



Pandas is a Python library. Pandas is used to analyze data.

[LinkedIn](#) [kaggle](#) [YouTube](#) [GitHub](#)





Learning by Reading

- We have created 14 Chapter's pages for you to learn more about Pandas.
- Starting with a basic introduction and ends up with cleaning and plotting data:

Pandas Course Content

Basic

Introduction

Getting Started

Pandas Series

DataFrames

Read CSV

Read JSON

Analyze Data

Cleaning Data

Clean Data

Clean Empty Cells

Clean Wrong Format

Clean Wrong Data

Remove Duplicates

Advanced

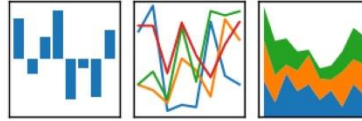
Correlations

Plotting



pandas

$$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$$



Python Pandas Introduction

What is Pandas?

- Pandas is a Python library used for working with data sets.
- It has functions for analyzing, cleaning, exploring, and manipulating data.
- The name "Pandas" has a reference to both "Panel Data", and "Python Data Analysis" and was created by Wes McKinney in 2008.

Why Use Pandas?

- Pandas allows us to analyze big data and make conclusions based on statistical theories.
- Pandas can clean messy data sets, and make them readable and relevant.
- Relevant data is very important in data science.



Data Science: is a branch of computer science where we study how to store, use and analyze data for deriving information from it.

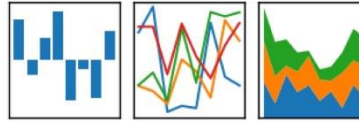
What Can Pandas Do?

- Pandas gives you answers about the data. Like:
- Is there a correlation between two or more columns?
- What is average value?
- Max value?
- Min value?
- Pandas are also able to delete rows that are not relevant, or contains wrong values, like empty or NULL values. This is called *cleaning* the data.



pandas

$$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$$



Pandas Getting Started

Installation of Pandas

- If you have [Python](#) and [PIP](#) already installed on a system, then installation of Pandas is very easy.
- Install it using this command:

```
C:\Users\Your Name>pip install pandas
```



If this command fails, then use a python distribution that already has Pandas installed like, Anaconda, Spyder etc.

Import Pandas

Once Pandas is installed, import it in your applications by adding the **import** keyword:

```
import pandas
```

Now **Pandas** is imported and ready to use.

Pandas as pd

Pandas is usually imported under the **pd** alias.

alias: In Python alias are an alternate name for referring to the same thing.

Create an alias with the **as** keyword while importing:

```
import pandas as pd
```



Now the Pandas package can be referred to as **pd** instead of **pandas**.

Checking Pandas Version

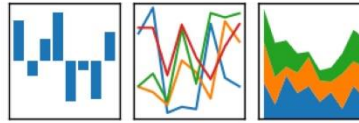
The version string is stored under `__version__` attribute.

```
import pandas as pd  
  
print(pd.__version__)
```



pandas

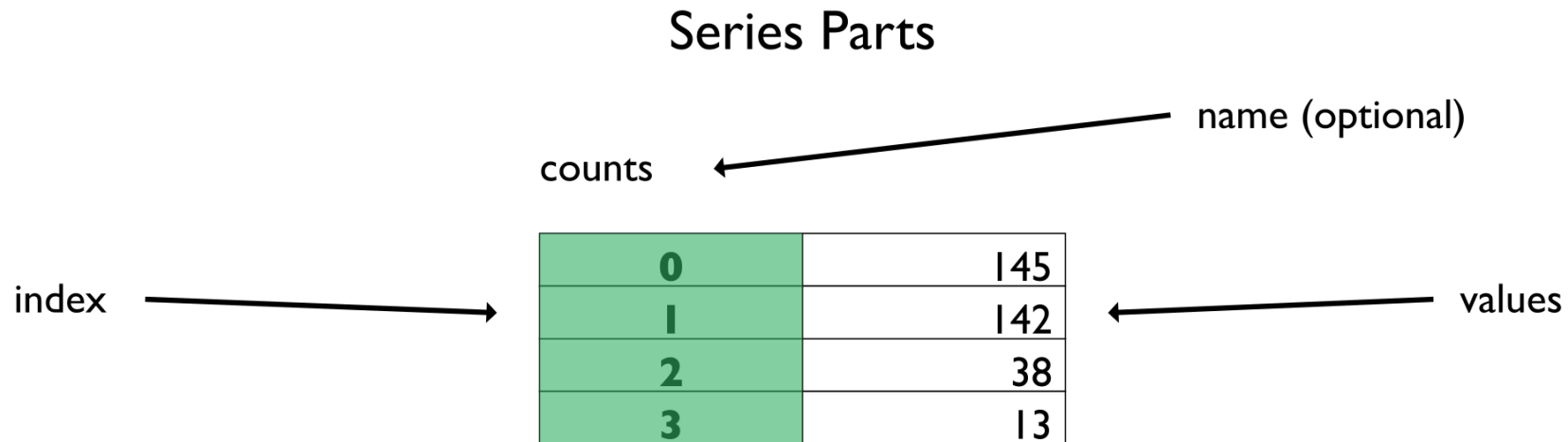
$$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$$



Pandas Series

What is a Series?

- A Pandas Series is like a column in a table.
- It is a one-dimensional array holding data of any type.





Example

Create a simple Pandas Series from a list:

```
import pandas as pd

a = [1, 7, 2]

myvar = pd.Series(a)

print(myvar)
```

Labels

If nothing else is specified, **the values are labeled with their index number**. First value has index 0, second value has index 1 etc.

This label can be used to access a specified value.



Example

Return the first value of the Series:

```
print(myvar[0])
```

```
import pandas as pd
```

```
a = [1, 7, 2]
```

```
myvar = pd.Series(a)
```

```
print(myvar)
```


Create Labels

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



Example

Create your own labels:

```
import pandas as pd

a = [1, 7, 2]

myvar = pd.Series(a, index = ["x", "y", "z"])

print(myvar)
```

When you have created labels, you can access an item by referring to the label.

Example

Return the value of "y":

```
print(myvar["y"])
```

```
import pandas as pd

a = [1, 7, 2]

myvar = pd.Series(a, index = ["x", "y", "z"])

print(myvar)
```

Key/Value Objects as Series

- You can also use a key/value object, like a dictionary, when creating a Series.



Example

Create a simple Pandas Series from a dictionary:

```
import pandas as pd

calories = {"day1": 420, "day2": 380, "day3": 390}

myvar = pd.Series(calories)

print(myvar)
```

Note: The keys of the dictionary become the labels.

To select only some of the items in the dictionary, use the **index** argument and specify only the items you want to include in the Series.

Example

Create a Series using only data from "**day1**" and "**day2**":

```
import pandas as pd

calories = {"day1": 420, "day2": 380, "day3": 390}

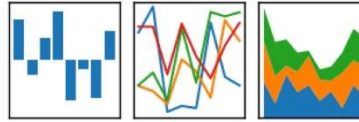
myvar = pd.Series(calories, index = ["day1", "day2"])

print(myvar)
```



pandas

$$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$$



Pandas DataFrames

What is a DataFrame?

- A Pandas DataFrame is a 2 dimensional data structure, like a 2 dimensional array, or a table with rows and columns.

Series 1			Series 2			Series 3			DataFrame			
Mango			Apple			Banana			Mango	Apple	Banana	
0	4		0	5		0	2		0	4	5	2
1	5		1	4		1	3		1	5	4	3
2	6		2	3		2	5		2	6	3	5
3	3		3	0		3	2		3	3	0	2
4	1		4	2		4	7		4	1	2	7



Example

Create a simple Pandas DataFrame:

```
import pandas as pd

data = {
    "calories": [420, 380, 390],
    "duration": [50, 40, 45]
}

#load data into a DataFrame object:
df = pd.DataFrame(data)

print(df)
```

Result

	calories	duration
0	420	50
1	380	40
2	390	45

Locate Row

- As you can see from the result above, the DataFrame is like a table with rows and columns.
- Pandas use the `loc` attribute to return one or more specified row(s)



Example

Return row 0:

```
#refer to the row index:  
print(df.loc[0])
```

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

Result

```
calories    420  
duration     50  
Name: 0, dtype: int64
```




Example

Return row 0 and 1:

```
#use a list of indexes:  
print(df.loc[[0, 1]])
```

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

Result

	calories	duration
0	420	50
1	380	40

Named Indexes

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



Example

Add a list of names to give each row a name:

```
import pandas as pd

data = {
    "calories": [420, 380, 390],
    "duration": [50, 40, 45]
}

df = pd.DataFrame(data, index = ["day1", "day2", "day3"])

print(df)
```

Result

	calories	duration
day1	420	50
day2	380	40
day3	390	45

Locate Named Indexes

Use the named index in the `loc` attribute to return the specified row(s).



Example

Return "day2":

```
#refer to the named index:  
print(df.loc["day2"])
```

Result

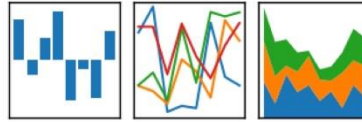
```
calories    380  
duration     40  
Name: day2, dtype: int64
```

	calories	duration
day1	420	50
day2	380	40
day3	390	45



pandas

$$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$$

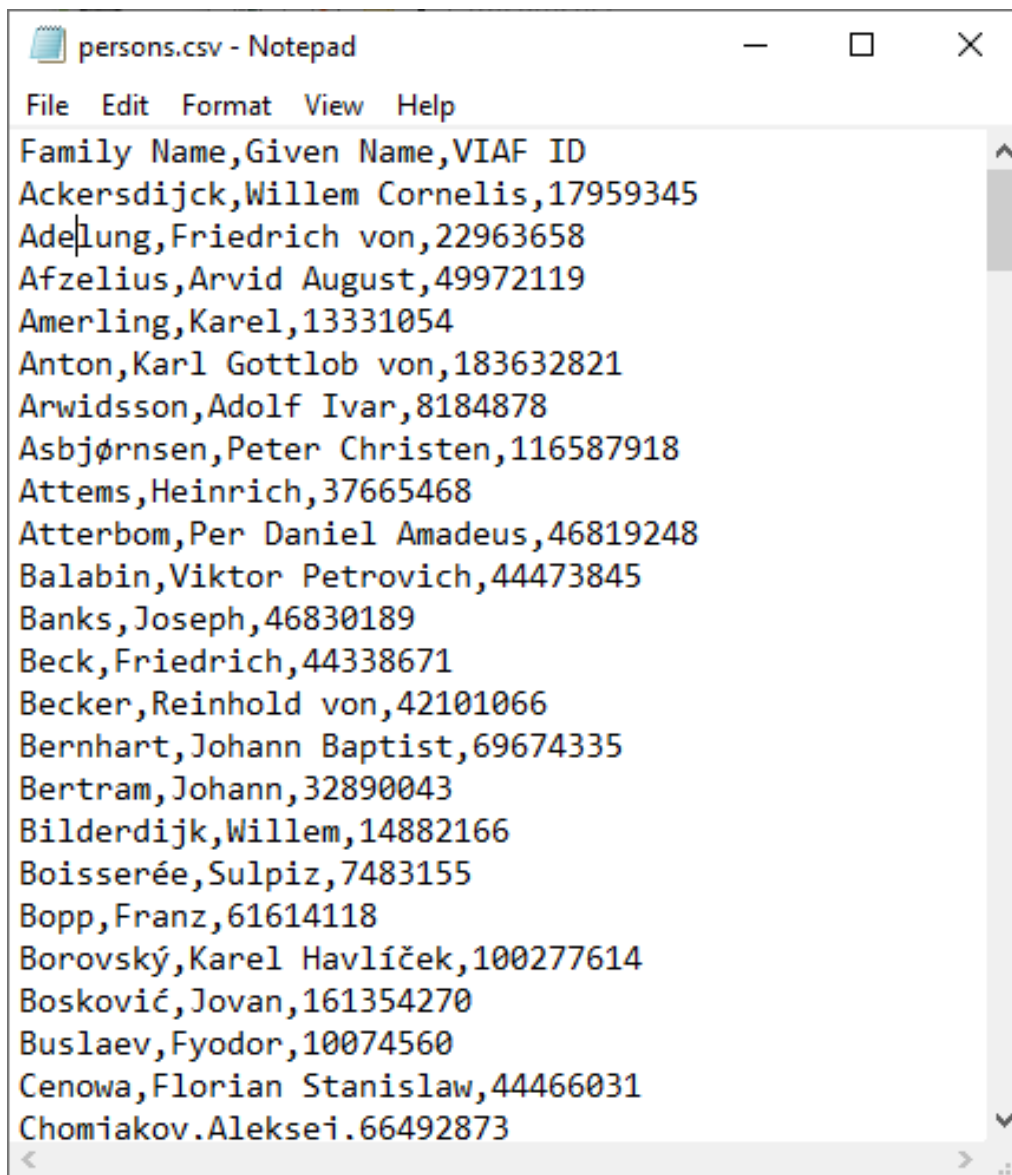


Pandas Read CSV

Read CSV Files

- A simple way to store big data sets is to use CSV files (comma separated files).
- CSV files contains plain text and is a well know format that can be read by everyone including Pandas.
- In our examples we will be using a CSV file called 'data.csv'.
- [Download data.csv](#). or [Open data.csv](#)

CSV Data Format



```
persons.csv - Notepad
File Edit Format View Help
Family Name,Given Name,VIAF ID
Ackersdijck,Willem Cornelis,17959345
Adelung,Friedrich von,22963658
Afzelius,Arvid August,49972119
Amerling,Karel,13331054
Anton,Karl Gottlob von,183632821
Arwidsson,Adolf Ivar,8184878
Asbjørnsen,Peter Christen,116587918
Attems,Heinrich,37665468
Atterbom,Per Daniel Amadeus,46819248
Balabin,Viktor Petrovich,44473845
Banks,Joseph,46830189
Beck,Friedrich,44338671
Becker,Reinhold von,42101066
Bernhart,Johann Baptist,69674335
Bertram,Johann,32890043
Bilderdijk,Willem,14882166
Boisserée,Sulpiz,7483155
Bopp,Franz,61614118
Borovský,Karel Havlíček,100277614
Bosković,Jovan,161354270
Buslaev,Fyodor,10074560
Cenowa,Florian Stanislaw,44466031
Chomiakov,Aleksei,66492873
```



Example

Load the CSV into a DataFrame:

```
import pandas as pd

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

print(df.to_string())
```

Tip: use `to_string()` to print the entire DataFrame.

If you have a large DataFrame with many rows, Pandas will only return the first 5 rows, and the last 5 rows:



Example

Print the DataFrame without the `to_string()` method:

```
import pandas as pd

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

print(df)
```


max_rows

The number of rows returned is defined in Pandas option settings. You can check your system's maximum rows with the `pd.options.display.max_rows` statement.

 **Example**
Check the number of maximum returned rows:

```
✓ [1] 1 import pandas as pd  
0s    2  
      3 print(pd.options.display.max_rows)
```

→ 60

- In my system the number is 60, which means that if the DataFrame contains more than 60 rows, the `print(df)` statement will return only the headers and the first and last 5 rows.
- You can change the maximum rows number with the same statement.

```
import pandas as pd

pd.options.display.max_rows = 9999

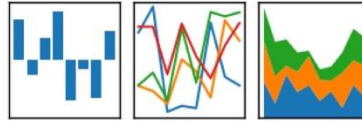
df = pd.read_csv('data.csv')

print(df)
```



pandas

$$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$$



Pandas Read JSON

Read JSON

- Big data sets are often stored, or extracted as JSON.
- JSON is plain text, but has the format of an object, and is well known in the world of programming, including Pandas.
- In our examples we will be using a JSON file called 'data.json'.
- [Open data.json](#).



Example

Load the JSON file into a DataFrame:

```
import pandas as pd

df = pd.read_json('data.json')

print(df.to_string())
```

Tip: use `to_string()` to print the entire DataFrame.

Dictionary as JSON

JSON = Python Dictionary

- JSON objects have the same format as Python dictionaries.
- If your JSON code is not in a file, but in a Python Dictionary, you can load it into a DataFrame directly:



Example

Load a Python Dictionary into a DataFrame:

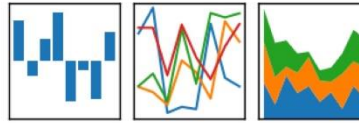
```
✓ [3] 1 import pandas as pd
0s    2
      3 data = {
      4     "Duration": {"0": 60, "1": 60, "2": 60, "3": 45, "4": 45, "5": 60},
      5     "Pulse": {"0": 110, "1": 117, "2": 103, "3": 109, "4": 117, "5": 102},
      6     "Maxpulse": {"0": 130, "1": 145, "2": 135, "3": 175, "4": 148, "5": 127},
      7     "Calories": {"0": 409, "1": 479, "2": 340, "3": 282, "4": 406, "5": 300},
      8 }
      9
     10 df = pd.DataFrame(data)
     11
     12 # Display only the first few rows using head()
     13 print(df.head())
     14
```

	Duration	Pulse	Maxpulse	Calories
0	60	110	130	409
1	60	117	145	479
2	60	103	135	340
3	45	109	175	282
4	45	117	148	406



pandas

$$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$$



Analyzing data using pandas

Analyzing data using pandas typically involves a variety of operations to gain insights into the structure and content of the dataset. Here are some common tasks you might perform using pandas:

Loading Data:

Read data from different file formats (CSV, Excel, SQL, etc.)

using **pd.read_csv()**, **pd.read_excel()**, **pd.read_sql()**, etc.

```
import pandas as pd
```

```
# Example: Reading data from a CSV file
```

```
df = pd.read_csv('your_data.csv')
```

Exploring Data:

- Use `df.head()` and `df.tail()` to quickly view the first and last few rows of the dataset.
- Check basic statistics with `df.describe()` for numerical columns.
- Get information about the DataFrame using `df.info()`.

```
# Display basic statistics
```

```
print(df.describe())
```

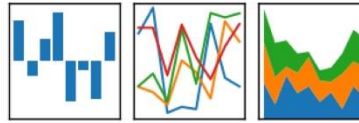
```
# Display information about the DataFrame
```

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



pandas

$$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$$



Selecting Data:

In pandas, you can select data by accessing columns and filter rows based on conditions. Here are examples for both operations:

Accessing Columns:

You can access columns in a DataFrame using either `df['column_name']` or `df.column_name`. Here's an example:

```
import pandas as pd

# Creating a sample DataFrame
data = {'Name': ['Alice', 'Bob', 'Charlie'],
        'Age': [25, 30, 35],
        'City': ['New York', 'San Francisco', 'Los Angeles']}
df = pd.DataFrame(data)

# Accessing columns using df['column_name']
name_column = df['Name']
age_column = df['Age']

# Or, you can use df.column_name
city_column = df.City

# Display the selected columns
print(name_column)
print(age_column)
print(city_column)
```

Output

```
0      Alice
1      Bob
2    Charlie
Name: Name, dtype: object
0      25
1      30
2      35
Name: Age, dtype: int64
0      New York
1    San Francisco
2    Los Angeles
Name: City, dtype: object
```

Filtering Rows Based on Conditions:

You can filter rows based on conditions using boolean indexing. Here's an example:

```
1
2 import pandas as pd
3
4 # Creating a sample DataFrame
5 data = {'Name': ['Alice', 'Bob', 'Charlie'],
6         'Age': [25, 30, 35],
7         'City': ['New York', 'San Francisco', 'Los Angeles']}
8 df = pd.DataFrame(data)
9
10 # Filtering rows based on conditions
11 # Selecting rows where Age is greater than 30
12 filtered_df = df[df['Age'] > 30]
13
14 # Display the filtered DataFrame
15 print("Filtered DataFrame:")
16 print(filtered_df)
17
```

Filtered DataFrame:

	Name	Age	City
2	Charlie	35	Los Angeles

Sorting Data:

Sort the DataFrame based on one or more columns.

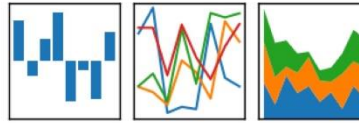
```
# Sort by a single column
df_sorted = df.sort_values(by='Pulse')

# Sort by multiple columns
df_sorted = df.sort_values(by=['Pulse', 'Duration'])
```



pandas

$$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$$



Grouping Data

Grouping data in pandas is a powerful operation that allows you to split a DataFrame into groups based on one or more columns and then perform various operations on these groups. Here's an example to demonstrate how to group data using the groupby function:


```
import pandas as pd

# Creating a sample DataFrame
data = {'Category': ['A', 'B', 'A', 'B', 'A', 'B'],
        'Value': [10, 15, 20, 25, 30, 35],
        'Quantity': [2, 3, 1, 4, 2, 3]}
df = pd.DataFrame(data)

# Grouping by the 'Category' column
grouped_df = df.groupby('Category')

# Performing aggregate operations on the groups
# For example, calculating the sum for each group
sum_by_category = grouped_df.sum()

# Displaying the original DataFrame and the grouped result
print("Original DataFrame:")
print(df)

print("\nGrouped DataFrame:")
print(sum_by_category)
```

In this example, the DataFrame is grouped based on the 'Category' column using `groupby('Category')`. Then, an aggregate operation (sum in this case) is performed on each group using the `sum()` function.

Output

Original DataFrame:

	Category	Value	Quantity
0	A	10	2
1	B	15	3
2	A	20	1
3	B	25	4
4	A	30	2
5	B	35	3

Grouped DataFrame:

	Value	Quantity
Category		
A	60	5
B	75	10

- In the result, the data is aggregated for each category, providing the sum of 'Value' and 'Quantity' for each group. You can perform various other aggregation functions such as mean, median, min, max, etc., based on your requirements.
- You can also group by multiple columns by passing a list of column names to the groupby function. For example, `df.groupby(['Category', 'AnotherColumn'])`. Adjust the code according to your specific DataFrame and grouping criteria.

Aggregating Data:

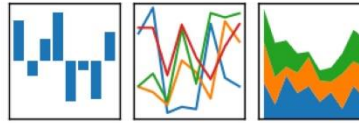
Use aggregation functions on grouped data.

```
# Calculate the mean, sum, and count for each group  
aggregated_data = grouped_data.agg({'Pulse': 'mean', 'Duration': 'sum',  
    'Calories': 'count'})
```



pandas

$$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$$



Pandas - Cleaning Data

Data Cleaning

Data cleaning means fixing bad data in your data set.

Bad data could be:

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

In this chapter you will learn how to deal with all of them.

Our Data Set

In the next chapters
we will use this data set:

The data set contains some empty cells ("Date" in row 22, and "Calories" in row 18 and 28).

The data set contains wrong format ("Date" in row 26).

The data set contains wrong data ("Duration" in row 7).

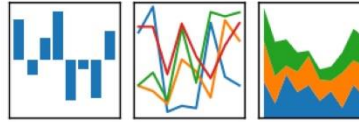
The data set contains duplicates (row 11 and 12).

	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



pandas

$$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$$



Pandas - Cleaning Empty Cells

Empty Cells

- 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 OK, since data sets can be very big, and removing a few rows will not have a big impact on the result.



Example

Return a new Data Frame with no empty cells:

```
import pandas as pd

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

new_df = df.dropna()

print(new_df.to_string())
```

Note: By default, the `dropna()` method returns a *new* DataFrame, and will not change the original.

If you want to change the original DataFrame, use the `inplace = True` argument:

Example

Remove all rows with NULL values:

```
import pandas as pd

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

df.dropna(inplace = True)

print(df.to_string())
```

Note: Now, the `dropna(inplace = True)` will NOT return a new DataFrame, but it 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.
- This way you do not have to delete entire rows just because of some empty cells.
- The `fillna()` method allows us to replace empty cells with a value:



Example

Replace NULL values with the number 130:

```
import pandas as pd

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

df.fillna(130, inplace = True)
```

Replace Only For Specified Columns

- The example above replaces all empty cells in the whole Data Frame.
- To only replace empty values for one column, specify the *column name* for the DataFrame:



Example

Replace NULL values in the "Calories" columns with the number 130:

```
import pandas as pd

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

df["Calories"].fillna(130, inplace = True)
```

Replace Using Mean, Median, or Mode

- A common way to replace empty cells, is to calculate the mean, median or mode value of the column.
- Pandas uses the `mean()` `median()` and `mode()` methods to calculate the respective values for a specified column:



Example

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)
```

Mean = the average value (the sum of all values divided by number of values).



Example

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

Median = the value in the middle, after you have sorted all values ascending.

```
import pandas as pd

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

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

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



Example

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

Mode = the value that appears most frequently.

```
import pandas as pd

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

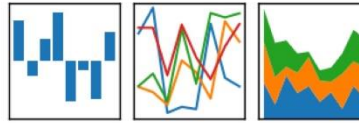
x = df["Calories"].mode()[0]

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



pandas

$$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$$



Pandas - Cleaning Data of Wrong Format

Data of Wrong Format

- Cells with data of wrong format can make it difficult, or even impossible, to analyze data.
- To fix it, you have two options: remove the rows, or convert all cells in the columns into the same format.

Convert Into a Correct Format

In our Data Frame, we have two cells with the wrong format. Check out row 22 and 26, the 'Date' column should be a string that represents a date:

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	20201226	100	120	250.0
27	60	'2020/12/27'	92	118	241.0

Convert Into a Correct Format

- Let's try to convert all cells in the 'Date' column into dates.
- Pandas has a `to_datetime()` method for this:



Example

Convert to date:

```
import pandas as pd

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

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

print(df.to_string())
```

Result:

21	60	'2020/12/21'	108	131	364.2
22	45	NaT	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

As you can see from the result, the date in row 26 was fixed, but the empty date in row 22 got a NaT (Not a Time) value, in other words an empty value. One way to deal with empty values is simply removing the entire row.

Removing Rows

The result from the converting in the example above gave us a NaT value, which can be handled as a NULL value, and we can remove the row by using the `dropna()` method.

Example

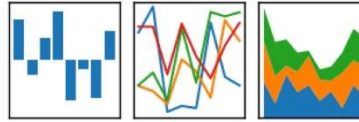
Remove rows with a NULL value in the "Date" column:

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



pandas

$$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$$



Pandas - Fixing Wrong Data

Wrong Data

- "Wrong data" does not have to be "empty cells" or "wrong format", it can just be wrong, like if someone registered "199" instead of "1.99".
- Sometimes you can spot wrong data by looking at the data set, because you have an expectation of what it should be.
- If you take a look at our data set, you can see that in row 7, the duration is 450, but for all the other rows the duration is between 30 and 60.
- It doesn't have to be wrong, but taking in consideration that this is the data set of someone's workout sessions, we conclude with the fact that this person did not work out in 450 minutes.

Wrong Data

	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

How can we fix wrong values, like the one for "Duration" in row 7?

Replacing Values

- One way to fix wrong values is to replace them with something else.
- In our example, it is most likely a typo, and the value should be "45" instead of "450", and we could just insert "45" in row 7:

Example

Set "Duration" = 45 in row 7:

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


- For small data sets you might be able to replace the wrong data one by one, but not for big data sets.
- To replace wrong data for larger data sets you can create some rules, e.g. **set some boundaries for legal values, and replace any values that are outside of the boundaries.**

Example

Loop through all values in the "Duration" column.
If the value is higher than 120, set it to 120:

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

Removing Rows

- Another way of handling wrong data is to remove the rows that contains wrong data.
- This way you do not have to find out what to replace them with, and there is a good chance you do not need them to do your analyses.



Example

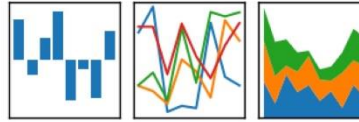
Delete rows where "Duration" is higher than 120:

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



pandas

$$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$$



Pandas - Removing Duplicates

Discovering Duplicates

Duplicate rows are rows that have been registered more than one time.

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

- By taking a look at our test data set, we can assume that row 11 and 12 are duplicates.
- To discover duplicates, we can use the `uplicated()` method.
- The `uplicated()` method returns a Boolean values for each row:

Example

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

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

Removing Duplicates

To remove duplicates, use the `drop_duplicates()` method.

Example

Remove all duplicates:

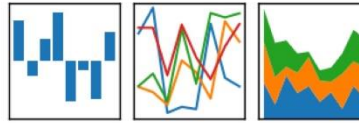
```
df.drop_duplicates(inplace = True)
```

Remember: The `(inplace = True)` will make sure that the method does NOT return a *new* DataFrame, but it will remove all duplicates from the *original* DataFrame.



pandas

$$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$$



Pandas - Data Correlations

Finding Relationships

- A great aspect of the Pandas module is the `corr()` method.
- The `corr()` method calculates the relationship between each column in your data set.
- The examples in this page uses a CSV file called: 'data.csv'.
- [Download data.csv](#). or [Open data.csv](#)



Example

Show the relationship between the columns:

```
df.corr()
```

```
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
```

Note: The `corr()` method ignores "not numeric" columns.

Result Explained

- The Result of the `corr()` method is a table with a lot of numbers that represents how well the relationship is between two columns.
- The number varies from -1 to 1.
- 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.
- 0.9 is also a good relationship, and if you increase one value, the other will probably increase as well.
- -0.9 would be just as good relationship as 0.9, but if you increase one value, the other will probably go down.
- 0.2 means NOT a good relationship, meaning that if one value goes up does not mean that the other will.

What is a good correlation? It depends on the use, but I think it is safe to say you have to have at least 0.6 (or -0.6) to call it a good correlation.

Perfect Correlation:

We can see that "Duration" and "Duration" got the number **1.000000**, which makes sense, each column always has a perfect relationship with itself.

Good Correlation:

"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.

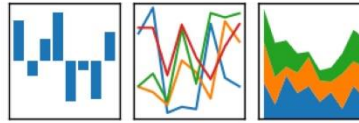
Bad Correlation:

"Duration" and "Maxpulse" got a 0.009403 correlation, which is a very bad correlation, meaning that we can not predict the max pulse by just looking at the duration of the work out, and vice versa.

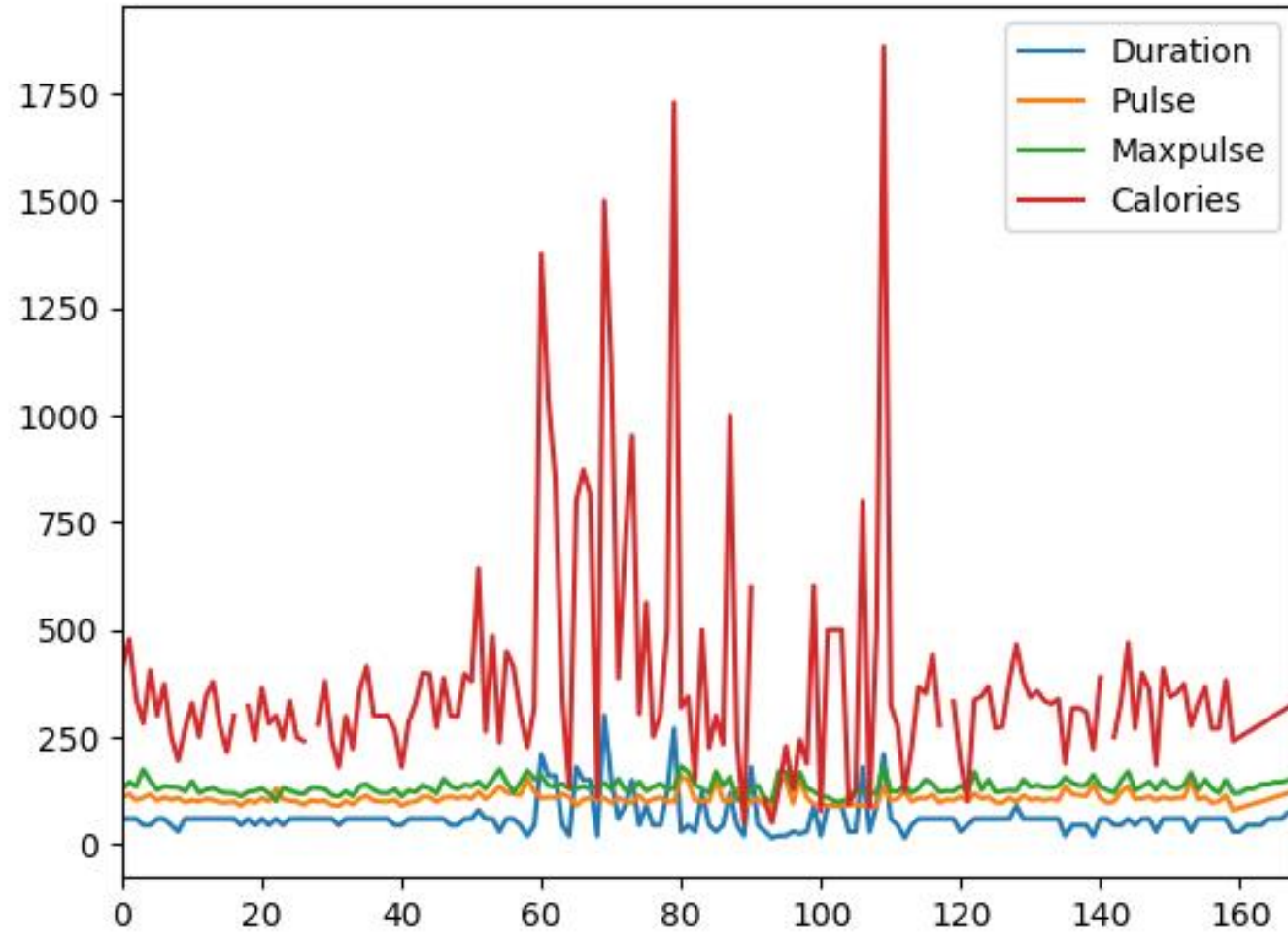


pandas

$$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$$



Pandas - Plotting



Plotting

Pandas uses the `plot()` method to create diagrams.

We can use Pyplot, a submodule of the Matplotlib library to visualize the diagram on the screen.



Example

Import pyplot from Matplotlib and visualize our DataFrame:

The examples in this page uses a CSV file called: 'data.csv'.

```
import pandas as pd
import matplotlib.pyplot as plt

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

df.plot()

plt.show()
```

Scatter Plot

- Specify that you want a scatter plot with the `kind` argument:
- `kind = 'scatter'`
- A scatter plot needs an x- and a y-axis.
- In the example below we will use "Duration" for the x-axis and "Calories" for the y-axis.
- Include the x and y arguments like this:
- `x = 'Duration', y = 'Calories'`

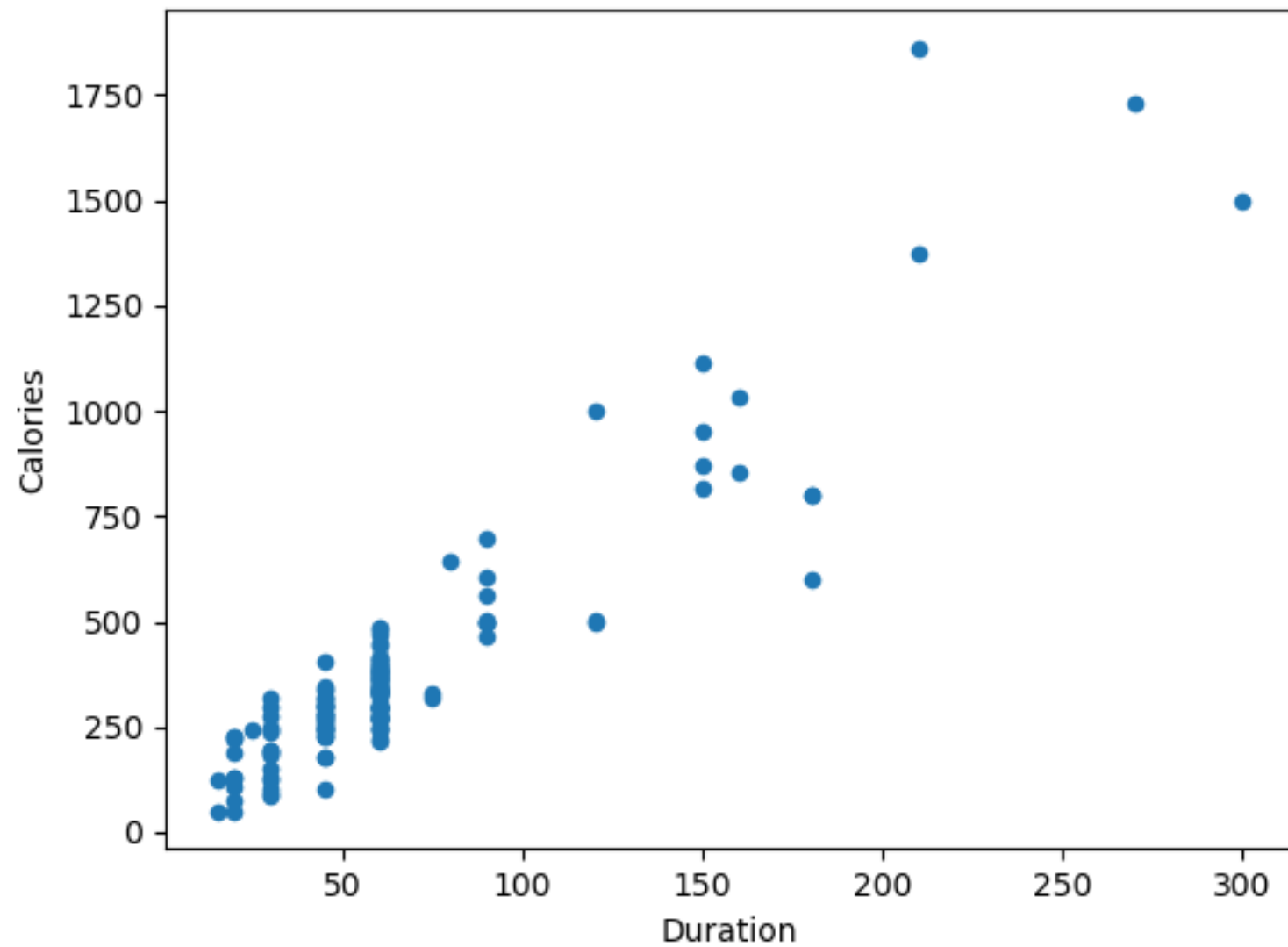
```
import pandas as pd
import matplotlib.pyplot as plt

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

df.plot(kind = 'scatter', x = 'Duration', y
= 'Calories')

plt.show()
```


Result



Remember: In the previous example, we learned that the correlation between "Duration" and "Calories" was **0.922721**, and we concluded with the fact that higher duration means more calories burned.

By looking at the scatterplot, I will agree.

Let's create another scatterplot, where there is a bad relationship between the columns, like "Duration" and "Maxpulse", with the correlation **0.009403**:

Example

A scatterplot where there are no relationship between the columns:

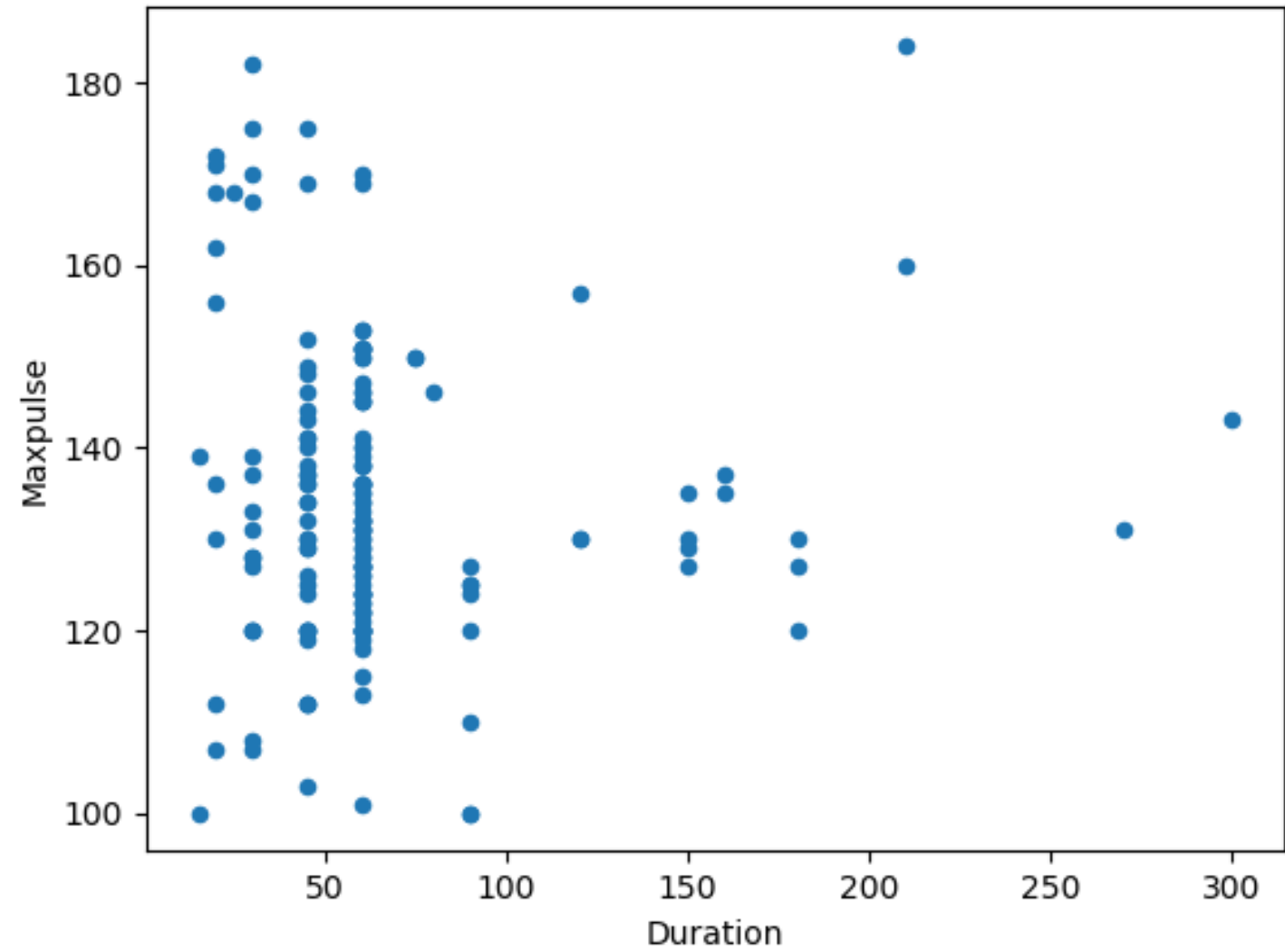
```
import pandas as pd
import matplotlib.pyplot as plt

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

df.plot(kind = 'scatter', x = 'Duration', y = 'Maxpulse')

plt.show()
```

Result



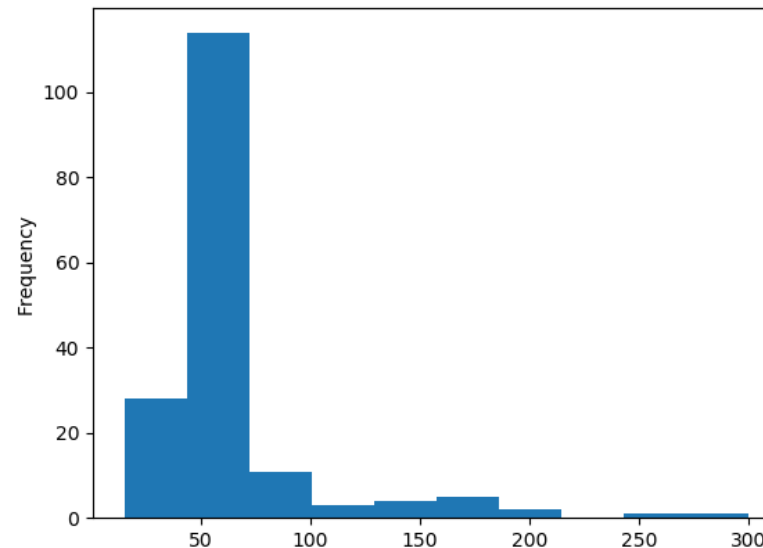
Histogram

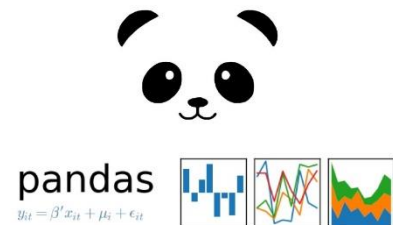
- Use the `kind` argument to specify that you want a histogram:
- `kind = 'hist'`
- A histogram needs only one column.
- A histogram shows us the frequency of each interval, e.g. how many workouts lasted between 50 and 60 minutes?
- In the example below we will use the "Duration" column to create the histogram:

Example

```
df["Duration"].plot(kind = 'hist')
```

Note: The histogram tells us that there were over 100 workouts that lasted between 50 and 60 minutes.





Thanks. Pandas!

- Wishing you all a happy learning experience.

Linked  [kaggle](#)  YouTube  GitHub