The Center of Applied Data Science

python™

Statistical Data Analysis

Day 1.1

Content Outline

In [ ]:

```python
import numpy as np
import pandas as pd
import math

import seaborn as sns
import matplotlib.pyplot as plt

from statistics import mode
from scipy.special import binom
from scipy.stats import iqr
```

In [ ]:

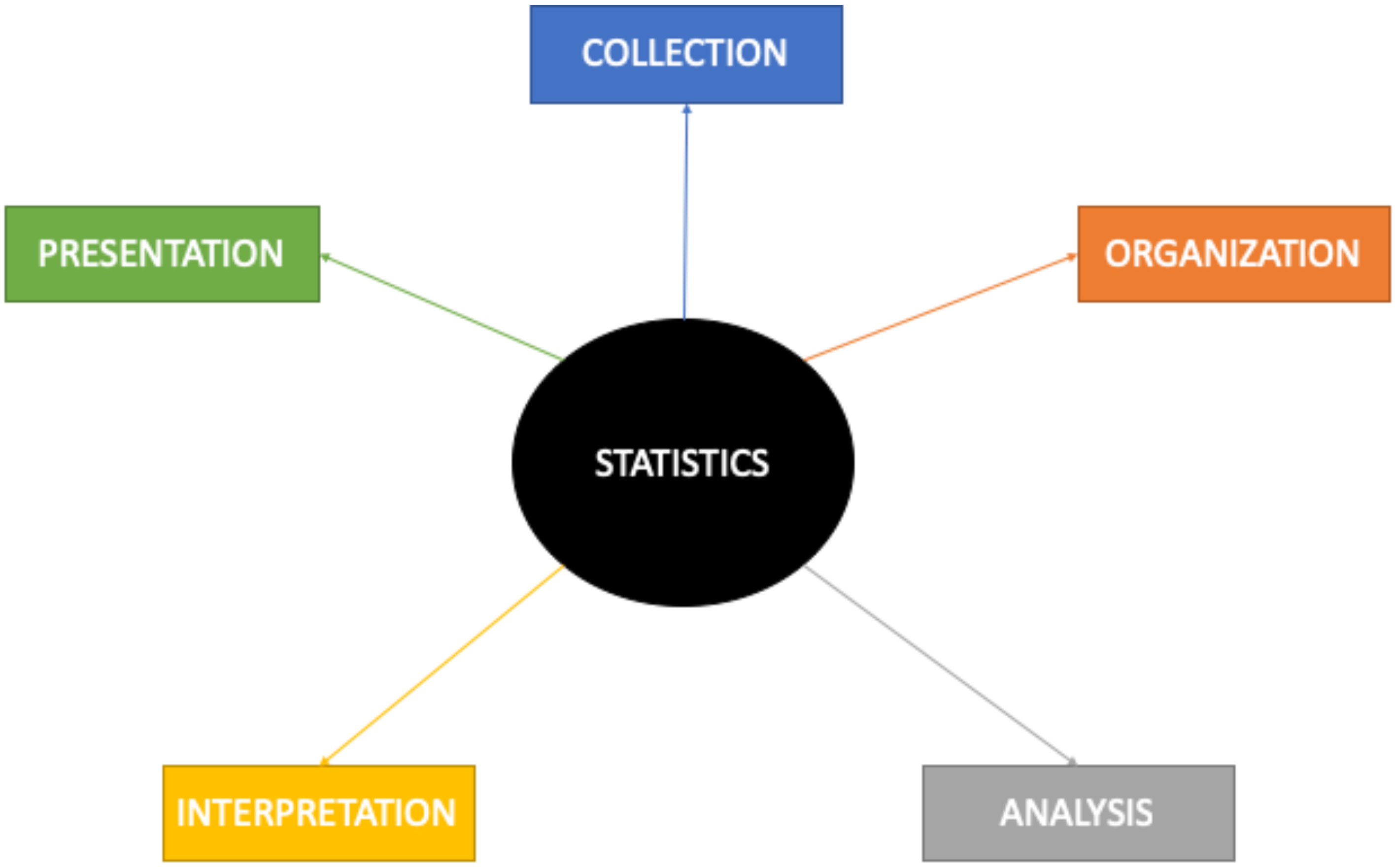
```python
sns.__version__ >= '0.9.0'
np.__version__ >= '1.15.4'
pd.__version__ >= '0.23.4'
```

In [ ]:

```python
sns.set()
```

1. Introduction

Statistics is the science of **collecting**, **organizing**, **analyzing**, **interpreting**, and **presenting** data. It helps us convert measurements we gather from the world around us into knowledge in a **systematic** manner to ensure both **consistency** and **reproducibility** of results.



Our journey begins with the simplest of questions - *"What would we like to know about a certain group of objects?"* The objects here can refer to people, animals, buildings, household items, etc. - basically anything that is of interest to us. The grouping of said objects is referred to as a **population**, and our aim is to generalize any results we derive from our investigations to this entire group. Populations are usually defined to encompass the largest possible collection of objects that are of interest in a given study.

Example

1. A researcher wishes to study the happiness levels among students currently undergoing tertiary education. The population defined here would be the set of all university students in Malaysia.
2. The Kuala Lumpur City Council (DBKL) is interested in identifying the average household expenditure of families in the Bottom 40% (B40) category under its purview. The population defined here would be the set of all families with household income RM3,000 and below.

In most cases, the population is so large that collecting data from the entire group becomes **infeasible** (sometimes impossible). For example, suppose that we are interested in the average price of all the 2-bedroom apartments in KL - this population is so large that we cannot study all of it without expending significant effort. In situations such as these, we **collect** data only from a **subgroup** of the population, which we call a **sample**. Ideally, a sample should represent the population well. For instance, a sample of 2-bedroom apartment prices in KL should include apartments from various municipalities to ensure no one part of the city is **under/over-represented**. Further details on populations and samples will be discussed later in this course.

Having gathered some data on a problem of interest, our data can then be **organized** easily with the use of spreadsheets or databases. In this course, we will operate on the assumption that the data has been cleaned and organized on flat files (CSV, XLS, etc.) as we wish to *focus on the analysis* of our data and not the logistics behind how we store it.

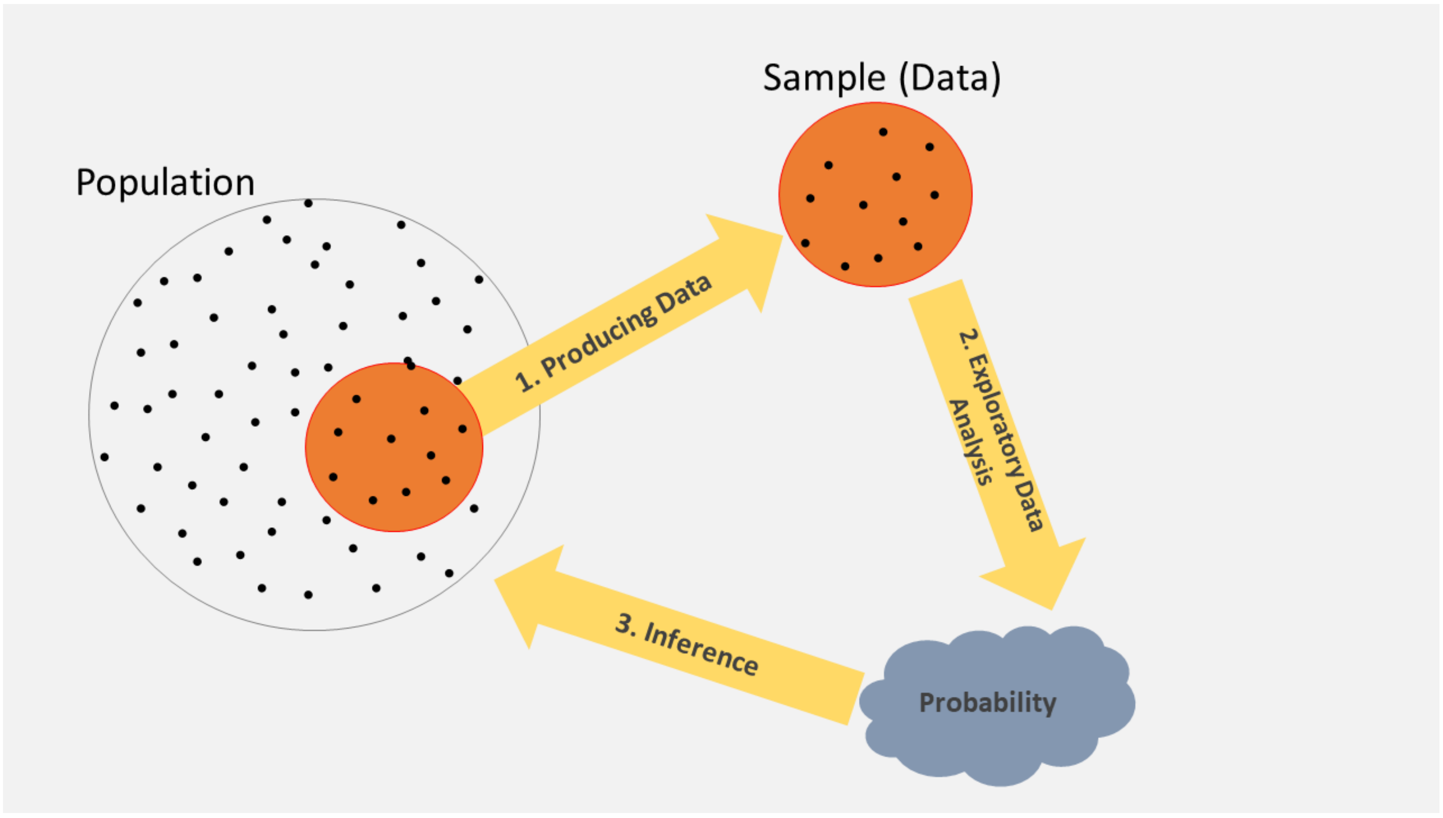
The first step of **analyzing** data is to **explore** it - this involves examining the type of data we've collected, how it behaves, and summarize our findings. This process is called **exploratory data analysis (EDA)**, and is one of the most crucial steps to understanding our data.

Using said findings, we can employ probability and estimation to make generalizations about the population based on our sample. This process is called **inference**, and it represents the culmination of the analysis stage. Inference allows us to use the data collected via a sample to model the behavior of the entire population, e.g. using customer product reviews to determine what product features are in demand.

Last but not least, we **interpret** and **present** our data with appropriate visual aids.

Interpretation often calls for domain-specific knowledge, which is subject to the context of our study. As always, it is best to let the data do the talking and that's precisely where **visualization** comes in.

The diagram below gives an overview of what is commonly known as **the big picture** of statistics. It represents the 3 core processes, which are producing/procuring data, exploratory analysis, and inference. Used well, statistics forms a powerful tool which can reap benefits for business decision making.



A. Course Scope and Layout

1. **Day 1**: We will explore the idea of using descriptive statistics to perform **exploratory data analysis**. Utilizing numerical measures and visual tools, we will examine sample data sets to extract valuable insight.
2. **Day 2**: We will discuss the various shapes that the **distribution** of our data can take and its implications on our analyses. We will also discuss the process of **sampling**, i.e. selecting samples from a population.

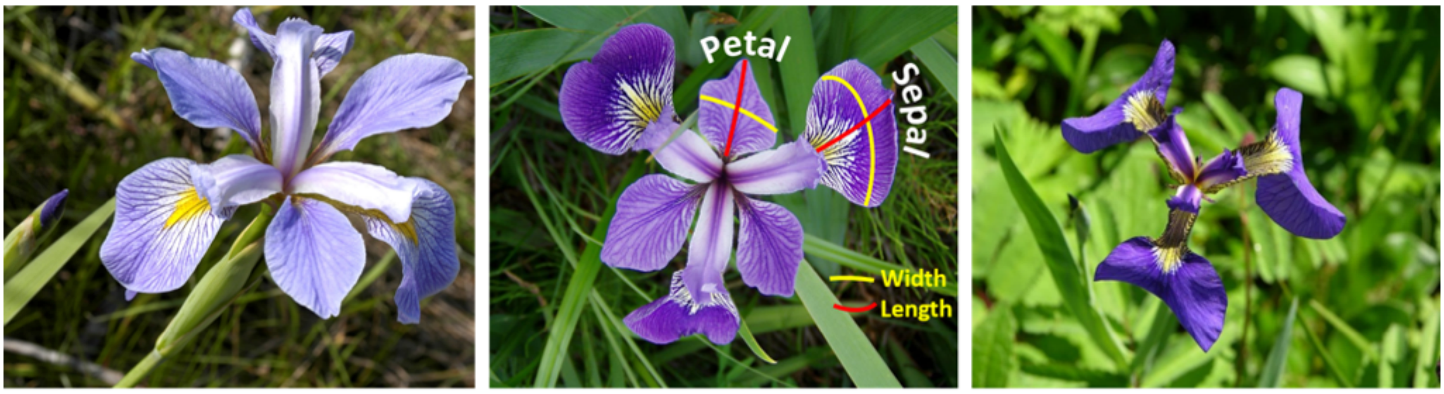
2. Data Exploration: Examining Distributions

We will begin by describing some typical datasets a data practitioner (such as yourself) will encounter.

Let us look at the *iris* open data set (read more (https://archive.ics.uci.edu/ml/datasets/iris)). This data set contains information about the length of various features of irises and their corresponding species.

In [ ]:

```
iris = pd.read_csv("../data/iris.csv")
iris.head()
```



Iris Virginica Iris Versicolor Iris Setosa

As you can see, the data is a table consisting of rows and columns. Each column represents a characteristic (or 'feature') of a flower, whereas each row corresponds to an individual flower (or 'observation').

Each column has restrictions on what values the fields can hold. The values of these fields can either be quantitative (as in `Sepal.Width`) or categorical (as in `Species`).

Categorical data can be either **nominal** or **ordinal**. There is no implicit ordering in the categories of nominal data. We can think of gender as a nominal value - it would not make sense to 'rank' women above men, or vice versa. In the *iris* dataset, the column `Species` has nominal values. For ordinal data, there is an implicit ordering defined for the categories. For instance, the size of a cup measured as "small", "medium", and "big" is ordinal.

Quantitative data on the other hand can be either **interval** or **ratio**. Interval data refers to data with an arbirtrary 0 such as time in A.D. and temperature in Celcius. On the contrary, ratio data refers to data with an absolute 0, such as physical measurements like length and weight.

In Python, the `pd.DataFrame.info()` method returns the type of each column (variable).

In [ ]:

```
iris.info()
```

In [ ]:

```
iris.Species=iris.Species.astype('category')
```

In [ ]:

```
iris.info()
```

In [ ]:

```
iris.describe()
```

In [ ]:

```
iris.describe(include='category')
```

In [ ]:

```
iris.describe(include=['float','category'])
```

Let's explore this data. Each variable or column in the data is a long list, which is not very informative. To convert this raw data into meaningful information, we need to summarize and examine its **distribution**. The distribution of a variable shows *what values the variable takes* and *how often the variable takes those values*.

A. One Categorical Variable

For data that consists of one categorical variable, we are often interested in demographic information. For example, we may ask:

1. *How many different species of irises do we have in this sample?*
2. *What is the percentage breakdown of species in the sampled flowers?*

The unique categories of species are:

In [ ]:

```
print("The unique categories of species are:")
iris.Species.unique()
```

Below is a frequency table that shows us how many observations are available of each species in our data set:

In [ ]:

```
print("The frequency table of species are:\n")
iris.Species.value_counts()
```

We can further modify the frequency table to give us the breakdown of species in percentages with the `normalize` argument:

In [ ]:

```
print("The percentages for each category of species are:\n")
iris.Species.value_counts(normalize = True) * 100
```

In [ ]:

```
iris.Species.value_counts().plot.pie(autopct='%1.1f%%',title
plt.axis('equal')
plt.show()
```

In [ ]:

```
iris.Species.value_counts().plot.bar(title = "Bar Chart of Sp
```

Exercise

A survey asked 1,200 college students the following question: "With whom do you find it easiest to make friends?" (opposite sex, same sex or no difference).

1. Print the frequency table and table of percentages.
2. Draw a pie chart and bar chart for the data.

In [ ]:

```
friends = pd.read_csv("../data/friends.csv")
friends.head()
```

In [ ]:

```

```

In [ ]:

```

```

In [ ]:

```

```

In [ ]:

```

```

In [ ]:

```

```

B. One Numerical Variable

For data that consists of one numerical variable, we can use numerical measures to represent the data. There are two main types of numerical measures - **measures of central tendency**, and **measures of dispersion/variability.**

Measures of Central Tendency

Measures of central tendency are used to describe the 'body' of a distribution by providing a reference point to its center. There are 3 measures of central tendency, namely the mean, median, and mode.

Let's begin by generating a random list of 20 integers from a uniform distribution with integers ranging from 0 to 8.

In [ ]:

```
np.random.seed(42)
```

In [ ]:

```
array_1 = np.random.choice(np.arange(0, 9), size = 20)
array_1
```

Mean

The **mean** for a set of numerical data is given by the formula:

$$\bar{x} = \frac{\sum x}{n}$$

$$\bar{x} = \frac{\sum x}{n}$$

where $\bar{x}\bar{x}$ is the mean, nn is the number of elements/data points, and $\Sigma\Sigma$ denotes summation. It represents the center of a data set in the sense that the weight of the numbers to the left and right of the mean are equal.

`pandas` objects have a `mean` method:

In [ ]:

```
built_mean_pd = pd.Series(array_1).mean()
f"The mean of array_1 calculated using the pandas method is {
```

`numpy` provides a function `mean()` that calculates the mean.

In [ ]:

```
built_mean_np = np.mean(array_1)
f"The mean of array_1 calculated using the numpy function is
```

Let's compute the mean of `array_1` using the formula and show that the built-in function yields the same output.

In [ ]:

```python
manual_mean = np.sum(array_1)/len(array_1)
manual_mean
f"The mean of array_1 by manual computation is {manual_mean}"
```

In [ ]:

```python
# Finally, we'll check if these values align
f"Is built_mean = manual_mean? {built_mean_np == manual_mean}
```

Pros:

1. Low computational cost.
2. Gives a good representation of the 'center' as the sum of data points to the left and right of the mean are balanced.

Cons:

1. Easily influenced by outliers, i.e. values that are abnormally large/small compared to the rest of the set.

Let's examine the impact of an outlier on our set `array_1` . We'll simulate this by adding a large number to the set as below:

In [ ]:

```python
array_2 = np.append(array_1, 100)
array_2
```

Now let's compute the mean of the new set:

In [ ]:

```python
np.mean(array_2)
```

As we can see, there is a large swing in the mean (from 5.1 to ~9.62) due to the large value (100) that was added to the set. Here we say the mean is **sensitive** to outliers. As such, we should be careful about when we use the mean to describe a data set.

Example

1. The manager of a call center computes the mean number of calls attended to by an employee in an hour as a reference value to benchmark employee performance. We can safely use the mean in this scenario as we do not expect a sudden jump in the number of calls attended from employee to employee.

2. An analyst computes the mean salary of managers in a Fortune 500 company. Here the decision to use the mean is unwise as we expect that there may be a large gap in pay between senior and junior managers, which in turn may impact the value of the mean.

Median

The median of a numerical data set is the **center-most** number when your data is sorted in **ascending** order, i.e. from the smallest to largest. It can be computed in two ways:

1. If the data set has an **odd** number of observations, the median is the element in the **middle** that equally bisects the set such that the number of elements to the left and right are identical.
2. If the data set has an **even** number of observations, the median is the **mean** of the **middle** two elements when sorted in ascending order.

Unlike the mean, the median is **immune** to outliers. This behavior does come at a cost though - the median gives the physical center of a data set, i.e. it takes into account the location of a number in ascending order as compared to the rest of the data set, but ignores the weight of each value.

Take note that compared to the mean, the median is more computationally expensive as it requires us to sort the data set in ascending order first. As in the case for mean, we will show that the result from the built-in function is identical to that if we compute the median manually.

Let us begin by computing the median. As before, `pandas` objects have a method for computing the median:

In [ ]:

```
built_median_pd = pd.Series(array_1).median()
f"The median of array_1 calculated using the pandas method is
```

Using `numpy`, we can compute the median using the function `median()`.

In [ ]:

```
# We can find the median of this same list using the median (
built_median_np = np.median(array_1)
f"The median of array_1 calculated using the numpy function i
```

To compute the median manually, we must first determine the length of our data set (in this case `array_1`) so we can pick the appropriate method:

In [ ]:

```
# How long is array_1? We need to know if it is odd or even t
f"The length of array_1 is {len(array_1)}"
```

As the list contains an even number of elements (20), we should take the mean of the two middle (10th & 11th, or index 9 and 10) elements once sorted in ascending order.

In [ ]:

```
array_1 = np.sort(array_1)
manual_median = (array_1[9] + array_1[10]) / 2
f"The median of array_1 by manual computation is {manual_medi
```

In [ ]:

```
manual_eq_built = manual_median == built_median_pd
f"Is built_median = manual_median? {manual_eq_built}"
```

Now suppose that the data set has an *odd* number of elements, e.g. `array_2`. Once again, we can compute the median manually by selecting the **center-most** element in the set when sorted in ascending order:

In [ ]:

```
array_2 = np.sort(array_2)
manual_median_2 = array_2[math.floor(len(array_2)/2)]    # Thi

f"The median of array_2 by manual computation is {manual_medi
```

In [ ]:

```
f"Is built_median_2 = manual_median_2? {np.median(array_2) ==
```

Pros:

1. Immune to outliers. Abnormally small/large values will not affect the median as it gives the physical 'center' of the data set.

Cons:

1. Does not take into account the weights of each element in the data set.
2. Computationally expensive compared to the mean. This becomes apparent when dealing with large data sets.

To illustrate the increase in computational complexity associated with calculating the median, vary the value of `N` in the following chuck of code by factors of `10` to see the increase in time taken to perform the calculations:

In [ ]:

```
N = 1000000

large_array = np.random.choice(np.arange(1000), size = N)

%time np.median(large_array)
%time np.mean(large_array)
```

As we can see, the time taken to compute the median goes up significantly in comparison to the mean as the data set grows larger. This behavior is worth noting if you are working on large data sets and plan to integrate the median in your analyses. We would however like to point out that although beyond the scope of this course, there is a formal method for assessing the computational complexity of an algorithm (read more (https://en.wikipedia.org/wiki/Computational_complexity)).

Due to its immunity against outliers, the median is the measure of choice when dealing with data sets that contain large gaps in the value of its elements. For example, consider `array_2` which is graphed below:

In [ ]:

```
pd.Series(array_2).plot.hist(bins = 100)
```

In [ ]:

```
pd.Series(array_1).plot.hist(bins = 100)
```

In [ ]:

```
print('Median of array_2 is', np.median(array_2))

print('Median of array_1 is', np.median(array_1))
```

The graph above is called a histogram, and is useful for examining numerical distributions (we'll learn how to construct these later in the course). As we can see from the histogram, the values in the data set go from 1 to 100, but the median ignores the distance between the numbers and focuses on the **physical location** instead. In other words, the **gaps** between values in our data set **do not affect** the median.

Example

Recall the example from the previous section where the analyst wanted to examine the distribution of salary of managers in a Fortune 500 company. Instead of using the mean, he would obtain a better representation of the 'center' of the salary data by using the median.

Mode

The mode of a numerical data set is the value that occurs **most often**. We can do this by selecting the element of the data set with highest **frequency**, i.e. rate of occurrence.

Pandas objects have a `mode` method to compute the mode. Just as in the previous sections, we will illustrate the computation using both the pre-built package and by manual computation.

In [ ]:

```
print(array_1)
built_mode_pd = pd.Series(array_1).mode()
built_mode_pd
```

In [ ]:

```
array3 = pd.Series([1, 2, 3, 4, 5])
pd.Series(array3).mode()
```

We can also use the `mode` function from `statistics` :

In [ ]:

```
# only returns the smallest number if more than one mode
built_mode_stat = mode(array_1)
built_mode_stat
```

Alternately, use the `most_common()` function of the `Counter` class:

In [ ]:

```
from collections import Counter
Counter(array_1).most_common(3)
```

Pros:

1. The mode is easy to interpret as it represents the element with the highest frequency. In a business context this can be the best selling product, staff member with the most positive reviews, etc.

2. Can also be used for categorical data - the mode is the category with the highest frequency.

Cons:

1. Does not take into account the weights nor physical location of each element in the data set.

If a data set has only **one** value for the mode, it is called **unimodality** or a **unimodal distribution**.

It is worth noting that a data set may contain **more than one** mode. As such, we **cannot assume** unimodality for a data set - this is reflected in the list comprehension used to compute `values`. Take for example the following **bimodal** distribution, which contains 2 modal elements:

In [ ]:

```
bimodal_dist = [binom(8,x) + binom(8, x-4) for x in np.arange
```

In [ ]:

```
pd.Series(bimodal_dist).plot.bar(colormap = "viridis", title
```

We can also have **trimodal** (3 modal elements) or **multi-modal** (4 or more modal elements) sets. Given below is `array_3`, an example of a trimodal set:

In [ ]:

```
# mode() returns error in case of more than one mode

array_3 = [1,1,1,1,1,2,2,2,2,2,3,3,3,3,3,4,4,5,5,6,7,8,8,9,9,

array_3_mode, _ = mode(array_3)

f"The mode of list_3 is {array_3_mode}"
```

In [ ]:

```
#Solution1 for multimodal array:
Counter(array_3).most_common()
```

In [ ]:

```
#Solution2 for multimodal array:

def mode_man(my_list,):
    x = list(my_list)

    # obtain unique values and their counts as a list of tupl
    value_counts = [(i, x.count(i)) for i in list(set(x))]

    # calculate the maximum count
    max_count = np.max([elem[1] for elem in value_counts])

    # return the values of each unique value that appears as
    values = [i[0] for i in value_counts if i[1] == max_count
    return values, max_count
```

In [ ]:

```
values, max_count = mode_man(array_3)
```

In [ ]:

```
values, max_count
```

Example

1. The manager of a convenience store computes the mode of products he sells to identify his best-selling products.

2. The Road Transport Department tallies the number of fatal car accidents across various regions in Malaysia and computes the mode to identify which regions are the most dangerous.

Choosing the Best Measure of Central Tendency

Choosing the best measure of central tendency is subject to the context of the business problem and the type of data in hand, but the following considerations can serve as a general guideline:

1. If there are no outliers in the data, the mean gives the best approximation as it takes into account *both* physical location and weights.
2. If outliers are present in the data, the median serves as a good approximation as it is robust to outliers.
3. If the data is categorical or the business problem involves identifying the data point with highest occurrence, the mode should be used.

In the following example, we show how removing an outlier impacts the the measures of central tendency we have discussed.

In [ ]:

```
f"list_3: {array_3}"
```

In [ ]:

```
print(f"""
median of array_3: {np.median(array_3)}
mean of array_3: {np.mean(array_3)}
mode of array_3: {mode_man(array_3)[0]}
      """)
```

The set `array_3` contains an abnormally large value (24) in comparison to the rest of the values in the set. We classify this as an outlier and intend to remove it from the set.

In [ ]:

```
array_3e = array_3[:-1]
array_3e

print(f"""
median of array_3e: {np.median(array_3e)}
mean of array_3e: {np.mean(array_3e)}
mode of array_3e: {mode_man(array_3e)[0]}
""")
```

Removing the outlier, we see that the mean of the set has moved closer to the median, whereas the median has remain unchanged. This indicates that the median is a better approximation for the true center of the data set. The mode remains unchanged as the most frequently occurring numbers stay the same for both sets.

Guided Exercise

In each of these scenarios, how would you interpret the following measures:

1. The mean of rainfall in KL in 2017 was 250 mm.
2. The median population age in Malaysia is 27.7 years.
3. The modal income of individuals in Malaysia was $10,620 (USD) in 2017.

```
1. If the amount of rainfall in KL was evenly spread across all the days in the year, it would amount to 250mm per day.

2. Half the population in Malaysia is below 27.7 years of age, and the other half is above 27.7 years of age.

3. Of the individuals sureveyed in Malaysia for the year 2017, the most common income was $10,620.
```

Exercise

Using the `iris` dataset, calculate the measures of central tendency for each variable (column) and interpret the results.

In [ ]:

```

```

In [ ]:

```

```

In [ ]:

```

```

In [ ]:

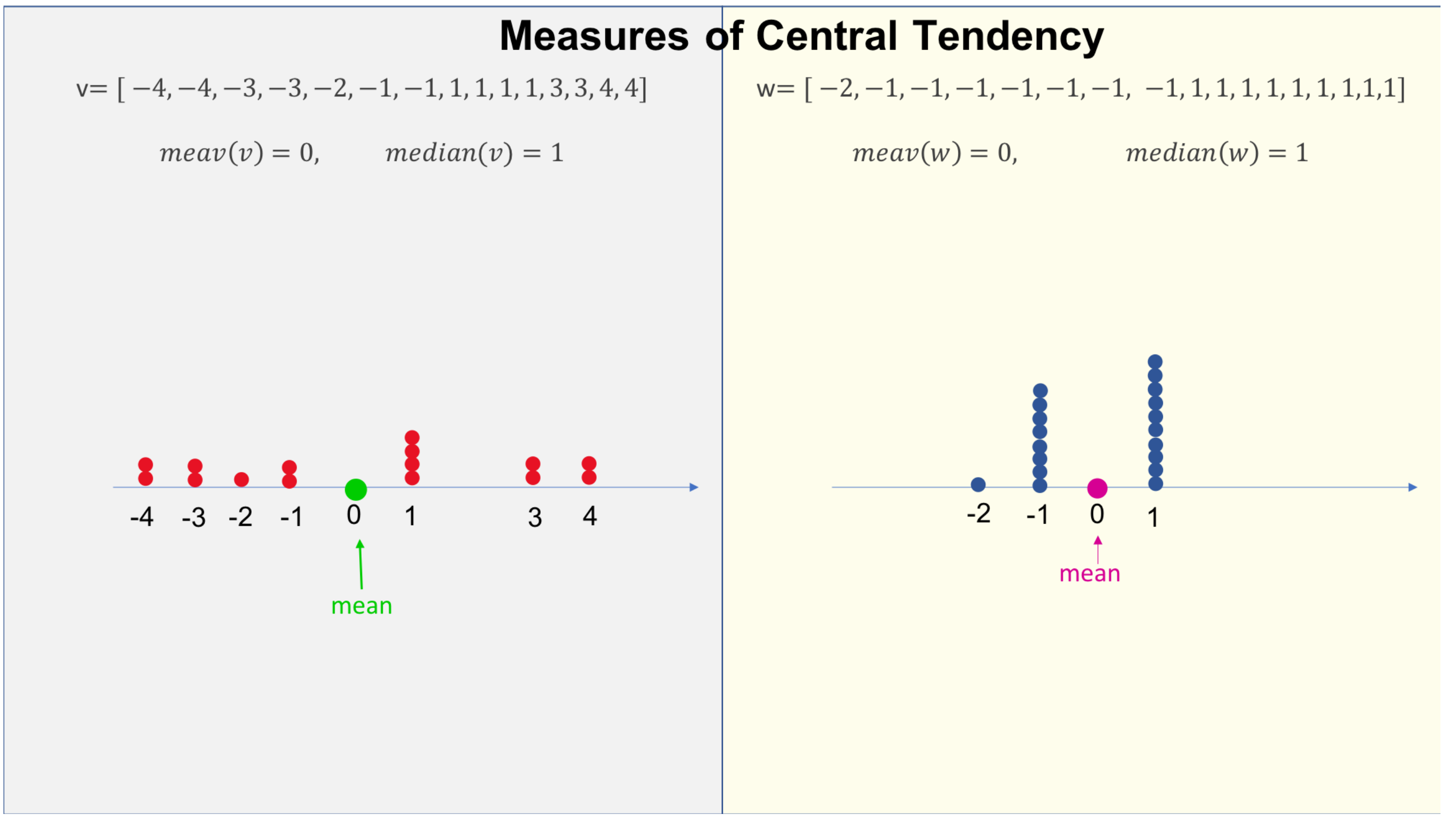
```

```

Measures of Dispersion/Variability

Though immensely useful, the center of a data set does not paint a clear enough picture to properly visualize the behavior of a data set. Consider the following comparison:

Example:

Do the following variables v v and w w have similar patterns?



As we see in the figure above, two sets of data can share the same measures of central tendency (mean & median in this case) yet still behave very differently. As such, we are also interested in examining how **spread out** the data is from the center. Numerical measures that measure this spread are known as **measures of dispersion** or **measures of variability**.

In this section, we will discuss the following 3 measures of dispersion:

1. Range
2. Percentiles, Interquartile Range (IQR), Outliers,
3. Variance & Standard Deviation

Range

Range is the simplest measure of dispersion, and is defined as the difference between the smallest and largest values in a set of data:

$$Range = x_{largest} - x_{smallest}$$

$$Range = x_{largest} - x_{smallest}$$

Suppose we have the following data on the number of siblings of individuals from 25 families:

In [ ]:

```
# Lets pretend we have collected data on the number of siblin
# everyone in the classroom has. This is saved as no_siblings

no_siblings = [1, 0, 2, 0, 3, 1, 2, 4, 2,
               2, 0, 1, 2, 5, 2, 1, 2, 5, 7, 2,
               3, 0, 0, 1, 3]
```

As per the formula, we can compute the range of this data set by taking the difference between the largest and smallest values in the data set:

In [ ]:

```
np.max(no_siblings) - np.min(no_siblings) # maximum - minimun
```

Range is often the first measure of dispersion calculated because it relies on only the smallest and largest elements in the set, and gives a quick overall description of how spread out the data is. However, due to its dependence on these two values (minimum and maximum), the range can be impacted by the presence of an outlier. Consider the following case where we add an outlier to the set `no_siblings` :

In [ ]:

```
no_siblings.append(45)
```

In [ ]:

```
f"The range of no_siblings is {max(no_siblings) - min(no_sibl
```

As we can see, the range here has spiked due to the presence of the outlier. As such, any information gathered based on the range of a data set should be cross-checked with the presence of outliers in the set - if an outlier is to be excluded, recompute the range as necessary.

Pros:

1. Easy to calculate, only requires knowledge of the smallest (minimum) and largest (maximum) values in a data set.
2. Gives a good overview of how spread out the data is.

Cons:

1. Can be influenced by outliers.

Example

1. A project manager computes the range on the data set of previous project milestones to plan the working schedule for future iterations.
2. A HR manager computes the range of time spent by employees to complete a given task to determine if there is a need for training/upskilling.

Quantiles

Quantiles are cutoff points across the range of data that are evenly spaced. They are usually computed based on the data being sorted in ascending order and being split into equal portions. Two commonly used quantiles are **percentiles** and **quartiles**.

A percentile is a value that is **greater than** a given **percentage** of observations. In other words, the data set is sorted in **ascending** order and split into 100 **equally-spaced** parts, with each percentile representing a cutoff point.

Quantiles can be computed using the pandas object method `quantile` and the `numpy` functions `percentile` and `quantile`. Using the `no_siblings` set from the last example, let's calculate the 27th percentile.

In [ ]:

```
np.quantile(no_siblings, .27)
```

In [ ]:

```
np.percentile(no_siblings, 27)
```

Here we see that the value in the data set corresponding to the 27th percentile is 1. In other words, 27% of the data in this set has value 1 or lower.

The `pandas` method `quantile` and the `np.quantile` function also supports computing multiple quantiles at once by passing through a vector argument. Suppose we are interested in finding the 30th, 55th, and 80th percentiles of our data:

In [ ]:

```
pd.Series(no_siblings).quantile((.30, .55, .8))
```

In [ ]:

```
np.quantile(no_siblings,[.30, .55, .8])
```

In [ ]:

```
np.percentile(no_siblings, [30, 55, 80])
```

Based on the resulting output, we can conclude that:

1. Less than 30% of the individuals in the data set have less than 1 (no) sibling.
2. Less than 55% of the individuals in the data set have less than 2 siblings.
3. Less than 80% of the individuals in the data set have less than 3 siblings.

Quartiles on the other hand are obtained when the data is split evenly across **4** portions across its entire range. The 4 quartiles are as follows:

1. 1st Quartile, Q_1 Q_1 - bottom 25% of oberservations, i.e. the *25th percentile*.
2. 2nd Quartile, Q_2 Q_2 - bottom 50% of oberservations, i.e. the *50th percentile*. The 2nd quartile also corresponds to the *median*.
3. 3rd Quartile, Q_3 Q_3 - bottom 75% of oberservations, i.e. the *75th percentile*.
4. 4th Quartile, Q_4 Q_4 - all oberservations, i.e. the *100th percentile*.

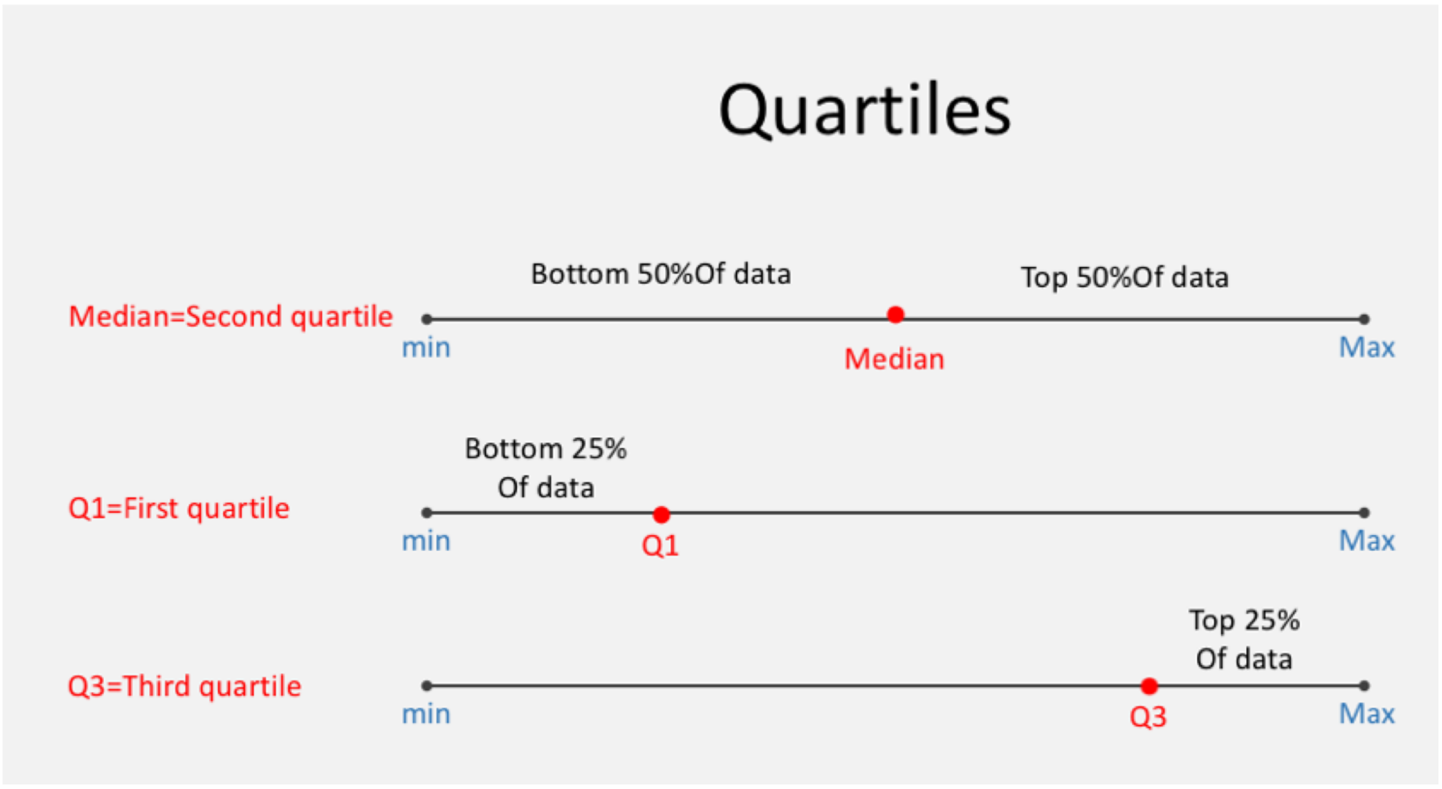
Pros:

1. Allows the data set to be segmented for more detailed analysis on the distribution of data.

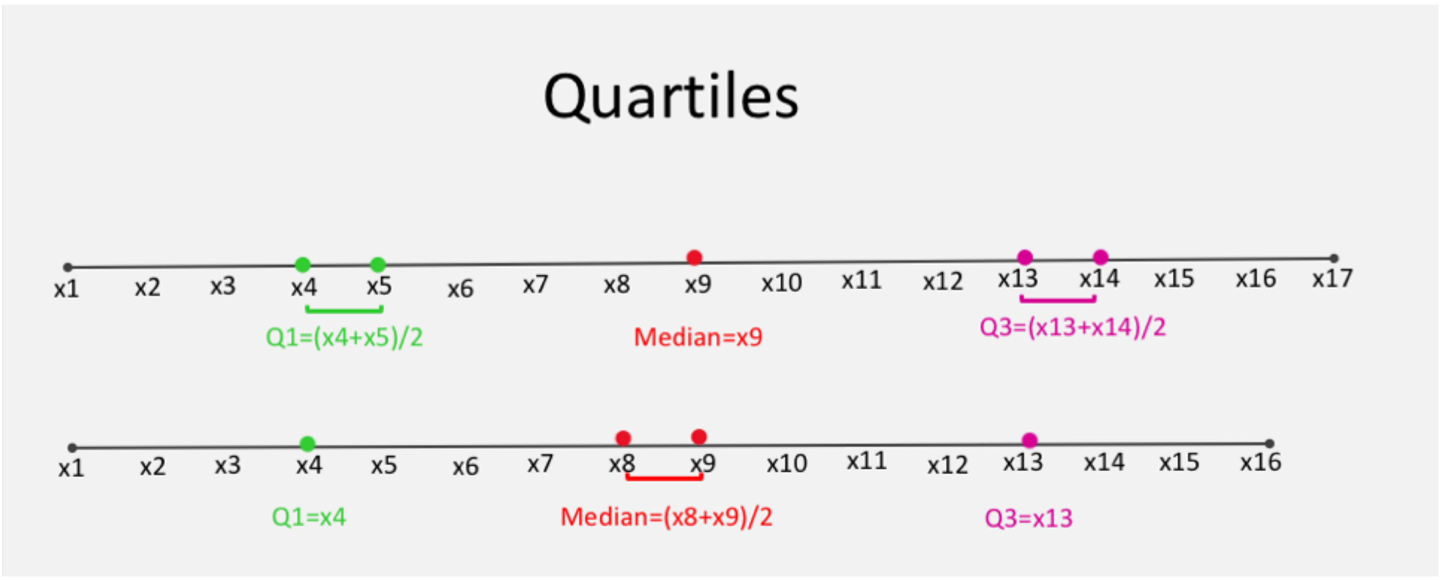
Cons:

1. Quantiles can sometimes be difficult to interpret if the resulting value is non-integer.

The figure below summarizes the relationship between the percentiles and quartiles:



Computation of the quartiles can also be done manually by taking the median twice in succession - this process is illustrated in the figure below:


Quartiles
x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12 x13 x14 x15 x16 x17
Q1=(x4+x5)/2
Median=x9
Q3=(x13+x14)/2
x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12 x13 x14 x15 x16
Q1=x4
Median=(x8+x9)/2
Q3=x13

Just like some of the previous functions, we can compute the quantiles manually by multiplying the number of elements in the set by the desired quantile and rounding up to the nearest index number. Selecting the element of that index from the ordered data set will yield the desired quantile. To illustrate this, we compute the 30th percentile manually as follows:

In [ ]:

```python
siblings_sorted = sorted(no_siblings)

# check if this code is OK!
p30 = np.floor(len(siblings_sorted) * .3)
siblings_sorted[np.int64(p30)]
```

Example

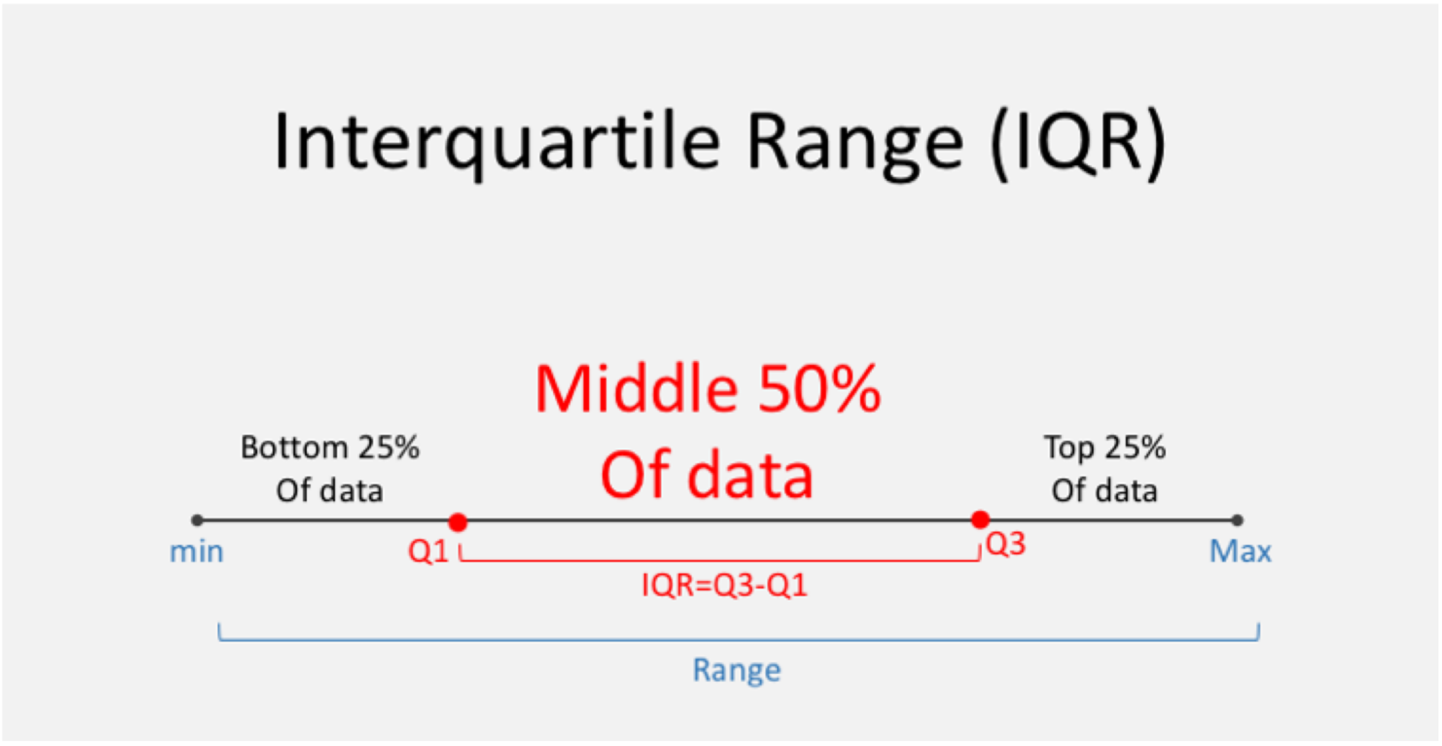
1. An Ivy League university selects only the students who placed in the top 5th percentile of SAT scores for admission.

2. A regional manager identifies low-performing stores by selecting all stores whose revenue for 2017 falls within the 1st quartile in the company's annual sales data set.

Interquartile Range

The interquartile range (IQR for short) is a measure of spread for the **middle 50%** of data. As extreme values occur at the low/high end of the spectrum, the interquartile range gives a good approximation for a distribution as it quantifies the dispersion for only the central portion of the data, making this measure robust to outliers.

The IQR can be computed by taking the difference between the 1st and 3rd quartiles, as summarized in the figure below:



Computation of this measure can be done using the `scipy` function `scipy.stats.iqr`. The code segments below illustrate the use of both manual computation and the `scipy` function, comparing the results obtained.

In [ ]:

```
q1, q3 = np.quantile(no_siblings,[.25, .75])
manual_iqr = q3 - q1
f"The IQR of no_siblings by manual compuation is {manual_iqr}
```

In [ ]:

```
scipy_iqr = iqr(no_siblings)
f"The IQR of no_siblings using the scipy function is {scipy_i
```

In [ ]:

```python
f"Is manual_iqr = scipy_iqr? {manual_iqr == scipy_iqr}"
```

Determination of Outliers using Tukey's Method

As mentioned in previous sections, an outlier is a value which is extremely small/large in comparison to the other values in a data set. Though removing outliers is context-sensitive, there exist systematic processes to identify outliers using measures of dispersion. In this course, we choose to highlight one commonly used technique called **Tukey's Method**.

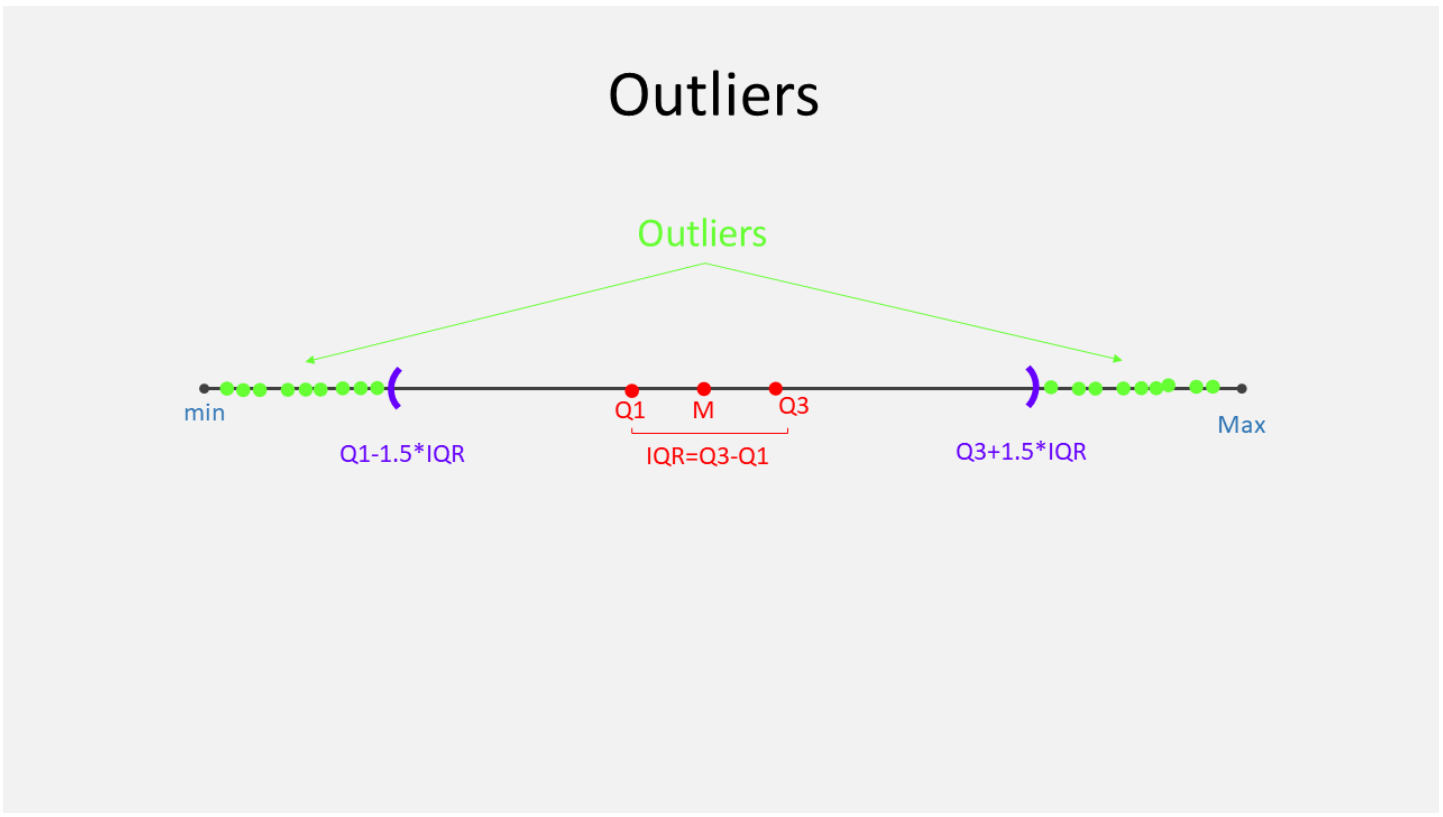
Tukey's Method defines outliers as values that fall outside the 1st and 3rd quartiles by 1.5 times the IQR. These boundaries are known as lower and upper **fences** and represent the outer limits of reasonable value to be included in the data set. The fences can be computed using the following formulae:

$$LF = Q1 - 1.5 \times IQR$$

$$LF = Q1 - 1.5 \times IQR$$

$$UF = Q3 + 1.5 \times IQR$$

$$UF = Q3 + 1.5 \times IQR$$



To strengthen these concepts, let's practice computing the fences (and some other numerical measures).

Guided Exercise

Compute the following measures for the `no_siblings` data set:

1. Minimum
2. First quartile, Q_1 Q_1
3. Median
4. Third quartile, Q_3 Q_3
5. Maximum
6. Range
7. IQR
8. Lower Fence
9. Upper Fence

In [ ]:

```

```

Based on the fences computed, the value 7 should be classified as an **outlier**.

Note: When presented together, the Minimum, 1st Quartile, Median, 3rd Quartile, and Maximum are known as a **Five Figure Summary**.

Pros:

1. Provides a clear boundary on which values should/shouldn't be classified as outliers.

Cons:

1. Fails for small sample sizes.

The code segment below illustrates a case in which Tukey's method fails to identify the value 20,000 as an outlier due to an extremely small sample size:

In [ ]:

```python
small_sample = [1, 50, 20000]

print(f"""
The IQR is {iqr(small_sample)}
The lower fence is {np.quantile(small_sample, .25) - 1.5 * iq
The upper fence is {np.quantile(small_sample, .75) + 1.5 * iq
""")

```

Example

1. An automated early warning system can be programmed to flag abnormal behavior if sensors record values outside of the prescribed fences.

2. A schoolteacher can identify students who exhibit exceptional performance by flagging those whose scores are past the upper fence for the entire school.

Variance

The variance of a data set is a numerical representation of how much the data is spread out from the mean. For a **sample** of data, it is given by the formula:

$$\mathrm{Var}(x) = \frac{\sum(x - \bar{x})^2}{n - 1}$$

$$\mathrm{Var}(x) = \frac{\sum(x - \bar{x})^2}{n - 1}$$

where $\bar{x}\bar{x}$ is the mean and nn is the sample size, i.e. number of elements in the data set.

In [ ]:

```python
def variance(x):
    series_mean = np.mean(x)              # Calculate our mean
    deviation = x - series_mean           # Subtract mean from
    squared_dev = deviation ** 2          # Square these values
    sum_squared_dev = sum(squared_dev)    # Sum the squares
    var = sum_squared_dev / (len(x)-1)    # divide by number of
    return var
```

From the formula, we can see that the variance is simply the average squared distance from the mean of the data set. A larger variance indicates that the data is more spread out from the mean, whereas a smaller variance indicates that the data is concentrated around the mean value.

Let's take another look at the `iris` data set.

In [ ]:

```
# Take a look at the dataset
iris.head() # head() shows us the first few observations
```

Using the formula, we can manually compute the variance of `Sepal.Length` for all observations in our dataset.

In [ ]:

```
sepal_mean = np.mean(iris.loc[:, "Sepal.Length"])
subtracted_mean = iris.loc[:, "Sepal.Length"] - sepal_mean
square_diff = subtracted_mean ** 2
var_manual = np.sum(square_diff)/(len(square_diff) - 1)
var_manual
```

In [ ]:

```
var_manual = variance(iris.loc[:, "Sepal.Length"])
```

`pandas` Series has a method `var()` that computes the sample variance:

In [ ]:

```
pd_var = iris.loc[:, "Sepal.Length"].var()
pd_var
```

`numpy` also provides a function to compute the variance, namely `var`. By default, `np.var` calculates the variance with 0 Delta degrees of freedom. As we are concerned with the **sample** variance, we need to set the `ddof` argument equal to 1.

In [ ]:

```
np_var = np.var(iris.loc[:, "Sepal.Length"], ddof = 1)
np_var
```

Let's compare the values generated by our manual computations and the built-in function.

In [ ]:

```
# numpy and formula-based methods may not be equal because o
from math import isclose
```

In [ ]:

```
print(f"""
Is pd_var = built_var? {isclose(pd_var, var_manual)}
Is np_var = built_var? {isclose(np_var, var_manual)}
""")
```

Standard Deviation

The standard deviation is the most commonly used measure of dispersion due to its easy interpretability. It is defined as the square root of the variance:

$$s = \sqrt{\text{Var}(x)}$$

$$s = \sqrt{\text{Var}(x)}$$

The standard deviation is usually in the same units as the observations in a data set, and hence it tends to be used more often than the variance itself.

Let's compare the results of using the formula and the built-in function.

Again, `pandas` objects have a `std()` method to calculate the sample standard deviation:

In [ ]:

```
sd_manual = np.sqrt(var_manual)
sd_built = iris.loc[:, 'Sepal.Length'].std()

f"Is sd_manual = sd_built? {isclose(sd_manual, sd_built)}"
```

In `numpy` , the standard deviation can be computed using the `std` function. Note that the `numpy` function computes the **population** standard deviation by default, i.e. there is no degrees-of-freedom adjustment. To estimate the **sample** standard deviation, do the following:

In [ ]:

```
np.std(iris.loc[:, 'Sepal.Length'], ddof = 1)
```

Visual Tools

Boxplot

The Five Figure Summary is often visualized using a **box and whisker plot** (or **boxplot** for short). Boxplots allow us to view the distribution of the entire variable in terms of its quartiles and decide on cutoff points for outliers.

A boxplot consists of a body with 3 lines representing the 1st, 2nd, and 3rd quartiles of the data set. Whiskers protrude from the boxplot up to the smallest/largest values *within* the upper and lower fences, with outliers indicated as disconnected points. The length of the boxplot's body is that of the IQR, so comparing boxplots side-by-side allows us to compare distributions in terms of both their central tendency and variation.

Let's start with a simple example - a boxplot for the data set `no_siblings` :

In [ ]:

```
# Lets pretend we have collected data on the number of siblin
# everyone in the classroom has. This is saved as no_siblings

no_siblings = [1, 0, 2, 0, 3, 1, 2, 4, 2,
               2, 0, 1, 2, 5, 2, 1, 2, 5, 7, 2,
               3, 0, 0, 1, 3]
```

In [ ]:

```
# Lets plot a boxplot with our sibling data from before
plt.boxplot(no_siblings)
plt.show()
```

We can clearly see from our boxplot that the median is 2, the 1st and 3rd quartiles are 1 and 3 respectively, and there exists an outlier, 7, in our data set.

Now let's compare two distributions side-by-side using `array_1` and `array_3` :

In [ ]:

```
fg, ax = plt.subplots(ncols = 2)

ax[0].boxplot(array_1)
ax[0].set_title('array_1')
ax[1].boxplot(array_3)
ax[1].set_title('array_3')
plt.show()
```

Here we see that there is a stark difference between the distributions for `array_1` and `array_3` . They are both centered at different (median) values, and the spread is dissimilar. `array_3` also contains an outlier, something that is not present in `array_1` .

Histogram

A **histogram** is similar to a bar chart, but works by separating numeric values according to their count/frequency. In lieu of categories, we use equal-sized intervals called **bins** to group values together. `pandas` Series have a method for histograms, `hist()` :

In [ ]:

```
iris.loc[:, 'Petal.Length'].hist(bins = 30)
plt.title('Histogram for iris Petal.Length, bins = 30' )
plt.show()
```

In [ ]:

```
iris.loc[:, 'Petal.Length'].hist(bins = 100)
plt.title('Histogram for iris Petal.Length, bins = 100' )
plt.show()
```

Selection of an optimal bin width is subjective - a smaller width gives more detail but results in a messier output, whereas a larger width can result in cleaner output but loses visual detail. Try experiment with the bin widths in the above code to see how it affects the output.

Exercise

Using the `iris` dataset:

1. Calculate the Range, Quartiles, IQR, number of outliers, variance & standard deviation for `Petal.Length`.
2. Draw a boxplot for `Petal.Length`.
3. Draw a histogram for `Petal.Length`.

In [ ]:

```
pl = iris.loc[:, 'Petal.Length']
```

In [ ]:

```
pl.describe()
```

In [ ]:

```

```

In [ ]:

```

```

In [ ]:

```

```

3. Data Exploration: Examining Relationships

One of the core skills in exploratory data analysis is examining the relationships between variables in a data set. In many cases, we wish to establish a basis for cause and effect that can be further investigated. By examining the relationship between variables we can develop a clearer understanding of how one affects the other, allowing us to narrow down on interesting portions of our data.

In this section, we will look at pairs of variables and explore their relationship (or lack thereof) using numerical summaries and visual displays. For each pair, we designate one variable as explanatory (cause) and one as response (effect). In practice, the choice of these designations is subject to the context of the business problem in question.

A. Categorical Explanatory and Response Variables

If both the explanatory and response variables are categorical in nature, we can utilize **two-way tables**, **conditional percentage tables**, or **multibar charts** to compare the variables head-to-head.

Example

1. Are the smoking habits of a person (Yes/No) related to the person's gender?
2. Is there a relationship between the type of light a baby sleeps with (No Light/Night light/Lamp) and whether or not the child develops nearsightedness?

Let's use the `nightlight` data set to see if we can answer the 2nd question in our example above.

Two-way Tables (Cross-tabulation)

Cross-tabulation involves forming tables where one the responses of one variable is sorted by columns, and the other is sorted by rows. The observations in each cell then become the count for the categories in a specific row & column combination. These tables, also known as **two-way tables** can be formed using `pd.crosstab()` .

In [ ]:

```
nightlight = pd.read_csv('../data/nightlight.csv')
nightlight.head()
```

In [ ]:

```
nightlight_tab = pd.crosstab(nightlight.loc[:, "Light"],
                             nightlight.loc[:, "Nearsightedne
nightlight_tab
```

Conditional Percentage Tables

Conditional percentage tables are similar to two-way tables, with the counts converted into percentages. They are useful for highlighting disparities in proportions between categories as illustrated in the following examples. Using `pd.crosstab()` , they can be constructed by setting the `normalize` argument to `index` , `columns` , or `"all"` :

In [ ]:

```
pd.crosstab(nightlight.loc[:, "Light"],
            nightlight.loc[:, "Nearsightedness"],
            normalize = "all") * 100
```

To calculate row-wise percentages, normalize by `"index"` :

In [ ]:

```
pd.crosstab(nightlight.loc[:, "Light"],
            nightlight.loc[:, "Nearsightedness"],
            normalize = "index") * 100
```

and for column-wise percentages, normalize by `"column"` :

In [ ]:

```
pd.crosstab(nightlight.loc[:, "Light"],
            nightlight.loc[:, "Nearsightedness"],
            normalize = "columns") * 100
```

Multibar Chart

A **multibar chart** is a paired bar chart where one variable is used as the x-axis, and the other is overlaid as categories spanning the first variable. This can be done using the `plot.bar()` method on the resulting DataFrame from `pd.crosstab()` .

In [ ]:

```
nightlight_tab.plot.bar(title = "Nearsightedness by Light Typ
```

B. Quantitative explanatory and categorical response variables

In the case of categorical explanatory variables with quantitative response variables, we can use numerical summaries and boxplots as we previously in the section on numerical variables. Here however, the focus will be on relating the relationship between the categorical and quantitative variables, and not just examining a single distribution.

Example

1. Is there a relationship between gender (Male/Female) and test scores on a particular standardized test?

2. How is the number of calories in a hot dog related to (or affected by) the type of hot dog (Beef/Meat/Poultry)? In other words, are there differences in the number of calories between the three types of hot dogs?

Numerical Summaries

To explore the concept of using a numerical summary, let's take a look at the `OrangeJuice_quality` data set to see if the amount of added sugar and a juice's rated quality are related.

In [ ]:

```
Juice = pd.read_csv('../data/OrangeJuice_quality.csv' )
Juice.head(10)
```

In [ ]:

```
Juice = Juice.drop('rank', axis=1)
```

Using the `describe()` method in conjunction with `groupby()` we can generate a table that summarizes the numerical results associated with each category of our explanatory variable:

In [ ]:

```
Juice.groupby('quality').describe()
```

We see here that the average amount of sugar is higher for Juices that rate higher on the quality scale. The distribution of values however is heavy to one side for the `Good` and `Poor` categories, which indicates that we may want to have a closer look at these two to see if there are any outliers affecting what we see.

Side-by-Side Boxplots

As we saw in the comparison of `array_1` and `array_3` , we can compare boxplots side-by-side to see if their distributions are similar. If your data set includes a categorical explanatory variable, setting this variable as the x axis will automatically yield a **side-by-side boxplot** without the need to stack the boxplots together manually.

Let's have another look at the residual sugar data:

In [ ]:

```
Juice.head()
```

In [ ]:

```
Juice.boxplot(by = "quality",fontsize=12, figsize = (5, 5))
plt.show()
```

In [ ]:

```
sns.boxplot(x = "quality", y = "sugar", data = Juice).set_tit
```

Aha! We see that although there seems to be a marked difference in the residual sugar content of each category, there also seem to be quite a few outliers in our set. How do you think removing these outliers will affect our data set?

C. Quantitative Explanatory and Quantitative Response

If both our explanatory and response variables are quantitative, we can investigate the existince of a relationship by gauging the increment/decrement in the response variable with respect to changes in the explanatory variable. This is best gauged using a **scatter plot**, and can further be refined using **Pearson's correlation coefficient**.

Example

1. Can we predict a student's freshman year GPA from his/her SAT score?
2. What is the relationship between driver's age and the legibility distance of a sign (the maximum distance at which the driver can read a sign)?

Scatter Plots

Scatter plots are the go-to tool for determining the nature of relationship between two quantitative variables. They are obtained by plotting one variable against another on the Cartesian plane, with points scattered across the xy-coordinates according to the data set.

Let's examine if there is any relationship between a student's math score and his/her reading score for a group of students as given in the `exams` data set. First and foremost, let's have a quick look around at the data to see what we're working with:

In [ ]:

```
examScore = pd.read_csv("../data/exams.csv")
examScore = examScore.loc[:,("math.score", "reading.score", "
examScore.info()
```

In [ ]:

```
examScore.head()
```

Alright. So we see that there are three variables - `math.score` , `reading.score` , and `writing.score` . Let's have a look at the math and reading scores.

In [ ]:

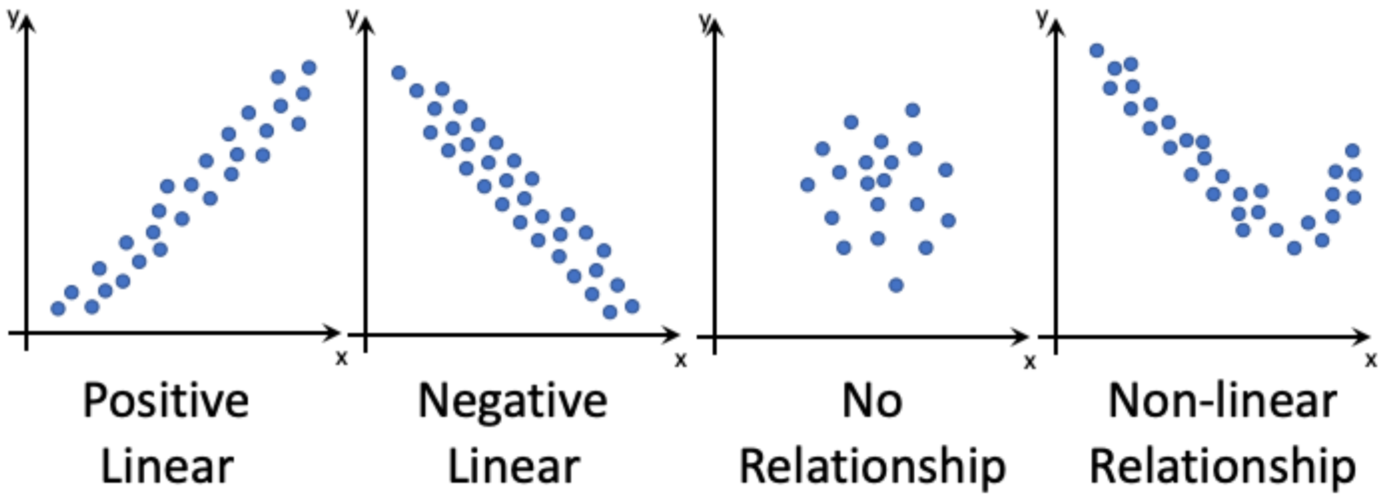
```
plt.scatter(x = "math.score", y = "reading.score", data = exa
plt.title("Math Score by Reading Score")
```

In [ ]:

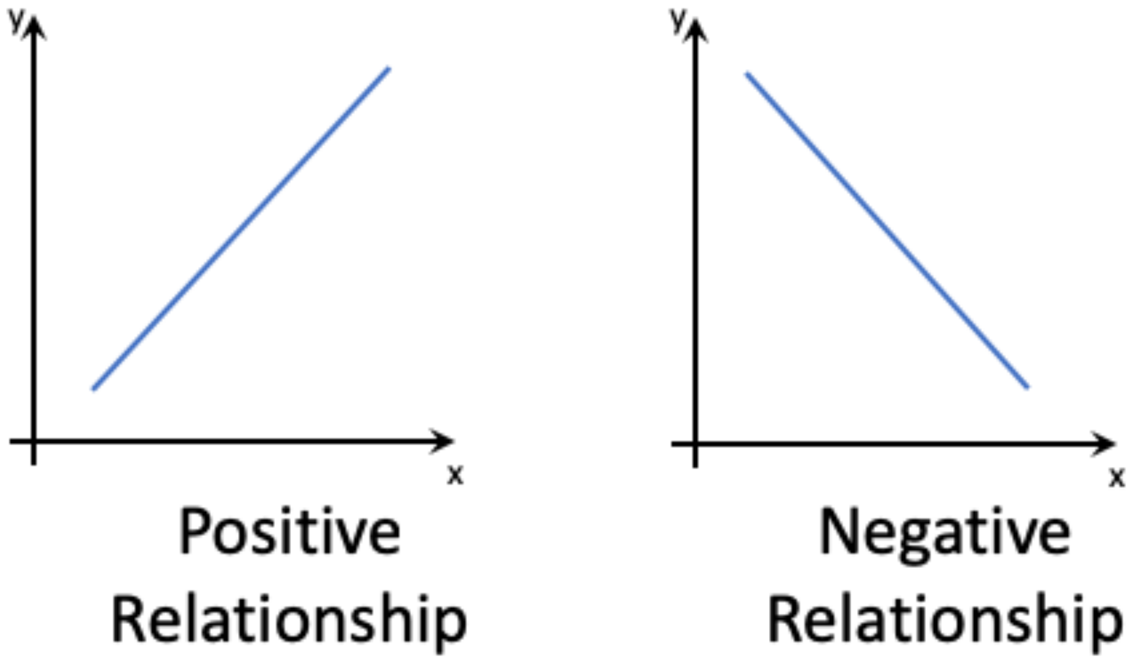
```
sns.scatterplot(x = "reading.score", y = "math.score", data =
plt.title("Math Score by Reading Score")
```

We see that as the reading scores go higher, so do the math scores. This makes sense - a student who's poorer at reading may not be able to understand the exam questions, translating to a lower math score. We don't have enough information to verify this theory just yet, but examining the relationship using a scatter plot has given us some insight into how the two are related!

When describing the distribution of a single quantitative variable with a histogram, we describe the overall pattern of the distribution (shape, center, spread) and any deviations from that pattern (outliers). The scatter plot here does the same thing, albeit with 2 variables. Two important pieces of information that the scatter plot gives us is the nature (linear/non-linear) of relationship and its direction (positive/negative).



A relationship between two variables is classified as **linear** if the pattern of distribution for our data is centered about a straight line, i.e. we can plot a straight line that cuts through the data such that the points are evenly distributed across the line. If the pattern is distributed along some other form of curve, we say it is **non-linear**. If the points appear to be randomly strewn across the plot, we say that no relationship between the variables exist.



On the other hand, the direction of a relationship between two quantitative variables can be **positive** or **negative**. A positive (or **increasing**) relationship means that an increase in one of the variables is associated with an increase in the other. A negative (or **decreasing**) relationship means that an increase in one of the variables is associated with a decrease in the other. One important thing to note here is that not all relationships can be classified as either positive or negative - you can encounter cases where the values swing back and forth in both directions.

Linear Relationships and Pearson's Correlation Coefficient

Although a scatter plot gives us valuable information in terms of the nature and direction of a relationship between two quantitative variables, what it fails to do is quantify the **strength** of said relationship. To that extent, we would like to utilize some numerical measure to assess *how much* of an impact one variable has on another.

Recall the case of a **linear relationship** - a relationship between two quantitative variables such that the pattern of distribution for our data is centered about a **straight line.** The plot below shows that we can indeed see somewhat of a linear pattern in our selected data.

In [ ]:

```
sns.lmplot(x = "reading.score", y  = "math.score", data = exa
           ci = False).fig.suptitle("Math Score by Reading Sc
plt.show()
```

The **strength** of a linear relationship can be assessed using a numerical measure

The **strength** of a linear relationship can be assessed using a numerical measure called **Pearson's correlation coefficient**, or **Pearson's r** for short.

For any two quantitative variables X and Y, Pearson's correlation coefficient (denoted by $\text{cor}(X,Y)$ or r_{xy}) measures on average, how much an increase (decrease) in X is related to an increase (decrease) in Y. The sign of the correlation value (+/-) designates the direction of the relationship, i.e. positive or negative.

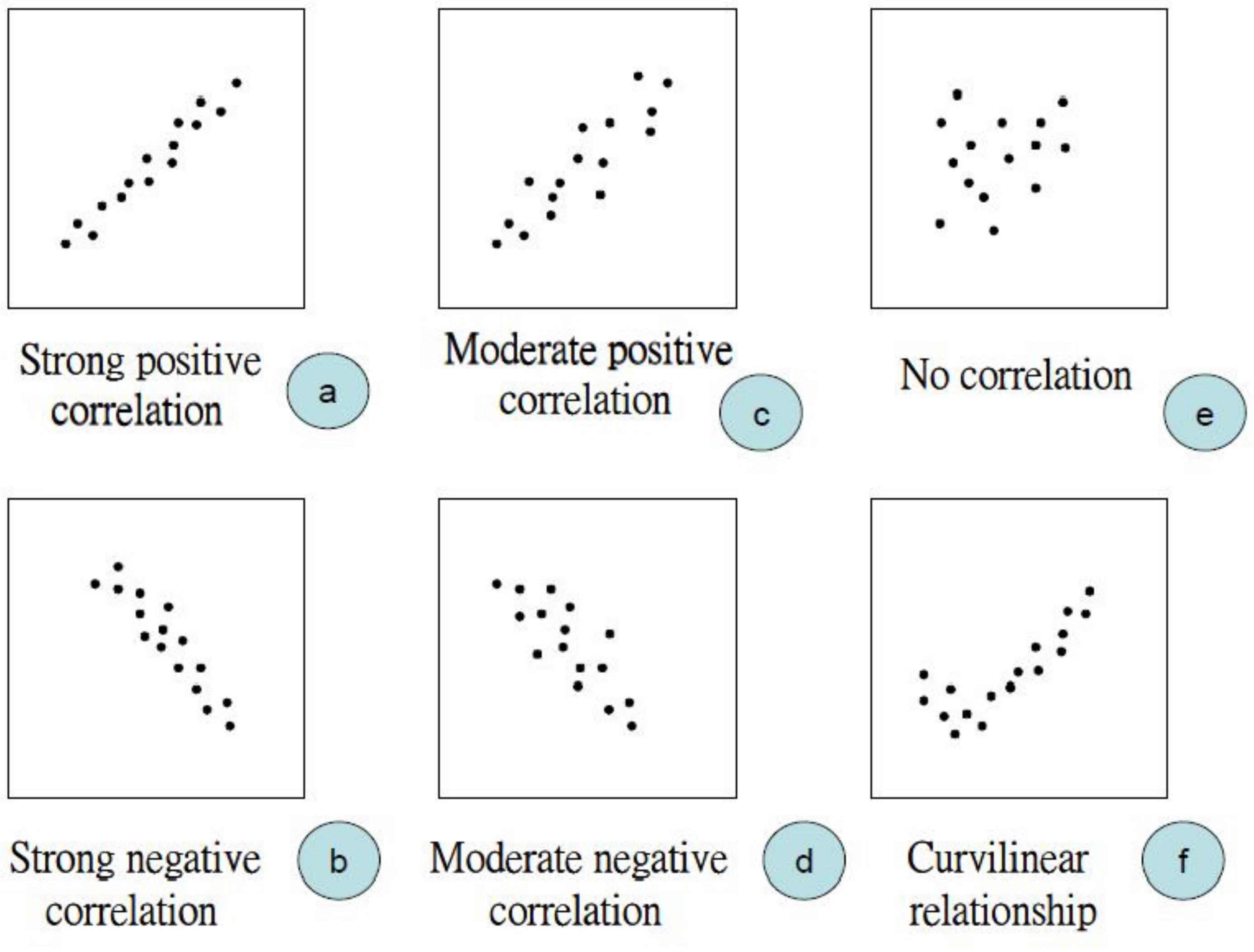
Mathematically, Pearson's correlation coefficient is defined as:

$$\text{cor}(X,Y) = r_{xy} = \frac{\sum_i (x_i - \bar{x})(y_i - \bar{y})}{s_x * s_y}$$

$$\text{cor}(X,Y) = r_{xy} = \frac{\sum_i (x_i - \bar{x})(y_i - \bar{y})}{s_x * s_y}$$

where $\bar{x}$ and $\bar{y}$ are the means of X and Y, and s_x and s_y are the standard deviation of X and Y respectively.

The correlation coefficient takes values **only** between -1 and 1, with +/-1 denoting a perfect positive/negative linear relationship and 0 denoting the absence of any relationship. Any other values taken within the $(-1,1)$ interval indicate the strength of the relationship in a given direction (+/-). The figure below gives a summary of how we classify these relationships:



Unfortunately, there are no clear rules on what values of the correlation coefficient constitute "Strong", "Moderate", or "Weak" correlations as various authors have been known prescribe arbitrary limits for their respective fields. For the purpose of this course, we'll use the following definition:

Strength of Relationship	Correlation
Very Weak	0.01 - 0.20
Weak	0.21 - 0.40
Moderate	0.41 - 0.60
Strong	0.61 - 0.80
Very Strong	0.81 - 0.99

Now let's have a look at quantifying the correlation for the relationship we examined earlier between the math and reading scores. In `numpy` , this can be done using the `corrcoef` function, which calculates the correlation between any two specified variables.

In [ ]:

```
np.corrcoef(examScore.loc[:, "math.score"], examScore.loc[:,
```

or alternatively, using the `corr()` method of `pandas` DataFrame objects:

In [ ]:

```
examScore.corr()
```

These functions generate a correlation matrix, i.e. a table of all possible pairs of variables and their related correlations. The diagonal of this matrix is always 1 as the correlation between any variable and itself must be perfect. Additionally, correlation is also symmetric, i.e. corr(X,Y)=corr(Y,X). This can be seen in the table as values above the diagonal are "reflected" to the bottom half.

Now let's have a look at the scatter plot for the reading and writing scores to verify if the strong positive relationship between these 2 variables that we see in the table is true.

In [ ]:

```
sns.lmplot(x = "reading.score", y = "writing.score", data = e
           ci = False)

ax = plt.gca() # get current axis
ax.set_title("There is a strong positive correlation between
ax.set_xlabel("Reading score")
ax.set_ylabel("Writing score")
plt.show()
```

Yup, it does!

Exercise

Explore all two-way relationships between variables in the `iris` dataset.

In [ ]:

```
iris.head()
```

In [ ]:

```
iris.describe()
```

In [ ]:

```
# Species ---> Petal.Length ,  Petal.Width , etc. (Categorica

# Numerical Summaries
# Side by Side Box Plot
```

In [ ]:

```

```

In [ ]:

```

```

In [ ]:

```
# Petal.Length ---> Petal.Width ,  Setal.Length , etc. (Categ

# Pearson Correlation Coefficient
# Scatter Plot
```

In [ ]:

```

```

In [ ]:

```

```

In [ ]:

```

```