[Designing and Implementing a Data Science Solution on Azure](https://docs.microsoft.com/learn/certifications/exams/dp-100/).

Contents

1. [Explore and analyze data with Python](https://docs.microsoft.com/en-us/learn/modules/explore-analyze-data-with-python/)

[Exam DP-100: Designing and Implementing a Data Science Solution on Azure](https://docs.microsoft.com/learn/certifications/exams/dp-100/).

# [Explore and analyze data with Python](https://docs.microsoft.com/en-us/learn/modules/explore-analyze-data-with-python/)

Python provides extensive functionality with powerful statistical and numerical libraries:

* NumPy and Pandas simplify analyzing and manipulating data
* Matplotlib provides attractive data visualizations
* Scikit-learn offers simple and effective predictive data analysis
* TensorFlow and PyTorch supply machine learning and deep learning capabilities

Usually, **a data analysis project** is designed to establish insights around a particular scenario or to test a hypothesis.

# **Explore data with NumPy and Pandas**

**NumPy** is a Python library that gives functionality comparable to mathematical tools such as MATLAB and R. While NumPy significantly simplifies the user experience, it also offers comprehensive mathematical functions.

**Pandas** is an extremely popular Python library for data analysis and manipulation. Pandas is like excel for Python - providing easy-to-use functionality for data tables.

## **Testing hypotheses**

Data exploration and analysis is typically **an *iterative* process, i**n which the data scientist takes a sample of data and performs the following kinds of task to analyze it and test hypotheses:

* **Clean data** to handle errors, missing values, and other issues.
* **Apply statistical techniques to better understand the data**, and how the sample might be expected to represent the real-world population of data, allowing for random variation.
* **Visualize data** to determine relationships between variables, and in the case of a machine learning project, identify *features* that are potentially predictive of the *label*.
* **Revise the hypothesis** and repeat the process.

## **Jupyter Notebooks environment**

* Notebooks are made up of *cells*. Some cells (like this one) contain *markdown* text, while others (like the one beneath this one) contain code.
* You can run each code cell by using the ► Run button. the ► Run button will show up when you hover over the cell.
* The output from each code cell will be displayed immediately below the cell.
* Even though the code cells can be run individually, some variables used in the code are global to the notebook. That means that you should run all of the code cells in order. There may be dependencies between code cells, so if you skip a cell, subsequent cells might not run correctly.

## [**Exploring data arrays with NumPy**](https://docs.microsoft.com/en-us/learn/modules/explore-analyze-data-with-python/3-exercise-explore-data)

Load a sample of student grades for a data science class into a **Python List.**

data = [50,50,47,97,49,3,53,42,26,74,82,62,37,15,70,27,36,35,48,52,63,64]

print(data)

OUTPUT: [50, 50, 47, 97, 49, 3, 53, 42, 26, 74, 82, 62, 37, 15, 70, 27, 36, 35, 48, 52, 63, 64]

Load the data into a **NumPy array.**

import numpy as np

grades = np.array(data)

print(grades)

OUTPUT:[50 50 47 97 49 3 53 42 26 74 82 62 37 15 70 27 36 35 48 52 63 64]

**The differences between a list and a NumPy array**

How to IDENTIFY each: Both are in [ ]**.But** a list has a comma , between entries, Numphy Array does not.

Multiplying a list by 2 creates a new list of twice the length with the original sequence of list elements repeated.

Multiplying a NumPy array by 2 performs an element-wise calculation, so we end up with an array of the same size in which each element has been multiplied by 2. (like a vector would)

NumPy arrays are specifically designed to support mathematical operations on numeric data - which makes them more useful for data analysis than a generic list.

print (type(data),'x 2:', data \* 2)

print('---')

print (type(grades),'x 2:', grades \* 2)

OUTPUT: <class 'list'> x 2: [50, 50, 47, 97, 49, 3, 53, 42, 26, 74, 82, 62, 37, 15, 70, 27, 36, 35, 48, 52, 63, 64, 50, 50, 47, 97, 49, 3, 53, 42, 26, 74, 82, 62, 37, 15, 70, 27, 36, 35, 48, 52, 63, 64]

---

OUTPUT: <class 'numpy.ndarray'> x 2: [100 100 94 194 98 6 106 84 52 148 164 124 74 30 140 54 72 70 96 104 126 128]

The class type for the numpy array above is a **numpy.ndarray**. The nd indicates that this is a structure that can consists of **multiple *dimensions*** (it can have *n* dimensions). Our specific instance has a single dimension of student grades.

grades.shape

OUTPUT:(22,)

this array has only one dimension, which contains 22 elements (there are 22 grades in the original list). You can access the individual elements in the array by their zero-based ordinal position.

**Get the first element (the one in position 0).**

grades[0]

OUTPUT: 50

grades.mean()

OUTPUT: 50

the mean grade is just around 50 - more or less in the middle of the possible range from 0 to 100.

Add a second set of data for the same students, the typical number of hours per week they devoted to studying.

# Define an array of study hours

study\_hours = [10.0,11.5,9.0,16.0,9.25,1.0,11.5,9.0,8.5,14.5,15.5,

13.75,9.0,8.0,15.5,8.0,9.0,6.0,10.0,12.0,12.5,12.0]

# Create a 2D array (an array of arrays)

student\_data = np.array([study\_hours, grades])

# display the array

student\_data

OUTPUT: array([[10., 11.5, 9., 16., 9.25, 1., 11.5, 9., 8.5, 14.5,15.5, 13.75, 9., 8., 15.5, 8., 9., 6., 10., 12., 12.5, 12.],

[50., 50., 47., 97., 49., 3., 53., 42., 26., 74., 82., 62., 37., 15., 70., 27., 36., 35., 48., 52., 63., 64.]])

student\_data.shape

OUTPUT: (2, 22)

Now you have a **multidimensional array.**

To navigate this structure, you need to specify the position of each element in the hierarchy. So to find the first value in the first array (which contains the study hours data), you can use the following code.

# Show the first element of the first element

student\_data[0][0]

OUTPUT: 10.0

# Get the mean value of each sub-array

avg\_study = student\_data[0].mean()

avg\_grade = student\_data[1].mean()

print('Average study hours: {:.2f}\nAverage grade: {:.2f}'.format(avg\_study, avg\_grade))

OUTPUT: Average study hours: 10.52

Average grade: 49.18

## **Exploring tabular data with Pandas**

the Pandas DataFrame is a more convenient structure to work with two-dimensional tables of data,

import pandas as pd

df\_students = pd.DataFrame({'Name': ['Dan', 'Joann', 'Pedro', 'Rosie', 'Ethan', 'Vicky', 'Frederic', 'Jimmie', 'Rhonda', 'Giovanni', 'Francesca', 'Rajab', 'Naiyana', 'Kian', 'Jenny', 'Jakeem', 'Helena',' Ismat','Anila', 'Skye', 'Daniel', 'Aisha'],

'StudyHours':student\_data[0],

'Grade':student\_data[1]})

df\_students

OUTPUT:

|  | **Name** | **StudyHours** | **Grade** |
| --- | --- | --- | --- |
| **0** | Dan | 10.00 | 50.0 |
| **1** | Joann | 11.50 | 50.0 |
| **2** | Pedro | 9.00 | 47.0 |
| **3** | Rosie | 16.00 | 97.0 |
| **4** | Ethan | 9.25 | 49.0 |
| **5** | Vicky | 1.00 | 3.0 |
| **6** | Frederic | 11.50 | 53.0 |
| **7** | Jimmie | 9.00 | 42.0 |
| **8** | Rhonda | 8.50 | 26.0 |
| **9** | Giovanni | 14.50 | 74.0 |
| **10** | Francesca | 15.50 | 82.0 |
| **11** | Rajab | 13.75 | 62.0 |
| **12** | Naiyana | 9.00 | 37.0 |
| **13** | Kian | 8.00 | 15.0 |
| **14** | Jenny | 15.50 | 70.0 |
| **15** | Jakeem | 8.00 | 27.0 |
| **16** | Helena | 9.00 | 36.0 |
| **17** | Ismat | 6.00 | 35.0 |
| **18** | Anila | 10.00 | 48.0 |
| **19** | Skye | 12.00 | 52.0 |
| **20** | Daniel | 12.50 | 63.0 |
| **21** | Aisha | 12.00 | 64.0 |

the DataFrame includes an ***index*** to uniquely identify each row.

### **Finding and filtering data in a DataFrame**

use the DataFrame's **loc** method to retrieve data for a specific index value, like this.

# Get the data for index value 5

df\_students.loc[5]

OUTPUT:

Name Vicky

StudyHours 1.0

Grade 3.0

Name: 5, dtype: object

You can also get the data at a range of index values, like this:

# Get the rows with index values from 0 to 5

df\_students.loc[0:5]

OUTPUT:

|  | **Name** | **StudyHours** | **Grade** |
| --- | --- | --- | --- |
| 0 | Dan | 10.00 | 50.0 |
| 1 | Joann | 11.50 | 50.0 |
| 2 | Pedro | 9.00 | 47.0 |
| 3 | Rosie | 16.00 | 97.0 |
| 4 | Ethan | 9.25 | 49.0 |
| 5 | Vicky | 1.00 | 3.0 |

you can use the **iloc** method to find rows based on their ordinal position in the DataFrame (regardless of the index):

# Get data in the first five rows

Df\_students.iloc[0:5]

OUTPUT:

|  | Name | StudyHours | Grade |
| --- | --- | --- | --- |
| 0 | Dan | 10.00 | 50.0 |
| 1 | Joann | 11.50 | 50.0 |
| 2 | Pedro | 9.00 | 47.0 |
| 3 | Rosie | 16.00 | 97.0 |
| 4 | Ethan | 9.25 | 49.0 |

The **loc method** returned rows with **index** *label* in the list of values from *0* to *5* - which includes *0*, *1*, *2*, *3*, *4*, and *5* (six rows).

**The iloc method** returns the rows in the ***positions*** included in the range 0 to 5, and since integer ranges don't include the upper-bound value, this includes positions *0*, *1*, *2*, *3*, and *4* (five rows).

iloc identifies data values in a DataFrame by *position*,by rows AND columns. Yyou can use it to find the values for the columns in positions 1 and 2 in row 0, like this:

df\_students.iloc[0,[1,2]]

OUTPUT:

StudyHours 10.0

Grade 50.0

Name: 0, dtype: object

the loc method, the **rows** in our dataframe are indexed as **integer** values, but the **columns** are identified by **name**:

df\_students.loc[0,'Grade']

OUTPUT: 50.0

use the loc method to find indexed rows based on a filtering expression that references named columns other than the index

df\_students.loc[df\_students['Name']=='Aisha']

OUTPUT:

|  | **Name** | **StudyHours** | **Grade** |
| --- | --- | --- | --- |
| **21** | Aisha | 12.0 | 64.0 |

you don't need to explicitly use the loc method to do this - you can simply apply a DataFrame filtering expression, like this:

df\_students[df\_students['Name']=='Aisha']

OUTPUT:

|  | **Name** | **StudyHours** | **Grade** |
| --- | --- | --- | --- |
| **21** | Aisha | 12.0 | 64.0 |

you can achieve the same results by using the DataFrame's query method, like this:

df\_students.query('Name=="Aisha"')

OUTPUT:

|  | **Name** | **StudyHours** | **Grade** |
| --- | --- | --- | --- |
| **21** | Aisha | 12.0 | 64.0 |

**Referring to a DataFrame column name.**

1. You can specify the column name as a named index value (as in the df\_students['Name'] or
2. you can use the column as a property of the DataFrame::

df\_students[df\_students.Name == 'Aisha']

### **Loading a DataFrame from a file**

DataFrame's **read\_csv** method is used to load data from text files

!wget https://raw.githubusercontent.com/MicrosoftDocs/mslearn-introduction-to-machine-learning/main/Data/ml-basics/grades.csv

df\_students = pd.read\_csv('grades.csv',delimiter=',',header='infer')

df\_students.head()

OUTPUT:

|  | Name | StudyHours | Grade |
| --- | --- | --- | --- |
| 0 | Dan | 10.00 | 50.0 |
| 1 | Joann | 11.50 | 50.0 |
| 2 | Pedro | 9.00 | 47.0 |
| 3 | Rosie | 16.00 | 97.0 |
| 4 | Ethan | 9.25 | 49.0 |

### **Handling missing values**

identify which individual values are null

df\_students.isnull()

can get the sum of missing values for each column

df\_students.isnull().sum()

filter the dataframe to include only rows where any of the columns (axis 1 of the DataFrame) are null.

df\_students[df\_students.isnull().any(axis=1)]

missing numeric values show up as **NaN (*not a number*).**

*impute* replacement values with say the average, with the **fillna** method, like this:

df\_students.StudyHours = df\_students.StudyHours.fillna(df\_students.StudyHours.mean())

df\_students

Alternatively, drop rows or columns that contains null values by using the dropna method.

df\_students = df\_students.dropna(axis=0, how='any')

df\_students

### **Explore data in the DataFrame**

# Get the mean study hours using to column name as an index

mean\_study = df\_students['StudyHours'].mean()

# Get the mean grade using the column name as a property (just to make the point!)

mean\_grade = df\_students.Grade.mean()

# Print the mean study hours and mean grade

print('Average weekly study hours: {:.2f}\nAverage grade: {:.2f}'.format(mean\_study, mean\_grade))

# Get students who studied for the mean or more hours

df\_students[df\_students.StudyHours > mean\_study]

# What was their mean grade?

df\_students[df\_students.StudyHours > mean\_study].Grade.mean()

**add a new column** to the DataFrame, indicating whether or not each student passed.

passes = pd.Series(df\_students['Grade'] >= 60)

df\_students = pd.concat([df\_students, passes.rename("Pass")], axis=1)

Use the **groupby** method to group the student data into groups based on the Pass column

print(df\_students.groupby(df\_students.Pass).Name.count())

**aggregate multiple fields i**n a group using any available aggregation function

# Find the mean study time and grade for the groups of students who passed and failed the course.

print(df\_students.groupby(df\_students.Pass)['StudyHours', 'Grade'].mean())

Many DataFrame operations return a new copy of the DataFrame; so if you want to **modify a DataFrame but keep the existing variable(dataframe)**, you need to assign the result of the operation to the existing variable(dataframe).

# Create a DataFrame with the data sorted by Grade (descending)

df\_students = df\_students.sort\_values('Grade', ascending=False)

# [**Visualize data**](https://docs.microsoft.com/en-us/learn/modules/explore-analyze-data-with-python/4-visualize-data)

typically means graphing it. This is done to provide a fast qualitative assessment of our data, which can be useful for understanding results, finding outlier values, understanding how numbers are distributed, and so on.

Often, when we're trying to understand raw data or results, we may experiment with different types of graphs until we come across one that explains the data in a visually intuitive way.

Graphs can be used to explore other kinds of data, like proportions - shown through pie charts, stacked bar graphs - how data are spread - with histograms, box and whisker plots - and how two data sets differ.

## [**Visualize data with Matplotlib**](https://docs.microsoft.com/en-us/learn/modules/explore-analyze-data-with-python/5-exercise-visualize-data)

a simple bar chart that shows the grade of each student.

# Ensure plots are displayed inline in the notebook

%matplotlib inline

from matplotlib import pyplot as plt

# Create a bar plot of name vs grade

plt.bar(x=df\_students.Name, height=df\_students.Grade)

# Display the plot

plt.show()

you used the pyplot class from Matplotlib to plot the chart. This class provides a whole bunch of ways to improve the visual elements of the plot. For example, the following code:

* Specifies the color of the bar chart.
* Adds a title to the chart (so we know what it represents)
* Adds labels to the X and Y (so we know which axis shows which data)
* Adds a grid (to make it easier to determine the values for the bars)
* Rotates the X markers (so we can read them)

# Create a bar plot of name vs grade

plt.bar(x=df\_students.Name, height=df\_students.Grade, color='orange')

# Customize the chart

plt.title('Student Grades')

plt.xlabel('Student')

plt.ylabel('Grade')

plt.grid(color='#95a5a6', linestyle='--', linewidth=2, axis='y', alpha=0.7)

plt.xticks(rotation=90)

# Display the plot

plt.show()

A plot is technically contained with a Figure. In the previous examples, the figure was created implicitly for you; but you can create it explicitly. For example, the following code creates a figure with a specific size.

# Create a Figure

fig = plt.figure(figsize=(8,3))

# Create a bar plot of name vs grade

plt.bar(x=df\_students.Name, height=df\_students.Grade, color='orange')

# Customize the chart

plt.title('Student Grades')

plt.xlabel('Student')

plt.ylabel('Grade')

plt.grid(color='#95a5a6', linestyle='--', linewidth=2, axis='y', alpha=0.7)

plt.xticks(rotation=90)

# Show the figure

plt.show()

A figure can contain multiple subplots, each on its own *axis*.

For example, the following code creates a figure with two subplots - one is a bar chart showing student grades, and the other is a pie chart comparing the number of passing grades to non-passing grades.

# Create a figure for 2 subplots (1 row, 2 columns)

fig, ax = plt.subplots(1, 2, figsize = (10,4))

# Create a bar plot of name vs grade on the first axis

ax[0].bar(x=df\_students.Name, height=df\_students.Grade, color='orange')

ax[0].set\_title('Grades')

ax[0].set\_xticklabels(df\_students.Name, rotation=90)

# Create a pie chart of pass counts on the second axis

pass\_counts = df\_students['Pass'].value\_counts()

ax[1].pie(pass\_counts, labels=pass\_counts)

ax[1].set\_title('Passing Grades')

ax[1].legend(pass\_counts.keys().tolist())

# Add a title to the Figure

fig.suptitle('Student Data')

# Show the figure

fig.show()

many packages, including Pandas, provide methods that abstract the underlying Matplotlib functions and simplify plotting. For example, the DataFrame provides its own methods for plotting data, as shown in the following example to plot a bar chart of study hours.

df\_students.plot.bar(x='Name', y='StudyHours', color='teal', figsize=(6,4))

## **Getting started with statistical analysis**

### **Descriptive statistics and data distribution**

data scientists are particularly interested in the *distribution* of data

visualize the data as a histogram, and see how frequently each value for the variable occurs.

# Get the variable to examine

var\_data = df\_students['Grade']

# Create a Figure

fig = plt.figure(figsize=(10,4))

# Plot a histogram

plt.hist(var\_data)

# Add titles and labels

plt.title('Data Distribution')

plt.xlabel('Value')

plt.ylabel('Frequency')

# Show the figure

fig.show()

#### **Measures of central tendency**

Ie. describing statistics that represent the "middle" of the data

Common ways to define the middle of the data include:

* The *mean*: A simple average based on adding together all of the values in the sample set, and then dividing the total by the number of samples.
* The *median*: The value in the middle of the range of all of the sample values.
* The *mode*: The most commonly occuring value in the sample set\*.

# Get the variable to examine

var = df\_students['Grade']

# Get statistics

min\_val = var.min()

max\_val = var.max()

mean\_val = var.mean()

med\_val = var.median()

mod\_val = var.mode()[0]

print('Minimum:{:.2f}\nMean:{:.2f}\nMedian:{:.2f}\nMode:{:.2f}\nMaximum:{:.2f}\n'.format(min\_val,

mean\_val,

med\_val,

mod\_val,

max\_val))

# Create a Figure

fig = plt.figure(figsize=(10,4))

# Plot a histogram

plt.hist(var)

# Add lines for the statistics

plt.axvline(x=min\_val, color = 'gray', linestyle='dashed', linewidth = 2)

plt.axvline(x=mean\_val, color = 'cyan', linestyle='dashed', linewidth = 2)

plt.axvline(x=med\_val, color = 'red', linestyle='dashed', linewidth = 2)

plt.axvline(x=mod\_val, color = 'yellow', linestyle='dashed', linewidth = 2)

plt.axvline(x=max\_val, color = 'gray', linestyle='dashed', linewidth = 2)

# Add titles and labels

plt.title('Data Distribution')

plt.xlabel('Value')

plt.ylabel('Frequency')

# Show the figure

fig.show()

Another way to visualize the distribution of a variable is to use a *box* plot (sometimes called a *box-and-whiskers* plot). Let's create one for the grade data.

The *box* part of the plot shows where the inner two *quartiles* of the data reside

The *whiskers* extending from the box show the outer two quartiles

The line in the box indicates the *median* value.

# Get the variable to examine

var = df\_students['Grade']

# Create a Figure

fig = plt.figure(figsize=(10,4))

# Plot a histogram

plt.boxplot(var)

# Add titles and labels

plt.title('Data Distribution')

# Show the figure

fig.show()

**Combine histograms and box plots,** with the box plot's orientation changed to align it with the histogram

# Create a function that we can re-use

def show\_distribution(var\_data):

from matplotlib import pyplot as plt

# Get statistics

min\_val = var\_data.min()

max\_val = var\_data.max()

mean\_val = var\_data.mean()

med\_val = var\_data.median()

mod\_val = var\_data.mode()[0]

print('Minimum:{:.2f}\nMean:{:.2f}\nMedian:{:.2f}\nMode:{:.2f}\nMaximum:{:.2f}\n'.format(min\_val, mean\_val,

med\_val,

mod\_val,

max\_val))

# Create a figure for 2 subplots (2 rows, 1 column)

ig, ax = plt.subplots(2, 1, figsize = (10,4))

Plot the histogram

ax[0].hist(var\_data)

ax[0].set\_ylabel('Frequency')

# Add lines for the mean, median, and mode

ax[0].axvline(x=min\_val, color = 'gray', linestyle='dashed', linewidth = 2)

ax[0].axvline(x=mean\_val, color = 'cyan', linestyle='dashed', linewidth = 2)

ax[0].axvline(x=med\_val, color = 'red', linestyle='dashed', linewidth = 2)

ax[0].axvline(x=mod\_val, color = 'yellow', linestyle='dashed', linewidth = 2)

ax[0].axvline(x=max\_val, color = 'gray', linestyle='dashed', linewidth = 2)

# Plot the boxplot

ax[1].boxplot(var\_data, vert=False)

ax[1].set\_xlabel('Value')

# Add a title to the Figure

fig.suptitle('Data Distribution')

# Show the figure

fig.show()

# Get the variable to examine

col = df\_students['Grade']

# Call the function

show\_distribution(col)

**Statistics is fundamentally about taking *samples* of data and using probability functions to extrapolate information about the full *population* of data.**

***Samples*** refer to the data we have on hand - such as information about these 22 students' study habits and grades.

***Population*** refers to all possible data we could collect - such as every student's grades and study habits across every educational institution throughout the history of time.

Usually we're interested in the population but it's simply not practical to collect all of that data. Instead, we need to try estimate what the population is like from the small amount of data (samples) that we have.

If we have enough samples, we can calculate a ***probability density function***, which **estimates the distribution of grades for the full population.**

The pyplot class from Matplotlib provides a helpful plot function to show this density.

def show\_density(var\_data):

from matplotlib import pyplot as plt

fig = plt.figure(figsize=(10,4))

# Plot density

var\_data.plot.density()

# Add titles and labels

plt.title('Data Density')

# Show the mean, median, and mode

plt.axvline(x=var\_data.mean(), color = 'cyan', linestyle='dashed', linewidth = 2)

plt.axvline(x=var\_data.median(), color = 'red', linestyle='dashed', linewidth = 2)

plt.axvline(x=var\_data.mode()[0], color = 'yellow', linestyle='dashed', linewidth = 2)

# Show the figure

plt.show()

# Get the density of Grade

col = df\_students['Grade']

show\_density(col)

"bell curve" is what statisticians call a ***normal* distribution** with the

mean at the center and

mode at the center and

symmetric tails.

See the following documentation:

* [NumPy](https://numpy.org/doc/stable/)
* [Pandas](https://pandas.pydata.org/pandas-docs/stable/)
* [Matplotlib](https://matplotlib.org/contents.html)

## [**Examine real world data**](https://docs.microsoft.com/en-us/learn/modules/explore-analyze-data-with-python/6-examine-real-world-data)

‘real world’ data, raw data has to be inspected for issues before being used.

Best practice is to **inspect the raw data and process it before use**, which reduces errors or issues, typically by:

removing erroneous data points or

modifying the data into a more useful form.

Most real-world data are **influenced by factors that weren't recorded** at the time. For example, we might have a table of race-car track times alongside engine sizes, but various other factors that weren't written down—such as the weather—probably also played a role. If problematic, the influence of these factors can often be reduced by increasing the size of the dataset.

Data points can be clearly outside of what is expected—also known as **‘*outliers*’**—can sometimes be safely removed from analyses, though care must be taken to not remove data points that provide real insights.

**Bias** refers to a tendency to select certain types of values more frequently than others, in a way that misrepresents the underlying population, or ‘real world’. Bias can sometimes be identified by exploring data while keeping in mind basic knowledge about where the data came from.

Real-world data will always have issues, but this is often a surmountable problem. Remember to:

* Check for missing values and badly recorded data
* Consider removal of obvious outliers
* Consider what real-world factors might affect your analysis and consider if your dataset size is large enough to handle this
* Check for biased raw data and consider your options to fix this, if found

# [**Exercise - Examine real world data**](https://docs.microsoft.com/en-us/learn/modules/explore-analyze-data-with-python/7-exercise-real-world-data)

look into more complex cases, describe the data more fully, and discuss how to make basic comparisons between data.

Outliers can occur for many reasons. Maybe a student meant to record "10" hours of study time, but entered "1" and missed the "0". Or maybe the student was abnormally lazy when it comes to studying! Either way, it's a statistical anomaly that doesn't represent a typical student.

In the real world, though, it would be **unusual to exclude data at the extremes without more justification** when our sample size is so small.

use the Pandas **quantile** function to exclude observations below say, 0.01th percentile (the value above which 99% of the data reside).

Tip: You can also eliminate outliers at the upper end of the distribution by defining a threshold at a high percentile value - for example, you could use the quantile function to find the 0.99 percentile below which 99% of the data reside.

***right skewed* distribution.** The mass of the data is on the left side of the distribution, creating a long tail to the right because of the values at the extreme high end; which pull the mean to the right.

#### **Measures of variance**

We should examine **how much variability is there in the data**?

Typical statistics that measure variability in the data include:

* **Range**: The difference between the maximum and minimum. There's no built-in function for this, but it's easy to calculate using the min and max functions.
* **Variance**: The average of the squared difference from the mean. You can use the built-in var function to find this.
* **Standard** **Deviation**: The square root of the variance. You can use the built-in std function to find this.

The **standard deviation is generally the most useful**

The **higher** the standard deviation, the **more variance** there is when comparing values in the distribution to the distribution mean - in other words, the **data is more spread out.**

***normal* distribution,** the standard deviation works with the particular characteristics of a normal distribution to provide even greater insight.

## **Comparing data**

### **Comparing numeric and categorical variables**

comparing the numeric StudyHours column to the categorical Pass column to see if there's an apparent relationship

### **Comparing numeric variables**

A common technique when dealing with numeric data in different scales is to ***normalize*** the data so that the **values retain their proportional distribution**, but are **measured on the same scale.**

To accomplish this, we'll use a technique called ***MinMax* scaling** that distributes the values proportionally on a **scale of 0 to 1.**

Scikit-Learn library provides MinMaxScaler to do it for you.

The **correlation** statistic is a value between **-1 and 1** that indicates the **strength of a relationship**.

Values **above** **0 indicate a *positive* correlation (**high values of one variable tend to coincide with high values of the other)

values **below 0 indicate a *negative* correlation** (high values of one variable tend to coincide with low values of the other).

Warning - Math Ahead!Cast your mind back to when you were learning how to solve linear equations in school, and recall that the *slope-intercept* form of a linear equation looks like this:

*y = mx + b*

In this equation,

*y* and *x* are the coordinate variables,

*m* is the slope of the line, and

*b* is the y-intercept (where the line goes through the Y-axis).

In the case of our scatter plot for our student data, we already have our values for *x* (*StudyHours*) and *y* (*Grade*), so we just need to calculate the intercept and slope of the straight line that lies closest to those points. Then we can form a linear equation that calculates a new *y* value on that line for each of our *x* (*StudyHours*) values - to avoid confusion, we'll call this new *y* value *f(x)* (because it's the output from a linear equation *f*unction based on *x*). The difference between the original *y* (*Grade*) value and the *f(x)* value is the *error* between our regression line and the actual *Grade* achieved by the student. Our goal is to calculate the slope and intercept for a line with the lowest overall error.Specifically, we define the overall error by taking the error for each point, squaring it, and adding all the squared errors together. The line of best fit is the line that gives us the lowest value for the sum of the squared errors - hence the name *least squares regression*.

you don't need to code the regression calculation yourself - the SciPy package includes a stats class that provides a linregress method to do the hard work for you. This returns (among other things) the coefficients you need for the slope equation - slope (*m*) and intercept (*b*) based on a given pair of variable samples you want to compare.