

# JPMC Machine Learning Essentials





# Table of Contents

1. Introduction to Data Analysis: 5
  2. Introduction to Statistics: 79
  3. Probability Distributions: 123
  4. Inferential Statistics and Python: 161
  5. Data Cleaning and Exploratory Data Analysis: 229
  6. Regression Analysis: 272
  7. Working with Pandas DataFrames: 357
- 

- 
- 
8. Data Wrangling with Pandas: 527
  9. Aggregating Pandas DataFrames: 709
  10. Visualizing Data with Pandas and Matplotlib: 883
  11. Plotting with Seaborn & Customization Techniques: 1002
  12. Financial Analysis - Bitcoin & the Stock Market: 1113
  13. Rule-Based Anomaly Detection: 1271
  14. Summary: The Road Ahead: 1386

# DAY 1



# 1: Introduction to Data Analysis



# Introduction to Data Analysis

The following topics will be covered in this lesson:

- The fundamentals of data analysis
- Statistical foundations
- Setting up a virtual environment



# lesson materials

In order to get a local copy of the files, we have a few options (ordered from least useful to most useful):

- Download the ZIP file and extract the files locally.
- Clone the repository without forking it.
- Fork the repository and then clone it.



# lesson materials

The relevant buttons for initiating this process are circled in the following screenshot:

A screenshot of a GitHub repository page for a project titled "Hands-On Data Analysis with Pandas". The top navigation bar includes "Watch", "Star", and a circled "Fork" button. Below the bar, tabs for "Code", "Issues", "Pull requests", "Actions", "Security", and "Insights" are visible, with "Code" being the active tab. A secondary navigation bar shows "master", "1 branch", and "2 tags". The main content area displays a list of recent commits:

Commit	Message	Time Ago
656cd88	Start updating README.	10 hours ago
256 commits		
_img		7 days ago
appendix	Update the image for the choosing the appropriate plot diagram.	14 months ago
ch_01	Fix references.	10 hours ago

On the right side, there is an "About" section with the following text:

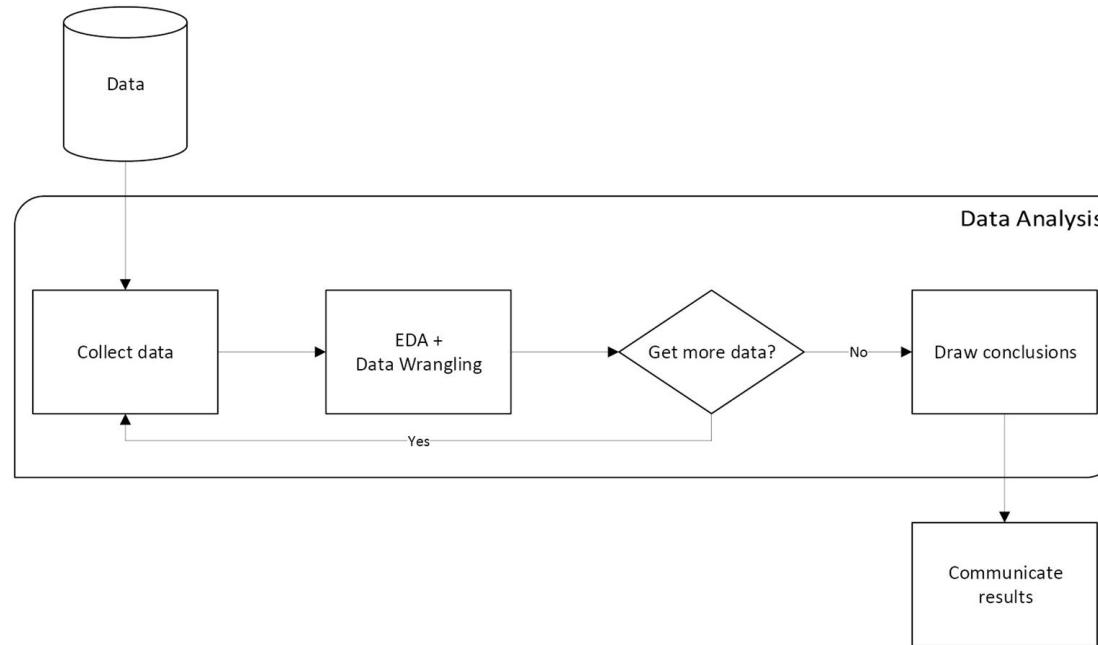
Materials for following along with  
Hands-On Data Analysis with Pandas,  
2nd edition

[www.packtpub.com/product/hands-o...](http://www.packtpub.com/product/hands-o...)

Tags include: data-analysis, data-analysis-python, data-analysis-pandas, data-science.

# The fundamentals of data analysis

- The following diagram depicts a generalized workflow:





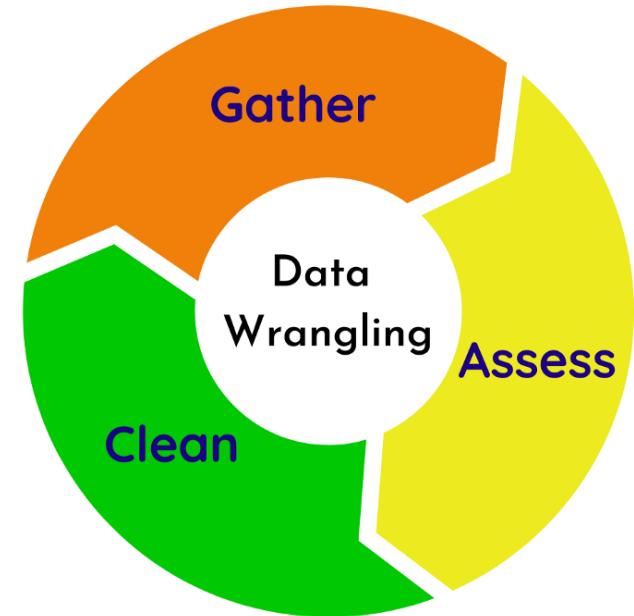
DATA COLLECTION

# Data collection

- Data collection is the natural first step for any data analysis—we can't analyze data we don't have.
- In reality, our analysis can begin even before we have the data.
- When we decide what we want to investigate or analyze, we have to think about what kind of data we can collect that will be useful for our analysis.

# Data wrangling

- Data wrangling is the process of preparing the data and getting it into a format that can be used for analysis.
- The unfortunate reality of data is that it is often dirty, meaning that it requires cleaning (preparation) before it can be used.

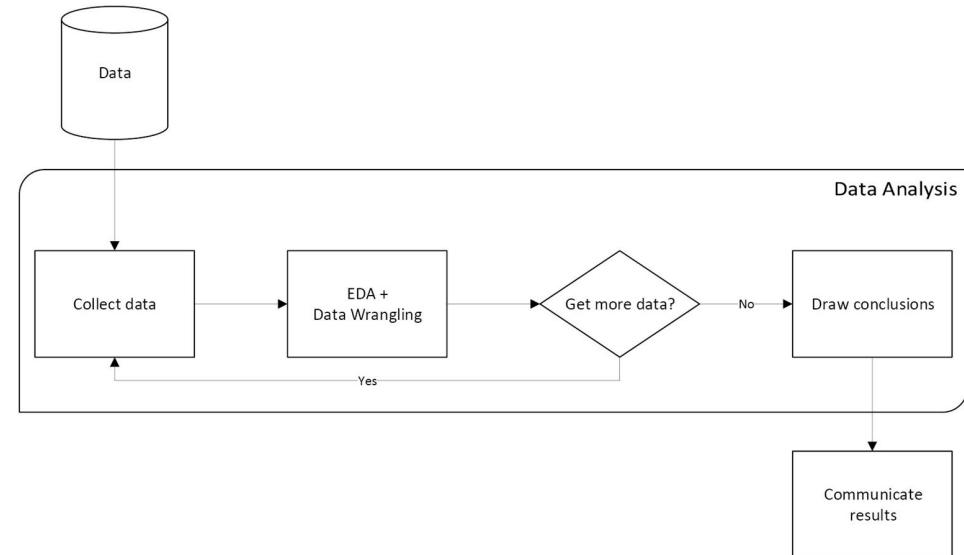


# Exploratory data analysis

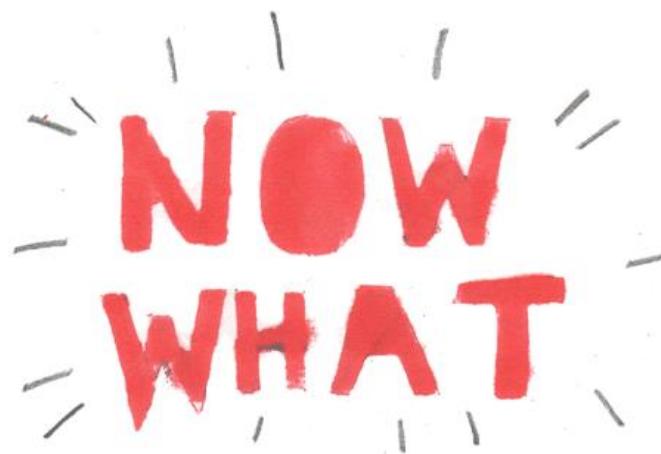
EDA and data wrangling shared a box.

This is because they are closely tied:

- Data needs to be prepped before EDA.
- Visualizations that are created during EDA may indicate the need for additional data cleaning.
- Data wrangling uses summary statistics to look for potential data issues, while EDA uses them to understand the data.



# Drawing conclusions



- After we have collected the data for our analysis, cleaned it up, and performed some thorough EDA, it is time to draw conclusions.
- This is where we summarize our findings from EDA and decide the next steps.

# Statistical foundations



- When we want to make observations about the data we are analyzing, we often, if not always, turn to statistics in some fashion.
- The data we have is referred to as the sample, which was observed from (and is a subset of) the population.
- Two broad categories of statistics are descriptive and inferential statistics.



# Sampling

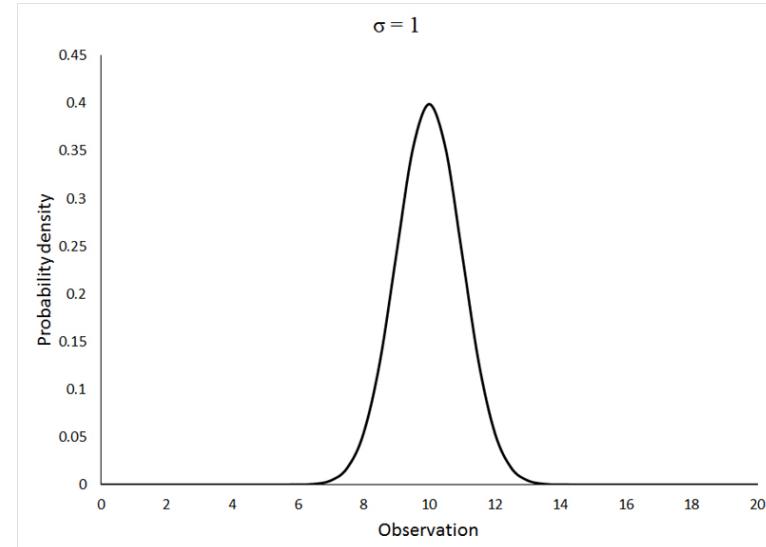
- There's an important thing to remember before we attempt any analysis: our sample must be a random sample that is representative of the population.
  - This means that the data must be sampled without bias (for example, if we are asking people whether they like a certain sports team, we can't only ask fans of the team)
  - We should have (ideally) members of all distinct groups from the population in our sample (in the sports team example, we can't just ask men)



# Descriptive statistics



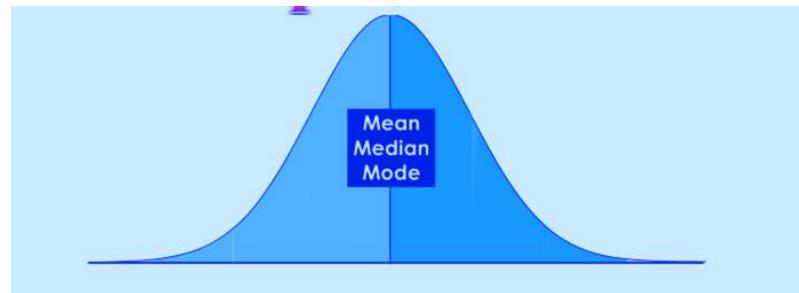
- Descriptive statistics are used to describe and/or summarize the data we are working with.
  - We can start our summarization of the data with a measure of central tendency, which describes where most of the data is centered around, and a measure of spread or dispersion, which indicates how far apart values are.



# Descriptive statistics

## Measures of central tendency

- Measures of central tendency describe the center of our distribution of data.
- There are three common statistics that are used as measures of center: mean, median, and mode.
- Each has its own strengths, depending on the data we are working with.





# Descriptive statistics

## Mean

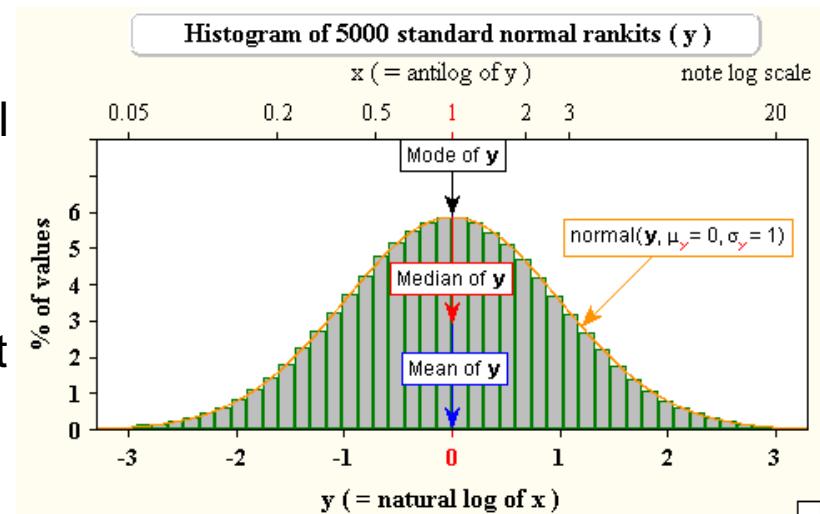
- Perhaps the most common statistic for summarizing data is the average, or mean.
- The population mean is denoted by  $\mu$  (the Greek letter mu), and the sample mean is written as  $\bar{x}$  (pronounced X-bar).
- The sample mean is calculated by summing all the values and dividing by the count of values; for example, the mean of the numbers 0, 1, 1, 2, and 9 is 2.6  $((0 + 1 + 1 + 2 + 9)/5)$ :

$$\bar{x} = \frac{\sum_1^n x_i}{n}$$

# Descriptive statistics

## Median

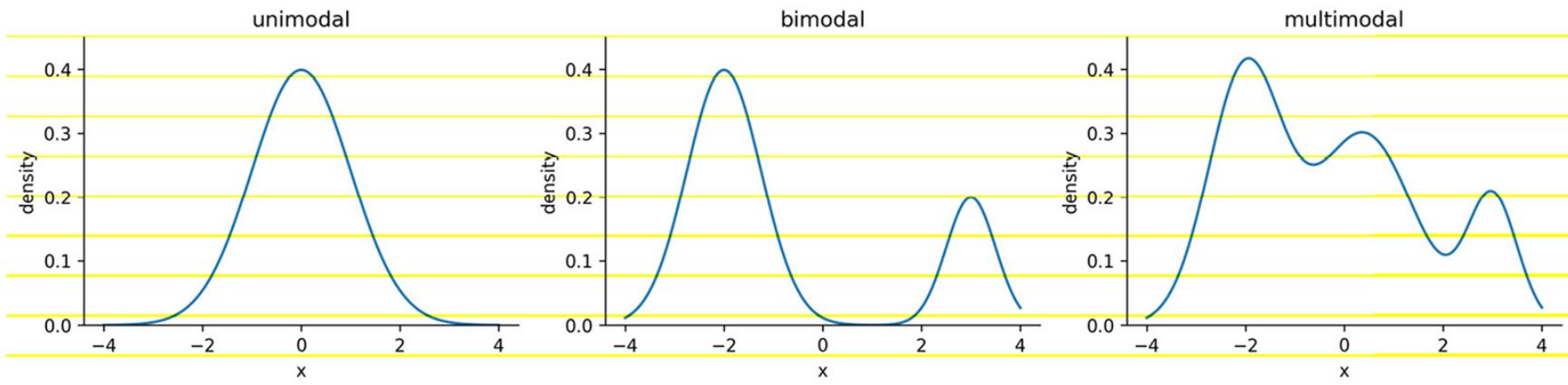
- Unlike the mean, the median is robust to outliers.
- Consider income in the US; the top 1% is much higher than the rest of the population, so this will skew the mean to be higher and distort the perception of the average person's income.
- However, the median will be more representative of the average income because it is the 50th percentile of our data; this means that 50% of the values are greater than the median and 50% are less than the median.



# Descriptive statistics

## Mode

- The mode is the most common value in the data (if we, once again, have the numbers 0, 1, 1, 2, and 9, then 1 is the mode).





# Descriptive statistics

## Measures of spread

- Knowing where the center of the distribution is only gets us partially to being able to summarize the distribution of our data—we need to know how values fall around the center and how far apart they are.

# Descriptive statistics



## Range

- The range is the distance between the smallest value (minimum) and the largest value (maximum).
- The units of the range will be the same units as our data.

$$\text{range} = \max(X) - \min(X)$$

# Descriptive statistics

## Variance

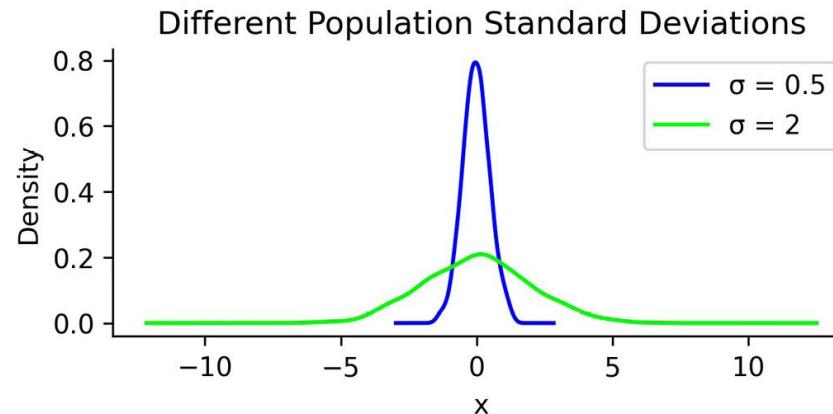
- The variance describes how far apart observations are spread out from their average value (the mean).
- The population variance is denoted as  $\sigma^2$  (pronounced sigma-squared), and the sample variance is written as  $s^2$ .
- It is calculated as the average squared distance from the mean.

$$s^2 = \frac{\sum_1^n (x_i - \bar{x})^2}{n - 1}$$

# Descriptive statistics

## Standard deviation

- We can use the standard deviation to see how far from the mean data points are on average.
- A small standard deviation means that values are close to the mean, while a large standard deviation means that values are dispersed more widely.



# Descriptive statistics

- The standard deviation is simply the square root of the variance
- By performing this operation, we get a statistic in units that we can make sense of again (\$ for our income example):

$$s = \sqrt{\frac{\sum_{i=1}^n (x_i - \bar{x})^2}{n-1}} = \sqrt{s^2}$$

# Descriptive statistics

## Coefficient of variation

- When we moved from variance to standard deviation, we were looking to get to units that made sense; however, if we then want to compare the level of dispersion of one dataset to another, we would need to have the same units once again.
- One way around this is to calculate the coefficient of variation (CV), which is unitless.
- The CV is the ratio of the standard deviation to the mean:

$$CV = \frac{s}{\bar{x}}$$



- One common measure for this is the interquartile range (IQR), which is the distance between the 3rd and 1st quartiles:

$$IQR = Q_3 - Q_1$$

# Descriptive statistics

## Quartile coefficient of dispersion

- This statistic is also unitless, so it can be used to compare datasets.
- It is calculated by dividing the semi-quartile range (half the IQR) by the midhinge (midpoint between the first and third quartiles):

$$QCD = \frac{\frac{Q_3 - Q_1}{2}}{\frac{Q_1 + Q_3}{2}} = \frac{Q_3 - Q_1}{Q_3 + Q_1}$$

# Descriptive statistics

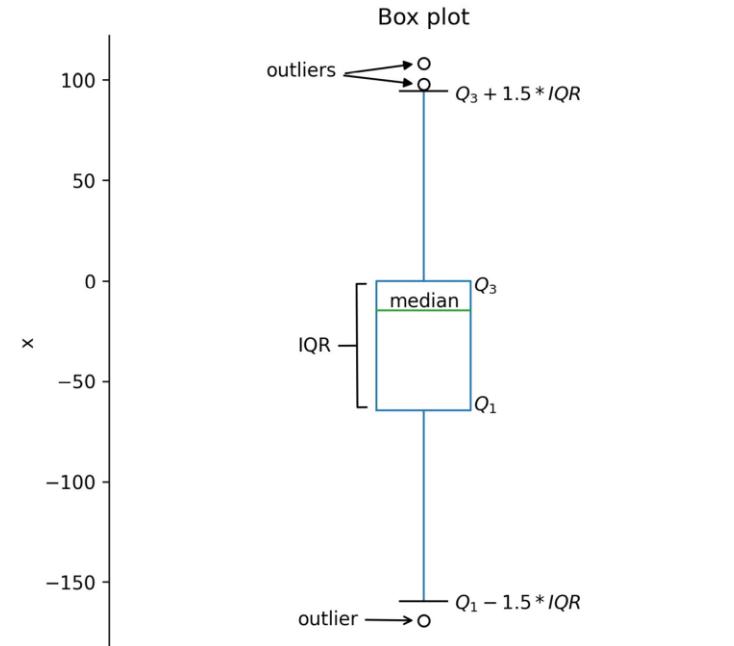
## Summarizing data

- We have seen many examples of descriptive statistics that we can use to summarize our data by its center and dispersion; in practice, looking at the 5-number summary and visualizing the distribution prove to be helpful first steps before diving into some of the other aforementioned metrics.
- The 5-number summary, as its name indicates, provides five descriptive statistics that summarize our data:

	Quartile	Statistic	Percentile
1.	$Q_0$	minimum	$0^{th}$
2.	$Q_1$	N/A	$25^{th}$
3.	$Q_2$	median	$50^{th}$
4.	$Q_3$	N/A	$75^{th}$
5.	$Q_4$	maximum	$100^{th}$

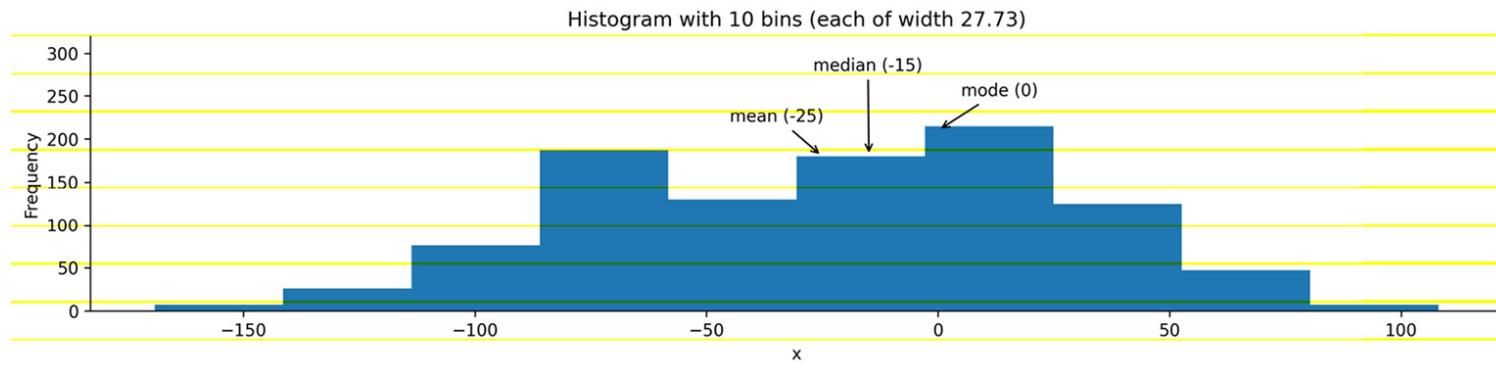
# Descriptive statistics

- A box plot (or box and whisker plot) is a visual representation of the 5-number summary.
- The median is denoted by a thick line in the box.
- The top of the box is  $Q_3$  and the bottom of the box is  $Q_1$ .
- Lines (whiskers) extend from both sides of the box boundaries toward the minimum and maximum.



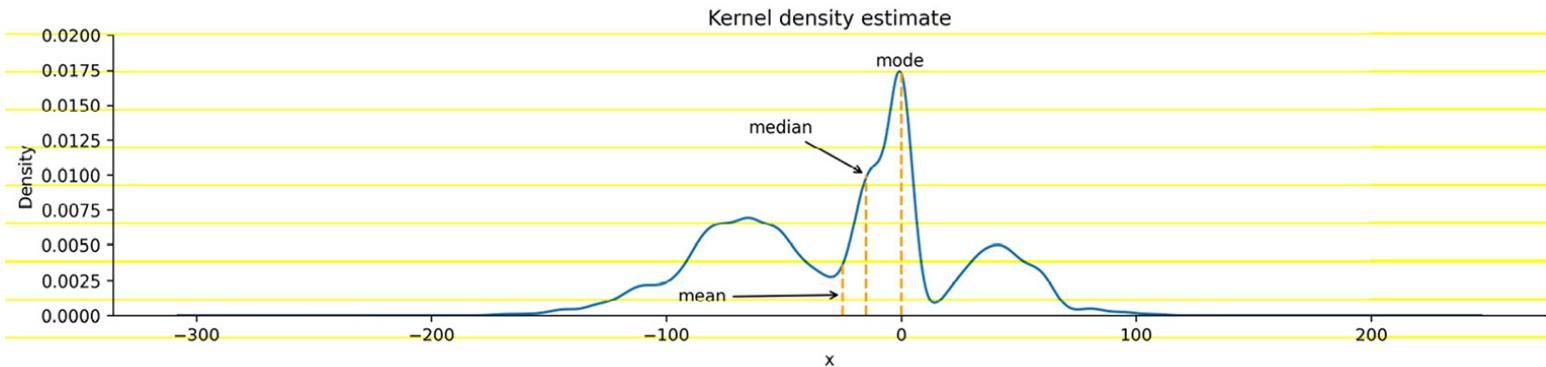
# Descriptive statistics

- To make a histogram, a certain number of equal-width bins are created, and then bars with heights for the number of values we have in each bin are added.
- The following plot is a histogram with 10 bins, showing the three measures of central tendency for the same data that was used to generate the box plot in Figure:



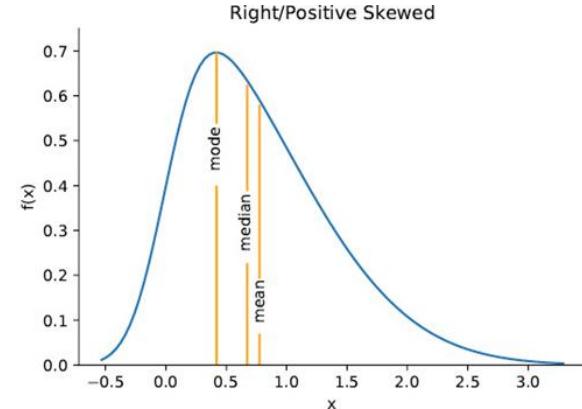
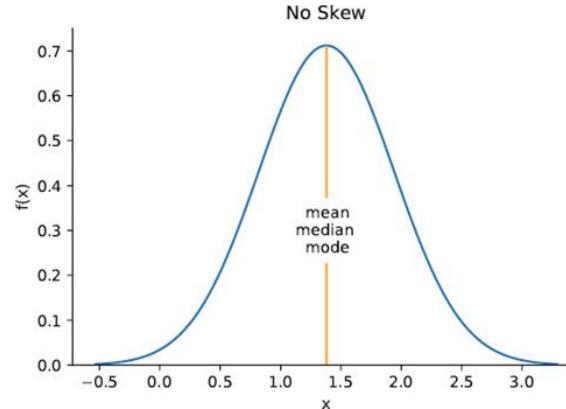
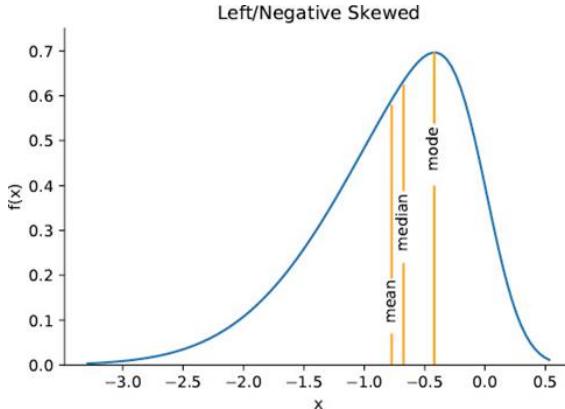
# Descriptive statistics

- KDEs are similar to histograms, except rather than creating bins for the data, they draw a smoothed curve, which is an estimate of the distribution's probability density function (PDF).
- The PDF is for continuous variables and tells us how probability is distributed over the values.
- Higher values for the PDF indicate higher likelihoods:



# Descriptive statistics

- A left (negative) skewed distribution has a long tail on the left-hand side; a right (positive) skewed distribution has a long tail on the right-hand side.
- In the presence of negative skew, the mean will be less than the median, while the reverse happens with a positive skew.
- When there is no skew, both will be equal:



# Descriptive statistics



- When we are interested in the probability of getting a value  $c$ , we use the cumulative distribution function (CDF), which is the integral (area under the curve) of the PDF:

$$CDF = F(x) = \int_{-\infty}^x f(t)dt$$

where  $f(t)$  is the PDF and  $\int_{-\infty}^{\infty} f(t)dt = 1$

# Descriptive statistics

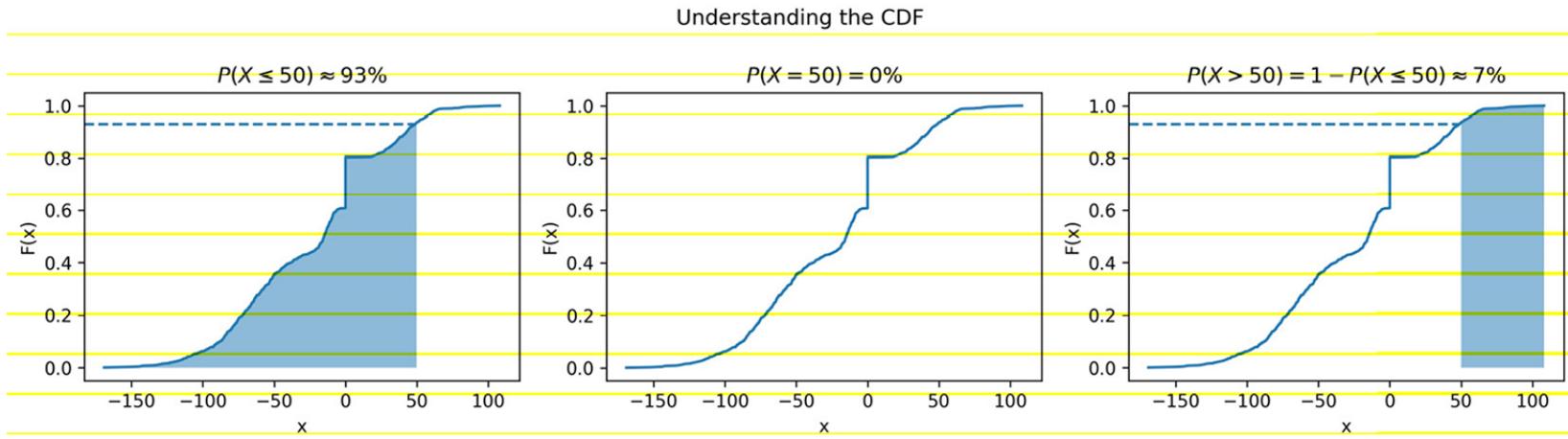


- This is because the probability will be the integral of the PDF from  $x$  to  $x$  (area under a curve with zero width), which is 0:

$$P(X = x) = \int_x^x f(t)dt = 0$$

# Descriptive statistics

- Let's visualize  $P(X \leq 50)$ ,  $P(X = 50)$ , and  $P(X > 50)$  as an example:





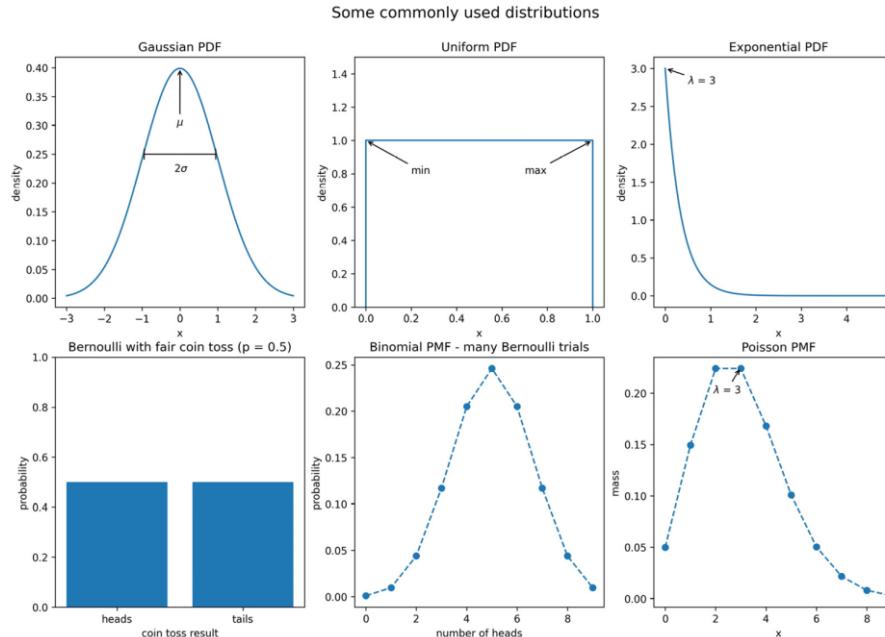
# Descriptive statistics

## Common distributions

- While there are many probability distributions, each with specific use cases, there are some that we will come across often.
- The Gaussian, or normal, looks like a bell curve and is parameterized by its mean ( $\mu$ ) and standard deviation ( $\sigma$ ).
- The standard normal (Z) has a mean of 0 and a standard deviation of 1.
- Many things in nature happen to follow the normal distribution, such as heights.

# Descriptive statistics

- We can visualize both discrete and continuous distributions; however, discrete distributions give us a probability mass function (PMF) instead of a PDF:



# Descriptive statistics

## Scaling data

- In order to compare variables from different distributions, we would have to scale the data, which we could do with the range by using min-max scaling.
- We take each data point, subtract the minimum of the dataset, then divide by the range. This normalizes our data (scales it to the range [0, 1]):

$$x_{scaled} = \frac{x - \min(X)}{range(X)}$$

# Descriptive statistics



- This isn't the only way to scale data; we can also use the mean and standard deviation.
- In this case, we would subtract the mean from each observation and then divide by the standard deviation to standardize the data.
- This gives us what is known as a Z-score:

$$Z_i = \frac{x_i - \bar{x}}{s}$$

# Descriptive statistics



- Quantifying relationships between variables.
- The covariance is a statistic for quantifying the relationship between variables by showing how one variable changes with respect to another (also referred to as their joint variance):

$$cov(X, Y) = E[(X - E[X])(Y - E[Y])]$$

# Descriptive statistics

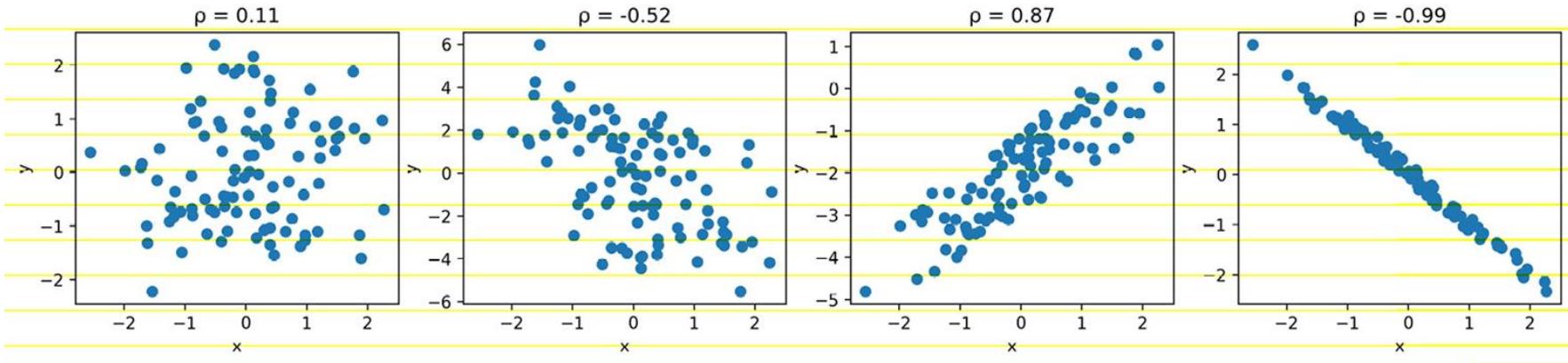


- To find the correlation, we calculate the Pearson correlation coefficient, symbolized by  $\rho$  (the Greek letter rho), by dividing the covariance by the product of the standard deviations of the variables:

$$\rho_{X,Y} = \frac{cov(X, Y)}{s_X s_Y}$$

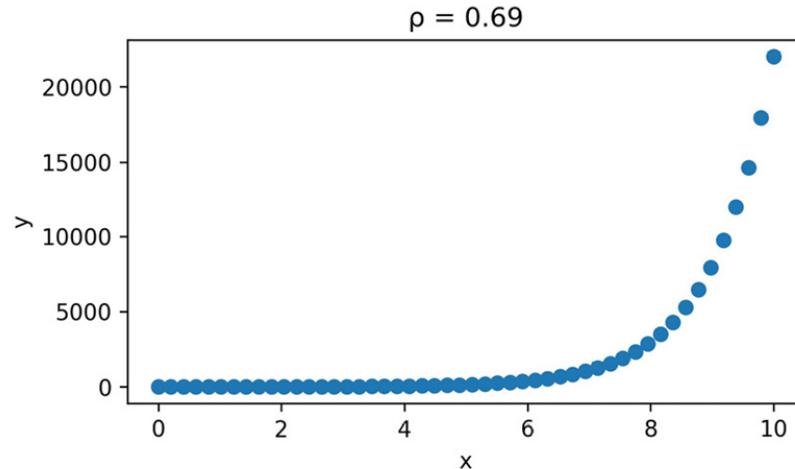
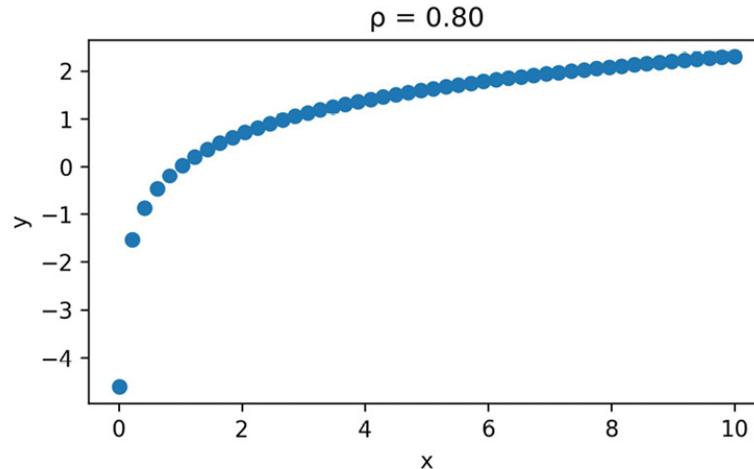
# Descriptive statistics

- We can also see how the points form a line:



# Descriptive statistics

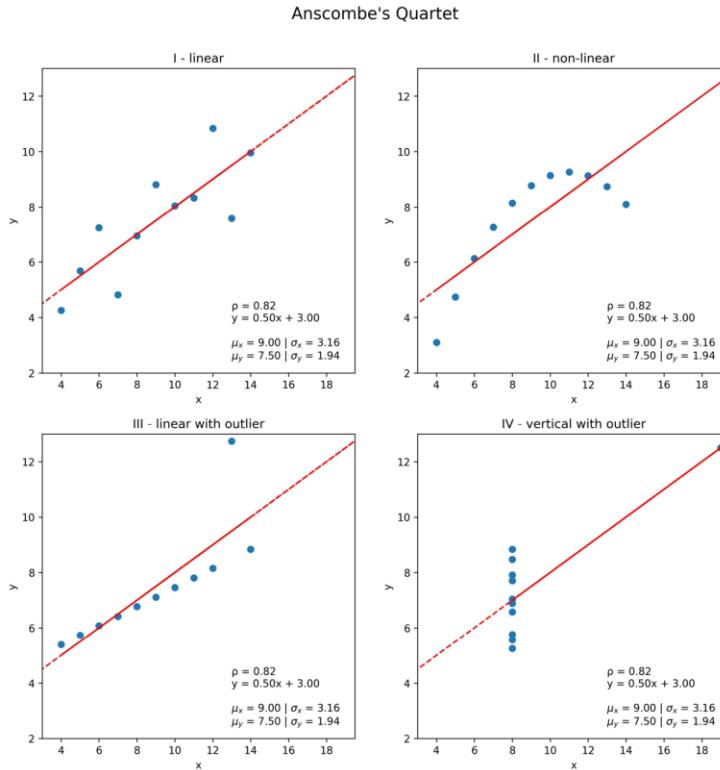
- Both of the following plots depict data with strong positive correlations, but it's pretty obvious when looking at the scatter plots that these are not linear.
- The one on the left is logarithmic, while the one on the right is exponential:



# Descriptive statistics

## Pitfalls of summary statistics

- Anscombe's quartet is a collection of four different datasets that have identical summary statistics and correlation coefficients, but when plotted, it is obvious they are not similar:





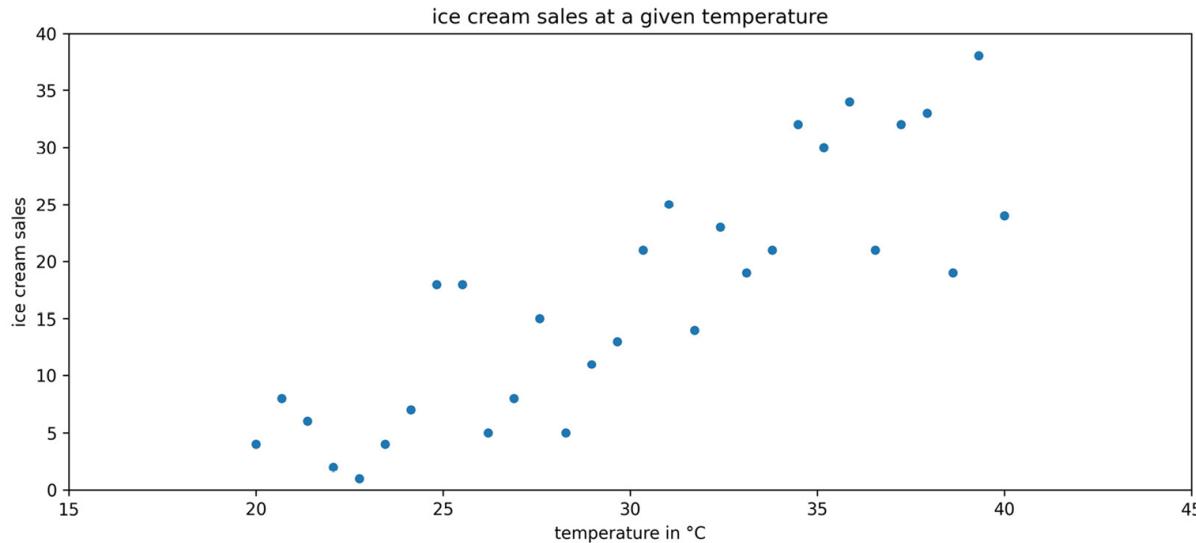
# Prediction and forecasting

- Say our favorite ice cream shop has asked us to help predict how many ice creams they can expect to sell on a given day.
- They are convinced that the temperature outside has a strong influence on their sales, so they have collected data on the number of ice creams sold at a given temperature.



# Prediction and forecasting

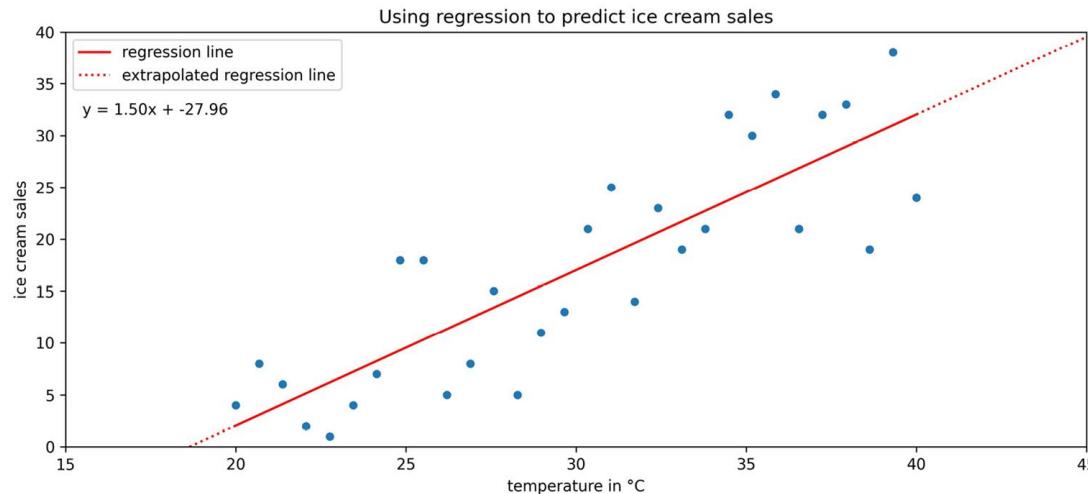
- We agree to help them, and the first thing we do is make a scatter plot of the data they collected:





# Prediction and forecasting

- While we can have many independent variables, our ice cream sales example only has one: temperature.
- Therefore, we will use simple linear regression to model the relationship as a line:





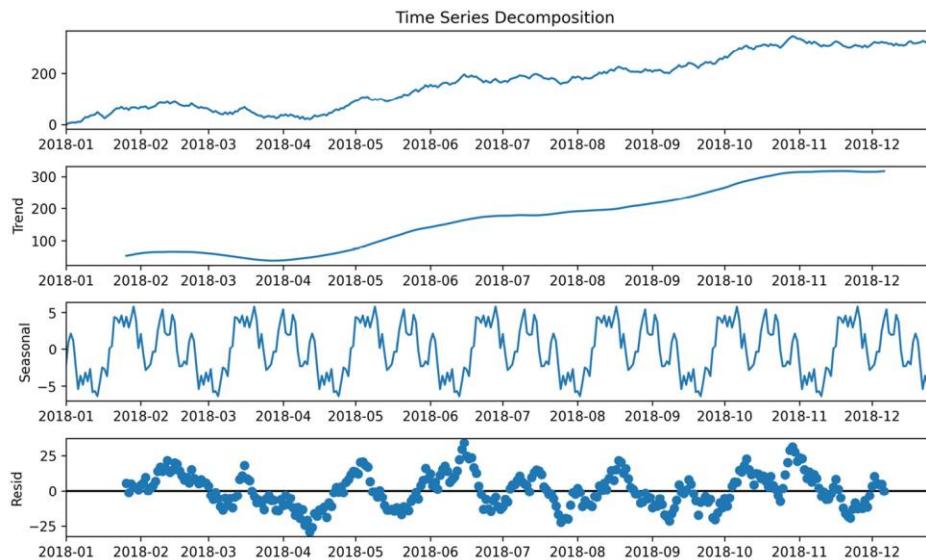
# Prediction and forecasting

- The regression line in the previous scatter plot yields the following equation for the relationship:

$$\text{ice cream sales} = 1.50 \times \text{temperature} - 27.96$$

# Prediction and forecasting

- We can use Python to decompose the time series into trend, seasonality, and noise or residuals.



# Inferential statistics

- With an experiment, we are able to directly influence the independent variable and randomly assign subjects to the control and test groups, such as A/B tests (for anything from website redesigns to ad copy).

# Setting up a virtual environment

- This course was written using Python 3.7.3, but the code should work for Python 3.7.1+, which is available on all major operating systems.
- In this section, we will go over how to set up the virtual environment in order to follow along with this course.

# Virtual environments

- A virtual environment allows us to create separate environments for each of our projects.
- Each of our environments will only have the packages that it needs installed.
- It's good practice to make a dedicated virtual environment for any projects we work on.



# Virtual environments

## Venv

- Python 3 comes with the `venv` module, which will create a virtual environment in the location of our choice.

The process of setting up and using a development environment is as follows (after Python is installed):

- Create a folder for the project.
- Use `venv` to create an environment in this folder.
- Activate the environment.
- Install Python packages in the environment with `pip`.
- Deactivate the environment when finished.

# Virtual environments

- To make a new directory and move to that directory, we can use the following command:

```
$ mkdir my_project && cd my_project
```



# Virtual environments

- Before moving on, use cd to navigate to the directory containing this course's repository.
- Note that the path will depend on where it was cloned/downloaded:

```
$ cd path/to/Directory
```

# Virtual environments



## Windows

- To create our environment for this course, we will use the venv module from the standard library.
- Note that we must provide a name for our environment (course\_env).
- Remember, if your Windows setup has python associated with Python 3, then use python instead of python3 in the following command:

```
C:\...> python3 -m venv course_env
```

# Virtual environments

- Now, we have a folder for our virtual environment named course\_env inside the repository folder that we cloned/downloaded earlier.
- In order to use the environment, we need to activate it:

```
C:\...> %cd%\course_env\Scripts\activate.bat
```



# Virtual environments

- Note that after we activate the virtual environment, we can see (course\_env) in front of our prompt on the command line; this lets us know we are in the environment:

```
(course_env) C:\...>
```

- When we are finished using the environment, we simply deactivate it:

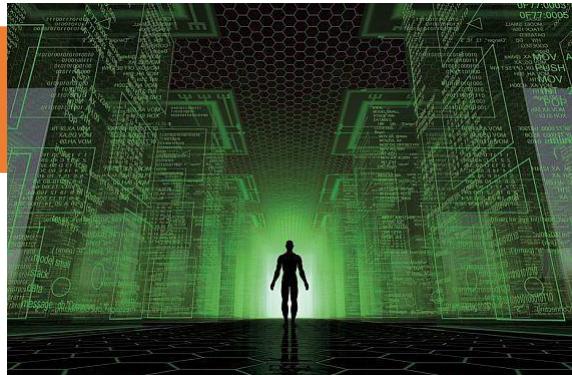
```
(course_env) C:\...> deactivate
```

# Virtual environments

## Linux/macOS

- To create our environment for this course, we will use the `venv` module from the standard library.
- Note that we must provide a name for our environment (`course_env`):

```
$ python3 -m venv course_env
```



# Virtual environments

- Now, we have a folder for our virtual environment named course\_env inside of the repository folder we cloned/downloaded earlier.

- In order to use the environment, we need to activate it:

```
$ source course_env/bin/activate
```

- Note that after we activate the virtual environment, we can see (course\_env) in front of our prompt on the command line; this lets us know we are in the environment:

```
(course_env) $
```

# Virtual environments

- When we are finished using the environment, we simply deactivate it:

(course\_env) \$ deactivate



# Virtual environments

## Conda

- Anaconda provides a way to set up a Python environment specifically for data science.
- It includes some of the packages we will use in this course, along with several others that may be necessary for tasks that aren't covered in this course (and also deals with dependencies outside of Python that might be tricky to install otherwise).

# Virtual environments

- To create a new conda environment for this course, called course\_env, run the following:

(base) \$ conda create --name course\_env



# Virtual environments

- Running `conda env list` will show all the conda environments on the system, which will now include `course_env`.
- The current active environment will have an asterisk (\*) next to it—by default, `base` will be active until we activate another environment:

```
(base) $ conda env list
# conda environments:
#
base          * /miniconda3
course_env      /miniconda3/envs/course_env
```

# Virtual environments

- To activate the course\_env environment, we run the following command:

```
(base) $ conda activate course_env
```

- Note that after we activate the virtual environment, we can see (course\_env) in front of our prompt on the command line; this lets us know we are in the environment:

```
(course_env) $
```

# Virtual environments

- When we are finished using the environment, we deactivate it:

```
(course_env) $ conda deactivate
```



# Installing the required Python packages

- Before installing anything, be sure to activate the virtual environment that you created with either venv or conda.
- Be advised that if the environment is not activated before running the following command, the packages will be installed outside the environment:

```
(course_env) $ pip3 install -r requirements.txt
```

# Why pandas?

- When it comes to data science in Python, the pandas library is pretty much ubiquitous.
- It is built on top of the NumPy library, which allows us to perform mathematical operations on arrays of single-type data efficiently.



# Jupyter Notebooks

## Launching JupyterLab

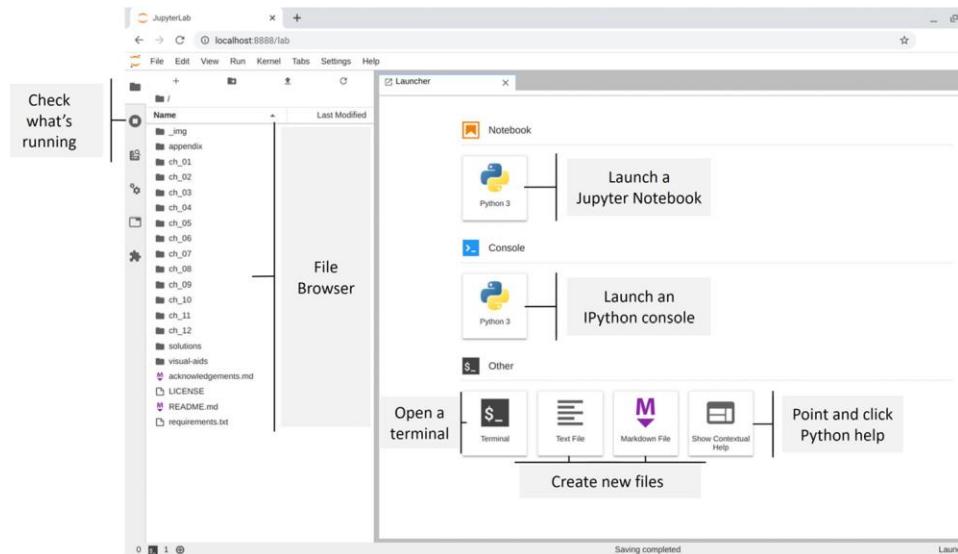
- JupyterLab is an IDE that allows us to create Jupyter Notebooks and Python scripts, interact with the terminal, create text documents, reference documentation, and much more from a clean web interface on our local machine.
- First, we activate our environment, and then we launch JupyterLab:

(course\_env) \$ jupyter lab



# Jupyter Notebooks

- This will then launch a window in the default browser with JupyterLab.
- We will be greeted with the Launcher tab and the File Browser pane to the left:



# Jupyter Notebooks

## Validating the virtual environment

- Open the checking\_your\_setup.ipynb notebook in the ch\_01 folder, as shown in the following screenshot:

The screenshot shows a Jupyter Notebook interface with the following details:

- File Structure:** The left sidebar shows a tree view of files in the "ch\_01" directory, including "check\_environment.py", "exercises.ipynb", "introduction\_to\_data\_analysis.ipynb", and "python\_101.ipynb".
- Current Notebook:** The main area displays "checking\_your\_setup.ipynb".
- Cell Content:** A code cell contains the following Python code:

```
[1]: from check_environment import run_checks
run_checks()
```
- Output:** The output shows the results of the package check:

```
Using Python in /home/stefanemolin/book_env:
Python version is 3.7.3 (default, Dec 29 2019, 18:57:59)
[GCC 8.3.0]
graphviz
imblearn
jupyterlab
matplotlib
numba
pandas
requests
sklearn
scipy
seaborn
sqlalchemy
statsmodels
wheel
low_level_attempt_simulator
ml_utils
stock_analysis
visual_nids
```
- Toolbar:** The top toolbar includes options like "Cut cell(s)", "Copy cell(s)", "Paste cell(s)", "Interrupt kernel", "Restart kernel", "Status", and "Kernel" (set to "Python 3").
- Context Menu:** A context menu is open over the first cell, with options: "Add new cell", "Save", "Selected cell", "Run cell(s)", "Cell type", and "Run this cell to check your setup".
- Status Bar:** The bottom status bar shows "Mode: Command", "Ln 1, Col 1", and the file name "checking\_your\_setup.ipynb".

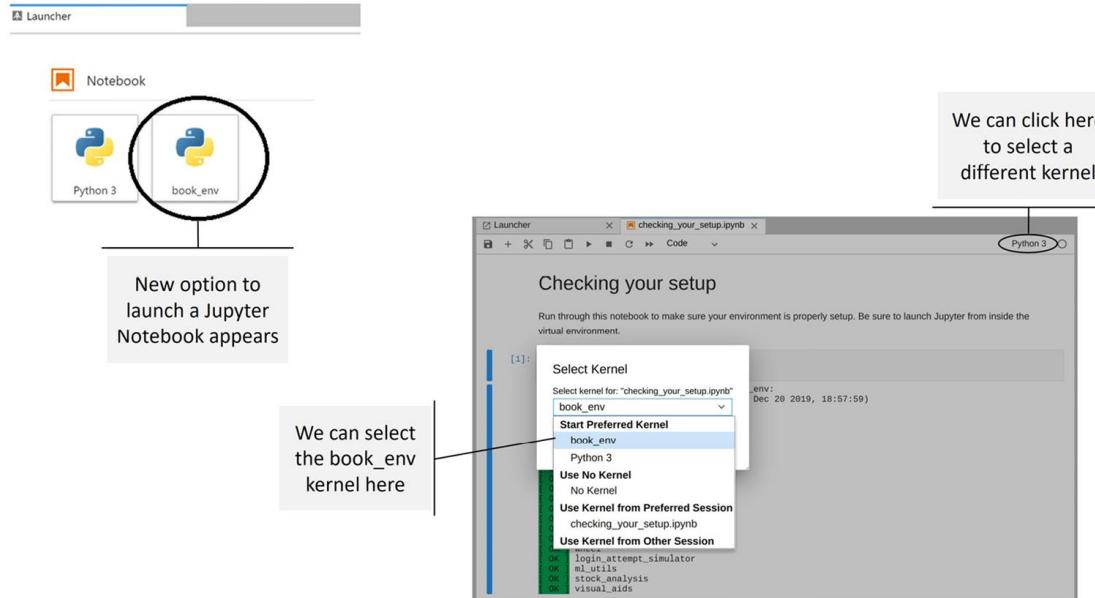
# Jupyter Notebooks

- Click on the code cell indicated in the previous screenshot and run it by clicking the play (▶) button.
- If everything shows up in green, the environment is all set up.
- However, if this isn't the case, run the following command from the virtual environment to create a special kernel with the course\_env virtual environment for use with Jupyter:

```
(course_env) $ ipython kernel install --user --name=course_env
```

# Jupyter Notebooks

- This adds an additional option in the Launcher tab, and we can now switch to the course\_env kernel from a Jupyter Notebook as well:



# Jupyter Notebooks

## Closing JupyterLab

- Closing the browser with JupyterLab in it doesn't stop JupyterLab or the kernels it is running (we also won't get the command-line interface back).
- To shut down JupyterLab entirely, we need to hit Ctrl + C (which is a keyboard interrupt signal that lets JupyterLab know we want to shut it down) a couple of times in the terminal until we get the prompt back:

...

```
[I 17:36:53.166 LabApp] Interrupted...
[I 17:36:53.168 LabApp] Shutting down 1 kernel
[I 17:36:53.770 LabApp] Kernel shutdown: a38e1[...]b44f
(course_env) $
```

# Summary

- In this lesson, we learned about the main processes in conducting data analysis: data collection, data wrangling, EDA, and drawing conclusions.
- We followed that up with an overview of descriptive statistics and learned how to describe the central tendency and spread of our data; how to summarize it both numerically and visually using the 5-number summary, box plots, histograms, and kernel density estimates
- How to scale our data; and how to quantify relationships between variables in our dataset.

# "Complete Exercises"

# "Complete Lab 1"

# 2: Introduction to Statistics



# Theory

## What is Statistics?

Statistics is “A telescope that allows us to study the large terrain and make it accessible to our unaided vision”



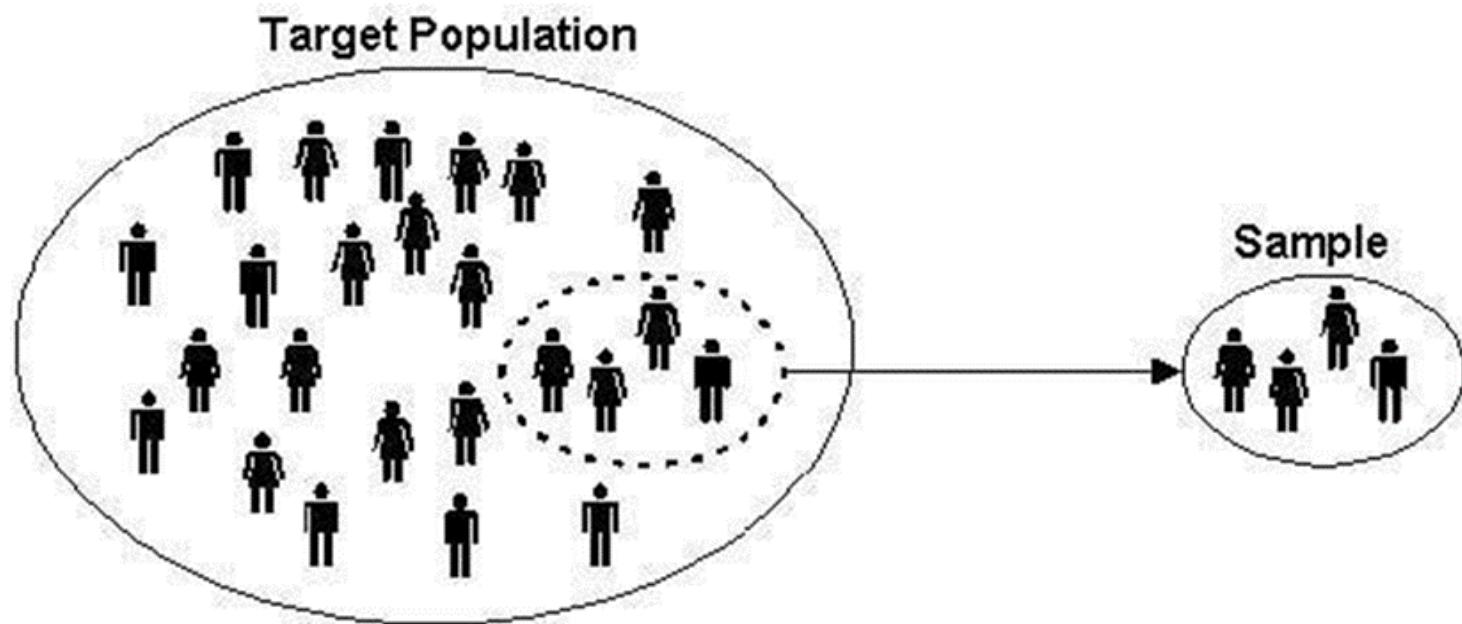
# Statistics – Big Picture

Statistics provides a way of organizing data to extract information on a wider and objective basis than relying on personal experience. It is a branch of mathematics working with

- Data Gathering
- Data Understanding
- Data Analysis/Interpretation
- Data Presentation



# Basic Statistical Terminology



# Parameter and Statistic

**Parameter:** A descriptive measure of the population.

For example,

- population mean -  $\mu$
- population variance –  $\sigma^2$
- population standard deviation -  $\sigma$

**Statistic:** A descriptive measure of the sample. For example,

- sample mean -  $\bar{x}$
- sample variance -  $s^2$
- sample standard deviation –  $s$

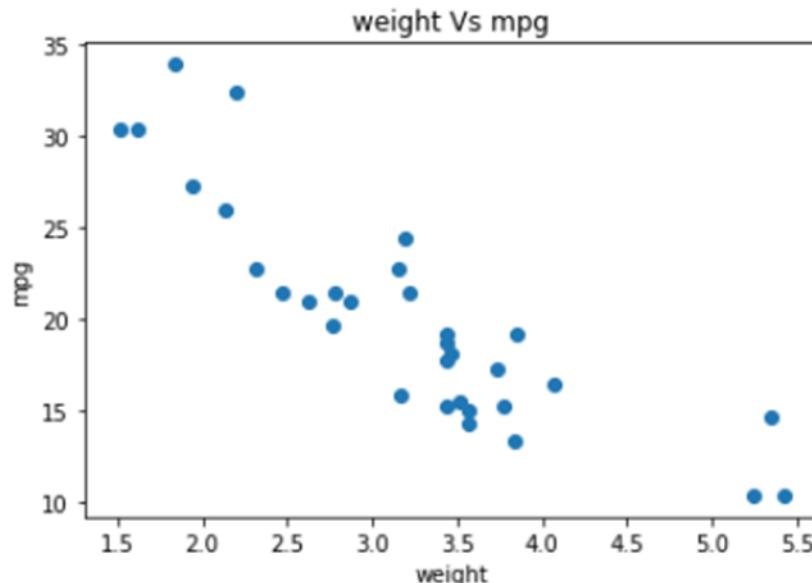


# Variables and Data (Example of data)

model	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21	6	160	110	3.9	2.62	16.46	0	1	4	4
Mazda RX4 Wag	21	6	160	110	3.9	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108	93	3.85	2.32	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258	110	3.08	3.215	19.44	1	0	3	1
Hornet Sportabout	18.7	8	360	175	3.15	3.44	17.02	0	0	3	2
Valiant	18.1	6	225	105	2.76	3.46	20.22	1	0	3	1
Duster 360	14.3	8	360	245	3.21	3.57	15.84	0	0	3	4
Merc 240D	24.4	4	146.7	62	3.69	3.19	20	1	0	4	2
Merc 230	22.8	4	140.8	95	3.92	3.15	22.9	1	0	4	2
Merc 280	19.2	6	167.6	123	3.92	3.44	18.3	1	0	4	4
Merc 280C	17.8	6	167.6	123	3.92	3.44	18.9	1	0	4	4

# Variables – Dependent and Independent

- An independent variable (experimental or predictor) is a variable that is being manipulated in an experiment in order to observe the effect on dependent variable (Outcome).



# Data

Data is classified into two types Numerical and Categorical

- Categorical Data
- Numerical Data

# Levels of Measurement Scales

- **Nominal scale:** The nominal scale could simply be called “labels

Gender	Car Color	Name
Male	Black	Sam
Female	Red	Jack
Male	Blue	John
Female	White	Don

# Levels of Measurement Scales

- **Ordinal scale:** The order of the values is what's important and significant, but the difference between each one is not really known. Here are some examples, below

Shirt Size	Feedback
Small	Poor
Medium	Good
Large	Better
Extra Large	Excellent

# Descriptive Statistics



- Descriptive statistics involves organizing, summarizing, and presenting data in an informative way.
- Descriptive statistics, unlike inferential statistics, seeks to describe the data, but does not attempt to make inferences from the sample to the whole population.

# Different types of Descriptive Statistics



Descriptive statistics are broken down into two categories

- Measure of Central Tendency
- Measure of Variability (Spread)

# Mean:

- Mean is a central tendency of the data i.e. a number around which a whole data is spread out.
- Formula for sample mean:

$$\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i$$

mean  
median  
mode

# Mean:

- Similarly, for a population data of size  $N$ , the population mean is:

$$\mu = \frac{1}{N} \sum_{i=1}^N x_i$$

# Median:

Median is the value which divides the data into 2 equal parts i.e. number of terms on right side of it is same as number of terms on left side of it when data is arranged in either ascending or descending order.

- May not exist as a data point in the set
- Influenced by position of items, but not their values
- Median is not influenced by extreme values

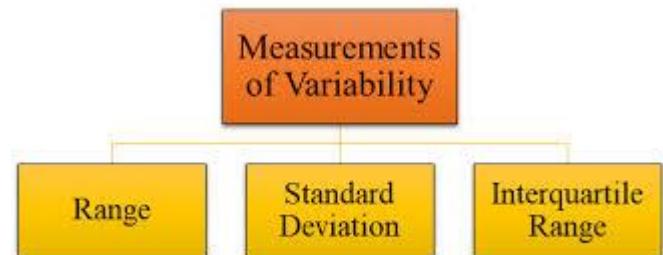
# Mode

Mode: Mode is the most commonly occurring value

- Mode exists as a data point.
- Useful for qualitative data.

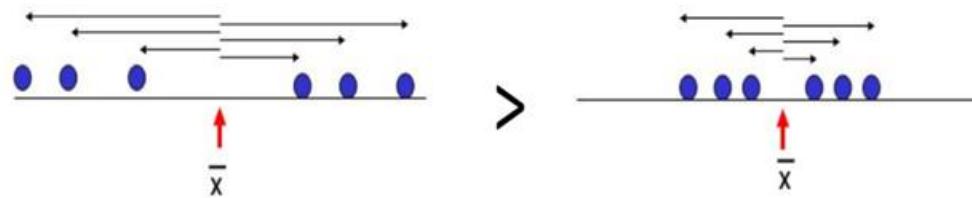
# Measure of Variability (Spread / Dispersion)

- The measures that help us to know about the spread of a data set are called measures of dispersion.



# Measure of Variability (Spread / Dispersion)

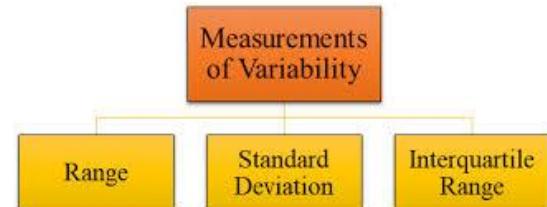
- **Standard deviation:** Standard deviation is the measurement of average distance between each quantity and mean, That is, how data is spread out from mean.
- A low standard deviation indicates that the data points tend to be close to the mean of the data set, while a high standard deviation indicates that the data points are spread out over a wider range of values.



# Measure of Variability (Spread / Dispersion)

- Sample Standard Deviation is denoted by “S”

$$s = \sqrt{\frac{\sum_{i=1}^n (x_i - \bar{x})^2}{n-1}}$$



# Measure of Variability (Spread / Dispersion)

- Population Standard Deviation is denoted by “ $\sigma$ ” (sigma)

$$\sigma = \sqrt{\frac{1}{N} \sum_{i=1}^N (x_i - \mu)^2}, \text{ where } \mu = \frac{1}{N} \sum_{i=1}^N x_i.$$

# Variance

- Variance is a square of average distance between each quantity and mean.
- That is, it is a square of standard deviation.

$$\text{Variance} = (S.D.)^2$$



# Variance

## The variance of Population and Sample

The variance of a population is:

$$\sigma^2 = \frac{\sum_{i=1}^N (x_i - \mu)^2}{N}$$

Population Mean  
Population Size

➤ The variance of a sample is:

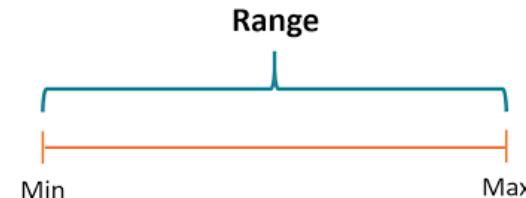
$$s^2 = \frac{\sum_{i=1}^n (x_i - \bar{x})^2}{n-1}$$

Sample Mean  
Note! the denominator is sample size (n) minus one !

# Range:

Range is one of the simplest techniques of descriptive statistics. It is the difference between lowest and highest value.

- It is easy to calculate.
- It is implemented for both “best” or “worst” case scenarios.
- Too sensitive for extreme values.



# Levels of Measurement Scales

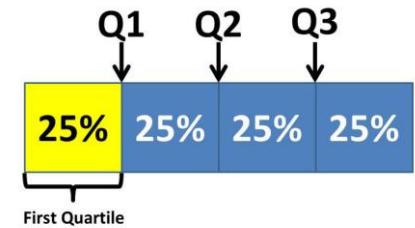
- **Percentile:** Percentile is a way to represent the position of a value in a data set.
- To calculate percentile, values in the data set should always be in ascending order.

Example:

12, 24, 41, 51, 67, 67, 85, 99

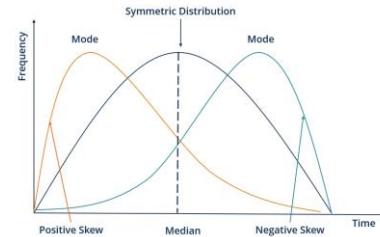
# Quartile:

- In statistics and probability, quartile are values that divide your data into quarters provided data is sorted in an ascending order.

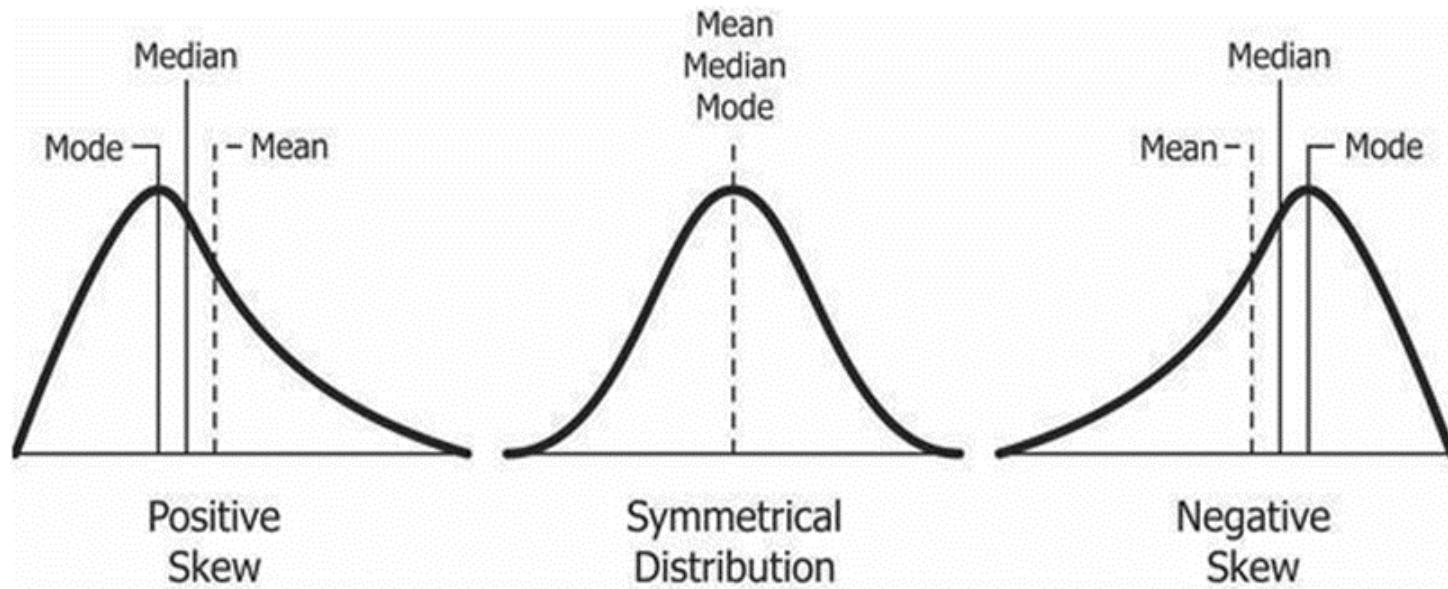


# Skewness:

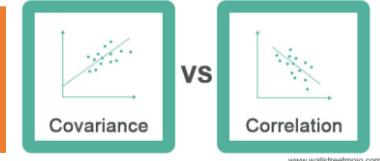
- Skewness is a measure of the asymmetry of the probability distribution of a real-valued random variable about its mean.
- The skewness value can be positive or negative or undefined.



# Skewness:



# Covariance and Correlation



- Covariance studies the direction between two continuous variables and Correlation studies the direction and strength between two continuous variables and helps in understanding how strongly those two continuous variables are associated with each other.

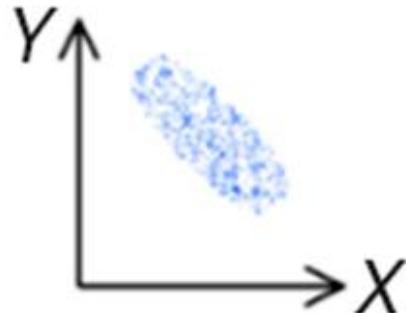
# What is Covariance Matrix?

- Suppose we have two variables X and Y, then the covariance between these two variables is represented as  $\text{Cov}(X, Y)$ .
- If  $\sum(X)$  and  $\sum(Y)$  are the expected values of the variables, the covariance formula can be represented as:

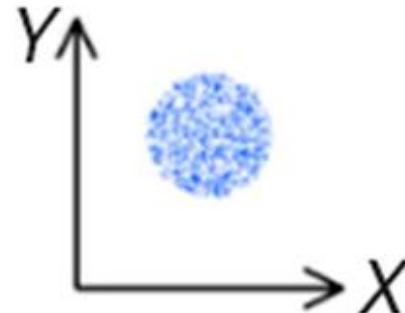
$$\text{COV}(X, Y) = \frac{1}{n-1} \sum_{i=1}^n (x_i - E(X))(y_i - E(Y))$$

# What is Covariance Matrix?

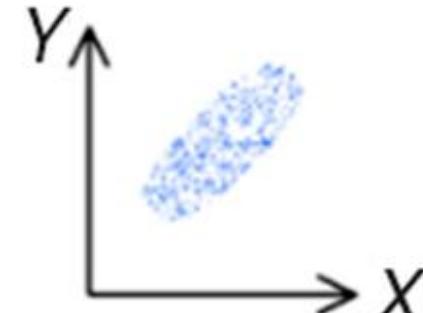
- Here are some plots that highlight how the covariance between two variables could look like in different directions.



$$\text{cov}(X, Y) < 0$$



$$\text{cov}(X, Y) = 0$$



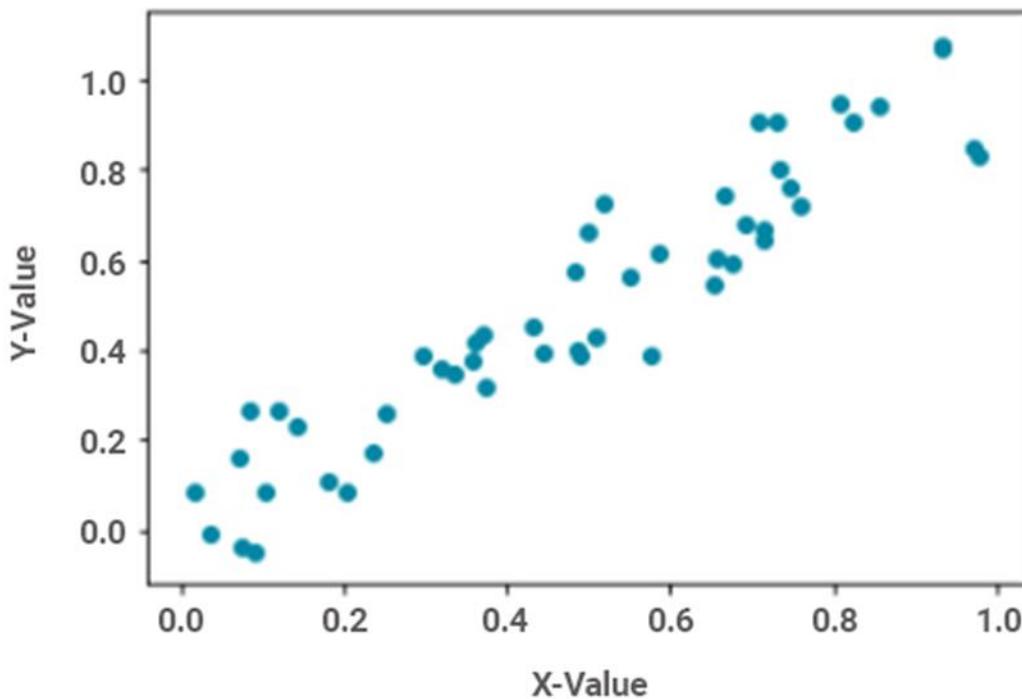
$$\text{cov}(X, Y) > 0$$

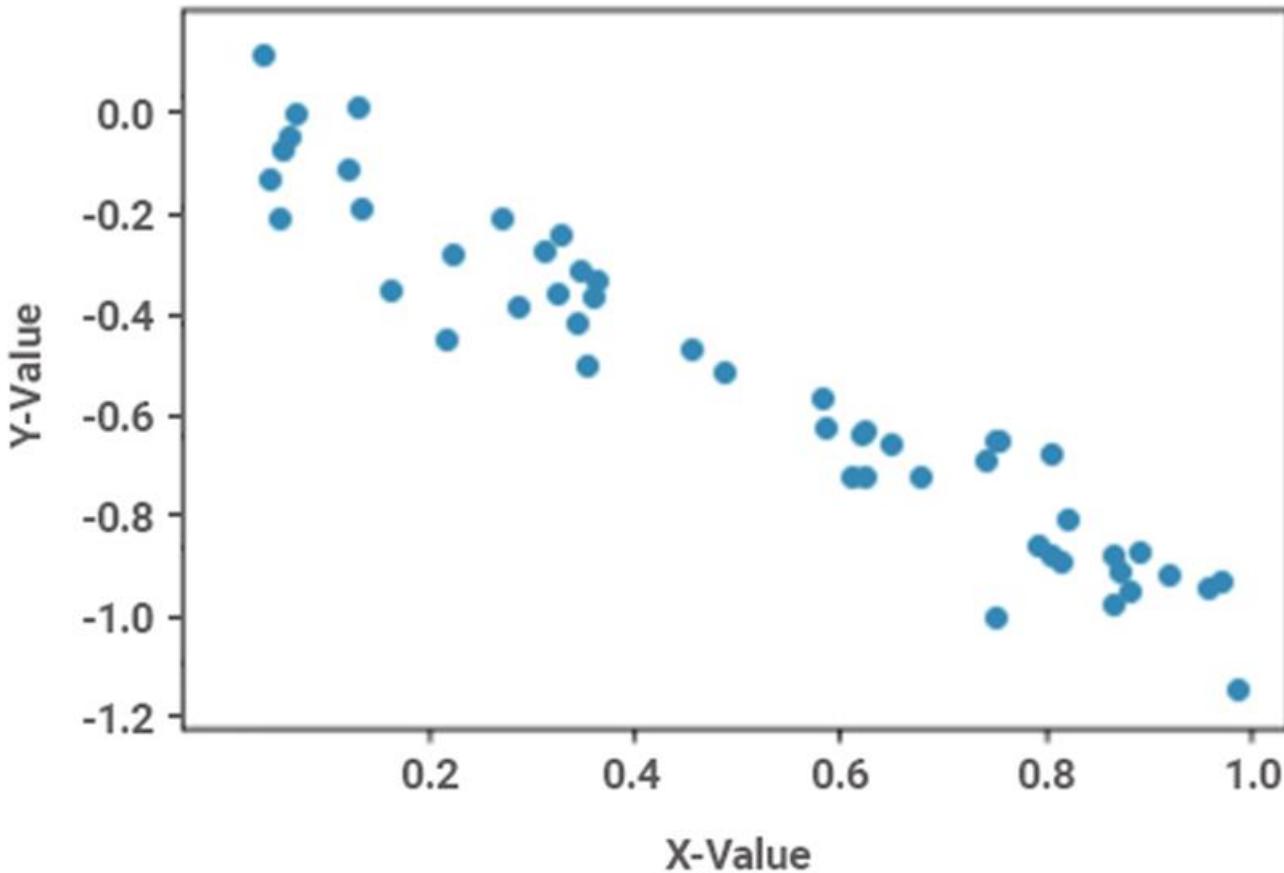
# What is a Correlation Matrix?

- A correlation matrix is used to study the strength of a relationship between two variables.
- It not only shows the direction of the relationship, but also shows how strong the relationship is.
- The correlation formula can be represented as:

$$\text{COR}(X, Y) = \frac{\text{COV}(X, Y)}{\sqrt{\text{VAR}(X)\text{VAR}(Y)}}$$

# What is Covariance Matrix?





# "Complete Lab 2"

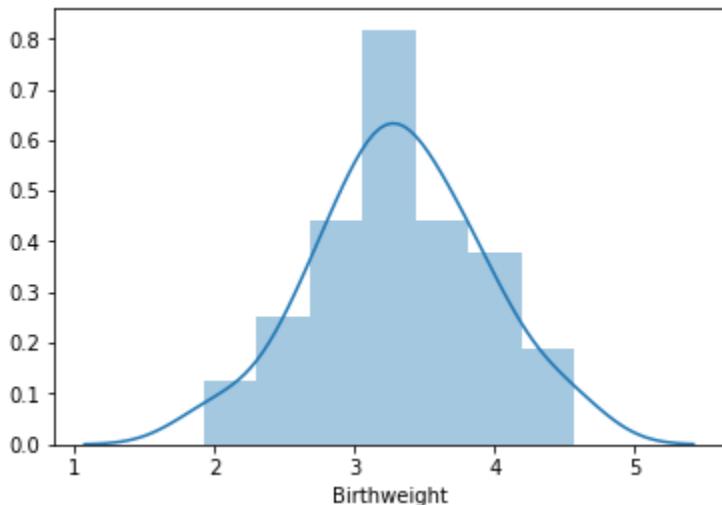
# "Complete Case Study"

# Descriptive Statistics

	count	mean	std	min	25%	50%	75%	max
<b>ID</b>	42.0	894.071429	467.616186	27.00	537.25	821.000	1269.5000	1764.00
<b>Length</b>	42.0	51.333333	2.935624	43.00	50.00	52.000	53.0000	58.00
<b>Birthweight</b>	42.0	3.312857	0.603895	1.92	2.94	3.295	3.6475	4.57
<b>Headcirc</b>	42.0	34.595238	2.399792	30.00	33.00	34.000	36.0000	39.00
<b>Gestation</b>	42.0	39.190476	2.643336	33.00	38.00	39.500	41.0000	45.00
<b>smoker</b>	42.0	0.523810	0.505487	0.00	0.00	1.000	1.0000	1.00
<b>mage</b>	42.0	25.547619	5.666342	18.00	20.25	24.000	29.0000	41.00
<b>mnocig</b>	42.0	9.428571	12.511737	0.00	0.00	4.500	15.7500	50.00
<b>mheight</b>	42.0	164.452381	6.504041	149.00	161.00	164.500	169.5000	181.00
<b>mppwt</b>	42.0	57.500000	7.198408	45.00	52.25	57.000	62.0000	78.00
<b>fage</b>	42.0	28.904762	6.863866	19.00	23.00	29.500	32.0000	46.00
<b>fedyrs</b>	42.0	13.666667	2.160247	10.00	12.00	14.000	16.0000	16.00
<b>fnocig</b>	42.0	17.190476	17.308165	0.00	0.00	18.500	25.0000	50.00
<b>fheight</b>	42.0	180.500000	6.978189	169.00	175.25	180.500	184.7500	200.00
<b>lowbwt</b>	42.0	0.142857	0.354169	0.00	0.00	0.000	0.0000	1.00
<b>mage35</b>	42.0	0.095238	0.297102	0.00	0.00	0.000	0.0000	1.00

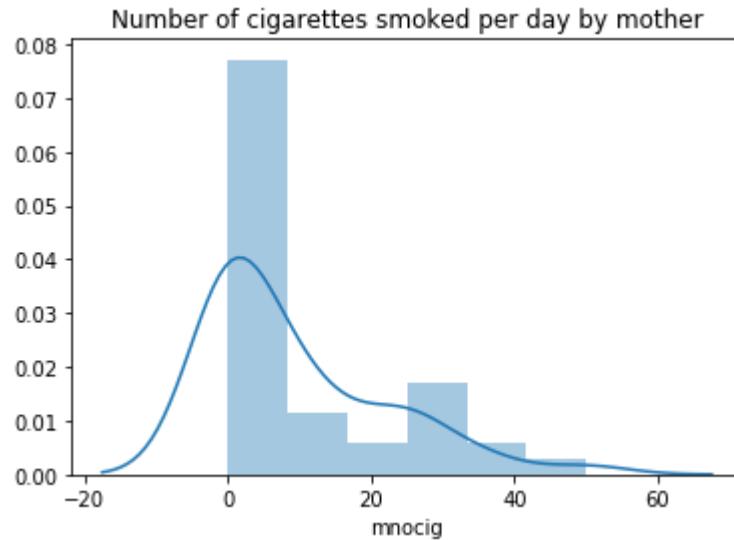
- We can analyze the distribution of the birth weight variable.

```
#plot distributions of birth weight  
sns.distplot(birth_weight['Birthweight'], label="Birth Weight")
```

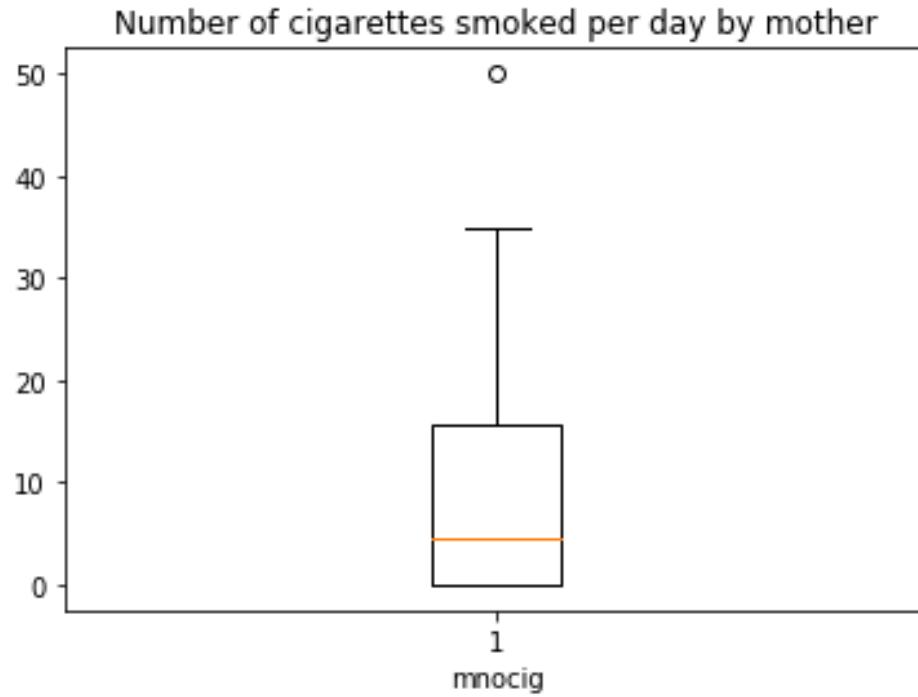


- Let's analyze the distribution of "mnocig" (Number of cigarettes smoked per day by mother) variable

```
#plot distribution of Number of cigarettes smoked per day by mother  
sns.distplot(birth_weight['mnocig'])  
plt.title("Number of cigarettes smoked per day by mother")
```



# Descriptive Statistics



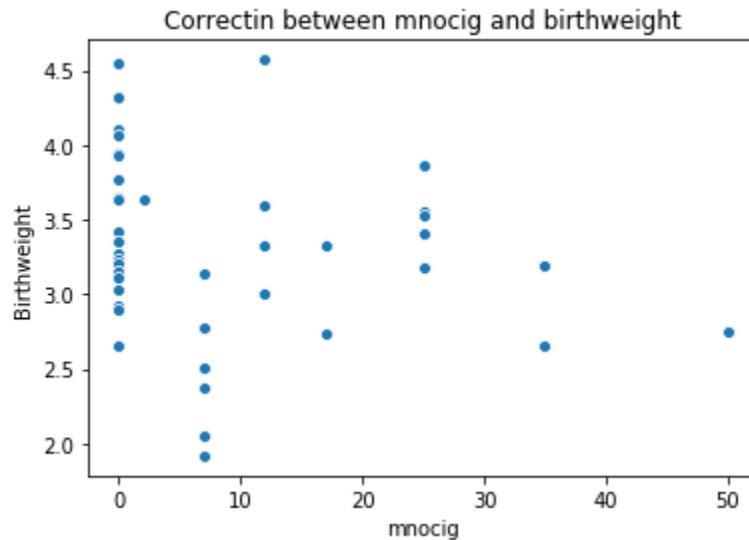
# Descriptive Statistics

```
mnocig_46 = np.percentile(birth_weight['mnocig'], 46)
mnocig_75 = np.percentile(birth_weight['mnocig'], 75)
mnocig_90 = np.percentile(birth_weight['mnocig'], 90)

print("46th percentile: ", round(mnocig_46, 0))
print("75th percentile: ", round(mnocig_75, 0))
print("90th percentile: ", round(mnocig_90, 0))
```

```
46th percentile:  0.0
75th percentile: 16.0
90th percentile: 25.0
```

```
#Correlation between birthweight and mnocig  
sns.scatterplot(birth_weight['mnocig'], birth_weight['Birthweight'])  
plt.title("Correctin between mnocig and birthweight")
```



```
#correlation value  
birth_weight['Birthweight'].corr(birth_weight['mnocig'])
```

-0.1523351844506074



# SUMMARY

- Statistics deals with collecting, interpreting, and drawing a conclusion from the data.
- Data is measured on different scales like nominal, ordinal, interval and ratio.
- Descriptive statistics aims to summarize a sample data with a single value with the help of mean, median and mode.

# "Complete Assessment"

# DAY 2



# 3: Probability Distributions



# Probability Distributions

- Probability implies ‘likelihood’ or ‘chance’.
- When an event is certain to happen then the probability of occurrence of that event is 1 and when it is certain that the event cannot happen then the probability of that event is 0.

# Assigning Probabilities

- **Classical method** – A prior or Theoretical Probability can be determined prior to conducting any experiment.

$$P(E) = \frac{\text{# of outcomes in which the event occurs}}{\text{total possible # of outcomes}}$$

# Assigning Probabilities

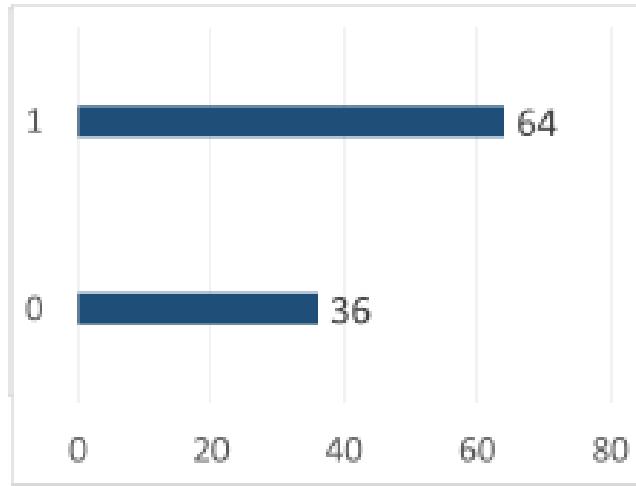
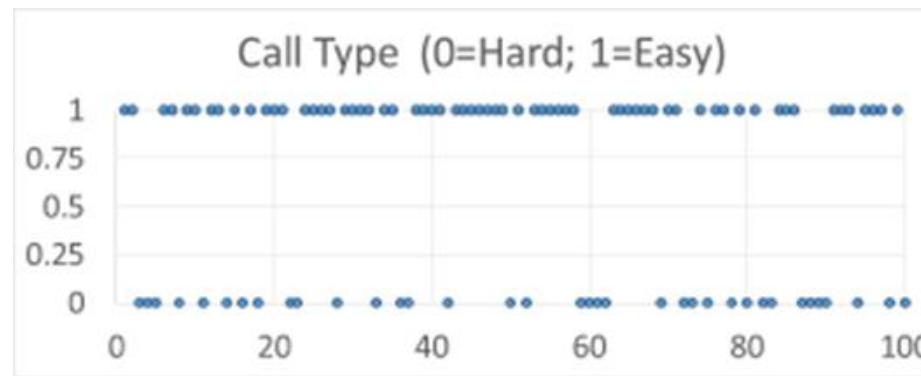
Experiment: Tossing of a fair dice



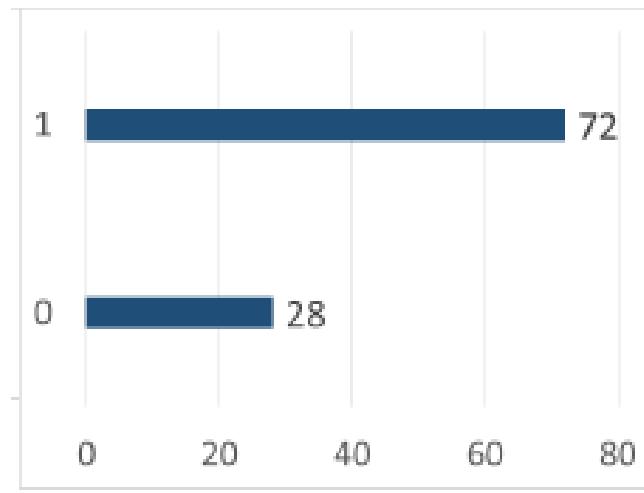
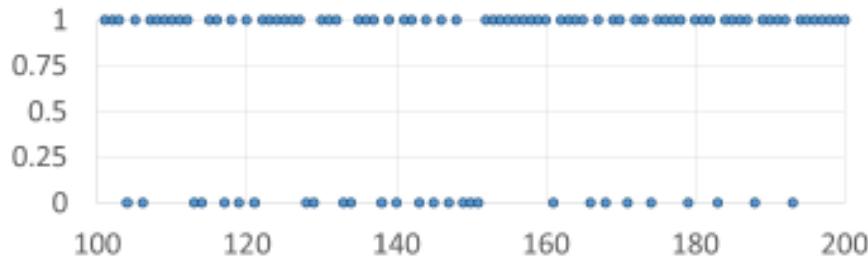
# Assigning Probabilities

- Empirical Method – A posteriori or Frequentist Probability can be determined post conducting a thought experiment.

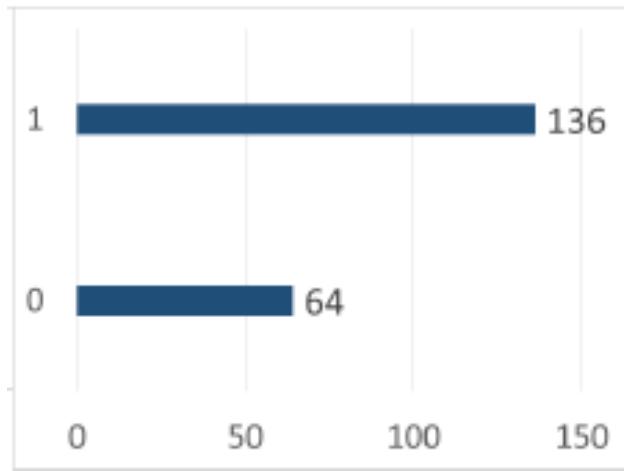
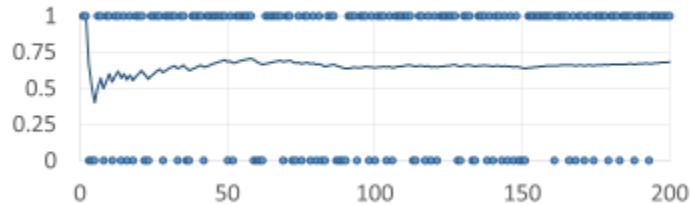
$$P(E) = \frac{\text{\# of times an event occurred}}{\text{total \# of opportunities for the event to have occurred}}$$



Call Type (0=Hard; 1=Easy)



Call Type (0=Hard; 1=Easy)



$$P(\text{easy}) = 0.7$$

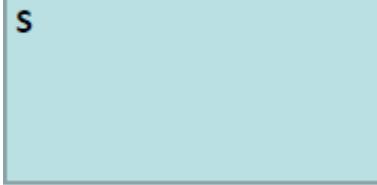
# Probability Terminology

- Sample Space – Set of all possible outcomes, denoted S.

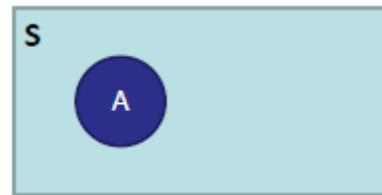
Example: After 2 coin tosses, the set of all possible outcomes are {HH, HT, TH, TT}

- Event – A subset of the samples space.  
An Event of interest might be – HH

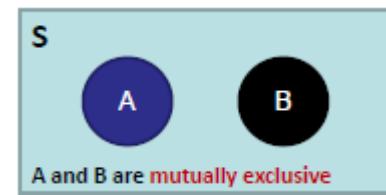
# Probability – Rules



$$P(S) = 1$$



$$0 \leq P(A) \leq 1$$



$$P(A \text{ or } B) = P(A) + P(B)$$

# Mutually Exclusive

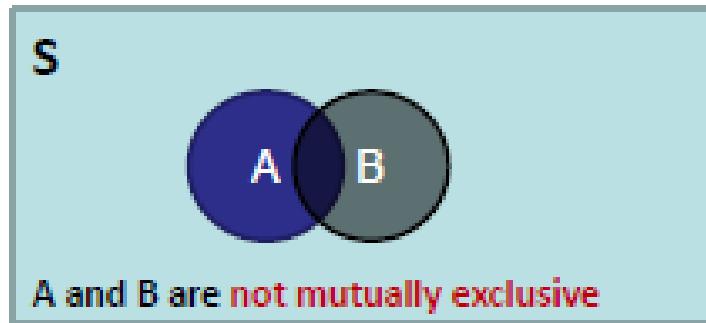
When two events (call them “A” and “B”) are Mutually Exclusive than it is impossible for them to happen together.

- If A and B are mutually exclusive  
 $P(A \text{ and } B) = 0$
- But the probability of A or B is the sum of the individual probabilities.

$$P(A \text{ or } B) = P(A) + P(B)$$

# Mutually Exclusive

- When we combine those two events:  $P(\text{King or Queen}) = (1/13) + (1/13) = 2/13$



$$P(A \text{ or } B) = P(A) + P(B) - P(A \text{ and } B)$$

# Mutually Non-Exclusive Events

- Two events A and B are said to be mutually non-exclusive events if both the events A and B have at least one common outcome between them.

# Probability – Types

- Contingency table summarizing 2 variables, Loan Default and Age:

		Age			
		Young	Middle-aged	Old	Total
Loan Default	No	10,503	27,368	259	38,130
	Yes	3,586	4,851	120	8,557
	Total	14,089	32,219	379	46,687

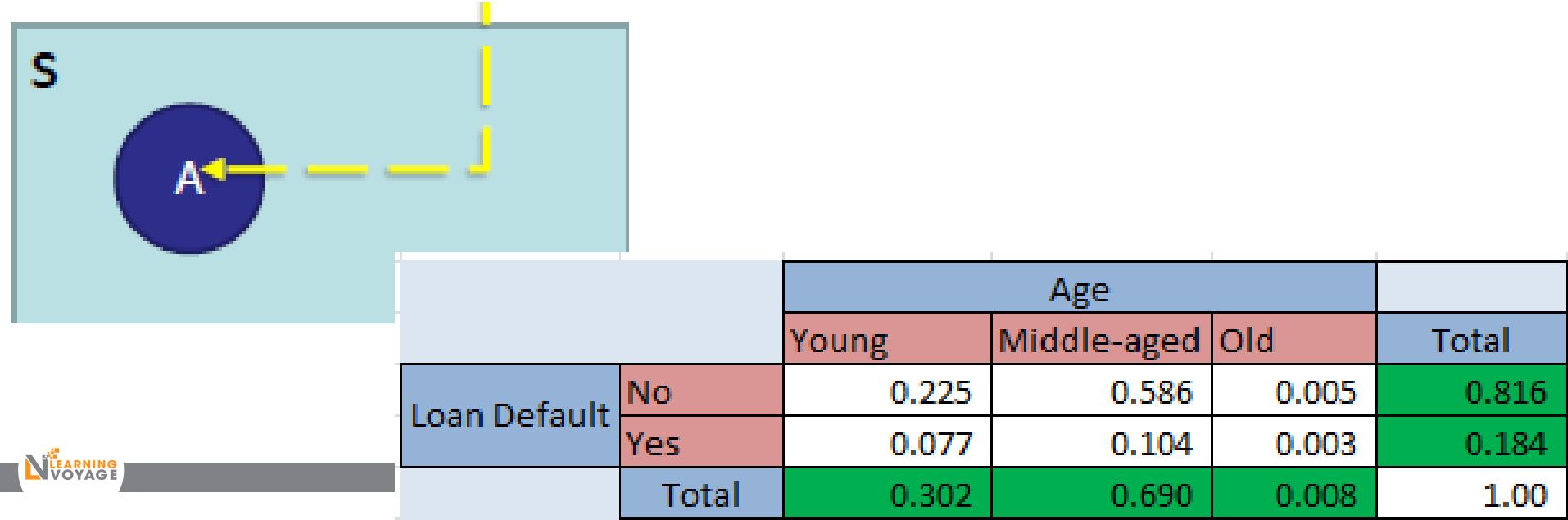
# Probability – Types

- Convert it into probabilities:

		Age			Total
		Young	Middle-aged	Old	
Loan Default	No	0.225	0.586	0.005	0.816
	Yes	0.077	0.104	0.003	0.184
	Total	0.302	0.690	0.008	1.00

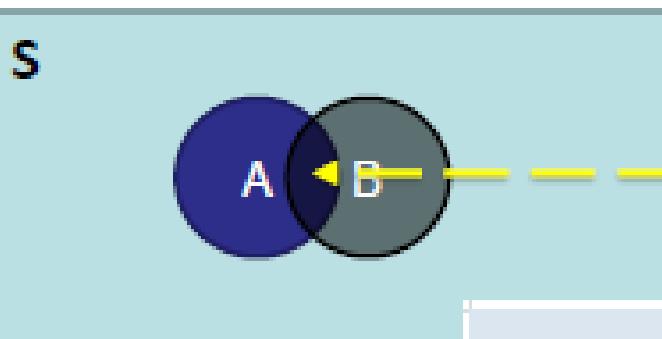
# Probability – Types

- Marginal Probability: Probability describing a single attribute



# Probability – Types

Joint Probability: Probability describing a combination of attributes



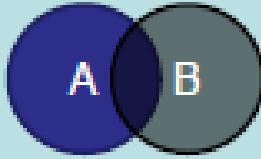
		Age			
		Young	Middle-aged	Old	Total
Loan Default	No	0.225	0.586	0.005	0.816
	Yes	0.077	0.104	0.003	0.184
	Total	0.302	0.690	0.008	1.00

**LEARNING VOYAGE**

# Probability – Types

- Union Probability: Probability describing a new set that contains all of the elements that are in at least one of the two sets.

S

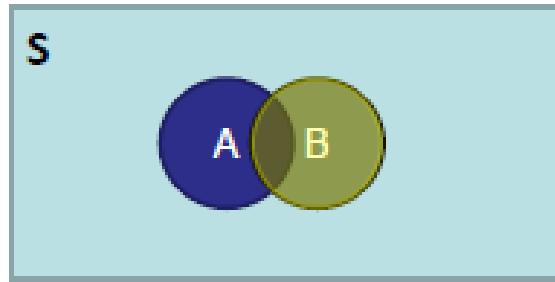


Loan Default	Age			Total
	Young	Middle-aged	Old	
No	0.225	0.586	0.005	0.816
Yes	0.077	0.104	0.003	0.184
Total	0.302	0.690	0.008	1.00

# Conditional Probability

The probability of an event (A), given that another event (B) has already occurred.

- The sample space is restricted to a single row or column. This makes the rest of the sample irrelevant.



# Example:

What is the probability that a person will not default on the loan payment given he/she is middle-age?

		Age			
		Young	Middle-aged	Old	Total
Loan Default	No	0.225	0.586	0.005	0.816
	Yes	0.077	0.104	0.003	0.184
	Total	0.302	0.69	0.008	1.00

# Probability – Types

- Note that this is the ratio of Joint Probability to Marginal Probability, i.e.

$$P(A|B) = \frac{P(A \text{ and } B)}{P(B)}$$

# Probability – Types

- Equating, we get

$$P(A/B) * P(B) = P(A) * P(B/A)$$

$$\therefore P(A|B) = \frac{P(A) * P(B|A)}{P(B)}$$

- Now, given that the probability that someone defaults on a loan is 0.184, find the probability that an older person defaults on the loan.
- Older people make up only 0.8% of the clientele.  $P(\text{Yes/Old}) = ?$

$$P(\text{Yes/Old}) = (P(\text{Yes}) * P(\text{Old/Yes}))/P(\text{Old})$$

		Age			
		Young	Middle-aged	Old	Total
Loan Default	No	10,503	27,368	259	38,130
	Yes	3,586	4,851	120	8,557
	Total	14,089	32,219	379	46,687

# Histogram:

A series of contiguous rectangles that represent the frequency of data in the given class intervals.

## How many class intervals?

- Rule of thumb: 5-15 (not too many and not too few)

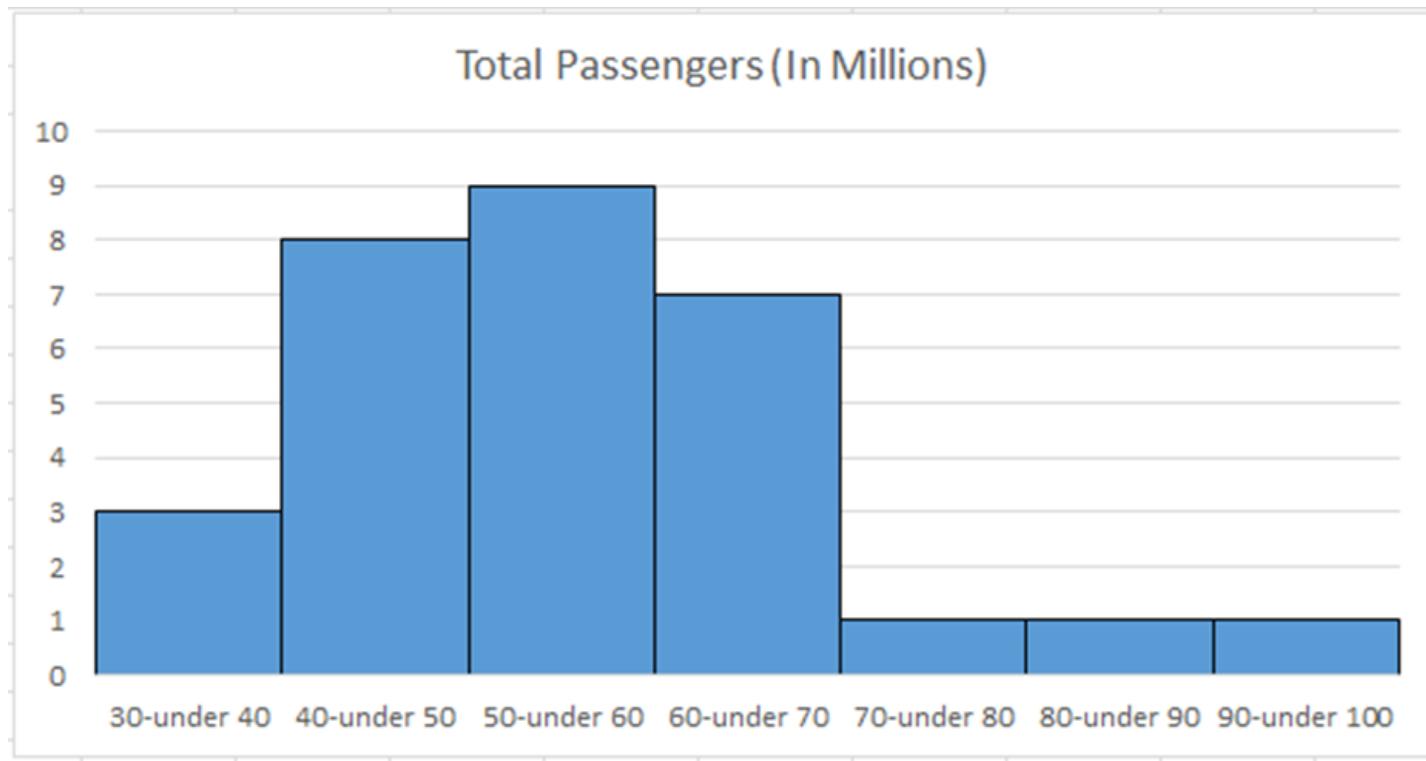
The Freedman-diaconis rule:

$$\text{No. of bins} = \frac{(max - min)}{2 * IQR * n^{-\frac{1}{3}}},$$

# Histogram – Excel

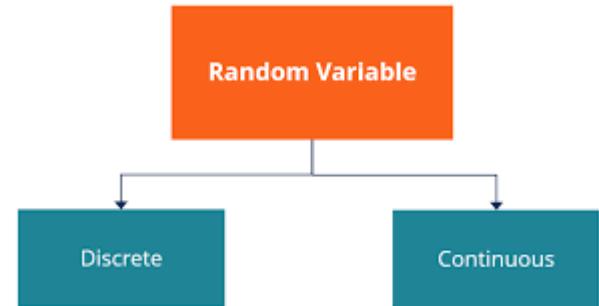
Passenger Traffic 2013 FINAL (Annual)			
Last Update: 22 December 2014			
Passenger Traffic			
Total passengers enplaned and deplaned, passengers in transit counted once			
Rank	City (Airport)	Passengers 2013	% Change
1	ATLANTA GA, US (ATL)	94,431,224	-1.1
2	BEIJING, CN (PEK)	83,712,355	2.2
3	LONDON, GB (LHR)	72,368,061	3.3
4	TOKYO, JP (HND)	68,906,509	3.2
5	CHICAGO IL, US (ORD)	66,777,161	0.2
6	LOS ANGELES CA, US (LAX)	66,667,619	4.7
7	DUBAI, AE (DXB)	66,431,533	15.2
8	PARIS, FR (CDG)	62,052,917	0.7
9	DALLAS/FORT WORTH TX, US (DFW)	60,470,507	3.2
10	JAKARTA, ID (CGK)	60,137,347	4.1
11	HONG KONG, HK (HKG)	59,588,081	6.3
12	FRANKFURT, DE (FRA)	58,036,948	0.9
13	SINGAPORE, SG (SIN)	53,726,087	5
14	AMSTERDAM, NL (AMS)	52,569,200	3
15	DENVER CO, US (DEN)	52,556,359	-1.1
16	GUANGZHOU, CN (CAN)	52,450,262	8.6
17	BANGKOK, TH (BKK)	51,363,451	-3.1
18	ISTANBUL, TR (IST)	51,304,654	13.7
19	NEW YORK NY, US (JFK)	50,423,765	2.3

# Histogram – Excel



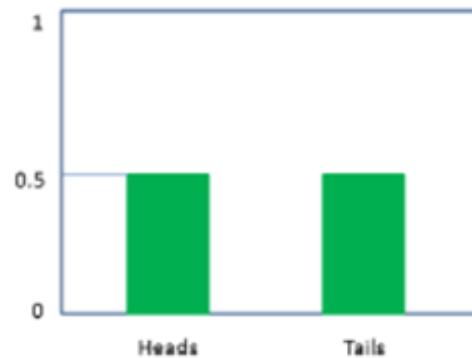
# Random Variable

- A Random Variable is a set of possible values from a random experiment.



# Discrete Random Variable

- The discrete random variable is a variable that may take on only a countable number of distinct values.



Countable

# Probability Distributions

## Types of Discrete Probability Distributions

- Bernoulli Distribution.
- Binomial Distribution.
- Poisson Distribution.

# Bernoulli distribution

- A Bernoulli distribution is a discrete probability distribution for a Bernoulli trial – a random experiment that has only two outcomes (usually called “Success” or “Failure”).

# Binomial Distribution

- A binomial distribution is the probability of the “success” or “failure” outcome of an experiment or survey that is repeated multiple times.
- A binomial distribution is the probability of a “success” or “failure” outcome in an experiment or survey that is repeated multiple times.

Notation:  $X \sim \text{Bio}(n, P)$

n: number of times the experiment runs

p: probability of one specific outcome

# Probability Mass Function:

$$b(x; n, P) = {}_n C_x \cdot P^x \cdot (1 - P)^{n - x}$$

Where:

- $b$  = binomial probability.
- $x$  = total number of “success”.
- $P$  = probability of success on an individual trial.
- $n$  = number of trials.

# Mean and Variance of Binomial distribution:

$$\begin{aligned}E(X) &= np \\Var(X) &= npq\end{aligned}$$

Criteria – Binomial distribution must meet the following three criteria:

- The number of trials is fixed.
- Each trial is independent of others.
- The probability of “success” (trial, head, fail or pass) is the same from one trial to another.

# Poisson distribution

- The Poisson distribution is the discrete probability distribution of the number of events occurring in a given time period, provided that the events occur at a constant mean rate and are independent of the time since last event.

# Probability Mass Function:

$$P(X) = \frac{e^{-\mu} \mu^x}{x!}$$

Where:

- The symbol “!” is a factorial.
- M (The expected number of occurrences) is sometimes written as  $\lambda$ . It is sometimes called the event rate or rate parameter.

# "Complete Lab 3"

# "Complete Case Study"

# "Complete Programming Assignment "

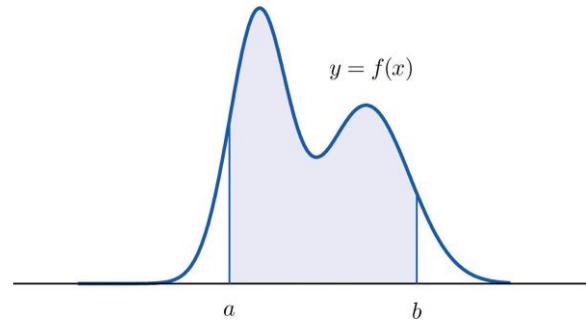
# 4: Inferential Statistics and Python



# Continuous Random Variable

- A continuous random variable is a random variable, in which the data can take infinitely many values.
- Continuous random variables usually are the measurements.

$P(a < X < b) = \text{area of shaded region}$



# Continuous Random Variable

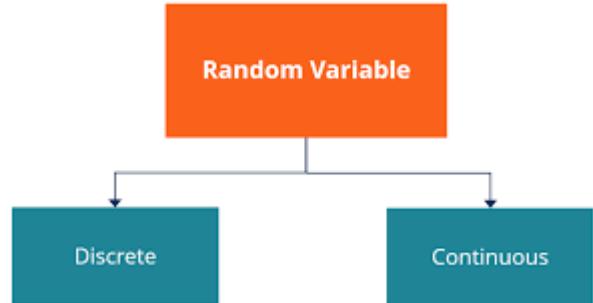
The probability function of the continuous random variable is probability density function (PDF) and represented by the area under a curve.

It is formulated as follows:

- The formula for  $E(X)$  is  $E(X) = \int_{-\infty}^{\infty} xf(x)dx$ .
- The formula for  $E(X^2)$  is  $E(X^2) = \int_{-\infty}^{\infty} x^2 f(x)dx$ .
- Formula for  $P(a < X < b)$  is  $P(a < X < b) = \int_a^b f(x)dx$ .
- Formula for cumulative distribution function is  $F(X) = \int_{-\infty}^x f(z)dz$ .

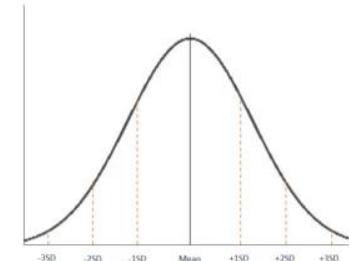
# Types of Continuous Random Variables

- Normal Distribution.
- Standard Normal Distribution.
- Uniform Distribution / Rectangular Distribution.



# Normal Distribution.

- Normal distribution also known as the Gaussian distribution, is the probability distribution which is symmetric about the mean
- Which shows that the data close to the mean are more frequent in occurrence than the data further away from the mean

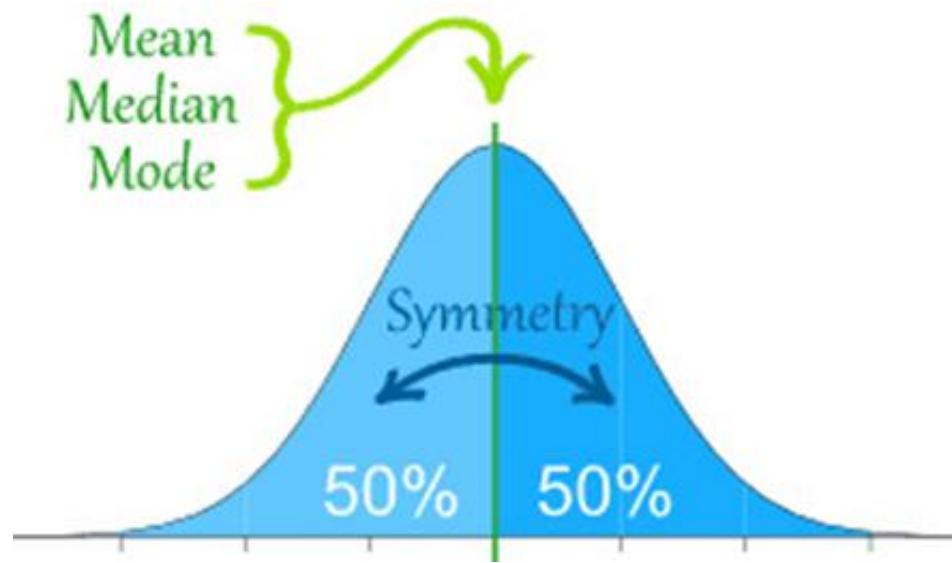


# Normal Distribution.

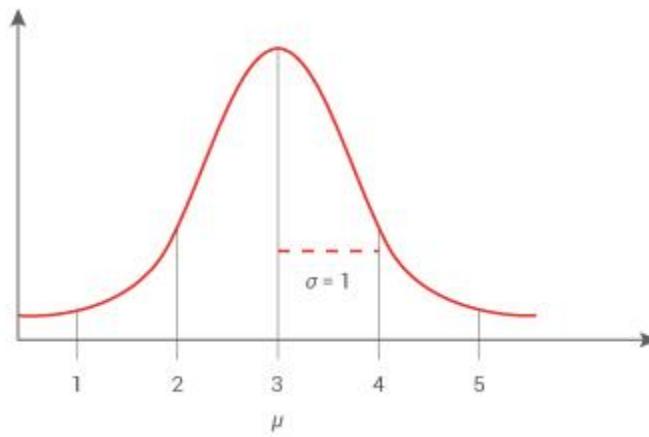
- Probability Density Function

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2}$$

# Normal Distribution.

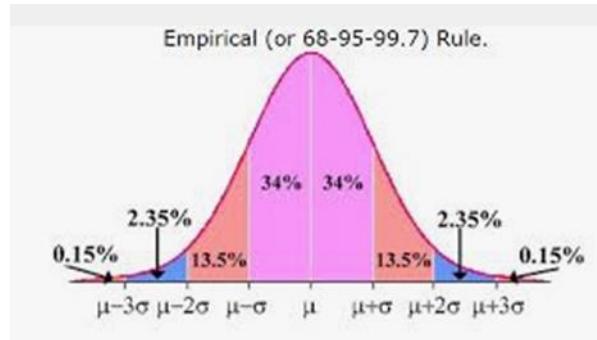


# Normal Distribution.

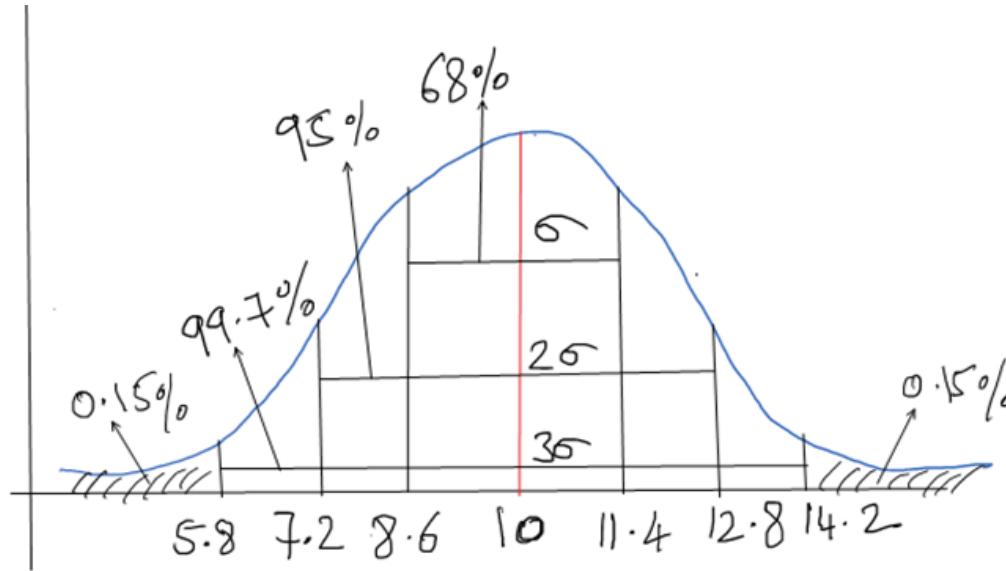


# Empirical Formula

- The empirical rule states that for the Normal Distribution, nearly all observed data sets will fall within 3 standard deviations.
- The empirical rule is also known as Three-Sigma rule or 68-95-99.7 rule.



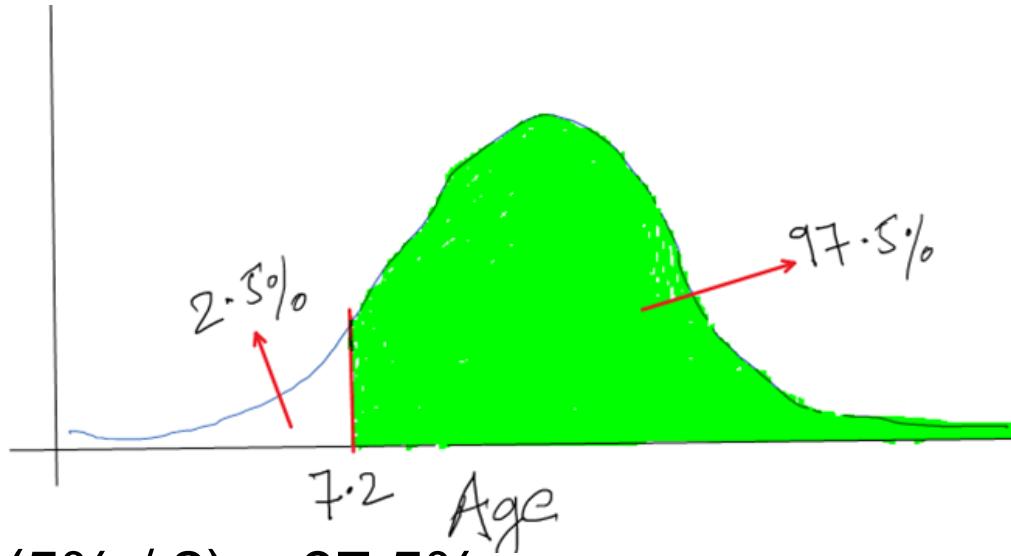
# Empirical Formula



Age

$\Delta r^+$

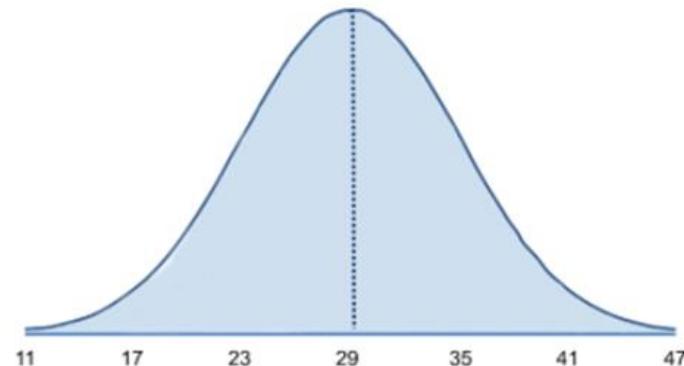
# Empirical Formula



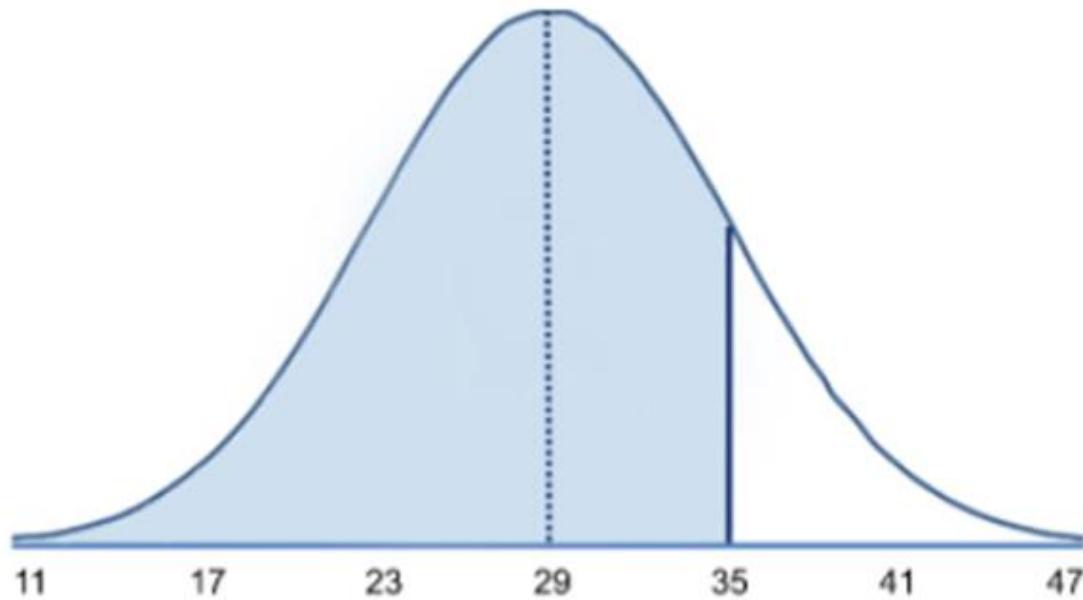
- $95\% + (5\% / 2) = 97.5\%$
- Thus, the probability of living more than 7.2 years is 97.5%.

# Empirical Formula

- Example 2: Consider the weight values of a population of 15 years old males where the weight is normally distributed and has a mean value =29 and a standard deviation = 6.



# Empirical Formula



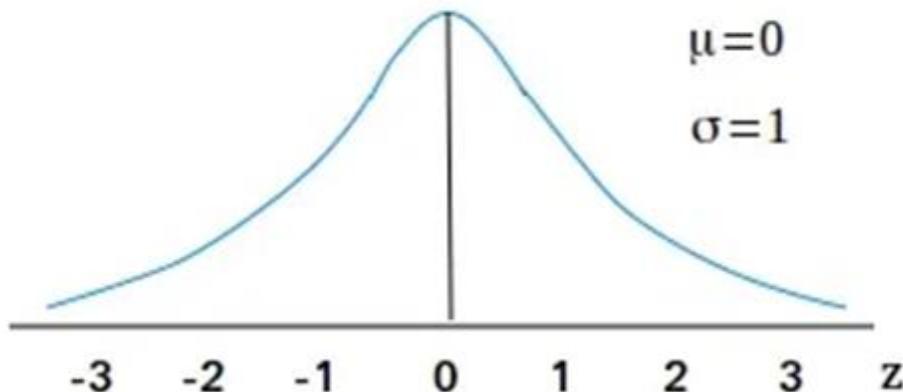
# Standard Normal Distribution

- The standard normal distribution is a special case of the normal distribution.
- It is the distribution that occurs when a normal random variable has the mean value of zero and standard deviation of one

$$Z = \frac{X - \mu}{\sigma}$$

# Standard Normal Distribution

The standard normal distribution curve



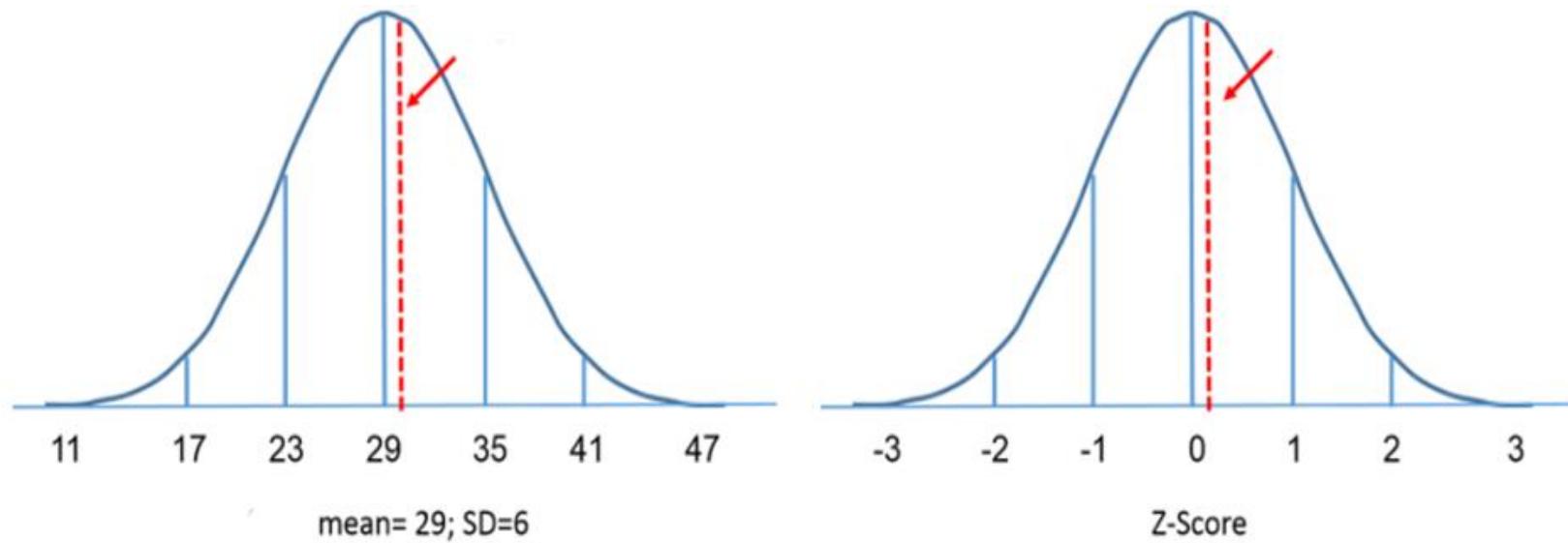
# Standard Normal Distribution

Solution: To compute  $P(X < 30)$  we convert the  $X=30$  to its corresponding Z score (this is called standardizing):

$$Z = 30 - 29 / 6 \Rightarrow 1/6$$

$$Z = 0.1667$$

# Standard Normal Distribution

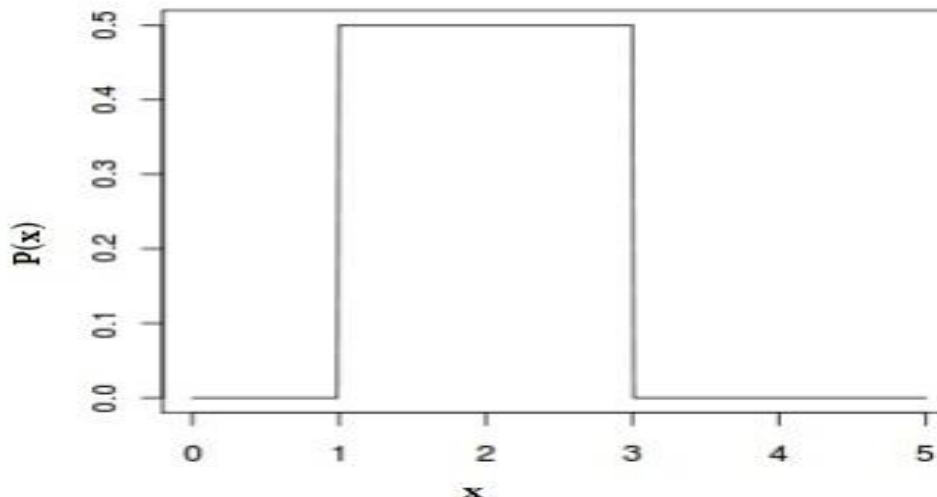


# Standard Normal Distribution

<i>z</i>	.00	.01	.02	.03	.04	.05	.06	.07	.08	.09
0.0	.5000	.5040	.5080	.5120	.5160	.5199	.5239	.5279	.5319	.5359
0.1	.5398	.5438	.5478	.5517	.5557	.5596	.5636	.5675	.5714	.5753
0.2	.5793	.5832	.5871	.5910	.5948	.5987	.6026	.6064	.6103	.6141
0.3	.6179	.6217	.6255	.6293	.6331	.6368	.6406	.6443	.6480	.6517
0.4	.6554	.6591	.6628	.6664	.6700	.6736	.6772	.6808	.6844	.6879
0.5	.6915	.6950	.6985	.7019	.7054	.7088	.7123	.7157	.7190	.7224
0.6	.7257	.7291	.7324	.7357	.7389	.7422	.7454	.7486	.7517	.7549
0.7	.7580	.7611	.7642	.7673	.7704	.7734	.7764	.7794	.7823	.7852
0.8	.7881	.7910	.7939	.7967	.7995	.8023	.8051	.8078	.8106	.8133
0.9	.8159	.8186	.8212	.8238	.8264	.8289	.8315	.8340	.8365	.8389
1.0	.8413	.8438	.8461	.8485	.8508	.8531	.8554	.8577	.8599	.8621

# Uniform Distribution / Rectangular Distribution:

- A uniform distribution, also called a rectangular distribution, is a probability distribution that has a constant probability

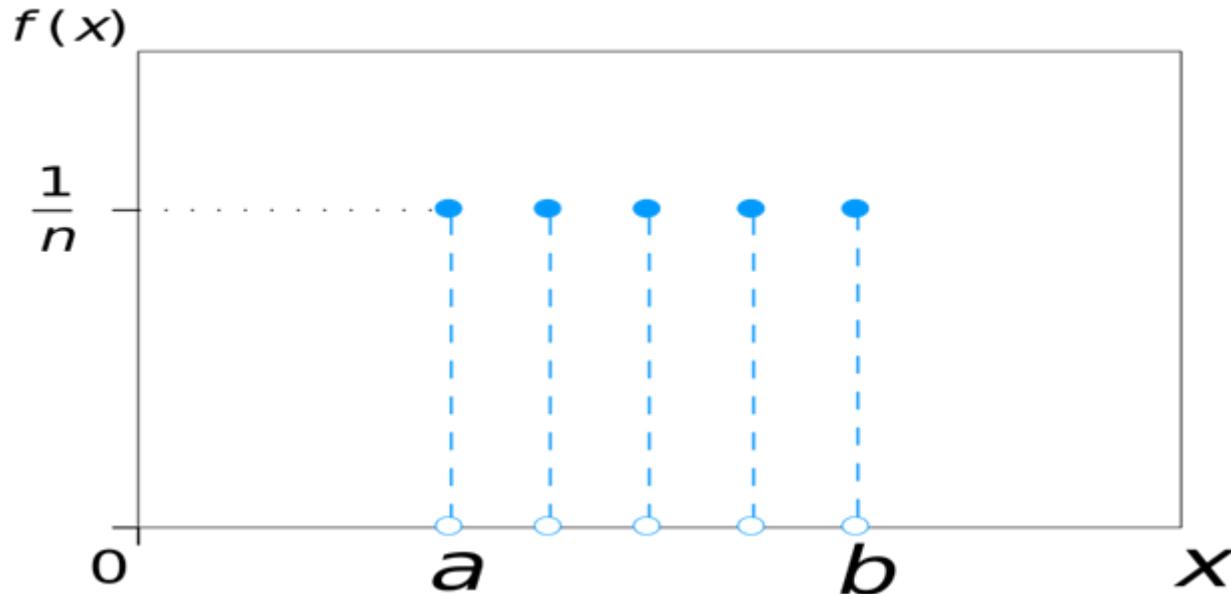


# Uniform Distribution / Rectangular Distribution:

- Probability density function:

$$f(x) = \frac{1}{b - a}$$

# Uniform Distribution / Rectangular Distribution:



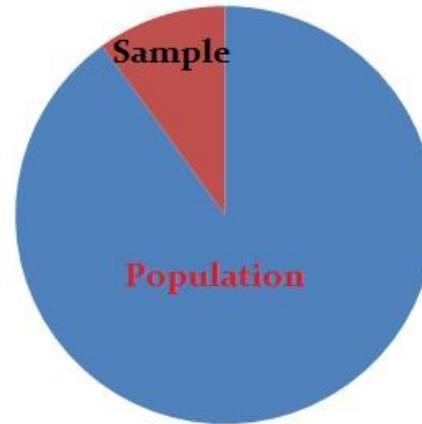
# Continuity Correction Factor:

- A continuity correction factor is applied when you use a continuous probability distribution to approximate a discrete probability distribution.
- For example: when you want to use the normal distribution to approximate a binomial distribution, following conditions needs to be satisfied.

$$(n * p) \text{ and } (n * q) \geq 5$$

# Inferential statistics

- Inferential statistics use a random sample of data from a population to make inference about the whole population.



# Central Limit Theorem

- The central limit theorem states that the sampling distribution of the sample means approaches a normal distribution as the sample size gets larger, irrespective of the shape of the population distribution.

# Expectation and Standard deviation of $X$ bar

- Mean of all sample means of size  $n$  is the mean of the population.

$$E(\bar{X}) = \mu$$

- Standard deviation tells us how far away from the population mean the sample mean is likely to be and is called the Standard Error of the Mean, and is represented as follows

$$\text{Standard Error of the Mean} = \frac{\sigma}{\sqrt{n}}$$

If  $X \sim N(\mu, \sigma^2)$ , then  $\bar{X} \sim N(\mu, \sigma^2/n)$

# Point Estimate

- A point estimate of a population parameter is a single value of a statistic.
- Example:  $\bar{X}$  is a point estimate of the population mean  $\mu$ . Similarly, the population proportion “ $p$ ” is a point estimate of the population proportion “ $P$ ” .

# Confidence Intervals of Mean, Proportion, & Variance

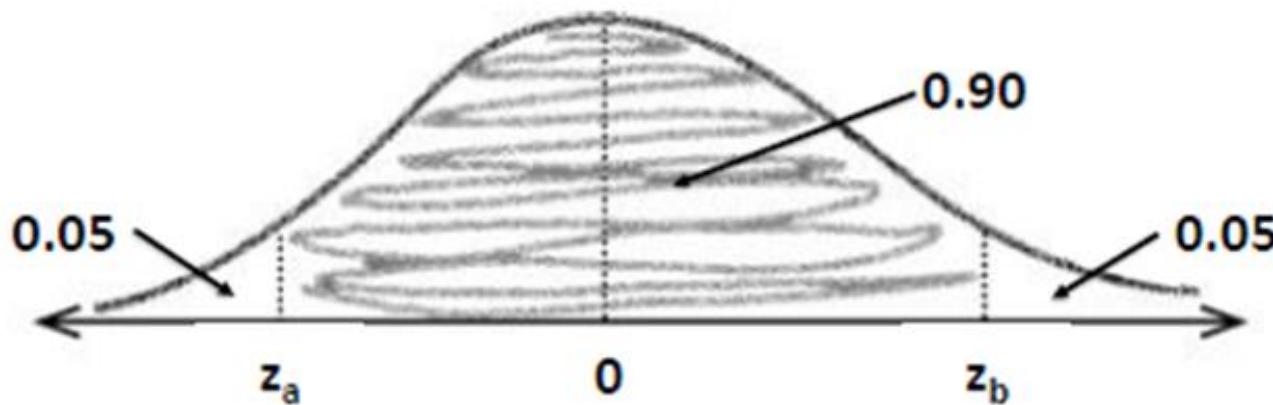
Mean:

$$CI = \bar{x} \pm z \frac{s}{\sqrt{n}}$$

# Confidence Intervals of Mean, Proportion, & Variance

Sample mean  $\pm$  Margin of Error

Find  $Z_a$  and  $Z_b$  where  $P(Z_a < Z < Z_b) = 0.90$



<i>z</i>	.00	.01	.02	.03	.04	.05	.06	.07	.08	.09
0.0	.5000	.4960	.4920	.4880	.4840	.4801	.4761	.4721	.4681	.4641
0.1	.4602	.4562	.4522	.4483	.4443	.4404	.4364	.4325	.4286	.4247
0.2	.4207	.4168	.4129	.4090	.4052	.4013	.3974	.3936	.3897	.3859
0.3	.3821	.3783	.3745	.3707	.3669	.3632	.3594	.3557	.3520	.3483
0.4	.3446	.3409	.3372	.3336	.3300	.3264	.3228	.3192	.3156	.3121
0.5	.3085	.3050	.3015	.2981	.2946	.2912	.2877	.2843	.2810	.2776
0.6	.2743	.2709	.2676	.2643	.2611	.2578	.2546	.2514	.2483	.2451
0.7	.2420	.2389	.2358	.2327	.2296	.2266	.2236	.2206	.2177	.2148
0.8	.2119	.2090	.2061	.2033	.2005	.1977	.1949	.1922	.1894	.1867
0.9	.1841	.1814	.1788	.1762	.1736	.1711	.1685	.1660	.1635	.1611
1.0	.1587	.1562	.1539	.1515	.1492	.1469	.1446	.1423	.1401	.1379
1.1	.1357	.1335	.1314	.1292	.1271	.1251	.1230	.1210	.1190	.1170
1.2	.1151	.1131	.1112	.1093	.1075	.1056	.1038	.1020	.1003	.0985
1.3	.0968	.0951	.0934	.0918	.0901	.0885	.0869	.0853	.0838	.0823
1.4	.0808	.0793	.0778	.0764	.0749	.0735	.0721	.0708	.0694	.0681
1.5	.0668	.0655	.0643	.0630	.0618	.0606	.0594	.0582	.0571	.0559
1.6	.0548	.0537	.0526	.0516	.0505	.0495	.0485	.0475	.0465	.0455

# Proportion:

$$\mu_{\hat{p}} = p$$

$$\sigma_{\hat{p}} = \sqrt{\frac{p(1-p)}{n}}$$

$$CI = \hat{p} \pm z_{\alpha/2} \sqrt{\frac{\hat{p}(1-\hat{p})}{n}}$$

# Proportion:

- Example: In a poll by FOX News conducted between November 1 and 3, 2016, a survey of 1107 randomly sampled likely voters predicted that 45% would vote for Hillary Clinton:

Fox News Poll			
	Nov	Oct 22 - 25	Oct 15 - 17
Clinton	45%	44%	45%
Trump	43%	41%	39%
November 1 - 3, 2016			
Likely Voters +/- 3% Pts			

# t-Distribution:

- If the sample size is small (<30), the variance of the population is not adequately captured by the variance of the sample.
- In such cases, instead of z-distribution, t-distribution is used.
- It is also the appropriate distribution type to use when the population variance is not known.

$$t \text{ statistic (or } t \text{ score}), t = \frac{(\bar{x} - \mu)}{\frac{s}{\sqrt{n}}}$$

# t-Distribution:

- Confidence Interval to Estimate  $\mu$ :

$$\bar{x} - t_{n-1, \frac{\alpha}{2}} \frac{s}{\sqrt{n}} \leq \mu \leq \bar{x} + t_{n-1, \frac{\alpha}{2}} \frac{s}{\sqrt{n}}$$

# t-Distribution:

- Data are as follows (in mg):

98.6	102.1	100.7	102	97
103.4	98.9	101.6	102.9	105.2

# t-Distribution:

DF	Table of Critical Values for T Two Tailed Significance					
	0.2	0.1	0.05	0.01	0.005	0.001
2	1.89	2.92	4.30	9.92	14.09	31.60
3	1.64	2.35	3.18	5.84	7.45	12.92
4	1.53	2.13	2.78	4.60	5.60	8.61
5	1.48	2.02	2.57	4.03	4.77	6.87
6	1.44	1.94	2.45	3.71	4.32	5.96
7	1.41	1.89	2.36	3.50	4.03	5.41
8	1.40	1.86	2.31	3.36	3.83	5.04
9	1.38	1.83	2.26	3.25	3.69	4.78
10	1.37	1.81	2.23	3.17	3.58	4.59
...	...	...	...	...	...	...

# t-Distribution:

$$\bar{x} - t_{n-1, \frac{\alpha}{2}} \frac{s}{\sqrt{n}} \leq \mu \leq \bar{x} + t_{n-1, \frac{\alpha}{2}} \frac{s}{\sqrt{n}}$$

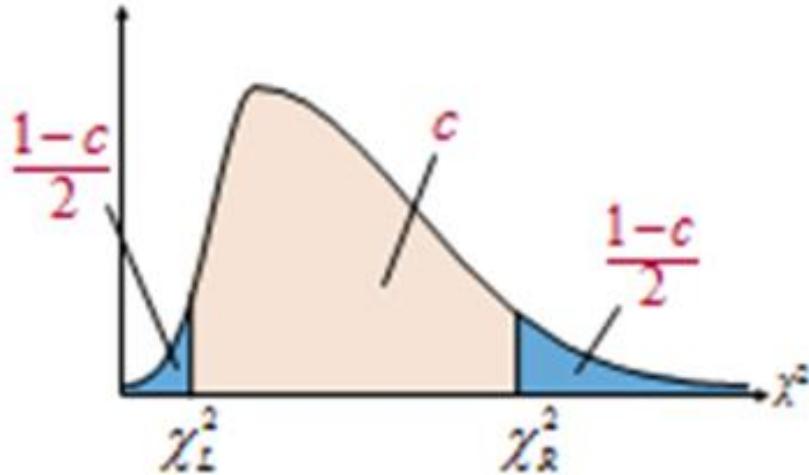
# Chi Square Distribution for Variance:

- You can use the Chi Square distribution to construct the confidence interval for the variance and standard deviation.

Formula:

$$C.I. = \frac{(n - 1) S^2}{\chi^2_{\alpha/2, n-1}} \leq \sigma^2 \leq \frac{(n - 1) S^2}{\chi^2_{1-\alpha/2, n-1}}$$

# Chi Square Distribution for Variance:



The area between the left and right critical values is  $c$ .

df	0.995	0.99	0.975	0.95	0.90	0.10	0.05	0.025	0.01	0.005
1	---	---	0.001	0.004	0.016	2.706	3.841	5.024	6.635	7.879
2	0.010	0.020	0.051	0.103	0.211	4.605	5.991	7.378	9.210	10.597
3	0.072	0.115	0.216	0.352	0.584	6.251	7.815	9.348	11.345	12.838
4	0.207	0.297	0.484	0.711	1.064	7.779	9.488	11.143	13.277	14.860
5	0.412	0.554	0.831	1.145	1.610	9.236	11.070	12.833	15.086	16.750
6	0.676	0.872	1.237	1.635	2.204	10.645	12.592	14.449	16.812	18.548
7	0.989	1.239	1.690	2.167	2.833	12.017	14.067	16.013	18.475	20.278
8	1.344	1.646	2.180	2.733	3.490	13.362	15.507	17.535	20.090	21.955
9	1.735	2.088	2.700	3.325	4.168	14.684	16.919	19.023	21.666	23.589
10	2.156	2.558	3.247	3.940	4.865	15.987	18.307	20.483	23.209	25.188
11	2.603	3.053	3.816	4.575	5.578	17.275	19.675	21.920	24.725	26.757
12	3.074	3.571	4.404	5.226	6.304	18.549	21.026	23.337	26.217	28.300
13	3.565	4.107	5.009	5.892	7.042	19.812	22.362	24.736	27.688	29.819
14	4.075	4.660	5.629	6.571	7.790	21.064	23.685	26.119	29.141	31.319
15	4.601	5.229	6.262	7.261	8.547	22.307	24.996	27.488	30.578	32.801
16	5.142	5.812	6.908	7.962	9.312	23.542	26.296	28.845	32.000	34.267
17	5.697	6.408	7.564	8.672	10.085	24.769	27.587	30.191	33.409	35.718
18	6.265	7.015	8.231	9.390	10.865	25.989	28.869	31.526	34.805	37.156
19	6.844	7.633	8.907	10.117	11.651	27.204	30.144	32.852	36.191	38.582
20	7.434	8.260	9.591	10.851	12.443	28.412	31.410	34.170	37.566	39.997
21	8.034	8.897	10.283	11.591	13.240	29.615	32.671	35.479	38.932	41.401
22	8.643	9.542	10.982	12.338	14.041	30.813	33.924	36.781	40.289	42.796
23	9.260	10.196	11.689	13.091	14.848	32.007	35.172	38.076	41.638	44.181
24	9.886	10.856	12.401	13.848	15.659	33.196	36.415	39.364	42.980	45.559
25	10.520	11.524	13.120	14.611	16.473	34.382	37.652	40.646	44.314	46.928
26	11.160	12.198	13.844	15.379	17.292	35.563	38.885	41.923	45.642	48.290
27	11.808	12.879	14.573	16.151	18.114	36.741	40.113	43.195	46.963	49.645
28	12.461	13.565	15.308	16.928	18.939	37.916	41.337	44.461	48.278	50.993
29	13.121	14.256	16.047	17.708	19.768	39.087	42.557	45.722	49.588	52.336

For Mean

with known variance &  $n \geq 30$

with unknown variance or  $n < 30$

For Variance

For Proportion

Use Z Dstn  
Confidence Interval

Use t Dstn  
Confidence Interval

Use the  $\chi^2$  Dstn  
Confidence Interval

Use Z Dstn Confidence  
Interval if  $n \times \hat{p}$  and  
 $n \times \hat{q}$  are each  $\geq 5$

$$C.I. = \bar{X} \pm Z_{\frac{\alpha}{2}} \frac{\sigma}{\sqrt{n}}$$

$$C.I. = \bar{X} \pm t_{\frac{\alpha}{2}} \frac{s}{\sqrt{n}}$$

$$C.I. = \frac{(n-1) s^2}{\chi^2_{\frac{\alpha}{2}, n-1}} \leq \sigma^2 \leq \frac{(n-1) s^2}{\chi^2_{1-\frac{\alpha}{2}, n-1}}$$

$$C.I. = \hat{p} \pm Z_{\frac{\alpha}{2}} \sqrt{\frac{\hat{p}\hat{q}}{n}}$$

# Test of Hypothesis

- Hypothesis testing is a statistical method for making the statistical decisions about the hypothesis using experimental data.
- Hypothesis Testing is basically makes an assumption about the population parameter and uses a given sample to test whether or not the statistical claims are likely to be true or not.

# Test of Hypothesis

Normal distribution is a good approximation,

$$\text{Standard Error} = s / \sqrt{n} = 1.0 / \sqrt{40}$$

$$\text{Standard Error} = 0.158$$

$$X \sim N(0.7, 0.158^2) = N(0.7, 0.025)$$

$$Z = \frac{\bar{x} - \mu}{\frac{\sigma}{\sqrt{n}}}$$

$$Z = (0.55 - 0.7) / 0.158$$

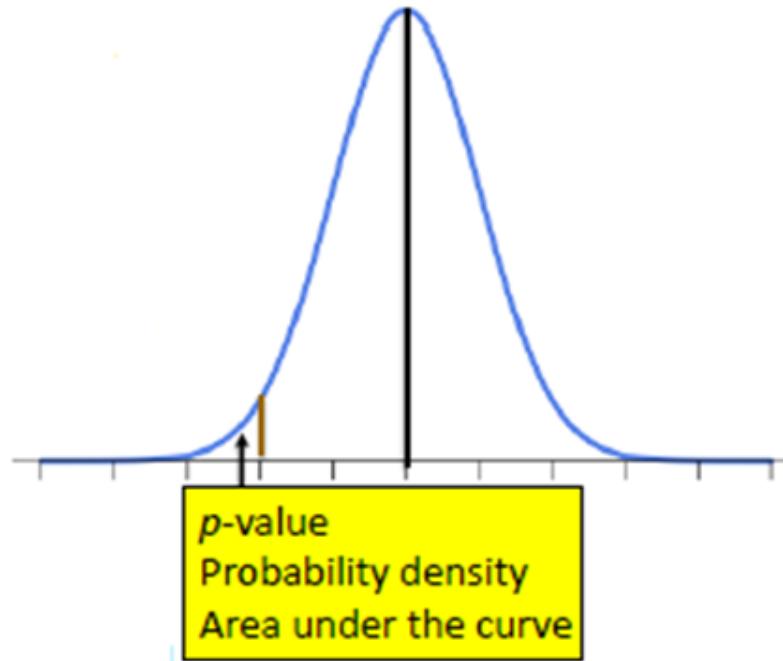
$$Z = 0.94$$

# Step 4: Determine the critical region.

- If  $X$  represents the mean score of the sample, the critical region is defined as  $P(X < c) < \alpha$  where  $\alpha=5\%$



# Test of Hypothesis



# Test of Hypothesis

- If  $P(X \leq 0.55) < 0.05$  (Significance Level), it indicates that 0.55 is inside the critical region, and hence  $H_0$  can be rejected.

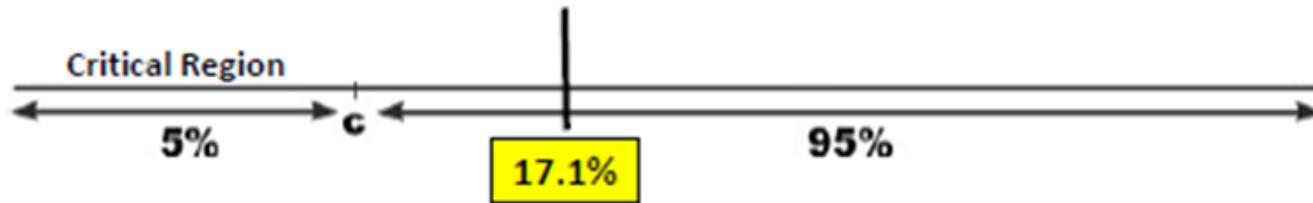
Given that  $Z = -0.94$ ,  $P(X \leq 0.55) = 0.171$

```
> pnorm(0.55,0.7,1/sqrt(40))
[1] 0.1713909
```

- Thus, there is a 17% probability of finding a mean score of 5.5/10 or less.

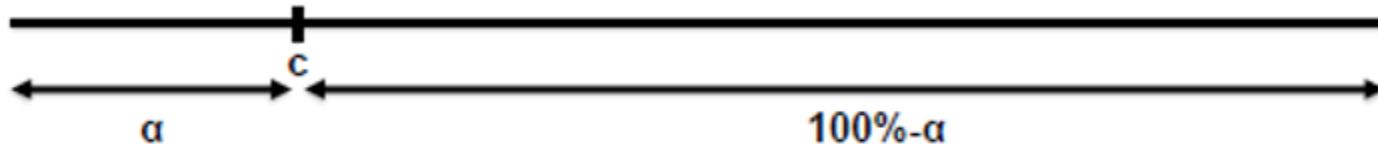
# Test of Hypothesis

- Step 6: Is the sample result in the critical region?



# Critical Region up Close

If  $H_1$  includes a “<” sign (lesser than), then the lower tail is utilized.

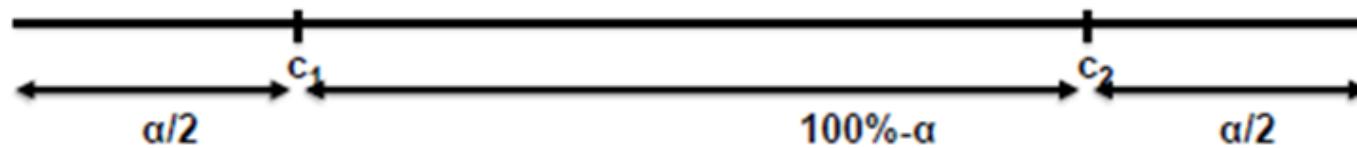


If  $H_1$  includes a “>” sign (greater than), then the upper tail is utilized.

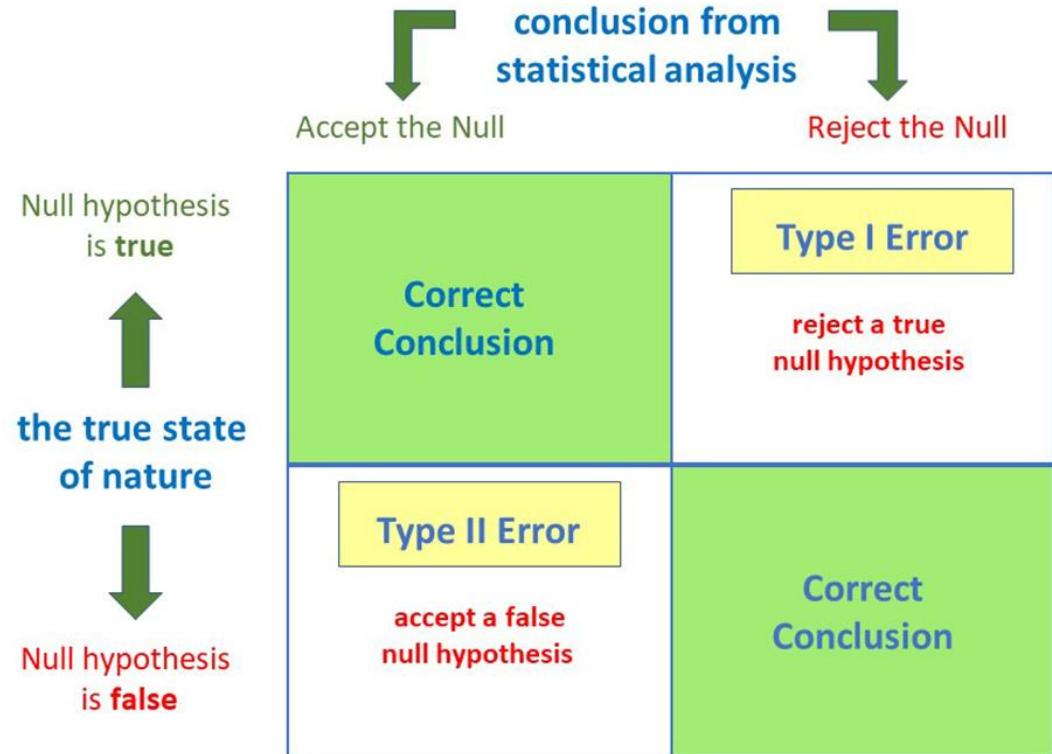


# Two-tailed tests:

- Critical region is split over both ends. Both ends contain  $\alpha/2$ , making a total of  $\alpha$ .
- If  $H_1$  includes a “ $\neq$ ” sign (Not equal to), then the two-tailed test is used, since we then look for a change in parameter, rather than an increase or a decrease.

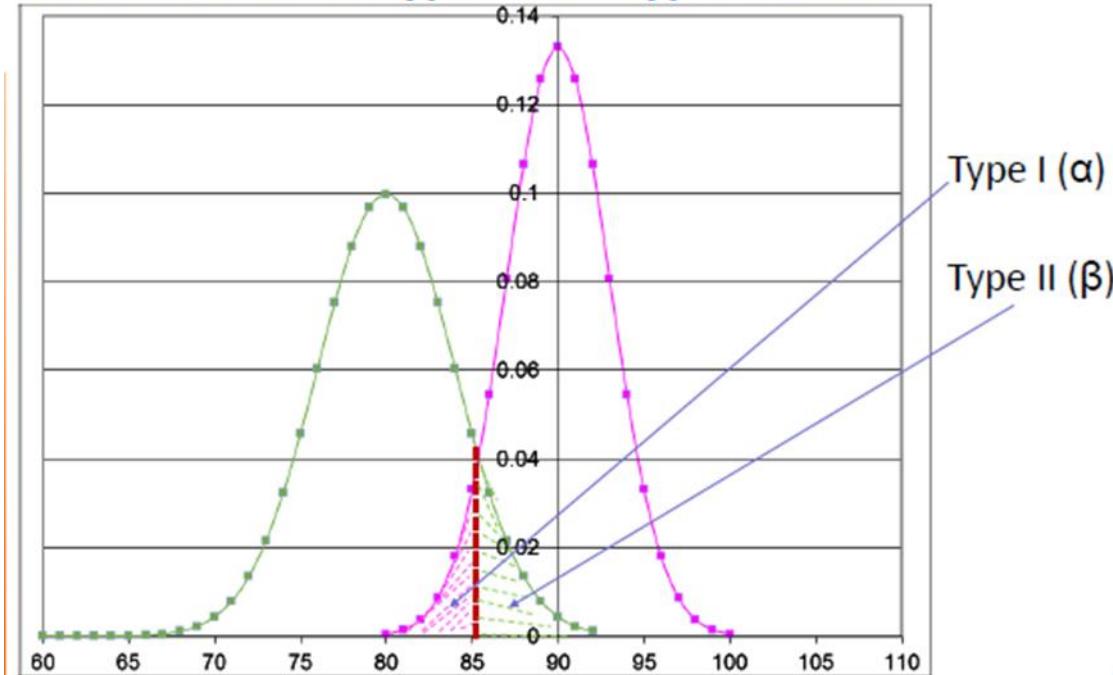


# Types of Errors



# Critical Region up Close

Probabilities of Type I and Type II Errors



# Common Test Statistics for Inferential Techniques

- Inferential techniques (Confidence Intervals and Hypothesis Testing) most commonly use 4 test statistical methods:
    - $z$
    - $t$
    - $\chi^2$  (Chi-squared)
    - F
- Closely related to Sampling Distribution of Means
- Closely related to Sampling Distribution of Variances
- Derived from Normal Distribution

# Two-Sample t-test for Paired Data:

- To study if their mean values are the same – we can create a new data set from the difference of the individual data points.

$$X_{\text{new}} = X_1 - X_2$$

- Subsequently, we can look at how far away from zero is the mean  $E(X_{\text{new}})$

$$t = \frac{\overline{X}_{\text{new}} - 0}{SE(\overline{X}_{\text{new}})}$$

Time to Solve the puzzle (Minutes)			
Volume	After	Before	A-B
1	63	55	8
2	54	62	-8
3	79	108	-29
4	68	77	-9
5	87	83	4
6	84	78	6
7	92	79	13
8	57	94	-37
9	66	69	-3
10	53	66	-13
11	76	72	4
12	63	77	-14
Total	842	920	-78
Mean	70.17	76.67	-6.50

# Two-Sample t-test for Paired Data:

Mean of the difference,

$$\bar{d} = -6.5$$

Standard Deviation of the differences,

$$s_d = 15.1$$

# Critical Region up Close

Standard Error of the mean,

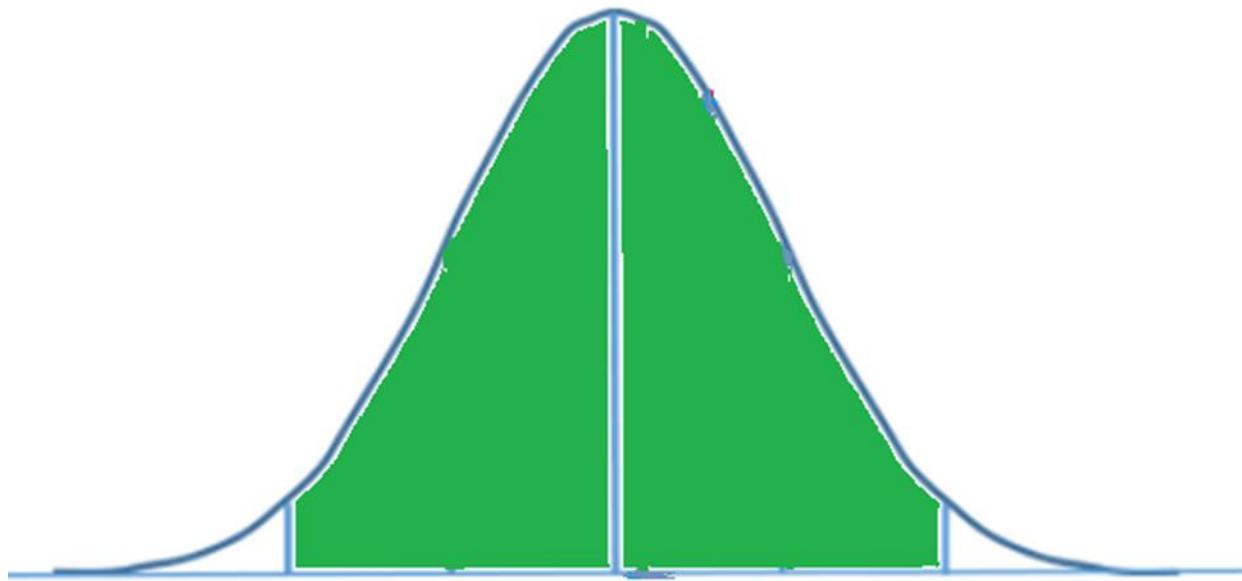
$$SE(\bar{d}) = \frac{s_d}{\sqrt{n}} = 4.37$$

$$t = \frac{\bar{d}}{SE(\bar{d})}$$

$$t = -6.5 / 4.37$$

$$\mathbf{t = -1.487}$$

# Critical Region up Close



# Two-Sample t-Test for Unpaired Data:

$$\mu_{\bar{x}_1 - \bar{x}_2} = \mu_1 - \mu_2$$

$$\sigma_{\bar{x}_1 - \bar{x}_2} = \sqrt{\frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}}$$

$$Z = \frac{\text{observed difference} - \text{expected difference}}{\text{SE of the difference}} = \frac{(\bar{x}_1 - \bar{x}_2) - (\mu_1 - \mu_2)}{\sqrt{\frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}}}$$

# Two-Sample t-Test for Unpaired Data:

- This is the test statistic for a 2-sample z-test.

$$H_0: \mu_1 = \mu_2$$

$$H_1: \mu_1 \neq \mu_2$$

- t-test statistic,

$$t = \frac{(\bar{x}_1 - \bar{x}_2)}{\sqrt{\frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}}}$$

# Two-Sample t-Test for Unpaired Data:

$$s_p^2 = \frac{(n_1 - 1)s_1^2 + (n_2 - 1)s_2^2}{(n_1 - 1) + (n_2 - 1)}$$

$$t = \frac{(\bar{x}_1 - \bar{x}_2)}{s_p \sqrt{\frac{1}{n_1} + \frac{1}{n_2}}}$$

With  $(n_1 + n_2 - 2)$  degrees of freedom

Clearance of theophylline	
Control Group	Treated Group
0.81	1.15
1.06	1.28
0.43	1.00
0.54	0.95
0.68	1.06
0.56	1.15
0.45	0.72
0.88	0.79
0.73	0.67
0.43	1.21
0.46	0.92
0.43	0.67
0.37	0.76
0.73	0.82
0.93	0.82

It is a Two-tailed test.

$$S_p^2 = \frac{(n_1-1)s_1^2 + (n_2-1)s_2^2}{(n_1-1)+(n_2-1)}; t = \frac{(\bar{x}_1 - \bar{x}_2)}{s_p \sqrt{\frac{1}{n_1} + \frac{1}{n_2}}} \text{ with } (n_1 + n_2 - 2) \text{ df.}$$

$$S_p^2 = \frac{((15-1) * (0.0408)) + ((15-1) * (0.0467))}{(15 - 1) * (15 + 1)}$$

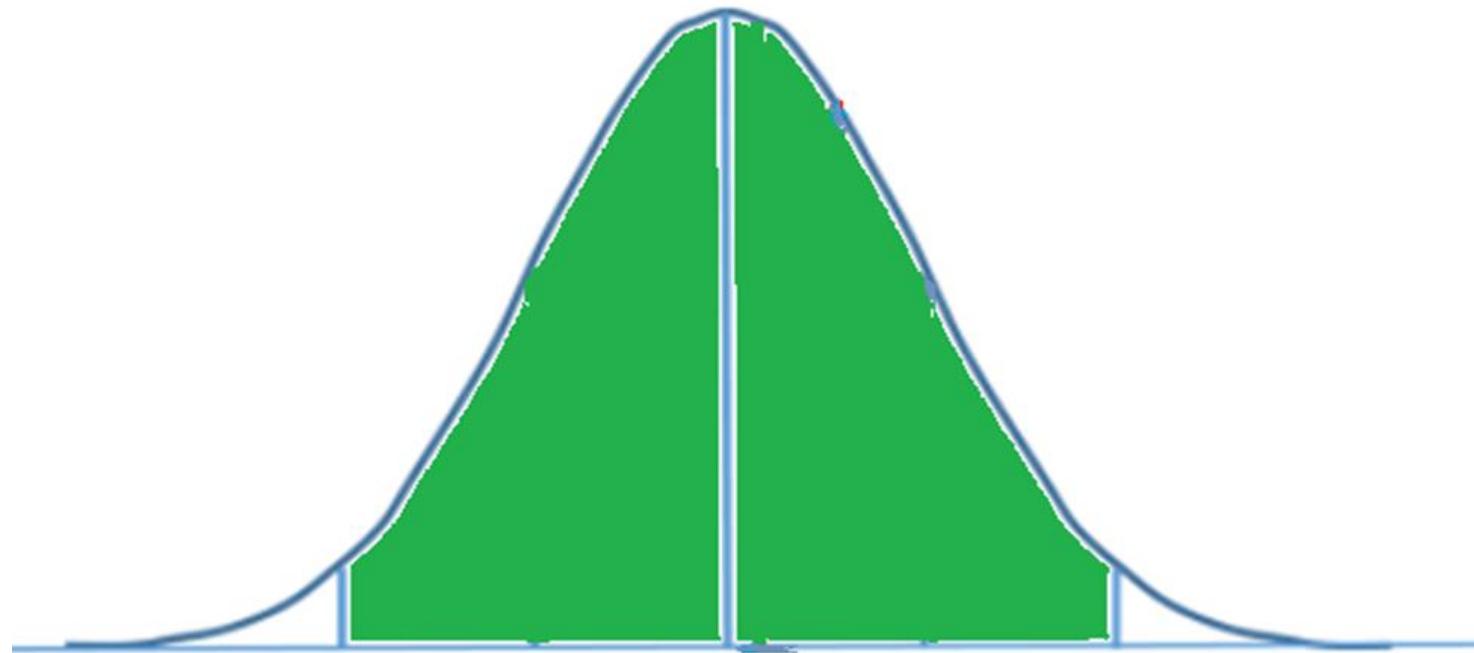
$$S_p^2 = 0.04375$$

$$S_p = 0.209$$

$$t = \frac{0.931 - 0.633}{0.209 * \sqrt{\frac{1}{15} + \frac{1}{15}}}$$

$$t = 3.91$$

# Two-Sample t-Test for Unpaired Data:



# Confidence Intervals

$$t = \frac{(\bar{x}_1 - \bar{x}_2) - (\mu_1 - \mu_2)}{s_p \sqrt{\frac{1}{n_1} + \frac{1}{n_2}}}$$

Rewriting

$$(\bar{x}_1 - \bar{x}_2) - ts_p \sqrt{\frac{1}{n_1} + \frac{1}{n_2}} \leq \mu_1 - \mu_2 \leq (\bar{x}_1 - \bar{x}_2) + ts_p \sqrt{\frac{1}{n_1} + \frac{1}{n_2}}$$

$$((0.298) - (2.048 * (0.0763))) \leq \mu_1 - \mu_2 \leq ((0.298) + (2.48 * (0.0763)))$$

$$95\% \text{ CI: } (0.142, 0.454)$$

## Welch's t-test using Welch-Satterthwaite equation to calculate the degrees of freedom

$$H_0: \mu_1 = \mu_2; H_1: \mu_1 \neq \mu_2; \text{Test statistic, } t = \frac{(\bar{x}_1 - \bar{x}_2)}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}}$$

When two standard deviations are not equal, we use the above formula.

In this case, the degree of freedom is calculated as:

$$v = \left[ \frac{\left( \frac{s_1^2}{n_1} + \frac{s_2^2}{n_2} \right)^2}{\frac{\left( \frac{s_1^2}{n_1} \right)^2}{n_1-1} + \frac{\left( \frac{s_2^2}{n_2} \right)^2}{n_2-1}} \right]$$

# "Complete Lab 4"

# "Complete Case Study"

**"Complete Assessment, Fill in  
the spaces & Programming  
Assignment "**

# DAY 3



# 5: Data Cleaning and Exploratory Data Analysis



# Data Cleaning and Data Wrangling

- Data cleaning or data cleansing is the process of detecting and correcting (or removing) corrupt or inaccurate records from a record of set, table or database.
- The process on the raw data that makes it “clean” enough to input to our analytical algorithm is called data wrangling



# "Complete Case Study"

# Imputing Missing Values

- we need to subset the categorical and numerical variables for the imputation process.

```
numeric_subset = data.select_dtypes('number')
categorical_subset = data.select_dtypes('object')
```

# Imputing Missing Values

```
from sklearn.impute import SimpleImputer
```

```
# Create an imputer object with a median filling strategy
num_imputer = SimpleImputer(strategy='median')

num_imputer.fit(numeric_subset)

num_data = num_imputer.transform(numeric_subset)
```

```
# Create an imputer object with a mode filling strategy
cat_imputer = SimpleImputer(strategy='most_frequent')

cat_imputer.fit(categorical_subset)

cat_data = cat_imputer.transform(categorical_subset)
```

# Imputing Missing Values

```
num_df = pd.DataFrame(num_data, columns = numeric_subset.columns)

cat_df = pd.DataFrame(cat_data, columns = categorical_subset.columns)

mod_data = pd.concat([num_df, cat_df], axis=1)
mod_data.head()
```

We concatenated the data and saved it as modified data set.

Now let's check the presence of any null values in the data.

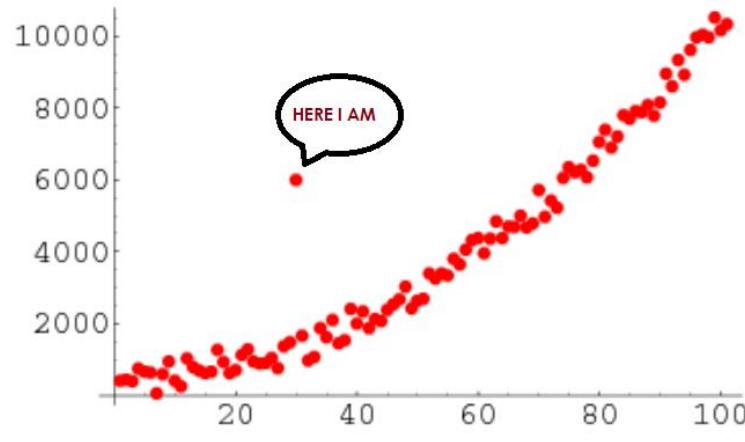
```
mod_data.isnull().sum().sum()
```

0

We can see that there are no null values present in the data set.

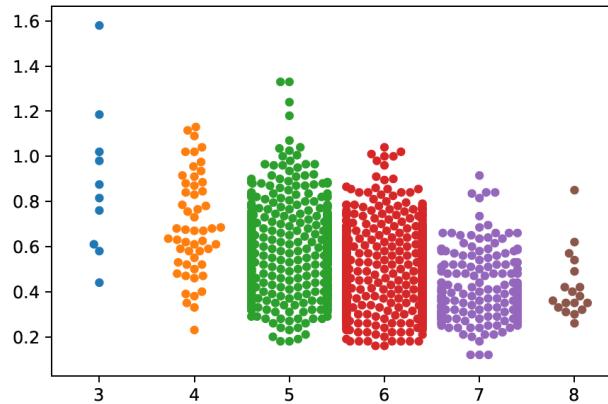
# Outliers

- Outliers are the observations that lie at an abnormal distance from the other values in a random sample taken from a population.



# Exploratory Data Analysis (EDA)

- Exploratory data analysis is an open-ended process, where we perform statistics and make figures to find trends, anomalies, patterns or relationships within the given data.



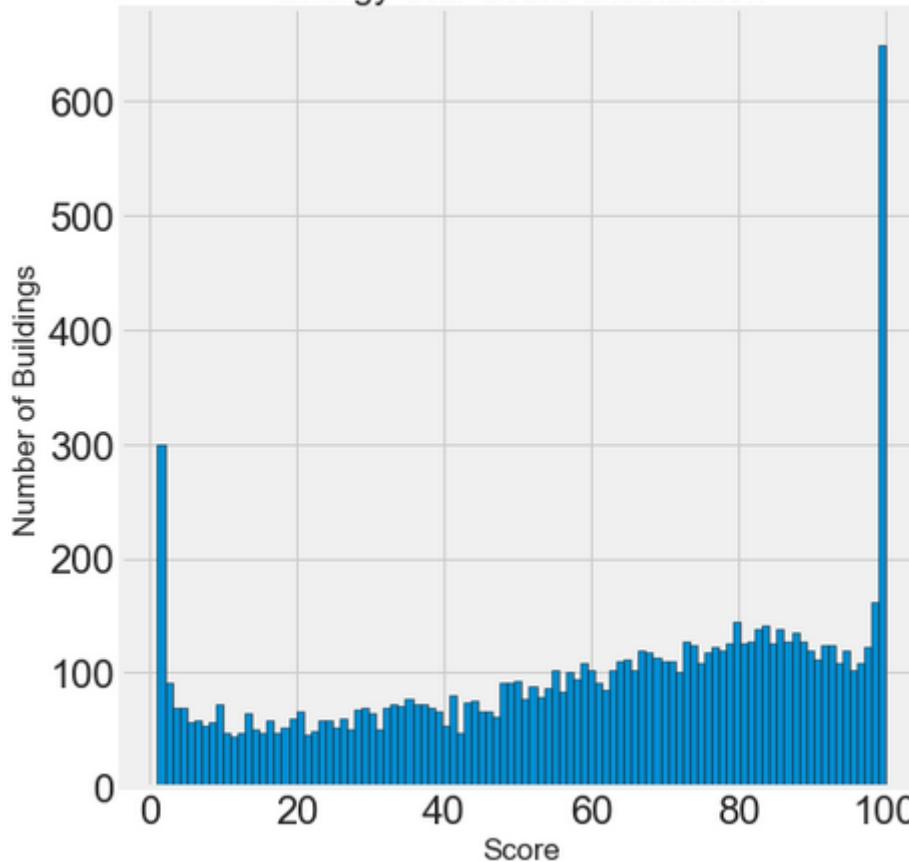
# Univariate analysis (Single variable plots)

```
plt.figure(figsize=(8,8))

# Rename the score
data = data.rename(columns = {'ENERGY STAR Score' : 'Score'})

# Histogram of the Energy Star Score
plt.style.use('fivethirtyeight')
plt.hist(data['Score'].dropna(), bins = 100, edgecolor = 'k')
plt.xlabel('Score')
plt.ylabel('Number of Buildings')
plt.title('Energy Star Score Distribution')
```

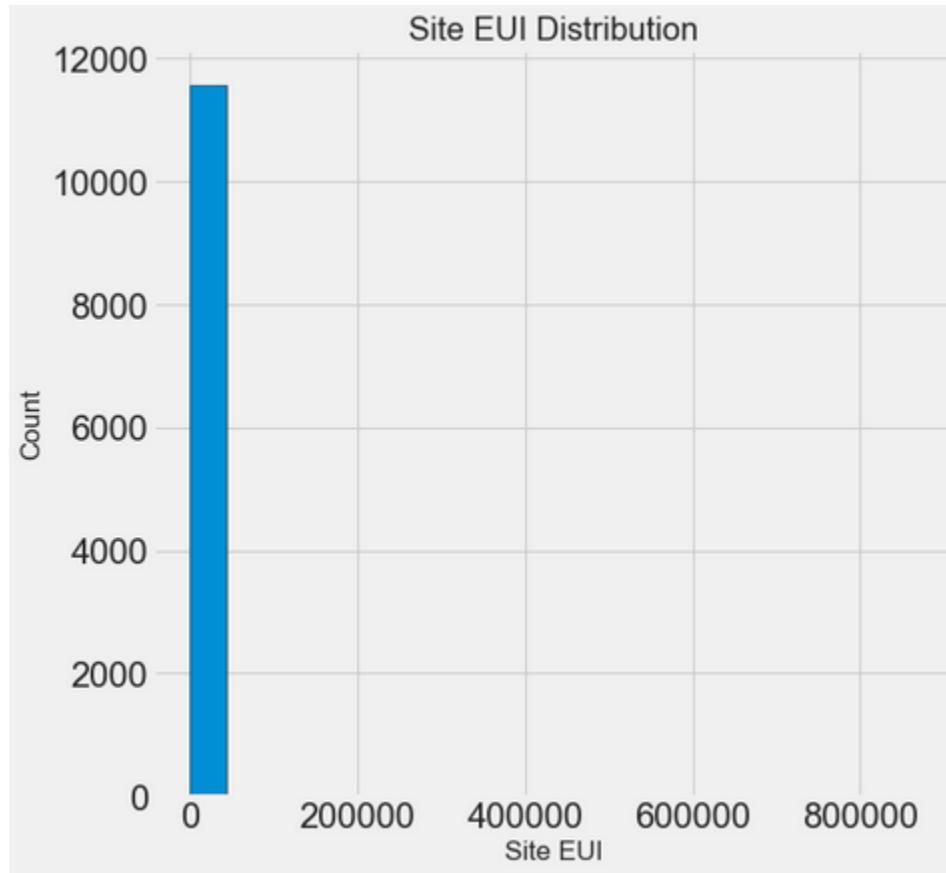
### Energy Star Score Distribution



# Univariate analysis (Single variable plots)

- Let us look at the distribution of the ‘site EUI’ variable.

```
# Histogram plot of site EUI
plt.figure(figsize=(8,8))
plt.hist(data['Site EUI (kBtu/ft²)'].dropna(),bins=20,edgecolor='black')
plt.xlabel('Site EUI')
plt.ylabel('Count')
plt.title('Site EUI Distribution')
```



```
data['Site EUI (kBtu/ft²)'].describe()
```

```
count      11583.000000
mean       280.071484
std        8607.178877
min        0.000000
25%       61.800000
50%       78.500000
75%       97.600000
max       869265.000000
Name: Site EUI (kBtu/ft²), dtype: float64
```

```
data['Site EUI (kBtu/ft²)'].dropna().sort_values(ascending = False).head(10)
```

```
8068      869265.0
7          143974.4
3898      126307.4
8174      112173.6
8268      103562.7
3263      95560.2
8269      84969.6
3383      78360.1
3170      51831.2
3173      51328.8
Name: Site EUI (kBtu/ft²), dtype: float64
```

# Univariate analysis (Single variable plots)

- Wow! One building is clearly far above the rest

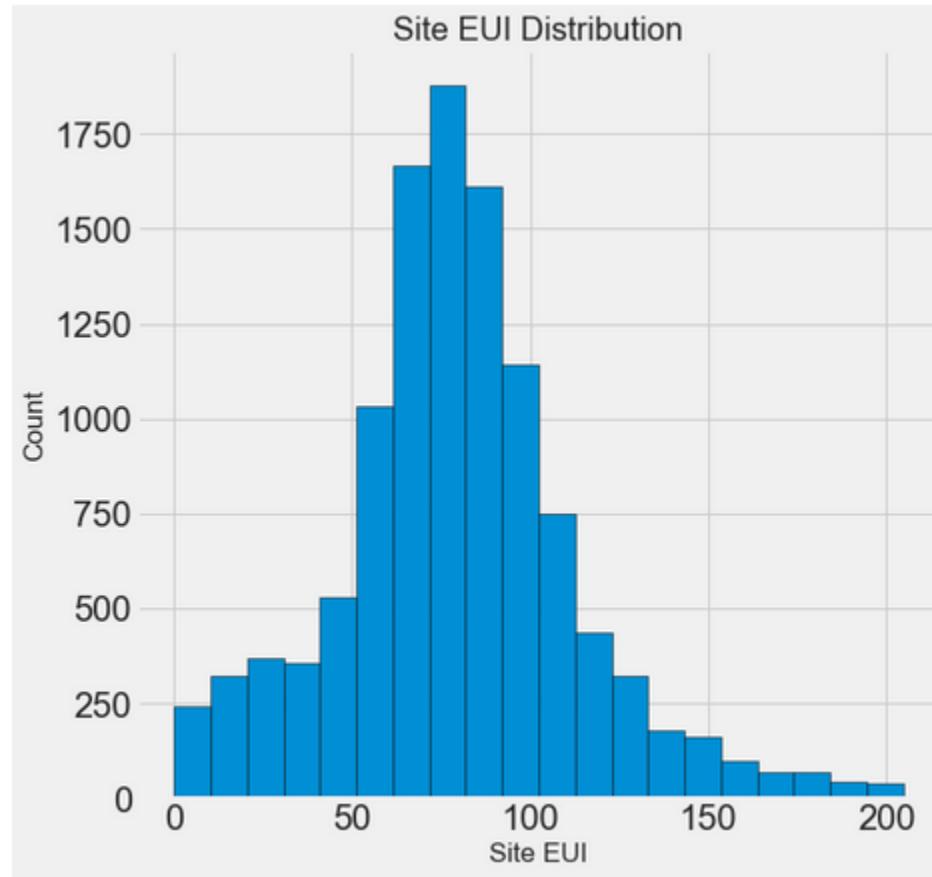
Street Name	Borough	DOF Gross Floor Area	Primary Property Type - Self Selected	List of All Property Use Types at Property	Largest Property Use Type	Largest Property Use Type - Gross Floor Area (ft <sup>2</sup> )	Year Built	Number of Buildings - Self-reported	Occupancy	Metered Areas (Energy)	Metered Areas (Water)	Score	Site EUI (kBtu/ft <sup>2</sup> )
SKILLMAN AVENUE	Brooklyn	61811.0	Multifamily Housing	Multifamily Housing	Multifamily Housing	56900.0	2004	1	90	Whole Building	NaN	1.0	869265.0

# Removing Outliers

```
# Calculate first and third quartile
first_quantile = data['Site EUI (kBtu/ft²)'].describe()['25%']
third_quantile = data['Site EUI (kBtu/ft²)'].describe()['75%']

# Interquartile range
iqr = third_quantile - first_quantile

# Remove outliers
data = data[(data['Site EUI (kBtu/ft²)'] > (first_quantile - 3 * iqr)) &
            (data['Site EUI (kBtu/ft²)'] < (third_quantile + 3 * iqr))]
```



# Removing Outliers

```
data['Site EUI (kBtu/ft2)'].describe()
```

count	11319.000000
mean	79.086377
std	33.317277
min	0.000000
25%	61.200000
50%	77.800000
75%	95.800000
max	204.800000
Name:	Site EUI (kBtu/ft <sup>2</sup> ), dtype: float64

# Bivariate Analysis

- The first plot we will make shows the distribution of scores by the property type.
- In order to not clutter the plot with large data, we will limit the graph to building types that have more than 100 observations in the dataset.

```
types = data.dropna(subset=['Score'])
types = types['Largest Property Use Type'].value_counts()
types = list(types[types.values > 100].index)
types

['Multifamily Housing', 'Office', 'Hotel', 'Non-Refrigerated Warehouse']
```

# Bivariate Analysis

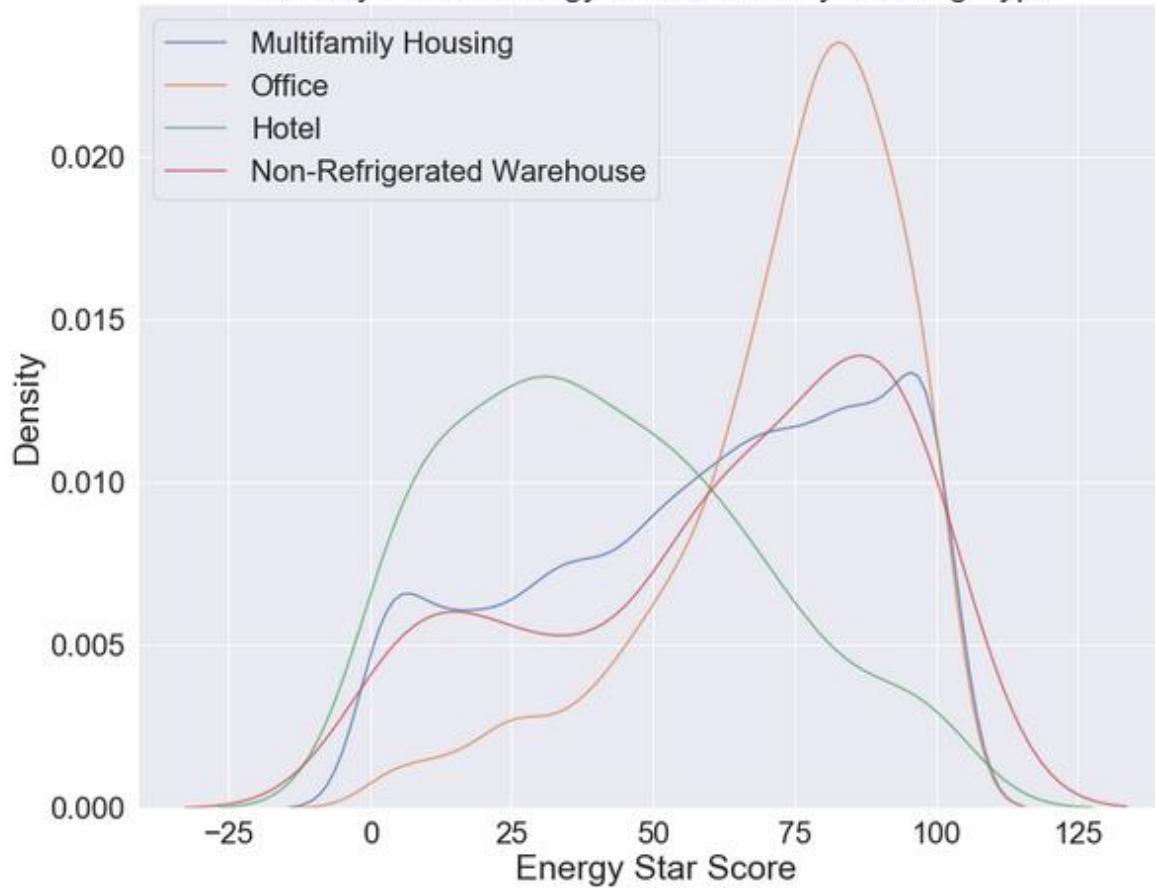
```
# Plot of distribution of scores for building categories

plt.figure(figsize=(12,10))

# plot each building
for b_type in types:
    #select the building type
    subset = data[data['Largest Property Use Type'] == b_type]
    # Density plot of Energy Star Scores
    sns.kdeplot(subset['Score'].dropna(),
                 label = b_type, shade = False,
                 alpha = 0.8)

# label the plot
plt.xlabel('Energy Star Score', size = 25)
plt.ylabel('Density', size = 25);
plt.title('Density Plot of Energy Star Scores by Building Type', size = 25);
```

## Density Plot of Energy Star Scores by Building Type



```
# Create a list of boroughs with more than 100 observations
boroughs = data.dropna(subset=['Score'])
boroughs = boroughs['Borough'].value_counts()
boroughs = list(boroughs[boroughs.values > 100].index)
boroughs

['Manhattan', 'Brooklyn', 'Queens', 'Bronx', 'Staten Island']
```

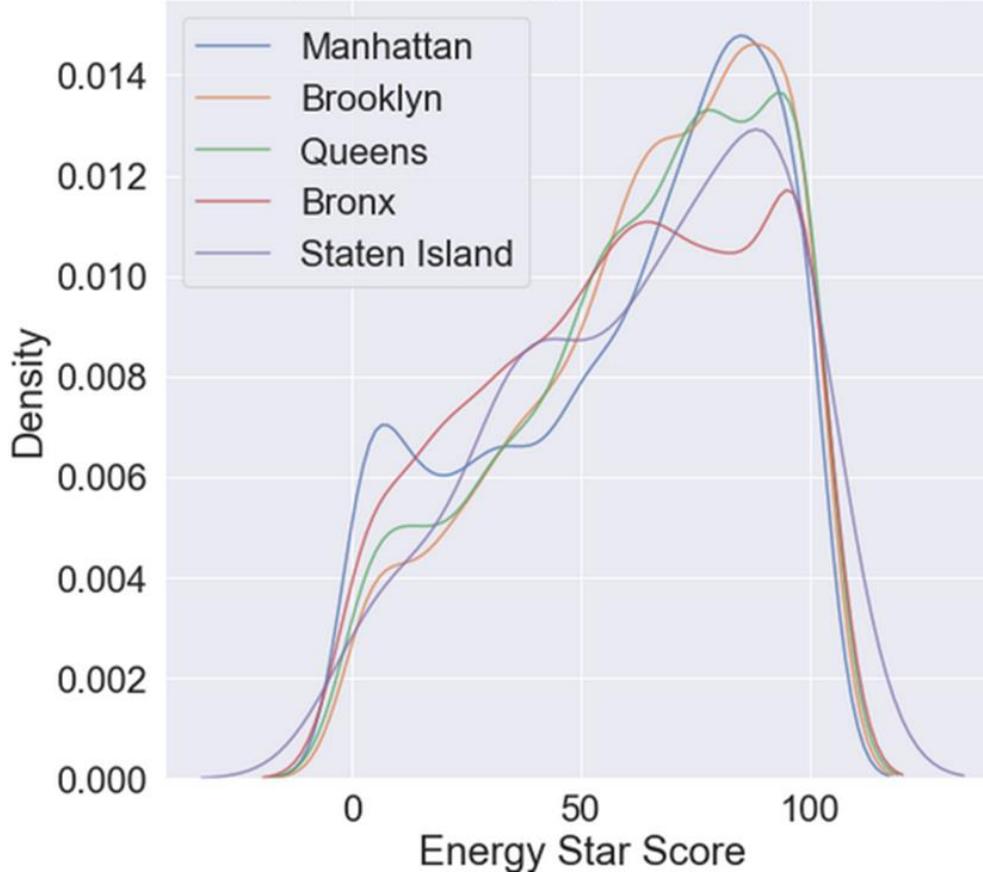
```
# Plot of distribution scores of boroughs

plt.figure(figsize=(8,8))

# Plot each borough
for b_borough in boroughs:
    subset = data[data['Borough'] == b_borough]
    sns.kdeplot(subset['Score'].dropna(),
                label = b_borough, shade = False,
                alpha = 0.8)

plt.xlabel("Energy Star Score", size = 5)
plt.ylabel("Density")
plt.title("Density plot of Energy Star Score by Borough")
```

## Density plot of Energy Star Score by Borough



# Correlations between Features and Target

```
# Find all correlations and sort
correlations_data = data.corr()['Score'].sort_values()

# Print the most negative correlations
print(correlations_data.head(15), '\n')

# Print the most positive correlations
print(correlations_data.tail(15))
```

# Correlations between Features and Target

```
Site EUI (kBtu/ft2)                                -0.723864
Weather Normalized Site EUI (kBtu/ft2)              -0.713993
Weather Normalized Source EUI (kBtu/ft2)            -0.645542
Source EUI (kBtu/ft2)                               -0.641037
Weather Normalized Site Electricity Intensity (kWh/ft2) -0.358394
Weather Normalized Site Natural Gas Intensity (therms/ft2) -0.346046
Direct GHG Emissions (Metric Tons CO2e)                -0.147792
Weather Normalized Site Natural Gas Use (therms)        -0.135211
Natural Gas Use (kBtu)                                 -0.133648
Year Built                                              -0.121249
Total GHG Emissions (Metric Tons CO2e)                 -0.113136
Electricity Use - Grid Purchase (kBtu)                  -0.050639
Weather Normalized Site Electricity (kWh)               -0.048207
Latitude                                                 -0.048196
Property Id                                             -0.046605
Name: Score, dtype: float64
```

# Correlations between Features and Target

Property Id	-0.046605
Indirect GHG Emissions (Metric Tons CO <sub>2</sub> e)	-0.043982
Longitude	-0.037455
Occupancy	-0.033215
Number of Buildings - Self-reported	-0.022407
Water Use (All Water Sources) (kgal)	-0.013681
Water Intensity (All Water Sources) (gal/ft <sup>2</sup> )	-0.012148
Census Tract	-0.002299
DOF Gross Floor Area	0.013001
Property GFA - Self-Reported (ft <sup>2</sup> )	0.017360
Largest Property Use Type - Gross Floor Area (ft <sup>2</sup> )	0.018330
Order	0.036827
Community Board	0.056612
Council District	0.061639
Score	1.000000
Name: Score, dtype: float64	

```
numeric_subset = data.select_dtypes('number')

# Create columns with square root and log of numeric columns
for col in numeric_subset.columns:
    # Skip the Energy Star Score column
    if col == 'Score':
        next
    else:
        numeric_subset['sqrt_' + col] = np.sqrt(numeric_subset[col])
        numeric_subset['log_' + col] = np.log(numeric_subset[col])

# Select the categorical columns
categorical_subset = data[['Borough', 'Largest Property Use Type']]

# One hot encode
categorical_subset = pd.get_dummies(categorical_subset)

# Join the two dataframes using concat
# Make sure to use axis = 1 to perform a column bind
features = pd.concat([numeric_subset, categorical_subset], axis = 1)

# Drop buildings without an energy star score
features = features.dropna(subset = ['Score'])

# Find correlations with the score
correlations = features.corr()['Score'].dropna().sort_values()
```

# Correlations between Features and Target

```
# Display most negative correlations
correlations.head(15)|
```

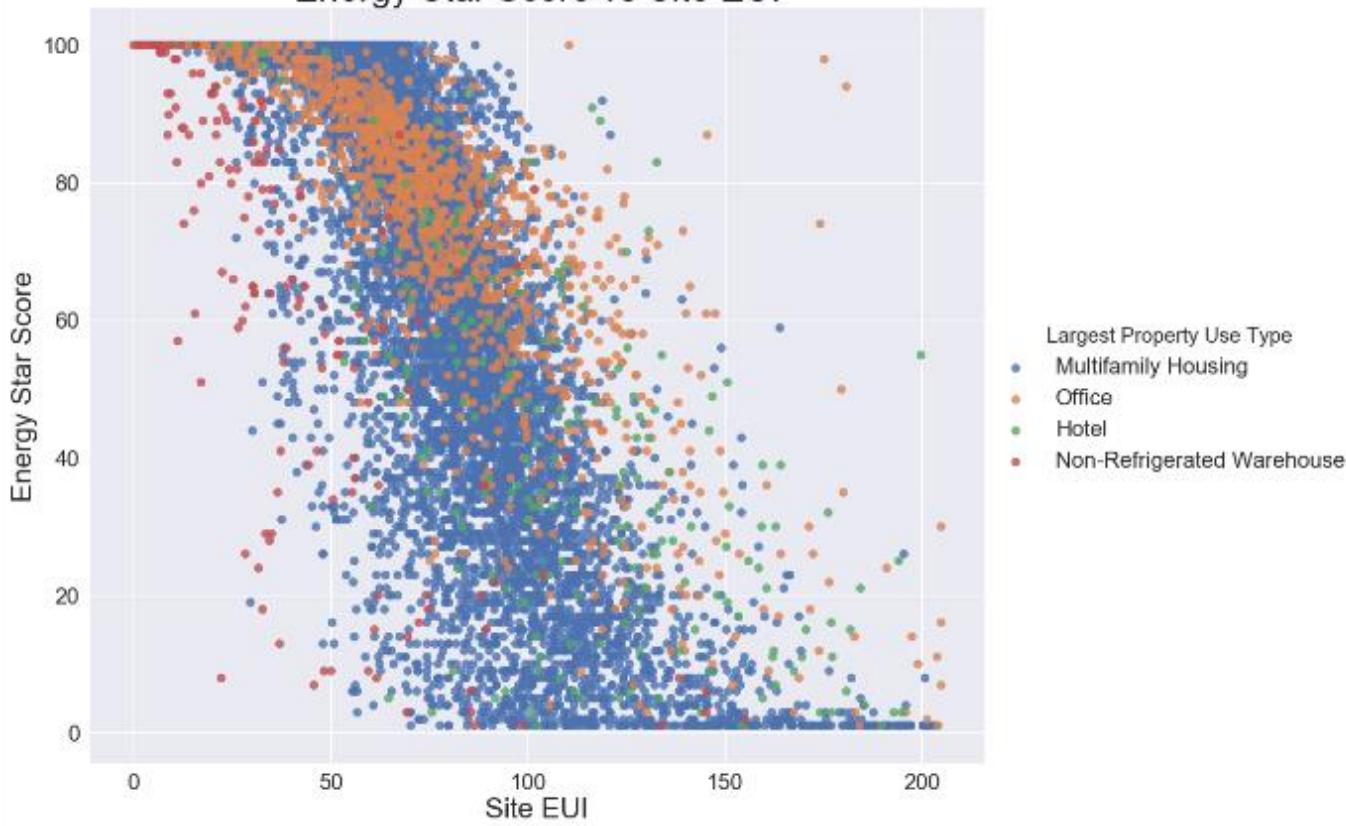
Site EUI (kBtu/ft <sup>2</sup> )	-0.723864
Weather Normalized Site EUI (kBtu/ft <sup>2</sup> )	-0.713993
sqrt_Site EUI (kBtu/ft <sup>2</sup> )	-0.699817
sqrt_Weather Normalized Site EUI (kBtu/ft <sup>2</sup> )	-0.689019
sqrt_Weather Normalized Source EUI (kBtu/ft <sup>2</sup> )	-0.671044
sqrt_Source EUI (kBtu/ft <sup>2</sup> )	-0.669396
Weather Normalized Source EUI (kBtu/ft <sup>2</sup> )	-0.645542
Source EUI (kBtu/ft <sup>2</sup> )	-0.641037
log_Source EUI (kBtu/ft <sup>2</sup> )	-0.622892
log_Weather Normalized Source EUI (kBtu/ft <sup>2</sup> )	-0.620329
log_Site EUI (kBtu/ft <sup>2</sup> )	-0.612039
log_Weather Normalized Site EUI (kBtu/ft <sup>2</sup> )	-0.601332
log_Weather Normalized Site Electricity Intensity (kWh/ft <sup>2</sup> )	-0.424246
sqrt_Weather Normalized Site Electricity Intensity (kWh/ft <sup>2</sup> )	-0.406669
Weather Normalized Site Electricity Intensity (kWh/ft <sup>2</sup> )	-0.358394
Name: Score, dtype: float64	

# Correlations between Features and Target

```
# Display most positive correlations  
correlations.tail(15)
```

sqrt_Order	0.028662
Borough_Queens	0.029545
Largest Property Use Type_Supermarket/Grocery Store	0.030038
Largest Property Use Type_Residence Hall/Dormitory	0.035407
Order	0.036827
Largest Property Use Type_Hospital (General Medical & Surgical)	0.048410
Borough_Brooklyn	0.050486
log_Community Board	0.055495
Community Board	0.056612
sqrt_Community Board	0.058029
sqrt_Council District	0.060623
log_Council District	0.061101
Council District	0.061639
Largest Property Use Type_Office	0.158484
Score	1.000000
Name: Score, dtype: float64	

## Energy Star Score vs Site EUI



# Multivariate Analysis

```
# Extract the columns to plot
plot_data = features[['Score', 'Site EUI (kBtu/ft²)',
                      'Weather Normalized Source EUI (kBtu/ft²)',
                      'log_Total GHG Emissions (Metric Tons CO2e)']]

# Replace the inf with nan
plot_data = plot_data.replace({np.inf: np.nan, -np.inf: np.nan})

# Rename columns
plot_data = plot_data.rename(columns = {'Site EUI (kBtu/ft²)': 'Site EUI',
                                         'Weather Normalized Source EUI (kBtu/ft²)': 'Weather Norm EUI',
                                         'log_Total GHG Emissions (Metric Tons CO2e)': 'log GHG Emissions'})

# Drop na values
plot_data = plot_data.dropna()

# Function to calculate correlation coefficient between two columns
def corr_func(x, y, **kwargs):
    r = np.corrcoef(x, y)[0][1]
    ax = plt.gca()
    ax.annotate("r = {:.2f}".format(r),
                xy=(.2, .8), xycoords = ax.transAxes,
                size = 10)
```

# Multivariate Analysis

```
# Create the pairgrid object
grid = sns.PairGrid(data = plot_data, size = 3)

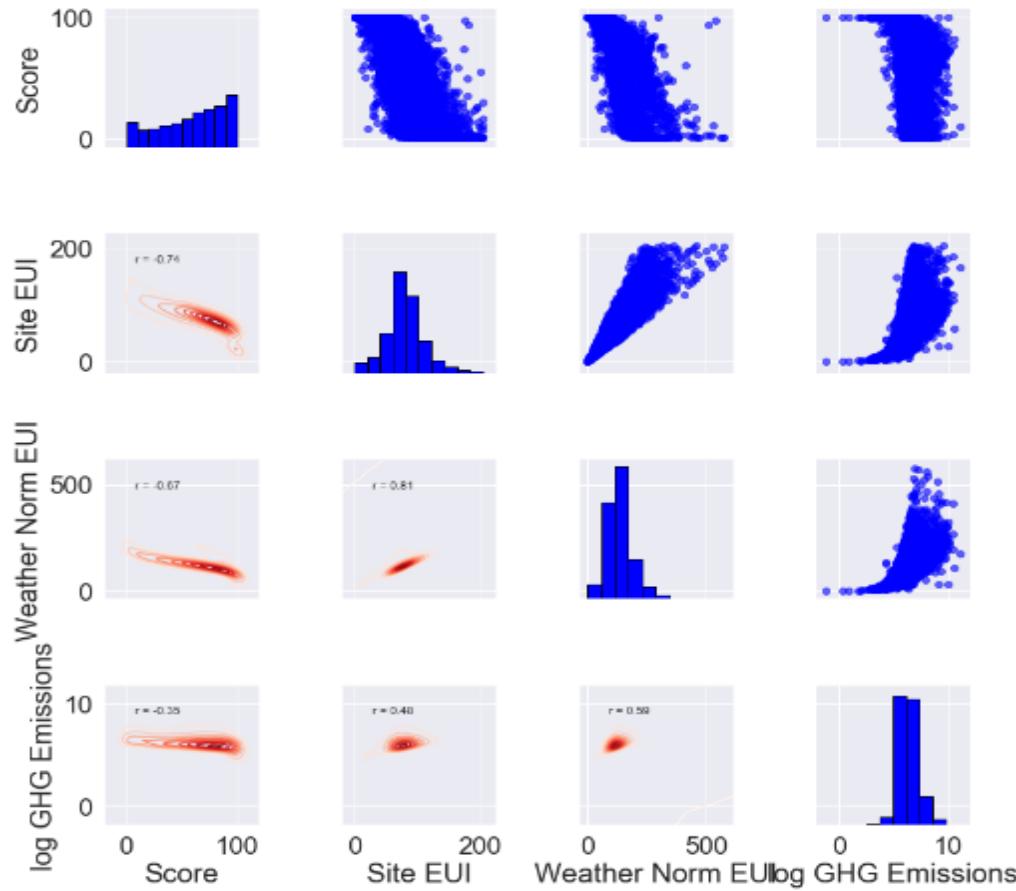
# Upper is a scatter plot
grid.map_upper(plt.scatter, color = 'blue', alpha = 0.6)

# Diagonal is a histogram
grid.map_diag(plt.hist, color = 'blue', edgecolor = 'black')

# Bottom is correlation and density plot
grid.map_lower(corr_func)
grid.map_lower(sns.kdeplot,cmap = plt.cm.Reds)

#Title for entire plot
plt.suptitle('Pairs Plot of Engery Data', size = 25, y = 1.02)
```

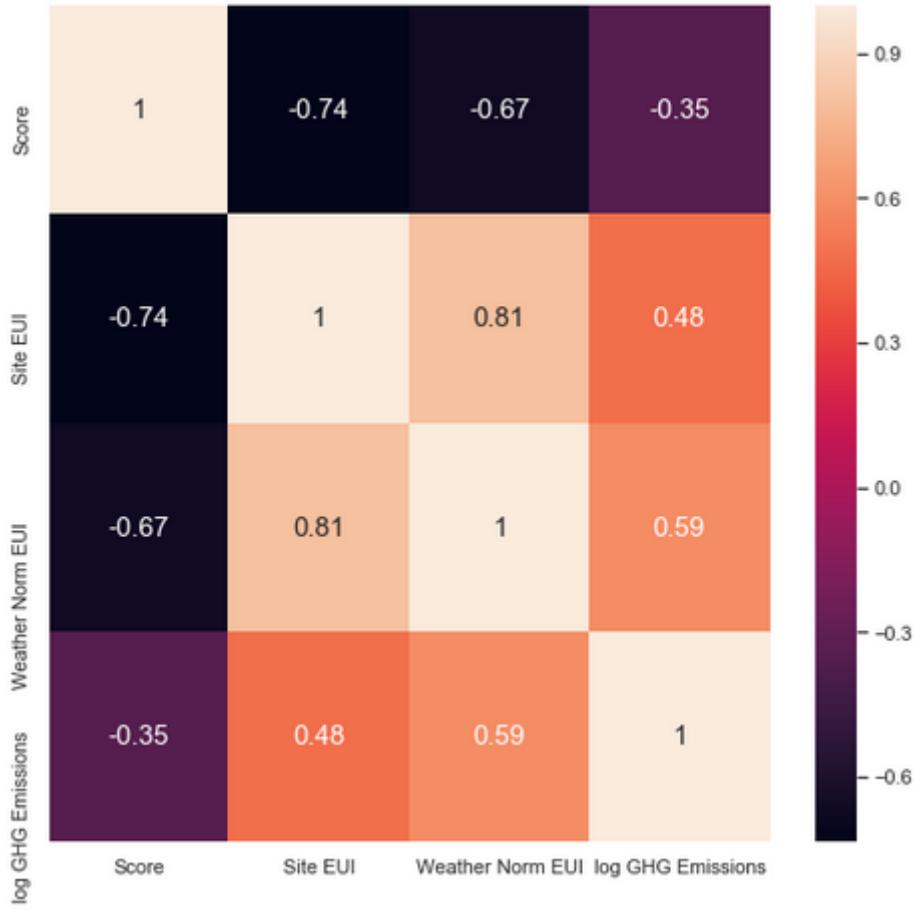
Pairs Plot of Engery Data



# Multivariate Analysis

- We also utilize the heat map to visualize the correlation between the continuous variables.

```
corr = plot_data.corr()
f, ax = plt.subplots(figsize=(8, 8))
sns.heatmap(corr, vmax=1, annot_kws={'size': 15}, annot=True);
```



# Multivariate Analysis

```
# we will limit the graph to building types that have
# more than 100 observations in the dataset.
building_types = data.dropna(subset=['Score'])
building_types = building_types['Largest Property Use Type'].value_counts()
building_types = list(building_types[building_types.values > 100].index)
print("Buidling types with more than 100 observations ",building_types)

# Create a list of boroughs with more than 100 observations
boroughs = data.dropna(subset=['Score'])
boroughs = boroughs['Borough'].value_counts()
boroughs = list(boroughs[boroughs.values > 100].index)
print("Boroughs with more than 100 observations ",boroughs)
```

Buidling types with more than 100 observations ['Multifamily Housing', 'Office', 'Hotel', 'Non-Refrigerated Warehouse']  
Boroughs with more than 100 observations ['Manhattan', 'Brooklyn', 'Queens', 'Bronx', 'Staten Island']

# Multivariate Analysis

- We filter the dataset based on the building types and boroughs.

```
multivari_data = data[data['Largest Property Use Type'].isin(building_types) &
                      data['Borough'].isin(boroughs)].dropna()
multivari_data.rename(columns = {'Largest Property Use Type':'BuildingType'},
                      inplace = True)
multivari_data.head()
```

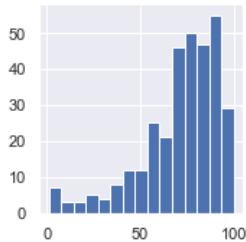
# Multivariate Analysis

Order	Property Id	Property Name	Parent Property Id	Parent Property Name	BBL - 10 digits	NYC Borough, Block and Lot (BBL) self-reported	NYC Building Identification Number (BIN)	Address 1 (self-reported)	Postal Code	Street Number	
99	102	2605684	Hammer Health Sciences Center	3614737	Columbia University Medical Center	1021390051	1021390051	1063402	1 Haven Ave; 701 W 168 Street	10032	1
103	106	2741656	154 Haven Dormitory	3614737	Columbia University Medical Center	1021390275	1021390275	1063430	154 Haven Avenue	10032	154
161	165	2809891	434 West 120th Street	3618216	435 W 119 and 434 W 120	1019620070	1019620070	1059514	434 West 120th Street	10027	1211
323	332	4414870	Dayton Towers: 76-00 Shore Front Parkway	4994297	1-50/76-00 Dayton Towers	4161280001	4-16128-0001	4457805	76-00 Shore Front Parkway	11692	7600
327	336	4994375	Riverbend 2301-2311 (WW)	4994371	Riverbend (WW)	1017640001	1-01764-0001	1054345	2289-2311 5th Avenue	10037	2301

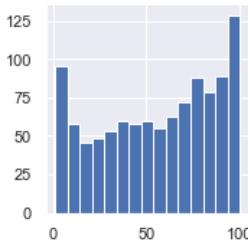
# Multivariate Analysis

```
plt.figure(figsize=(20,12))
x=sns.FacetGrid(multivari_data, row='Borough', col = 'BuildingType',
                 palette='husl',sharex=False,sharey=False, margin_titles=True)
x=x.map(plt.hist, 'Score', bins=15)
x=x.fig.subplots_adjust(wspace=0.5, hspace=0.5)
```

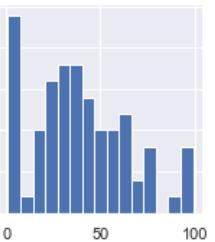
BuildingType = Office



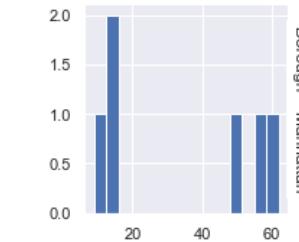
BuildingType = Multifamily Housing



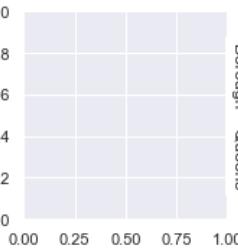
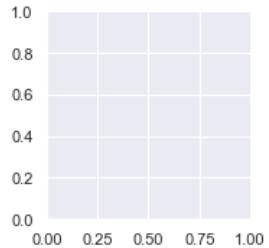
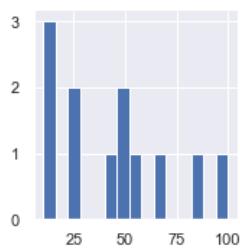
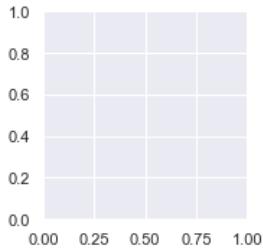
BuildingType = Hotel



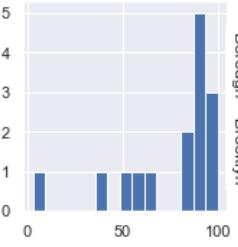
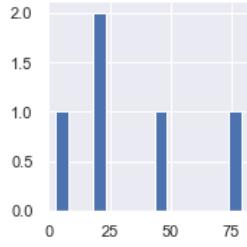
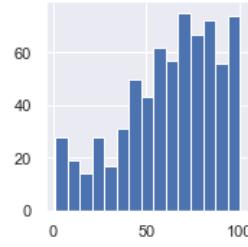
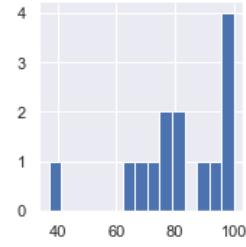
BuildingType = Non-Refrigerated Warehouse



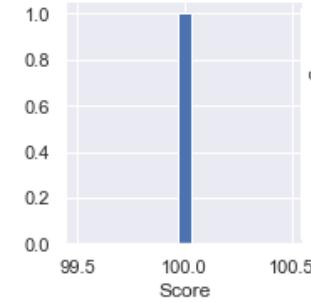
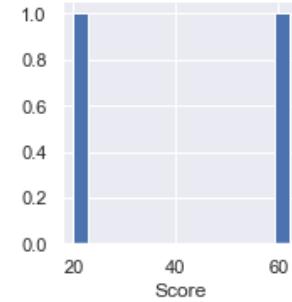
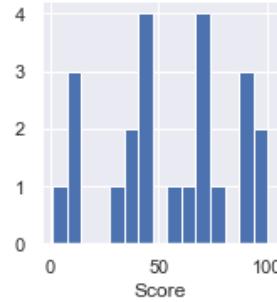
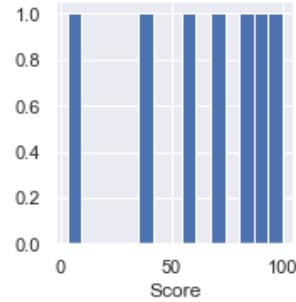
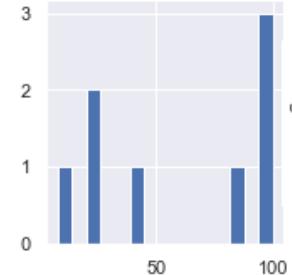
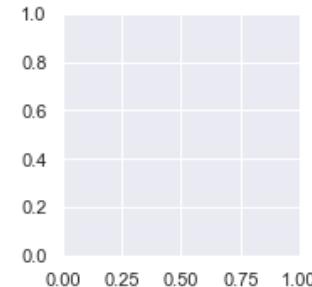
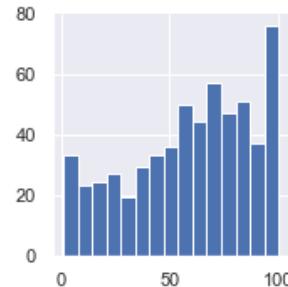
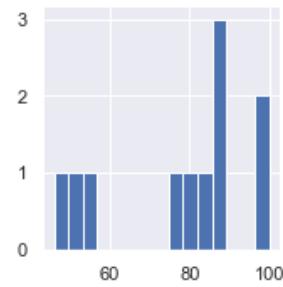
Borough = Manhattan



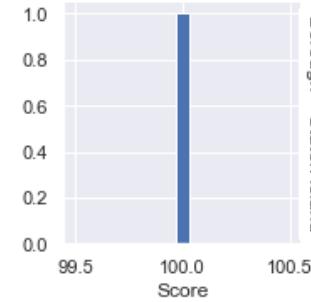
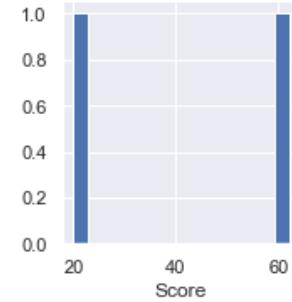
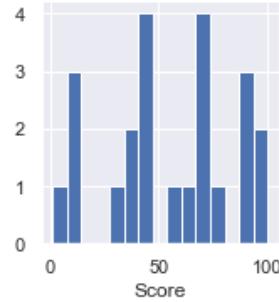
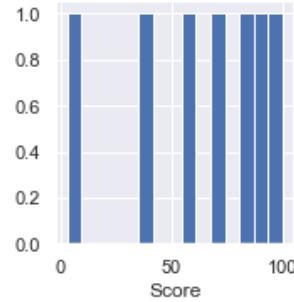
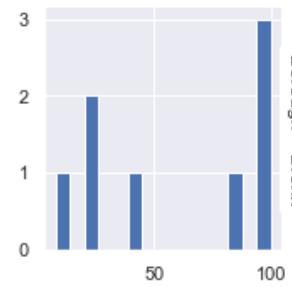
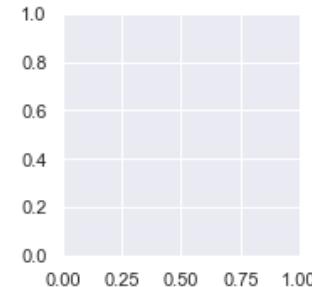
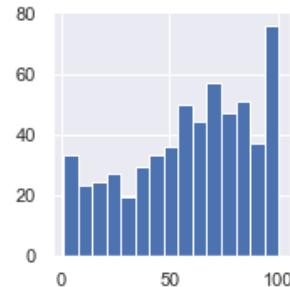
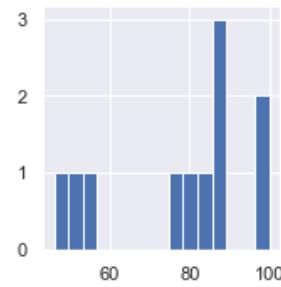
Borough = Brooklyn



# Multivariate Analysis



# Multivariate Analysis



Borough = Bronx

Borough = Staten Island

# "Complete Assessment "

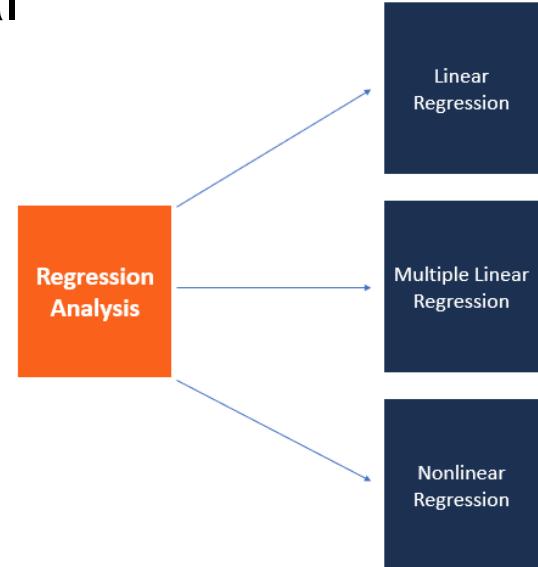
# "Complete Lab 5"

# 6: Regression Analysis



# Regression Analysis

- Regression analysis is a set of statistical procedures for estimating the relationship between a dependent variable (often called the ‘outcome variable’ or ‘target variable’)
- One or more independent variables (often called ‘predictors’ or ‘features’).





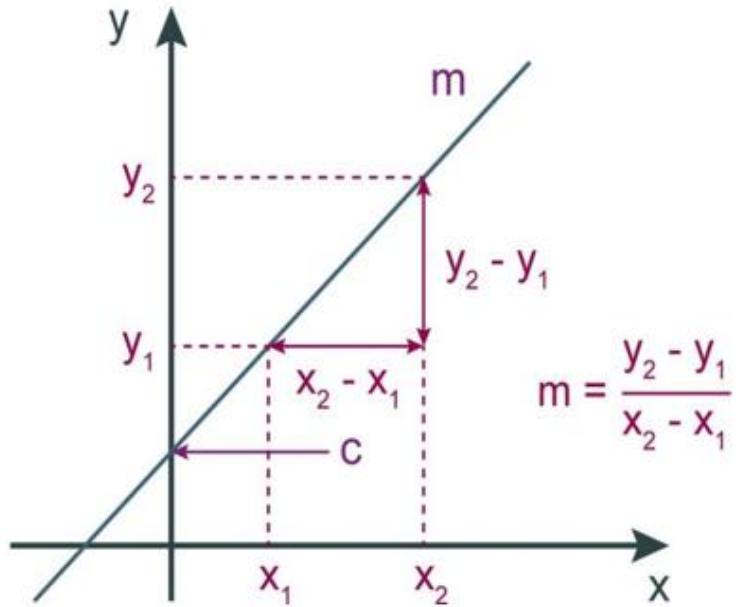
# The use of Regression Analysis

The benefits of using Regression analysis for calculating these chances are as follows:

- It provides the significant relationships between the label (dependent variable) and the feature (independent variable).
- It shows the extent of the impact of multiple independent variables on the dependent variable.
- It can also quantify these effects even if the variables are on a different scale

# Linear Regression

$$Y = MX + C$$



# Problem Statement

- This data is about the amount spent on advertising certain products through various media channels like TV, radio, and newspaper.
- The goal is to predict how the expense on each channel affects the sales and is there a way to optimize the sales?



Importing the necessary packages.

```
# necessary Imports
import pandas as pd
import matplotlib.pyplot as plt
import pickle
%matplotlib inline
```

Loading and exploring the data.

```
# loading the data

data = pd.read_csv(r'D:\Data Sets\Advertising.csv',
                   index_col='Index')

# Checking the first five rows from the dataset
data.head()
```

Index	TV	Radio	Newspaper	Sales
1	230.1	37.8	69.2	22.1
2	44.5	39.3	45.1	10.4
3	17.2	45.9	69.3	9.3
4	151.5	41.3	58.5	18.5
5	180.8	10.8	58.4	12.9

# Problem Statement

- Dimensions of the data

```
data.shape
```

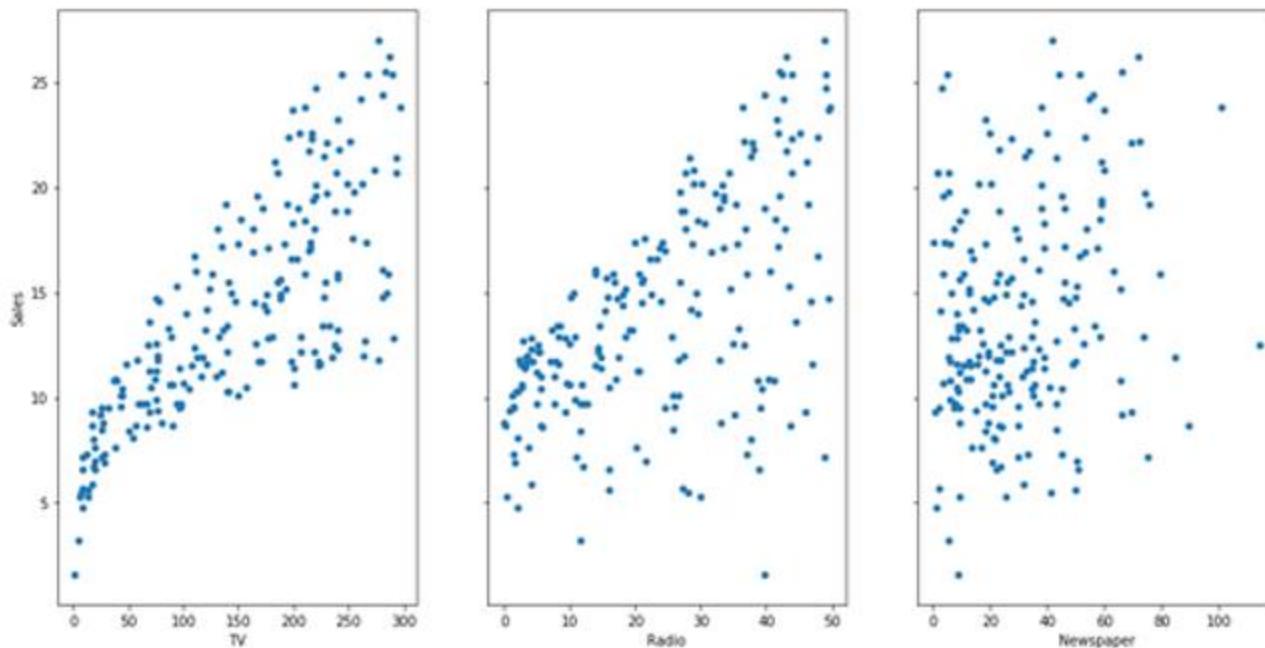
```
(200, 4)
```

- Find the missing values from different columns:

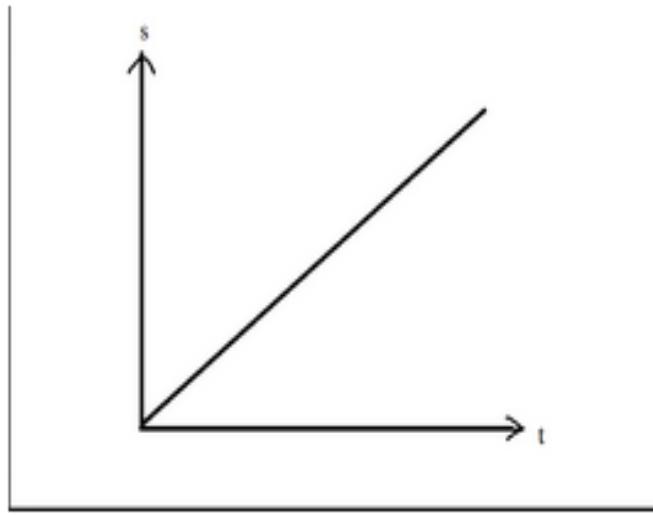
```
data.isna().sum()
```

```
TV          0
Radio       0
Newspaper   0
Sales        0
dtype: int64
```

```
# visualize the relationship between the features and the response using scatterplots
fig, axs = plt.subplots(1, 3, sharey=True)
data.plot(kind='scatter', x='TV', y='Sales', ax=axs[0], figsize=(16, 8))
data.plot(kind='scatter', x='Radio', y='Sales', ax=axs[1])
data.plot(kind='scatter', x='Newspaper', y='Sales', ax=axs[2])
```



# Linear Regression



- Hence, we can build a model using the Linear Regression Algorithm.

# Simple Linear Regression

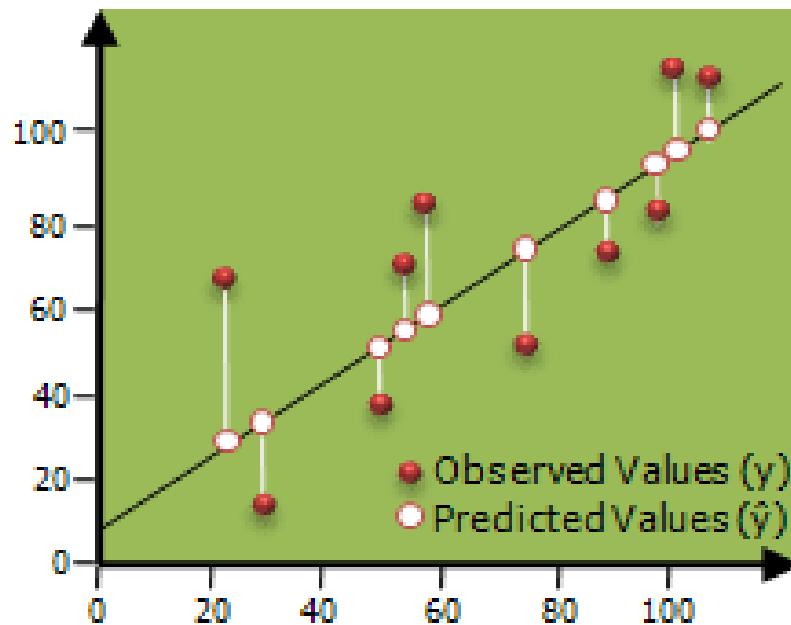
- A simple linear regression is a method for predicting a quantitative response using a single feature (“input variable”).
- The mathematical equation is given as:

$$y = \beta_0 + \beta_1 x$$

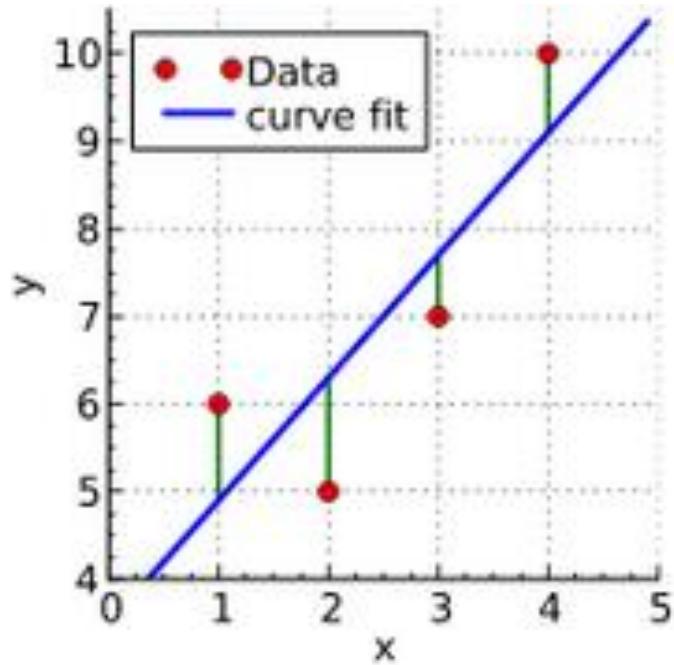
# Estimating (“Learning”) Model Coefficients

- The coefficients are estimated using the Ordinary Least Squares estimates criterion (OLS],
- i.e., a best fit line that minimizes the sum of the squared residuals (or “Sum of Squared Error”).

# Ordinary Least Squared Estimation



# The Mathematics involved



# Derivation of OLS by Minimizing Errors

- Minimize the sum of squared error term by substituting as below:

Minimize  $\sum_{i=1}^n e_i = \sum_{i=1}^n (y_i - \hat{y})^2$ ; where  $\hat{y} = \beta_0 + \beta_1 x_i$

Therefore,  $\sum_{i=1}^n e_i = \sum_{i=1}^n (y_i - \beta_0 - \beta_1 x_i)^2$

# Derivation of OLS by Minimizing Errors

- Solving the above equation by calculus – partial differencing by  $\beta_0$  and  $\beta_1$  respectively and solving for the two variables, we get the following equations,

$$\bullet \beta_1 = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^n (x_i - \bar{x})^2} = \frac{\bar{xy} - \bar{x}\bar{y}}{\bar{x}^2 - \bar{x}^2}$$

$$\bullet \beta_0 = \bar{y} - \beta_1 \bar{x}$$

# Derivation of OLS by Minimizing Errors

- Building Simple Linear Regression Model to predict the sales based on TV ads,

```
# create X and y
feature_cols = ['TV']
X = data[feature_cols]
y = data.Sales

# follow the usual sklearn pattern: import, instantiate, fit
from sklearn.linear_model import LinearRegression
lm = LinearRegression()
lm.fit(X, y)

# print intercept and coefficients
print(lm.intercept_)
print(lm.coef_)
```

7.032593549127695  
[0.04753664]

# Prediction using the model

If the expense on a TV ad is \$50000, what will be the sales prediction for that market?

$$y = \beta_0 + \beta_1 x$$

$$Y = 7.032594 + 0.047537 * (50)$$

```
#calculate the prediction  
7.032594 + 0.047537*50
```

9.409444

# Prediction using the model

Let's do the same calculation using the code.

```
# Let's create a DataFrame since the model expects it
X_new = pd.DataFrame({'TV': [50]})
X_new.head()
```

TV
0 50

```
# use the model to make predictions on a new value
lm.predict(X_new)
```

```
array([9.40942557])
```

# Plotting the least Squares Line:

```
# create a DataFrame with the minimum and maximum values of TV
X_new = pd.DataFrame({'TV': [data.TV.min(), data.TV.max()]})
X_new.head()
```

	TV
0	0.7
1	296.4

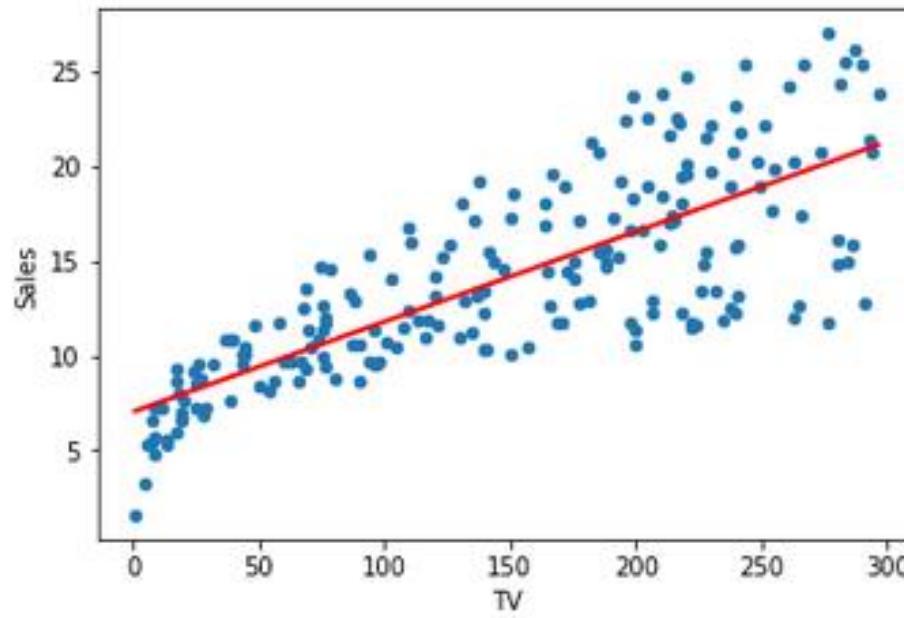
```
# make predictions for those x values and store them
preds = lm.predict(X_new)
preds

array([ 7.0658692 , 21.12245377])
```

```
# first, plot the observed data
data.plot(kind='scatter', x='TV', y='Sales')

# then, plot the least squares line
plt.plot(X_new, preds, c='red', linewidth=2)
```

# Plotting the least Squares Line:



# Plotting the least Squares Line:

```
import statsmodels.formula.api as smf
lm = smf.ols(formula='sales ~ TV', data=data).fit()
lm.conf_int()
```

	0	1
Intercept	6.129719	7.935468
TV	0.042231	0.052843

# Hypothesis Testing and p-values

```
import statsmodels.formula.api as smf
lm = smf.ols(formula='sales ~ TV', data=data).fit()
lm.conf_int()
```

	0	1
Intercept	6.129719	7.935468
TV	0.042231	0.052843

```
# print the p-values for the model coefficients
lm.pvalues
```

```
Intercept      1.406300e-35
TV              1.467390e-42
dtype: float64
```

# Multiple Linear Regression

- A multiple linear regression is a method for predicting a quantitative response using a multiple feature (“input variable”).

The diagram illustrates the components of a multiple linear regression equation. At the bottom, the equation is written as 
$$Y = \beta_0 + \beta_1 X_{1i} + \beta_2 X_{2i} + \dots + \beta_k X_{ki} + \varepsilon$$
. Above the equation, four labels point to specific terms: "Y-intercept" points to  $\beta_0$ , "Population slopes" points to the terms  $\beta_1, \beta_2, \dots, \beta_k$ , and "Error" points to the term  $\varepsilon$ . An additional arrow originates from the "Population slopes" label and points to the first term  $\beta_1 X_{1i}$ .

# Estimation of model parameters

- Consider the model where,

$$Y = X\beta + \epsilon$$
$$Y = \begin{pmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_n \end{pmatrix} \quad X = \begin{pmatrix} 1 & X_{11} & X_{12} & \dots & X_{1p} \\ 1 & X_{21} & X_{22} & \dots & X_{2p} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & X_{n1} & X_{n2} & \dots & X_{np} \end{pmatrix} \quad \beta = \begin{pmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_p \end{pmatrix} \quad \epsilon = \begin{pmatrix} \epsilon_1 \\ \epsilon_2 \\ \vdots \\ \epsilon_n \end{pmatrix}$$

# Estimation of model parameters

- Based on this model, we get the following expansion for the first subject:

$$Y_1 = \beta_0 + \beta_1 X_{11} + \beta_2 X_{12} + \dots + \beta_p X_{1p} + \epsilon_1$$

- Then using the matrix calculus, we can find that the least square estimate for  $\beta$  is given by,

$$\hat{\beta} = (X^T X)^{-1} X^T Y \quad \text{Var}(\hat{\beta}) = \sigma^2 (X^T X)^{-1}$$

# Estimation of model parameters

Hence, the least squares regression line is:

$$\hat{Y} = X\hat{\beta}$$

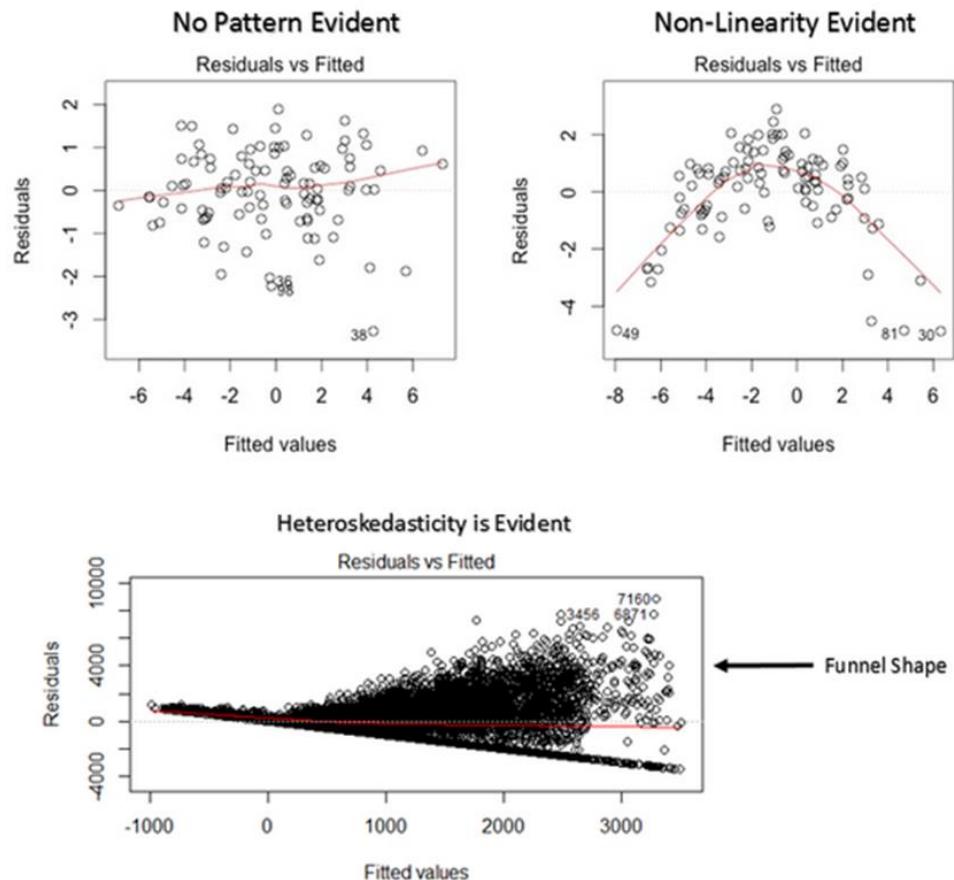
The beta values are obtained by calculating the below equation:

$$\hat{\beta} = (X^T X)^{-1} X^T Y$$

We know that,

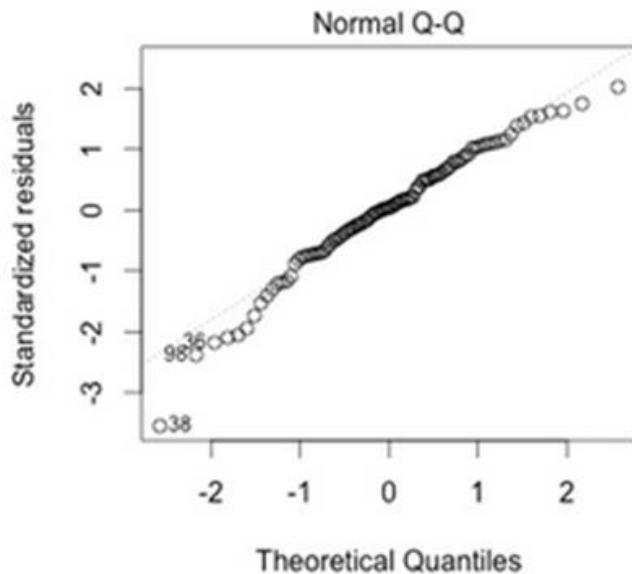
$$\begin{aligned}\hat{Y} &= X\hat{\beta} \\ &= X(X^T X)^{-1} X^T Y \\ &= HY\end{aligned}$$

# Interpretation of Regression Plots

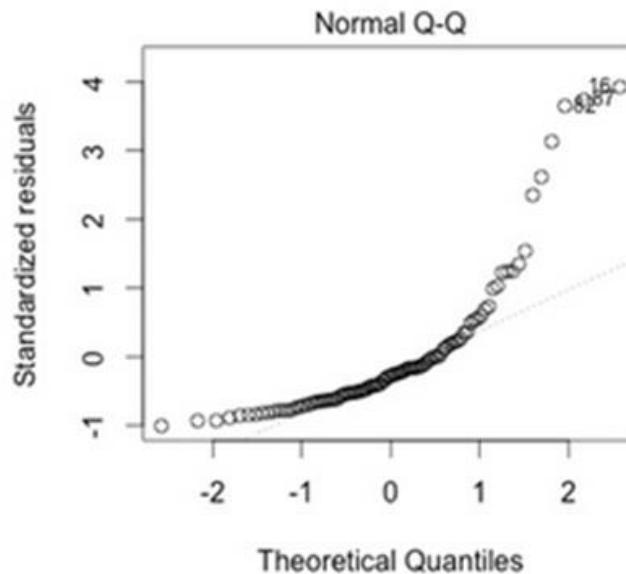


# Normal Q-Q Plot

Normal Distribution Evident

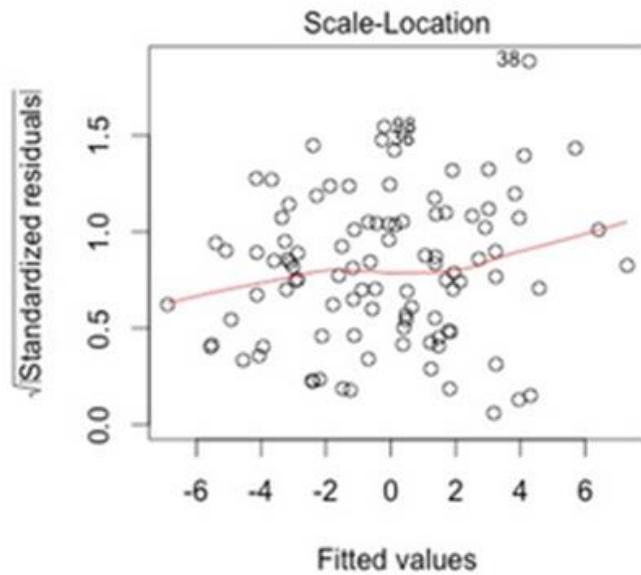


Not Normally Distributed

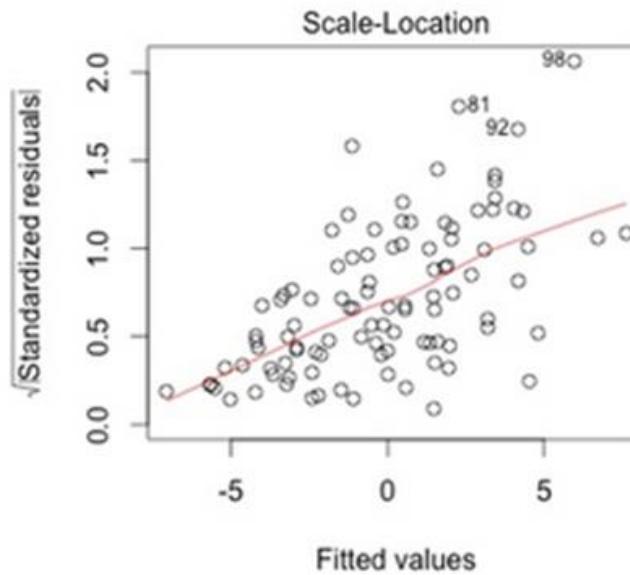


# Scale Location Plot

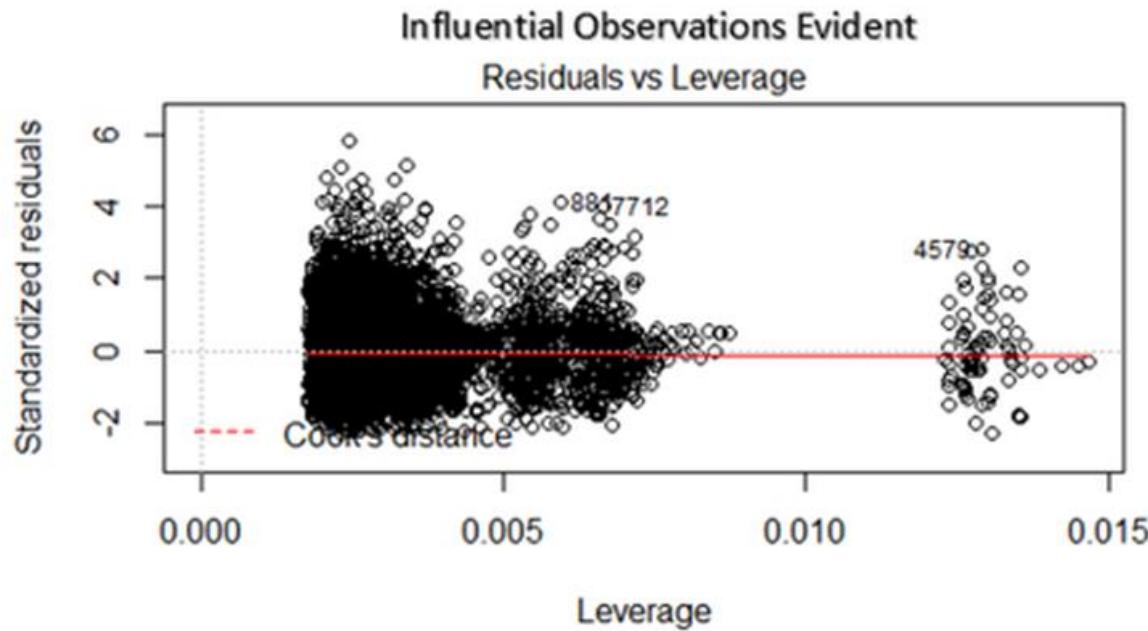
Homoskedasticity is Evident



Heteroskedasticity is Evident



# Residuals vs Leverage Plot



# Model Evaluation Metrics

There are 5 major metrics for model evaluation in regression analysis:

1. R Square
2. Adjusted R Square
3. Mean Square Error (MSE)
4. Root Mean Squared Error (RMSE)
5. Mean Absolute Error

# R Square/Adjusted R Square

Mathematically, **R<sup>2</sup>** statistic is calculated as:

$$R^2 = \frac{TSS - RSS}{TSS} = 1 - \frac{RSS}{TSS}$$

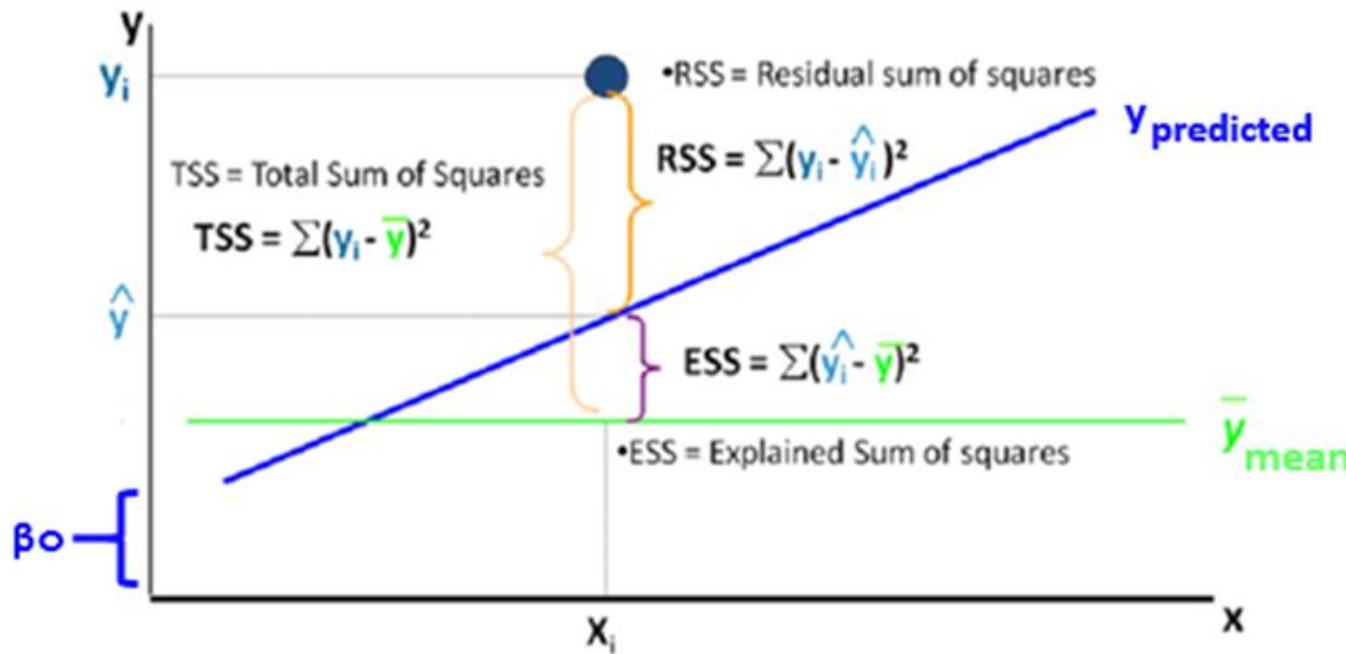
Where **RSS** is the Residual Sum of Squares and is given as:

$$RSS = \sum_{i=1}^n (y_i - \hat{y}_i)^2.$$

RSS is the residual (error) term, whereas, TSS is the Total Sum of Squares and given as:

$$TSS = \sum (y_i - \bar{y})^2$$

# R Square/Adjusted R Square



# Adjusted R Square

- Adjusted R Square is introduced to overcome this problem because it penalizes additional independent variables added to the model and adjust the metric to prevent the overfitting issues.
- Mathematically, it is calculated as:

$$Adjusted\ R^2 = 1 - \frac{(1 - R^2)(N - 1)}{N - p - 1}$$

# Mean Square Error (MSE)

- While R Square is a relative measure of how well the model fits the dependent variables, Mean Square Error (MSE) is an absolute measure of the goodness of fit.
- Mathematically, it is calculated as:

$$MSE = \frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2$$

# Root Mean Square Error (RMSE)

- Root Mean Square Error (RMSE) is the square root of MSE.
- It is used more commonly as compared to MSE, because sometimes MSE values can be too big for easy comparisons.
- Mathematically it is calculated as:

$$\text{RMSE} = \sqrt{\frac{1}{n} \sum_{i=1}^n (y_i - \hat{y})^2}$$

# Mean Absolute Error

- Mean Absolute Error (MAE) is similar to Mean Square Error (MSE).
- However, instead of the sum of square error in MSE, MAE is taking the Sum of Absolute value of error.
- Mathematically, MAE is calculated as:

$$MAE = \frac{1}{N} \sum_{i=1}^N |y_i - \hat{y}_i|$$

# "Complete Lab 6"



# Summary

- Regression models describe the relationship between variables by fitting a line to the observed dataset.
- Generally, the linear regression models use a straight line.
- Regression allows us to estimate how a dependent variable changes with the change in the independent variables.

# "Complete Assessment"

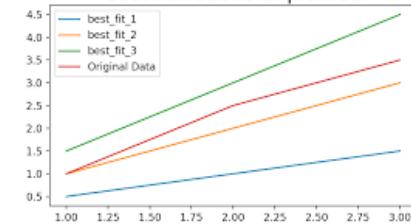
# 6: Regression Analysis (Part 2)



# Cost Function

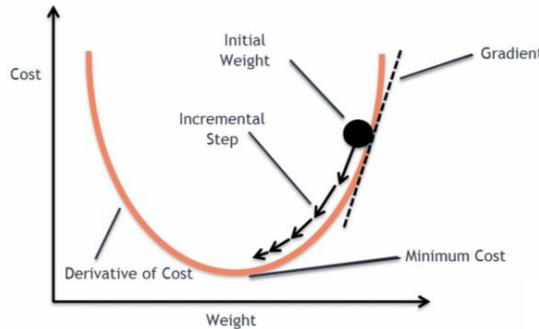
- It is a function that measures the performance of a model for any given data.
- Cost function quantifies the amount of error between predicted and expected values, and presents it as a single real number.

Cost Functions Explained



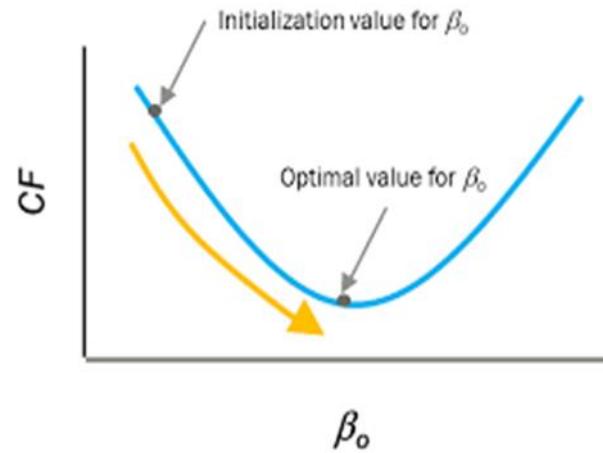
# Gradient Descent

- Gradient descent is an iterative optimization algorithm for finding the local minimum of a function



# Gradient Descent

- The algorithm of Gradient Descent was originally proposed by CAUCHY in 1847.
- It is also known as the steepest descent.



# Gradient Descent

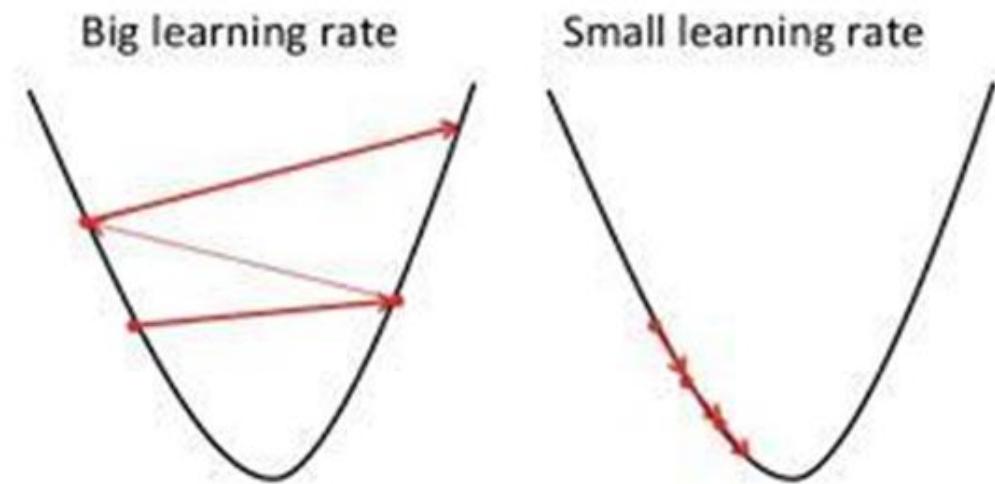
Repeat {

$$\theta_0 := \theta_0 - \alpha \underbrace{\frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)})}_{\frac{\partial}{\partial \theta_0} J(\theta)}$$

$$\theta_1 := \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) x^{(i)}$$

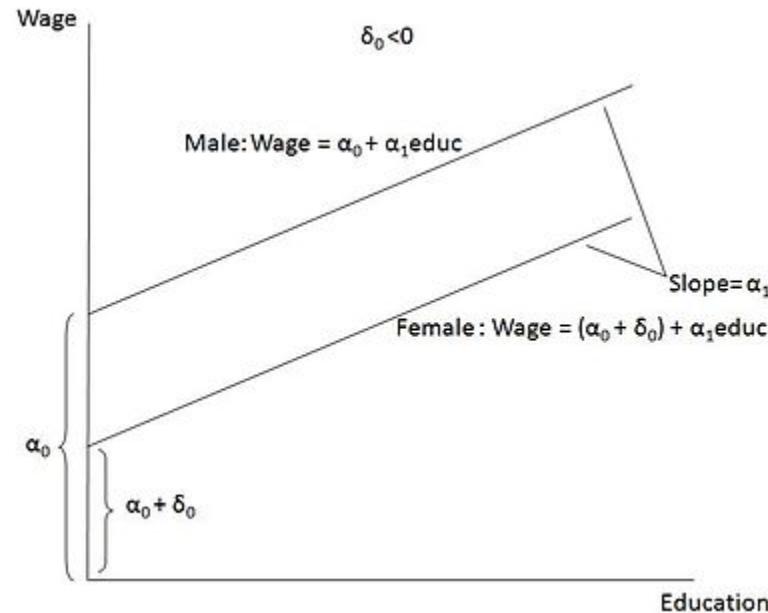
(simultaneously update  $\theta_0, \theta_1$ ) }

# Gradient Descent



# Dummy Variables

- A dummy variable is the one that takes only the value 0 or 1 to indicate the absence or presence of some categorical effect that may be expected to shift the outcome.



# Example:

Employee	Salary	Gender	Experience
1	7.5	Male	6
2	8.6	Male	10
3	9.1	Male	12
4	10.3	Male	18
5	13	Male	30
6	6.2	Female	5
7	8.7	Female	13
8	9.4	Female	15
9	9.8	Female	21

# Example:

Employee	Salary	Gender	Experience
1	7.5	0	6
2	8.6	0	10
3	9.1	0	12
4	10.3	0	18
5	13	0	30
6	6.2	1	5
7	8.7	1	13
8	9.4	1	15
9	9.8	1	21

# Example:

- Usually, the dummy variables take on the values 0 and 1 in order to identify the mutually exclusive classes of the explanatory variables.

For example,

$$D = \begin{cases} 1 & \text{if person is male} \\ 0 & \text{if person is female,} \end{cases}$$

$$D = \begin{cases} 1 & \text{if person is employed} \\ 0 & \text{if person is unemployed.} \end{cases}$$

# Example:

Consider the following model with  $X_1$  as quantitative and  $D_2$  as a dummy variable

$$y = \beta_0 + \beta_1 x_1 + \beta_2 D_2 + \varepsilon, E(\varepsilon) = 0, Var(\varepsilon) = \sigma^2$$

$$D_2 = \begin{cases} 0 & \text{if an observation belongs to group } A \\ 1 & \text{if an observation belongs to group } B. \end{cases}$$

The interpretation of the results is essential. We proceed as follows:

If  $D_2 = 0$ , then

$$\begin{aligned} y &= \beta_0 + \beta_1 x_1 + \beta_2 \cdot 0 + \varepsilon \\ &= \beta_0 + \beta_1 x_1 + \varepsilon \\ E(y / D_2 = 0) &= \beta_0 + \beta_1 x_1 \end{aligned}$$

# Example:

Where the relationship with intercept  $\beta_0$  and slope  $\beta_1$  is a straight line.

If  $D_2 = 1$ , then

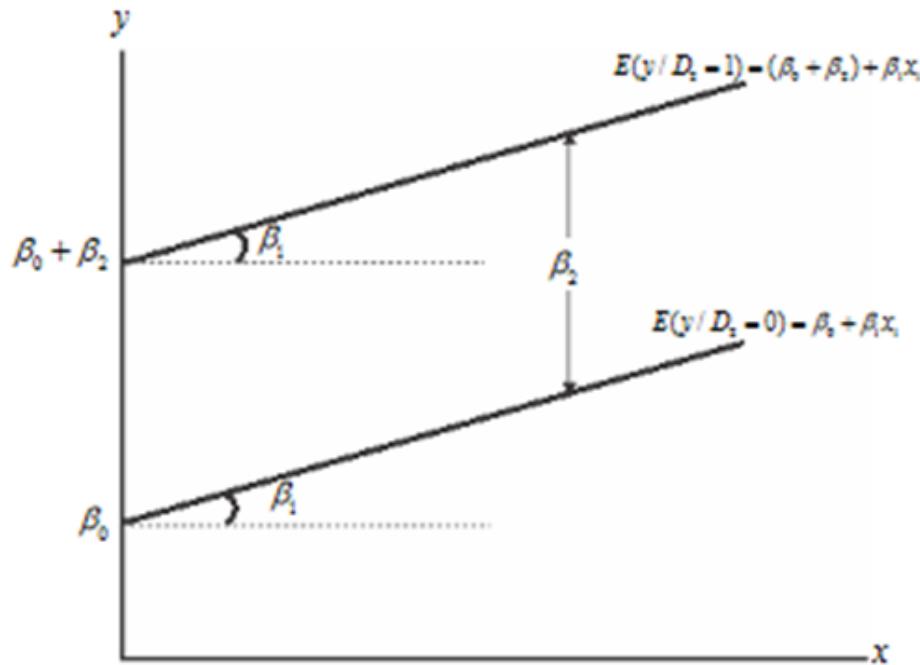
$$\begin{aligned}y &= \beta_0 + \beta_1 x_1 + \beta_2 \cdot 1 + \varepsilon \\&= (\beta_0 + \beta_2) + \beta_1 x_1 + \varepsilon \\E(y / D_2 = 1) &= (\beta_0 + \beta_2) + \beta_1 x_1\end{aligned}$$

Where the relationship with intercept  $(\beta_0 + \beta_2)$  and slope  $\beta_1$  is a straight line.

The quantities  $E(y / D_2 = 0)$  and  $E(y / D_1 = 0)$  are the average responses, when an observation belongs to the groups A and B, respectively. Thus

$$\beta_2 = E(y / D_2 = 1) - E(y / D_2 = 0)$$

# Example:



- If there are three explanatory variables in the model with two dummy variables  $D_2$ , and  $D_3$  then they will describe three levels, e.g., groups A, B, and C.
- In such case, the levels of dummy variables will be as follows:
  1.  $D_2 = 0, D_3 = 0$  if the observation is from group A
  2.  $D_2 = 1, D_3 = 0$  if the observation is from group B
  3.  $D_2 = 0, D_3 = 1$  if the observation is from group C

The concerned regression model is given as:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 D_2 + \beta_3 D_3 + \varepsilon, E(\varepsilon) = 0, \text{var}(\varepsilon) = \sigma^2.$$

# Example:

- Suppose the symbol “Y” denotes the monthly salary of a person and “D” denotes whether the person is a graduate or non-graduate.

The model is

$$y = \beta_0 + \beta_1 D + \varepsilon, E(\varepsilon) = 0, \text{var}(\varepsilon) = \sigma^2.$$

- With n number of observations, the model is

$$y_i = \beta_0 + \beta_1 D_i + \varepsilon_i, i = 1, 2, \dots, n$$

$$E(y_i | D_i = 0) = \beta_0$$

$$E(y_i | D_i = 1) = \beta_0 + \beta_1$$

$$\beta_1 = E(y_i | D_i = 1) - E(y_i | D_i = 0)$$

Now consider the same model with two dummy variables, as defined in the following way:

$$D_{i1} = \begin{cases} 1 & \text{if person is graduate} \\ 0 & \text{if person is nongraduate,} \end{cases}$$

$$D_{i2} = \begin{cases} 1 & \text{if person is nongraduate} \\ 0 & \text{if person is graduate.} \end{cases}$$

The model with  $n$  number of observations is

$$y_i = \beta_0 + \beta_1 D_{i1} + \beta_2 D_{i2} + \varepsilon_i, E(\varepsilon_i) = 0, Var(\varepsilon_i) = \sigma^2, i = 1, 2, \dots, n.$$

Then we have

1.  $E[y_i | D_{i1} = 0, D_{i2} = 1] = \beta_0 + \beta_2$  : Average salary of a non-graduate
2.  $E[y_i | D_{i1} = 1, D_{i2} = 0] = \beta_0 + \beta_1$  : Average salary of a graduate
3.  $E[y_i | D_{i1} = 0, D_{i2} = 0] = \beta_0$  : cannot exist
4.  $E[y_i | D_{i1} = 1, D_{i2} = 1] = \beta_0 + \beta_1 + \beta_2$  : cannot exist.

# Example:

Notice that in this case

$$D_{i1} + D_{i2} = 1 \text{ for all } i$$

Which is an exact constraint and indicates the contradiction in the analysis as follows:

$$D_{i1} + D_{i2} = 1 \Rightarrow \text{person is graduate}$$

$$D_{i1} + D_{i2} = 1 \Rightarrow \text{person is non-graduate}$$

# Example:

However, if the intercept term is ignored, then the model becomes

$$y_i = \beta_1 D_{i1} + \beta_2 D_{i2} + \varepsilon_i, E(\varepsilon_i) = 0, Var(\varepsilon_i) = \sigma^2, i = 1, 2, \dots, n$$

Then

$$E(y_i / D_{i1} = 1, D_{i2} = 0) = \beta_1 \Rightarrow \text{Average salary of a graduate.}$$

$$E(y_i / D_{i1} = 0, D_{i2} = 1) = \beta_2 \Rightarrow \text{Average salary of a non-graduate.}$$

# Interaction term:

- Suppose a model has two explanatory variables – one is quantitative and the other is a dummy.
- Suppose both variables interact and we add an explanatory variable as the interaction of them to the model.

$$y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 D_{i2} + \beta_3 x_{i1}D_{i2} + \varepsilon_i, E(\varepsilon_i) = 0, Var(\varepsilon_i) = \sigma^2, i = 1, 2, \dots, n.$$

$$D_{i2} = \begin{cases} 1 & \text{if } i^{\text{th}} \text{ person belongs to group } A \\ 0 & \text{if } i^{\text{th}} \text{ person belongs to group } B \end{cases}$$

$y_i$  = Salary of  $i^{\text{th}}$  person.

Then,

$$\begin{aligned} E(y_i / D_{i2} = 0) &= \beta_0 + \beta_1 x_{i1} + \beta_2 \cdot 0 + \beta_3 x_{i1} \cdot 0 \\ &= \beta_0 + \beta_1 x_{i1}. \end{aligned}$$

This is a straight line with intercept  $\beta_0$  and slope  $\beta_1$ . Next, we get

$$\begin{aligned} E(y_i / D_{i2} = 1) &= \beta_0 + \beta_1 x_{i1} + \beta_2 \cdot 1 + \beta_3 x_{i1} \cdot 1 \\ &= (\beta_0 + \beta_2) + (\beta_1 + \beta_3) x_{i1}. \end{aligned}$$

This is a straight line with the intercept term  $(\beta_0 + \beta_2)$  and slope  $(\beta_1 + \beta_3)$ .

Hence, we obtain this model,

$$E(y_i) = \beta_0 + \beta_1 x_{i1} + \beta_2 D_{i2} + \beta_3 x_{i1} D_{i2}$$

Which has different slopes and intercept terms.

# Interaction term:

Fitting of the model results in,

$$y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 D_{i2} + \beta_3 x_{i1} D_{i2} + \varepsilon_i$$

Which is equivalent to fitting two separate regression models corresponding to  $D_{i2} = 1$  and  $D_{i2} = 0$ , i.e.

$$y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 \cdot 1 + \beta_3 x_{i1} \cdot 1 + \varepsilon_i$$

$$y_i = (\beta_0 + \beta_2) + (\beta_1 + \beta_3) x_{i1} D_{i2} + \varepsilon_i$$

$$y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 \cdot 0 + \beta_3 x_{i1} \cdot 0 + \varepsilon_i$$

$$y_i = \beta_0 + \beta_1 x_{i1} + \varepsilon_i$$

# Interaction term:

In this way, using a dummy variable makes the testing of hypothesis more convenient. For example, if we want to test whether the two regression models are identical, the test of hypothesis involves testing of

$$H_0 : \beta_2 = \beta_3 = 0$$

$$H_1 : \beta_2 \neq 0 \text{ and/or } \beta_3 \neq 0.$$

Here, the acceptance of  $H_0$  indicates that only a single model is necessary to explain the relationship.

In another example, if our objective is to test whether the two models differ with respect to intercepts only and have the same slopes, then we will test this hypothesis using

$$H_0 : \beta_3 = 0$$

$$H_1 : \beta_3 \neq 0.$$

# Dummy variables vs quantitative explanatory variable

- The quantitative explanatory variable can be converted into dummy variables.
- For example, if the ages of people in certain groups are as follows:

Group 1: 1 day to 3 years

Group 2: 3 years to 8 years

Group 3: 8 years to 12 years

Group 4: 12 years to 17 years

Group 5: 17 years to 25 years

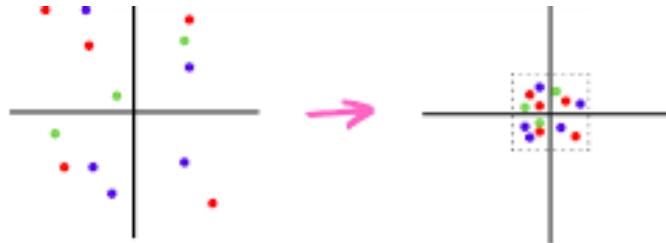
# Dummy variables vs quantitative explanatory variable

- For example, if the ages are in years 2, 3, 4, 5, 6, 7, then the dummy variable is defined as

$$D_i = \begin{cases} 1 & \text{if age of } i^{\text{th}} \text{ person is } > 5 \text{ years} \\ 0 & \text{if age of } i^{\text{th}} \text{ person is } \leq 5 \text{ years.} \end{cases}$$

# Feature Scaling

- Feature scaling is a method used to normalize the range of independent variables or features of a dataset.



# Normalization

- Normalization is a scaling technique in which values are shifted and rescaled, so that they end up in a range from values 0 and 1, This technique is also known as Min-Max scaling.
- Here is the formula for normalization,

$$X' = \frac{X - X_{min}}{X_{max} - X_{min}}$$

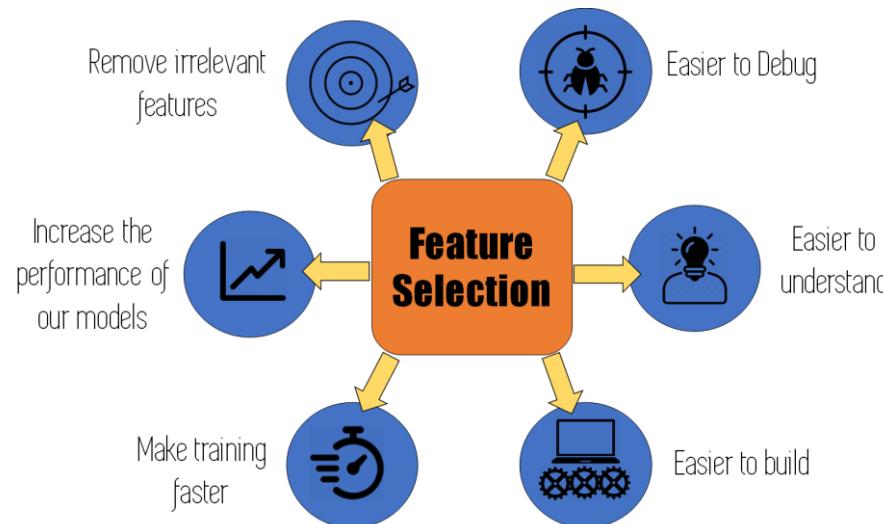
# Standardization

- Standardization is another scaling technique, in which the values are centered around the mean value with a unit standard deviation.
- This implies that the mean value of the attribute becomes zero and the resultant distribution has a unit standard deviation.
- Here is the formula for standardization:

$$X' = \frac{X - \mu}{\sigma}$$

# Feature Selection

- It is the process of identifying and selecting a subset of input variables that are most relevant to the target variable



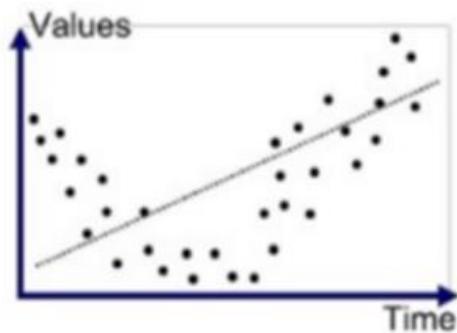
# Underfitting and Overfitting

## Underfitting

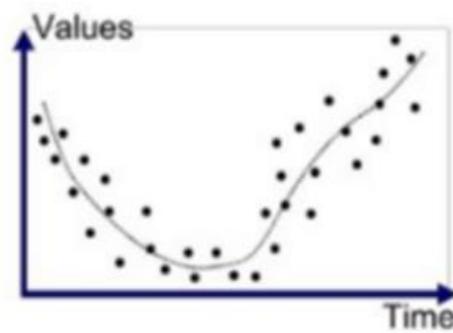
- A statistical model is said to have Underfitting when it cannot capture the underlying trend of the data.
- For example, what if I send a 3rd grade kid to a Differential Calculus Class; the kid is only familiar with the basic arithmetic operations.
- On a similar analogy, If the data contains too much information that the model cannot take, the model is likely going to underfit.

# Underfitting and Overfitting

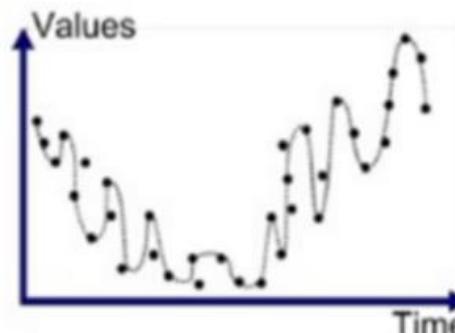
## Overfitting



Underfitted



Good Fit/Robust

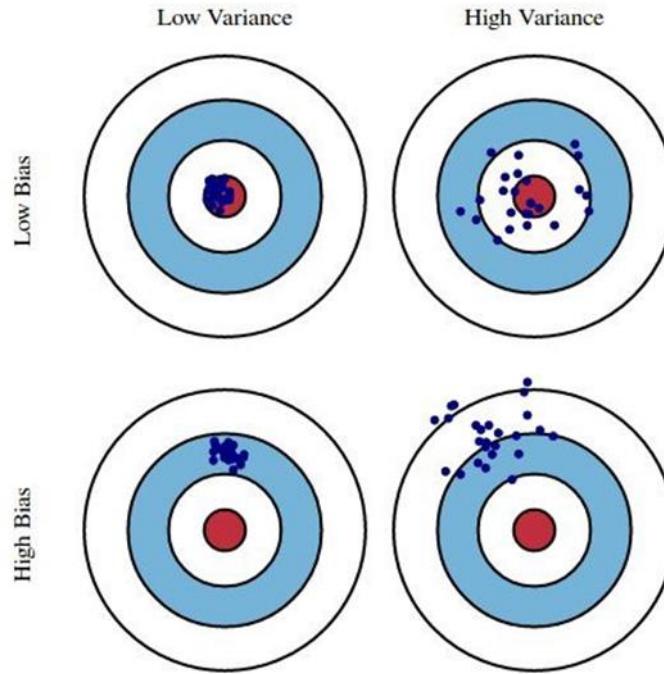


Overfitted

# Bias and Variance Tradeoff

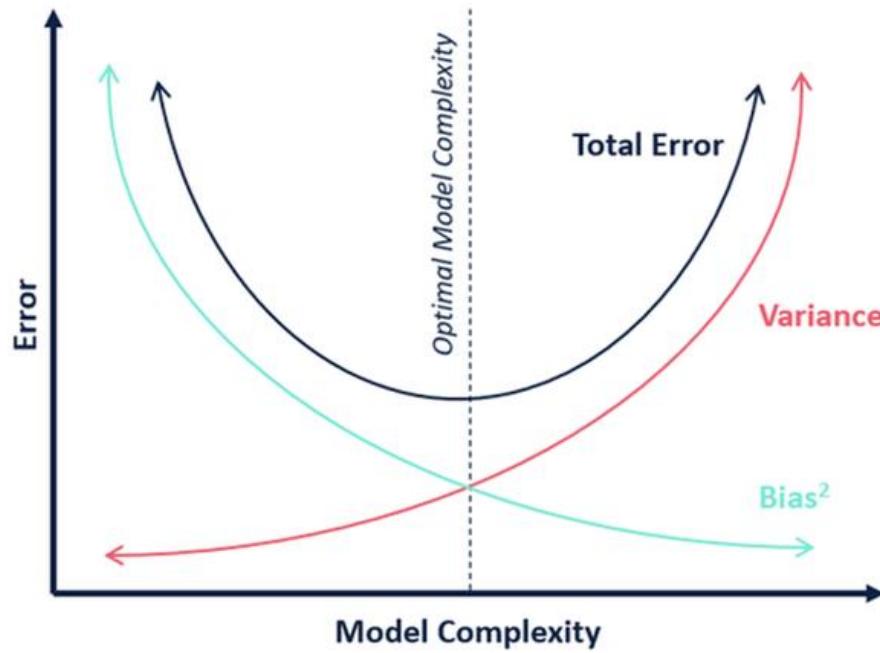
- Bias: This part of the generalization error is due to wrong assumptions.
  - For example, assuming that the data is linear, when it is actually quadratic. A high-bias model is most likely to underfit the training data.
- Variance: This part is due to the model's excessive sensitivity to small variations in the training data.
- Irreducible error: This part of generalization error is due to the noisiness of the data itself.
  - The only way to reduce this part of error is to clean the data (e.g., fix the data sources, such as broken sensors, or detect and remove the outliers).

# Bias and Variance Tradeoff



Bias and variance bulls-eye diagram.

# Bias and Variance Tradeoff



# Regularization

- Simple linear regression is given as:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \dots + \beta_P x_P$$

- Using the OLS method, we try to minimize the cost function given as:

$$\text{RSS} = \sum_{i=1}^n \left( y_i - \beta_0 - \sum_{j=1}^p \beta_j x_{ij} \right)^2.$$

# Ridge Regression (L2 Form):

- Ridge regression is a technique used when the data suffers from multicollinearity (independent variables are highly correlated).
- Ridge regression solves the multicollinearity problem through shrinking parameter  $\lambda$ , Look at the equation below

$$= \underset{\beta \in \mathbb{R}^p}{\operatorname{argmin}} \underbrace{\|y - X\beta\|_2^2}_{\text{Loss}} + \lambda \underbrace{\|\beta\|_2^2}_{\text{Penalty}}$$

# LASSO (L1 form):

- The LASSO (Least Absolute Shrinkage and Selection Operator) regression penalizes the model based on the sum of magnitude of the coefficients.
- The regularization term is given by the following equation:

$$\text{regularization} = \lambda * \sum |\beta_j|$$

Where,  $\lambda$  is the shrinkage factor and hence the formula for loss after regularization is:

$$= \operatorname{argmin}_{\beta \in \mathbb{R}^p} \underbrace{\|y - X\beta\|_2^2}_{\text{Loss}} + \lambda \underbrace{\|\beta\|_1}_{\text{Penalty}}$$

# ElasticNet Regression

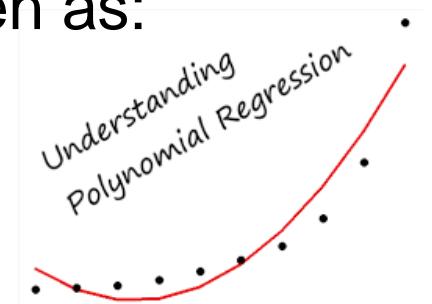
- ElasticNet is a hybrid of LASSO and ridge regression techniques.
- It is trained with L1 and L2 prior to regularization.
- ElasticNet is useful when there are multiple correlated features.
- LASSO is likely to pick one of these at random, while ElasticNet is likely to pick both.

$$\hat{\beta} = \underset{\beta}{\operatorname{argmin}} (\|y - X\beta\|^2 + \lambda_2 \|\beta\|^2 + \lambda_1 \|\beta\|_1).$$

# Polynomial Regression

- Simply said, poly means many, Therefore, a polynomial is an aggregation of many monomials (or Variables).
- A simple polynomial equation can be written as:

$$y = a + bx + cx^2 + \dots + nx^n + \dots$$



# Polynomial Regression

- Polynomial Regression can be defined as a mechanism to predict a dependent variable based on the polynomial relationship with the independent variable.

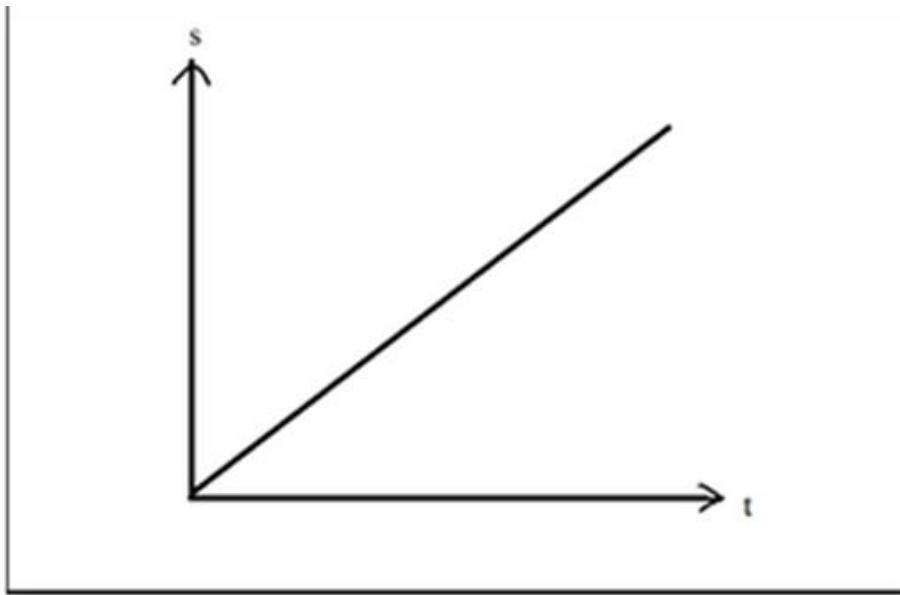
The maximum power of 'X' is called the *degree* of the polynomial equation. For example, if the degree is 1, the equation becomes

$$y = a + bx$$

This is a simple linear equation. If the degree is 2, then the polynomial equation becomes:

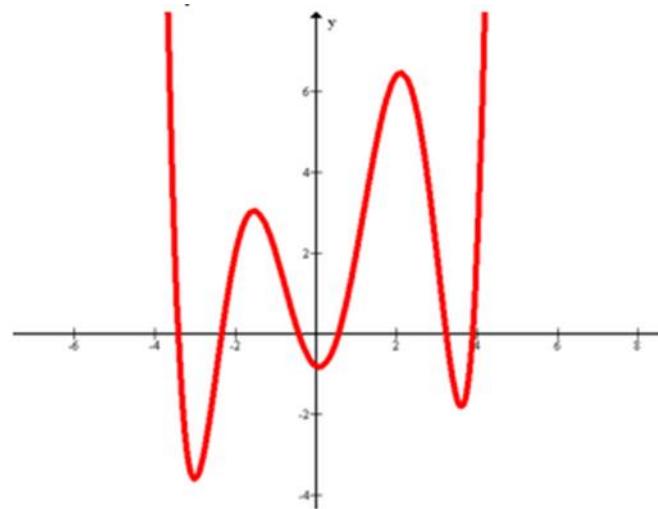
$$y = a + bx + cx^2$$

# When to use Polynomial Regression?



# When to use Polynomial Regression?

- What if the relationship looks like this?



# When to use Polynomial Regression?

$$\begin{bmatrix} n & \sum_{i=0}^n x_i & \sum_{i=0}^n x_i^2 & \cdots & \sum_{i=0}^n x_i^m \\ \sum_{i=0}^n x_i & \sum_{i=0}^n x_i^2 & \sum_{i=0}^n x_i^3 & \cdots & \sum_{i=0}^n x_i^{(m+1)} \\ \sum_{i=0}^n x_i^2 & \sum_{i=0}^n x_i^3 & \sum_{i=0}^n x_i^4 & \cdots & \sum_{i=0}^n x_i^{(m+2)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \sum_{i=0}^n x_i^m & \sum_{i=0}^n x_i^{(m+1)} & \sum_{i=0}^n x_i^{(m+2)} & \cdots & \sum_{i=0}^n x_i^{2m} \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_m \end{bmatrix} = \begin{bmatrix} \sum_{i=0}^n y_i \\ \sum_{i=0}^n x_i y_i \\ \sum_{i=0}^n x_i^2 y_i \\ \vdots \\ \sum_{i=0}^n x_i^m y_i \end{bmatrix}$$

# Complete Lab 6

# Summary

In this lesson, we have learned the following:

- Cost function optimization technique
- Categorical Features in a Regression model
- Feature Scaling
- Feature Selection
- Regularization
- Polynomial Regression



# **"Complete Assessment & Programming Assignment "**

# DAY 4



# 7: Working with Pandas DataFrames



# Working with Pandas DataFrames

The following topics will be covered in this lesson:



- Pandas data structures
- Creating DataFrame objects from files, API requests, SQL queries, and other Python objects
- Inspecting DataFrame objects and calculating summary statistics
- Grabbing subsets of the data via selection, slicing, indexing, and filtering
- Adding and removing data

# lesson materials

- The files we will be working with in this lesson can be found in the GitHub repository at <https://github.com/fenago/machine-learning-essentials-module1>.
- We will be working with earthquake data from the US Geological Survey (USGS) by using the USGS API and CSV files, which can be found in the data/ directory.



# Pandas data structures

- The aforementioned data structures are implemented as Python classes; when we actually create one, they are referred to as objects or instances.
- This is an important distinction, since, as we will see, some actions can be performed using the object itself (a method), whereas others will require that we pass our object in as an argument to some function.

# Pandas data structures

- Note that this is the only time we will use NumPy to read in a file and that this is being done for illustrative purposes only; the important part is to look at the way the data is represented with NumPy:

```
>>> import numpy as np
>>> data = np.genfromtxt(
...     'data/example_data.csv', delimiter=';',
...     names=True, dtype=None, encoding='UTF'
... )
>>> data
array([('2018-10-13 11:10:23.560',
       '262km NW of Ozernovskiy, Russia',
       'mww', 6.7, 'green', 1),
       ('2018-10-13 04:34:15.580',
       '25km E of Bitung, Indonesia', 'mww', 5.2, 'green', 0),
       ('2018-10-13 00:13:46.220',
       '42km WNW of Sola, Vanuatu',
       'mww', 5.7, 'green', 0),
       ('2018-10-12 21:09:49.240',
       '13km E of Nueva Concepcion, Guatemala',
       'mww', 5.7, 'green', 0),
       ('2018-10-12 02:52:03.620',
       '128km SE of Kimbe, Papua New Guinea',
       'mww', 5.6, 'green', 1)],
      dtype=[('time', '<U23'), ('place', '<U37'),
             ('magType', '<U3'), ('mag', '<f8'),
             ('alert', '<U5'), ('tsunami', '<i8')])
```

# Pandas data structures

- We now have our data in a NumPy array.
- Using the `shape` and `dtype` attributes, we can gather information about the dimensions of the array and the data types it contains, respectively:

```
>>> data.shape  
(5,)  
>>> data.dtype  
dtype([('time', '<U23'), ('place', '<U37'), ('magType', '<U3'),  
       ('mag', '<f8'), ('alert', '<U5'), ('tsunami', '<i8')])
```

# Pandas data structures

- We can use the %%%timeit magic command from IPython (a special command preceded by %) to see how long this implementation takes (times will vary):

```
>>> %%%timeit  
>>> max([row[3] for row in data])  
9.74 µs ± 177 ns per loop  
(mean ± std. dev. of 7 runs, 100000 loops each)
```



# Pandas data structures

- Again, the important part here is how the data is now represented using NumPy:

```
>>> array_dict = {
...     col: np.array([row[i] for row in data])
...     for i, col in enumerate(data.dtype.names)
... }
>>> array_dict
{'time': array(['2018-10-13 11:10:23.560',
   '2018-10-13 04:34:15.580', '2018-10-13 00:13:46.220',
   '2018-10-12 21:09:49.240', '2018-10-12 02:52:03.620'],
  dtype='<U23'),
 'place': array(['262km NW of Ozernovskiy, Russia',
    '25km E of Bitung, Indonesia',
    '42km WNW of Sola, Vanuatu',
    '13km E of Nueva Concepcion, Guatemala',
    '128km SE of Kimbe, Papua New Guinea'], dtype='<U37'),
 'magType': array(['mww', 'mww', 'mww', 'mww', 'mww'],
   dtype='<U3'),
 'mag': array([6.7, 5.2, 5.7, 5.7, 5.6]),
 'alert': array(['green', 'green', 'green', 'green', 'green'],
   dtype='<U5'),
 'tsunami': array([1, 0, 0, 0, 1])}
```

# Pandas data structures

- Grabbing the maximum magnitude is now simply a matter of selecting the mag key and calling the max() method on the NumPy array.
- This is nearly twice as fast as the list comprehension implementation, when dealing with just five entries—imagine how much worse the first attempt will perform on large datasets:

```
>>> %%timeit  
>>> array_dict['mag'].max()  
5.22 µs ± 100 ns per loop  
(mean ± std. dev. of 7 runs, 100000 loops each)
```

# Pandas data structures

- The result is now a NumPy array of strings (our numeric values were converted), and we are now in the format that we saw earlier:

```
>>> np.array([  
...     value[array_dict['mag'].argmax()]  
...     for key, value in array_dict.items()  
... ])  
array(['2018-10-13 11:10:23.560',  
       '262km NW of Ozernovskiy, Russia',  
       'mww', '6.7', 'green', '1'], dtype='|<U31')
```



# Series

- The Series class provides a data structure for arrays of a single type, just like the NumPy array.
- However, it comes with some additional functionality.
- This one-dimensional representation can be thought of as a column in a spreadsheet.

# Series

- We have a name for our column, and the data we hold in it is of the same type (since we are measuring the same variable):

```
>>> import pandas as pd  
>>> place = pd.Series(array_dict['place'], name='place')  
>>> place  
0      262km NW of Ozernovskiy, Russia  
1      25km E of Bitung, Indonesia  
2      42km WNW of Sola, Vanuatu  
3      13km E of Nueva Concepcion, Guatemala  
4      128km SE of Kimbe, Papua New Guinea  
Name: place, dtype: object
```

# Series

- To access attributes of the Series object, we use attribute notation of the form `<object>.<attribute_name>`.
- The following are some common attributes we will access.
- Notice that `dtype` and `shape` are available, just as we saw with the NumPy array:

Attribute	Returns
<code>name</code>	The name of the <code>Series</code> object
<code>dtype</code>	The data type of the <code>Series</code> object
<code>shape</code>	Dimensions of the <code>Series</code> object in a tuple of the form <code>(number of rows,)</code>
<code>index</code>	The <code>Index</code> object that is part of the <code>Series</code> object
<code>values</code>	The data in the <code>Series</code> object

# Index

- The addition of the Index class makes the Series class significantly more powerful than a NumPy array.
- The Index class gives us row labels, which enable selection by row. Depending on the type, we can provide a row number, a date, or even a string to select our row.
- It plays a key role in identifying entries in the data and is used for a multitude of operations in pandas, as we will see throughout this course.

# Index

- We can access the index through the `index` attribute:

```
>>> place_index = place.index  
>>> place_index  
RangeIndex(start=0, stop=5, step=1)
```

# Index

- As with Series objects, we can access the underlying data via the values attribute.
- Note that this Index object is built on top of a NumPy array:

```
>>> place_index.values  
array([0, 1, 2, 3, 4], dtype=int64)
```

# Index

- Some of the useful attributes of Index objects include the following:

Attribute	Returns
<code>name</code>	The name of the <code>Index</code> object
<code>dtype</code>	The data type of the <code>Index</code> object
<code>shape</code>	Dimensions of the <code>Index</code> object
<code>values</code>	The data in the <code>Index</code> object
<code>is_unique</code>	Check if the <code>Index</code> object has all unique values

# Index

- Both NumPy and pandas support arithmetic operations, which will be performed element-wise.
- NumPy will use the position in the array for this:

```
>>> np.array([1, 1, 1]) + np.array([-1, 0, 1])
array([0, 1, 2])
```

# Index

- In Data Wrangling with Pandas, we will discuss some ways to change and align the index so that we can perform these types of operations without losing data:

```
>>> numbers = np.linspace(0, 10, num=5) # [0, 2.5, 5, 7.5, 10]
>>> x = pd.Series(numbers) # index is [0, 1, 2, 3, 4]
>>> y = pd.Series(numbers, index=pd.Index([1, 2, 3, 4, 5]))
>>> x + y
0      NaN
1    2.5
2    7.5
3   12.5
4   17.5
5      NaN
dtype: float64
```

# DataFrame

- With the Series class, we essentially had columns of a spreadsheet, with the data all being of the same type.
- The DataFrame class builds upon the Series class and can have many columns, each with its own data type; we can think of it as representing the spreadsheet as a whole.
- We can turn either of the NumPy representations we built from the example data into a DataFrame object:

```
>>> df = pd.DataFrame(array_dict)  
>>> df
```

# DataFrame

- In this case, it is just the row number, but we could easily use the time column for this, which would enable some additional pandas features, as we will see in Aggregating Pandas DataFrames:

	time	place	magType	mag	alert	tsunami
0	2018-10-13 11:10:23.560	262km NW of Ozernovskiy, Russia	mww	6.7	green	1
1	2018-10-13 04:34:15.580	25km E of Bitung, Indonesia	mww	5.2	green	0
2	2018-10-13 00:13:46.220	42km WNW of Sola, Vanuatu	mww	5.7	green	0
3	2018-10-12 21:09:49.240	13km E of Nueva Concepcion, Guatemala	mww	5.7	green	0
4	2018-10-12 02:52:03.620	128km SE of Kimbe, Papua New Guinea	mww	5.6	green	1

# DataFrame

- Our columns each have a single data type, but they don't all share the same data type:

```
>>> df.dtypes  
time    object  
place   object  
magType  object  
mag     float64  
alert   object  
tsunami int64  
dtype: object
```

# DataFrame

- The values of the dataframe look very similar to the initial NumPy representation we had:

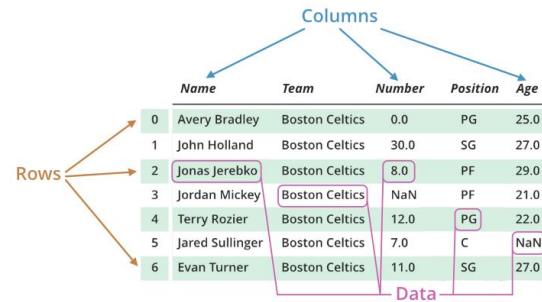
```
>>> df.values
array([['2018-10-13 11:10:23.560',
       '262km NW of Ozernovskiy, Russia',
       'mww', 6.7, 'green', 1],
      ['2018-10-13 04:34:15.580',
       '25km E of Bitung, Indonesia', 'mww', 5.2, 'green', 0],
      ['2018-10-13 00:13:46.220', '42km WNW of Sola, Vanuatu',
       'mww', 5.7, 'green', 0],
      ['2018-10-12 21:09:49.240',
       '13km E of Nueva Concepcion, Guatemala',
       'mww', 5.7, 'green', 0],
      ['2018-10-12 02:52:03.620', '128 km SE of Kimbe,
       Papua New Guinea', 'mww', 5.6, 'green', 1]],
     dtype=object)
```

# DataFrame

- We can access the column names via the columns attribute.
- Note that they are actually stored in an Index object as well:

```
>>> df.columns
```

```
Index(['time', 'place', 'magType', 'mag', 'alert', 'tsunami'],  
      dtype='object')
```



# DataFrame

- The following are some commonly used dataframe attributes:

Attribute	Returns
<code>dtypes</code>	The data types of each column
<code>shape</code>	Dimensions of the <code>DataFrame</code> object in a tuple of the form (number of rows, number of columns)
<code>index</code>	The <code>Index</code> object along the rows of the <code>DataFrame</code> object
<code>columns</code>	The name of the columns (as an <code>Index</code> object)
<code>values</code>	The data in the <code>DataFrame</code> object
<code>empty</code>	Check if the <code>DataFrame</code> object is empty

# DataFrame

- Note that we can also perform arithmetic on dataframes.
- For example, we can add df to itself, which will sum the numeric columns and concatenate the string columns:

```
>>> df + df
```

# DataFrame

- Pandas will only perform the operation when both the index and column match.
- Here, pandas concatenated the string columns (time, place, magType, and alert) across dataframes.
- The numeric columns (mag and tsunami) were summed:

	time	place	magType	mag	alert	tsunami
0	2018-10-13 11:10:23.5602018-10-13 11:10:23.560	262km NW of Ozernovskiy, Russia262km NW of Oze...	mwwmww	13.4	green	2
1	2018-10-13 04:34:15.5802018-10-13 04:34:15.580	25km E of Bitung, Indonesia25km E of Bitung, I...	mwwmww	10.4	green	0
2	2018-10-13 00:13:46.2202018-10-13 00:13:46.220	42km WNW of Sola, Vanuatu42km WNW of Sola, Van...	mwwmww	11.4	green	0
3	2018-10-12 21:09:49.2402018-10-12 21:09:49.240	13km E of Nueva Concepcion, Guatemala13km E of...	mwwmww	11.4	green	0
4	2018-10-12 02:52:03.6202018-10-12 02:52:03.620	128km SE of Kimbe, Papua New Guinea128km SE of...	mwwmww	11.2	green	2

# Creating a pandas DataFrame

- Let's now turn to the next notebook, 2-creating\_dataframes.ipynb, and import the packages we will need for the upcoming examples.
- We will be using datetime from the Python standard library, along with the third-party packages numpy and pandas:

```
>>> import datetime as dt  
>>> import numpy as np  
>>> import pandas as pd
```

# From a Python object

- To ensure that the result is reproducible, we will set the seed here.
- The seed gives a starting point for the generation of pseudorandom numbers.
- No algorithms for random number generation are truly random—they are deterministic, so by setting this starting point, the numbers that are generated will be the same each time the code is run.



# From a Python object

- This is good for testing things, but not for simulation (where we want randomness), which we will look at in Rule-Based Anomaly Detection.
- In this fashion, we can make a Series object with any list-like structure (such as NumPy arrays):

```
>>> np.random.seed(0) # set a seed for reproducibility
>>> pd.Series(np.random.rand(5), name='random')
0    0.548814
1    0.715189
2    0.602763
3    0.544883
4    0.423655
Name: random, dtype: float64
```



# From a Python object

- Since columns can all be different data types, let's get a little fancy with this example.

We are going to create a DataFrame object containing three columns, with five observations each:

- random: Five random numbers between 0 and 1 as a NumPy array
- text: A list of five strings or None
- truth: A list of five random Booleans

# From a Python object

- All we have to do is package the columns in a dictionary using the desired column names as the keys and pass this in when we call the pd.DataFrame() constructor.
- The index gets passed as the index argument:

```
>>> np.random.seed(0) # set seed so result is reproducible
>>> pd.DataFrame(
...     {
...         'random': np.random.rand(5),
...         'text': ['hot', 'warm', 'cool', 'cold', None],
...         'truth': [np.random.choice([True, False])
...                   for _ in range(5)]
...     },
...     index=pd.date_range(
...         end=dt.date(2019, 4, 21),
...         freq='1D', periods=5, name='date'
...     )
... )
```

# From a Python object

- Having dates in the index makes it easy to select entries by date (or even in a date range), as we will see in Data Wrangling with Pandas:

	random	text	truth
	date		
2019-04-17	0.548814	hot	False
2019-04-18	0.715189	warm	True
2019-04-19	0.602763	cool	True
2019-04-20	0.544883	cold	False
2019-04-21	0.423655	None	True

# From a Python object

- In cases where the data isn't a dictionary, but rather a list of dictionaries, we can still use pd.DataFrame().
- Data in this format is what we would expect when consuming from an API.
- Each entry in the list will be a dictionary, where the keys of the dictionary are the column names and the values of the dictionary are the values for that column at that index:

```
>>> pd.DataFrame([
...     {'mag': 5.2, 'place': 'California'},
...     {'mag': 1.2, 'place': 'Alaska'},
...     {'mag': 0.2, 'place': 'California'},
... ])
```

# From a Python object

- This gives us a dataframe of three rows (one for each entry in the list) with two columns (one for each key in the dictionaries):

	<b>mag</b>	<b>place</b>
<b>0</b>	5.2	California
<b>1</b>	1.2	Alaska
<b>2</b>	0.2	California

# From a Python object

- In fact, pd.DataFrame() also works for lists of tuples.
- Note that we can also pass in the column names as a list through the columns argument:

```
>>> list_of_tuples = [(n, n**2, n**3) for n in range(5)]
>>> list_of_tuples
[(0, 0, 0), (1, 1, 1), (2, 4, 8), (3, 9, 27), (4, 16, 64)]
>>> pd.DataFrame(
...     list_of_tuples,
...     columns=['n', 'n_squared', 'n_cubed']
... )
```



# From a Python object

- Each tuple is treated like a record and becomes a row in the dataframe:

	<b>n</b>	<b>n_squared</b>	<b>n_cubed</b>
<b>0</b>	0	0	0
<b>1</b>	1	1	1
<b>2</b>	2	4	8
<b>3</b>	3	9	27
<b>4</b>	4	16	64

# From a Python object

- We also have the option of using pd.DataFrame() with NumPy arrays:

```
>>> pd.DataFrame(  
...     np.array([  
...         [0, 0, 0],  
...         [1, 1, 1],  
...         [2, 4, 8],  
...         [3, 9, 27],  
...         [4, 16, 64]  
...     ]), columns=['n', 'n_squared', 'n_cubed'])  
... )
```

# From a file

- The data we want to analyze will most often come from outside Python.
- In many cases, we may obtain a data dump from a database or website and bring it into Python to sift through it.
- A data dump gets its name from containing a large amount of data (possibly at a very granular level) and often not discriminating against any of it initially; for this reason, they can be unwieldy.

# From a file

- To check the number of lines, we use the `wc` utility (word count) with the `-l` flag to count the number of lines.
- We have 9,333 rows in the file:

```
>>> !wc -l data/earthquakes.csv  
9333 data/earthquakes.csv
```



# From a file

Now, let's check the file's size.

- For this task, we will use ls on the data directory.
- This will show us the list of files in that directory.
- We can add the -lh flag to get information about the files in a human-readable format.
- Finally, we send this output to the grep utility, which will help us isolate the files we want.
- This tells us that the earthquakes.csv file is 3.4 MB:

```
>>> !ls -lh data | grep earthquakes.csv  
-rw-r--r-- 1 stefanie stefanie 3.4M ... earthquakes.csv
```



# From a file

- Note that IPython also lets us capture the result of the command in a Python variable, so if we aren't comfortable with pipes (|) or grep, we can do the following:

```
>>> files = !ls -lh data  
>>> [file for file in files if 'earthquake' in file]  
['-rw-r--r-- 1 stefanie stefanie 3.4M ... earthquakes.csv']
```



# From a file

- Now, let's take a look at the top few rows to see if the file comes with headers.

```
>>> !head -n 2 data/earthquakes.csv
alert,cdi,code,detail,dmin,felt,gap,ids,mag,magType,mmi,net,nst,place,rms,sig,sources,status,
time,title,tsunami,type,types,tz,updated,url
,,37389218,https://earthquake.usgs.gov/ [...],0.008693,,85.0,",ci37389218,",1.35,ml,,ci,26.0,"9
km NE of Aguanga, CA",0.19,28,",ci,",automatic,1539475168010,"M 1.4 - 9km NE of
Aguanga, CA",0,earthquake,",geoserve,nearby-cities,origin,phase-data,",-
480.0,1539475395144,https://earthquake.usgs.gov/earthquakes/eventpage/ci37389218
```

# From a file

- This will look a little complicated, but this is by no means something we need to memorize:

```
>>> !awk -F',' '{print NF; exit}' data/earthquakes.csv
```

26



# From a file

- Since we know that the first line of the file contains headers and that the file is comma-separated, we can also count the columns by using head to get the headers and Python to parse them:

```
>>> headers = !head -n 1 data/earthquakes.csv
```

```
>>> len(headers[0].split(','))
```

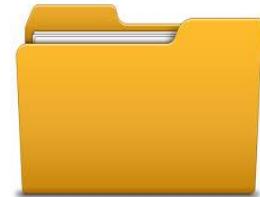
26



# From a file

- To summarize, we now know that the file is 3.4 MB and is comma-delimited with 26 columns and 9,333 rows, with the first one being the header.
- This means that we can use the `pd.read_csv()` function with the defaults:

```
>>> df = pd.read_csv('earthquakes.csv')
```



# From a file

- Note that we aren't limited to reading in data from files on our local machines; file paths can be URLs as well.
- As an example, let's read in the same CSV file from GitHub:

```
>>> df = pd.read_csv(  
...     'https://github.com/fenago/'  
...     'machine-learning-essentials-module1'  
...     '/blob/master/ch_02/data/earthquakes.csv?raw=True'  
... )
```



# From a file

- Pandas is usually very good at figuring out which options to use based on the input data, so we often won't need to add arguments to this call; however, there are many options available should we need them, some of which include the following:

Parameter	Purpose
<code>sep</code>	Specifies the delimiter
<code>header</code>	Row number where the column names are located; the default option has <code>pandas infer</code> whether they are present
<code>names</code>	List of column names to use as the header
<code>index_col</code>	Column to use as the index
<code>usecols</code>	Specifies which columns to read in
<code>dtype</code>	Specifies data types for the columns
<code>converters</code>	Specifies functions for converting data in certain columns
<code>skiprows</code>	Rows to skip
<code>nrows</code>	Number of rows to read at a time (combine with <code>skiprows</code> to read a file bit by bit)
<code>parse_dates</code>	Automatically parse columns containing dates into datetime objects
<code>chunksize</code>	For reading the file in chunks
<code>compression</code>	For reading in compressed files without extracting beforehand
<code>encoding</code>	Specifies the file encoding

# From a file

- We can write our data without the index by passing in `index=False`:

```
>>> df.to_csv('output.csv', index=False)
```



# From a database

- Let's write the tsunami data from the data/tsunamis.csv file to a table in the database called tsunamis, replacing the table if it already exists:

```
>>> import sqlite3  
>>> with sqlite3.connect('data/quakes.db') as connection:  
...     pd.read_csv('data/tsunamis.csv').to_sql(  
...         'tsunamis', connection, index=False,  
...         if_exists='replace'  
...     )
```



# From a database

- To actually query the database, we use `pd.read_sql()`, passing in our query and the database connection:

```
>>> import sqlite3  
>>> with sqlite3.connect('data/quakes.db') as connection:  
...     tsunamis = \  
...         pd.read_sql('SELECT * FROM tsunamis', connection)  
>>> tsunamis.head()
```

# From a database

- We now have the tsunamis data in a dataframe:

	alert	type	title	place	magType	mag	time
0	None	earthquake	M 5.0 - 165km NNW of Flying Fish Cove, Christm...	165km NNW of Flying Fish Cove, Christmas Island	mww	5.0	1539459504090
1	green	earthquake	M 6.7 - 262km NW of Ozernovskiy, Russia	262km NW of Ozernovskiy, Russia	mww	6.7	1539429023560
2	green	earthquake	M 5.6 - 128km SE of Kimbe, Papua New Guinea	128km SE of Kimbe, Papua New Guinea	mww	5.6	1539312723620
3	green	earthquake	M 6.5 - 148km S of Severo-Kuril'sk, Russia	148km S of Severo-Kuril'sk, Russia	mww	6.5	1539213362130
4	green	earthquake	M 6.2 - 94km SW of Kokopo, Papua New Guinea	94km SW of Kokopo, Papua New Guinea	mww	6.2	1539208835130

# From an API

- For this section, we will be working in the 3-making\_dataframes\_from\_api\_requests.ipynb notebook, so we have to import the packages we need once again.
- As with the previous notebook, we need pandas and datetime, but we also need the requests package to make API requests:

```
>>> import datetime as dt  
>>> import pandas as pd  
>>> import requests
```



# From an API

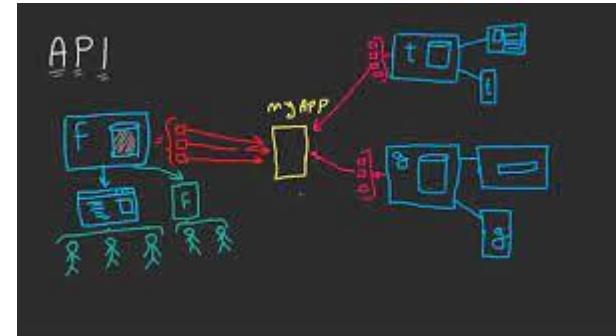
- Note that we are using yesterday as the end of our date range, since the API won't have complete information for today yet:

```
>>> yesterday = dt.date.today() - dt.timedelta(days=1)
>>> api = 'https://earthquake.usgs.gov/fdsnws/event/1/query'
>>> payload = {
...     'format': 'geojson',
...     'starttime': yesterday - dt.timedelta(days=30),
...     'endtime': yesterday
... }
>>> response = requests.get(api, params=payload)
```

# From an API

- A listing of status codes and their meanings can be found at [https://en.wikipedia.org/wiki/List\\_of\\_HTTP\\_status\\_codes](https://en.wikipedia.org/wiki/List_of_HTTP_status_codes).
- A 200 response will indicate that everything is OK:

```
>>> response.status_code  
200
```



# From an API

- We need to isolate the JSON payload from the HTTP response (stored in the response variable), and then look at the keys to view the main sections of the resulting data:

```
>>> earthquake_json = response.json()  
>>> earthquake_json.keys()  
dict_keys(['type', 'metadata', 'features', 'bbox'])
```



# From an API

- While this can certainly be useful, it isn't what we are after right now:

```
>>> earthquake_json['metadata']
{'generated': 1604267813000,
 'url': 'https://earthquake.usgs.gov/fdsnws/event/1/query?
format=geojson&starttime=2020-10-01&endtime=2020-10-31',
 'title': 'USGS Earthquakes',
 'status': 200,
 'api': '1.10.3',
 'count': 13706}
```

# From an API

- The features key looks promising; if this does indeed contain all our data, we should check what type it is so that we don't end up trying to print everything to the screen:

```
>>> type(earthquake_json['features'])  
list
```



# From an API

- For this reason, the following is an example of what an entry looks like:

```
>>> earthquake_json['features'][0]
{'type': 'Feature',
 'properties': {'mag': 1,
                 'place': '50 km ENE of Susitna North, Alaska',
                 'time': 1604102395919, 'updated': 1604103325550, 'tz': None,
                 'url': 'https://earthquake.usgs.gov/earthquakes/eventpage/ak020dz5f85a',
                 'detail': 'https://earthquake.usgs.gov/fdsnws/event/1/query?
eventid=ak020dz5f85a&format=geojson',
                 'felt': None, 'cdi': None, 'mmi': None, 'alert': None,
                 'status': 'reviewed', 'tsunami': 0, 'sig': 15, 'net': 'ak',
                 'code': '020dz5f85a', 'ids': ',ak020dz5f85a,',
                 'sources': ',ak,', 'types': ',origin,phase-data,', 'nst': None,
                 'dmin': None, 'rms': 1.36, 'gap': None,
                 'magType': 'ml', 'type': 'earthquake',
                 'title': 'M 1.0 - 50 km ENE of Susitna North, Alaska'},
 'geometry': {'type': 'Point', 'coordinates': [-148.9807, 62.3533, 5]},
 'id': 'ak020dz5f85a'}
```

# From an API

- We can use a list comprehension to isolate the properties section from each of the dictionaries in the features list:

```
>>> earthquake_properties_data = [  
...     quake['properties']  
...     for quake in earthquake_json['features']  
... ]
```



# From an API

- Finally, we are ready to create our dataframe.
- Pandas knows how to handle data in this format already (a list of dictionaries), so all we have to do is pass in the data when we call pd.DataFrame():

```
>>> df = pd.DataFrame(earthquake_properties_data)
```

# Inspecting a DataFrame object

- Since this is a new notebook, we must once again handle our setup.
- This time, we need to import pandas and numpy, as well as read in the CSV file with the earthquake data:

```
>>> import numpy as np  
>>> import pandas as pd  
>>> df = pd.read_csv('data/earthquakes.csv')
```

# Examining the data

- First, we want to make sure that we actually have data in our dataframe.
- We can check the empty attribute to find out:

```
>>> df.empty  
False
```



# Examining the data

- So far, so good; we have data.
- Next, we should check how much data we read in; we want to know the number of observations (rows) and the number of variables (columns) we have.
- For this task, we use the shape attribute.
- Our data contains 9,332 observations of 26 variables, which matches our initial inspection of the file:

```
>>> df.shape  
(9332, 26)
```

# Examining the data

- Now, let's use the columns attribute to see the names of the columns in our dataset:

```
>>> df.columns  
Index(['alert', 'cdi', 'code', 'detail', 'dmin', 'felt', 'gap',  
       'ids', 'mag', 'magType', 'mmi', 'net', 'nst', 'place',  
       'rms', 'sig', 'sources', 'status', 'time', 'title',  
       'tsunami', 'type', 'types', 'tz', 'updated', 'url'],  
       dtype='object')
```

# Examining the data

- We know the dimensions of our data, but what does it actually look like?
- For this task, we can use the head() and tail() methods to look at the top and bottom rows, respectively.
- This will default to five rows, but we can change this by passing a different number to the method.
- Let's take a look at the first few rows:  
`>>> df.head()`

# Examining the data

- The following are the first five rows we get using head():

	alert	...	dmin	felt	...	mag	magType	...	place	...	time	title	tsunami	...	updated	url
0	NaN	...	0.008693	NaN	...	1.35	ml	...	9km NE of Aguanga, CA	...	1539475168010	M 1.4 - 9km NE of Aguanga, CA	0	...	1539475395144	https...
1	NaN	...	0.020030	NaN	...	1.29	ml	...	9km NE of Aguanga, CA	...	1539475129610	M 1.3 - 9km NE of Aguanga, CA	0	...	1539475253925	https...
2	NaN	...	0.021370	28.0	...	3.42	ml	...	8km NE of Aguanga, CA	...	1539475062610	M 3.4 - 8km NE of Aguanga, CA	0	...	1539536756176	https...
3	NaN	...	0.026180	NaN	...	0.44	ml	...	9km NE of Aguanga, CA	...	1539474978070	M 0.4 - 9km NE of Aguanga, CA	0	...	1539475196167	https...
4	NaN	...	0.077990	NaN	...	2.16	md	...	10km NW of Avenal, CA	...	1539474716050	M 2.2 - 10km NW of Avenal, CA	0	...	1539477547926	https...

# Examining the data

- To get the last two rows, we use the tail() method and pass 2 as the number of rows:

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



# Examining the data

- The following is the result:

	alert	...	dmin	felt	...	mag	magType	...	place	...	time	title	tsunami	...	updated	url
<b>9330</b>	NaN	...	0.01865	NaN	...	1.10	ml	...	9km NE of Aguanga, CA	...	1537229545350	M 1.1 - 9km NE of Aguanga, CA	0	...	1537230211640	<a href="https...">https...</a>
<b>9331</b>	NaN	...	0.01698	NaN	...	0.66	ml	...	9km NE of Aguanga, CA	...	1537228864470	M 0.7 - 9km NE of Aguanga, CA	0	...	1537305830770	<a href="https...">https...</a>

# Examining the data

- Here, the time column is stored as an integer, which is something we will learn how to fix in Data Wrangling with Pandas:

```
>>> df.dtypes
alert          object
...
mag           float64
magType        object
...
time          int64
title          object
tsunami       int64
...
tz            float64
updated       int64
url           object
dtype: object
```

# Examining the data

- Lastly, we can use the info() method to see how many non-null entries of each column we have and get information on our index.
- Null values are missing values, which, in pandas, will typically be represented as None for objects and NaN (Not a Number) for non-numeric values in a float or integer column:

```
>>> df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 9332 entries, 0 to 9331
Data columns (total 26 columns):
 #   Column    Non-Null Count  Dtype  
--- 
 0   alert      59 non-null    object 
  ...
 8   mag        9331 non-null  float64
 9   magType   9331 non-null  object 
  ...
 18  time       9332 non-null  int64  
 19  title      9332 non-null  object 
 20  tsunami    9332 non-null  int64  
  ...
 23  tz         9331 non-null  float64
 24  updated    9332 non-null  int64  
 25  url        9332 non-null  object 
dtypes: float64(9), int64(4), object(13)
memory usage: 1.9+ MB
```

# Describing and summarizing the data

- The next step is to calculate summary statistics, which will help us get to know our data better.
- Pandas provides several methods for easily doing so; one such method is `describe()`, which also works on Series objects if we are only interested in a particular column.
- Let's get a summary of the numeric columns in our data:  
`>>> df.describe()`

# Describing and summarizing the data

- This gives us the 5-number summary, along with the count, mean, and standard deviation of the numeric columns:

	cdi	dmin	felt	gap	mag	...	sig	time	tsunami	tz	updated
<b>count</b>	329.000000	6139.000000	329.000000	6164.000000	9331.000000	...	9332.000000	9.332000e+03	9332.000000	9331.000000	9.332000e+03
<b>mean</b>	2.754711	0.544925	12.310030	121.506588	1.497345	...	56.899914	1.538284e+12	0.006537	-451.990140	1.538537e+12
<b>std</b>	1.010637	2.214305	48.954944	72.962363	1.203347	...	91.872163	6.080306e+08	0.080589	231.752571	6.564135e+08
<b>min</b>	0.000000	0.000648	0.000000	12.000000	-1.260000	...	0.000000	1.537229e+12	0.000000	-720.000000	1.537230e+12
<b>25%</b>	2.000000	0.020425	1.000000	66.142500	0.720000	...	8.000000	1.537793e+12	0.000000	-540.000000	1.537996e+12
<b>50%</b>	2.700000	0.059050	2.000000	105.000000	1.300000	...	26.000000	1.538245e+12	0.000000	-480.000000	1.538621e+12
<b>75%</b>	3.300000	0.177250	5.000000	159.000000	1.900000	...	56.000000	1.538766e+12	0.000000	-480.000000	1.539110e+12
<b>max</b>	8.400000	53.737000	580.000000	355.910000	7.500000	...	2015.000000	1.539475e+12	1.000000	720.000000	1.539537e+12

# Describing and summarizing the data

- By default, `describe()` won't give us any information about the columns of type object, but we can either provide `include='all'` as an argument or run it separately for the data of type `np.object`:

```
>>> df.describe(include=np.object)
```



# Describing and summarizing the data

- When describing non-numeric data, we still get the count of non-null occurrences (count); however, instead of the other summary statistics, we get the number of unique values (unique), the mode (top), and the number of times the mode was observed (freq):

	alert	code	detail	ids	magType	net	place	sources	status	title	type	types	url
<b>count</b>	59	9332	9332	9332	9331	9332	9332	9332	9332	9332	9332	9332	9332
<b>unique</b>	2	9332	9332	9332	10	14	5433	52	2	7807	5	42	9332
<b>top</b>	green	70628507	https://ear...	,pr201827...	ml	ak	10km NE of Aguanga, CA	,ak,	reviewed	M 0.4 - 10km NE of Aguanga, CA	earthquake	,geoserve, origin, phase- data,	https://ear...
<b>freq</b>	58	1	1	1	6803	3166	306	2981	7797	55	9081	5301	1

# Describing and summarizing the data

- Pandas makes this a cinch as well.
- The following table includes methods that will work for both Series and DataFrame objects:

Method	Description	Data types
<code>count()</code>	The number of non-null observations	Any
<code>nunique()</code>	The number of unique values	Any
<code>sum()</code>	The total of the values	Numerical or Boolean
<code>mean()</code>	The average of the values	Numerical or Boolean
<code>median()</code>	The median of the values	Numerical
<code>min()</code>	The minimum of the values	Numerical
<code>idxmin()</code>	The index where the minimum value occurs	Numerical
<code>max()</code>	The maximum of the values	Numerical
<code>idxmax()</code>	The index where the maximum value occurs	Numerical
<code>abs()</code>	The absolute values of the data	Numerical
<code>std()</code>	The standard deviation	Numerical
<code>var()</code>	The variance	Numerical
<code>cov()</code>	The covariance between two <code>Series</code> , or a covariance matrix for all column combinations in a <code>DataFrame</code>	Numerical
<code>corr()</code>	The correlation between two <code>Series</code> , or a correlation matrix for all column combinations in a <code>DataFrame</code>	Numerical
<code>quantile()</code>	Calculates a specific quantile	Numerical
<code>cumsum()</code>	The cumulative sum	Numerical or Boolean
<code>cummin()</code>	The cumulative minimum	Numerical
<code>cummax()</code>	The cumulative maximum	Numerical

# Describing and summarizing the data

With Series objects, we have some additional methods for describing our data:

- `unique()`: Returns the distinct values of the column.
- `value_counts()`: Returns a frequency table of the number of times each unique value in a given column appears, or, alternatively, the percentage of times each unique value appears when passed `normalize=True`.
- `mode()`: Returns the most common value of the column.

# Describing and summarizing the data

- What is the other unique value, though?

```
>>> df.alert.unique()  
array([nan, 'green', 'red'], dtype=object)
```



# Describing and summarizing the data

- Now that we understand what this field means and the values we have in our data, we expect there to be far more 'green' than 'red'; we can check our intuition with a frequency table by using `value_counts()`.
- Notice that we only get counts for the non-null entries:

```
>>> df.alert.value_counts()
```

```
green    58
```

```
red      1
```

```
Name: alert, dtype: int64
```



# Describing and summarizing the data

- Note that `Index` objects also have several methods that can help us describe and summarize our data:

Method	Description
<code>argmax()</code> / <code>argmin()</code>	Find the location of the maximum/minimum value in the index
<code>equals()</code>	Compare the index to another <code>Index</code> object for equality
<code>isin()</code>	Check if the index values are in a list of values and return an array of Booleans
<code>max()</code> / <code>min()</code>	Find the maximum/minimum value in the index
<code>nunique()</code>	Get the number of unique values in the index
<code>to_series()</code>	Create a <code>Series</code> object from the index
<code>unique()</code>	Find the unique values of the index
<code>value_counts()</code>	Create a frequency table for the unique values in the index

# Grabbing subsets of the data

- For this section, we will work in the 5-subsetting\_data.ipynb notebook.
- Our setup is as follows:

```
>>> import pandas as pd  
>>> df = pd.read_csv('data/earthquakes.csv')
```



# Selecting columns

- Remember that a column is a Series object, so, for example, selecting the mag column in the earthquake data gives us the magnitudes of the earthquakes as a Series object:

```
>>> df.mag  
0      1.35  
1      1.29  
2      3.42  
3      0.44  
4      2.16  
      ...  
9327    0.62  
9328    1.00  
9329    2.40  
9330    1.10  
9331    0.66  
Name: mag, Length: 9332, dtype: float64
```

# Selecting columns

- Pandas provides us with a few ways to select columns.
- An alternative to using attribute notation to select a column is to access it with a dictionary-like notation:

```
>>> df[ 'mag' ]  
0      1.35  
1      1.29  
2      3.42  
3      0.44  
4      2.16  
      ...  
9327    0.62  
9328    1.00  
9329    2.40  
9330    1.10  
9331    0.66  
Name: mag, Length: 9332, dtype: float64
```

# Selecting columns

- Note that we aren't limited to selecting one column at a time.
- By passing a list to the dictionary lookup, we can select many columns, giving us a DataFrame object that is a subset of our original dataframe:

```
>>> df[['mag', 'title']]
```

# Selecting columns

- This gives us the full mag and title columns from the original dataframe:

	mag	title
0	1.35	M 1.4 - 9km NE of Aguanga, CA
1	1.29	M 1.3 - 9km NE of Aguanga, CA
2	3.42	M 3.4 - 8km NE of Aguanga, CA
3	0.44	M 0.4 - 9km NE of Aguanga, CA
4	2.16	M 2.2 - 10km NW of Avenal, CA
...	...	...
9327	0.62	M 0.6 - 9km ENE of Mammoth Lakes, CA
9328	1.00	M 1.0 - 3km W of Julian, CA
9329	2.40	M 2.4 - 35km NNE of Hatillo, Puerto Rico
9330	1.10	M 1.1 - 9km NE of Aguanga, CA
9331	0.66	M 0.7 - 9km NE of Aguanga, CA

# Selecting columns

- String methods are a very powerful way to select columns.
- For example, if we wanted to select all the columns that start with mag, along with the title and time columns, we would do the following:

```
>>> df[  
...     ['title', 'time']  
...     + [col for col in df.columns if col.startswith('mag')]  
... ]
```

# Selecting columns

- We get back a dataframe composed of the four columns that matched our criteria.
- Notice how the columns were returned in the order we requested, which is not the order they originally appeared in.
- This means that if we want to reorder our columns, all we have to do is select them in the order we want them to appear:

	title	time	mag	magType
0	M 1.4 - 9km NE of Aguanga, CA	1539475168010	1.35	ml
1	M 1.3 - 9km NE of Aguanga, CA	1539475129610	1.29	ml
2	M 3.4 - 8km NE of Aguanga, CA	1539475062610	3.42	ml
3	M 0.4 - 9km NE of Aguanga, CA	1539474978070	0.44	ml
4	M 2.2 - 10km NW of Avenal, CA	1539474716050	2.16	md
...	...	...	...	...
9327	M 0.6 - 9km ENE of Mammoth Lakes, CA	1537230228060	0.62	md
9328	M 1.0 - 3km W of Julian, CA	1537230135130	1.00	ml
9329	M 2.4 - 35km NNE of Hatillo, Puerto Rico	1537229908180	2.40	md
9330	M 1.1 - 9km NE of Aguanga, CA	1537229545350	1.10	ml
9331	M 0.7 - 9km NE of Aguanga, CA	1537228864470	0.66	ml

# Selecting columns

- Let's break this example down.
- We used a list comprehension to go through each of the columns in the dataframe and only keep the ones whose names started with mag:

```
>>> [col for col in df.columns if col.startswith('mag')]  
['mag', 'magType']
```

# Selecting columns

- Then, we added this result to the other two columns we wanted to keep (title and time):

```
>>> ['title', 'time'] \
... + [col for col in df.columns if col.startswith('mag')]
['title', 'time', 'mag', 'magType']
```

# Selecting columns

- Finally, we were able to use this list to run the actual column selection on the dataframe, resulting in the dataframe in previous Figure:

```
>>> df[  
...     ['title', 'time']  
...     + [col for col in df.columns if col.startswith('mag')]  
... ]
```

# Slicing

- When we want to extract certain rows (slices) from our dataframe, we use slicing.
- DataFrame slicing works similarly to slicing with other Python objects, such as lists and tuples, with the first index being inclusive and the last index being exclusive:

```
>>> df[100:103]
```

# Slicing

- When specifying a slice of 100:103, we get back rows 100, 101, and 102:

	alert	...	dmin	felt	...	mag	magType	...	place	...	time	title	tsunami	...	updated	url
100	NaN	...	NaN	NaN	...	1.20	ml	...	25km NW of Ester, Alaska	...	1539435449480	M 1.2 - 25km NW of Ester, Alaska	0	...	1539443551010	https...
101	NaN	...	0.01355	NaN	...	0.59	md	...	8km ESE of Mammoth Lakes, CA	...	1539435391320	M 0.6 - 8km ESE of Mammoth Lakes, CA	0	...	1539439802162	https...
102	NaN	...	0.02987	NaN	...	1.33	ml	...	8km ENE of Aguanga, CA	...	1539435293090	M 1.3 - 8km ENE of Aguanga, CA	0	...	1539435940470	https...

# Slicing

- We can combine our row and column selections by using what is known as chaining:

```
>>> df[['title', 'time']][100:103]
```

# Slicing

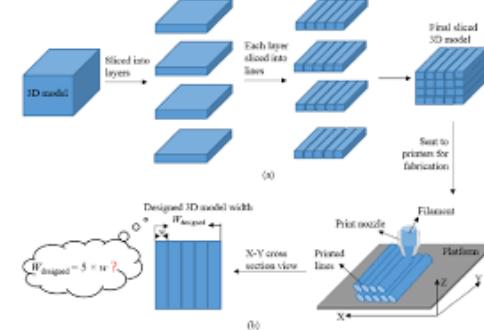
- First, we selected the title and time columns for all the rows, and then we pulled out rows with indices 100, 101, and 102:

		title	time
100	M 1.2 - 25km NW of Ester, Alaska	1539435449480	
101	M 0.6 - 8km ESE of Mammoth Lakes, CA	1539435391320	
102	M 1.3 - 8km ENE of Aguanga, CA	1539435293090	

# Slicing

- In the preceding example, we selected the columns and then sliced the rows, but the order doesn't matter:

```
>>> df[100:103][['title', 'time']].equals(  
...     df[['title', 'time']][100:103]  
... )  
True
```



# Slicing

- Let's trigger this warning to understand it better.
- We will try to update the entries in the title column for a few earthquakes so that they're in lowercase:

```
>>> df[110:113]['title'] = df[110:113]['title'].str.lower()
.../book_env/lib/python3.7/[...]:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead
See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
"""Entry point for launching an IPython kernel.
```

# Slicing

- As indicated by the warning, to be an effective pandas user, it's not enough to know selection and slicing—we must also master indexing.
- Since this is just a warning, our values have been updated, but this may not always be the case:

```
>>> df[110:113]['title']  
110      m 1.1 - 35km s of ester, alaska  
111  m 1.9 - 93km wnw of arctic village, alaska  
112  m 0.9 - 20km wsw of smith valley, nevada  
Name: title, dtype: object
```

# Indexing

- Pandas indexing operations provide us with a one-method way to select both the rows and the columns we want.
- We can use `loc[]` and `iloc[]` to subset our dataframe using label-based or integer-based lookups, respectively.
- A good way to remember the difference is to think of them as location versus integer location.
- For all indexing methods, we provide the row indexer first and then the column indexer, with a comma separating them:  
`df.loc[row_indexer, column_indexer]`

# Indexing

- Note that by using `loc[]`, as indicated in the warning message, we no longer trigger any warnings from pandas for this operation.
- We also changed the end index from 113 to 112 because `loc[]` is inclusive of endpoints:

```
>>> df.loc[110:112, 'title'] = \  
...     df.loc[110:112, 'title'].str.lower()  
>>> df.loc[110:112, 'title']  
110          m 1.1 - 35km s of ester, alaska  
111      m 1.9 - 93km wnw of arctic village, alaska  
112      m 0.9 - 20km wsw of smith valley, nevada  
Name: title, dtype: object
```

# Indexing

- We can select all the rows (columns) if we use : as the row (column) indexer, just like with regular Python slicing.
- Let's grab all the rows of the title column with loc[]:

```
>>> df.loc[:, 'title']
0                  M 1.4 - 9km NE of Aguanga, CA
1                  M 1.3 - 9km NE of Aguanga, CA
2                  M 3.4 - 8km NE of Aguanga, CA
3                  M 0.4 - 9km NE of Aguanga, CA
4                  M 2.2 - 10km NW of Avenal, CA
...
9327      M 0.6 - 9km ENE of Mammoth Lakes, CA
9328          M 1.0 - 3km W of Julian, CA
9329      M 2.4 - 35km NNE of Hatillo, Puerto Rico
9330          M 1.1 - 9km NE of Aguanga, CA
9331          M 0.7 - 9km NE of Aguanga, CA
Name: title, Length: 9332, dtype: object
```

# Indexing

- We can select multiple rows and columns at the same time with loc[]:

```
>>> df.loc[10:15, ['title', 'mag']]
```

# Indexing

- This leaves us with rows 10 through 15 for the title and mag columns only:

		title	mag
10		M 0.5 - 10km NE of Aguanga, CA	0.50
11	M 2.8 - 53km SE of Punta Cana, Dominican Republic		2.77
12		M 0.5 - 9km NE of Aguanga, CA	0.50
13	M 4.5 - 120km SSW of Banda Aceh, Indonesia		4.50
14		M 2.1 - 14km NW of Parkfield, CA	2.13
15	M 2.0 - 156km WNW of Haines Junction, Canada		2.00

# Indexing

- As we have seen, when using loc[], our end index is inclusive.
- This isn't the case with iloc[]:

```
>>> df.iloc[10:15, [19, 8]]
```

# Indexing

- Observe how we had to provide a list of integers to select the same columns; these are the column numbers (starting from 0).
- Using `iloc[]`, we lost the row at index 15; this is because the integer slicing that `iloc[]` employs is exclusive of the end index, as with Python slicing syntax:

		title	mag
10		M 0.5 - 10km NE of Aguanga, CA	0.50
11	M 2.8 - 53km SE of Punta Cana, Dominican Republic		2.77
12		M 0.5 - 9km NE of Aguanga, CA	0.50
13	M 4.5 - 120km SSW of Banda Aceh, Indonesia		4.50
14		M 2.1 - 14km NW of Parkfield, CA	2.13

# Indexing

- We aren't limited to using the slicing syntax for the rows, though; columns work as well:

```
>>> df.iloc[10:15, 6:10]
```



# Indexing

- By using slicing, we can easily grab adjacent rows and columns:

	gap	ids	mag	magType
10	57.0	,ci37389162,	0.50	ml
11	186.0	,pr2018286010,	2.77	md
12	76.0	,ci37389146,	0.50	ml
13	157.0	,us1000hbt1,	4.50	mb
14	71.0	,nc73096921,	2.13	md

# Indexing

- When using `loc[]`, this slicing can be done on the column names as well.
- This gives us many ways to achieve the same result:

```
>>> df.iloc[10:15, 6:10].equals(df.loc[10:14, 'gap':'magType'])  
True
```

# Indexing

- To look up scalar values, we use `at[]` and `iat[]`, which are faster.
- Let's select the magnitude (the `mag` column) of the earthquake that was recorded in the row at index 10:

```
>>> df.at[10, 'mag']  
0.5
```



# Indexing

- The magnitude column has a column index of 8; therefore, we can also look up the magnitude with iat[]:

```
>>> df.iat[10, 8]  
0.5
```



# Filtering

- There are endless possibilities for creating Boolean masks—all we need is some code that returns one Boolean value for each row.
- For example, we can see which entries in the mag column had a magnitude greater than two:

```
>>> df.mag > 2
0      False
1      False
2      True
3     False
...
9328   False
9329   True
9330   False
9331   False
Name: mag, Length: 9332, dtype: bool
```

# Filtering

- While we can run this on the entire dataframe, it wouldn't be too useful with our earthquake data since we have columns of various data types.
- However, we can use this strategy to get the subset of the data where the magnitude of the earthquake was greater than or equal to 7.0:

```
>>> df[df.mag >= 7.0]
```

# Filtering

- Our resulting dataframe has just two rows:

	alert	...	dmin	felt	...	mag	magType	...	place	...	time	title	tsunami	...	updated	url
837	green	...	1.763	3.0	...	7.0	mww	...	117km E of Kimbe, Papua New Guinea	...	1539204500290	M 7.0 - 117km E of Kimbe, Papua New Guinea	1	...	1539378744253	https...
5263	red	...	1.589	18.0	...	7.5	mww	...	78km N of Palu, Indonesia	...	1538128963480	M 7.5 - 78km N of Palu, Indonesia	1	...	1539123134531	https...

# Filtering

- We got back a lot of columns we didn't need, though.
- We could have chained a column selection to the end of the last code snippet; however, `loc[]` can handle Boolean masks as well:

```
>>> df.loc[  
...     df.mag >= 7.0,  
...     ['alert', 'mag', 'magType', 'title', 'tsunami', 'type']  
... ]
```

# Filtering

- The following dataframe has been filtered so that it only contains relevant columns:

	alert	mag	magType		title	tsunami	type
837	green	7.0	mww	M 7.0 - 117km E of Kimbe, Papua New Guinea		1	earthquake
5263	red	7.5	mww	M 7.5 - 78km N of Palu, Indonesia		1	earthquake

# Filtering

- We aren't limited to just one criterion, either.
- Let's grab the earthquakes with a red alert and a tsunami.
- To combine masks, we need to surround each of our conditions with parentheses and use the bitwise AND operator (`&`) to require both to be true:

```
>>> df.loc[  
...     (df.tsunami == 1) & (df.alert == 'red'),  
...     ['alert', 'mag', 'magType', 'title', 'tsunami', 'type']  
... ]
```

# Filtering

- There was only a single earthquake in the data that met our criteria:

alert	mag	magType		title	tsunami	type
5263	red	7.5	mww	M 7.5 - 78km N of Palu, Indonesia	1	earthquake

# Filtering

- If, instead, we want at least one of our conditions to be true, we can use the bitwise OR operator (|):

```
>>> df.loc[  
...     (df.tsunami == 1) | (df.alert == 'red'),  
...     ['alert', 'mag', 'magType', 'title', 'tsunami', 'type']  
... ]
```

# Filtering

- Notice that this filter is much less restrictive since, while both conditions can be true, we only require that one of them is:

	alert	mag	magType		title	tsunami	type
36	NaN	5.0	mww	M 5.0 - 165km NNW of Flying Fish Cove, Christm...	1	earthquake	
118	green	6.7	mww	M 6.7 - 262km NW of Ozernovskiy, Russia	1	earthquake	
501	green	5.6	mww	M 5.6 - 128km SE of Kimbe, Papua New Guinea	1	earthquake	
799	green	6.5	mww	M 6.5 - 148km S of Severo-Kuril'sk, Russia	1	earthquake	
816	green	6.2	mww	M 6.2 - 94km SW of Kokopo, Papua New Guinea	1	earthquake	
...	...	...	...	...	...	...	...
8561	NaN	5.4	mb	M 5.4 - 228km S of Taron, Papua New Guinea	1	earthquake	
8624	NaN	5.1	mb	M 5.1 - 278km SE of Pondaguitan, Philippines	1	earthquake	
9133	green	5.1	ml	M 5.1 - 64km SSW of Kaktovik, Alaska	1	earthquake	
9175	NaN	5.2	mb	M 5.2 - 126km N of Dili, East Timor	1	earthquake	
9304	NaN	5.1	mb	M 5.1 - 34km NW of Finschhafen, Papua New Guinea	1	earthquake	

# Filtering

- In the previous two examples, our conditions involved equality; however, we are by no means limited to this.
- Let's select all the earthquakes in Alaska where we have a non-null value for the alert column:

```
>>> df.loc[  
...     (df.place.str.contains('Alaska'))  
...     & (df.alert.notnull()),  
...     ['alert', 'mag', 'magType', 'title', 'tsunami', 'type']  
... ]
```

# Filtering

- All the earthquakes in Alaska that have a value for alert are green, and some were accompanied by tsunamis, with the highest magnitude being 5.1:

	alert	mag	magType		title	tsunami	type
1015	green	5.0	ml	M 5.0 - 61km SSW of Chignik Lake, Alaska		1	earthquake
1273	green	4.0	ml	M 4.0 - 71km SW of Kaktovik, Alaska		1	earthquake
1795	green	4.0	ml	M 4.0 - 60km WNW of Valdez, Alaska		1	earthquake
2752	green	4.0	ml	M 4.0 - 67km SSW of Kaktovik, Alaska		1	earthquake
3260	green	3.9	ml	M 3.9 - 44km N of North Nenana, Alaska		0	earthquake
4101	green	4.2	ml	M 4.2 - 131km NNW of Arctic Village, Alaska		0	earthquake
6897	green	3.8	ml	M 3.8 - 80km SSW of Kaktovik, Alaska		0	earthquake
8524	green	3.8	ml	M 3.8 - 69km SSW of Kaktovik, Alaska		0	earthquake
9133	green	5.1	ml	M 5.1 - 64km SSW of Kaktovik, Alaska		1	earthquake

# Filtering

- Let's break down how we got this.
- Series objects have some string methods that can be accessed via the str attribute.
- Using this, we can create a Boolean mask of all the rows where the place column contained the word Alaska:

```
df.place.str.contains('Alaska')
```

# Filtering

- To get all the rows where the alert column was not null, we used the Series object's `notnull()` method (this works for DataFrame objects as well) to create a Boolean mask of all the rows where the alert column was not null:

```
df.alert.notnull()
```



# Filtering

- Then, like we did previously, we combine the two conditions with the & operator to complete our mask:

`(df.place.str.contains('Alaska')) & (df.alert.notnull())`



# Filtering

- The \$ character means end and 'CA\$' gives us entries that end in CA, so we can use 'CA|California\$' to get entries that end in either:

```
>>> df.loc[  
...     (df.place.str.contains(r'CA|California$'))  
...     & (df.mag > 3.8),  
...     ['alert', 'mag', 'magType', 'title', 'tsunami', 'type']  
... ]
```

# Filtering

- There were only two earthquakes in California with magnitudes greater than 3.8 during the time period we are studying:

	<b>alert</b>	<b>mag</b>	<b>magType</b>		<b>title</b>	<b>tsunami</b>	<b>type</b>
<b>1465</b>	green	3.83	mw	M 3.8 - 109km WNW of Trinidad, CA		0	earthquake
<b>2414</b>	green	3.83	mw	M 3.8 - 5km SW of Tres Pinos, CA		1	earthquake

# Filtering

- Thankfully, pandas makes this type of mask much easier to create by providing us with the `between()` method:

```
>>> df.loc[  
...     df.mag.between(6.5, 7.5),  
...     ['alert', 'mag', 'magType', 'title', 'tsunami', 'type']  
... ]
```

# Filtering

- The result contains all the earthquakes with magnitudes in the range [6.5, 7.5]—it's inclusive of both ends by default, but we can pass in `inclusive=False` to change this:

	alert	mag	magType			title	tsunami	type
118	green	6.7	mww	M 6.7 - 262km NW of Ozernovskiy, Russia			1	earthquake
799	green	6.5	mww	M 6.5 - 148km S of Severo-Kuril'sk, Russia			1	earthquake
837	green	7.0	mww	M 7.0 - 117km E of Kimbe, Papua New Guinea			1	earthquake
4363	green	6.7	mww	M 6.7 - 263km NNE of Ndoi Island, Fiji			1	earthquake
5263	red	7.5	mww	M 7.5 - 78km N of Palu, Indonesia			1	earthquake

# Filtering

- We will take a look at earthquakes measured with either the mw or mwb magnitude type:

```
>>> df.loc[  
...     df.magType.isin(['mw', 'mwb']),  
...     ['alert', 'mag', 'magType', 'title', 'tsunami', 'type']  
... ]
```

# Filtering

- We have two earthquakes that were measured with the mwb magnitude type and four that were measured with the mw magnitude type:

	alert	mag	magType			title	tsunami	type
995	NaN	3.35	mw	M 3.4 - 9km WNW of Cobb, CA		0	0	earthquake
1465	green	3.83	mw	M 3.8 - 109km WNW of Trinidad, CA		0	0	earthquake
2414	green	3.83	mw	M 3.8 - 5km SW of Tres Pinos, CA		1	1	earthquake
4988	green	4.41	mw	M 4.4 - 1km SE of Delta, B.C., MX		1	1	earthquake
6307	green	5.80	mwb	M 5.8 - 297km NNE of Ndoi Island, Fiji		0	0	earthquake
8257	green	5.70	mwb	M 5.7 - 175km SSE of Lambasa, Fiji		0	0	earthquake

# Filtering

- We can use `idxmin()` and `idxmax()` for the indices of the minimum and maximum, respectively.
- Let's grab the row numbers for the lowest-magnitude and highest-magnitude earthquakes:

```
>>> [df.mag.idxmin(), df.mag.idxmax()]  
[2409, 5263]
```



# Filtering

- The minimum magnitude earthquake occurred in Alaska and the highest magnitude earthquake occurred in Indonesia, accompanied by a tsunami.
- We will discuss the earthquake in Indonesia in Visualizing Data with Pandas and Matplotlib, and Plotting with Seaborn and Customization Techniques:

	alert	mag	magType		title	tsunami	type
2409	NaN	-1.26	ml	M -1.3 - 41km ENE of Adak, Alaska		0	earthquake
5263	red	7.50	mww	M 7.5 - 78km N of Palu, Indonesia		1	earthquake

# Adding and removing data

- Should we find ourselves in a situation where we don't want to change the original data, but rather want to return a new copy of the data that has been modified, we must be sure to copy our dataframe before making any changes:

```
df_to_modify = df.copy()
```

# Adding and removing data

- We will once again be working with the earthquake data, but this time, we will only read in a subset of the columns:

```
>>> import pandas as pd  
>>> df = pd.read_csv(  
...     'data/earthquakes.csv',  
...     usecols=[  
...         'time', 'title', 'place', 'magType',  
...         'mag', 'alert', 'tsunami'  
...     ]  
... )
```

# Creating new data

- Creating new columns can be achieved in the same fashion as variable assignment.
- For example, we can create a column to indicate the source of our data; since all our data came from the same source, we can take advantage of broadcasting to set every row of this column to the same value:

```
>>> df['source'] = 'USGS API'  
>>> df.head()
```

# Creating new data

- The new column is created to the right of the original columns, with a value of USGS API for every row:

	alert	mag	magType	place	time	title	tsunami	source
0	NaN	1.35	ml	9km NE of Aguanga, CA	1539475168010	M 1.4 - 9km NE of Aguanga, CA	0	USGS API
1	NaN	1.29	ml	9km NE of Aguanga, CA	1539475129610	M 1.3 - 9km NE of Aguanga, CA	0	USGS API
2	NaN	3.42	ml	8km NE of Aguanga, CA	1539475062610	M 3.4 - 8km NE of Aguanga, CA	0	USGS API
3	NaN	0.44	ml	9km NE of Aguanga, CA	1539474978070	M 0.4 - 9km NE of Aguanga, CA	0	USGS API
4	NaN	2.16	md	10km NW of Avenal, CA	1539474716050	M 2.2 - 10km NW of Avenal, CA	0	USGS API

# Creating new data

- With our earthquake data, let's create a column that tells us whether the earthquake's magnitude was negative:

```
>>> df['mag_negative'] = df.mag < 0  
>>> df.head()
```

# Creating new data

- Note that the new column has been added to the right:

	alert	mag	magType	place	time	title	tsunami	source	mag_negative
0	NaN	1.35	ml	9km NE of Aguanga, CA	1539475168010	M 1.4 - 9km NE of Aguanga, CA	0	USGS API	False
1	NaN	1.29	ml	9km NE of Aguanga, CA	1539475129610	M 1.3 - 9km NE of Aguanga, CA	0	USGS API	False
2	NaN	3.42	ml	8km NE of Aguanga, CA	1539475062610	M 3.4 - 8km NE of Aguanga, CA	0	USGS API	False
3	NaN	0.44	ml	9km NE of Aguanga, CA	1539474978070	M 0.4 - 9km NE of Aguanga, CA	0	USGS API	False
4	NaN	2.16	md	10km NW of Avenal, CA	1539474716050	M 2.2 - 10km NW of Avenal, CA	0	USGS API	False

# Creating new data

- By using a regular expression to extract everything in the place column after the comma, we can see some of the issues firsthand:

```
>>> df.place.str.extract(r', (.*)')[0].sort_values().unique()
array(['Afghanistan', 'Alaska', 'Argentina', 'Arizona',
       'Arkansas', 'Australia', 'Azerbaijan', 'B.C., MX',
       'Barbuda', 'Bolivia', ..., 'CA', 'California', 'Canada',
       'Chile', ..., 'East Timor', 'Ecuador', 'Ecuador region',
       ..., 'Mexico', 'Missouri', 'Montana', 'NV', 'Nevada',
       ..., 'Yemen', nan], dtype=object)
```

# Creating new data

- Then, we can use the replace() method to replace patterns in the place column as we see fit:

```
>>> df['parsed_place'] = df.place.str.replace(  
...      r'.* of ', '', regex=True # remove <x> of <x>  
... ).str.replace(  
...     'the ', '' # remove "the "  
... ).str.replace(  
...     r'CA$', 'California', regex=True # fix California  
... ).str.replace(  
...     r'NV$', 'Nevada', regex=True # fix Nevada  
... ).str.replace(  
...     r'MX$', 'Mexico', regex=True # fix Mexico  
... ).str.replace(  
...     r' region$', '', regex=True # fix " region" endings  
... ).str.replace(  
...     'northern ', '' # remove "northern "  
... ).str.replace(  
...     'Fiji Islands', 'Fiji' # line up the Fiji places  
... ).str.replace( # remove anything else extraneous from start  
...     r'^.*', '', regex=True  
... ).str.strip() # remove any extra spaces
```

# Creating new data

- We could address this with another call to replace(); however, this goes to show that entity recognition can be quite challenging:

```
>>> df.parsed_place.sort_values().unique()
array([... , 'California', 'Canada', 'Carlsberg Ridge', ... ,
      'Dominican Republic', 'East Timor', 'Ecuador',
      'El Salvador', 'Fiji', 'Greece', ... ,
      'Mexico', 'Mid-Indian Ridge', 'Missouri', 'Montana',
      'Nevada', 'New Caledonia', ... ,
      'South Georgia and South Sandwich Islands',
      'South Sandwich Islands', ... , 'Yemen'], dtype=object)
```

# Creating new data

- Rather than just show the first five entries (which are all in California), we will use sample() to randomly select five rows:

```
>>> df.assign(  
...     in_ca=df.parsed_place.str.endswith('California'),  
...     in_alaska=df.parsed_place.str.endswith('Alaska'))  
... ).sample(5, random_state=0)
```

# Creating new data

- If we want to replace our original dataframe with this, we just use variable assignment to store the result of assign() in df (for example, df = df.assign(...)):

	alert	mag	magType	place	time	title	tsunami	source	mag_negative	parsed_place	in_ca	in_alaska
7207	NaN	4.80	mwr	73km SSW of Masachapa, Nicaragua	1537749595210	M 4.8 - 73km SSW of Masachapa, Nicaragua	0	USGS API	False	Nicaragua	False	False
4755	NaN	1.09	ml	28km NNW of Packwood, Washington	1538227540460	M 1.1 - 28km NNW of Packwood, Washington	0	USGS API	False	Washington	False	False
4595	NaN	1.80	ml	77km SSW of Kaktovik, Alaska	1538259609862	M 1.8 - 77km SSW of Kaktovik, Alaska	0	USGS API	False	Alaska	False	True
3566	NaN	1.50	ml	102km NW of Arctic Village, Alaska	1538464751822	M 1.5 - 102km NW of Arctic Village, Alaska	0	USGS API	False	Alaska	False	True
2182	NaN	0.90	ml	26km ENE of Pine Valley, CA	1538801713880	M 0.9 - 26km ENE of Pine Valley, CA	0	USGS API	False	California	True	False

# Creating new data

- For example, let's once again create the in\_ca and in\_alaska columns, but this time also create a new column, neither, which is True if both in\_ca and in\_alaska are False:

```
>>> df.assign(  
...     in_ca=df.parsed_place == 'California',  
...     in_alaska=df.parsed_place == 'Alaska',  
...     neither=lambda x: ~x.in_ca & ~x.in_alaska  
... ).sample(5, random_state=0)
```

# Creating new data

- Remember that `~` is the bitwise negation operator, so this allows us to create a column with the result of NOT `in_ca` AND NOT `in_alaska` per row:

	alert	mag	magType	place	time	title	tsunami	source	mag_negative	parsed_place	in_ca	in_alaska	neither
7207	NaN	4.80	mwr	73km SSW of Masachapa, Nicaragua	1537749595210	M 4.8 - 73km SSW of Masachapa, Nicaragua	0	USGS API	False	Nicaragua	False	False	True
4755	NaN	1.09	ml	28km NNW of Packwood, Washington	1538227540460	M 1.1 - 28km NNW of Packwood, Washington	0	USGS API	False	Washington	False	False	True
4595	NaN	1.80	ml	77km SSW of Kaktovik, Alaska	1538259609862	M 1.8 - 77km SSW of Kaktovik, Alaska	0	USGS API	False	Alaska	False	True	False
3566	NaN	1.50	ml	102km NW of Arctic Village, Alaska	1538464751822	M 1.5 - 102km NW of Arctic Village, Alaska	0	USGS API	False	Alaska	False	True	False
2182	NaN	0.90	ml	26km ENE of Pine Valley, CA	1538801713880	M 0.9 - 26km ENE of Pine Valley, CA	0	USGS API	False	California	True	False	False

# Creating new data

- Say we were working with two separate dataframes; one with earthquakes accompanied by tsunamis and the other with earthquakes without tsunamis:

```
>>> tsunami = df[df.tsunami == 1]
>>> no_tsunami = df[df.tsunami == 0]
>>> tsunami.shape, no_tsunami.shape
((61, 10), (9271, 10))
```

# Creating new data

- Let's use pd.concat() with the default axis of 0 for rows:

```
>>> pd.concat([tsunami, no_tsunami]).shape  
(9332, 10) # 61 rows + 9271 rows
```

# Creating new data

- Note that the previous result is equivalent to running the `append()` method on the `DataFrame`.
- This still returns a new `DataFrame` object, but it saves us from having to remember which axis is which, since `append()` is actually a wrapper around the `concat()` function:

```
>>> tsunami.append(no_tsunami).shape  
(9332, 10) # 61 rows + 9271 rows
```

# Creating new data

- Instead, we will concatenate along the columns (axis=1) to add back what we are missing:

```
>>> additional_columns = pd.read_csv(  
...     'data/earthquakes.csv', usecols=['tz', 'felt', 'ids']  
... )  
>>> pd.concat([df.head(2), additional_columns.head(2)], axis=1)
```

# Creating new data

- Since the indices of the dataframes align, the additional columns are placed to the right of our original columns:

	alert	mag	magType	place	time	title	tsunami	source	mag_negative	parsed_place	felt	ids	tz
0	NaN	1.35	ml	9km NE of Aguanga, CA	1539475168010	M 1.4 - 9km NE of Aguanga, CA	0	USGS API	False	California	NaN ,ci37389218,	-480.0	
1	NaN	1.29	ml	9km NE of Aguanga, CA	1539475129610	M 1.3 - 9km NE of Aguanga, CA	0	USGS API	False	California	NaN ,ci37389202,	-480.0	

# Creating new data

- Say we forgot that our original dataframe had the row numbers as the index, and we read in the additional columns by setting the time column as the index:

```
>>> additional_columns = pd.read_csv(  
...     'data/earthquakes.csv',  
...     usecols=['tz', 'felt', 'ids', 'time'],  
...     index_col='time'  
... )  
>>> pd.concat([df.head(2), additional_columns.head(2)], axis=1)
```

# Creating new data

- In Data Wrangling with Pandas, we will see how to reset the index and set the index, both of which could resolve this issue:

	alert	mag	magType	place	time	title	tsunami	source	mag_negative	parsed_place	felt	ids	tz
0	NaN	1.35	ml	9km NE of Aguanga, CA	1.539475e+12	M 1.4 - 9km NE of Aguanga, CA	0.0	USGS API	False	California	NaN	NaN	NaN
1	NaN	1.29	ml	9km NE of Aguanga, CA	1.539475e+12	M 1.3 - 9km NE of Aguanga, CA	0.0	USGS API	False	California	NaN	NaN	NaN
1539475129610	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	,ci37389202,	-480.0
1539475168010	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	,ci37389218,	-480.0

# Creating new data

- By default, this is outer, so we keep everything; however, if we use inner, we will only keep what they have in common:

```
>>> pd.concat(  
...     [  
...         tsunami.head(2),  
...         no_tsunami.head(2).assign(type='earthquake')  
...     ],  
...     join='inner'  
... )
```

# Creating new data

- Take a look at the index, though; these were the row numbers from the original dataframe before we divided it into tsunami and no\_tsunami:

	alert	mag	magType	place	time	title	tsunami	source	mag_negative	parsed_place
36	NaN	5.00	mww	165km NNW of Flying Fish Cove, Christmas Island	1539459504090	M 5.0 - 165km NNW of Flying Fish Cove, Christm...	1	USGS API	False	Christmas Island
118	green	6.70	mww	262km NW of Ozernovskiy, Russia	1539429023560	M 6.7 - 262km NW of Ozernovskiy, Russia	1	USGS API	False	Russia
0	NaN	1.35	ml	9km NE of Aguanga, CA	1539475168010	M 1.4 - 9km NE of Aguanga, CA	0	USGS API	False	California
1	NaN	1.29	ml	9km NE of Aguanga, CA	1539475129610	M 1.3 - 9km NE of Aguanga, CA	0	USGS API	False	California

# Creating new data

- If the index is not meaningful, we can also pass in ignore\_index to get sequential values in the index:

```
>>> pd.concat(  
...     [  
...         tsunami.head(2),  
...         no_tsunami.head(2).assign(type='earthquake')  
...     ],  
...     join='inner', ignore_index=True  
... )
```

# Creating new data

- The index is now sequential, and the row numbers no longer match the original dataframe:

	alert	mag	magType	place	time	title	tsunami	source	mag_negative	parsed_place
0	NaN	5.00	mww	165km NNW of Flying Fish Cove, Christmas Island	1539459504090	M 5.0 - 165km NNW of Flying Fish Cove, Christm...	1	USGS API	False	Christmas Island
1	green	6.70	mww	262km NW of Ozernovskiy, Russia	1539429023560	M 6.7 - 262km NW of Ozernovskiy, Russia	1	USGS API	False	Russia
2	NaN	1.35	ml	9km NE of Aguanga, CA	1539475168010	M 1.4 - 9km NE of Aguanga, CA	0	USGS API	False	California
3	NaN	1.29	ml	9km NE of Aguanga, CA	1539475129610	M 1.3 - 9km NE of Aguanga, CA	0	USGS API	False	California

# Deleting unwanted data

- Let's use dictionary notation to delete the source column.
- Notice that it no longer appears in the result of df.columns:

```
>>> del df['source']
>>> df.columns
Index(['alert', 'mag', 'magType', 'place', 'time', 'title',
       'tsunami', 'mag_negative', 'parsed_place'],
      dtype='object')
```

# Deleting unwanted data

- Note that if we aren't sure whether the column exists, we should put our column deletion code in a try...except block:

```
try:  
    del df['source']  
except KeyError:  
    pass # handle the error here
```

# Deleting unwanted data

- We can use pop() to grab the series for the mag\_negative column, which we can use as a Boolean mask later without having it in our dataframe:

```
>>> mag_negative = df.pop('mag_negative')
>>> df.columns
Index(['alert', 'mag', 'magType', 'place', 'time', 'title',
       'tsunami', 'parsed_place'],
      dtype='object')
```

# Deleting unwanted data

- We now have a Boolean mask in the mag\_negative variable that used to be a column in df:

```
>>> mag_negative.value_counts()  
False    8841  
True     491  
Name: mag_negative, dtype: int64
```

# Deleting unwanted data

- Since we used `pop()` to remove the `mag_negative` series rather than deleting it, we can still use it to filter our dataframe:

```
>>> df[mag_negative].head()
```

# Deleting unwanted data

- This leaves us with the earthquakes that had negative magnitudes.
- Since we also called head(), we get back the first five such earthquakes:

	alert	mag	magType	place	time	title	tsunami	parsed_place
39	NaN	-0.10	ml	6km NW of Lemmon Valley, Nevada	1539458844506	M -0.1 - 6km NW of Lemmon Valley, Nevada	0	Nevada
49	NaN	-0.10	ml	6km NW of Lemmon Valley, Nevada	1539455017464	M -0.1 - 6km NW of Lemmon Valley, Nevada	0	Nevada
135	NaN	-0.40	ml	10km SSE of Beatty, Nevada	1539422175717	M -0.4 - 10km SSE of Beatty, Nevada	0	Nevada
161	NaN	-0.02	md	20km SSE of Ronan, Montana	1539412475360	M -0.0 - 20km SSE of Ronan, Montana	0	Montana
198	NaN	-0.20	ml	60km N of Pahrump, Nevada	1539398340822	M -0.2 - 60km N of Pahrump, Nevada	0	Nevada

# Deleting unwanted data

- Let's remove the first two rows:

```
>>> df.drop([0, 1]).head(2)
```

- Notice that the index starts at 2 because we dropped 0 and 1:

	alert	mag	magType	place	time	title	tsunami	parsed_place
2	NaN	3.42	ml	8km NE of Aguanga, CA	1539475062610	M 3.4 - 8km NE of Aguanga, CA	0	California
3	NaN	0.44	ml	9km NE of Aguanga, CA	1539474978070	M 0.4 - 9km NE of Aguanga, CA	0	California

# Deleting unwanted data

- By default, drop() assumes that we want to delete rows (axis=0).
- If we want to drop columns, we can either pass axis=1 or specify our list of column names using the columns argument.
- Let's delete some more columns:

```
>>> cols_to_drop = [  
...     col for col in df.columns  
...     if col not in [  
...         'alert', 'mag', 'title', 'time', 'tsunami'  
...     ]  
... ]  
>>> df.drop(columns=cols_to_drop).head()
```

# Deleting unwanted data

- This drops all the columns that aren't in the list we wanted to keep:

	alert	mag	time		title	tsunami
0	NaN	1.35	1539475168010	M 1.4 - 9km NE of Aguanga, CA		0
1	NaN	1.29	1539475129610	M 1.3 - 9km NE of Aguanga, CA		0
2	NaN	3.42	1539475062610	M 3.4 - 8km NE of Aguanga, CA		0
3	NaN	0.44	1539474978070	M 0.4 - 9km NE of Aguanga, CA		0
4	NaN	2.16	1539474716050	M 2.2 - 10km NW of Avenal, CA		0

# Deleting unwanted data

- Whether we decide to pass `axis=1` to `drop()` or use the `columns` argument, our result will be equivalent:

```
>>> df.drop(columns=cols_to_drop).equals(  
...     df.drop(cols_to_drop, axis=1)  
... )  
True
```

# Deleting unwanted data

- By default, drop() will return a new DataFrame object; however, if we really want to remove the data from our original dataframe, we can pass in inplace=True, which will save us from having to reassign the result back to our dataframe.
- The result is the same as in Figure:

```
>>> df.drop(columns=cols_to_drop, inplace=True)  
>>> df.head()
```



# Summary

- In this lesson, we learned how to use pandas for the data collection portion of data analysis and to describe our data with statistics, which will be helpful when we get to the drawing conclusions phase.
- We learned the main data structures of the pandas library, along with some of the operations we can perform on them.

# "Complete Exercises"

# "Complete Lab 7"

# 8: Data Wrangling with Pandas



# Data Wrangling with Pandas

In this lesson, we will cover the following topics:

- Understanding data wrangling
- Exploring an API to find and collect temperature data
- Cleaning data
- Reshaping data
- Handling duplicate, missing, or invalid data

# lesson materials

- The following files are in the data/ directory:

File	Description	Source
<code>bitcoin.csv</code>	Daily opening, high, low, and closing price of bitcoin, along with volume traded and market capitalization for 2017 through 2018.	CoinMarketCap
<code>dirty_data.csv</code>	2018 weather data for New York City, manipulated to introduce data issues.	Modified version of the data from the NCEI API's GHCND dataset.
<code>long_data.csv</code>	Long format temperature data for New York City in October 2018 from the Boonton 1 station, containing daily temperature at time of observation, minimum temperature, and maximum temperature.	The NCEI API's GHCND dataset
<code>nyc_temperatures.csv</code>	Temperature data for New York City in October 2018 measured from LaGuardia airport, containing daily minimum, maximum, and average temperature.	The NCEI API's GHCND dataset
<code>sp500.csv</code>	Daily opening, high, low, and closing price of the S&P 500 stock index, along with volume traded and adjusted close for 2017 through 2018.	The <code>stock_analysis</code> package (see <i>Chapter 7, Financial Analysis – Bitcoin and the Stock Market</i> ).
<code>wide_data.csv</code>	Wide format temperature data for New York City in October 2018 from the Boonton 1 station, containing daily temperature at time of observation, minimum temperature, and maximum temperature.	The NCEI API's GHCND dataset

# Understanding data wrangling

- When we perform data wrangling, we are taking our input data from its original state and putting it in a format where we can perform meaningful analysis on it.
- Data manipulation is another way to refer to this process.
- There is no set list of operations; the only goal is that the data post-wrangling is more useful to us than when we started.

# Data cleaning

Some essential data cleaning tasks to master include the following:

- Renaming
- Sorting and reordering
- Data type conversions
- Handling duplicate data
- Addressing missing or invalid data
- Filtering to the desired subset of data

# Data transformation

- Often, people will record and present data in wide format, but there are certain visualizations that require the data to be in long format:

observations		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	

repeated values for **date** column

	date	variable names	variable values
		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

# Data transformation

- read in the CSV files containing wide and long format data:

```
>>> import matplotlib.pyplot as plt
>>> import pandas as pd
>>> wide_df = \
...     pd.read_csv('data/wide_data.csv', parse_dates=['date'])
>>> long_df = pd.read_csv(
...     'data/long_data.csv',
...     usecols=['date', 'datatype', 'value'],
...     parse_dates=['date']
... )[['date', 'datatype', 'value']] # sort columns
```

# Data transformation

- Let's look at the top six observations from the wide format data in `wide_df`:

```
>>> wide_df.head(6)
```

# Data transformation

- Each column contains the top six observations of a specific class of temperature data in degrees Celsius—maximum temperature (TMAX), minimum temperature (TMIN), and temperature at the time of observation (TOBS)—at a daily frequency:

	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

# Data transformation

- When working with wide format data, we can easily grab summary statistics on this data by using the describe() method.
- Note that while older versions of pandas treated datetimes as categorical, pandas is moving toward treating them as numeric, so we pass datetime\_is\_numeric=True to suppress the warning:

```
>>> wide_df.describe(include='all', datetime_is_numeric=True)
```

# Data transformation

- With hardly any effort on our part, we get summary statistics for the dates, maximum temperature, minimum temperature, and temperature at the time of observation:

	date	TMAX	TMIN	TOBS
<b>count</b>	31	31.000000	31.000000	31.000000
<b>mean</b>	2018-10-16 00:00:00	16.829032	7.561290	10.022581
<b>min</b>	2018-10-01 00:00:00	7.800000	-1.100000	-1.100000
<b>25%</b>	2018-10-08 12:00:00	12.750000	2.500000	5.550000
<b>50%</b>	2018-10-16 00:00:00	16.100000	6.700000	8.300000
<b>75%</b>	2018-10-23 12:00:00	21.950000	13.600000	16.100000
<b>max</b>	2018-10-31 00:00:00	26.700000	17.800000	21.700000
<b>std</b>	NaN	5.714962	6.513252	6.596550

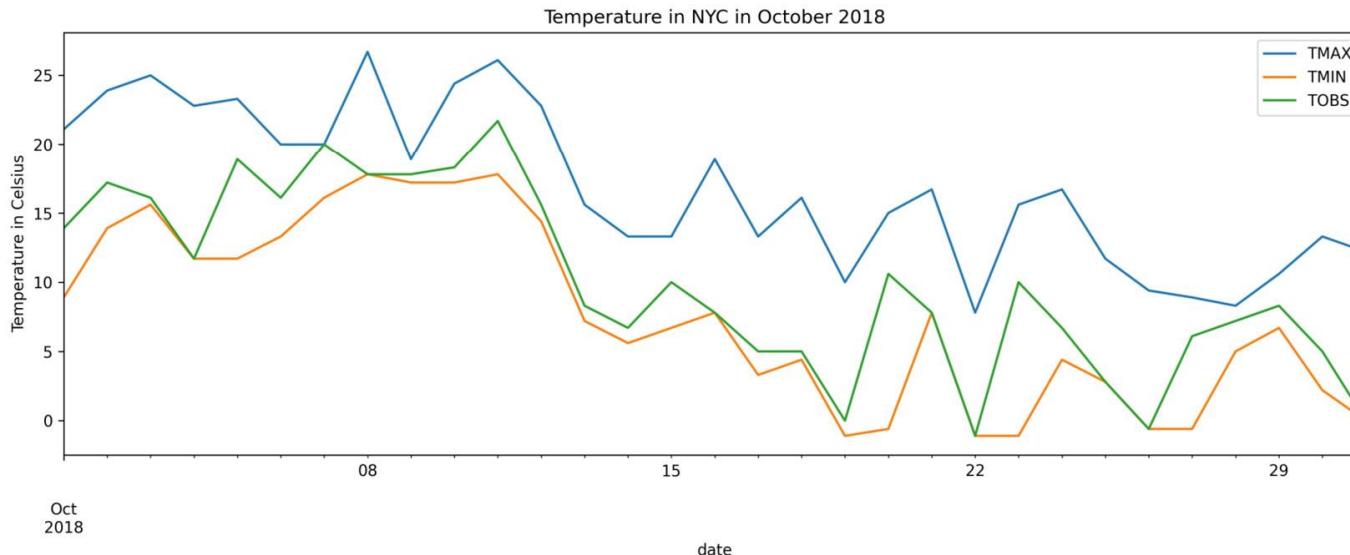
# Data transformation

- As we discussed previously, the summary data in the preceding table is easy to obtain and is informative.
- This format can easily be plotted with pandas as well, provided we tell it exactly what we want to plot:

```
>>> wide_df.plot(  
...     x='date', y=['TMAX', 'TMIN', 'TOBS'], figsize=(15, 5),  
...     title='Temperature in NYC in October 2018'  
... ).set_ylabel('Temperature in Celsius')  
>>> plt.show()
```

# Data transformation

- Pandas plots the daily maximum temperature, minimum temperature, and temperature at the time of observation as their own lines on a single line plot:



# Data transformation

- We can look at the top six rows of the long format data in `long_df` to see the difference between wide format and long format data:

```
>>> long_df.head(6)
```

# Data transformation

- Notice that we now have three entries for each date, and the datatype column tells us what the data in the value column is for that row:

	<b>date</b>	<b>datatype</b>	<b>value</b>
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

# Data transformation

- If we try to get summary statistics, like we did with the wide format data, the result isn't as helpful:

```
>>> long_df.describe(include='all', datetime_is_numeric=True)
```

# Data transformation

- This means that this summary data is not very helpful:

	date	datatype	value
<b>count</b>	93	93	93.000000
<b>unique</b>	NaN	3	NaN
<b>top</b>	NaN	TOBS	NaN
<b>freq</b>	NaN	31	NaN
<b>mean</b>	2018-10-16 00:00:00	NaN	11.470968
<b>min</b>	2018-10-01 00:00:00	NaN	-1.100000
<b>25%</b>	2018-10-08 00:00:00	NaN	6.700000
<b>50%</b>	2018-10-16 00:00:00	NaN	11.700000
<b>75%</b>	2018-10-24 00:00:00	NaN	17.200000
<b>max</b>	2018-10-31 00:00:00	NaN	26.700000
<b>std</b>	NaN	NaN	7.362354

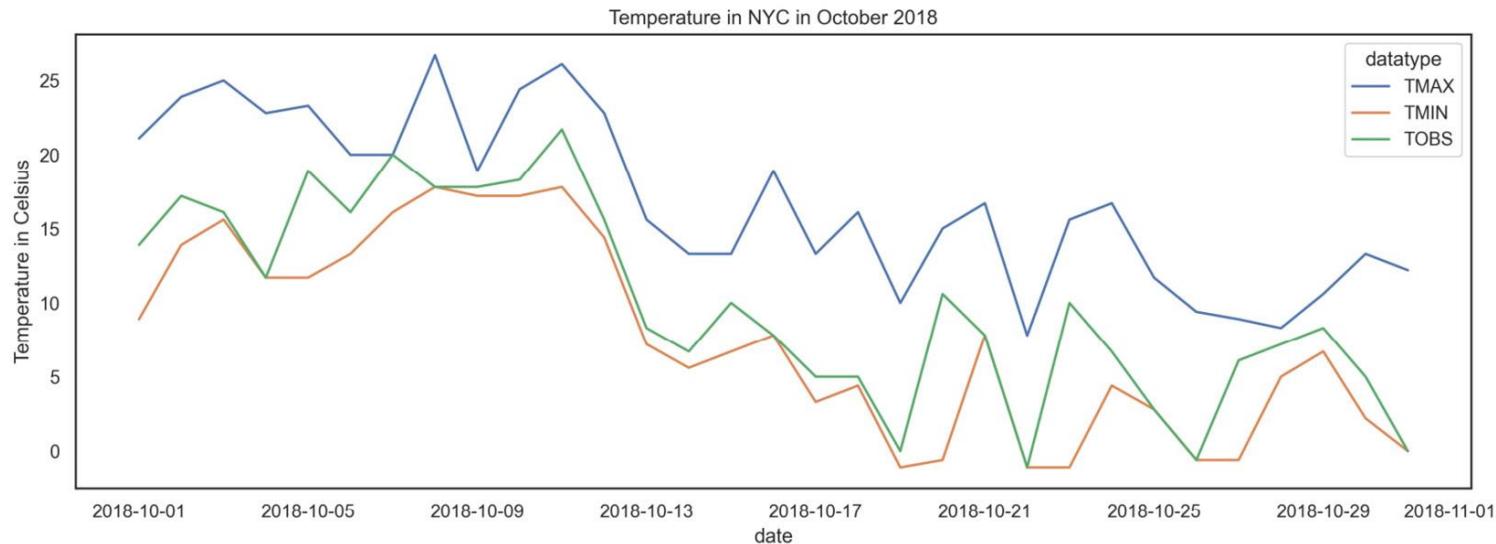
# Data transformation

- Pandas expects its data for plotting to be in wide format, so, to easily make the same plot that we did with the wide format data, we must use another plotting library, called seaborn, which we will cover in Plotting with Seaborn and Customization Techniques:

```
>>> import seaborn as sns
>>> sns.set(rc={'figure.figsize': (15, 5)}, style='white')
>>> ax = sns.lineplot(
...     data=long_df, x='date', y='value', hue='datatype'
... )
>>> ax.set_ylabel('Temperature in Celsius')
>>> ax.set_title('Temperature in NYC in October 2018')
>>> plt.show()
```

# Data transformation

- Seaborn can subset based on the datatype column to give us individual lines for the daily maximum temperature, minimum temperature, and temperature at the time of observation:



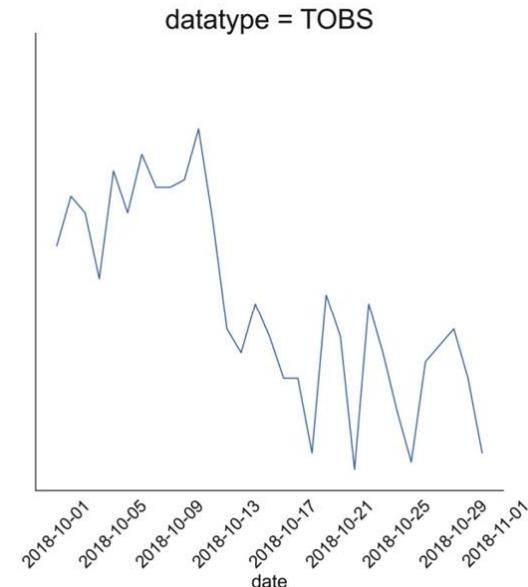
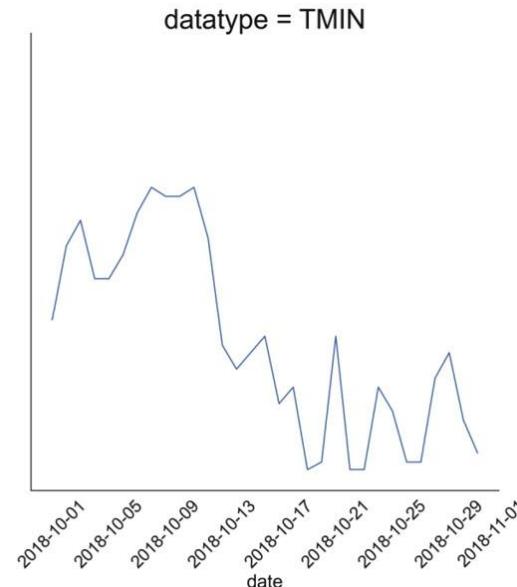
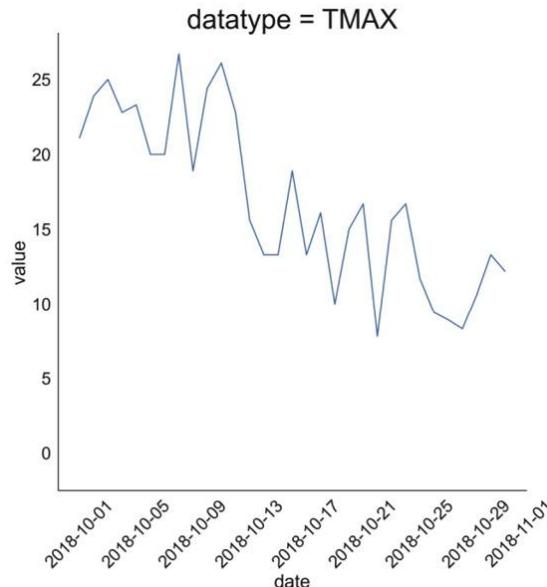
# Data transformation

- Seaborn lets us specify the column to use for hue, which colored the lines in Figure by the temperature type.
- We aren't limited to this, though; with long format data, we can easily facet our plots:

```
>>> sns.set(  
...     rc={'figure.figsize': (20, 10)},  
...     style='white', font_scale=2  
... )  
>>> g = sns.FacetGrid(long_df, col='datatype', height=10)  
>>> g.map(plt.plot, 'date', 'value')  
>>> g.set_titles(size=25)  
>>> g.set_xticklabels(rotation=45)  
>>> plt.show()
```

# Data transformation

- Seaborn can use long format data to create subplots for each distinct value in the datatype column:



# Data enrichment

The following are ways to enhance our data using the original data:

- **Adding new columns:** Using functions on the data from existing columns to create new values.
- **Binning:** Turning continuous data or discrete data with many distinct values into buckets, which makes the column discrete while letting us control the number of possible values in the column.
- **Aggregating:** Rolling up the data and summarizing it.
- **Resampling:** Aggregating time series data at specific intervals.

# Exploring an API to find and collect temperature data

- For this section, we will be working in the 2-using\_the\_weather\_api.ipynb notebook to request temperature data from the NCEI API.
- As we learned in Working with Pandas DataFrames, we can use the requests library to interact with APIs.
- In the following code block, we import the requests library and create a convenience function for making the requests to a specific endpoint, sending our token along.

# Exploring an API to find and collect temperature data

- To use this function, we need to provide a token, as indicated in bold:

```
>>> import requests
>>> def make_request(endpoint, payload=None):
...     """
...     Make a request to a specific endpoint on the
...     weather API passing headers and optional payload.
...     Parameters:
...         - endpoint: The endpoint of the API you want to
...                     make a GET request to.
...         - payload: A dictionary of data to pass along
...                     with the request.
...
...     Returns:
...         A response object.
...
...     """
...     return requests.get(
...         'https://www.ncdc.noaa.gov/cdo-web/'
...         f'api/v2/{endpoint}',
...         headers={'token': 'PASTE_YOUR_TOKEN_HERE'},
...         params=payload
...     )
```

# Exploring an API to find and collect temperature data

- Let's check which datasets have data within the date range of October 1, 2018 through today:

```
>>> response = \  
...     make_request('datasets', {'startdate': '2018-10-01'})
```

# Exploring an API to find and collect temperature data

- Remember that we check the `status_code` attribute to make sure the request was successful.
- Alternatively, we can use the `ok` attribute to get a Boolean indicator if everything went as expected:

```
>>> response.status_code
```

```
200
```

```
>>> response.ok
```

```
True
```

# Exploring an API to find and collect temperature data

- Once we have our response, we can use the json() method to get the payload.
- Then, we can use dictionary methods to determine which part we want to look at:

```
>>> payload = response.json()  
>>> payload.keys()  
dict_keys(['metadata', 'results'])
```

# Exploring an API to find and collect temperature data

- The metadata portion of the JSON payload tells us information about the result, while the results section contains the actual results.
- Let's see how much data we got back, so that we know whether we can print the results or whether we should try to limit the output:

```
>>> payload['metadata']
{'resultset': {'offset': 1, 'count': 11, 'limit': 25}}
```

# Exploring an API to find and collect temperature data

- We got back 11 rows, so let's see what fields are in the results portion of the JSON payload.
- The results key contains a list of dictionaries.
- If we select the first one, we can look at the keys to see what fields the data contains.
- We can then reduce the output to the fields we care about:

```
>>> payload['results'][0].keys()  
dict_keys(['uid', 'mindate', 'maxdate', 'name',  
          'datacoverage', 'id'])
```

# Exploring an API to find and collect temperature data

- For our purposes, we want to look at the IDs and names of the datasets, so let's use a list comprehension to look at those only:

```
>>> [(data['id'], data['name']) for data in payload['results']]  
[('GHCND', 'Daily Summaries'),  
 ('GSOM', 'Global Summary of the Month'),  
 ('GSOY', 'Global Summary of the Year'),  
 ('NEXRAD2', 'Weather Radar (Level II)'),  
 ('NEXRAD3', 'Weather Radar (Level III)'),  
 ('NORMAL_ANN', 'Normals Annual/Seasonal'),  
 ('NORMAL_DLY', 'Normals Daily'),  
 ('NORMAL_HLY', 'Normals Hourly'),  
 ('NORMAL_MLY', 'Normals Monthly'),  
 ('PRECIP_15', 'Precipitation 15 Minute'),  
 ('PRECIP_HLY', 'Precipitation Hourly')]
```

# Exploring an API to find and collect temperature data

- Here, we can print the JSON payload since it isn't that large (only nine entries):

```
>>> response = make_request(  
...     'datacategories', payload={'datasetid': 'GHCND'}  
... )  
>>> response.status_code  
200  
>>> response.json()['results']  
[{'name': 'Evaporation', 'id': 'EVAP'},  
 {'name': 'Land', 'id': 'LAND'},  
 {'name': 'Precipitation', 'id': 'PRCP'},  
 {'name': 'Sky cover & clouds', 'id': 'SKY'},  
 {'name': 'Sunshine', 'id': 'SUN'},  
 {'name': 'Air Temperature', 'id': 'TEMP'},  
 {'name': 'Water', 'id': 'WATER'},  
 {'name': 'Wind', 'id': 'WIND'},  
 {'name': 'Weather Type', 'id': 'WXTYPE'}]
```

# Exploring an API to find and collect temperature data

- We will use a list comprehension once again to only print the names and IDs; this is still a rather large list, so the output has been abbreviated:

```
>>> response = make_request(  
...     'datatypes',  
...     payload={'datacategoryid': 'TEMP', 'limit': 100}  
... )  
>>> response.status_code  
200  
>>> [(datatype['id'], datatype['name'])  
...     for datatype in response.json()['results']]  
[('CDSD', 'Cooling Degree Days Season to Date'),  
 ...,  
 ('TAVG', 'Average Temperature.'),  
 ('TMAX', 'Maximum temperature'),  
 ('TMIN', 'Minimum temperature'),  
 ('TOBS', 'Temperature at the time of observation')]
```

# Exploring an API to find and collect temperature data

- To determine a value for locationcategoryid, we must use the locationcategories endpoint:

```
>>> response = make_request(  
...     'locationcategories', payload={'datasetid': 'GHCND'}  
... )  
>>> response.status_code  
200
```

# Exploring an API to find and collect temperature data

- Note that we can use pprint from the Python standard library (<https://docs.python.org/3/library/pprint.html>) to print our JSON payload in an easier-to-read format:

```
>>> import pprint
>>> pprint.pprint(response.json())
{'metadata': {
    'resultset': {'count': 12, 'limit': 25, 'offset': 1}},
 'results': [{"id': 'CITY', 'name': 'City'},
              {"id": "CLIM_DIV", "name": "Climate Division"}, 
              {"id": "CLIM_REG", "name": "Climate Region"}, 
              {"id": "CNTRY", "name": "Country"}, 
              {"id": "CNTY", "name": "County"}, 
              ...,
              {"id": "ST", "name": "State"}, 
              {"id": "US_TERR", "name": "US Territory"}, 
              {"id": "ZIP", "name": "Zip Code"}]}
```

# Exploring an API to find and collect temperature data

- Each time, we are slicing the data in half, so when we grab the middle entry to test, we are moving closer to the value we seek:

```
>>> def get_item(name, what, endpoint, start=1, end=None):
...     """
...     Grab the JSON payload using binary search.
...
...     Parameters:
...         - name: The item to look for.
...         - what: Dictionary specifying what item `name` is.
...         - endpoint: Where to look for the item.
...         - start: The position to start at. We don't need
...             to touch this, but the function will manipulate
...             this with recursion.
...         - end: The last position of the items. Used to
...             find the midpoint, but like `start` this is not
...             something we need to worry about.
...
...     Returns: Dictionary of the information for the item
...             if found, otherwise an empty dictionary.
...     """
...     # find the midpoint to cut the data in half each time
...     mid = (start + (end or 1)) // 2
```

# Exploring an API to find and collect temperature data

```
...
...     # lowercase the name so this is not case-sensitive
...     name = name.lower()
...     # define the payload we will send with each request
...     payload = {
...         'datasetid': 'GHCND', 'sortfield': 'name',
...         'offset': mid, # we'll change the offset each time
...         'limit': 1 # we only want one value back
...     }
...
...     # make request adding additional filters from `what`
...     response = make_request(endpoint, {**payload, **what})
...
...     if response.ok:
...         payload = response.json()
...
...         # if ok, grab the end index from the response
...         # metadata the first time through
...         end = end or \
...             payload['metadata']['resultset']['count']
```

# Exploring an API to find and collect temperature data

```
...
...     # grab the lowercase version of the current name
...     current_name = \
...         payload['results'][0]['name'].lower()
...
...     # if what we are searching for is in the current
...     # name, we have found our item
...     if name in current_name:
...         # return the found item
...         return payload['results'][0]
...     else:
...         if start >= end:
...             # if start index is greater than or equal
...             # to end index, we couldn't find it
...             return {}
...         elif name < current_name:
...             # name comes before the current name in the
...             # alphabet => search further to the left
...             return get_item(name, what, endpoint,
...                             start, mid - 1)
...         elif name > current_name:
...             # name comes after the current name in the
...             # alphabet => search further to the right
...             return get_item(name, what, endpoint,
...                             mid + 1, end)
...     else:
...         # response wasn't ok, use code to determine why
...         print('Response not OK, '
...               f'status: {response.status_code}')
```

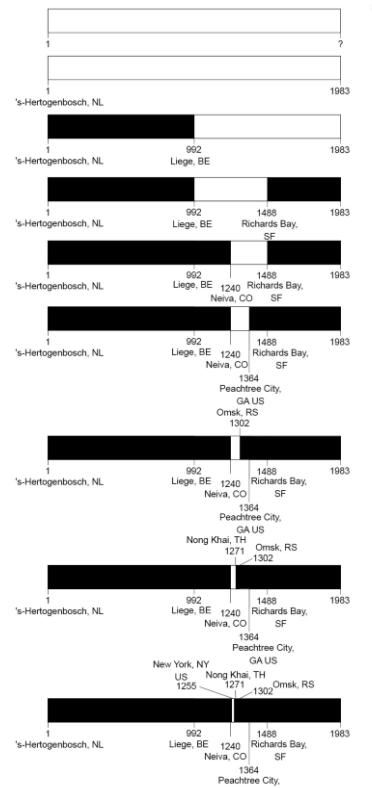
# Exploring an API to find and collect temperature data

- Now, let's use the binary search implementation to find the ID for New York City, which will be the value we will use for locationid in subsequent queries:

```
>>> nyc = get_item(  
...     'New York', {'locationcategoryid': 'CITY'}, 'locations'  
... )  
>>> nyc  
{'mindate': '1869-01-01',  
 'maxdate': '2021-01-14',  
 'name': 'New York, NY US',  
 'datacoverage': 1,  
 'id': 'CITY:US360019'}
```

# Exploring an API to find and collect temperature data

- In the following diagram, we can see how binary search eliminates sections of the list of locations systematically, which is represented by black on the number line (white means it is still possible that the desired value is in that section):



# Exploring an API to find and collect temperature data

- Using binary search again, we can grab the station ID for the Central Park station:

```
>>> central_park = get_item(  
...     'NY City Central Park',  
...     {'locationid': nyc['id']}, 'stations'  
... )  
>>> central_park  
{'elevation': 42.7,  
 'mindate': '1869-01-01',  
 'maxdate': '2020-01-13',  
 'latitude': 40.77898,  
 'name': 'NY CITY CENTRAL PARK, NY US',  
 'datacoverage': 1,  
 'id': 'GHCND:USW00094728',  
 'elevationUnit': 'METERS',  
 'longitude': -73.96925}
```

# Exploring an API to find and collect temperature data

- Now, let's request NYC's temperature data in Celsius for October 2018, recorded from Central Park.
- For this, we will use the data endpoint and provide all the parameters we picked up throughout our exploration of the API:

```
>>> response = make_request(  
...     'data',  
...     {'datasetid': 'GHCND',  
...      'stationid': central_park['id'],  
...      'locationid': nyc['id'],  
...      'startdate': '2018-10-01',  
...      'enddate': '2018-10-31',  
...      'datatypeid': ['TAVG', 'TMAX', 'TMIN'],  
...      'units': 'metric',  
...      'limit': 1000}  
... )  
>>> response.status_code  
200
```

# Exploring an API to find and collect temperature data

- Lastly, we will create a DataFrame object; since the results portion of the JSON payload is a list of dictionaries, we can pass it directly to pd.DataFrame():

```
>>> import pandas as pd  
>>> df = pd.DataFrame(response.json()['results'])  
>>> df.head()
```

# Exploring an API to find and collect temperature data

- We get back long format data.
- The datatype column is the temperature variable being measured, and the value column contains the measured temperature:

	<b>date</b>	<b>datatype</b>		<b>station</b>	<b>attributes</b>	<b>value</b>
<b>0</b>	2018-10-01T00:00:00	TMAX	GHCND:USW00094728	„W,2400		24.4
<b>1</b>	2018-10-01T00:00:00	TMIN	GHCND:USW00094728	„W,2400		17.2
<b>2</b>	2018-10-02T00:00:00	TMAX	GHCND:USW00094728	„W,2400		25.0
<b>3</b>	2018-10-02T00:00:00	TMIN	GHCND:USW00094728	„W,2400		18.3
<b>4</b>	2018-10-03T00:00:00	TMAX	GHCND:USW00094728	„W,2400		23.3

# Exploring an API to find and collect temperature data

- We asked for TAVG, TMAX, and TMIN, but notice that we didn't get TAVG.
- This is because the Central Park station isn't recording average temperature, despite being listed in the API as offering it—real-world data is dirty:

```
>>> df.datatype.unique()
array(['TMAX', 'TMIN'], dtype=object)
>>> if get_item(
...     'NY City Central Park',
...     {'locationid': nyc['id'], 'datatypeid': 'TAVG'},
...     'stations'
... ):
...     print('Found!')
Found!
```

# Cleaning data

- For this section, we will be using the nyc\_temperatures.csv file, which contains the maximum daily temperature (TMAX), minimum daily temperature (TMIN), and the average daily temperature (TAVG) from the LaGuardia Airport station in New York City for October 2018:

```
>>> import pandas as pd  
>>> df = pd.read_csv('data/nyc_temperatures.csv')  
>>> df.head()
```

# Cleaning data

- We retrieved long format data from the API; for our analysis, we want wide format data, but we will address that in the Pivoting DataFrames section, later in this lesson:

	date	datatype	station	attributes	value
0	2018-10-01T00:00:00	TAVG	GHCND:USW00014732	H,,S,	21.2
1	2018-10-01T00:00:00	TMAX	GHCND:USW00014732	,,W,2400	25.6
2	2018-10-01T00:00:00	TMIN	GHCND:USW00014732	,,W,2400	18.3
3	2018-10-02T00:00:00	TAVG	GHCND:USW00014732	H,,S,	22.7
4	2018-10-02T00:00:00	TMAX	GHCND:USW00014732	,,W,2400	26.1

# Renaming columns

- Since the API endpoint we used could return data of any units and category, it had to call that column value.
- We only pulled temperature data in Celsius, so all our observations have the same units.
- This means that we can rename the value column so that it's clear what data we are working with:

```
>>> df.columns
```

```
Index(['date', 'datatype', 'station', 'attributes', 'value'],  
      dtype='object')
```

# Renaming columns

- The DataFrame class has a `rename()` method that takes a dictionary mapping the old column name to the new column name.
- In addition to renaming the value column, let's rename the attributes column to flags since the API documentation mentions that that column contains flags for information about data collection:

```
>>> df.rename(  
...     columns={'value': 'temp_C', 'attributes': 'flags'},  
...     inplace=True  
... )
```

# Renaming columns

- Most of the time, pandas will return a new DataFrame object; however, since we passed in `inplace=True`, our original dataframe was updated instead.
- Always be careful with in-place operations, as they might be difficult or impossible to undo.
- Our columns now have their new names:

```
>>> df.columns
```

```
Index(['date', 'datatype', 'station', 'flags', 'temp_C'],  
      dtype='object')
```

# Renaming columns

- We can also do transformations on the column names with `rename()`.
- For instance, we can put all the column names in uppercase:

```
>>> df.rename(str.upper, axis='columns').columns  
Index(['DATE', 'DATATYPE', 'STATION', 'FLAGS', 'TEMP_C'],  
      dtype='object')
```

# Type conversion

- Note that, sometimes, we may have data that we believe should be a certain type, such as a date, but it is stored as a string; this could be for a very valid reason—data could be missing.
- In the case of missing data encoded as text (for example, ? or N/A), pandas will store it as a string when reading it in to allow for this data.
- It will be marked as object when we use the dtypes attribute on our dataframe.

# Type conversion

- That being said, let's examine the data types in our temperature data.
- Note that the date column isn't actually being stored as a datetime:

```
>>> df.dtypes  
date        object  
datatype    object  
station     object  
flags       object  
temp_C      float64  
dtype: object
```

# Type conversion

- We can use the pd.to\_datetime() function to convert it into a datetime:

```
>>> df.loc[:, 'date'] = pd.to_datetime(df.date)
```

```
>>> df.dtypes
```

```
date    datetime64[ns]
```

```
datatype    object
```

```
station    object
```

```
flags    object
```

```
temp_C    float64
```

```
dtype: object
```

# Type conversion

- This is much better.
- Now, we can get useful information when we summarize the date column:

```
>>> df.date.describe(datetime_is_numeric=True)
```

```
count           93
mean   2018-10-16 00:00:00
min    2018-10-01 00:00:00
25%    2018-10-08 00:00:00
50%    2018-10-16 00:00:00
75%    2018-10-24 00:00:00
max    2018-10-31 00:00:00
Name: date, dtype: object
```

# Type conversion

- For example, when working with a DatetimeIndex object, if we need to keep track of time zones, we can use the tz\_localize() method to associate our datetimes with a time zone:

```
>>> pd.date_range(start='2018-10-25', periods=2, freq='D')\
...     .tz_localize('EST')
DatetimeIndex(['2018-10-25 00:00:00-05:00',
               '2018-10-26 00:00:00-05:00'],
              dtype='datetime64[ns, EST]', freq=None)
```

# Type conversion

- This also works with Series and DataFrame objects that have an index of type DatetimeIndex.
- We can read in the CSV file again and, this time, specify that the date column will be our index and that we should parse any dates in the CSV file into datetimes:

```
>>> eastern = pd.read_csv(  
...      'data/nyc_temperatures.csv',  
...      index_col='date', parse_dates=True  
... ).tz_localize('EST')  
>>> eastern.head()
```

# Type conversion

- Note that we have added the Eastern Standard Time offset (-05:00 from UTC) to the datetimes in the index:

datatype	station	attributes	value
date			
2018-10-01 00:00:00-05:00	TAVG	GHCND:USW00014732	H,,S, 21.2
2018-10-01 00:00:00-05:00	TMAX	GHCND:USW00014732	,,W,2400 25.6
2018-10-01 00:00:00-05:00	TMIN	GHCND:USW00014732	,,W,2400 18.3
2018-10-02 00:00:00-05:00	TAVG	GHCND:USW00014732	H,,S, 22.7
2018-10-02 00:00:00-05:00	TMAX	GHCND:USW00014732	,,W,2400 26.1

# Type conversion

- We can use the `tz_convert()` method to change the time zone into a different one.
- Let's change our data into UTC:

```
>>> eastern.tz_convert('UTC').head()
```

# Type conversion

- Now, the offset is UTC (+00:00), but note that the time portion of the date is now 5 AM; this conversion took into account the -05:00 offset:

date	datatype	station	attributes	value
2018-10-01 05:00:00+00:00	TAVG	GHCND:USW00014732	H,,S,	21.2
2018-10-01 05:00:00+00:00	TMAX	GHCND:USW00014732	,,W,2400	25.6
2018-10-01 05:00:00+00:00	TMIN	GHCND:USW00014732	,,W,2400	18.3
2018-10-02 05:00:00+00:00	TAVG	GHCND:USW00014732	H,,S,	22.7
2018-10-02 05:00:00+00:00	TMAX	GHCND:USW00014732	,,W,2400	26.1

# Type conversion

- This is because the underlying data for a PeriodIndex object is stored as a PeriodArray object:

```
>>> eastern.tz_localize(None).to_period('M').index  
PeriodIndex(['2018-10', '2018-10', ..., '2018-10', '2018-10'],  
            dtype='period[M]', name='date', freq='M')
```

# Type conversion

- We can use the `to_timestamp()` method to convert our `PeriodIndex` object into a `DatetimeIndex` object; however, the datetimes all start at the first of the month now:

```
>>> eastern.tz_localize(None) \
...     .to_period('M').to_timestamp().index
DatetimeIndex(['2018-10-01', '2018-10-01', '2018-10-01', ...,
               '2018-10-01', '2018-10-01', '2018-10-01'],
              dtype='datetime64[ns]', name='date', freq=None)
```

# Type conversion

- Here, we will create a new one.
- Note that our original conversion of the dates modified the column, so, in order to illustrate that we can use `assign()`, we need to read our data in once more:

```
>>> df = pd.read_csv('data/nyc_temperatures.csv').rename(  
...      columns={'value': 'temp_C', 'attributes': 'flags'}  
... )  
>>> new_df = df.assign(  
...      date=pd.to_datetime(df.date),  
...      temp_F=(df.temp_C * 9/5) + 32  
... )  
>>> new_df.dtypes  
date           datetime64[ns]  
datatype        object  
station         object  
flags          object  
temp_C          float64  
temp_F          float64  
dtype: object  
>>> new_df.head()
```

# Type conversion

- We now have datetimes in the date column and a new column, temp\_F:

	<b>date</b>	<b>datatype</b>	<b>station</b>	<b>flags</b>	<b>temp_C</b>	<b>temp_F</b>
<b>0</b>	2018-10-01	TAVG	GHCND:USW00014732	H,,S,	21.2	70.16
<b>1</b>	2018-10-01	TMAX	GHCND:USW00014732	,,W,2400	25.6	78.08
<b>2</b>	2018-10-01	TMIN	GHCND:USW00014732	,,W,2400	18.3	64.94
<b>3</b>	2018-10-02	TAVG	GHCND:USW00014732	H,,S,	22.7	72.86
<b>4</b>	2018-10-02	TMAX	GHCND:USW00014732	,,W,2400	26.1	78.98

# Type conversion

- It is very common (and useful) to use lambda functions with `assign()`:

```
>>> df = df.assign(  
...     date=lambda x: pd.to_datetime(x.date),  
...     temp_C_whole=lambda x: x.temp_C.astype('int'),  
...     temp_F=lambda x: (x.temp_C * 9/5) + 32,  
...     temp_F_whole=lambda x: x.temp_F.astype('int'))  
>>> df.head()
```

# Type conversion

- If all the numbers are whole, they will be converted into integers (obviously, we will still get errors if the data isn't numeric at all):

	date	datatype	station	flags	temp_C	temp_C_whole	temp_F	temp_F_whole
0	2018-10-01	TAVG	GHCND:USW00014732	H,,S,	21.2	21	70.16	70
1	2018-10-01	TMAX	GHCND:USW00014732	,,W,2400	25.6	25	78.08	78
2	2018-10-01	TMIN	GHCND:USW00014732	,,W,2400	18.3	18	64.94	64
3	2018-10-02	TAVG	GHCND:USW00014732	H,,S,	22.7	22	72.86	72
4	2018-10-02	TMAX	GHCND:USW00014732	,,W,2400	26.1	26	78.98	78

# Type conversion

- We can use the `astype()` method to cast these into categories and look at the summary statistics:

```
>>> df_with_categories = df.assign(  
...     station=df.station.astype('category'),  
...     datatype=df.datatype.astype('category')  
... )  
>>> df_with_categories.dtypes  
date           datetime64[ns]  
datatype        category  
station         category  
flags           object  
temp_C          float64  
temp_C_whole    int64  
temp_F          float64  
temp_F_whole    int64  
dtype: object  
>>> df_with_categories.describe(include='category')
```

# Type conversion

- The summary statistics for categories are just like those for strings.
- We can see the number of non-null entries (count), the number of unique values (unique), the mode (top), and the number of occurrences of the mode (freq):

datatype	station
count	93
unique	3
top	TAVG GHCND:USW00014732
freq	31

# Type conversion

- The categories we just made don't have any order to them, but pandas does support this:

```
>>> pd.Categorical(  
...      ['med', 'med', 'low', 'high'],  
...      categories=['low', 'med', 'high'],  
...      ordered=True  
... )  
['med', 'med', 'low', 'high']  
Categories (3, object): ['low' < 'med' < 'high']
```

# Reordering, reindexing, and sorting data

- We will often find the need to sort our data by the values of one or many columns.
- Say we wanted to find the days that reached the highest temperatures in New York City during October 2018; we could sort our values by the temp\_C (or temp\_F) column in descending order and use head() to select the number of days we wanted to see.
- To accomplish this, we can use the sort\_values() method.
- Let's look at the top 10 days:

```
>>> df[df.datatype == 'TMAX']  
...     .sort_values(by='temp_C', ascending=False).head(10)
```

# Reordering, reindexing, and sorting data

		date	datatype	station	flags	temp_C	temp_C_whole	temp_F	temp_F_whole
19		2018-10-07	TMAX	GHCND:USW00014732	,,W,2400	27.8	27	82.04	82
28		2018-10-10	TMAX	GHCND:USW00014732	,,W,2400	27.8	27	82.04	82
31		2018-10-11	TMAX	GHCND:USW00014732	,,W,2400	26.7	26	80.06	80
10		2018-10-04	TMAX	GHCND:USW00014732	,,W,2400	26.1	26	78.98	78
4		2018-10-02	TMAX	GHCND:USW00014732	,,W,2400	26.1	26	78.98	78
1		2018-10-01	TMAX	GHCND:USW00014732	,,W,2400	25.6	25	78.08	78
25		2018-10-09	TMAX	GHCND:USW00014732	,,W,2400	25.6	25	78.08	78
7		2018-10-03	TMAX	GHCND:USW00014732	,,W,2400	25.0	25	77.00	77
13		2018-10-05	TMAX	GHCND:USW00014732	,,W,2400	22.8	22	73.04	73
22		2018-10-08	TMAX	GHCND:USW00014732	,,W,2400	22.8	22	73.04	73

# Reordering, reindexing, and sorting data

- The `sort_values()` method can be used with a list of column names to break ties.
- The order in which the columns are provided will determine the sort order, with each subsequent column being used to break ties.
- As an example, let's make sure the dates are sorted in ascending order when breaking ties:

```
>>> df[df.datatype == 'TMAX'].sort_values(  
...     by=['temp_C', 'date'], ascending=[False, True]  
... ).head(10)
```

# Reordering, reindexing, and sorting data

- Notice how October 2nd is now above October 4th, despite both having the same temperature reading:

	date	datatype	station	flags	temp_C	temp_C_whole	temp_F	temp_F_whole
19	2018-10-07	TMAX	GHCND:USW00014732	,,W,2400	27.8	27	82.04	82
28	2018-10-10	TMAX	GHCND:USW00014732	,,W,2400	27.8	27	82.04	82
31	2018-10-11	TMAX	GHCND:USW00014732	,,W,2400	26.7	26	80.06	80
4	2018-10-02	TMAX	GHCND:USW00014732	,,W,2400	26.1	26	78.98	78
10	2018-10-04	TMAX	GHCND:USW00014732	,,W,2400	26.1	26	78.98	78
1	2018-10-01	TMAX	GHCND:USW00014732	,,W,2400	25.6	25	78.08	78
25	2018-10-09	TMAX	GHCND:USW00014732	,,W,2400	25.6	25	78.08	78
7	2018-10-03	TMAX	GHCND:USW00014732	,,W,2400	25.0	25	77.00	77
13	2018-10-05	TMAX	GHCND:USW00014732	,,W,2400	22.8	22	73.04	73
22	2018-10-08	TMAX	GHCND:USW00014732	,,W,2400	22.8	22	73.04	73

# Reordering, reindexing, and sorting data

- Let's grab the top 10 days by average temperature this time:

```
>>> df[df.datatype == 'TAVG'].nlargest(n=10, columns='temp_C')
```

# Reordering, reindexing, and sorting data

- We get the warmest days (on average) in October:

	date	datatype	station	flags	temp_C	temp_C_whole	temp_F	temp_F_whole
27	2018-10-10	TAVG	GHCND:USW00014732	H,,S,	23.8	23	74.84	74
30	2018-10-11	TAVG	GHCND:USW00014732	H,,S,	23.4	23	74.12	74
18	2018-10-07	TAVG	GHCND:USW00014732	H,,S,	22.8	22	73.04	73
3	2018-10-02	TAVG	GHCND:USW00014732	H,,S,	22.7	22	72.86	72
6	2018-10-03	TAVG	GHCND:USW00014732	H,,S,	21.8	21	71.24	71
24	2018-10-09	TAVG	GHCND:USW00014732	H,,S,	21.8	21	71.24	71
9	2018-10-04	TAVG	GHCND:USW00014732	H,,S,	21.3	21	70.34	70
0	2018-10-01	TAVG	GHCND:USW00014732	H,,S,	21.2	21	70.16	70
21	2018-10-08	TAVG	GHCND:USW00014732	H,,S,	20.9	20	69.62	69
12	2018-10-05	TAVG	GHCND:USW00014732	H,,S,	20.3	20	68.54	68

# Reordering, reindexing, and sorting data

- For instance, the `sample()` method will give us randomly selected rows, which will lead to a jumbled index, so we can use `sort_index()` to order them afterward:

```
>>> df.sample(5, random_state=0).index  
Int64Index([2, 30, 55, 16, 13], dtype='int64')  
>>> df.sample(5, random_state=0).sort_index().index  
Int64Index([2, 13, 16, 30, 55], dtype='int64')
```

# Reordering, reindexing, and sorting data

- Let's use this knowledge to sort the columns of our dataframe alphabetically:

```
>>> df.sort_index(axis=1).head()
```

# Reordering, reindexing, and sorting data

- Having our columns in alphabetical order can come in handy when using `loc[]` because we can specify a range of columns with similar names; for example, we could now use `df.loc[:, 'station':'temp_F_whole']` to easily grab all of our temperature columns, along with the station information:

	datatype	date	flags	station	temp_C	temp_C_whole	temp_F	temp_F_whole
0	TAVG	2018-10-01	H,,S,	GHCND:USW00014732	21.2	21	70.16	70
1	TMAX	2018-10-01	,,W,2400	GHCND:USW00014732	25.6	25	78.08	78
2	TMIN	2018-10-01	,,W,2400	GHCND:USW00014732	18.3	18	64.94	64
3	TAVG	2018-10-02	H,,S,	GHCND:USW00014732	22.7	22	72.86	72
4	TMAX	2018-10-02	,,W,2400	GHCND:USW00014732	26.1	26	78.98	78

# Reordering, reindexing, and sorting data

- We must sort the index to see that they are the same:

```
>>> df.equals(df.sort_values(by='temp_C'))
```

False

```
>>> df.equals(df.sort_values(by='temp_C').sort_index())
```

True

# Reordering, reindexing, and sorting data

- Sometimes, we don't care too much about the numeric index, but we would like to use one (or more) of the other columns as the index instead.
- In this case, we can use the `set_index()` method.
- Let's set the date column as our index:

```
>>> df.set_index('date', inplace=True)  
>>> df.head()
```

# Reordering, reindexing, and sorting data

- Notice that the date column has moved to the far left where the index goes, and we no longer have the numeric index:

	datatype	station	flags	temp_C	temp_C_whole	temp_F	temp_F_whole
date							
2018-10-01	TAVG	GHCND:USW00014732	H,,S,	21.2	21	70.16	70
2018-10-01	TMAX	GHCND:USW00014732	,,W,2400	25.6	25	78.08	78
2018-10-01	TMIN	GHCND:USW00014732	,,W,2400	18.3	18	64.94	64
2018-10-02	TAVG	GHCND:USW00014732	H,,S,	22.7	22	72.86	72
2018-10-02	TMAX	GHCND:USW00014732	,,W,2400	26.1	26	78.98	78

# Reordering, reindexing, and sorting data

- These can also be combined to build ranges.
- Note that `loc[]` is optional when using ranges:

```
>>> df['2018-10-11':'2018-10-12']
```

# Reordering, reindexing, and sorting data

- This gives us the data from October 11, 2018 through October 12, 2018 (inclusive of both endpoints):

	datatype	station	flags	temp_C	temp_C_whole	temp_F	temp_F_whole
date							
2018-10-11	TAVG	GHCND:USW00014732	H,,S,	23.4	23	74.12	74
2018-10-11	TMAX	GHCND:USW00014732	,,W,2400	26.7	26	80.06	80
2018-10-11	TMIN	GHCND:USW00014732	,,W,2400	21.7	21	71.06	71
2018-10-12	TAVG	GHCND:USW00014732	H,,S,	18.3	18	64.94	64
2018-10-12	TMAX	GHCND:USW00014732	,,W,2400	22.2	22	71.96	71
2018-10-12	TMIN	GHCND:USW00014732	,,W,2400	12.2	12	53.96	53

# Reordering, reindexing, and sorting data

- We can use the `reset_index()` method to restore the date column:

```
>>> df['2018-10-11':'2018-10-12'].reset_index()
```

# Reordering, reindexing, and sorting data

- Our index now starts at 0, and the dates are now in a column called date.
- This is especially useful if we have data that we don't want to lose in the index, such as the date, but need to perform an operation as if it weren't in the index:

	date	datatype	station	flags	temp_C	temp_C_whole	temp_F	temp_F_whole
0	2018-10-11	TAVG	GHCND:USW00014732	H,,S,,	23.4	23	74.12	74
1	2018-10-11	TMAX	GHCND:USW00014732	,,W,2400	26.7	26	80.06	80
2	2018-10-11	TMIN	GHCND:USW00014732	,,W,2400	21.7	21	71.06	71
3	2018-10-12	TAVG	GHCND:USW00014732	H,,S,,	18.3	18	64.94	64
4	2018-10-12	TMAX	GHCND:USW00014732	,,W,2400	22.2	22	71.96	71
5	2018-10-12	TMIN	GHCND:USW00014732	,,W,2400	12.2	12	53.96	53

# Reordering, reindexing, and sorting data

- Let's read it in, setting the date column as the index and parsing the dates:

```
>>> sp = pd.read_csv(  
...     'data/sp500.csv', index_col='date', parse_dates=True  
... ).drop(columns=['adj_close']) # not using this column
```

# Reordering, reindexing, and sorting data

- In this case, it's day\_name():

```
>>> sp.head(10)\n...     .assign(day_of_week=lambda x: x.index.day_name())
```

# Reordering, reindexing, and sorting data

- Since the stock market is closed on the weekend (and holidays), we only have data for weekdays:

	high	low	open	close	volume	day_of_week
date						
2017-01-03	2263.879883	2245.129883	2251.570068	2257.830078	3770530000	Tuesday
2017-01-04	2272.820068	2261.600098	2261.600098	2270.750000	3764890000	Wednesday
2017-01-05	2271.500000	2260.449951	2268.179932	2269.000000	3761820000	Thursday
2017-01-06	2282.100098	2264.060059	2271.139893	2276.979980	3339890000	Friday
2017-01-09	2275.489990	2268.899902	2273.590088	2268.899902	3217610000	Monday
2017-01-10	2279.270020	2265.270020	2269.719971	2268.899902	3638790000	Tuesday
2017-01-11	2275.320068	2260.830078	2268.600098	2275.320068	3620410000	Wednesday
2017-01-12	2271.780029	2254.250000	2271.139893	2270.439941	3462130000	Thursday
2017-01-13	2278.679932	2271.510010	2272.739990	2274.639893	3081270000	Friday
2017-01-17	2272.080078	2262.810059	2269.139893	2267.889893	3584990000	Tuesday

# Reordering, reindexing, and sorting data

- The bitcoin data also contains OHLC data and volume traded, but it comes with a column called market\_cap that we don't need, so we have to drop that first:

```
>>> bitcoin = pd.read_csv(  
...     'data/bitcoin.csv', index_col='date', parse_dates=True  
... ).drop(columns=['market_cap'])
```

# Reordering, reindexing, and sorting data

- For example, each day's closing price will be the sum of the closing price of the S&P 500 and the closing price of bitcoin:

```
# every day's closing price = S&P 500 close + Bitcoin close
# (same for other metrics)
>>> portfolio = pd.concat([sp, bitcoin], sort=False) \
...     .groupby(level='date').sum()
>>> portfolio.head(10).assign(
...     day_of_week=lambda x: x.index.day_name()
... )
```

# Reordering, reindexing, and sorting data

- Now, if we examine our portfolio, we will see that we have values for every day of the week; so far, so good:

	high	low	open	close	volume	day_of_week
date						
2017-01-01	1003.080000	958.700000	963.660000	998.330000	147775008	Sunday
2017-01-02	1031.390000	996.700000	998.620000	1021.750000	222184992	Monday
2017-01-03	3307.959883	3266.729883	3273.170068	3301.670078	3955698000	Tuesday
2017-01-04	3432.240068	3306.000098	3306.000098	3425.480000	4109835984	Wednesday
2017-01-05	3462.600000	3170.869951	3424.909932	3282.380000	4272019008	Thursday
2017-01-06	3328.910098	3148.000059	3285.379893	3179.179980	3691766000	Friday
2017-01-07	908.590000	823.560000	903.490000	908.590000	279550016	Saturday
2017-01-08	942.720000	887.250000	908.170000	911.200000	158715008	Sunday
2017-01-09	3189.179990	3148.709902	3186.830088	3171.729902	3359486992	Monday
2017-01-10	3194.140020	3166.330020	3172.159971	3176.579902	3754598000	Tuesday

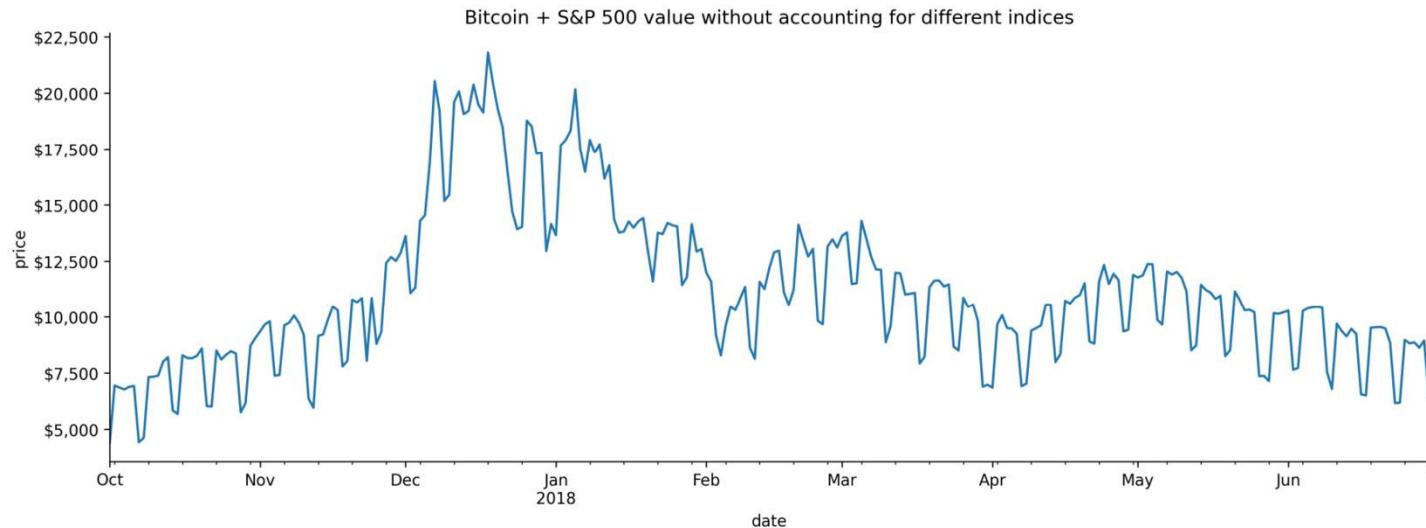
# Reordering, reindexing, and sorting data

- However, there is a problem with this approach, which is much easier to see with a visualization. Plotting will be covered in depth in Visualizing Data with Pandas and Matplotlib, and Plotting with Seaborn and Customization Techniques, so don't worry about the details for now:

```
>>> import matplotlib.pyplot as plt # module for plotting
>>> from matplotlib.ticker import StrMethodFormatter
# plot the closing price from Q4 2017 through Q2 2018
>>> ax = portfolio['2017-Q4':'2018-Q2'].plot(
...     y='close', figsize=(15, 5), legend=False,
...     title='Bitcoin + S&P 500 value without accounting '
...           'for different indices'
... )
# formatting
>>> ax.set_ylabel('price')
>>> ax.yaxis\
...     .set_major_formatter(StrMethodFormatter('${x:.0f}'))
>>> for spine in ['top', 'right']:
...     ax.spines[spine].set_visible(False)
# show the plot
>>> plt.show()
```

# Reordering, reindexing, and sorting data

- Notice how there is a cyclical pattern here? It is dropping every day the market is closed because the aggregation only had bitcoin data to sum for those days:



# Reordering, reindexing, and sorting data

- Forward-filling seems to be the best option, but since we aren't sure, we will see how this works on a few rows of the data first:

```
>>> sp.reindex(bitcoin.index, method='ffill').head(10)\n...     .assign(day_of_week=lambda x: x.index.day_name())
```

# Reordering, reindexing, and sorting data

- Notice any issues with this? Well, the volume traded (volume) column makes it seem like the days we used forward-filling for are actually days when the market is open:

	high	low	open	close	volume	day_of_week
date						
2017-01-01	NaN	NaN	NaN	NaN	NaN	Sunday
2017-01-02	NaN	NaN	NaN	NaN	NaN	Monday
2017-01-03	2263.879883	2245.129883	2251.570068	2257.830078	3.770530e+09	Tuesday
2017-01-04	2272.820068	2261.600098	2261.600098	2270.750000	3.764890e+09	Wednesday
2017-01-05	2271.500000	2260.449951	2268.179932	2269.000000	3.761820e+09	Thursday
2017-01-06	2282.100098	2264.060059	2271.139893	2276.979980	3.339890e+09	Friday
2017-01-07	2282.100098	2264.060059	2271.139893	2276.979980	3.339890e+09	Saturday
2017-01-08	2282.100098	2264.060059	2271.139893	2276.979980	3.339890e+09	Sunday
2017-01-09	2275.489990	2268.899902	2273.590088	2268.899902	3.217610e+09	Monday
2017-01-10	2279.270020	2265.270020	2269.719971	2268.899902	3.638790e+09	Tuesday

# Reordering, reindexing, and sorting data

- Lastly, we can use the `np.where()` function for the remaining columns, which allows us to build a vectorized if...else.
- It takes the following form:

`np.where(boolean condition, value if True, value if False)`

# Reordering, reindexing, and sorting data

- Since these come after the close column gets worked on, we will have the forward-filled value for close to use for the other columns where necessary:

```
>>> import numpy as np
>>> sp_reindexed = sp.reindex(bitcoin.index).assign(
...     # volume is 0 when the market is closed
...     volume=lambda x: x.volume.fillna(0),
...     # carry this forward
...     close=lambda x: x.close.fillna(method='ffill'),
...     # take the closing price if these aren't available
...     open=lambda x: \
...         np.where(x.open.isnull(), x.close, x.open),
...     high=lambda x: \
...         np.where(x.high.isnull(), x.close, x.high),
...     low=lambda x: np.where(x.low.isnull(), x.close, x.low)
... )
>>> sp_reindexed.head(10).assign(
...     day_of_week=lambda x: x.index.day_name()
... )
```

# Reordering, reindexing, and sorting data

- The OHLC prices are all equal to the closing price on Friday, the 6th:

	high	low	open	close	volume	day_of_week
date						
2017-01-01	NaN	NaN	NaN	NaN	0.000000e+00	Sunday
2017-01-02	NaN	NaN	NaN	NaN	0.000000e+00	Monday
2017-01-03	2263.879883	2245.129883	2251.570068	2257.830078	3.770530e+09	Tuesday
2017-01-04	2272.820068	2261.600098	2261.600098	2270.750000	3.764890e+09	Wednesday
2017-01-05	2271.500000	2260.449951	2268.179932	2269.000000	3.761820e+09	Thursday
2017-01-06	2282.100098	2264.060059	2271.139893	2276.979980	3.339890e+09	Friday
2017-01-07	2276.979980	2276.979980	2276.979980	2276.979980	0.000000e+00	Saturday
2017-01-08	2276.979980	2276.979980	2276.979980	2276.979980	0.000000e+00	Sunday
2017-01-09	2275.489990	2268.899902	2273.590088	2268.899902	3.217610e+09	Monday
2017-01-10	2279.270020	2265.270020	2269.719971	2268.899902	3.638790e+09	Tuesday

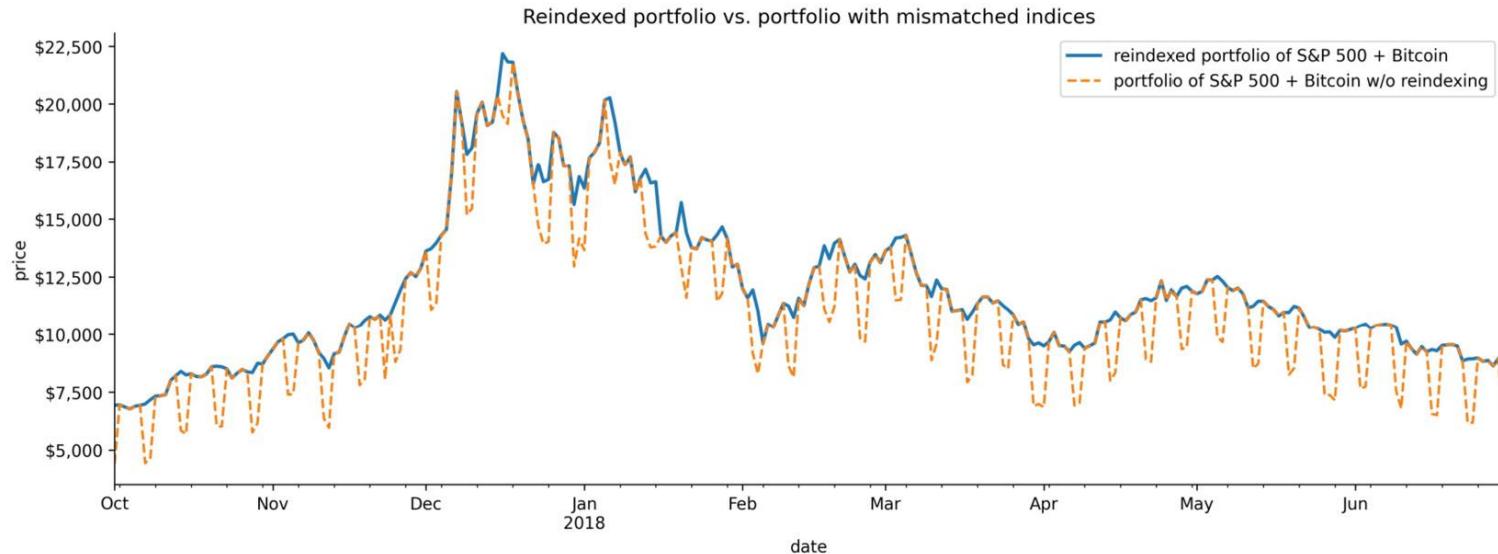
# Reordering, reindexing, and sorting data

- Now, let's recreate the portfolio with the reindexed S&P 500 data and use a visualization to compare it with the previous attempt (again, don't worry about the plotting code, which will be covered in Visualizing Data with Pandas and Matplotlib, and Plotting with Seaborn and Customization Techniques):

```
# every day's closing price = S&P 500 close adjusted for
# market closure + Bitcoin close (same for other metrics)
>>> fixed_portfolio = sp_reindexed + bitcoin
# plot the reindexed portfolio's close (Q4 2017 - Q2 2018)
>>> ax = fixed_portfolio['2017-Q4':'2018-Q2'].plot(
...     y='close', figsize=(15, 5), linewidth=2,
...     label='reindexed portfolio of S&P 500 + Bitcoin',
...     title='Reindexed portfolio vs.'
...             'portfolio with mismatched indices'
... )
# add line for original portfolio for comparison
>>> portfolio['2017-Q4':'2018-Q2'].plot(
...     y='close', ax=ax, linestyle='--',
...     label='portfolio of S&P 500 + Bitcoin w/o reindexing'
... )
# formatting
>>> ax.set_ylabel('price')
>>> ax.yaxis\
...     .set_major_formatter(StrMethodFormatter('${x:.0f}'))
>>> for spine in ['top', 'right']:
...     ax.spines[spine].set_visible(False)
# show the plot
>>> plt.show()
```

# Reordering, reindexing, and sorting data

- Keep this strategy in mind for the exercises in Financial Analysis – Bitcoin and the Stock Market:



# Reshaping data

- However, this isn't always as black and white as going from long format to wide format or vice versa.
- Consider the following data from the Exercises section:

	<b>ticker</b>	<b>date</b>	<b>high</b>	<b>low</b>	<b>open</b>	<b>close</b>	<b>volume</b>
<b>0</b>	AAPL	2018-01-02	43.075001	42.314999	42.540001	43.064999	102223600
<b>0</b>	AMZN	2018-01-02	1190.000000	1170.510010	1172.000000	1189.010010	2694500
<b>0</b>	FB	2018-01-02	181.580002	177.550003	177.679993	181.419998	18151900
<b>0</b>	GOOG	2018-01-02	1066.939941	1045.229980	1048.339966	1065.000000	1237600
<b>0</b>	NFLX	2018-01-02	201.649994	195.419998	196.100006	201.070007	10966900

# Reshaping data

- We will begin by importing pandas and reading in the long\_data.csv file, adding the temperature in Fahrenheit column (temp\_F), and performing some of the data cleaning we just learned about:

```
>>> import pandas as pd
>>> long_df = pd.read_csv(
...     'data/long_data.csv',
...     usecols=['date', 'datatype', 'value']
... ).rename(columns={'value': 'temp_C'}).assign(
...     date=lambda x: pd.to_datetime(x.date),
...     temp_F=lambda x: (x.temp_C * 9/5) + 32
... )
```

# Reshaping data

- Our long format data looks like this:

	datatype	date	temp_C	temp_F
0	TMAX	2018-10-01	21.1	69.98
1	TMIN	2018-10-01	8.9	48.02
2	TOBS	2018-10-01	13.9	57.02
3	TMAX	2018-10-02	23.9	75.02
4	TMIN	2018-10-02	13.9	57.02

# Transposing DataFrames

- While we will be pretty much only working with wide or long formats, pandas provides ways to restructure our data as we see fit, including taking the transpose (flipping the rows with the columns), which we may find useful to make better use of our display area when we're printing parts of our dataframe:

```
>>> long_df.set_index('date').head(6).T
```

# Transposing DataFrames

- Notice that the index is now in the columns, and that the column names are in the index:

date	2018-10-01	2018-10-01	2018-10-01	2018-10-02	2018-10-02	2018-10-02
datatype	TMAX	TMIN	TOBS	TMAX	TMIN	TOBS
temp_C	21.10	8.90	13.90	23.90	13.90	17.20
temp_F	69.98	48.02	57.02	75.02	57.02	62.96

# Pivoting DataFrames

- Let's pivot into wide format, where we have a column for each of the temperature measurements in Celsius and use the dates as the index:

```
>>> pivoted_df = long_df.pivot(  
...     index='date', columns='datatype', values='temp_C'  
... )  
>>> pivoted_df.head()
```

# Pivoting DataFrames

- Finally, the values for each combination of date and datatype are the corresponding temperatures in Celsius since we passed in `values='temp_C'`:

datatype	TMAX	TMIN	TOBS
date			
2018-10-01	21.1	8.9	13.9
2018-10-02	23.9	13.9	17.2
2018-10-03	25.0	15.6	16.1
2018-10-04	22.8	11.7	11.7
2018-10-05	23.3	11.7	18.9

# Pivoting DataFrames

- As we discussed at the beginning of this lesson, with the data in wide format, we can easily get meaningful summary statistics with the `describe()` method:

```
>>> pivoted_df.describe()
```

# Pivoting DataFrames

- We can see that we have 31 observations for all three temperature measurements and that this month has a wide range of temperatures (highest daily maximum of 26.7°C and lowest daily minimum of -1.1°C):

datatype	TMAX	TMIN	TOBS
count	31.000000	31.000000	31.000000
mean	16.829032	7.561290	10.022581
std	5.714962	6.513252	6.596550
min	7.800000	-1.100000	-1.100000
25%	12.750000	2.500000	5.550000
50%	16.100000	6.700000	8.300000
75%	21.950000	13.600000	16.100000
max	26.700000	17.800000	21.700000

# Pivoting DataFrames

- We lost the temperature in Fahrenheit, though.
- If we want to keep it, we can provide multiple columns to values:

```
>>> pivoted_df = long_df.pivot(  
...     index='date', columns='datatype',  
...     values=['temp_C', 'temp_F'])  
...)  
>>> pivoted_df.head()
```

# Pivoting DataFrames

- However, we now get an extra level above the column names. This is called a hierarchical index:

datatype	temp_C			temp_F		
	TMAX	TMIN	TOBS	TMAX	TMIN	TOBS
date						
2018-10-01	21.1	8.9	13.9	69.98	48.02	57.02
2018-10-02	23.9	13.9	17.2	75.02	57.02	62.96
2018-10-03	25.0	15.6	16.1	77.00	60.08	60.98
2018-10-04	22.8	11.7	11.7	73.04	53.06	53.06
2018-10-05	23.3	11.7	18.9	73.94	53.06	66.02

# Pivoting DataFrames

- With this hierarchical index, if we want to select TMIN in Fahrenheit, we will first need to select temp\_F and then TMIN:

```
>>> pivoted_df['temp_F']['TMIN'].head()  
date  
2018-10-01    48.02  
2018-10-02    57.02  
2018-10-03    60.08  
2018-10-04    53.06  
2018-10-05    53.06  
Name: TMIN, dtype: float64
```

# Pivoting DataFrames

- This gives us an index of type MultiIndex, where the outermost level corresponds to the first element in the list provided to `set_index()`:

```
>>> multi_index_df = long_df.set_index(['date', 'datatype'])
>>> multi_index_df.head().index
MultiIndex([('2018-10-01', 'TMAX'),
            ('2018-10-01', 'TMIN'),
            ('2018-10-01', 'TOBS'),
            ('2018-10-02', 'TMAX'),
            ('2018-10-02', 'TMIN')],
           names=['date', 'datatype'])
>>> multi_index_df.head()
```

# Pivoting DataFrames

- Notice that we now have two levels in the index—date is the outermost level and datatype is the innermost:

		temp_C	temp_F
date	datatype		
2018-10-01	TMAX	21.1	69.98
	TMIN	8.9	48.02
	TOBS	13.9	57.02
2018-10-02	TMAX	23.9	75.02
	TMIN	13.9	57.02

# Pivoting DataFrames

- To unstack a different level, simply pass in the index of the level to unstack, where 0 is the leftmost and -1 is the rightmost, or the name of the level (if it has one).
- Here, we will use the default:

```
>>> unstacked_df = multi_index_df.unstack()  
>>> unstacked_df.head()
```

# Pivoting DataFrames

- In Aggregating Pandas DataFrames, we will discuss a way to squash this back into a single level of columns:

datatype	temp_C			temp_F		
	TMAX	TMIN	TOBS	TMAX	TMIN	TOBS
date						
2018-10-01	21.1	8.9	13.9	69.98	48.02	57.02
2018-10-02	23.9	13.9	17.2	75.02	57.02	62.96
2018-10-03	25.0	15.6	16.1	77.00	60.08	60.98
2018-10-04	22.8	11.7	11.7	73.04	53.06	53.06
2018-10-05	23.3	11.7	18.9	73.94	53.06	66.02

# Pivoting DataFrames

- We could append this to long\_df and set our index to the date and datatype columns, as we did previously:

```
>>> extra_data = long_df.append([{
...     'datatype': 'TAVG',
...     'date': '2018-10-01',
...     'temp_C': 10,
...     'temp_F': 50
... }]).set_index(['date', 'datatype']).sort_index()
>>> extra_data['2018-10-01':'2018-10-02']
```

# Pivoting DataFrames

- We now have four temperature measurements for October 1, 2018, but only three for the remaining days:

		temp_C	temp_F
	date	datatype	
2018-10-01		TAVG	10.0 50.00
		TMAX	21.1 69.98
		TMIN	8.9 48.02
		TOBS	13.9 57.02
2018-10-02		TMAX	23.9 75.02
		TMIN	13.9 57.02
		TOBS	17.2 62.96

# Pivoting DataFrames

- Using unstack(), as we did previously, will result in NaN values for most of the TAVG data:

```
>>> extra_data.unstack().head()
```

# Pivoting DataFrames

- Take a look at the TAVG columns after we unstack:

datatype	temp_C					temp_F			
	TAVG	TMAX	TMIN	TOBS	TAVG	TMAX	TMIN	TOBS	
date									
2018-10-01	10.0	21.1	8.9	13.9	50.0	69.98	48.02	57.02	
2018-10-02	NaN	23.9	13.9	17.2	NaN	75.02	57.02	62.96	
2018-10-03	NaN	25.0	15.6	16.1	NaN	77.00	60.08	60.98	
2018-10-04	NaN	22.8	11.7	11.7	NaN	73.04	53.06	53.06	
2018-10-05	NaN	23.3	11.7	18.9	NaN	73.94	53.06	66.02	

# Pivoting DataFrames

- To address this, we can pass in an appropriate `fill_value`. However, we are restricted to passing in a value for this, not a strategy (as we saw when we discussed reindexing), so while there is no good value for this case, we can use `-40` to illustrate how this works:

```
>>> extra_data.unstack(fill_value=-40).head()
```

# Pivoting DataFrames

- Note that, in practice, it is better to be explicit about the missing data if we are sharing this with others and leave the NaN values:

datatype	temp_C				temp_F			
	TAVG	TMAX	TMIN	TOBS	TAVG	TMAX	TMIN	TOBS
date								
2018-10-01	10.0	21.1	8.9	13.9	50.0	69.98	48.02	57.02
2018-10-02	-40.0	23.9	13.9	17.2	-40.0	75.02	57.02	62.96
2018-10-03	-40.0	25.0	15.6	16.1	-40.0	77.00	60.08	60.98
2018-10-04	-40.0	22.8	11.7	11.7	-40.0	73.04	53.06	53.06
2018-10-05	-40.0	23.3	11.7	18.9	-40.0	73.94	53.06	66.02

# Melting DataFrames

- To go from wide format to long format, we need to melt the data.
- Melting undoes a pivot.
- For this example, we will read in the data from the `wide_data.csv` file:

```
>>> wide_df = pd.read_csv('data/wide_data.csv')
>>> wide_df.head()
```

# Melting DataFrames

- Our wide data contains a column for the date and a column for each temperature measurement we have been working with:

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

# Melting DataFrames

- Now, let's use the melt() method to turn the wide format data into long format:

```
>>> melted_df = wide_df.melt(  
...     id_vars='date', value_vars=['TMAX', 'TMIN', 'TOBS'],  
...     value_name='temp_C', var_name='measurement'  
... )  
>>> melted_df.head()
```

# Melting DataFrames

- We now have just three columns; the date, the temperature reading in Celsius (temp\_C), and a column indicating which temperature measurement is in that row's temp\_C cell (measurement):

	<b>date</b>	<b>measurement</b>	<b>temp_C</b>
<b>0</b>	2018-10-01	TMAX	21.1
<b>1</b>	2018-10-02	TMAX	23.9
<b>2</b>	2018-10-03	TMAX	25.0
<b>3</b>	2018-10-04	TMAX	22.8
<b>4</b>	2018-10-05	TMAX	23.3

# Melting DataFrames

- We can do the following to get a similar output to the melt() method:

```
>>> wide_df.set_index('date', inplace=True)
>>> stacked_series = wide_df.stack() # put datatypes in index
>>> stacked_series.head()
date
2018-10-01    TMAX    21.1
                  TMIN     8.9
                  TOBS    13.9
2018-10-02    TMAX    23.9
                  TMIN    13.9
dtype: float64
```

# Melting DataFrames

- Notice that the result came back as a Series object, so we will need to create the DataFrame object once more.
- We can use the `to_frame()` method and pass in a name to use for the column once it is a dataframe:

```
>>> stacked_df = stacked_series.to_frame('values')
>>> stacked_df.head()
```

# Melting DataFrames

- Now, we have a dataframe with a multi-level index, containing date and datatype, with values as the only column.
- Notice, however, that only the date portion of our index has a name:

	date	values
2018-10-01	TMAX	21.1
	TMIN	8.9
2018-10-02	TOBS	13.9
	TMAX	23.9
	TMIN	13.9

# Melting DataFrames

- However, this level was never named, so it shows up as None, but we know that the level should be called datatype:

```
>>> stacked_df.head().index  
MultiIndex([('2018-10-01', 'TMAX'),  
            ('2018-10-01', 'TMIN'),  
            ('2018-10-01', 'TOBS'),  
            ('2018-10-02', 'TMAX'),  
            ('2018-10-02', 'TMIN')],  
           names=['date', None])
```

# Melting DataFrames

- We can use the `set_names()` method to address this:

```
>>> stacked_df.index\  
...     .set_names(['date', 'datatype'], inplace=True)  
>>> stacked_df.index.names  
FrozenList(['date', 'datatype'])
```

# Handling duplicate, missing, or invalid data

- We will be working in the 5-handling\_data\_issues.ipynb notebook and using the dirty\_data.csv file.
- Let's start by importing pandas and reading in the data:

```
>>> import pandas as pd  
>>> df = pd.read_csv('data/dirty_data.csv')
```

# Handling duplicate, missing, or invalid data

It contains the following fields:

- PRCP: Precipitation in millimeters
- SNOW: Snowfall in millimeters
- SNWD: Snow depth in millimeters
- TMAX: Maximum daily temperature in Celsius
- TMIN: Minimum daily temperature in Celsius
- TOBS: Temperature at the time of observation in Celsius
- WESF: Water equivalent of snow in millimeters

# Finding the problematic data

- In Working with Pandas DataFrames, we learned the importance of examining our data when we get it; it's not a coincidence that many of the ways to inspect the data will help us find these issues.
- Examining the results of calling head() and tail() on the data is always a good first step:

```
>>> df.head()
```

# Finding the problematic data

- Lastly, we can observe many NaN values in several columns, including the inclement\_weather column, which appears to also contain Boolean values:

	date	station	PRCP	SNOW	SNWD	TMAX	TMIN	TOBS	WESF	inclement_weather
0	2018-01-01T00:00:00		?	0.0	0.0	-inf	5505.0	-40.0	NaN	NaN
1	2018-01-01T00:00:00		?	0.0	0.0	-inf	5505.0	-40.0	NaN	NaN
2	2018-01-01T00:00:00		?	0.0	0.0	-inf	5505.0	-40.0	NaN	NaN
3	2018-01-02T00:00:00	GHCND:USC00280907	0.0	0.0	-inf	-8.3	-16.1	-12.2	NaN	False
4	2018-01-03T00:00:00	GHCND:USC00280907	0.0	0.0	-inf	-4.4	-13.9	-13.3	NaN	False

# Finding the problematic data

- Using `describe()`, we can see if we have any missing data and look at the 5-number summary to spot potential issues:

```
>>> df.describe()
```

# Finding the problematic data

- If unknowns were encoded with another value, say 40°C, we couldn't be sure it wasn't actual data:

	PRCP	SNOW	SNWD	TMAX	TMIN	TOBS	WESF
<b>count</b>	765.000000	577.000000	577.0	765.000000	765.000000	398.000000	11.000000
<b>mean</b>	5.360392	4.202773	NaN	2649.175294	-15.914379	8.632161	16.290909
<b>std</b>	10.002138	25.086077	NaN	2744.156281	24.242849	9.815054	9.489832
<b>min</b>	0.000000	0.000000	-inf	-11.700000	-40.000000	-16.100000	1.800000
<b>25%</b>	0.000000	0.000000	NaN	13.300000	-40.000000	0.150000	8.600000
<b>50%</b>	0.000000	0.000000	NaN	32.800000	-11.100000	8.300000	19.300000
<b>75%</b>	5.800000	0.000000	NaN	5505.000000	6.700000	18.300000	24.900000
<b>max</b>	61.700000	229.000000	inf	5505.000000	23.900000	26.100000	28.700000

# Finding the problematic data

```
>>> df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 765 entries, 0 to 764
Data columns (total 10 columns):
 #   Column           Non-Null Count  Dtype  
--- 
 0   date             765 non-null    object  
 1   station          765 non-null    object  
 2   PRCP             765 non-null    float64 
 3   SNOW             577 non-null    float64 
 4   SNWD             577 non-null    float64 
 5   TMAX             765 non-null    float64 
 6   TMIN             765 non-null    float64 
 7   TOBS             398 non-null    float64 
 8   WESF             11 non-null     float64 
 9   inclement_weather 408 non-null    object  
dtypes: float64(7), object(3)
memory usage: 59.9+ KB
```

# Finding the problematic data

- This means that we will need to combine checks for each of the columns with the | (bitwise OR) operator:

```
>>> contain_nulls = df[  
...     df.SNOW.isna() | df.SNWD.isna() | df.T0BS.isna()  
...     | df.WESF.isna() | df.inclement_weather.isna()  
... ]  
>>> contain_nulls.shape[0]  
765  
>>> contain_nulls.head(10)
```

# Finding the problematic data

- Looking at the top 10 rows, we can see some NaN values in each of these rows:

	date	station	PRCP	SNOW	SNWD	TMAX	TMIN	TOBS	WESF	inclement_weather
0	2018-01-01T00:00:00		?	0.0	0.0	-inf	5505.0	-40.0	NaN	NaN
1	2018-01-01T00:00:00		?	0.0	0.0	-inf	5505.0	-40.0	NaN	NaN
2	2018-01-01T00:00:00		?	0.0	0.0	-inf	5505.0	-40.0	NaN	NaN
3	2018-01-02T00:00:00	GHCND:USC00280907	0.0	0.0	-inf	-8.3	-16.1	-12.2	NaN	False
4	2018-01-03T00:00:00	GHCND:USC00280907	0.0	0.0	-inf	-4.4	-13.9	-13.3	NaN	False
5	2018-01-03T00:00:00	GHCND:USC00280907	0.0	0.0	-inf	-4.4	-13.9	-13.3	NaN	False
6	2018-01-03T00:00:00	GHCND:USC00280907	0.0	0.0	-inf	-4.4	-13.9	-13.3	NaN	False
7	2018-01-04T00:00:00		?	20.6	229.0	inf	5505.0	-40.0	NaN	19.3
8	2018-01-04T00:00:00		?	20.6	229.0	inf	5505.0	-40.0	NaN	19.3
9	2018-01-05T00:00:00		?	0.3	NaN	NaN	5505.0	-40.0	NaN	NaN

# Finding the problematic data

- Note that we can't check whether the value of the column is equal to NaN because NaN is not equal to anything:

```
>>> import numpy as np  
>>> df[df.inclement_weather == 'NaN'].shape[0] # doesn't work  
0  
>>> df[df.inclement_weather == np.nan].shape[0] # doesn't work  
0
```

# Finding the problematic data

- We must use the aforementioned options (`isna()`/`isnull()`):

```
>>> df[df.inclement_weather.isna()].shape[0] # works  
357
```

# Finding the problematic data

- Note that inf and -inf are actually np.inf and -np.inf.
- Therefore, we can find the number of rows with inf or -inf values by doing the following:

```
>>> df[df.SNWD.isin([-np.inf, np.inf])].shape[0]  
577
```

# Finding the problematic data

- This only tells us about a single column, though, so we could write a function that will use a dictionary comprehension to return the number of infinite values per column in our dataframe:

```
>>> def get_inf_count(df):
...     """Find the number of inf/-inf values per column"""
...     return {
...         col: df[
...             df[col].isin([np.inf, -np.inf])
...         ].shape[0] for col in df.columns
...     }
```

# Finding the problematic data

- Using our function, we find that the SNWD column is the only column with infinite values, but the majority of the values in the column are infinite:

```
>>> get_inf_count(df)
{'date': 0, 'station': 0, 'PRCP': 0, 'SNOW': 0, 'SNWD': 577,
 'TMAX': 0, 'TMIN': 0, 'TOBS': 0, 'WESF': 0,
 'inclement_weather': 0}
```

# Finding the problematic data

- In addition, we will use the T attribute to transpose the data for easier viewing:

```
>>> pd.DataFrame({  
...     'np.inf Snow Depth':  
...         df[df.SNWD == np.inf].SNOW.describe(),  
...     '-np.inf Snow Depth':  
...         df[df.SNWD == -np.inf].SNOW.describe()  
... }).T
```

# Finding the problematic data

- They most certainly aren't that, but we can't decide what they should be, so it's probably best to leave them alone or not look at this column:

	<b>count</b>	<b>mean</b>	<b>std</b>	<b>min</b>	<b>25%</b>	<b>50%</b>	<b>75%</b>	<b>max</b>
<b>np.inf Snow Depth</b>	24.0	101.041667	74.498018	13.0	25.0	120.5	152.0	229.0
<b>-np.inf Snow Depth</b>	553.0	0.000000	0.000000	0.0	0.0	0.0	0.0	0.0

# Finding the problematic data

- We are working with a year of data, but somehow, we have 765 rows, so we should check why.
- The only columns we have yet to inspect are the date and station columns.
- We can use the describe() method to see the summary statistics for them:

```
>>> df.describe(include='object')
```

# Finding the problematic data

- We also know that ? occurs 367 times (765 - 398), without the need to use value\_counts():

	date	station	inclement_weather
<b>count</b>	765	765	408
<b>unique</b>	324	2	2
<b>top</b>	2018-07-05T00:00:00	GHCND:USC00280907	False
<b>freq</b>	8	398	384

# Finding the problematic data

- We can use the result of the duplicated() method as a Boolean mask to find the duplicate rows:

```
>>> df[df.duplicated()].shape[0]
```

```
284
```

# Finding the problematic data

- However, if we pass in `keep=False`, we will get all the rows that are present more than once, not just each additional appearance they make:

```
>>> df[df.duplicated(keep=False)].shape[0]  
482
```

# Finding the problematic data

- However, we don't know if this is actually a problem:

```
>>> df[df.duplicated(['date', 'station'])].shape[0]
```

284

- Now, let's examine a few of the duplicated rows:

```
>>> df[df.duplicated()].head()
```

# Finding the problematic data

- Just looking at the first five rows shows us that some rows are repeated at least three times.
- Remember that the default behavior of `duplicated()` is to not show the first occurrence, which means that rows 1 and 2 have another matching value in the data (same for rows 5 and 6):

	date	station	PRCP	SNOW	SNWD	TMAX	TMIN	TOBS	WESF	inclement_weather
1	2018-01-01T00:00:00		?	0.0	0.0	-inf	5505.0	-40.0	NaN	NaN
2	2018-01-01T00:00:00		?	0.0	0.0	-inf	5505.0	-40.0	NaN	NaN
5	2018-01-03T00:00:00	GHCND:USC00280907	0.0	0.0	-inf	-4.4	-13.9	-13.3	NaN	False
6	2018-01-03T00:00:00	GHCND:USC00280907	0.0	0.0	-inf	-4.4	-13.9	-13.3	NaN	False
8	2018-01-04T00:00:00		?	20.6	229.0	inf	5505.0	-40.0	NaN	19.3

# Mitigating the issues

- If we then decide to remove duplicate rows using the date column and keep the data for the station that wasn't ?, in the case of duplicates, we will lose all data we have for the WESF column because the ? station is the only one reporting WESF measurements:

```
>>> df[df.WESF.notna()].station.unique()  
array(['?'], dtype=object)
```

# Mitigating the issues

- One satisfactory solution in this case may be to carry out the following actions:
- Perform type conversion on the date column:

```
>>> df.date = pd.to_datetime(df.date)
```

- Save the WESF column as a series:

```
>>> station_qm_wesf = df[df.station == '?']\n...     .drop_duplicates('date').set_index('date').WESF
```

# Mitigating the issues

- Sort the dataframe by the station column in descending order to put the station with no ID (?) last:

```
>>> df.sort_values(  
...     'station', ascending=False, inplace=True  
... )
```

# Mitigating the issues

- Remove rows that are duplicated based on the date, keeping the first occurrences, which will be ones where the station column has an ID (if that station has measurements).

```
>>> df_deduped = df.drop_duplicates('date')
```

# Mitigating the issues

- Drop the station column and set the index to the date column (so that it matches the WESF data):

```
>>> df_deduped = df_deduped.drop(columns='station')\n...     .set_index('date').sort_index()
```

# Mitigating the issues

- Since both df\_deduped and station\_qm\_wesf are using the date as the index, the values are properly matched to the appropriate date:

```
>>> df_deduped = df_deduped.assign(WESF=
...     lambda x: x.WESF.combine_first(station_qm_wesf)
... )
```

# Mitigating the issues

- Let's take a look at the result using the aforementioned implementation:

```
>>> df_deduped.shape  
(324, 8)  
>>> df_deduped.head()
```

# Mitigating the issues

- We are now left with 324 rows—one for each unique date in our data.
- We were able to save the WESF column by putting it alongside the data from the other station:

	PRCP	SNOW	SNWD	TMAX	TMIN	TOBS	WESF	inclement_weather
date								
2018-01-01	0.0	0.0	-inf	5505.0	-40.0	NaN	NaN	NaN
2018-01-02	0.0	0.0	-inf	-8.3	-16.1	-12.2	NaN	False
2018-01-03	0.0	0.0	-inf	-4.4	-13.9	-13.3	NaN	False
2018-01-04	20.6	229.0	inf	5505.0	-40.0	NaN	19.3	True
2018-01-05	14.2	127.0	inf	-4.4	-13.9	-13.9	NaN	True

# Mitigating the issues

- To drop all the rows with any null data (this doesn't have to be true for all the columns of the row, so be careful), we use the `dropna()` method; in our case, this leaves us with just 4 rows:

```
>>> df_deduped.dropna().shape  
(4, 8)
```

# Mitigating the issues

- We can change the default behavior to only drop a row if all the columns are null with the how argument, except this doesn't get rid of anything:

```
>>> df_deduped.dropna(how='all').shape # default is 'any'  
(324, 8)
```

# Mitigating the issues

- This can be achieved with the subset argument:

```
>>> df_deduped.dropna(  
...     how='all', subset=['inclement_weather', 'SNOW', 'SNWD']  
... ).shape  
(293, 8)
```

# Mitigating the issues

- For example, if we say that at least 75% of the rows must be null to drop the column, we will drop the WESF column:

```
>>> df_deduped.dropna(  
...     axis='columns',  
...     thresh=df_deduped.shape[0] * .75 # 75% of rows  
... ).columns  
Index(['PRCP', 'SNOW', 'SNWD', 'TMAX', 'TMIN', 'TOBS',  
       'inclement_weather'],  
      dtype='object')
```

# Mitigating the issues

- Note that this can be done in-place (again, as a general rule of thumb, we should use caution with in-place operations):

```
>>> df_deduped.loc[:, 'WESF'].fillna(0, inplace=True)  
>>> df_deduped.head()
```

# Mitigating the issues

- The WESF column no longer contains NaN values:

	PRCP	SNOW	SNWD	TMAX	TMIN	TOBS	WESF	inclement_weather
date								
2018-01-01	0.0	0.0	-inf	5505.0	-40.0	NaN	0.0	NaN
2018-01-02	0.0	0.0	-inf	-8.3	-16.1	-12.2	0.0	False
2018-01-03	0.0	0.0	-inf	-4.4	-13.9	-13.3	0.0	False
2018-01-04	20.6	229.0	inf	5505.0	-40.0	NaN	19.3	True
2018-01-05	14.2	127.0	inf	-4.4	-13.9	-13.9	0.0	True

# Mitigating the issues

- We will also do so for TMIN, which currently uses -40°C for its placeholder, despite the coldest temperature ever recorded in NYC being -15°F (-26.1°C) on February 9, 1934 (<https://www.weather.gov/media/okx/Climate/CentralPark/extremes.pdf>):

```
>>> df_deduped = df_deduped.assign(  
...     TMAX=lambda x: x.TMAX.replace(5505, np.nan),  
...     TMIN=lambda x: x.TMIN.replace(-40, np.nan)  
... )
```

# Mitigating the issues

- Notice we don't have the 'nearest' option, like we did when we were reindexing, which would have been the best option; so, to illustrate how this works, let's use forward-filling:

```
>>> df_deduped.assign(  
...     TMAX=lambda x: x.TMAX.fillna(method='ffill'),  
...     TMIN=lambda x: x.TMIN.fillna(method='ffill')  
... ).head()
```

# Mitigating the issues

- Take a look at the TMAX and TMIN columns on January 1st and 4th.
- Both are NaN on the 1st because we don't have data before then to bring forward, but the 4th now has the same values as the 3rd:

	PRCP	SNOW	SNWD	TMAX	TMIN	TOBS	WESF	inclement_weather
date								
2018-01-01	0.0	0.0	-inf	NaN	NaN	NaN	0.0	NaN
2018-01-02	0.0	0.0	-inf	-8.3	-16.1	-12.2	0.0	False
2018-01-03	0.0	0.0	-inf	-4.4	-13.9	-13.3	0.0	False
2018-01-04	20.6	229.0	inf	-4.4	-13.9	NaN	19.3	True
2018-01-05	14.2	127.0	inf	-4.4	-13.9	-13.9	0.0	True

# Mitigating the issues

- If we want to handle the nulls and infinite values in the SNWD column, we can use the np.nan\_to\_num() function; it turns NaN into 0 and inf/-inf into very large positive/negative finite numbers, making it possible for machine learning models (discussed in Getting Started with Machine Learning in Python) to learn from this data:

```
>>> df_deduped.assign(  
...     SNWD=lambda x: np.nan_to_num(x.SNWD)  
... ).head()
```

# Mitigating the issues

- However, we don't know what to do with np.inf, and the large positive numbers, arguably, make this more confusing to interpret:

	PRCP	SNOW	SNWD	TMAX	TMIN	TOBS	WESF	inclement_weather
date								
2018-01-01	0.0	0.0	-1.797693e+308	NaN	NaN	NaN	0.0	NaN
2018-01-02	0.0	0.0	-1.797693e+308	-8.3	-16.1	-12.2	0.0	False
2018-01-03	0.0	0.0	-1.797693e+308	-4.4	-13.9	-13.3	0.0	False
2018-01-04	20.6	229.0	1.797693e+308	NaN	NaN	NaN	19.3	True
2018-01-05	14.2	127.0	1.797693e+308	-4.4	-13.9	-13.9	0.0	True

# Mitigating the issues

- To show how the upper bound works, we will use the snowfall (SNOW) as an estimate:

```
>>> df_deduped.assign(  
...     SNWD=lambda x: x.SNWD.clip(0, x.SNOW)  
... ).head()
```

# Mitigating the issues

- The values of SNWD for January 1st through 3rd are now 0 instead of -inf, while the values of SNWD for January 4th and 5th went from inf to that day's value for SNOW:

	PRCP	SNOW	SNWD	TMAX	TMIN	TOBS	WESF	inclement_weather
date								
2018-01-01	0.0	0.0	0.0	NaN	NaN	NaN	0.0	NaN
2018-01-02	0.0	0.0	0.0	-8.3	-16.1	-12.2	0.0	False
2018-01-03	0.0	0.0	0.0	-4.4	-13.9	-13.3	0.0	False
2018-01-04	20.6	229.0	229.0	NaN	NaN	NaN	19.3	True
2018-01-05	14.2	127.0	127.0	-4.4	-13.9	-13.9	0.0	True

# Mitigating the issues

- We can combine imputation with the `fillna()` method.
- As an example, let's fill in the `Nan` values for `TMAX` and `TMIN` with their medians and `TOBS` with the average of `TMIN` and `TMAX` (after imputing them):

```
>>> df_deduped.assign(  
...     TMAX=lambda x: x.TMAX.fillna(x.TMAX.median()),  
...     TMIN=lambda x: x.TMIN.fillna(x.TMIN.median()),  
...     # average of TMAX and TMIN  
...     TOBS=lambda x: x.TOBS.fillna((x.TMAX + x.TMIN) / 2)  
... ).head()
```

# Mitigating the issues

- This means that when we impute TOBS and also don't have TMAX and TMIN in the data, we get 10°C:

	PRCP	SNOW	SNWD	TMAX	TMIN	TOBS	WESF	inclement_weather
date								
2018-01-01	0.0	0.0	-inf	14.4	5.6	10.0	0.0	NaN
2018-01-02	0.0	0.0	-inf	-8.3	-16.1	-12.2	0.0	False
2018-01-03	0.0	0.0	-inf	-4.4	-13.9	-13.3	0.0	False
2018-01-04	20.6	229.0	inf	14.4	5.6	10.0	19.3	True
2018-01-05	14.2	127.0	inf	-4.4	-13.9	-13.9	0.0	True

# Mitigating the issues

- We will cover rolling calculations and apply() in Aggregating Pandas DataFrames, so this is just a preview:

```
>>> df_deduped.apply(lambda x:  
...     # Rolling 7-day median (covered in chapter 4).  
...     # we set min_periods (# of periods required for  
...     # calculation) to 0 so we always get a result  
...     x.fillna(x.rolling(7, min_periods=0).median())  
... ).head(10)
```

# Mitigating the issues

- In reality, it was slightly warmer that day (around -3°C):

	PRCP	SNOW	SNWD	TMAX	TMIN	TOBS	WESF	inclement_weather
date								
2018-01-01	0.0	0.0	-inf	NaN	NaN	NaN	0.0	NaN
2018-01-02	0.0	0.0	-inf	-8.30	-16.1	-12.20	0.0	False
2018-01-03	0.0	0.0	-inf	-4.40	-13.9	-13.30	0.0	False
2018-01-04	20.6	229.0	inf	-6.35	-15.0	-12.75	19.3	True
2018-01-05	14.2	127.0	inf	-4.40	-13.9	-13.90	0.0	True
2018-01-06	0.0	0.0	-inf	-10.00	-15.6	-15.00	0.0	False
2018-01-07	0.0	0.0	-inf	-11.70	-17.2	-16.10	0.0	False
2018-01-08	0.0	0.0	-inf	-7.80	-16.7	-8.30	0.0	False
2018-01-10	0.0	0.0	-inf	5.00	-7.8	-7.80	0.0	False
2018-01-11	0.0	0.0	-inf	4.40	-7.8	1.10	0.0	False

# Mitigating the issues

- Let's combine this with the apply() method to interpolate all of our columns at once:

```
>>> df_deduped.reindex(  
...     pd.date_range('2018-01-01', '2018-12-31', freq='D')  
... ).apply(lambda x: x.interpolate()).head(10)
```

# Mitigating the issues

- Check out January 9th, which we didn't have previously—the values for TMAX, TMIN, and TOBS are the average of the values for the day prior (January 8th) and the day after (January 10th):

	PRCP	SNOW	SNWD	TMAX	TMIN	TOBS	WESF	inclement_weather
2018-01-01	0.0	0.0	-inf	NaN	NaN	NaN	0.0	NaN
2018-01-02	0.0	0.0	-inf	-8.3	-16.10	-12.20	0.0	False
2018-01-03	0.0	0.0	-inf	-4.4	-13.90	-13.30	0.0	False
2018-01-04	20.6	229.0	inf	-4.4	-13.90	-13.60	19.3	True
2018-01-05	14.2	127.0	inf	-4.4	-13.90	-13.90	0.0	True
2018-01-06	0.0	0.0	-inf	-10.0	-15.60	-15.00	0.0	False
2018-01-07	0.0	0.0	-inf	-11.7	-17.20	-16.10	0.0	False
2018-01-08	0.0	0.0	-inf	-7.8	-16.70	-8.30	0.0	False
2018-01-09	0.0	0.0	-inf	-1.4	-12.25	-8.05	0.0	NaN
2018-01-10	0.0	0.0	-inf	5.0	-7.80	-7.80	0.0	False

# Summary

- In this lesson, we learned more about what data wrangling is (aside from a data science buzzword) and got some firsthand experience with cleaning and reshaping our data.
- Utilizing the requests library, we once again practiced working with APIs to extract data of interest; then, we used pandas to begin our introduction to data wrangling, which we will continue in the next lesson.
- Finally, we learned how to deal with duplicate, missing, and invalid data points in various ways and discussed the ramifications of those decisions.

# "Complete Exercises"

# "Complete Lab 8"