Chapter 4: Pandas

- Pandas is a Python library used for working with **datasets**
- It has functions for **analyzing, cleaning, exploring, and manipulating data**
- "Pandas" has a reference to "Python Data Analysis"

1. Installation and Import of Pandas

```
pip install pandas
```

```
import pandas
```

2. Pandas Series

A Pandas Series is like a **column** in a table.

Create a simple Pandas Series from an ndarray:

```
import numpy as np
import pandas as pd
a = np.array([3, 8, 1, 10])
myvar = pd.Series(a)
print(myvar)
print(myvar[0])  # return 3
```

The values are labeled with their index number starting by 0

a) Create index

With the `index` argument, you can name your own labels.

```
import numpy as np
import pandas as pd
a = np.array([3, 8, 1])
myvar = pd.Series(a, index = ["x", "y", "z"])
print(myvar)
print(myvar["y"])  # return 8
```

3. DataFrames

Datasets in Pandas are usually multi-dimensional tables, called DataFrames.

Series is like a column, a DataFrame is the whole table.

Create a DataFrame from 2-D numpy array:

```
import numpy as np
import pandas as pd
my_array = np.array([[11, 22, 33], [44, 55, 66]])
df = pd.DataFrame(my_array, columns=['Column_A', 'Column_B', 'Column_C'])
print(df)
```

a) Locate Row: the `loc` attribute returns one or more specified row(s)

- Return row 0: `print(df.loc[0])`

  Note: This example returns a Pandas **Series**.

- Return row 0 and 1: `print(df.loc[[0, 1]])`

  Note: When using `[]` inside, the result is a Pandas **DataFrame**.

- Return a specific data: `print(df.loc[0, 'Column_A'])`

  Note: the result is a **value**.

b) Named Indexes: With `index`, we can name our own indexes**.**

```
import numpy as np
import pandas as pd
arr = np.array([[100, 80, 50, 70], [40, 50, 60, 70], [70, 88, 91, 95]])
df = pd.DataFrame(arr, columns=['Attend', 'Quiz', 'Midterm', 'Final'],
                       index=['Samir', 'Mounir', 'Amir'])
print(df)
print(df.loc['Samir'])
print(df.loc[['Samir','Amir']])
print(df.loc[['Samir','Amir'],'Attendance'])
```

4. Read CSV Files

A simple way to store big datasets is to use CSV files that contains plain text.

Load the CSV into a DataFrame: use `to_string()` to print the entire DataFrame

```
import pandas as pd
df = pd.read_csv('data/cars_data.csv')
print(df.to_string())
print(df) # print first and last 5 in big data
```

5. Analyzing DataFrames

a) Viewing the Data: getting a quick overview of the DataFrame by using `head()` and `tail()` methods.

Get a quick overview by printing the first 3 rows of the DataFrame:

```
import pandas as pd
df = pd.read_csv('data/cars_data.csv')
print(df.head(3))
print(df.loc[[0,35],'Model'])
```

Print the First and Last 5 rows of the DataFrame:

```
print(df.head())
print(df.tail())
```

b) Read Specific columns

```
df = pd.read_csv('data/cars_data.csv', usecols=['Car', 'Model'])
```

c) Info about the Data (Null Values)

Print information about the data:

```
print(df.info())
```

- The `info()` method also tells us how many Non-Null values there are present in each column
- Empty values, or Null values, can be bad when analyzing data, and you should consider removing (cleaning) rows with empty values.

d) Data Cleaning

Data cleaning means fixing bad data in your dataset. Bad data could be:

- Empty cells
- Data in wrong format
- Wrong data
- Duplicates

- **Dataset Example**

	Duration	Date	Pulse	Maxpulse	Calories
0	60	'2020/12/01'	110	130	409.1
1	60	'2020/12/02'	117	145	479.0
2	60	'2020/12/03'	103	135	340.0
3	45	'2020/12/04'	109	175	282.4
4	45	'2020/12/05'	117	148	406.0
5	60	'2020/12/06'	102	127	300.0
6	60	'2020/12/07'	110	136	374.0
7	450	'2020/12/08'	104	134	253.3
8	30	'2020/12/09'	109	133	195.1
9	60	'2020/12/10'	98	124	269.0
10	60	'2020/12/11'	103	147	329.3
11	60	'2020/12/12'	100	120	250.7
12	60	'2020/12/12'	100	120	250.7
13	60	'2020/12/13'	106	128	345.3
14	60	'2020/12/14'	104	132	379.3
15	60	'2020/12/15'	98	123	275.0
16	60	'2020/12/16'	98	120	215.2
17	60	'2020/12/17'	100	120	300.0
18	45	'2020/12/18'	90	112	NaN
19	60	'2020/12/19'	103	123	323.0
20	45	'2020/12/20'	97	125	243.0
21	60	'2020/12/21'	108	131	364.2
22	45	NaN	100	119	282.0
23	60	'2020/12/23'	130	101	300.0
24	45	'2020/12/24'	105	132	246.0
25	60	'2020/12/25'	102	126	334.5
26	60	2020/12/26	100	120	250.0
27	60	'2020/12/27'	92	118	241.0
28	60	'2020/12/28'	103	132	NaN
29	60	'2020/12/29'	100	132	280.0
30	60	'2020/12/30'	102	129	380.3
31	60	'2020/12/31'	92	115	243.0

The dataset contains some empty cells, some wrong format and wrong data and duplicates.

1) Empty cells can potentially give you a wrong result when you analyze data.

- **Remove rows:** One way to deal with empty cells is to remove rows that contain empty cells. This is usually working if your dataset is very big, and it will not impact on the result.

Return a new DataFrame with no empty cells:

```
import pandas as pd
df = pd.read_csv('data/data.csv')
new_df = df.dropna()
print(new_df.to_string())
new_df.to_csv('data/newdata.csv')  #save to new csv file
```

Note: `dropna()` method returns a *new* DataFrame, if you want to change the original DataFrame, use the `inplace = True` argument:

```
import pandas as pd
df = pd.read_csv('data/data.csv')
df.dropna(inplace = True)
print(df.to_string())
```

Note: `dropna(inplace = True)` will remove all rows containing NULL values from the original DataFrame.

- **Replace Empty Values**: Another way of dealing with empty cells is to insert a *new* value instead using `fillna()` method

Replace NULL values with the number 130:

```
import pandas as pd
df = pd.read_csv('data/data.csv')
df.fillna(130, inplace = True)
```

- **Replace Only For Specified Columns**:

```
import pandas as pd
df = pd.read_csv('data.csv')
df["Calories"].fillna(130, inplace = True)
df.to_csv('data/data_filled.csv', index=False)
```

- **Replace Using Mean (Average):** Pandas uses the `mean()` methods to calculate the respective values for a specified column:

Calculate the MEAN, and replace any empty values with it:

```
import pandas as pd

df = pd.read_csv('data.csv')

x = df["Calories"].mean()

df["Calories"].fillna(x, inplace = True)
```

2) Data of Wrong Format

Data of wrong format can make it difficult to analyze data, to fix it, you have two options: remove the rows, or convert data in columns into the same format.

a) Convert into a Correct Format: We will convert all cells in the 'Date' column into dates. Using `to_datetime()` method for this:

Convert to date:

```
import pandas as pd

df = pd.read_csv('data.csv')

df['Date'] = pd.to_datetime(df['Date'])

print(df.to_string())
```

Note: The empty date got a NaT (Not a Time) value (empty value).

To deal with empty values is simply removing the entire row.

Remove rows with a NULL value in the "Date" column:

```
df.dropna(subset=['Date'], inplace = True)
```

3) Wrong Data

The duration in the example above (in row 7) is 450 minutes, but for all other duration is between 30 and 60 (person didn't work out for 450 minutes).

a) Replacing Values: Set "Duration" = 45 in row 7 instead of 450:

```
df.loc[7, 'Duration'] = 45
```

Note: Replacing wrong data one-by-one might can be done in small datasets, but not for big data sets.

To replace wrong data for larger datasets we can create some rules (e.g. set some boundaries for legal values, and replace any values that are outside).

In "Duration" column. If any value is higher than 120, change it to 120:

```
for x in df.index:
  if df.loc[x, "Duration"] > 120:
     df.loc[x, "Duration"] = 120
```

b) Removing Rows: Another way is to remove the rows that contains wrong data.

Delete rows where "Duration" is higher than 120:

```
for x in df.index:
  if df.loc[x, "Duration"] > 120:
     df.drop(x, inplace = True)
```

4) Removing Duplicates

a) Discovering Duplicates: rows that is registered more than one time.

To discover duplicates, the `duplicated()` method will returns a Boolean values for each row:

Returns `True` for every row that is a duplicate, otherwise `False`:

```
print(df.duplicated())
```

b) Removing Duplicates: To remove duplicates, use `drop_duplicates()`

Remove all duplicates:

```
df.drop_duplicates(inplace = True)
```

5) Data Correlations

a) **Finding Relationships:** The `corr()` method calculates the relationship between each column in your data set.

```
df.corr()
```

Note: The `corr()` method ignores "not numeric" columns. The Result is a table with a lot of numbers (varies from -1 to 1) that represents how well the relationship is between two columns.

	Duration	Pulse	Maxpulse	Calories
Duration	1.000000	-0.155408	0.009403	0.922721
Pulse	-0.155408	1.000000	0.786535	0.025120
Maxpulse	0.009403	0.786535	1.000000	0.203814
Calories	0.922721	0.025120	0.203814	1.000000

Perfect Correlation:

- 1 means that there is a 1 to 1 relationship (a perfect correlation), and for this data set, each time a value went up in the first column, the other one went up as well.
- "Duration" and "Duration" got the number 1.000000, which makes sense, each column always has a perfect relationship with itself.

Good Correlation:

- 0.9 is also a good relationship, and if you increase one value, the other will most likely increase as well.
- -0.9 would be just as good relationship as 0.9, but if you increase one value, the other will go down.
- "Duration" and "Calories" got a 0.922721 correlation, which is a very good correlation, and we can predict that the longer you work out, the more calories you burn, and the other way around: if you burned a lot of calories, you probably had a long work out.

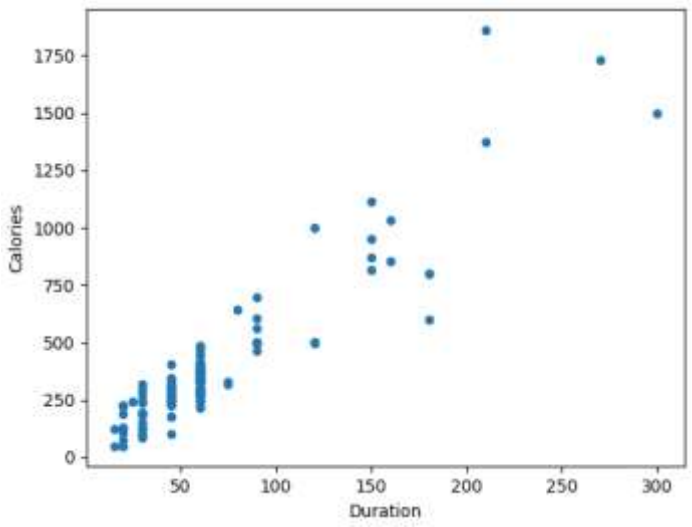
Bad Correlation:

- 0.2 means NOT a good relationship, meaning that if one value goes up does not mean that the other will.
- "Duration" and "Maxpulse" got a 0.009403 correlation, which is a very bad correlation, meaning that we cannot predict the max pulse by just looking at the duration of the work out, and vice versa.

What is a good correlation? At least 0.6 (or -0.6) to call it a good correlation.

6) Plotting (Scatter Plot)

```
import pandas as pd
import matplotlib.pyplot as plt
df = pd.read_csv('data.csv')
df.plot(kind = 'scatter',
x = 'Duration', y = 'Calories')
plt.show()
```



Remember: By looking at the scatterplot, we will see that higher duration means more calories burned.

A scatterplot where there are no relationship between the columns:

```
import pandas as pd
import matplotlib.pyplot as plt
df = pd.read_csv('data.csv')
df.plot(kind = 'scatter', x = 'Duration', y = 'Maxpulse')
plt.show()
```