Pandas

Installation instructions

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. 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.

pandas is part of the Anaconda distribution and can be installed with Anaconda or Miniconda:

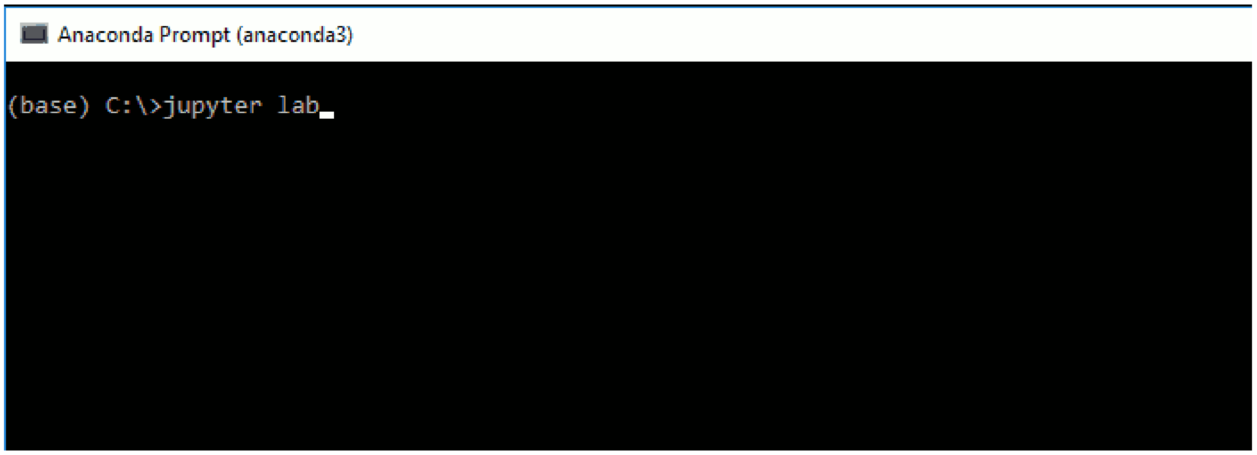
```
conda install pandas

pip install pandas
```

The next steps provides the easiest and recommended way to set up your environment to use pandas. Other installation options can be found in the advanced installation page.

1. Download Anaconda for your operating system and the latest Python version, run the installer, and follow the steps. Please note:
    - It is not needed (and discouraged) to install Anaconda as root or administrator.
    - When asked if you wish to initialize Anaconda3, answer yes.
    - Restart the terminal after completing the installation.

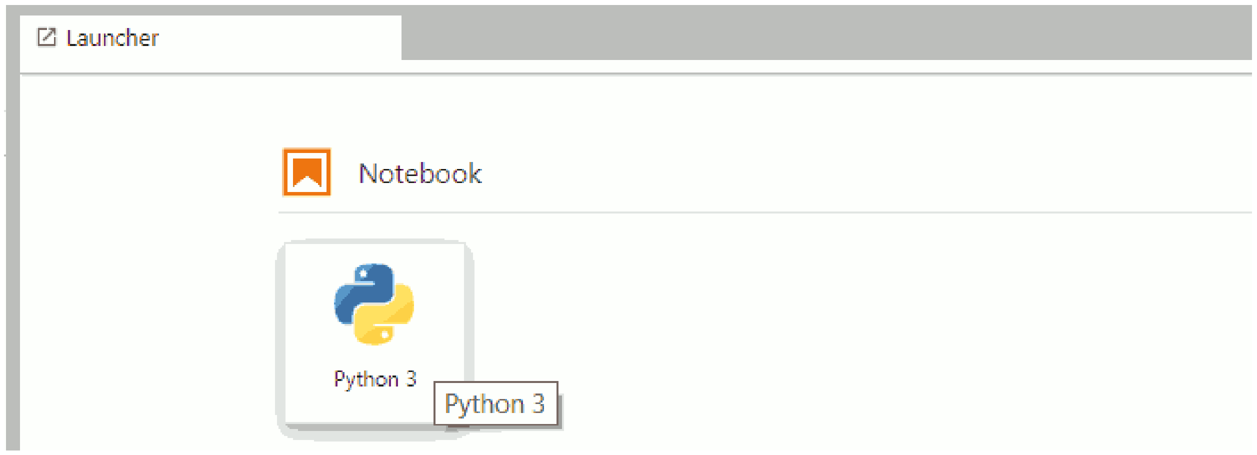
    Detailed instructions on how to install Anaconda can be found in the Anaconda documentation.

2. In the Anaconda prompt (or terminal in Linux or macOS), start JupyterLab:

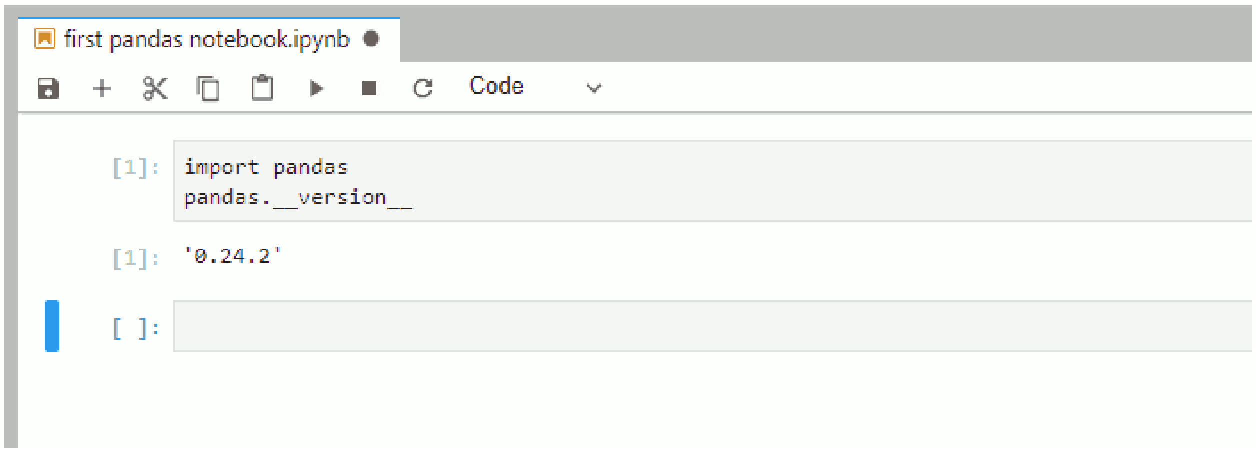


```
(base) C:\>jupyter lab
```

3. In JupyterLab, create a new (Python 3) notebook:



4. In the first cell of the notebook, you can import pandas and check the version with:



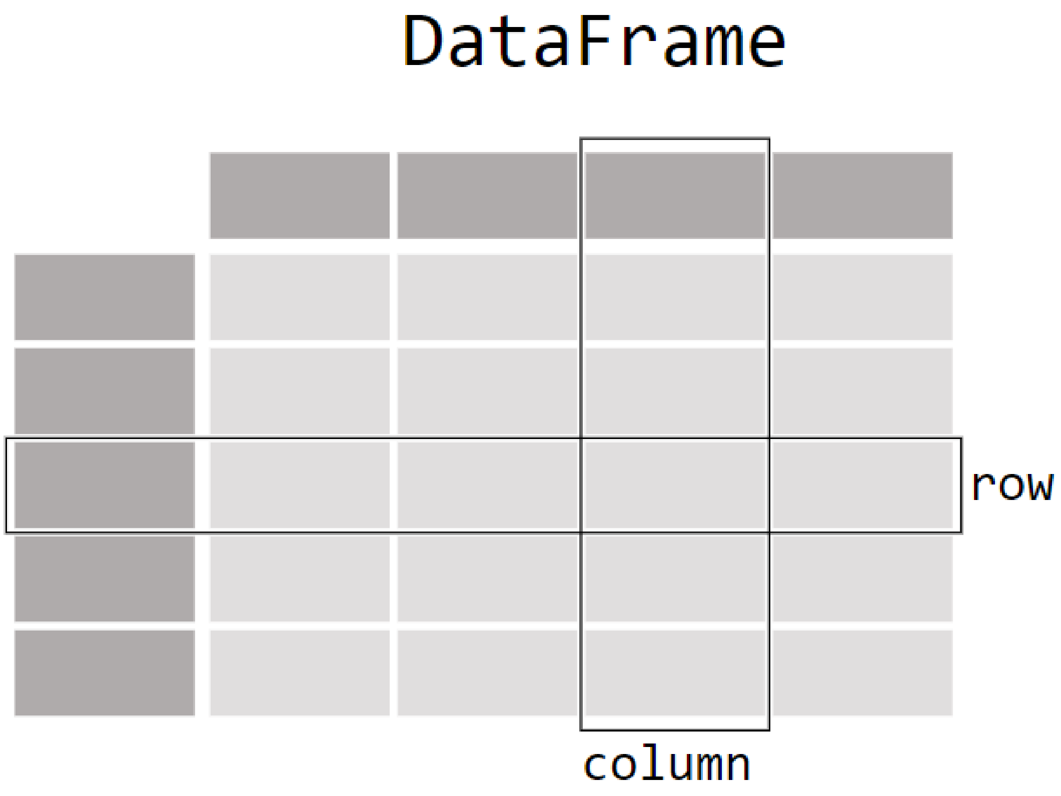
```
[1]: import pandas
     pandas.__version__

[1]: '0.24.2'

[ ]:
```

5. Now you are ready to use pandas, and you can write your code in the next cells.

Introduction To panda’s

When working with tabular data, such as data stored in spreadsheets or databases, pandas is the right tool for you. pandas will help you to explore, clean, and process your data. In pandas, a data table is called a `DataFrame`.

Pandas data table representation



I want to store passenger data of the Titanic. For a number of passengers, I know the name (characters), age (integers) and sex (male/female) data.

Example:

```
df = pd.DataFrame(
        {
            "Name": [
                "Braund, Mr. Owen Harris",
                "Allen, Mr. William Henry",
                "Bonnell, Miss. Elizabeth",
            ],
            "Age": [22, 35, 58],
            "Sex": ["male", "male", "female"],
            }
        )


    df
```

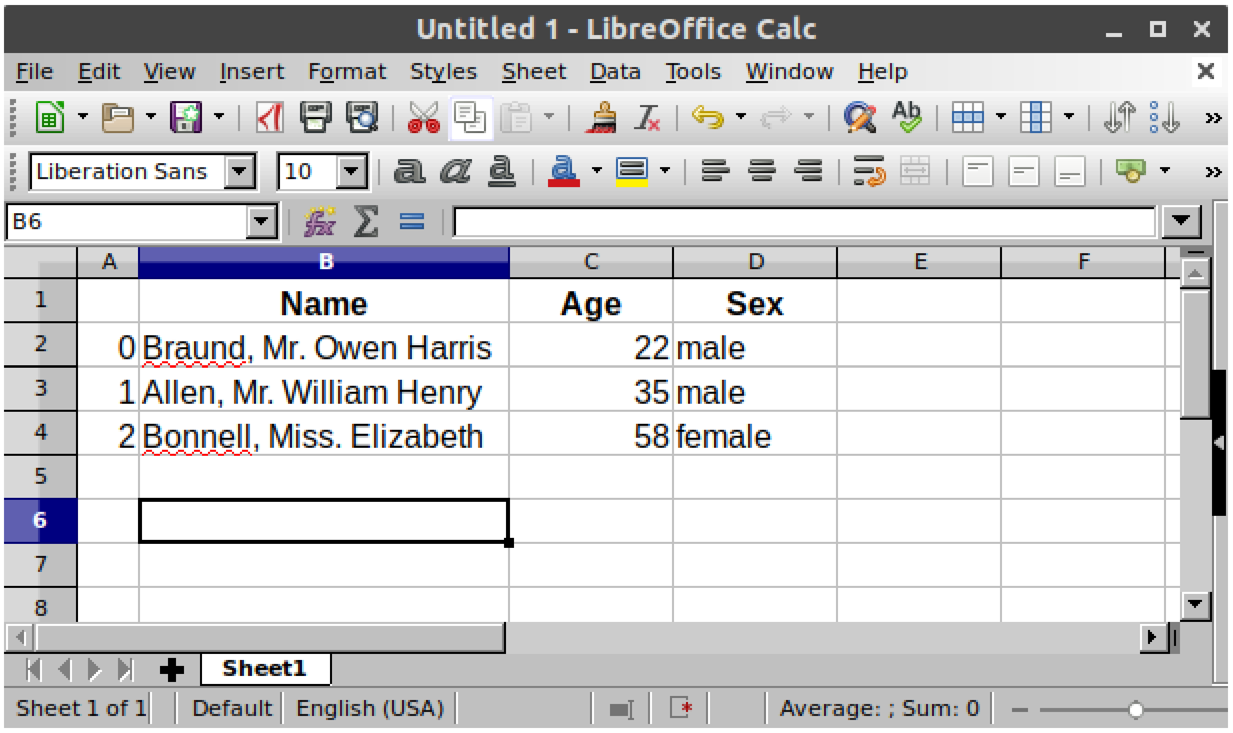
Example:

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

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

    print(df)
```

To manually store data in a table, create a `DataFrame`. When using a Python dictionary of lists, the dictionary keys will be used as column headers and the values in each list as columns of the `DataFrame`.

A **`DataFrame`** is a 2-dimensional data structure that can store data of different types (including characters, integers, floating point values, categorical data and more) in columns. It is similar to a spreadsheet, a SQL table or the `data.frame` in R.

- The table has 3 columns, each of them with a column label. The column labels are respectively `Name`, `Age` and `Sex`.
- The column `Name` consists of textual data with each value a string, the column `Age` are numbers and the column `Sex` is textual data.
- In spreadsheet software, the table representation of our data would look very similar:


Untitled 1 - LibreOffice Calc
File Edit View Insert Format Styles Sheet Data Tools Window Help
Liberation Sans
10
B6
Sheet1
Sheet 1 of 1
Default
English (USA)
Average: ; Sum: 0

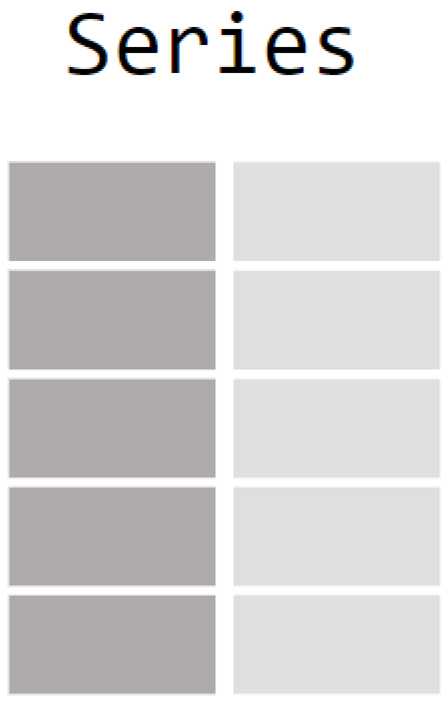
	A	B	C	D	E	F
1		Name	Age	Sex		
2	0	Braund, Mr. Owen Harris	22	male		
3	1	Allen, Mr. William Henry	35	male		
4	2	Bonnell, Miss. Elizabeth	58	female		
5						
6						
7						
8						

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)

```
print(df.loc[0])
output:
  calories    420
  duration     50
  Name: 0, dtype: int64


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

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

Each column in a **DataFrame** is a **Series**



```
df["Age"]

Out[4]:

0    22

1    35

2    58
```

Name: Age, dtype: int64

If you are familiar to Python dictionaries, the selection of a single column is very similar to selection of dictionary values based on the key.

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 Series

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

You can create a `Series` from scratch as well:

Example:

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

Example:

```
ages = pd.Series([22, 35, 58], name="Age")
ages
Out[6]:
0    22
1    35
2    58
Name: Age, dtype: int64
```

I want to know the maximum Age of the passengers

We can do this on the `DataFrame` by selecting the `Age` column and applying `max()`:

```
df["Age"].max()
Out[7]: 58
Or to the Series:
ages.max()
Out[8]: 58
```

I’m interested in some basic statistics of the numerical data of my data table

The describe() method provides a quick overview of the numerical data in a DataFrame. As the Name and Sex columns are textual data, these are by default not taken into account by the describe() method.

```
df.describe()
Out[9]:
             Age
count   3.000000
mean   38.333333
std    18.230012
min    22.000000
25%    28.500000
50%    35.000000
75%    46.500000
max    58.000000
```

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.

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

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

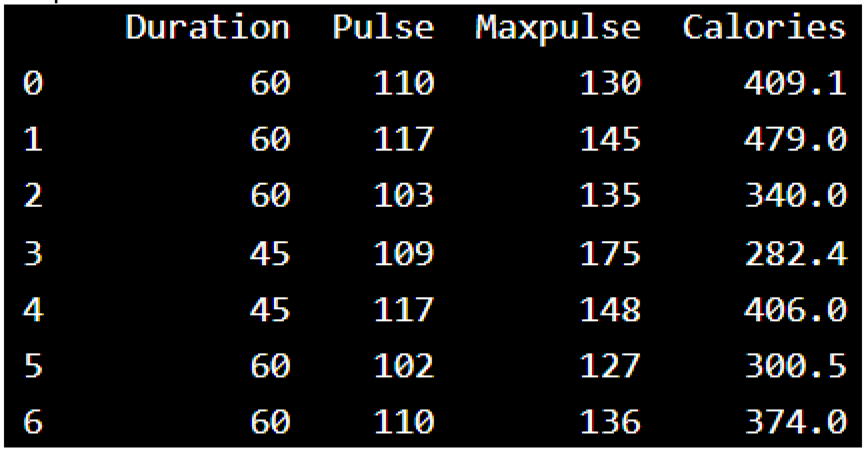
```
a = [1, 7, 2]

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

print(myvar)


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

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'.

https://www.w3schools.com/python/pandas/data.csv.txt

```
import pandas as pd

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

print(df.to_string())

output:
```



```
   Duration  Pulse  Maxpulse  Calories
0        60    110       130     409.1
1        60    117       145     479.0
2        60    103       135     340.0
3        45    109       175     282.4
4        45    117       148     406.0
5        60    102       127     300.5
6        60    110       136     374.0
```

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

```
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.

```
print(pd.options.display.max_rows)
pd.options.display.max_rows = 9999
print(pd.options.display.max_rows)
```

Read Excel File

pandas.read excel() is pandas read_excel function which is used to read the excel sheets with extensions (.xlsx, .xls, .xlsx, .xlsm, .xlsb, .odf, .ods and .odt) into pandas DataFrame object.

A "Pandas DataFrame object" is returned by reading a single sheet while reading two sheets results in a Dict of DataFrame.

We can also load excel files from a URL or which are stored in the local filesystem.

It supports http, ftp, s3, and file for URLs.

The Pandas Read Excel Function has various parameters.

Some of them are as follows :

```
import pandas as pd

# Read the Excel file
df = pd.read_excel('filename.xlsx')

# Print the first five rows of the DataFrame
print(df.head())
```

In this example, **`pd.read_excel()`** reads the Excel file into a Pandas DataFrame. You'll need to replace **`'filename.xlsx'`** with the path to your own Excel file.

You can also specify the sheet name by adding an additional argument to **`pd.read_excel()`**. For example, if you want to read the data from a sheet named "Sheet1", you can use the following code:

```
df = pd.read_excel('filename.xlsx', sheet_name='Sheet1')
```

After reading the Excel file into a DataFrame, you can perform various operations on the data, such as filtering, grouping, and aggregation.

Specify Data Types in Pandas read_excel

When we are reading an excel file, it is easy to specify the data type of the columns because of pandas.

The main three objectives served by this are :

- preventing the improper reading of data
- accelerating the reading process
- preserving memory

We can enter a dictionary where the data types are the values and the keys are the columns. This guarantees that the data are prepared correctly. Let's look at how to define the data types for our columns.

FilePath = //localhost/path/to/table.xlsx.

```
import pandas as pd
data=pd.read_excel(io='/content/Subject_Scores.xlsx' ,dtype={'Roll.
No.':int,'Student_Name':object,'English':int,'Maths':int})

print(data)
```

Read Jason File

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'.

```
import pandas as pd
df = pd.read_json('data.json')
print(df.to_string())
```

```
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)
```

To read a JSON file in Python with Pandas, you can use the read_json() method. Here's an example code snippet that shows how to do this:

```
import pandas as pd

# Read the JSON file
df = pd.read_json('filename.json')

# Print the first five rows of the DataFrame
print(df.head())
```

In this example, pd.read_json() reads the JSON file into a Pandas DataFrame. You'll need to replace 'filename.json' with the path to your own JSON file.

You can also specify additional arguments to pd.read_json() to customize the behavior of the method. For example, you can specify the orientation of the JSON file (i.e. whether it's "split" or "records") using the orient argument. Here's an example:

```
df = pd.read_json('filename.json', orient='records')
```

After reading the JSON file into a DataFrame, you can perform various operations on the data, such as filtering, grouping, and aggregation. Pandas provides a wide range of functions and methods for working with data.

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.

```
import pandas as pd

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

print(df.head(10)) # brings header and first 10 rows of data

Example:

import pandas as pd

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

print(df.head())# brings header and first 5 rows of data
```

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.

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

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

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

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

Empty Cells

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

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

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:

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

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:

```
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:

```
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:

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

```
import pandas as pd
df = pd.read_csv('data.csv')
x = df["Calories"].mean()
df["Calories"].fillna(x, inplace = True)
```

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)
```

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)
```

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:

```
import pandas as pd
df = pd.read_csv('data.csv')
df['Date'] = pd.to_datetime(df['Date'])
print(df.to_string())
```

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.

```
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:

```
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.

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

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.

```
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. y 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:

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

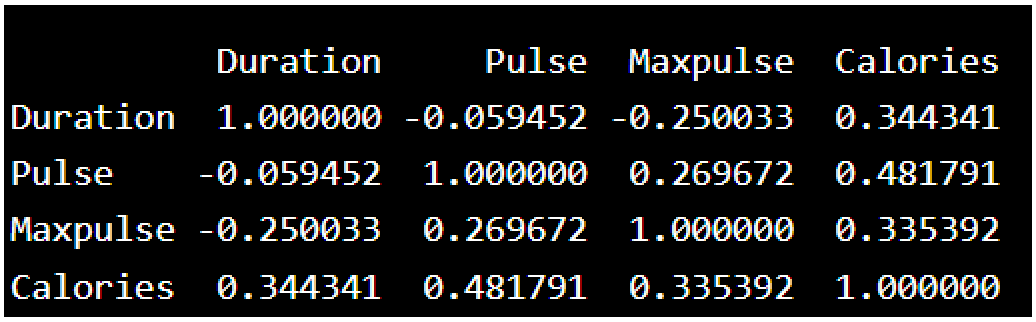
`for removing duplicates`

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

Correlation in data

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'.

```
import pandas as pd
df = pd.read_csv('data.csv')
print(df.corr())
```



```
          Duration     Pulse  Maxpulse  Calories
Duration  1.000000 -0.059452 -0.250033  0.344341
Pulse    -0.059452  1.000000  0.269672  0.481791
Maxpulse -0.250033  0.269672  1.000000  0.335392
Calories  0.344341  0.481791  0.335392  1.000000
```

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.