Python Libraries

NumPy and Pandas

- NumPy is a Python library.
- NumPy is used for working with **arrays**.
- NumPy is short for "Numerical Python".

Why Use NumPy?

In Python we have lists that serve the purpose of arrays, but they are slow to process.

NumPy aims to provide an array object that is up to 50x faster than traditional Python lists.

The array object in NumPy is called `ndarray`, it provides a lot of supporting functions that make working with `ndarray` very easy.

Arrays are very frequently used in data science, where speed and resources are very important.

Why is NumPy Faster Than Lists?

NumPy arrays are stored at one continuous place in memory unlike lists, so processes can access and manipulate them very efficiently.

This behavior is called locality of reference in computer science.

This is the main reason why NumPy is faster than lists. Also it is optimized to work with latest CPU architectures.

Which Language is NumPy written in?

NumPy is a Python library and is written partially in Python, but most of the parts that require fast computation are written in C or C++.

Installation of NumPy

If you have Python and PIP already installed on a system, then installation of NumPy is very easy.

Install it using this command:

```
C:\Users\*Your Name*>pip install numpy
```

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

Import NumPy

Once NumPy is installed, import it in your applications by adding the `import` keyword:

```
import numpy
```

Now NumPy is imported and ready to use.

Try this to confirm that you have successfully imported NumPy

```
import numpy

arr = numpy.array([1, 2, 3, 4, 5])
print(arr)
#output is [1 2 3 4 5]
```

NumPy as np

NumPy is usually imported under the `np` 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 numpy as np
```

Now the NumPy package can be referred to as `np` instead of `numpy`.

Example

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])

print(arr)
```

Do this yourself ****

```
#Check the np version installed by using this command
print(np.__version__)
```

Create a NumPy ndarray Object

NumPy is used to work with arrays. The array object in NumPy is called `ndarray`.

We can create a NumPy `ndarray` object by using the `array()` function.

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])
print(arr)
print(type(arr))
#output is
[1 2 3 4 5]
<class 'numpy.ndarray'> #code it shows that arr is numpy.ndarray type.
```

To create an `ndarray`, we can pass a list, tuple or any array-like object into the `array()` method, and it will be converted into an `ndarray`:

Use a tuple to create a NumPy array:

```
import numpy as np

arr = np.array((1, 2, 3, 4, 5))
print(arr)  #output is [1 2 3 4 5]
```

0-D Arrays

0-D arrays, or Scalars, are the elements in an array. Each value in an array is a 0-D array.

Create a 0-D array with value 42

```
import numpy as np

arr = np.array(42)
print(arr) #output is 42
```

1-D Arrays

An array that has 0-D arrays as its elements is called uni-dimensional or 1-D array.

These are the most common and basic arrays.

Create a 1-D array containing the values 1,2,3,4,5:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])
print(arr) #output is [1 2 3 4 5]
```

2-D Arrays

An array that has 1-D arrays as its elements is called a 2-D array.

These are often used to represent matrix or 2nd order tensors.

> NumPy has a whole sub module dedicated towards matrix operations called `numpy.mat`

Create a 2-D array containing two arrays with the values 1,2,3 and 4,5,6:

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]])
print(arr)
#output is
[[1 2 3]
 [4 5 6]]
```

3-D arrays

An array that has 2-D arrays (matrices) as its elements is called 3-D array.

These are often used to represent a 3rd order tensor.

Example

Create a 3-D array with two 2-D arrays, both containing two arrays with the values 1,2,3 and 4,5,6:

```
import numpy as np

arr = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])
print(arr)
#output is
[[[1 2 3]
  [4 5 6]]

 [[1 2 3]
  [4 5 6]]]
```

Access Array Elements Using indexing

- Array indexing is the same as accessing an array element.
- You can access an array element by referring to its index number.
- The indexes in NumPy arrays start with 0, meaning that the first element has index 0, and the second has index 1 etc.

Example

Get the first element from the following array:

```
import numpy as np

arr = np.array([1, 2, 3, 4])
print(arr[0])
```

Access 2-D Arrays

To access elements from 2-D arrays we can use comma separated integers representing the dimension and the index of the element.

Think of 2-D arrays like a table with rows and columns, where the dimension represents the row and the index represents the column.

Access the element on the first row, second column:

```
import numpy as np

arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])

print('2nd element on 1st row: ', arr[0, 1])
#output is ---------> 2nd element on 1st row:  2
```

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

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.

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.

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
```

Import Pandas

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

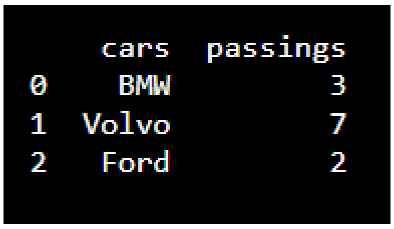
```
import pandas
```

Now Pandas is imported and ready to use.

Let's see an example:

```
import pandas

mydataset = {
  'cars': ["BMW", "Volvo", "Ford"],
  'passings': [3, 7, 2]
}

myvar = pandas.DataFrame(mydataset)

print(myvar)
```

Output



```
    cars  passings
0    BMW         3
1  Volvo         7
2   Ford         2
```

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`.

Check panda version

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

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

```
import pandas as pd

mydataset = {
  'cars': ["BMW", "Volvo", "Ford"],
  'passings': [3, 7, 2]
}

myvar = pd.DataFrame(mydataset)

print(myvar)
```

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.

For example:

Create a simple Pandas Series from a list:

```
import pandas as pd

a = [1, 7, 2]
myvar = pd.Series(a)
print(myvar)

#Return the first value of the Series:

print(myvar[0])
```

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)
#output
day1    420
day2    380
day3    390
dtype: int64
```

> **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)
#output
day1    420
day2    380
dtype: int64
```

DataFrames

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

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

Example

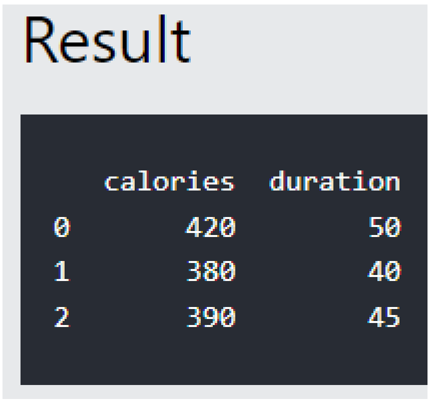
Create a DataFrame from two Series:

```
import pandas as pd

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

myvar = pd.DataFrame(data)
print(myvar)
```

Example 2

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

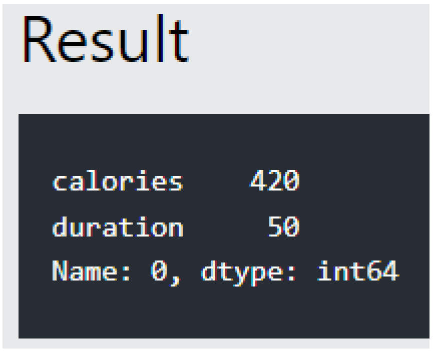
As you can see from the result above, the DataFrame is like a table with rows and columns.

Pandas use the `loc` attribute to return one or more specified row(s)

Example

Return row 0:

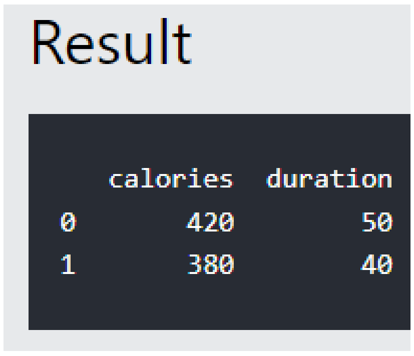
```
#refer to the row index:
print(df.loc[0])
```


Result
`calories    420`
`duration     50`
`Name: 0, dtype: int64`
	Note: This example returns a Pandas **Series**.

Example

Return row 0 and 1:

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


Result

```
   calories  duration
0       420        50
1       380        40
```

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

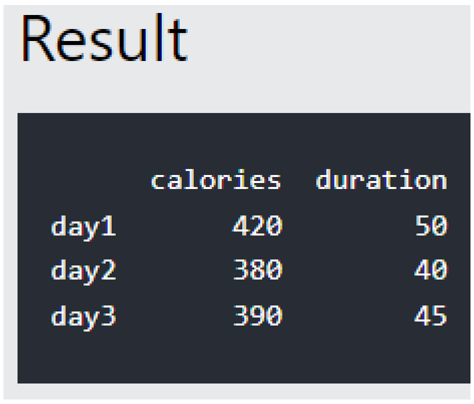
Named Indexes

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
```

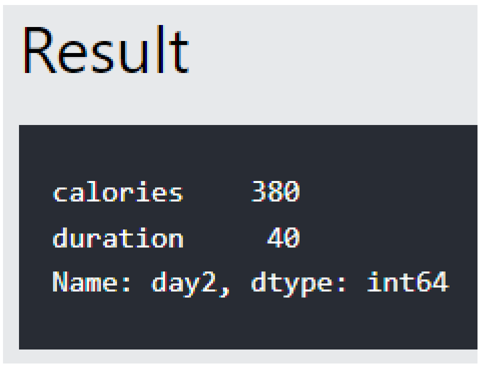
Locate Named Indexes

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
```

Load Files Into a DataFrame

If your data sets are stored in a file, Pandas can load them into a DataFrame.

Example

Load a comma separated file (CSV file) into a DataFrame:

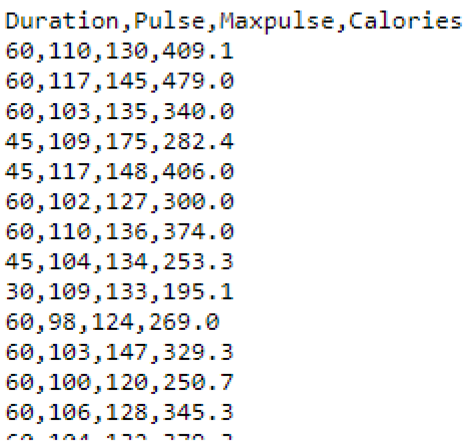
```
import pandas as pd

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

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 these examples, use the CSV file called '**data.csv**'.Or [Open data.csv]()



```
Duration,Pulse,Maxpulse,Calories
60,110,130,409.1
60,117,145,479.0
60,103,135,340.0
45,109,175,282.4
45,117,148,406.0
60,102,127,300.0
60,110,136,374.0
45,104,134,253.3
30,109,133,195.1
60,98,124,269.0
60,103,147,329.3
60,100,120,250.7
60,106,128,345.3
```

Load the CSV into a DataFrame:

```
import pandas as pd
```

```
df = pd.read_csv('data.csv')
print(df.to_string())
#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:

#Print the DataFrame without the to_string() method:

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

#Check the number of maximum returned rows:

print(pd.options.display.max_rows)
```

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.

Example

Increase the maximum number of rows to display the entire DataFrame:

```
import pandas as pd

pd.options.display.max_rows = 9999
df = pd.read_csv('data.csv')
print(df)
```

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.

Load the JSON file into a DataFrame:

```
import pandas as pd

df = pd.read_json('data.json')
print(df.to_string())   #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:

```
import pandas as pd

data = {
  "Duration":{
    "0":60,
    "1":60,
    "2":60,
    "3":45,
    "4":45,
    "5":60
  },
  "Pulse":{
    "0":110,
    "1":117,
    "2":103,
    "3":109,
    "4":117,
    "5":102
  },
  "Maxpulse":{
    "0":130,
    "1":145,
    "2":135,
    "3":175,
    "4":148,
    "5":127
  },
  "Calories":{
    "0":409,
    "1":479,
    "2":340,
    "3":282,
    "4":406,
    "5":300
  }
}

df = pd.DataFrame(data)

print(df)
```

Pandas - Analyzing DataFrames

Viewing the Data

One of the most used method for getting a quick overview of the DataFrame, is the `head()` method.

> The `head()` method returns the headers and a specified number of rows, starting from the top.

Get a quick overview by printing the first 10 rows of the DataFrame:

```
import pandas as pd

df = pd.read_csv('data.csv')
print(df.head(10)) #if the number of rows is not specified, the head() method
will return the top 5 rows.
```

In our examples we will be using a CSV file called 'data.csv'.

Download data.csv, or open data.csv in your browser.

Example

Print the first 5 rows of the DataFrame:

```
import pandas as pd

df = pd.read_csv('data.csv')
print(df.head())
```

There is also a `tail()` method for viewing the *last* rows of the DataFrame.

The `tail()` method returns the headers and a specified number of rows, starting from the bottom.

Example

Print the last 5 rows of the DataFrame:

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

Info About the Data

The DataFrames object has a method called `info()`, that gives you more information about the data set.

Example

Print information about the data:

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

Result

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 169 entries, 0 to 168
Data columns (total 4 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   Duration  169 non-null    int64
 1   Pulse     169 non-null    int64
 2   Maxpulse  169 non-null    int64
 3   Calories  164 non-null    float64
dtypes: float64(1), int64(3)
memory usage: 5.4 KB
None
```

Result Explained

The result tells us there are 169 rows and 4 columns:

```
RangeIndex: 169 entries, 0 to 168
Data columns (total 4 columns):
```

And the name of each column, with the data type:

```
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   Duration  169 non-null    int64
 1   Pulse     169 non-null    int64
 2   Maxpulse  169 non-null    int64
 3   Calories  164 non-null    float64
```

Null Values

The `info()` method also tells us how many Non-Null values there are present in each column, and in our data set it seems like there are 164 of 169 Non-Null values in the "Calories" column.

Which means that there are 5 rows with no value at all, in the "Calories" column, for whatever reason.

Empty values, or Null values, can be bad when analyzing data, and you should consider removing rows with empty values. This is a step towards what is called *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 tutorial you will learn how to deal with all of them.

Our Data Set

In the next chapters we will use this data set:

```
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 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).

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())
#Notice in the result that some rows have been removed (row 18, 22 and 28).
#These rows had cells with empty values.
```

> 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())
```

> 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)
#This operation inserts 130 in empty cells in the "Calories" column (row 18 and
28).
```

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)
```

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:

Let's try to convert all cells in the 'Date' column into dates.

Pandas has a `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())
```

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)
```

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.

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)
```

Discovering Duplicates

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

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 `duplicated()` method.

The `duplicated()` method returns a Boolean values for each row:

Example

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

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

Removing Duplicates

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

Example

Remove all duplicates:

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

> 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.

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.

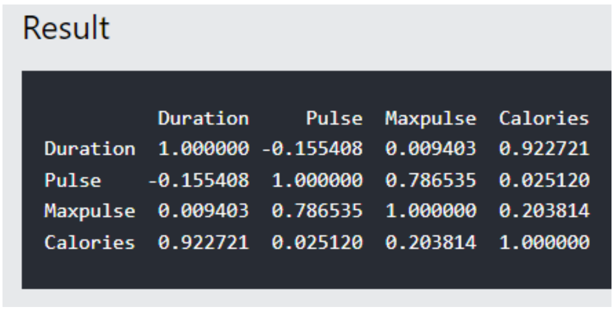
The examples in this page uses a CSV file called: 'data.csv'.

Download data.csv. or Open data.csv

Example

Show the relationship between the columns:

```
df.corr()
```



Result

```
          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
```

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.

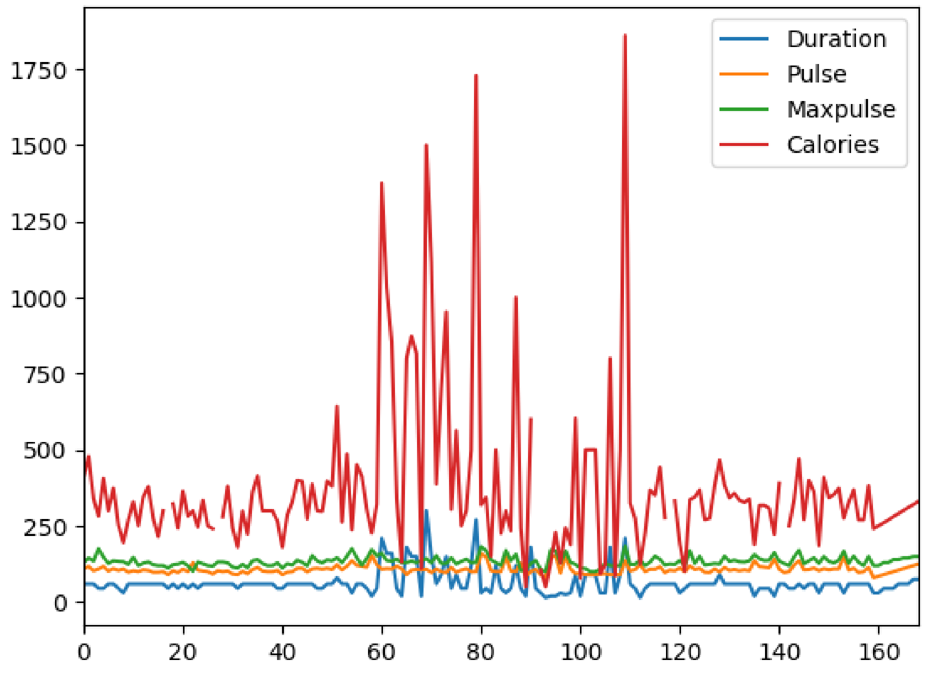
Good Correlation:

"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 - 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:

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

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

df.plot()

plt.show()
```

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

Download data.csv or Open data.csv

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.

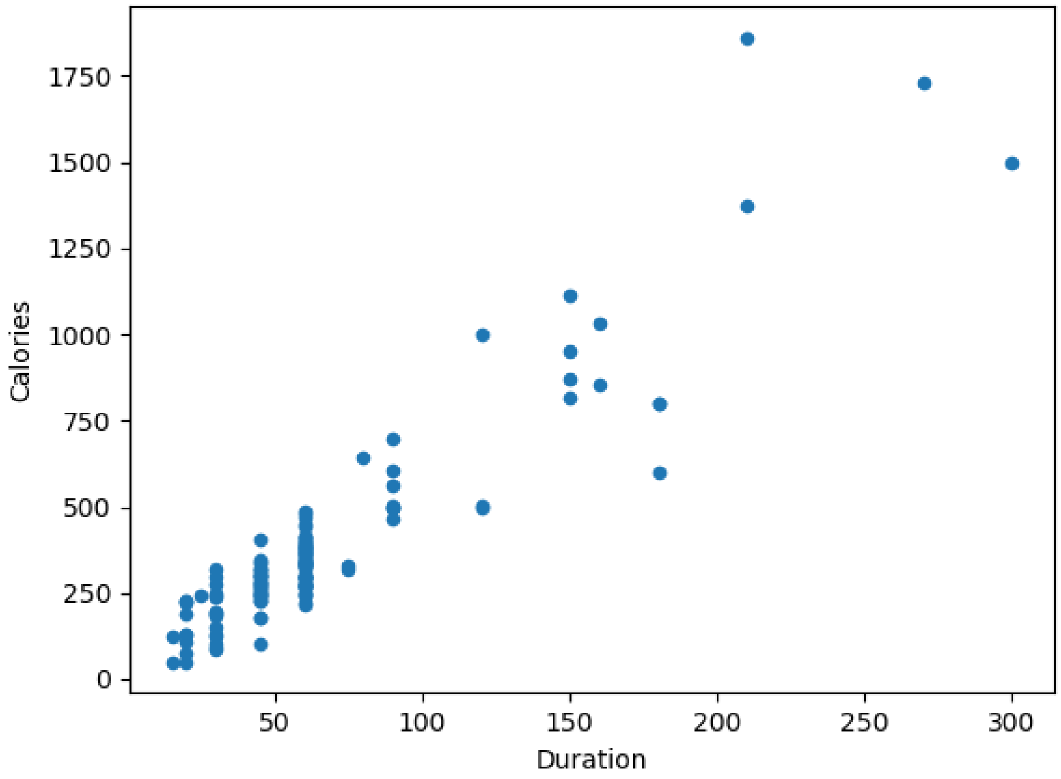
In the example below we will use "Duration" for the x-axis and "Calories" for the y-axis.

Include the x and y arguments like this:

```
x = 'Duration', y = 'Calories'
```

Example

```
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()
#See result below
```

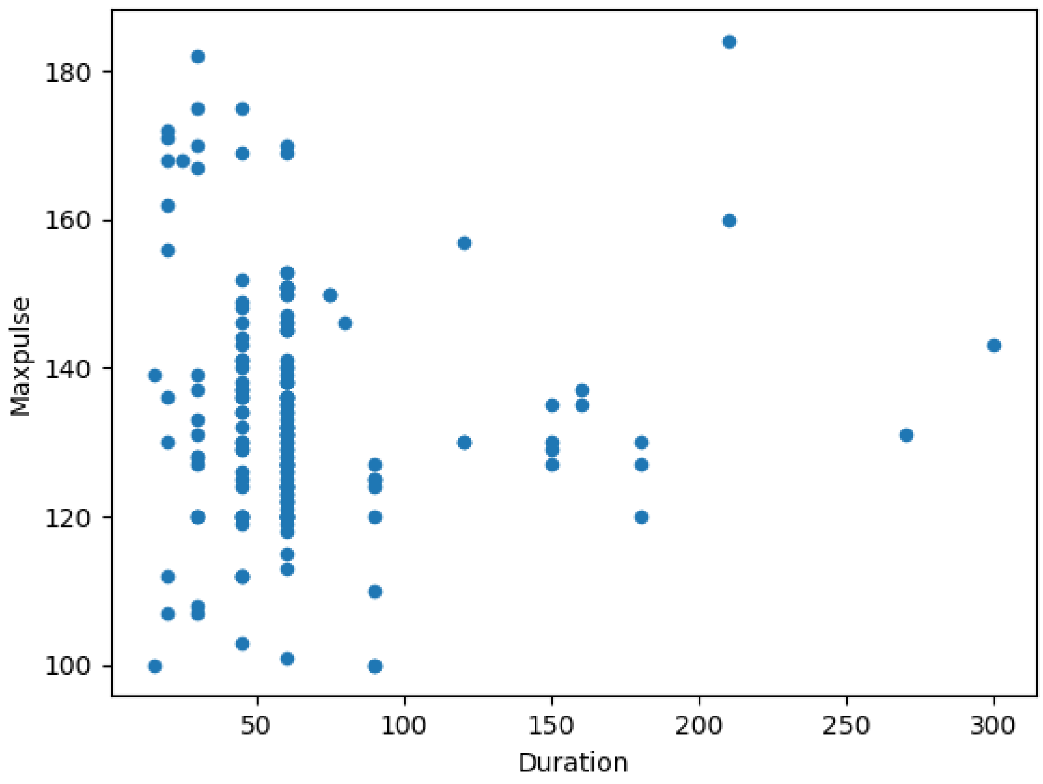


Let's create another scatterplot, where there is a bad relationship between the columns, like "Duration" and "Maxpulse", with the correlation `0.009403`:

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()
```



Histogram

Use the `kind` argument to specify that you want a histogram:

```
kind = 'hist'
```

A histogram needs only one column.

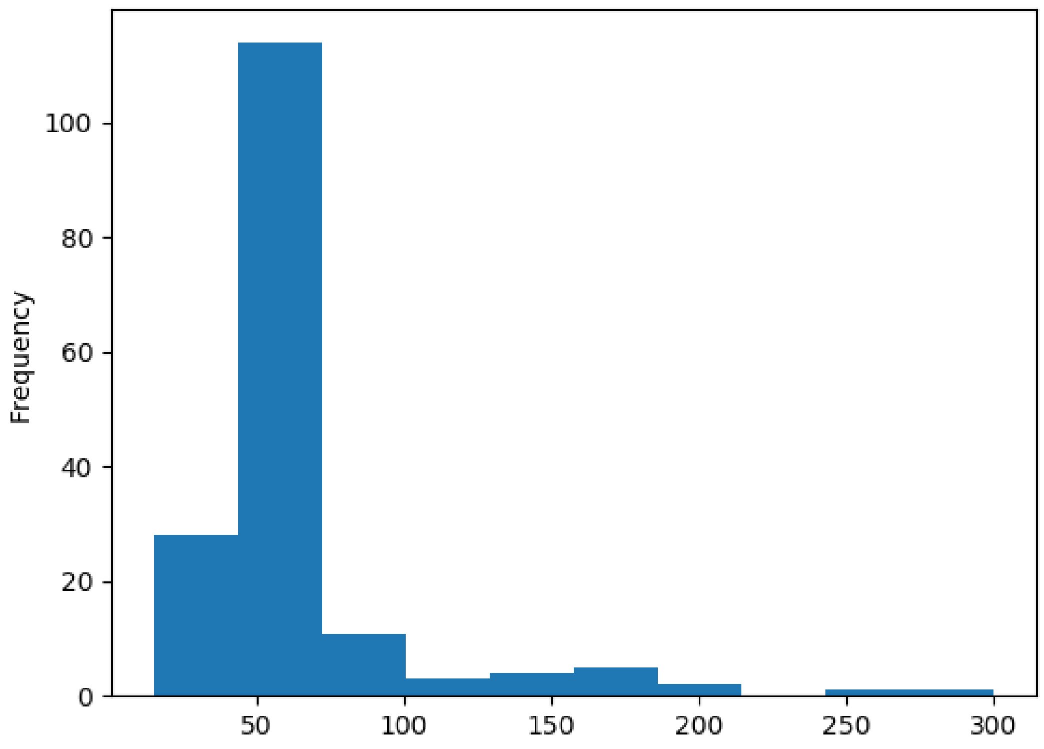
A histogram shows us the frequency of each interval, e.g. how many workouts lasted between 50 and 60 minutes?

In the example below we will use the "Duration" column to create the histogram:

Example

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

Result



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

- Do a summary of the sub-topics covered in this chapter.