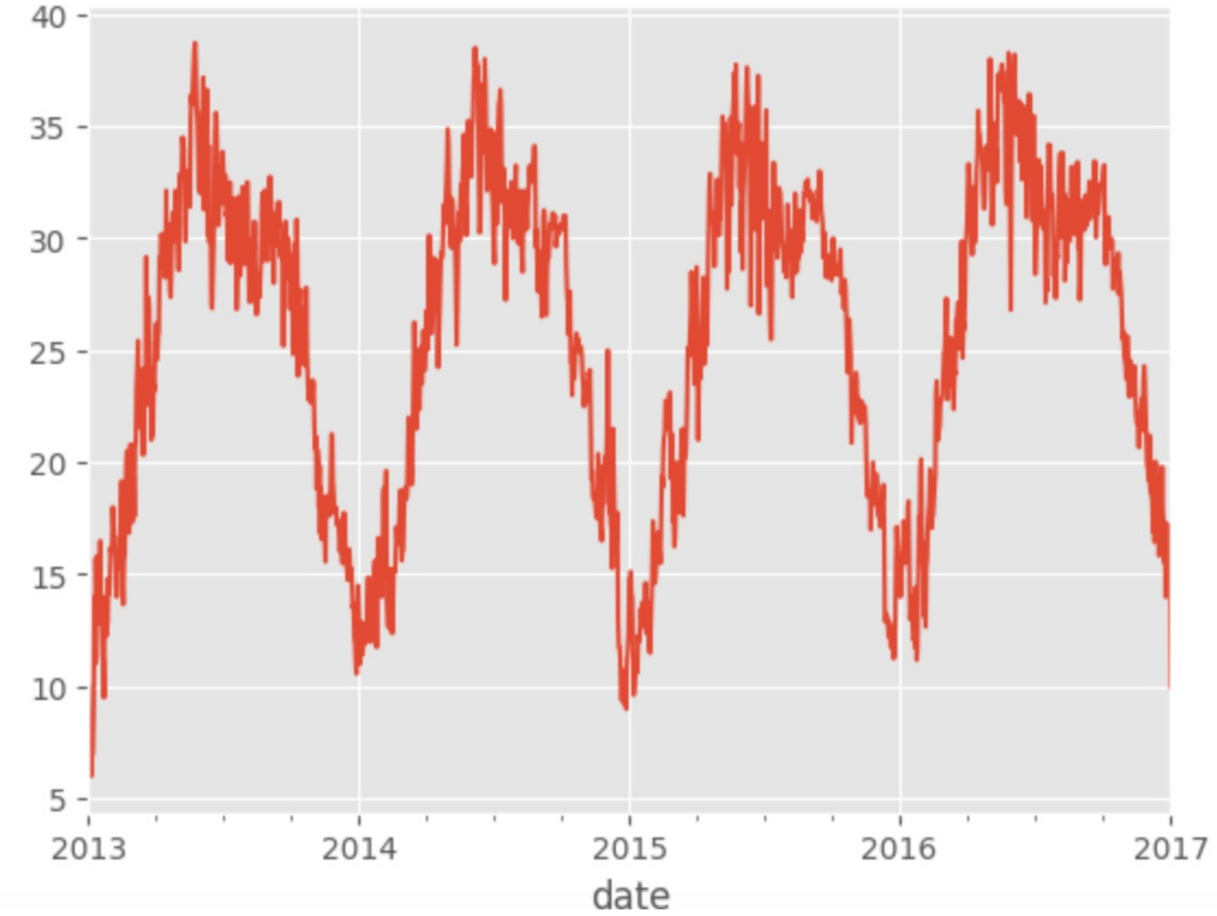
```
weather = isom2600.data.weather()
temp = weather["meantemp"]
```

```
weather.head()
```

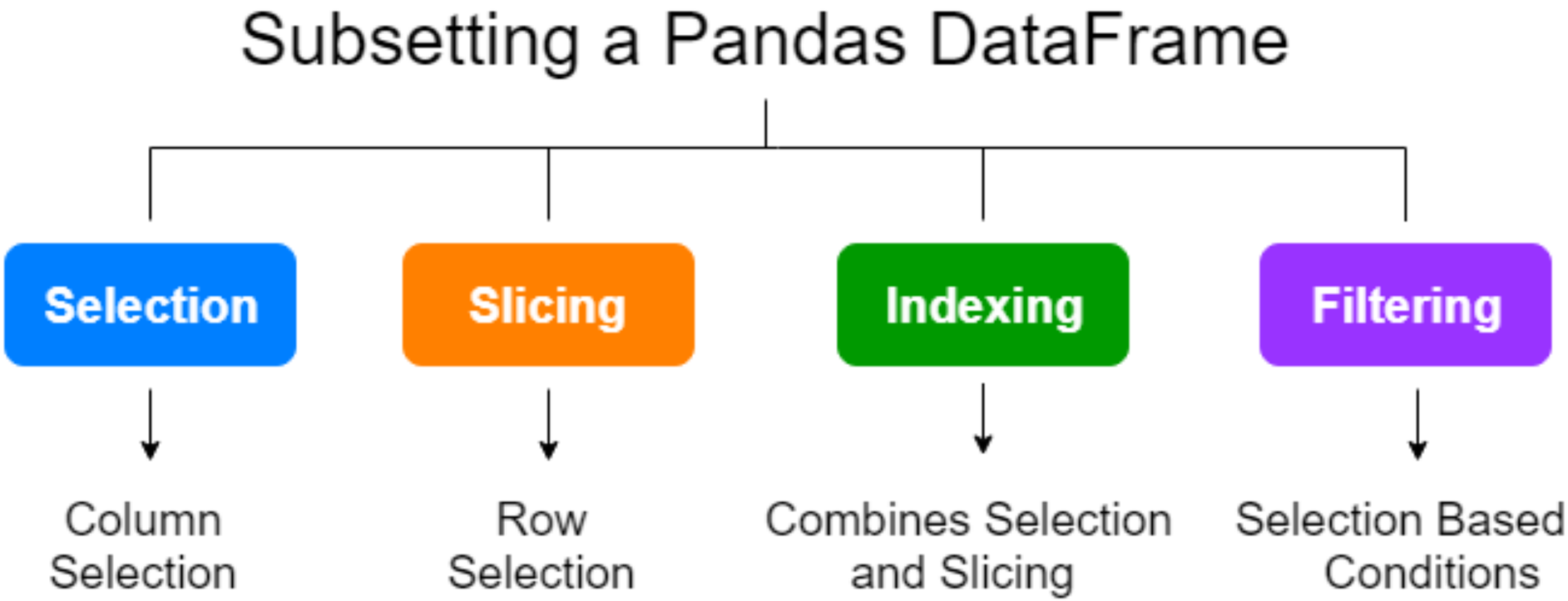
| date | meantemp | humidity | wind_speed | meanpressure |
|---|---|---|---|---|
| **2013-01-01** | 10.000000 | 84.500000 | 0.000000 | 1015.666667 |
| **2013-01-02** | 7.400000 | 92.000000 | 2.980000 | 1017.800000 |
| **2013-01-03** | 7.166667 | 87.000000 | 4.633333 | 1018.666667 |
| **2013-01-04** | 8.666667 | 71.333333 | 1.233333 | 1017.166667 |
| **2013-01-05** | 6.000000 | 86.833333 | 3.700000 | 1016.500000 |

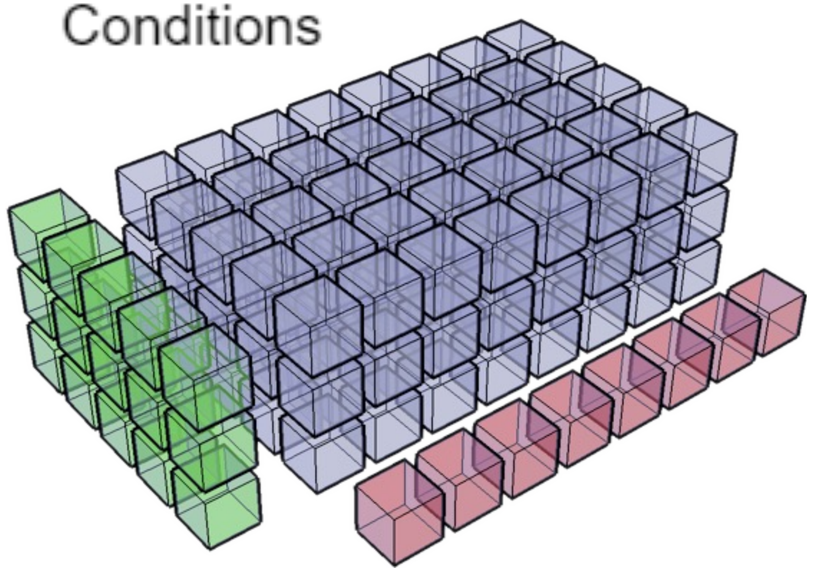
```
temp.plot.line()
```

```
<Axes: xlabel='date'>
```



# Data Slicing



# Column Selection

## Select One Column

```
df['AEP']
```

| Datetime | AEP |
|---|---|
| 2004-10-01 01:00:00 | 12379.0 |
| 2004-10-01 02:00:00 | 11935.0 |
| 2004-10-01 03:00:00 | 11692.0 |
| 2004-10-01 04:00:00 | 11597.0 |
| 2004-10-01 05:00:00 | 11681.0 |
| ... | ... |

## Select Multiple Columns

```
df[['AEP', 'DAYTON', 'PJME']]
```

| Datetime | AEP | DAYTON | PJME |
|---|---|---|---|
| 2004-10-01 01:00:00 | 12379.0 | 1621.0 | 24025.0 |
| 2004-10-01 02:00:00 | 11935.0 | 1536.0 | 22845.0 |
| 2004-10-01 03:00:00 | 11692.0 | 1500.0 | 22138.0 |
| 2004-10-01 04:00:00 | 11597.0 | 1434.0 | 21922.0 |
| 2004-10-01 05:00:00 | 11681.0 | 1489.0 | 22193.0 |
| ... | ... | ... | ... |

# Row Selection: loc()

**loc() method selects rows based on row names/labels/index, not positions**

## Index

```
df.loc[["2004-10-01 05:00:00", "2004-10-01 06:00:00"]]
```

| Datetime | AEP | DAYTON | PJME | PJMW |
|---|---|---|---|---|
| 2004-10-01 05:00:00 | 11681.0 | 1489.0 | 22193.0 | 4460.0 |
| 2004-10-01 06:00:00 | 12280.0 | 1620.0 | 23908.0 | 4768.0 |

## Slicing

```
df.loc["2004-10-01 05:00:00":"2004-10-01 07:00:00"]
```

| Datetime | AEP | DAYTON | PJME | PJMW |
|---|---|---|---|---|
| 2004-10-01 05:00:00 | 11681.0 | 1489.0 | 22193.0 | 4460.0 |
| 2004-10-01 06:00:00 | 12280.0 | 1620.0 | 23908.0 | 4768.0 |
| 2004-10-01 07:00:00 | 13692.0 | 1859.0 | 27487.0 | 5360.0 |

# Row Selection: iloc()

**iloc() method selects row numbers based on integer position**

```
df.iloc[0]
```

| | 2004-10-01 01:00:00 |
|---|---|
| AEP | 12379.0 |
| DAYTON | 1621.0 |
| PJME | 24025.0 |
| PJMW | 4628.0 |

Index

```
df.iloc[[0,1]]
```

| Datetime | AEP | DAYTON | PJME | PJMW |
|---|---|---|---|---|
| 2004-10-01 01:00:00 | 12379.0 | 1621.0 | 24025.0 | 4628.0 |
| 2004-10-01 02:00:00 | 11935.0 | 1536.0 | 22845.0 | 4520.0 |

```
df.iloc[-2:]# last two rows
```

Slicing

| Datetime | AEP | DAYTON | PJME | PJMW |
|---|---|---|---|---|
| 2018-08-02 23:00:00 | 15964.0 | 2250.0 | 38500.0 | 5892.0 |
| 2018-08-03 00:00:00 | 14809.0 | 2042.0 | 35486.0 | 5489.0 |

# Row Selection

**ATTN: loc method is right inclusive and iloc is right exclusive**

```
# right inclusive if loc
df.loc["2004-10-01 05:00:00" : "2004-10-01 07:00:00"]
```

| Datetime | AEP | DAYTON | PJME | PJMW |
|---|---|---|---|---|
| **2004-10-01 05:00:00** | 11681.0 | 1489.0 | 22193.0 | 4460.0 |
| **2004-10-01 06:00:00** | 12280.0 | 1620.0 | 23908.0 | 4768.0 |
| **2004-10-01 07:00:00** | 13692.0 | 1859.0 | 27487.0 | 5360.0 |

```
# right exclusive if iloc
df.iloc[0:2]
```

| Datetime | AEP | DAYTON | PJME | PJMW |
|---|---|---|---|---|
| **2004-10-01 01:00:00** | 12379.0 | 1621.0 | 24025.0 | 4628.0 |
| **2004-10-01 02:00:00** | 11935.0 | 1536.0 | 22845.0 | 4520.0 |

# In-class Exercise: TSLA Stock

**Question: get the TSLA stock price data and compare how it changed in 2024 and in 2026**

```
tsla = isom2600.data.getStock("TSLA")
tsla.tail()
```

| Price | Close | High | Low | Open | Volume |
|---|---|---|---|---|---|
| Ticker | TSLA | TSLA | TSLA | TSLA | TSLA |
| Date | | | | | |
| **2025-03-31** | 259.160004 | 260.559998 | 243.360001 | 249.309998 | 134008900 |
| **2025-04-01** | 268.459991 | 277.450012 | 259.250000 | 263.799988 | 146486900 |
| **2025-04-02** | 282.760010 | 284.989990 | 251.270004 | 254.600006 | 212787800 |
| **2025-04-03** | 267.279999 | 276.299988 | 261.510010 | 265.290009 | 136174300 |
| **2025-04-04** | 239.429993 | 261.000000 | 236.000000 | 255.380005 | 180324400 |

# Extra: Conditional Selection

**Titanic Data: count how many female passengers survived**

```
titanic.loc[titanic["gender"]=="female"]["survived"].value_counts()
```

| survived | count |
|---|---|
| 1 | 359 |
| 0 | 130 |

Conditional Selection: loc() **selects the rows** of female passengers by selecting rows satisfying condition: **titanic["gender"] == "female**
value_counts() counts how many female passengers survived (1 = survived) or not

# Mixed Selection

If you want to select both **rows** and **columns**: using **square brackets** with a **comma** inside to separate the row and column parts

```
df.loc["2008-12-31 01:00:00":"2008-12-31 03:00:00", ["AEP","PJMW"]]
```

| Datetime | AEP | PJMW |
|---|---|---|
| **2008-12-31 01:00:00** | 14548.0 | 5214.0 |
| **2008-12-31 02:00:00** | 14045.0 | 5044.0 |
| **2008-12-31 03:00:00** | 13851.0 | 4940.0 |

This code will select the rows between the timestamps '2008-12-31 01:00:00' and '2008-12-31 03:00:00', and the columns 'AEP' and 'PJMW' using label-based indexing with loc()

```
df.iloc[-2:,-2:]
```

| Datetime | PJME | PJMW |
|---|---|---|
| **2018-08-02 23:00:00** | 38500.0 | 5892.0 |
| **2018-08-03 00:00:00** | 35486.0 | 5489.0 |

This code selects the last 2 rows and the last 2 columns with iloc()

# Example: Split Data into Train and Test

**Time Series Data: first 80% is Training Data and latest 20% is Testing Data**

## Step 1

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

```
(97018, 24255)
```

## Step 2

```
traindata = df.iloc[0:trainsize, :]
testdata = df.iloc[trainsize:, :]
```

```
traindata.shape, testdata.shape
```

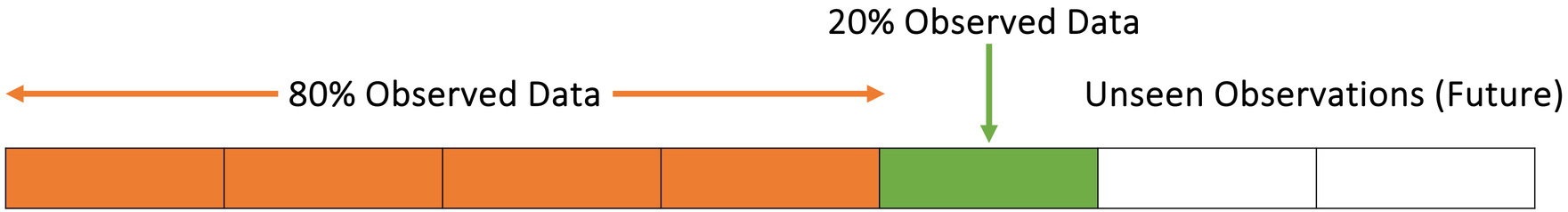
```
((97018, 4), (24255, 4))
```

```
traindata.tail(2)
```

| Datetime | AEP | DAYTON | PJME | PJMW |
|---|---|---|---|---|
| **2015-10-27 08:00:00** | 14867.0 | 2024.0 | 31707.0 | 5866.0 |
| **2015-10-27 09:00:00** | 14837.0 | 2029.0 | 31373.0 | 5773.0 |

```
testdata.head(2)
```

| Datetime | AEP | DAYTON | PJME | PJMW |
|---|---|---|---|---|
| **2015-10-27 10:00:00** | 15147.0 | 2026.0 | 30864.0 | 5677.0 |
| **2015-10-27 11:00:00** | 14796.0 | 2042.0 | 30510.0 | 5668.0 |



# Example: Split Data into Train and Test

**Cross-sectional Data : shuffle data first, then randomly select 80% be the Training Data and the rest 20% is Testing Data**

**Step 1**

```
from sklearn.utils import shuffle
profitdata = isom2600.data.profit() # cross-sectional data
shuffleddata = shuffle(profitdata)
shuffleddata.head()
```

**Step 2**

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

**Why shuffle the data first?**


# Making Changes to Data

# Add New Columns

## Add new columns to an existing DataFrame

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

| Datetime | AEP | DAYTON | PJME | PJMW | Month |
|---|---|---|---|---|---|
| 2018-08-02 20:00:00 | 17673.0 | 2554.0 | 44057.0 | 6545.0 | 8 |
| 2018-08-02 21:00:00 | 17303.0 | 2481.0 | 43256.0 | 6496.0 | 8 |
| 2018-08-02 22:00:00 | 17001.0 | 2405.0 | 41552.0 | 6325.0 | 8 |
| 2018-08-02 23:00:00 | 15964.0 | 2250.0 | 38500.0 | 5892.0 | 8 |
| 2018-08-03 00:00:00 | 14809.0 | 2042.0 | 35486.0 | 5489.0 | 8 |

Index is **Datetime** type, we can get the year, month, day or time
We add a new column: Month based on the index

```
type(df.index)
```

```
pandas.core.indexes.datetimes.DatetimeIndex
```

# Add New Columns

## Add new columns from algebraic operation on existing columns

```
df["new1"] = 3 * df["AEP"]
df["new2"] = df["AEP"] + 2
df["new3"] = df["AEP"] ** 2
df[["new1", "new2", "new3"]].tail()
```

| Datetime | new1 | new2 | new3 |
|---|---|---|---|
| 2018-08-02 20:00:00 | 53019.0 | 17675.0 | 312334929.0 |
| 2018-08-02 21:00:00 | 51909.0 | 17305.0 | 299393809.0 |
| 2018-08-02 22:00:00 | 51003.0 | 17003.0 | 289034001.0 |
| 2018-08-02 23:00:00 | 47892.0 | 15966.0 | 254849296.0 |
| 2018-08-03 00:00:00 | 44427.0 | 14811.0 | 219306481.0 |

```
df["ratio"] = df["AEP"] / df["PJME"]
df["AEP_pct_change"] = df["AEP"].pct_change()
df[["AEP", "PJME", "ratio", "AEP_pct_change"]].tail()
```

| Datetime | AEP | PJME | ratio | AEP_pct_change |
|---|---|---|---|---|
| 2018-08-02 20:00:00 | 17673.0 | 44057.0 | 0.401139 | -0.024561 |
| 2018-08-02 21:00:00 | 17303.0 | 43256.0 | 0.400014 | -0.020936 |
| 2018-08-02 22:00:00 | 17001.0 | 41552.0 | 0.409150 | -0.017454 |
| 2018-08-02 23:00:00 | 15964.0 | 38500.0 | 0.414649 | -0.060996 |
| 2018-08-03 00:00:00 | 14809.0 | 35486.0 | 0.417320 | -0.072350 |

# Update Columns

## Make a new column “Season” based on the “Month”

```
df = isom2600.data.energy()
df["Month"] = df.index.month
df["Season"] = [1 if t < 4
                else 2 if t < 7
                else 3 if t < 10
                else 4 for t in df["Month"]]
df.head(2)
```

| Datetime | AEP | DAYTON | PJME | PJMW | Month | Season |
|---|---|---|---|---|---|---|
| **2004-10-01 01:00:00** | 12379.0 | 1621.0 | 24025.0 | 4628.0 | 10 | 4 |
| **2004-10-01 02:00:00** | 11935.0 | 1536.0 | 22845.0 | 4520.0 | 10 | 4 |

**Make a new column “Season” (integer)**

```
df["Season"] = ["Spring" if t == 1
                else "Summer" if t == 2
                else "Autumn" if t == 3
                else "Winter" for t in df["Season"]]
# update the column
df.head(2)
```

| Datetime | AEP | DAYTON | PJME | PJMW | Month | Season |
|---|---|---|---|---|---|---|
| **2004-10-01 01:00:00** | 12379.0 | 1621.0 | 24025.0 | 4628.0 | 10 | Winter |
| **2004-10-01 02:00:00** | 11935.0 | 1536.0 | 22845.0 | 4520.0 | 10 | Winter |

**Update the new column directly (from integer to text)**

# Drop Column or Row

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

| Datetime | AEP | DAYTON | PJME | PJMW | Month | Season |
|---|---|---|---|---|---|---|
| 2004-10-01 01:00:00 | 12379.0 | 1621.0 | 24025.0 | 4628.0 | 10 | Winter |
| 2004-10-01 02:00:00 | 11935.0 | 1536.0 | 22845.0 | 4520.0 | 10 | Winter |
| 2004-10-01 03:00:00 | 11692.0 | 1500.0 | 22138.0 | 4431.0 | 10 | Winter |

```
df_clean = df.drop(["Month","Season"], axis=1)
df_clean.head(3)
```

| Datetime | AEP | DAYTON | PJME | PJMW |
|---|---|---|---|---|
| 2004-10-01 01:00:00 | 12379.0 | 1621.0 | 24025.0 | 4628.0 |
| 2004-10-01 02:00:00 | 11935.0 | 1536.0 | 22845.0 | 4520.0 |
| 2004-10-01 03:00:00 | 11692.0 | 1500.0 | 22138.0 | 4431.0 |

**Drop 2 Columns**

```
df.tail(5)
```

| Datetime | AEP | DAYTON | PJME | PJMW | Month | Season |
|---|---|---|---|---|---|---|
| 2018-08-02 20:00:00 | 17673.0 | 2554.0 | 44057.0 | 6545.0 | 8 | Autumn |
| 2018-08-02 21:00:00 | 17303.0 | 2481.0 | 43256.0 | 6496.0 | 8 | Autumn |
| 2018-08-02 22:00:00 | 17001.0 | 2405.0 | 41552.0 | 6325.0 | 8 | Autumn |
| 2018-08-02 23:00:00 | 15964.0 | 2250.0 | 38500.0 | 5892.0 | 8 | Autumn |
| 2018-08-03 00:00:00 | 14809.0 | 2042.0 | 35486.0 | 5489.0 | 8 | Autumn |

**Drop Last 3 Rows**

```
df_detail = df.drop(df.index[-3:], axis=0)
df_detail.tail(3)
```

| Datetime | AEP | DAYTON | PJME | PJMW | Month | Season |
|---|---|---|---|---|---|---|
| 2018-08-02 19:00:00 | 18118.0 | 2600.0 | 45641.0 | 6693.0 | 8 | Autumn |
| 2018-08-02 20:00:00 | 17673.0 | 2554.0 | 44057.0 | 6545.0 | 8 | Autumn |
| 2018-08-02 21:00:00 | 17303.0 | 2481.0 | 43256.0 | 6496.0 | 8 | Autumn |

# EDA: Standardization

**When you want to center your data around 0 and scale it to have unit variance 1, specially when you assume data is normal distributed**

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

```
sp500_mean = sp500["Close"].mean()
sp500_sd = sp500["Close"].std(ddof=1)
```

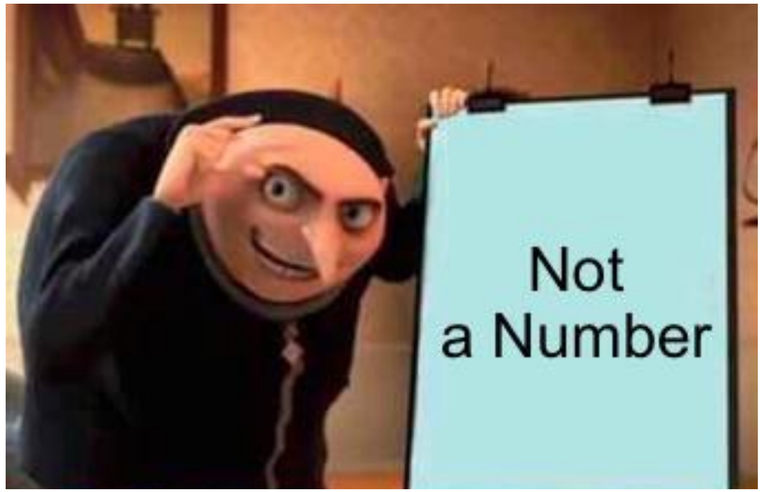
```
sp500["mean_sd"] = (sp500["Close"] - sp500_mean)/sp500_sd
sp500[["Close","mean_sd"]].head()
```

| Date | Close | mean_sd |
|---|---|---|
| 2019-12-31 | 3230.780029 | 0.040406 |
| 2020-01-02 | 3257.850098 | 0.125373 |
| 2020-01-03 | 3234.850098 | 0.053181 |
| 2020-01-06 | 3246.280029 | 0.089057 |
| 2020-01-07 | 3237.179932 | 0.060494 |

**Note: in topic 4: Clustering, we will standardize data first before modeling**

# EDA: Handle Missing Value

# What is NaN?




Pandas displays missing values as **NaN** (or *NaN, NAN, nan*)

**How Do Missing Values Occur?**

- ❑ Load a dataset that already contains missing values
- ❑ Merge datasets: a field exists in one dataset but not the other

| Datetime | AEP | COMED | DAYTON | DEOK | DOM | DUQ | EKPC | FE | NI | PJME | PJMW | PJM_Load |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **2018-08-02 20:00:00** | 17673.0 | 16437.0 | 2554.0 | 4052.0 | 14038.0 | 1966.0 | 1815.0 | 9866.0 | NaN | 44057.0 | 6545.0 | NaN |
| **2018-08-02 21:00:00** | 17303.0 | 15590.0 | 2481.0 | 3892.0 | 13832.0 | 1944.0 | 1769.0 | 9656.0 | NaN | 43256.0 | 6496.0 | NaN |
| **2018-08-02 22:00:00** | 17001.0 | 15086.0 | 2405.0 | 3851.0 | 13312.0 | 1901.0 | 1756.0 | 9532.0 | NaN | 41552.0 | 6325.0 | NaN |
| **2018-08-02 23:00:00** | 15964.0 | 14448.0 | 2250.0 | 3575.0 | 12390.0 | 1789.0 | 1619.0 | 8872.0 | NaN | 38500.0 | 5892.0 | NaN |
| **2018-08-03 00:00:00** | 14809.0 | 13335.0 | 2042.0 | 3281.0 | 11385.0 | 1656.0 | 1448.0 | 8198.0 | NaN | 35486.0 | 5489.0 | NaN |

# Count Missing Values

```
rawdata.isnull().tail(3)
```

| Datetime | AEP | COMED | DAYTON | DEOK | DOM | DUQ | EKPC | FE | NI | PJME | PJMW | PJM_Load |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 2018-08-02 22:00:00 | False | False | False | False | False | False | False | False | True | False | False | True |
| 2018-08-02 23:00:00 | False | False | False | False | False | False | False | False | True | False | False | True |
| 2018-08-03 00:00:00 | False | False | False | False | False | False | False | False | True | False | False | True |

```
rawdata.isnull().sum().sum()/(rawdata.shape[0]*rawdata.shape[1])
```

```
0.49037231715115953
```

Why sum().sum()?

**This is the missing rate**

# Impute NaN Values

**We can use fillna() method to recode the missing values to another value, for example: 0 or mean/mode/median of the column**

```
df = rawdata.fillna(0)
df.tail()
```

| Datetime | AEP | COMED | DAYTON | DEOK | DOM | DUQ | EKPC | FE | NI | PJME | PJMW | PJM_Load |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **2018-08-02 20:00:00** | 17673.0 | 16437.0 | 2554.0 | 4052.0 | 14038.0 | 1966.0 | 1815.0 | 9866.0 | 0.0 | 44057.0 | 6545.0 | 0.0 |
| **2018-08-02 21:00:00** | 17303.0 | 15590.0 | 2481.0 | 3892.0 | 13832.0 | 1944.0 | 1769.0 | 9656.0 | 0.0 | 43256.0 | 6496.0 | 0.0 |
| **2018-08-02 22:00:00** | 17001.0 | 15086.0 | 2405.0 | 3851.0 | 13312.0 | 1901.0 | 1756.0 | 9532.0 | 0.0 | 41552.0 | 6325.0 | 0.0 |
| **2018-08-02 23:00:00** | 15964.0 | 14448.0 | 2250.0 | 3575.0 | 12390.0 | 1789.0 | 1619.0 | 8872.0 | 0.0 | 38500.0 | 5892.0 | 0.0 |
| **2018-08-03 00:00:00** | 14809.0 | 13335.0 | 2042.0 | 3281.0 | 11385.0 | 1656.0 | 1448.0 | 8198.0 | 0.0 | 35486.0 | 5489.0 | 0.0 |

**It is not always appropriate to impute by “0”**

# Forward Imputation

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

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

Start from a “empty” DataFrame with all NaN values, and modify two rows by indexing

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

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

**This is suitable for time series data. Why?**

# Backward Imputation

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

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

Missing value should be taken from the next available valid entry, e.g. survey data

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

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

# Extra: Model-based Imputation

| | Salary | Experience | Age |
|---|---|---|---|
| 0 | 50000.0 | 2 | 25 |
| 1 | 60000.0 | 4 | 30 |
| 2 | NaN | 3 | 28 |
| 3 | 65000.0 | 5 | 35 |
| 4 | 70000.0 | 6 | 40 |
| 5 | NaN | 4 | 38 |

**Could you predict the missing values based on available information?**

# Drop NaN

```
newdf['C'] = 1
newdf
```

| Datetime | A | B | C |
|---|---|---|---|
| 1998-04-01 01:00:00 | NaN | NaN | 1 |
| 1998-04-01 02:00:00 | NaN | NaN | 1 |
| 1998-04-01 03:00:00 | NaN | NaN | 1 |
| 1998-04-01 04:00:00 | 3 | 3 | 1 |
| 1998-04-01 05:00:00 | 4 | 4 | 1 |
| 1998-04-01 06:00:00 | NaN | NaN | 1 |
| 1998-04-01 07:00:00 | NaN | NaN | 1 |
| 1998-04-01 08:00:00 | NaN | NaN | 1 |
| 1998-04-01 09:00:00 | NaN | NaN | 1 |
| 1998-04-01 10:00:00 | NaN | NaN | 1 |

## Drop Rows with NaN

```
newdf.dropna(axis=0)
```

| Datetime | A | B | C |
|---|---|---|---|
| 1998-04-01 04:00:00 | 3 | 3 | 1 |
| 1998-04-01 05:00:00 | 4 | 4 | 1 |

**How about newdf.dropna()?**

## Drop Columns with NaN

```
newdf.dropna(axis=1)
```

| Datetime | C |
|---|---|
| 1998-04-01 01:00:00 | 1 |
| 1998-04-01 02:00:00 | 1 |
| 1998-04-01 03:00:00 | 1 |
| 1998-04-01 04:00:00 | 1 |
| 1998-04-01 05:00:00 | 1 |
| 1998-04-01 06:00:00 | 1 |
| 1998-04-01 07:00:00 | 1 |
| 1998-04-01 08:00:00 | 1 |
| 1998-04-01 09:00:00 | 1 |
| 1998-04-01 10:00:00 | 1 |

# Best Practice: Handling Missing Value

First, evaluate the quality of the data

**Time Series Data**

- ❑ Use the **forward imputation method** to propagate the last known valid value
- ❑ If the **top rows** of the time series are missing, simply **drop them** if no prior values exist

**Cross-sectional Data**

- ❑ For **categorical data**, fill missing values using the **mode** (most frequent value)
- ❑ For **numerical data**, use the **median** (if skewed) or the **mean** (if normally distributed)

# Time Series Data: Rolling and Cumulative Methods

# Business Question: Stock Price Prediction

A financial analyst at a trading firm proposed a few ideas to predict tomorrow's stock price based on the historical prices:

- ❑ Yesterday's price
- ❑ Average of the last 3 days' close prices
- ❑ More weight on recent prices to get the weighted average of the last 3 days' close prices

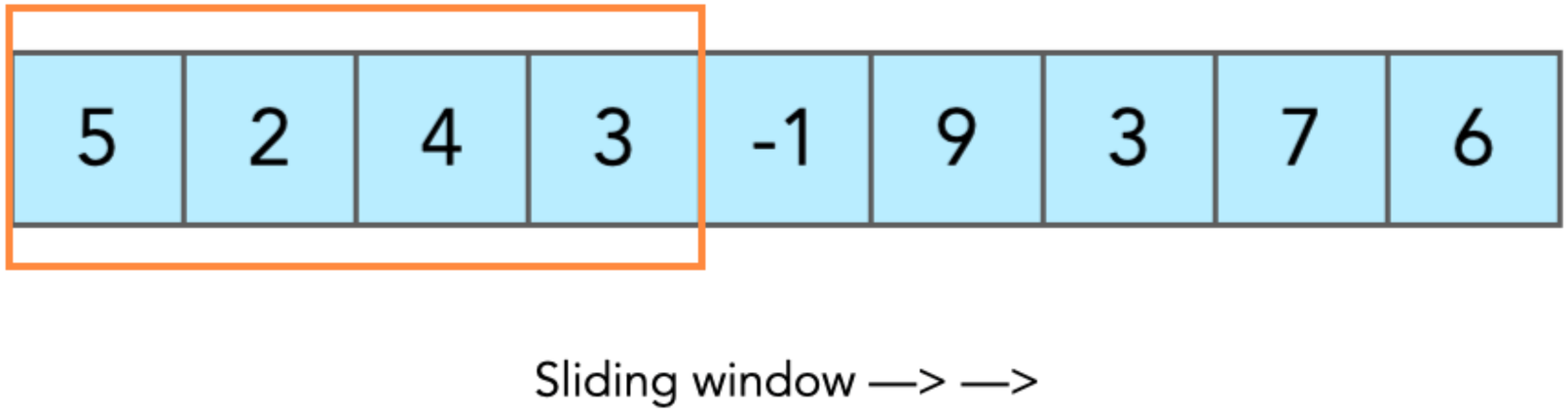
```
sp500['Close'].tail(5)
```

| Date | Close |
|---|---|
| 2025-03-31 | 559.390015 |
| 2025-04-01 | 560.969971 |
| 2025-04-02 | 564.520020 |
| 2025-04-03 | 536.700012 |
| 2025-04-04 | 505.279999 |

**What are the predictions of April 5th price based on the data?**

# Rolling Methods

A **rolling method** is used to apply a function (like mean, sum, etc.) over a specific range of data (moving window/sliding window) rather than the entire dataset



# Example: Rolling Average

```
sp500["RollMean3"] = sp500["Close"].rolling(3).mean()
sp500[["Close", "RollMean3"]].tail(5)
```

| Date | Close | RollMean3 |
|---|---|---|
| 2025-03-31 | 559.390015 | 560.710002 |
| 2025-04-01 | 560.969971 | 558.673319 |
| 2025-04-02 | 564.520020 | 561.626668 |
| 2025-04-03 | 536.700012 | 554.063334 |
| 2025-04-04 | 505.279999 | 535.500010 |

**Width of the Sliding Window is 3**

**The mean() function applied to all values in the window**

rolling() function can be applied to compute all kinds of **dynamic statistic,** e.g. rolling sum, min/max in the window

# Cumulative Methods

Cumulative method **applies** a function (sum, max or min) accumulatively from the start of the series to each point

| date | sales | cumulative_sales |
|---|---|---|
| 2023-01-01 | 4,578.00 | |
| 2023-01-02 | 6,486.00 | |
| 2023-01-03 | 7,839.00 | |
| 2023-01-04 | 9,044.00 | |
| 2023-01-05 | 7,063.00 | |
| 2023-01-06 | 9,145.00 | |

# Cumulative Sum

```
dailysales = pd.DataFrame({
    'Day': ['Mon', 'Tue', 'Wed', 'Thu', 'Fri'],
    'Sales': [100, 150, 130, 170, 200]
})

# Calculate cumulative sum of sales
dailysales['Cumulative Sales'] = dailysales['Sales'].cumsum()
dailysales
```

| | Day | Sales | Cumulative Sales |
|---|---|---|---|
| 0 | Mon | 100 | 100 |
| 1 | Tue | 150 | 250 |
| 2 | Wed | 130 | 380 |
| 3 | Thu | 170 | 550 |
| 4 | Fri | 200 | 750 |

cumsum() function compute the sum starting from the first row