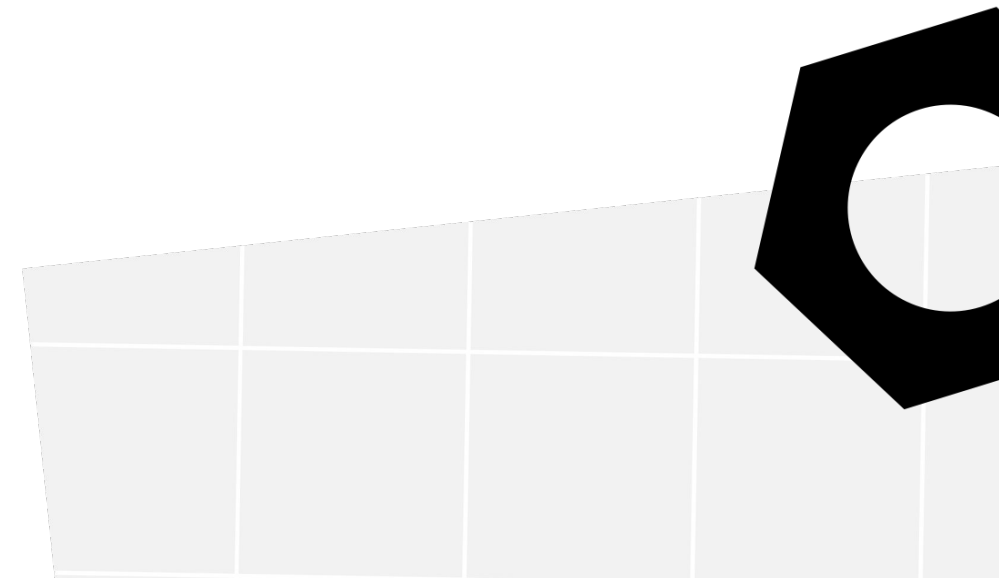
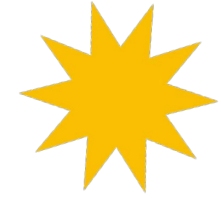




Data sets processing

Data Science course

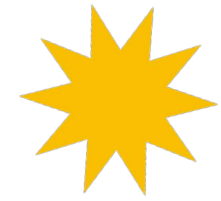




Practical tasks during classes

Additional practical tasks were included in the presentation for the Data Set Processing block.

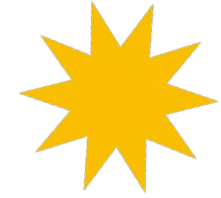
You can download the solutions from the section "Additional materials for the block" which can be found [here](#).



Practical tasks during classes

Note: In case of problems with the installation of pandas-profiling on Google Colaboratory, we recommend installing the older version using the command below:

```
import sys
!pip install -U pip
!{sys.executable} -m pip install -U pandas-profiling[notebook]
!pip install pandas-profiling==2.7.1
!jupyter nbextension enable --py widgetsnbextension
#after this restart the environment ctrl + M
```



Schedule:

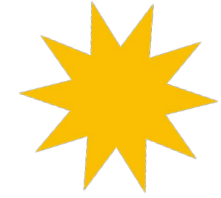
- 1) 5 hours of practice with a trainer + 2 hours of independent work:
 - data frames – pandas library
 - working with data, basic operations
 - feature engineering
- 1) 5 hours of practice with a trainer + 2 hours of independent work:
 - SQL language and databases
 - database programming
- 1) 5 hours of practice with a trainer + 2 hours of independent work:
 - sql_alchemy
 - working with files – serialization
 - pandas profiling
 - getting data from API

note: for the tasks performed during the classes we will use a jupyter notebook with tasks from Pandas – you can download it from the "Additional materials for the block" section [here](#)



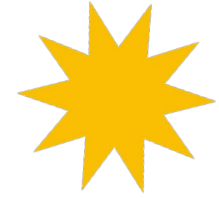
Pandas library





Pandas – library description

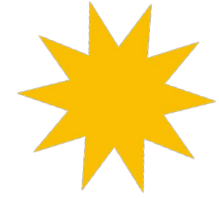
Pandas is another library essential for Python data analysis. It provides efficient data structures that make working with tabular data simple and intuitive. The aim of the creators is to maintain the status of a library necessary for everyday analysis. They also wish to become the most powerful open-source data analysis tool in any programming language. Currently, the project is still growing rapidly and its knowledge is essential for every data scientist.



Pandas – when to use it?

Pandas will be a good choice for the following uses:

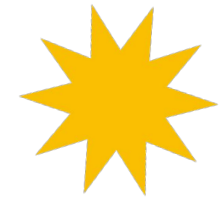
- Tabular data (columns as in SQL or Excel)
- Data representing time series
- Matrices
- Results of measurements and statistics



Pandas – strengths

Pandas strengths include::

- Simple handling of missing values (NaN),
- Ability to modify the size of the DataFrame – we can add and remove columns and rows,
- Automatic data alignment in calculations (as in NumPy),
- Groupby method working in the same way as in SQL,
- It's easy to create a DataFrame from another object,
- Cutting, indexing and creating subsets,
- Joining (join and merge) sets.



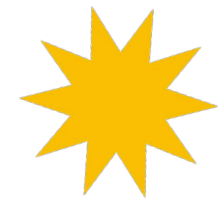
Pandas – data structures

Series is a one-dimensional data structure (one-dimensional numpy matrix) that stores a unique index in addition to the data.

Creating – *pd.Series(content)*

```
pd.Series(np.random.random(10))
```

```
0    0.661750
1    0.072319
2    0.758071
3    0.035651
4    0.992312
5    0.488344
6    0.083446
7    0.319852
8    0.419058
9    0.310146
dtype: float64
```

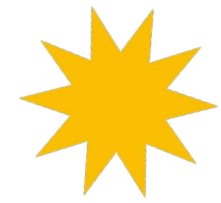


Pandas – data structures

The second structure in pandas is DataFrame. It is a two or more dimensional data structure, most often in the form of a table with rows and columns. Columns have names and rows have indexes.

Creating – `pd.DataFrame(content)`

Date	Open	High	Low	Close	Volume	Adj Close
2014-09-16	99.80	101.26	98.89	100.86	66818200	100.86
2014-09-15	102.81	103.05	101.44	101.63	61216500	101.63
2014-09-12	101.21	102.19	101.08	101.66	62626100	101.66
...



Pandas – reading data

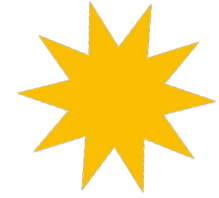
The first step is usually the same. The data is stored in csv, tsv files, databases, excel files etc. You can load them by using the `pd.read_csv` function

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html

The most important arguments are all worth reviewing. This is probably the most important function of pandas, proper loading of the file makes it easier to work on:

- file path
- column separator (comma by default)
- heading
- index column
- column names

```
# tsv file - separated by tabs  
chipotle = pd.read_csv('ML-datasets/chipotle.tsv', sep='\t')
```

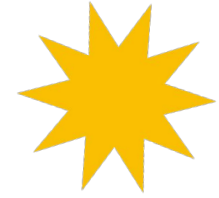


Pandas – displaying data

After the data is loaded, the next step is to display it. You can do it in many ways:

- `df.head(n)` – displays the first `n` data records
- `df.tail(n)` – displays the last `n` data records
- `df.sample(n)` – displays random data records in the number `n`
- `df["column"]` or `df.column` – displays a given column (as Series)
- `df[["column"]]` – displays a given column (as DataFrame)
- `df[["column1", "column2"]]` – displays several columns
- `df.index` – displays index names (lines)
- `df.columns` – displays column names
- `df.info()` – general information about the collection

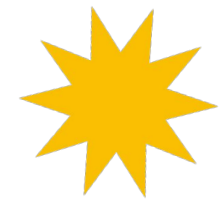
Pandas – displaying data – loc and iloc



For the dataframe, there are two functions for retrieving specific data:

- loc looks for column and index names
[[rows],[columns]]
- iloc searches by their ordinal numbers
[[row number],[column number]]

```
print(df['A']) # only column A
print('----')
display(df[0:3]) # rows from 0 to 2
print('----')
display(df['20130102':'20130104']) # from January 2nd to 3rd
print('----')
print(df.loc[dates[0]]) # by value in the index
print('----')
display(df.loc[:, ['A', 'B']]) # all rows but only columns A and B
print('----')
display(df.loc['20130102':'20130104', ['A', 'B']]) # rows range
print('----')
print(df.loc['20130102', ['A', 'B']]) # only one row
print('----')
print(df.loc[dates[0], 'A']) # one cell
print('----')
print(df.at[dates[0], 'A']) # one cell
print('----')
display(df.iloc[[3]]) # one row by row number as DF
display(df.iloc[3]) # as Series
print('----')
display(df.iloc[3:5, 0:2]) # by indices
print('----')
display(df[df.A < 0]) # indexing as a logical condition
print('----')
display(df[df > 0]) # searches for values below zero
```



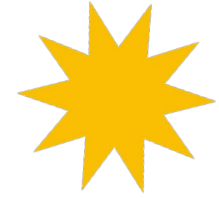
Pandas – displaying data – conditions

You can also put a logical condition in parentheses.

```
print(read.how == 'SEO') # Series with values True/False by rows
display(read[read.how == 'SEO']) # Displays all rows with True value
```

```
time
2018-01-01 00:01:01    True
2018-01-01 00:03:20    True
2018-01-01 00:04:01   False
2018-01-01 00:04:02   False
2018-01-01 00:05:03   False
...
2018-01-01 23:57:14   False
2018-01-01 23:58:33    True
2018-01-01 23:59:36   False
2018-01-01 23:59:36   False
2018-01-01 23:59:38   False
Name: how, Length: 1795, dtype: bool
```

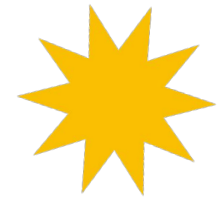
	status	country	identifier	how	continent
time					
2018-01-01 00:01:01	read	country_7	2458151261	SEO	North America
2018-01-01 00:03:20	read	country_7	2458151262	SEO	South America
2018-01-01 00:08:57	read	country_7	2458151272	SEO	Australia
2018-01-01 00:11:22	read	country_7	2458151276	SEO	North America
2018-01-01 00:13:05	read	country_8	2458151277	SEO	North America



Pandas – overwriting data

You can also overwrite or append data with the assignment command (=):

- `df["column"] = data_series`
- `df[["column"]] = dataframe`
- `df[["column1", "column2"]] = dataframe`
- `df.loc[column:row] = data`
- `df.iloc[column_number:row_number] = data`

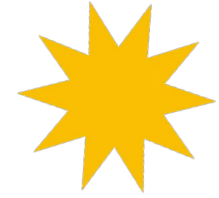


Pandas – operating on data

All the following operations for the dataframe will return values for each of the columns. We can also select the column for which we want to perform these operations:

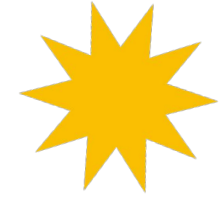
- `df.count()` – counting the number of items (not NaNs)
- `df.column.value_counts()` – counting the number of unique items
- `df.sum()` – sum of all items
- `df.min()` – minimal element
- `df.max()` – maximal element
- `df.mean()` – dataset's mean
- `df.median()` – dataset's median

Task 1

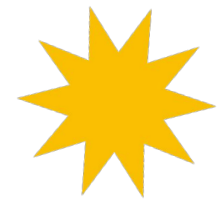


Create a dataframe with the 10 names of pupils and the number of points they obtained in the exam. Then check what the mean and median of the results were.

Task 2



Based on the data frame created in the previous task, display the fourth line. Then write those students who scored above average to a separate box.



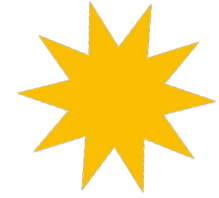
Pandas – operating on data

All the following operations for the dataframe will return values for each of the columns, we can select the column for which we want to perform these actions:

- `df.apply(function)` – applying a function to each column or cell in the set

```
print(df.apply(lambda x: x.max() - x.min()))
```

```
A    0.863928  
B    0.697781  
C    1.431593  
D    0.000000  
F    4.000000  
dtype: float64
```

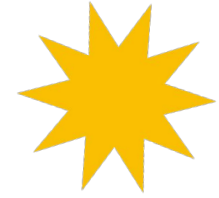


Task 3

Add a third column to the box which will contain the percentage of student scores on this exam (that is, each numerical value must be divided by the maximum score that could be obtained on this exam).

Next, we want to anonymize the first name column: we want only the first and last letters of the first name to be left. For example, for the name "John" we want to keep "J ... n"

Pandas - operating on data - grouping

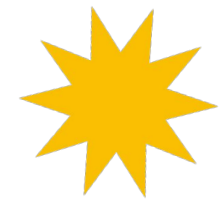


You need to segment your database from time to time. In addition to computing statistics for all values, sometimes these values can be grouped. In pandas, the *groupby* method is used for this purpose.

- `df.groupby("column").operation().column`

```
display(cars.groupby('cylinders').mean().horsepower)
```

```
cylinders
3      99.250000
4      78.281407
5      82.333333
6     101.506024
8     158.300971
Name: horsepower, dtype: float64
```



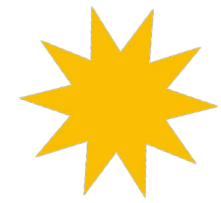
Pandas – removing data

The `df.drop()` command is used to delete data – as a parameter we give a list of indexes or columns to be deleted and `axis` – whether an index or a column should be deleted.

We can also remove incomplete data (NaNy) with the `df.dropna()` command – the `axis` parameter removes columns or rows.

```
wine = wine.drop(wine.columns[[0,3,6,8,11,12,13]], axis = 1)
wine.head()
```

	Malic acid	Ash	Magnesium	Total phenols	Nonflavanoid phenols	Color intensity	Hue
0	1.71	2.43	127	2.80	0.28	5.64	1.04
1	1.78	2.14	100	2.65	0.26	4.38	1.05
2	2.36	2.67	101	2.80	0.30	5.68	1.03
3	1.95	2.50	113	3.85	0.24	7.80	0.86
4	2.59	2.87	118	2.80	0.39	4.32	1.04

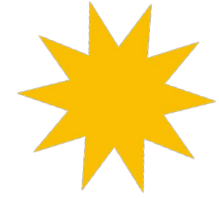


Pandas – merging data

In fact, often instead of using one large database, we connect many smaller ones (it is easier to manage them, avoid redundancy, additionally we save disk space and achieve higher speed). We can combine the data in the library in two ways:

- `df.append(object)` method adds new rows to the end of the existing dataframe
- `df.merge()` method, which in its assumptions is very similar to SQL JOIN – you can choose the joining method – inner (common), outer (sum), left, right. On parameter – the name of the column to be a hyphen.

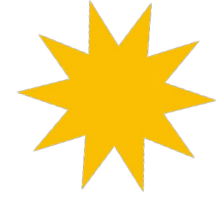
Additionally, we can give the new column name ourselves and thus add the data to the existing data: `df ["new column"] = data`



Pandas – sorting data

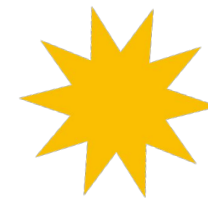
- method `df.sort_values(by="column name")`
- ascending parameter – used whether ascending or descending
- often, after sorting, we want to reset the indexes – the `df.reset_index(drop=True)` is used for this – this parameter removes the old index

Task 4



Sort the data frame by the number of points obtained for the exam, from highest to lowest. In case of the same number of points, the persons should be displayed in alphabetical order.

Task 5

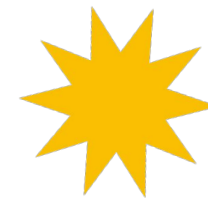


Join the two frames below.

```
student_data1 = pd.DataFrame({  
    'student_id': ['S1', 'S2', 'S3', 'S4', 'S5'],  
    'name': ['Danniella Fenton', 'Ryder Storey', 'Bryce Jensen', 'Ed Bernal', 'Kwame Morin']  
})
```

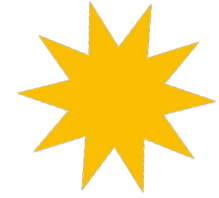
```
student_data2 = pd.DataFrame({  
    'student_id': ['S6', 'S7', 'S8', 'S9', 'S10'],  
    'name': ['Scarlette Fisher', 'Carla Williamson', 'Dante Morse', 'Kaiser William', 'Madeeha Preston']  
})
```

Task 6



To the box created in the previous task, add the information about the students' results from the box below.

```
exam_results = pd.DataFrame({  
    'student_id': ['S2', 'S10', 'S3', 'S1', 'S7', 'S9', 'S5', 'S4', 'S8', 'S6'],  
    'marks': [200, 210, 190, 222, 199, 201, 200, 198, 219, 201]  
})
```

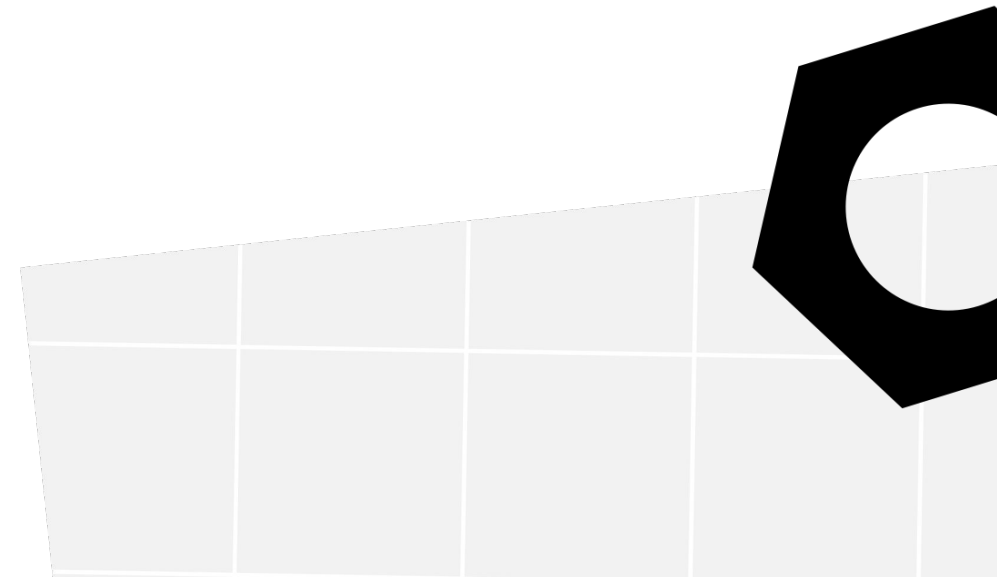


Pandas – NaNs

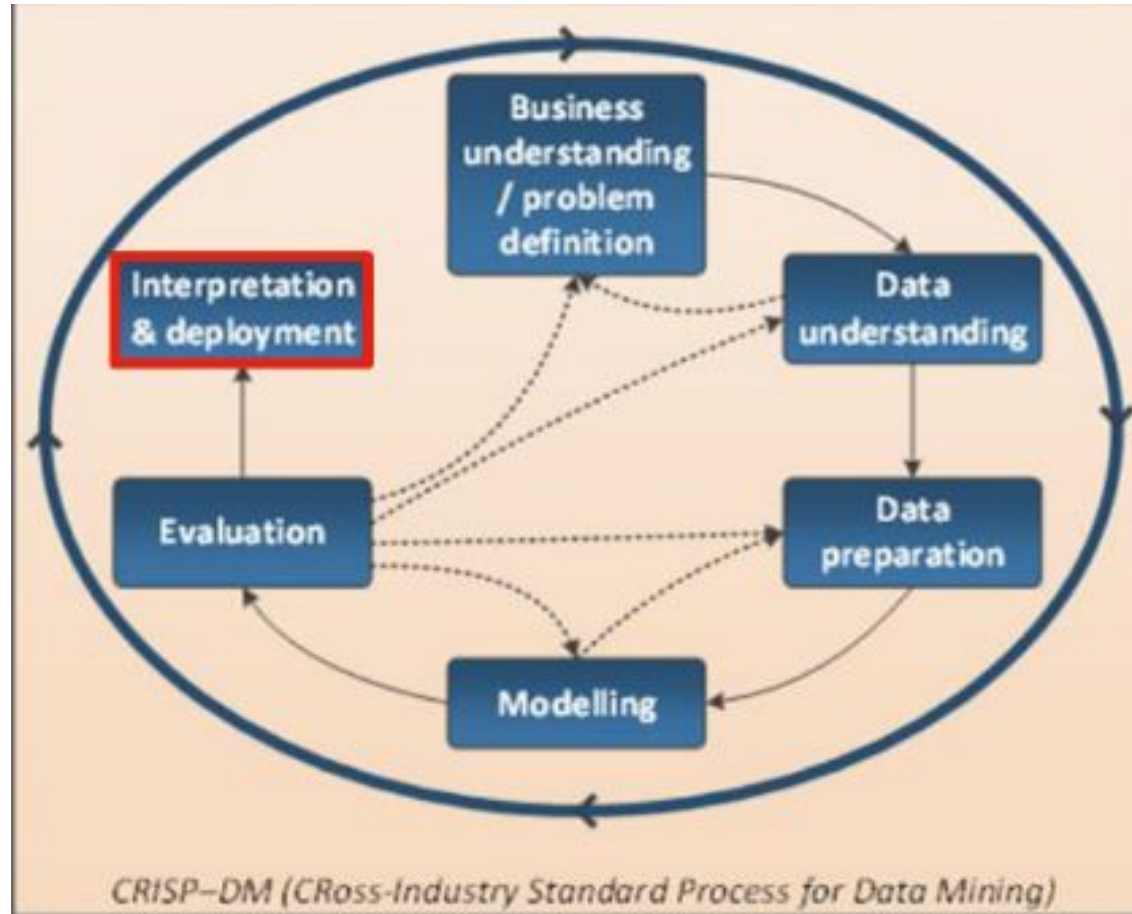
- `df.fillna(value)` method- fills the data with the specified value
- `df.dropna()` method - removes rows with empty data from the table

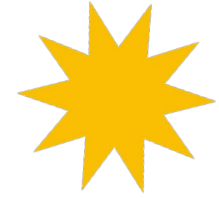


Data analysis process



Schema





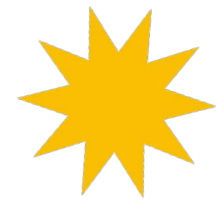
Data analysis process

Data understanding – collecting specific data needed to perform the following tasks: solve the problem, provide data description (sometimes also labeling), exploration and quality verification.

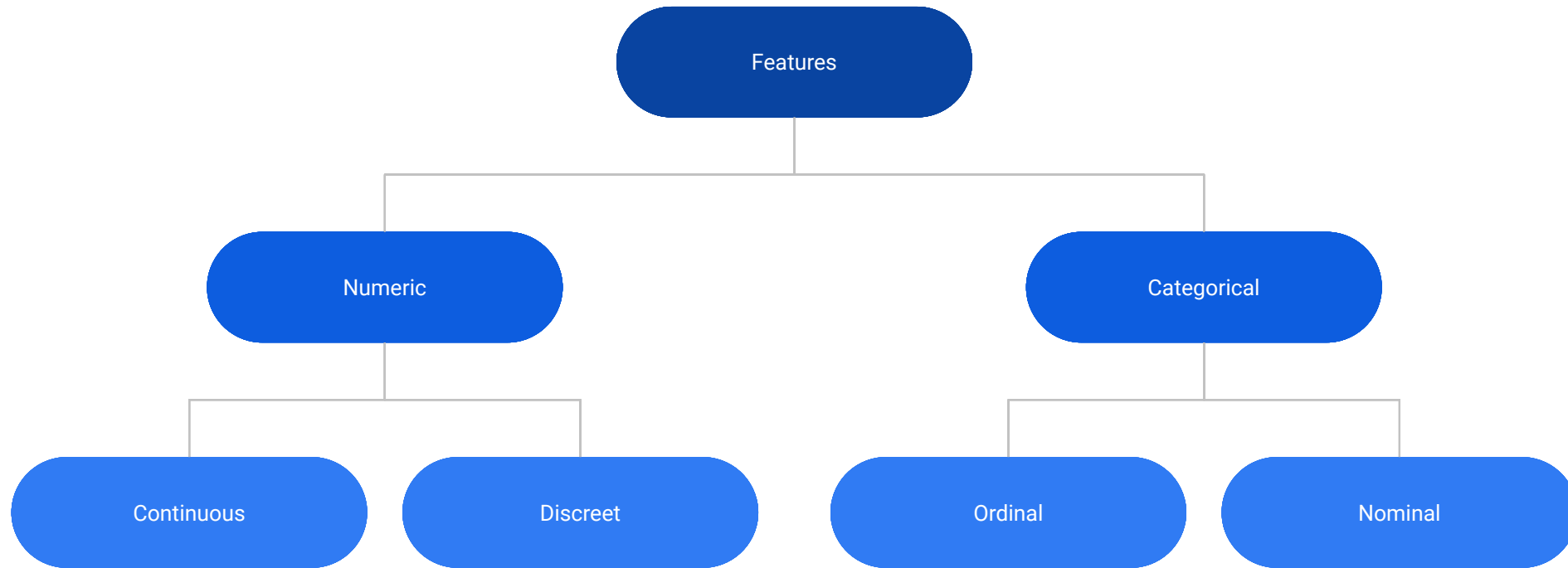
Data types – CSV sheets (these are the ones we will be working on today), text, image, sound, video, weather data, stock data, public archives, etc.

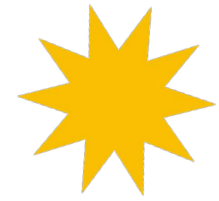
The most important task – **FEATURE EXTRACTION (feature engineering)**.

A lot depends on the size of the set and the number of features – depending on these parameters, we will take different steps in the data analysis process.



Features – classification





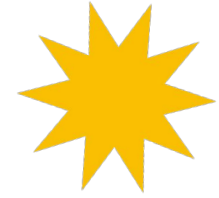
Data aggregation methods

Numeric:

- mean
- sum,
- max, min, standard deviation

Categorical:

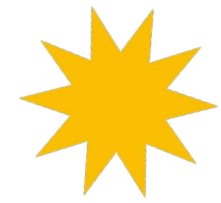
- counting the appearances
- searching for the most common
- percentage results, e.g. 23% of the population are blondes



Data preparation – incomplete values

Incomplete values – how to deal with them:

- ignore them
- insert the value "unknown"
- manually fill in on the basis of accumulated knowledge
- delete records with incomplete data
- complete algorithmically – look for the nearest neighbor, take the average, insert random values, classification task

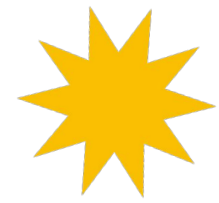


Data preparation – incomplete values

```
threshold = 0.7

#Dropping columns with missing value rate higher than threshold
data = data[data.columns[data.isnull().mean() < threshold]]

#Dropping rows with missing value rate higher than threshold
data = data.loc[data.isnull().mean(axis=1) < threshold]
```

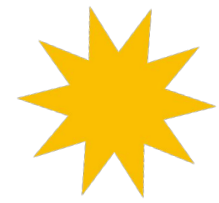


Data preparation – incomplete values

Introducing new values in place of NaNs

```
#Filling all missing values with 0  
data = data.fillna(0)
```

```
#Filling missing values with medians of the columns  
data = data.fillna(data.median())
```

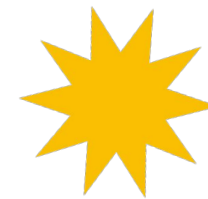


Data preparation – incomplete values

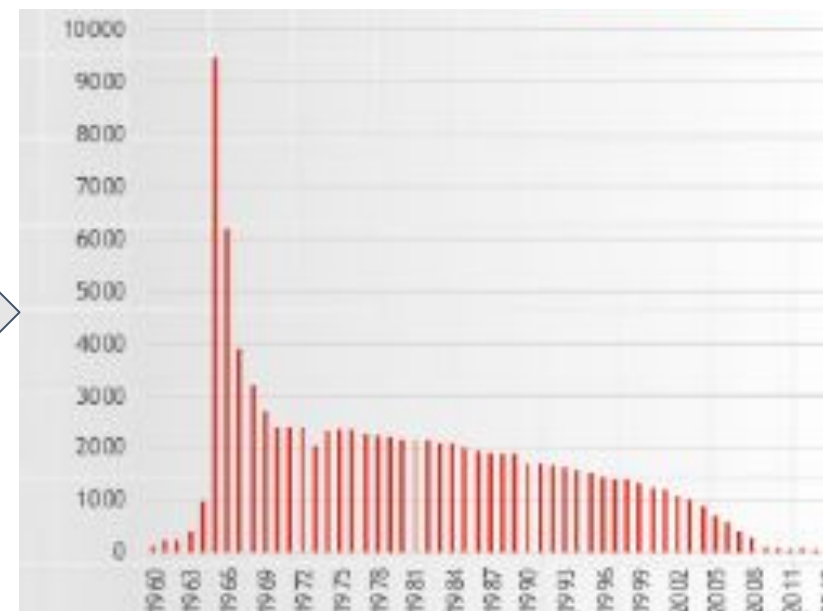
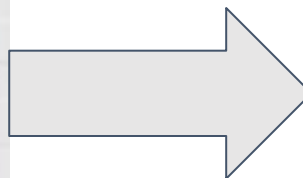
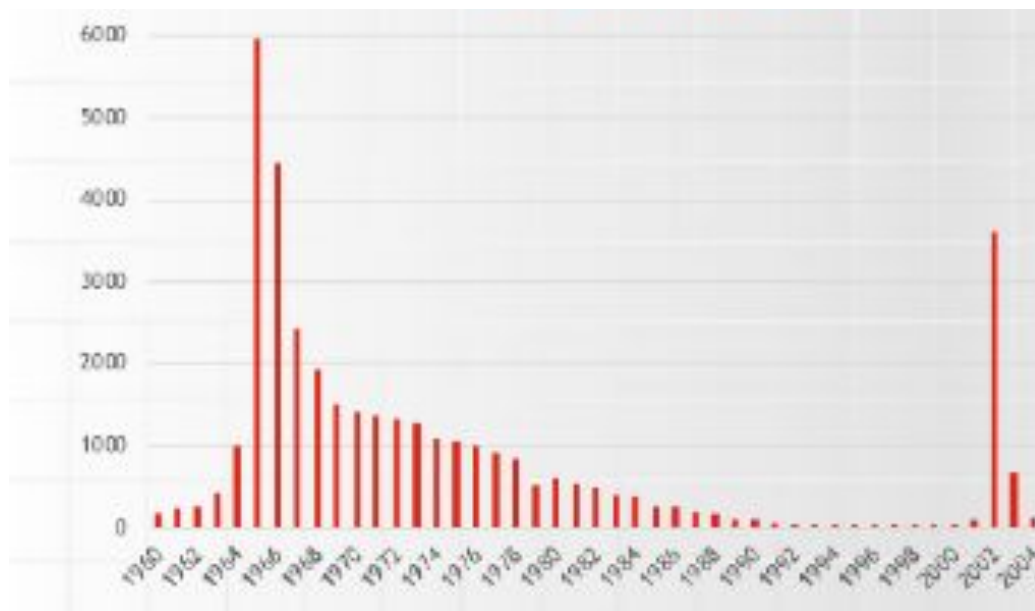
Introducing new values in place of NaNs – categorical features

```
#Max fill function for categorical columns  
data['column_name'].fillna(data['column_name'].value_counts()  
.idxmax(), inplace=True)
```

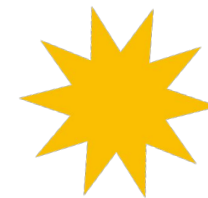
Data Preparation – Cleaning (Noise Removal)



Data cleaning – removing errors, noise, outliers



Data Preparation – Cleaning (Noise Removal)

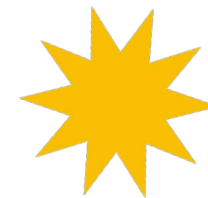


Remove outliers with standard deviation.

```
#Dropping the outlier rows with standard deviation
factor = 3
upper_lim = data['column'].mean () + data['column'].std () *
factor
lower_lim = data['column'].mean () - data['column'].std () *
factor

data = data[(data['column'] < upper_lim) & (data['column'] >
lower_lim)]
```

Data Preparation – Cleaning (Noise Removal)

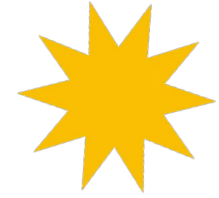


Removing outliers with percentiles.

```
#Dropping the outlier rows with Percentiles
upper_lim = data['column'].quantile(.95)
lower_lim = data['column'].quantile(.05)

data = data[(data['column'] < upper_lim) & (data['column'] >
lower_lim)]
```

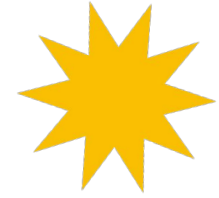

Data Preparation – Cleaning (Noise Removal)



Limiting outliers instead of deleting – small datasets (for large datasets, removing redundant information is not a problem).

```
#Capping the outlier rows with Percentiles
upper_lim = data['column'].quantile(.95)
lower_lim = data['column'].quantile(.05)

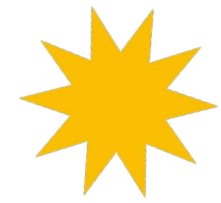
data.loc[(df[column] > upper_lim), column] = upper_lim
data.loc[(df[column] < lower_lim), column] = lower_lim
```



Data preparation – standardization

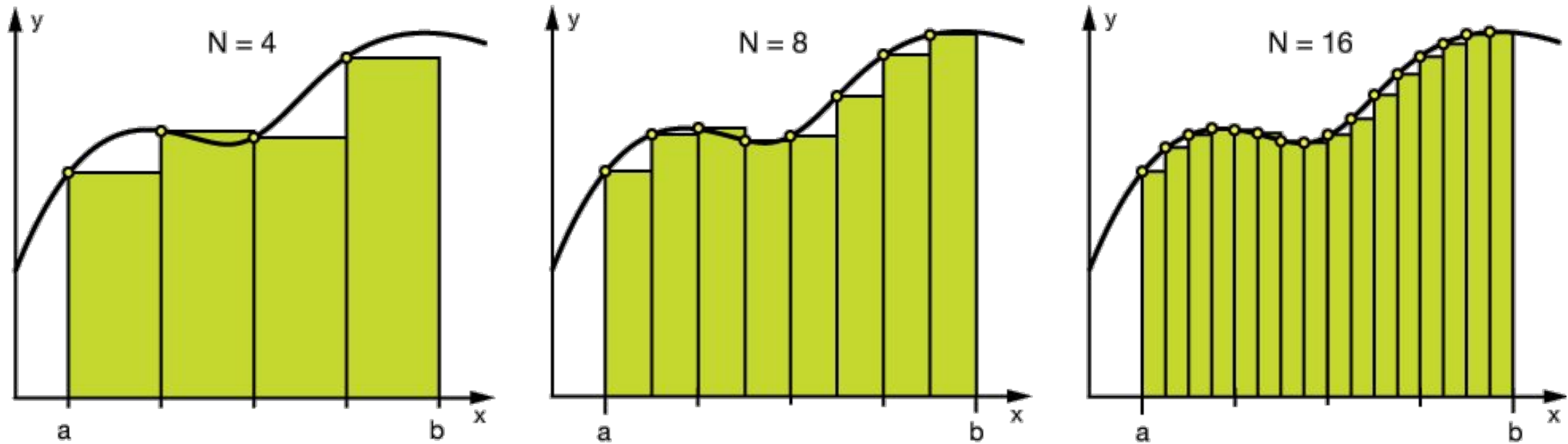
Standardization consists in standardizing the various data formats in a set:

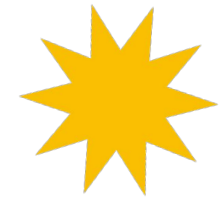
- language coding style: UTF-8, ANSI etc.,
- Dates: Sep 12, 2019, Sep 12, 2019, 120919 etc.,
- New York City, NY, Z. New York,
- MIT, Massachusetts Institute of Technology
- Unrecognizable signs.



Data preparation – discretization

Continuous data distribution is bad for most machine learning algorithms.





Data preparation – discretization

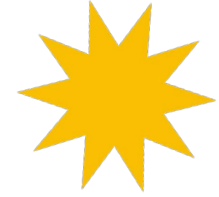
#Numerical Binning Example

Value		Bin
0-30	->	Low
31-70	->	Mid
71-100	->	High

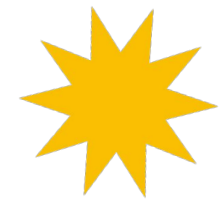
#Categorical Binning Example

Value		Bin
Spain	->	Europe
Italy	->	Europe
Chile	->	South America
Brazil	->	South America

Data preparation – logarithmic transformation



- helps to deal with skewed data
- organizes the size of the data
- reduces the impact of outliers
- increases the reliability of the model
- negative values need to be dealt with



Data preparation – logarithmic transformation

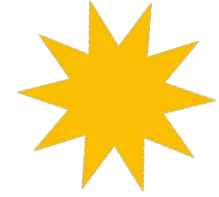
```
#Log Transform Example
data = pd.DataFrame({'value':[2,45, -23, 85, 28, 2, 35, -12]})

data['log+1'] = (data['value']+1).transform(np.log)

#Negative Values Handling
#Note that the values are different
data['log'] = (data['value']-data['value'].min()+1)
               .transform(np.log)
```

	value	log(x+1)	log(x-min(x)+1)
0	2	1.09861	3.25810
1	45	3.82864	4.23411
2	-23	nan	0.00000
3	85	4.45435	4.69135
4	28	3.36730	3.95124
5	2	1.09861	3.25810
6	35	3.58352	4.07754
7	-12	nan	2.48491

Data preparation - One - hot encoding

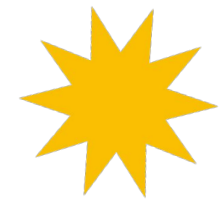


Encoding 1 from n

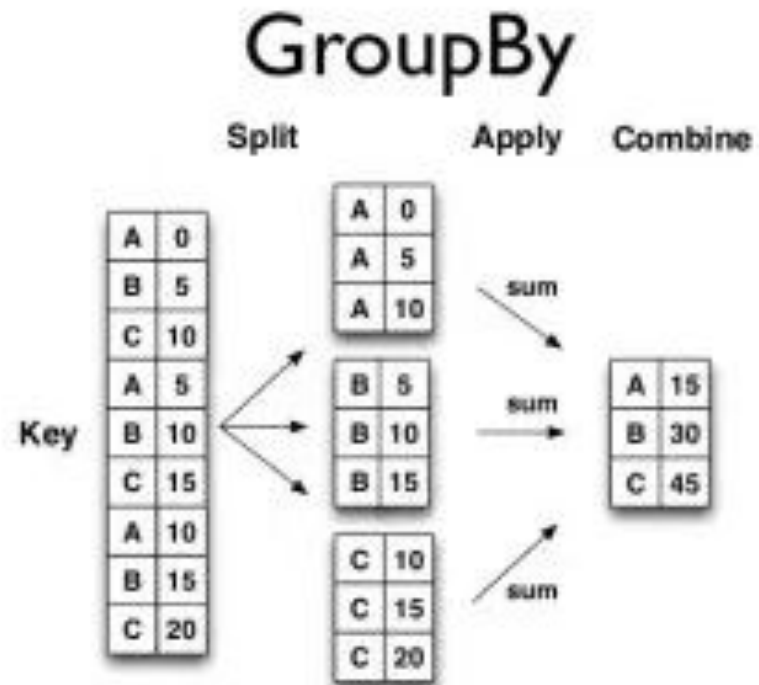
User	City
1	Roma
2	Madrid
1	Madrid
3	Istanbul
2	Istanbul
1	Istanbul
1	Roma



User	Istanbul	Madrid
1	0	0
2	0	1
1	0	1
3	1	0
2	1	0
1	1	0
1	0	0



Data preparation – grouping

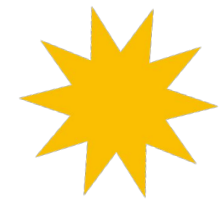


```
display(cars.groupby('cylinders').mean().horsepower)
display(cars.groupby('modelyear').max()[['acceleration']])
```

```
cylinders
3      99.250000
4      78.281407
5      82.333333
6     101.506024
8     158.300971
Name: horsepower, dtype: float64
```

acceleration

```
modelyear
70      20.5
71      20.5
72      23.5
73      21.0
74      21.0
```

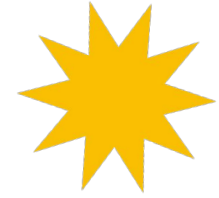
Data preparation – splitting

Useful when you have multiple traits in one column.

```
data.name
0  Luther N. Gonzalez
1   Charles M. Young
2     Terry Lawson
3   Kristen White
4   Thomas Logsdon

#Extracting first names
data.name.str.split(" ").map(lambda x: x[0])
0      Luther
1    Charles
2     Terry
3    Kristen
4     Thomas

#Extracting last names
data.name.str.split(" ").map(lambda x: x[-1])
0    Gonzalez
1     Young
2    Lawson
3     White
4    Logsdon
```

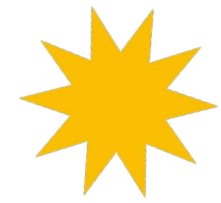


Data preparation – scaling

In most cases, the numerical characteristics of a dataset are not specific and vary from one another. In fact, it doesn't make sense to expect the age and income columns to have the same range. But from a machine learning perspective, the same scope helps improve the model.

Scaling solves this problem. Continuous functions become range identical after the scaling process. This process is not mandatory for many algorithms, but still worthwhile. However, distance-based algorithms such as k-NN or k-Means must have scaled continuous functions as model inputs.

Basically, there are two common ways to scale: **normalization and standardization**.

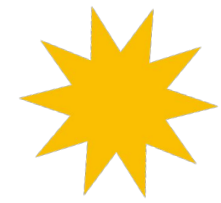


Data preparation – normalization

$$X_{norm} = \frac{X - X_{min}}{X_{max} - X_{min}}$$

```
data = pd.DataFrame({'value':[2,45, -23, 85, 28, 2, 35, -12]})  
  
data['normalized'] = (data['value'] - data['value'].min()) /  
(data['value'].max() - data['value'].min())
```

	value	normalized
0	2	0.23
1	45	0.63
2	-23	0.00
3	85	1.00
4	28	0.47
5	2	0.23
6	35	0.54
7	-12	0.10

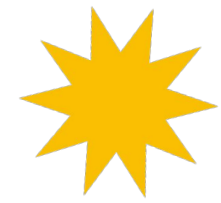


Data preparation – standardization

$$z = \frac{x - \mu}{\sigma}$$

```
data = pd.DataFrame({'value':[2,45, -23, 85, 28, 2, 35, -12]})  
  
data['standardized'] = (data['value'] - data['value'].mean()) /  
data['value'].std()
```

	value	standardized
0	2	-0.52
1	45	0.70
2	-23	-1.23
3	85	1.84
4	28	0.22
5	2	-0.52
6	35	0.42
7	-12	-0.92



Data Preparation – Date Extraction

```
from datetime import date

data = pd.DataFrame({'date':
['01-01-2017',
'04-12-2008',
'23-06-1988',
'25-08-1999',
'20-02-1993',
]})

#Transform string to date
data['date'] = pd.to_datetime(data.date, format="%d-%m-%Y")

#Extracting Year
data['year'] = data['date'].dt.year

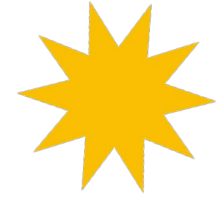
#Extracting Month
data['month'] = data['date'].dt.month

#Extracting passed years since the date
data['passed_years'] = date.today().year - data['date'].dt.year

#Extracting passed months since the date
data['passed_months'] = (date.today().year - data['date'].dt.year)
* 12 + date.today().month - data['date'].dt.month

#Extracting the weekday name of the date
data['day_name'] = data['date'].dt.day_name()
```

	date	year	month	passed_years	passed_months	day_name
0	2017-01-01	2017	1	2	26	Sunday
1	2008-12-04	2008	12	11	123	Thursday
2	1988-06-23	1988	6	31	369	Thursday
3	1999-08-25	1999	8	20	235	Wednesday
4	1993-02-20	1993	2	26	313	Saturday

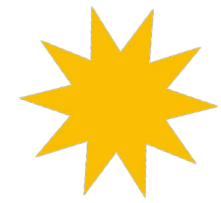


Feature engineering – example

How to assess whether a given profile on Facebook is not a bot?

- number of friends
- number of posts published
- number of photos
- number of likes
- likes under posts
- the length of the posts
- the origin of the phone number
- e-mail address domain

And the rating of the attractiveness of a movie on YouTube?

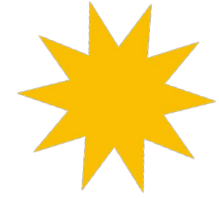


Feature engineering – realistic task

The car rental company ARRA wants to evaluate the reliability and maintenance costs of the different car models available in the fleet. It has the following data:

- a table with individual activities of all cars in the fleet: date, place of rental, status (driving, breakdown), renting data (age, sex, nationality), number of kilometers traveled, amount of fuel consumed, car identification number
- a table for each car: year of production, engine, equipment, brand, model, etc.

What features can we find here?

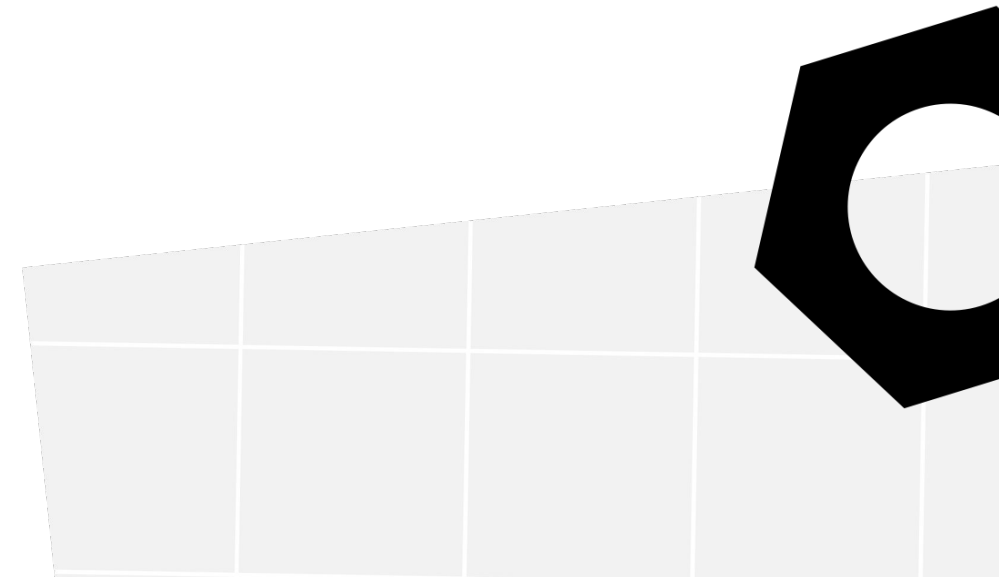


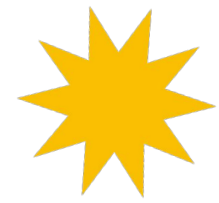
Fun fact

The process of obtaining, preprocessing data and extracting features from them takes up to **80% of a data engineer's work time.**



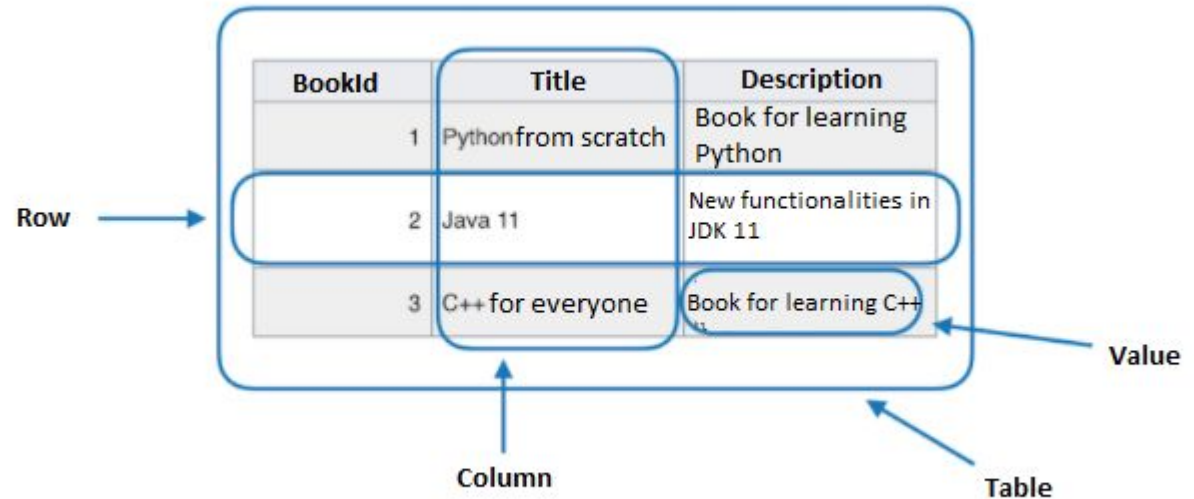
CONCEPT OF RELATIONAL DATABASES

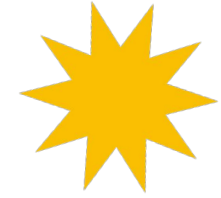




TABLE

- According to the relational concept of databases, it is a place to store data.
- The table consists of rows and records.
- Each line describes one record (single information/data set).
- The column is responsible for describing the properties / structure of a given record.



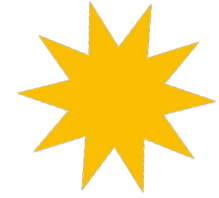


PRIMARY KEY

In order to uniquely identify records for the purposes of database tables, an identifier is introduced, which is referred to as the **primary key**.

Name	Surname	Birth date
John	Kowalsky	31-03-1987
Wally	Novak	12-08-1968
Richard	Johnson	13-07-1955
Wally	Novak	12-08-1968

Id	Name	Surname	Birth date
1	John	Kowalsky	31-03-1989
2	Wally	Novak	12-08-1968
3	Richart	Johnson	13-07-1955
4	Wally	Novak	12-08-1968



PRIMARY KEY

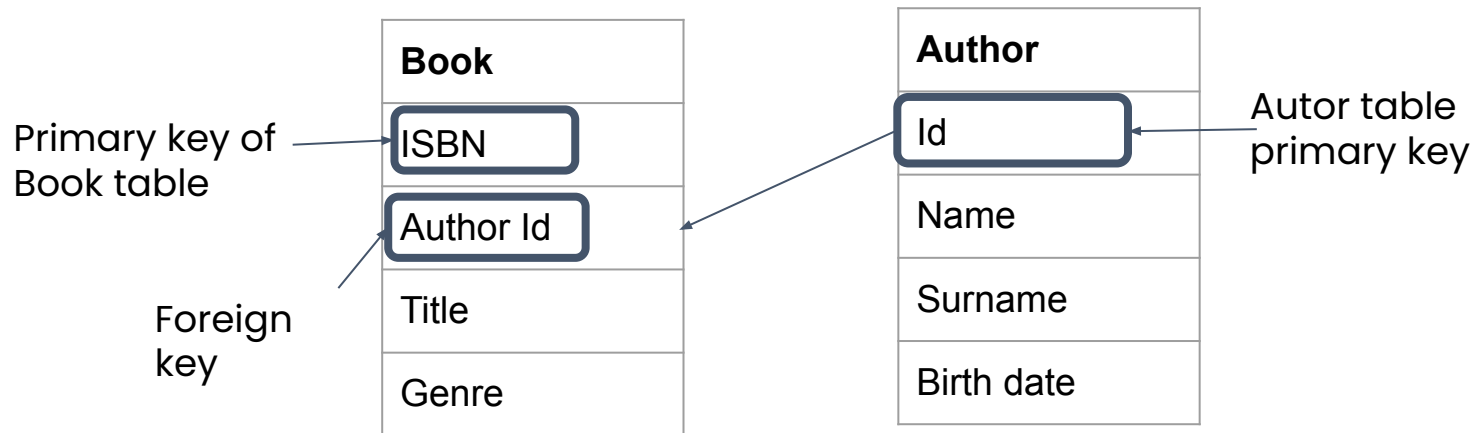
It can be a real (or natural) identifier, such as a identification number. However, many times an abstract identifier is implemented in databases, which has nothing to do with the actual data structure that the database is to store.

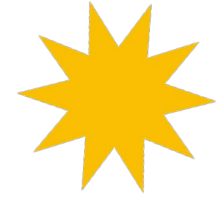
BookId	Title	Description
1	Python from scratch	Book for learning Python
2	Java 11	New functionalities in JDK 1.11
3	C++ for everyone	Book for learning C++

↑
PRIMARY KEY

RELATIONS

In relational database systems, tables are combined in the so-called relations that define the type of dependency between two data carriers, e.g. a book and an author. This binding is performed with the use of a **foreign key**.





TYPES OF RELATIONS

According to the theoretical relational database model, there are three types of relationships:

- one-to-one
- one to many
- many to many



1:1 RELATION

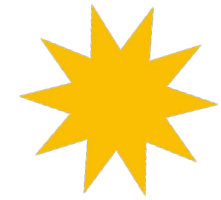
The type of relationship whereby one record of table A can only be associated with one record of table B.

In the example below, we deal with a situation where one record from the Person table corresponds to one record from the department table.

id	Student Id	Department	Name
1	2234	Computer Studies	Software Engineering
2	3489	Biology	Microbiology



Index number	Name	Surname	Birth date
2234	Thomas	Novak	12-09-1987
2256	Wally	Kowalsky	31-04-1988
3489	Thomas	Novak	12-09-1987



1:N RELATION

The type of relationship whereby one record in table A can be associated with multiple records in table B.

In the above example, we are dealing with a situation where one customer may place many different orders, while one order may be assigned only to one customer.

Id	Customer Id	Order name	Order date
1	1	Macbook Pro	12-03-2020
2	1	Iphone 11	14-03-2020
3	2	Dell XPS	15-03-2020

Id	Name	Surname
1	John	Kowalsky
2	Wally	Kowalsky
3	Thomas	Novak

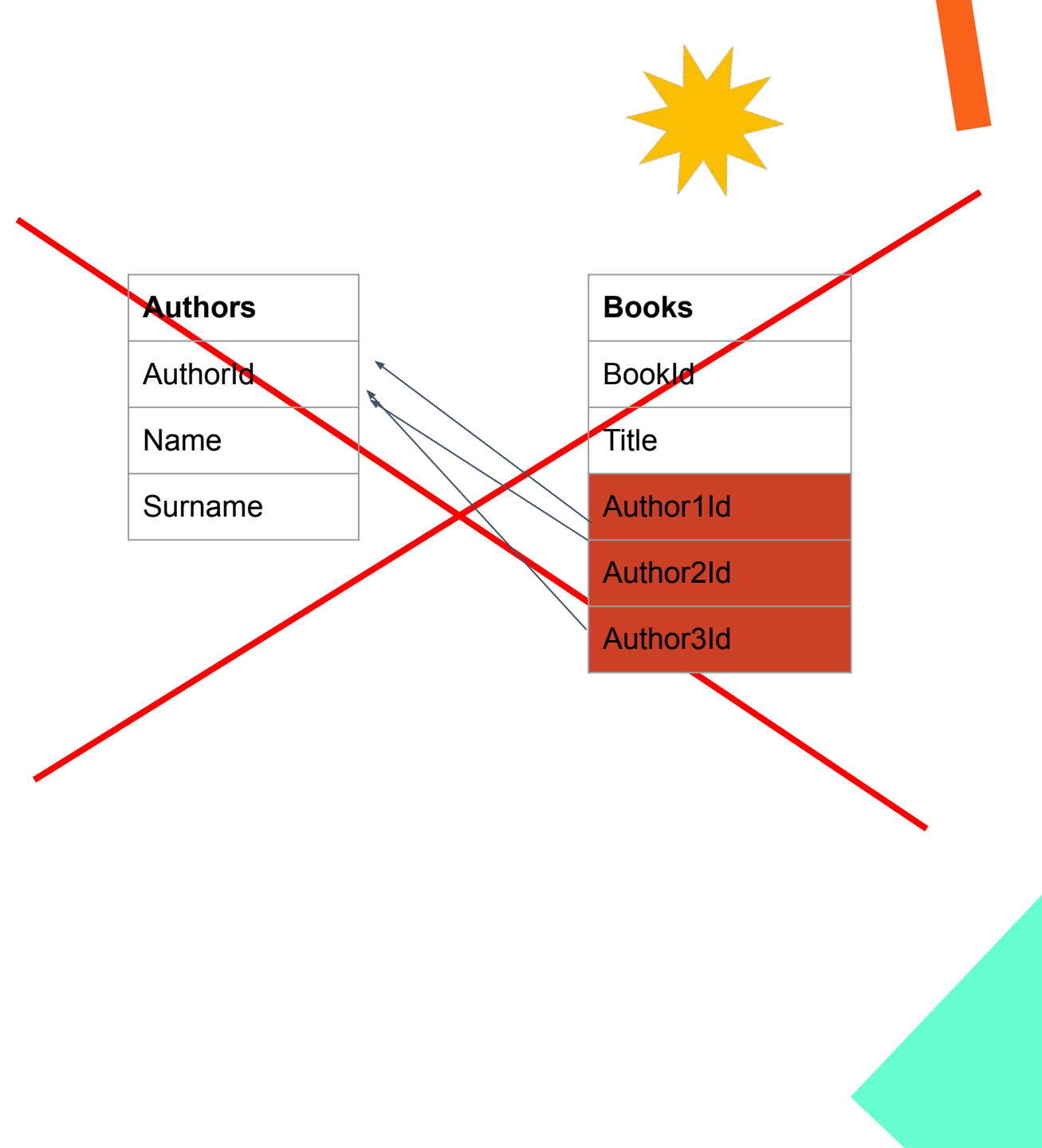


N:M RELATION

The type of relationship whereby one record in table A can be associated with multiple records in table B.

An example is an **erroneous** realization of this relationship ->

This solution may work correctly under several conditions, but if we wanted to include more than three authors of the book, the above example would not implement the target functionality.



N:M RELATION

In practice, many-to-many relationships are realized using an additional table, the so-called auxiliary table.

In the above example, a new table of Authors was created (its name is completely arbitrary), which is responsible for linking the Authors table and the Books table into the relationship.

In this case, the BookId column points to the primary key of the Books table, and the AuthorId column points to the AuthorId column.

BookId	Title
1	Python from scratch
2	Java 11
3	C++ for everyone

AuthorId	Name	Surname
1	John	Kowalsky
2	Wally	Kowalsky
3	Thomas	Novak

BookId	AuthorId
1	1
1	2
2	3
3	1

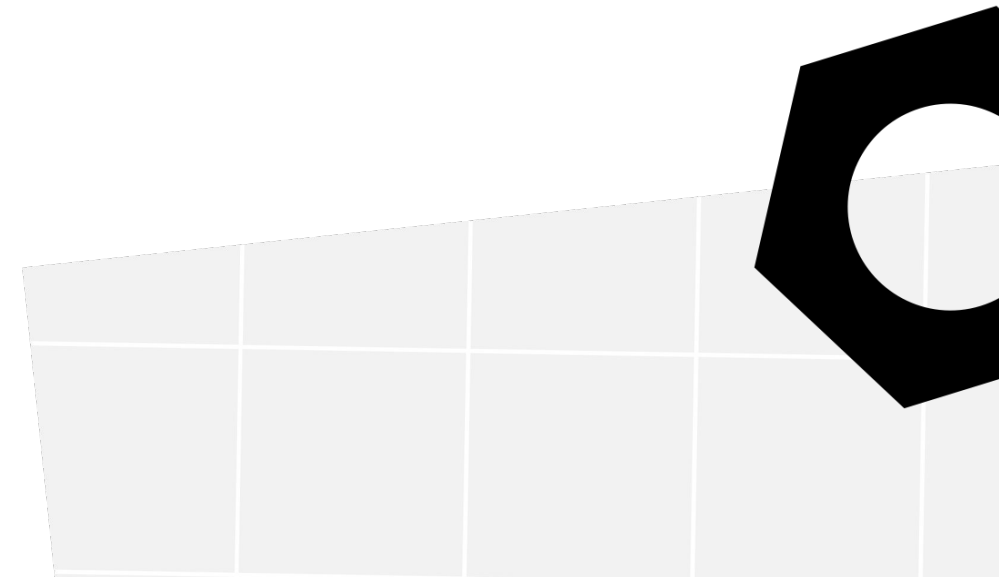
Authors
AuthorId
Name
Surname

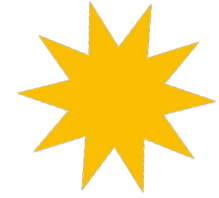
BooksAuthors
BookId
AuthorId

Books
BookId
Title



PRINCIPLES OF DESIGNING TABLES

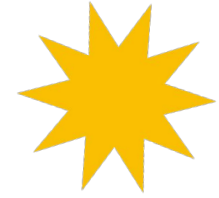




Definition of the goal

The structure of the proposed database should be selected according to the specifics of a particular system. The modeled system will be modeled differently, in which personal data are its important point (preparation of separate tables for persons and addresses along with the relationship), and the system in which personal data has a secondary function and is not an important point of the system will be visualized differently (a common table connecting personal data and addresses).

Avoiding duplicate data



For example, some information, such as the manufacturer's name and the manufacturer's address, are repeated many times in the database. Firstly, this is a waste of memory and, in addition, any mistake will result in inconsistent data.

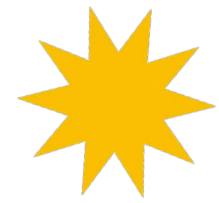
It is absolutely necessary to avoid duplications of the same data in the following lines.

Id	Name	Producer name	Producer address
1	Macbook Pro 16	Apple	Cupertino, CA 95014, US
2	Iphone 11 Pro	Apple	Cupertino, CA 95014, US
3	Dell XPS 15	Dell	The Boulevard, Cain Road, Bracknell, Berkshire, RG12 1LF

In this case, it is much more efficient to break a given table into two and enter a specific type of relationship, e.g .:

Id	Name	Print Id
1	Macbook Pro 16	1
2	Iphone 11 Pro	1
3	Dell XPS 15	3

Print Id	Producer name	Producer address
1	Apple	Cupertino, CA 95014, US
3	Dell	The Boulevard, Cain Road, Bracknell, Berkshire, RG12 1LF

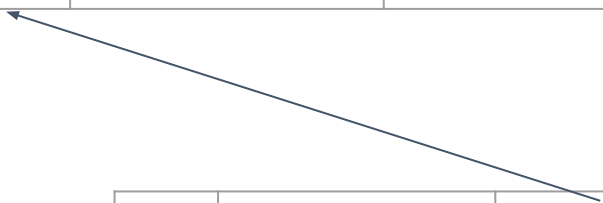


The atomicity of information

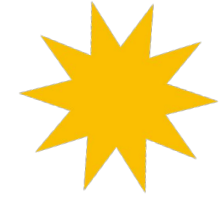
Each field in the database should contain a single piece of information (ie, atomic). Atomicity is system specific, but in short, it is about minimizing the amount of information stored in a given field.

For the example above, the manufacturer's address contains a large amount of information and, for example, searching for a specific product by address can be very difficult. For this purpose, the manufacturer's address should be broken down into information such as Country or Region.

Print Id	Producer name	Producer address
1	Apple	Cupertino, CA 95014, US
3	Dell	The Boulevard, Cain Road, Bracknell, Berkshire, RG12 1LF



Id	Name	Print Id
1	Macbook Pro 16	1
2	Iphone 11 Pro	1
3	Dell XPS 15	3



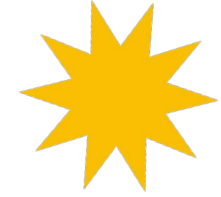
Multiplicating references

For example, having a situation where a customer rents a lot of computer equipment, he can implement these assumptions using the table ->

In the example above, the ProductId field contains information about many products. However, this form is **incorrect** due to the great difficulty in obtaining all the necessary statistical data, e.g. extracting information on the number of times a given product has been borrowed. The correct table structure should look like this ->

CustomerId	ProductId	Date
1	3, 6, 90	14-03-2020
2	5	16-03-2020
3	4, 11, 15	17-08-2020

CustomerId	ProductId	Date
1	3	14-03-2020
1	6	14-03-2020
1	90	14-03-2020
2	5	16-03-2020
3	4	17-08-2020
3	11	17-08-2020
3	15	17-08-2020




Avoidance of empty fields

In tables, you should avoid leaving empty fields (with no data). In certain specific situations, however, blank spaces may be unavoidable. ->

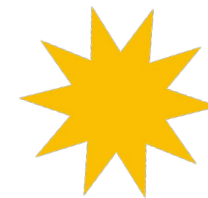
In the above case, 75% of the records have empty the Remarks field. For the above case, it is definitely better to introduce an additional table that will be responsible for storing additional information and reducing empty fields.

OrderId	CustomerId	ProductId	Date	Notes
1	1	3	14-03-2020	
2	1	6	14-03-2020	pickup in person only after 29-03-2020
3	1	90	14-03-2020	
4	2	5	16-03-2020	

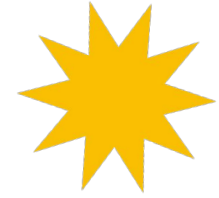


OrderId	Notes
2	pickup in person only after 29-03-2020

Unique record identifiers



Each record of the table should be **uniquely identified**. Otherwise, it will not be possible to distinguish between them. The primary key must be well identified. Sometimes it is abstract form may be replaced with a real identifier, eg ISBN for books, PESEL for Polish citizens. It all depends primarily on the context of the modeled system.

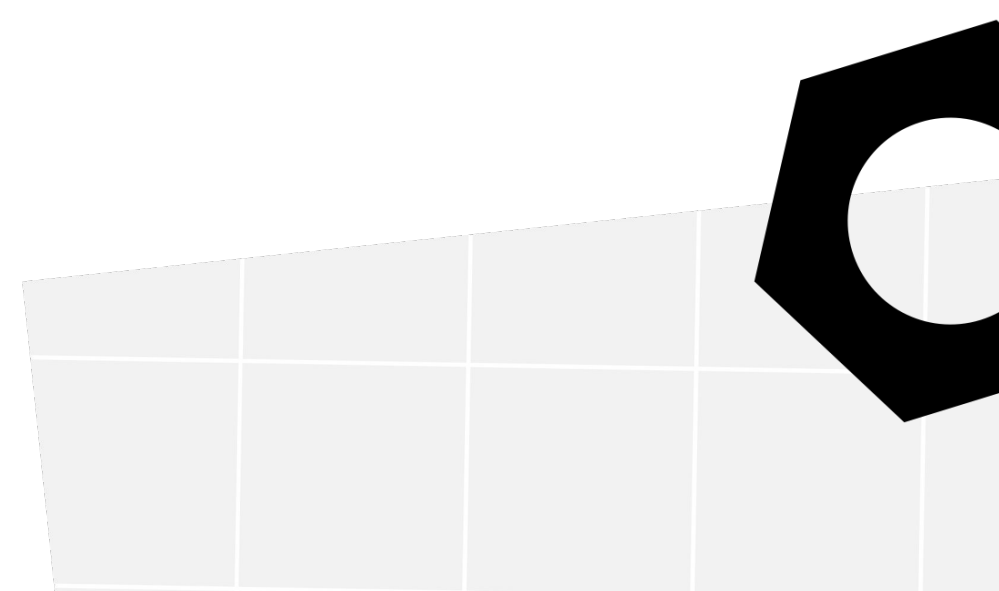


Task 7

1. What tables will we need to create a skeleton database for car rental management.
2. What relationship should there be between the car, customer and booking tables?
3. What could be the primary key in each table?

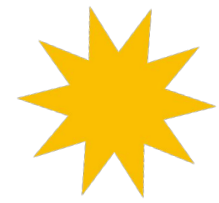


SQL



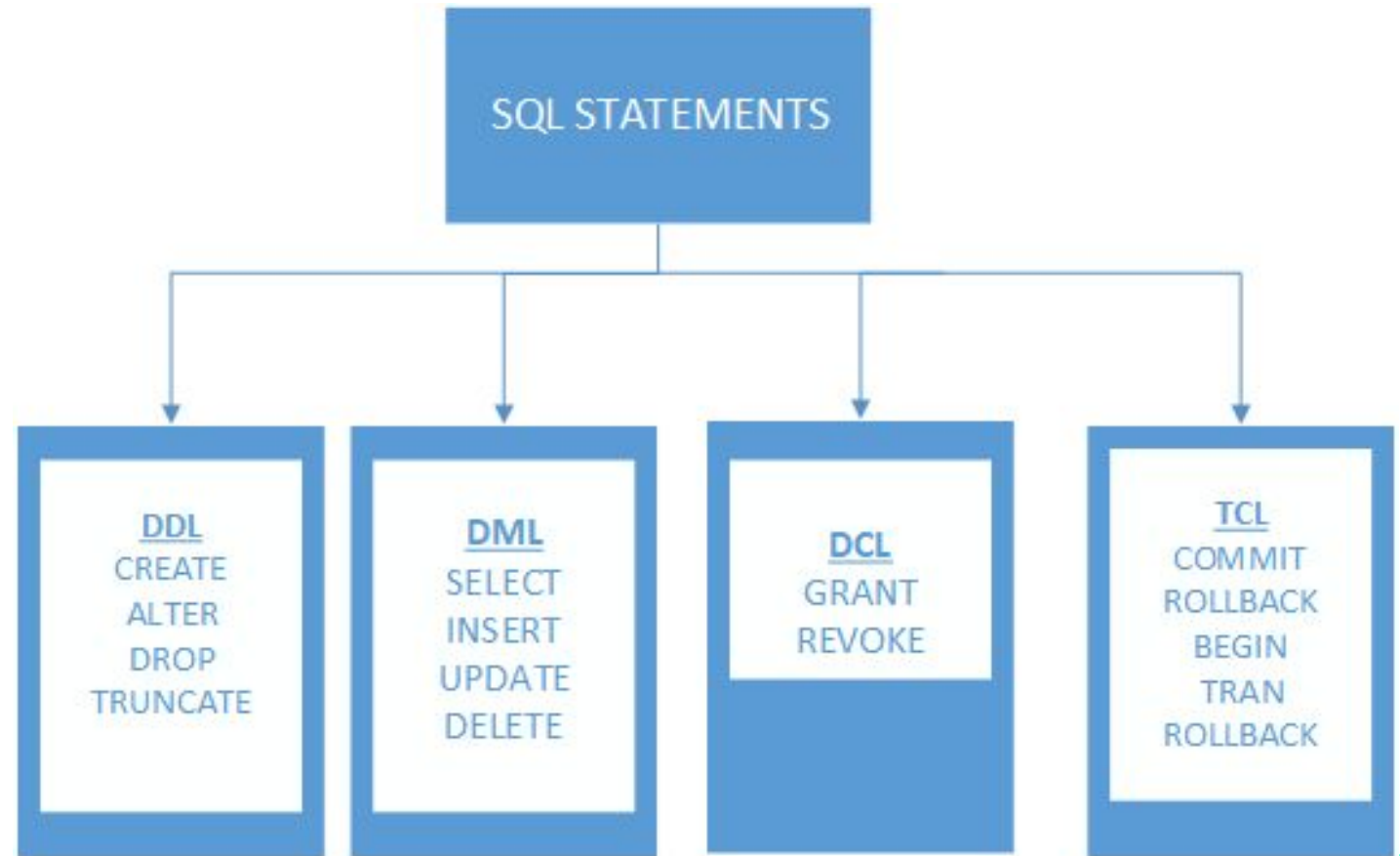
SQL is a structured query language that enables operations to be performed on relational databases. It is a universal language used, among others, in MySQL, PostgreSQL, Microsoft Server SQL and many others.



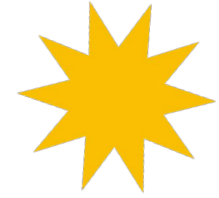


SQL – Types of tasks

- DDL (Data Definition Language) – implements data structure definition
- DML (Data Manipulation Language) – performs data retrieval and modification
- DCL (Data Control Language) – data access control language
- TCL (Transaction Control Language) – allows you to manage DML query groups.



DATA TYPES



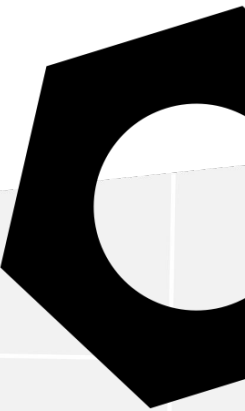
Each column of a database table has a defined type that determines the kind of data that can be stored in a given cell.

We distinguish:

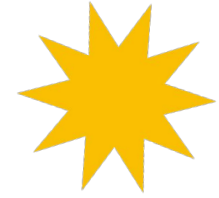
- **numeric types: INT, FLOAT, DOUBLE**
- **string types: VARCHAR, TEXT**
- **date and time: DATE, TIME**



DDL – DATA DEFINITION LANGUAGE



CREATE



```
CREATE TABLE table_name(  
    column_name_1 column_type_1 [attributes],  
    column_name_2 column_type_2 [attributes],  
    column_name_3 column_type_3 [attributes]  
    ...  
);
```

```
CREATE TABLE Product(  
    id INTEGER,  
    Name VARCHAR(20),  
    Producer VARCHAR(25)  
);
```

SQL command to create tables ->

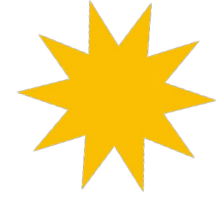
Table names in the basic version may contain:

- letters
- numbers
- \$, _ characters
- characters with codes: U + 0080-U + FFFF
- upper and lower case letters - however, whether their interpretation will be: case sensitive or incase sensitive depends on the server and database configuration.

However, table names cannot:

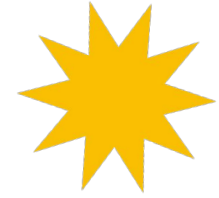
- consist of only numbers
- consist of only SQL keywords (unless they are enclosed in single quotes)
- end with a space.

COLUMN ATTRIBUTES (CONSTRAINTS)



Tables created with the SQL command can declare various attributes (constraints) for the columns being defined.

- PRIMARY KEY
- NOT NULL
- AUTO_INCREMENT
- DEFAULT
- INDEX
- UNIQUE



SHOW

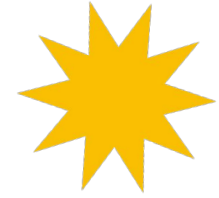
With the SHOW command it is possible to return information about the structure of a given table:

```
SHOW COLUMNS FROM table  
[FROM database]  
[LIKE `table_template`];
```

```
SHOW COLUMNS FROM Product.sda;
```

```
SHOW COLUMNS FROM Product.sda  
LIKE 'name%';
```

Instead of using SHOW COLUMNS, we can use the DESCRIBE command, which, however, has fewer options for selecting details.



ALTER

With the ALTER command it is possible to change the structure of an existing table:

This instruction allows:

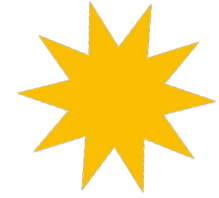
- adding and removing columns
- modifying types, names
- adding indexes
- removing indexes

The FIRST and AFTER keywords allow you to place a column in a specific table structure, e.g .:

```
ALTER TABLE table_name change1[,  
change2[, ...]];
```

```
ALTER TABLE table_name  
ADD [COLUMN] column_definition  
[FIRST | AFTER column_name];
```

```
ALTER TABLE Product  
ADD product_description VARCHAR(255)  
AFTER product_price;
```



DROP

Deleting a table is performed using the DROP instruction.

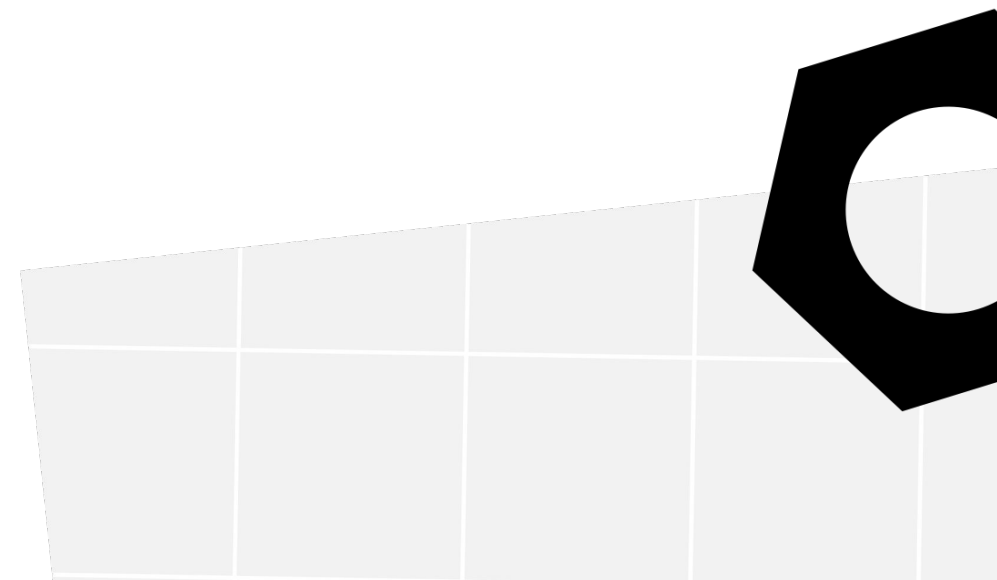
```
DROP TABLE [IF EXISTS] table1, table2, ..., tableN;
```

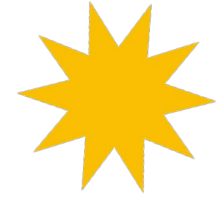
```
DROP TABLE Product;
```

You can delete one table or the entire set of tables, as long as the table does exist.



FOREIGN KEYS





FOREIGN KEY CREATION

To join tables into relationships, you must specify a foreign key in the tables being joined. This can be done when creating the table:

The foreign key can also be entered in an existing table.

```
CREATE TABLE Product
(
    Id INTEGER PRIMARY KEY,
    Name VARCHAR(20),
    ProducerId INTEGER,
    CONSTRAINT producerId_fk FOREIGN KEY
(ProducerId) REFERENCES Producer(Id)
);
```

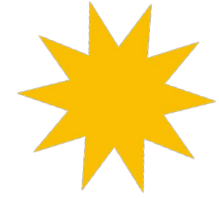
```
ALTER TABLE Product ADD CONSTRAINT
producer_fk FOREIGN KEY (ProducerId)
REFERENCES Producer(Id);
```



FOREIGN KEY REMOVAL

The foreign key can be deleted with the command below:

```
ALTER TABLE Product DROP FOREIGN KEY  
producer_fk;
```



Task 8

1. Create the car_rental database and make sure it is currently the default database for operations. (USE car_rental)
2. Add the appropriate tables containing:
 - a. cars – car_id, producer, model, year, horse_power, price_per_day
 - b. clients – client_id, name, surname, address, city
 - c. bookings – booking_id, client_id, car_id, start_date, end_date, total_amount
3. *Add auto-increment for keys.
4. *Update the bookings table with the two foreign keys it has.

Answer

1)

```
CREATE DATABASE car_rental;  
USE car_rental;
```

2)

```
CREATE TABLE cars  
(  
    car_id INTEGER PRIMARY KEY,  
    producer VARCHAR(30),  
    model VARCHAR(30),  
    year INTEGER,  
    horse_power INTEGER,  
    price_per_day INTEGER  
);
```

CREATE TABLE clients

```
(  
    client_id INTEGER PRIMARY KEY,  
    name VARCHAR(30),  
    surname VARCHAR(30),  
    address TEXT,  
    city VARCHAR(30)  
);
```

CREATE TABLE bookings

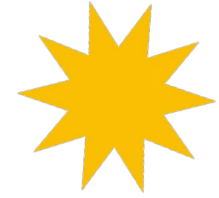
```
(  
    booking_id INTEGER PRIMARY KEY,  
    client_id INTEGER,  
    car_id INTEGER,  
    start_date DATE,  
    end_date DATE,  
    total_amount INTEGER  
);
```

3)

```
ALTER TABLE clients MODIFY COLUMN client_id INTEGER  
AUTO_INCREMENT;  
ALTER TABLE cars MODIFY COLUMN car_id INTEGER  
AUTO_INCREMENT;  
ALTER TABLE bookings MODIFY COLUMN booking_id INTEGER  
AUTO_INCREMENT;
```

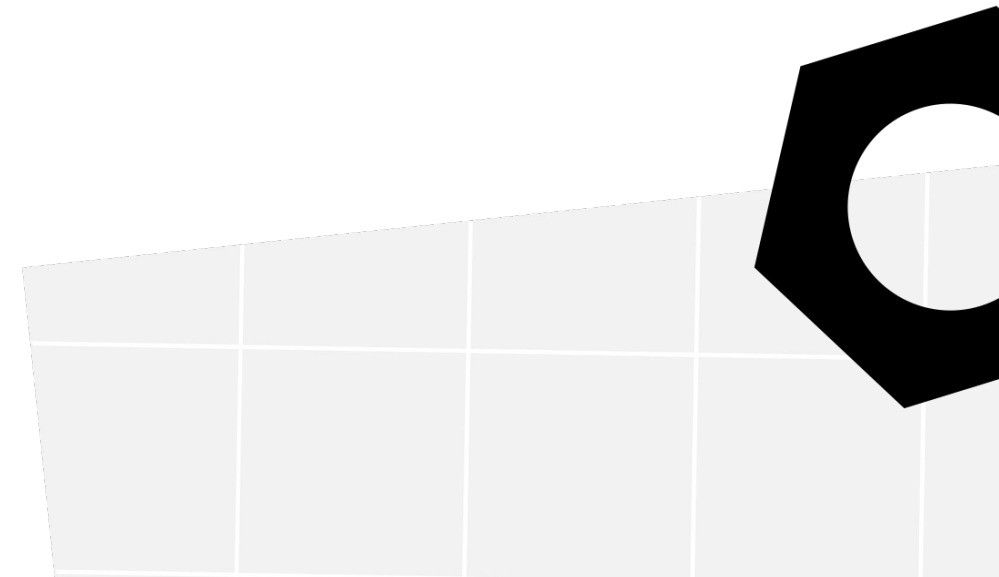
4)

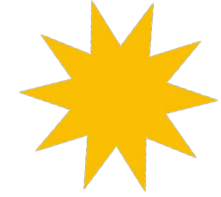
```
ALTER TABLE bookings  
ADD CONSTRAINT client_id_fk FOREIGN KEY (client_id)  
REFERENCES clients (client_id),  
ADD CONSTRAINT car_id_fk FOREIGN KEY (car_id) REFERENCES  
cars (car_id);
```





DATA MODIFICATION LANGUAGE





INSERT

By means of the INSERT statement, it is possible to enter new records into the created tables.

This command allows you to enter a new row in the table, where each value corresponds to a column with an analogous position:

value1 -> column1

value2 -> column2

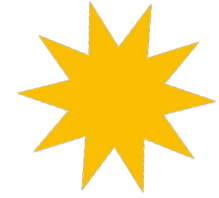
valueN -> column3

Data can also be entered without specifying column names, e.g.:

```
INSERT INTO tabela [(column1, column2, ...,  
columnN)] VALUES (value1, value2, ...,  
valueN);
```

```
INSERT INTO Produkt (ProductId, Name,  
Description)  
VALUES (1, 'Macbook Pro 16', 'Late 2019');
```

```
INSERT INTO Product  
VALUES (1, 'Macbook Pro 16', 'Late 2019');
```

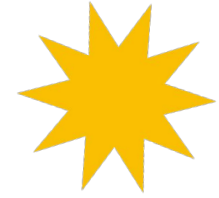


UPDATE

The UPDATE statement allows you to update records that have already been entered.

```
UPDATE table SET column1=value1, column2=value2, ...,  
columnN=valueN [WHERE condition];
```

```
UPDATE Product SET Name='Macbook Pro' where Name='Macbook';
```



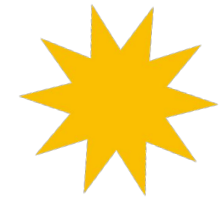
DELETE

The DELETE statement allows you to delete records from database tables.

```
DELETE FROM Product [WHERE condition];
```

```
DELETE FROM Product;
```

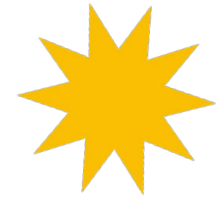
```
DELETE FROM Product WHERE Name='Macbook' AND ProductID  
In(2,3,10);
```



Task 9

1. Add data to the database:
 - a. clients:
 - i. 'John', 'Smith', 'Front Street 12', 'Los Angeles'
 - ii. 'Andrew', 'Jones', 'Back Street 43', 'New York'
 - b. cars:
 - i. 'Seat', 'Leon', 2016, 80, 200
 - ii. 'Toyota', 'Avensis', 2014, 72, 100
 - c. bookings:
 - i. 1, 2, '2020-07-05', '2020-07-06', 100
 - ii. 2, 2, '2020-07-10', '2020-07-12', 200
2. *Replace the data of the selected customer with your data.
3. *In connection with the GDPR - delete your data.
4. *Add 2 new customers.

Answer



```
INSERT INTO clients (name, surname, address, city)
VALUES
('John', 'Smith', 'Front Street 12', 'Los Angeles'),
('Andrew', 'Jones', 'Back Street 43', 'New York' );
```

```
INSERT INTO cars (producer, model, year, horse_power,
price_per_day)
VALUES
('Seat', 'Leon', 2016, 80, 200),
('Toyota', 'Avensis', 2014, 72, 100);
```

```
INSERT INTO bookings (client_id, car_id, start_date,
end_date, total_amount)
VALUES
(1, 2, '2020-07-05', '2020-07-06', 100),
(2, 2, '2020-07-10', '2020-07-12', 200);
```

SELECT

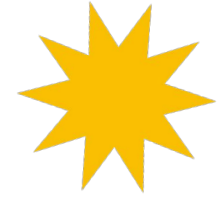
The data from the tables can be retrieved using the SELECT statement. The instruction itself can be very complex due to the large number of available clauses.

The output can be sorted (ASC ascending, DSC descending).

The SELECT statement can additionally limit the set of returned data to those meeting the given condition, which is done using the WHERE clause:

Additionally, with BETWEEN you can:

```
SELECT column1, column2, ..., columnN
FROM table
[WHERE condition]
[ORDER BY column1, column2, ..., columnN [ASC |
DSC]]
```



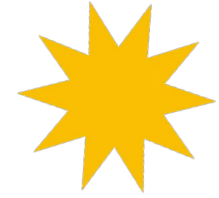
```
SELECT *
FROM Product
```

```
SELECT *
FROM Product
ORDER BY Name ASC
```

```
SELECT *
FROM Product
WHERE Name='Macbook' AND Description LIKE 'Late
%';
```

```
SELECT *
FROM Product
WHERE ProductId BETWEEN 3 and 10;
```


Display the table based on the selected columns

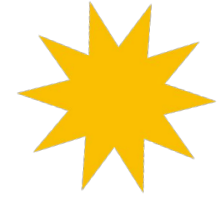


In order to display data from specific columns, indicate their names, separating them with a comma, e.g .:

```
SELECT Name, Description FROM Product;
```

The above query will display data only for selected columns, columns not included will be ignored.

Aliases

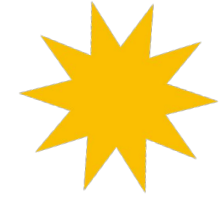


As part of SELECT queries, you can modify the names of the displayed columns, which is done using the so-called aliases. We define the alias by adding the AS keyword and the name of our choice after the column name, e.g.:

```
SELECT  
Name AS product_name,  
Description AS product_description  
FROM Product;
```

The above query will display the names product_name and product_description in the returned result, instead of the actual names of these columns.

Criteria for data collection



Specifying specific criteria for data collection is performed using the WHERE clause. Within it, we can use many operators, both relational and logical.

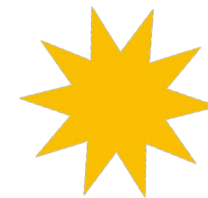
Relational operators

Operator	Description
=	Equality
>	Greater than
<	Lesser than
>=	Greater or equal
<=	Lesser or equal
!=	Different
BETWEEN	inside the range
LIKE	Searching by the template
IN	In the given set

Logical operators

Operator	Description
AND	Boolean, returns true if both arguments are true
&&	Boolean, the same as AND
OR	Logical sum, returns true if one of the arguments is true
	Logical sum, the same as OR
NOT	Logical negation, changes the value of the logical argument to the opposite one
!	Logical negation, the same as NOT

Examples



```
SELECT * FROM Product WHERE Name='Macbook';
```

The above query is responsible for finding all records whose name is Macbook.

```
SELECT * FROM Product WHERE Name='Macbook' AND Description LIKE 'Late%';
```

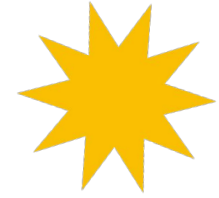
The above query is responsible for retrieving all records that have the name Macbook and the description indicates that the value starts with Late characters. The LIKE operator, in addition to the % sign, which stands for many (also zero) characters, also allows you to use the _ sign, which represents any single character.

```
SELECT * FROM Product WHERE ProductId BETWEEN 3 and 10;
```

The above query will find all products that have a ProductId between 3 and 10. Note that both the lower and upper limits are included in the search.

```
SELECT * FROM Product WHERE Name IN('Macbook', 'Dell');
```

As part of the query above, you can retrieve all records whose Name is one of the ones defined within the IN operator. Remember that by using many values inside the IN set, the execution time of such a query can increase significantly.



Limiting the number of records

The limitation of the returned records can be implemented using the LIMIT clause, e.g.:

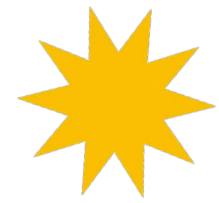
```
SELECT * FROM Product LIMIT 10;
```

The above query will return the first 10 records from the Product table.

Under the same clause, we can restrict the collection of data starting from a specific item:

```
SELECT * FROM Product LIMIT 4,3; -- the first number is the position, the second is  
the number of records
```

The above query allows you to return three records from the Product table starting from item 4.



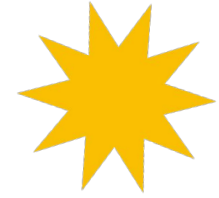
Data for the task:

```
INSERT INTO clients (name, surname, address, city) VALUES
('Michał', 'Taki', 'os. Srodkowe 12', 'Poznan'),
('Paweł', 'Ktory', 'ul. Stara 11', 'Gdynia'),
('Anna', 'Inna', 'os. Srednie 1', 'Gniezno'),
('Alicja', 'Panna', 'os. Duze 33', 'Torun'),
('Damian', 'Papa', 'ul. Skosna 66', 'Warszawa'),
('Marek', 'Troska', 'os. Male 90', 'Radom'),
('Jakub', 'Klos', 'os. Polskie 19', 'Wadowice'),
('Lukasz', 'Lis', 'os. Podlaskie 90', 'Bialystok');
```

```
INSERT INTO cars (producer, model, year, horse_power, price_per_day) VALUES
('Mercedes', 'CLK', 2018, 190, 400),
('Hyundai', 'Coupe', 2014, 165, 300),
('Dacia', 'Logan', 2015, 103, 150),
('Saab', '95', 2012, 140, 140),
('BMW', 'E36', 2007, 110, 80),
('Fiat', 'Panda', 2016, 77, 190),
('Honda', 'Civic', 2019, 130, 360),
('Volvo', 'XC70', 2013, 180, 280);
```

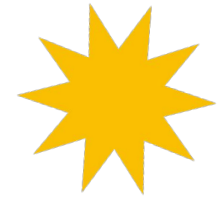
```
INSERT INTO bookings (client_id, car_id, start_date,
end_date, total_amount) VALUES
(3, 3, '2020-07-06', '2020-07-08', 400),
(6, 10, '2020-07-10', '2020-07-16', 1680),
(4, 5, '2020-07-11', '2020-07-14', 450),
(5, 4, '2020-07-11', '2020-07-13', 600),
(7, 3, '2020-07-12', '2020-07-14', 800),
(5, 7, '2020-07-14', '2020-07-17', 240),
(3, 8, '2020-07-14', '2020-07-16', 380),
(5, 9, '2020-07-15', '2020-07-18', 1080),
(6, 10, '2020-07-16', '2020-07-20', 1120),
(8, 1, '2020-07-16', '2020-07-19', 600),
(9, 2, '2020-07-16', '2020-07-21', 500),
(10, 6, '2020-07-17', '2020-07-19', 280),
(1, 9, '2020-07-17', '2020-07-19', 720),
(3, 7, '2020-07-18', '2020-07-21', 240),
(5, 4, '2020-07-18', '2020-07-22', 1200);
```

Task 10



1. List all cars with a production year greater than 2015.
2. List all reservations with a total cost in the range of 1000–2555.
3. List the id of all customers whose last name starts with 'N' and first name ends with 'ew'.

Answer



1)

```
SELECT *  
FROM cars  
WHERE year > 2015;
```

2)

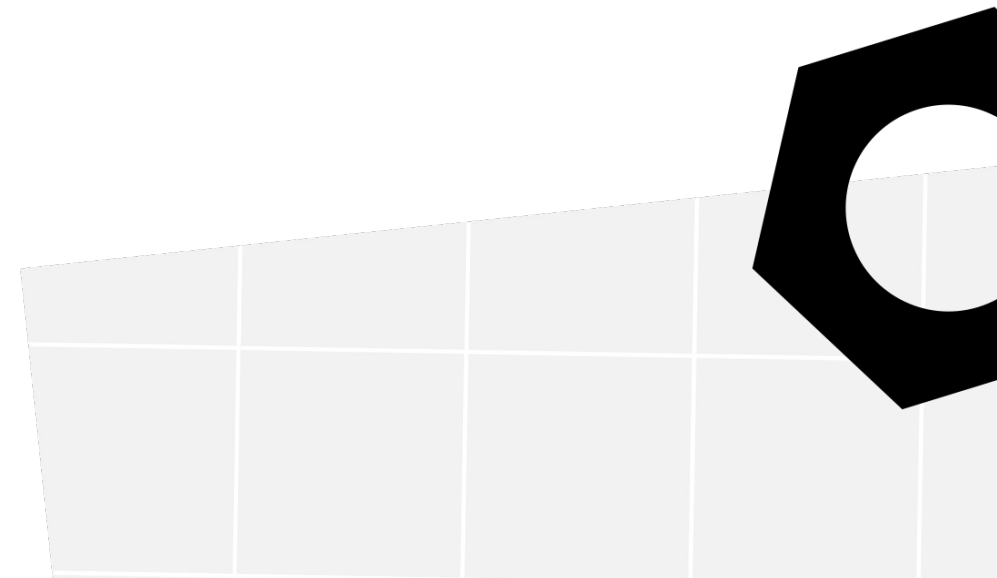
```
SELECT *  
FROM bookings  
WHERE total_amount  
BETWEEN 1000 AND 2555;
```

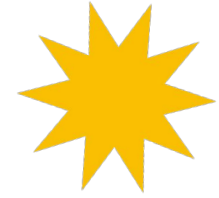
3)

```
SELECT client_id  
FROM clients  
WHERE surname  
LIKE "N%" AND name LIKE "%ew";
```




DATA CONTROL LANGUAGE



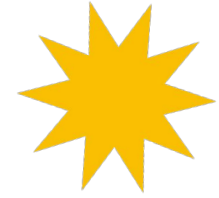


GRANT

By means of the GRANT instruction, it is possible to modify the authorizations of users that are part of the database system.

```
GRANT permissions [columns] ON level TO user  
[IDENTIFIED By password] [WITH [GRANT OPTION |  
MAX_QUERIES_PER_HOUR how_many |  
MAX_UPDATES_PER_HOUR how_many |  
MAX_USER_CONNECTIONS how_many |  
MAX_CONNECTIONS_PER_HOUR]  
];
```

```
GRANT CREATE, SELECT, INSERT, UPDATE, DELETE  
ON * TO sda_user;
```



REVOKE

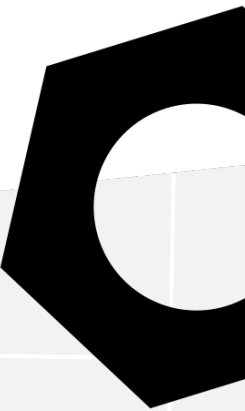
Using the Revoke instruction it is possible to revoke users' permissions.

```
REVOKE permissions [columns] ON object FROM user;
```

```
REVOKE UPDATE, DELETE ON sda.* FROM sda_user,  
sda_employee;
```



Joins in databases



RETRIEVING DATA FROM SEVERAL TABLES

The Select statement allows you to retrieve data from several tables:

We often make such queries by giving aliases to individual tables, e.g .:

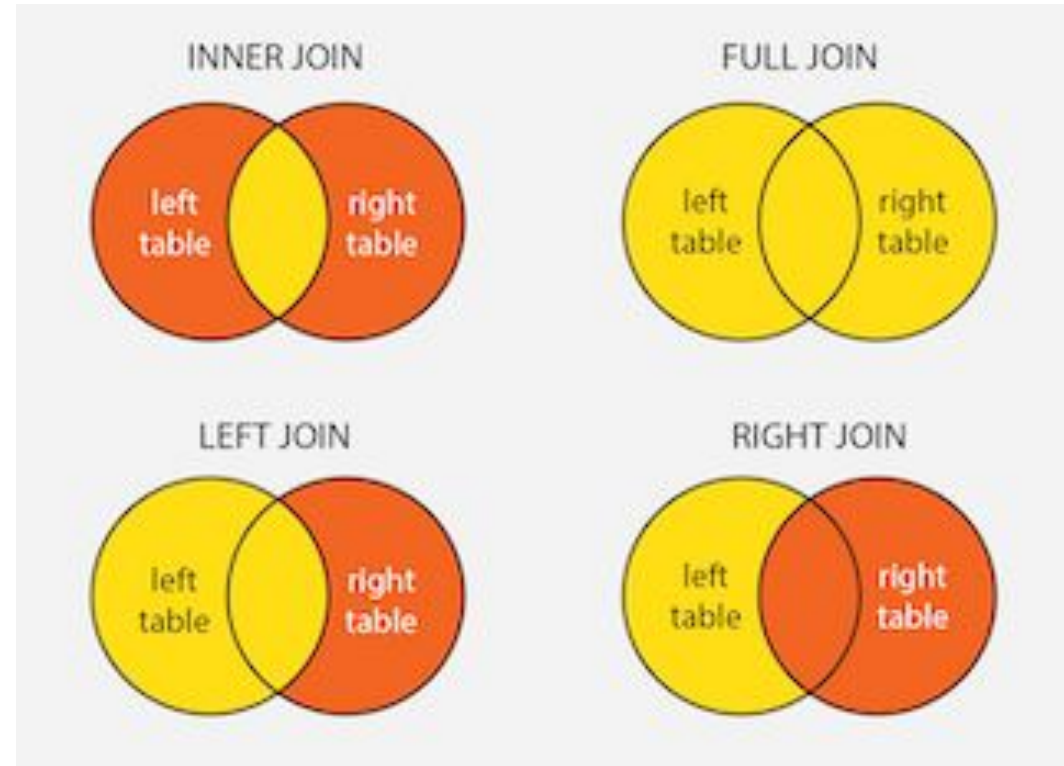
```
SELECT column1, column2, column3, ...,columnN
FROM table1, table2, ..., tableN
[WHERE condition]
...;
```

```
SELECT k.Print, p.Name
FROM Book AS b, Product AS p;
```

JOINS

There are different types of joins in SQL including:

- INNER JOIN
- LEFT JOIN/RIGHT JOIN
- FULL JOIN



Example:

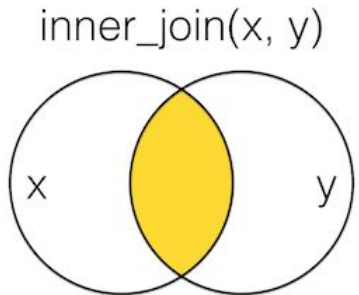
For the following examples, we will use the following tables with data:

ProducerId	Name	Address
1	Apple	Cupertino, CA 95014, US
3	Dell	Berkshire, RG12 1 LF, UK
4	Microsoft	1045 Avenida St, Mountain View, CA 94043, US

Id	Name	ProducerId
1	Macbook Pro 16	1
2	Iphone 11 Pro	1
3	Dell XPS 15	3
4	Lenovo ThinkPad 13	

INNER JOIN

This type of join allows you to join the table data like a regular SELECT statement with regard to tables (the ON clause defines the join condition).



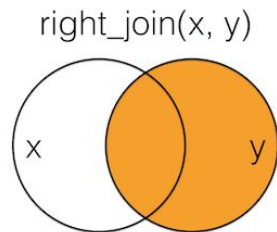
```
SELECT column1, column2, ..., columnN  
FROM table1  
[INNER] JOIN table2 [ON condition];
```

For the example presented, the command will take the following form:

```
SELECT ProductId, Product.Name, Product.ProducerId,  
       Producer.ProducerId, Producer.Name, Producer.Address  
FROM Product  
INNER JOIN Producer ON  
       Product.ProducerId=Producer.Producer.Id;
```


LEFT JOIN/RIGHT JOIN

This type of join allows you to include the resulting data that is not related to the joined table. In short, if there are records in table1 that are not correlated with records in table2, they will be included in the join anyway, and the missing values will be filled with NULL values.



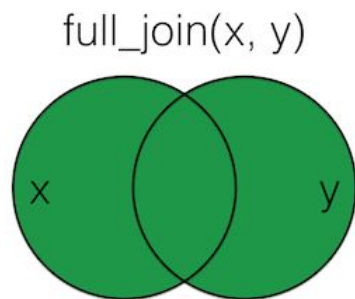
```
SELECT column1, column2, ..., columnN  
FROM table1  
[LEFT] JOIN table2 [ON condition];
```

For the example presented, the command will take the following form:

```
SELECT ProductId, Product.Name, Product.ProducerId,  
Producer.ProducerId, Producer.Name, Producer.Address  
FROM Product  
LEFT JOIN Producer ON  
Product.ProducerId=Producer.Producer.Id;
```

FULL JOIN

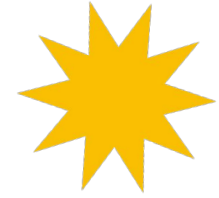
This join allows you to get all the records from the two tables. Taking into account both the records from table1, which do not have their counterparts in table2, and the opposite situation.



```
SELECT column1, column2, ..., columnN FROM table1 [FULL  
OUTER] JOIN table2 [ON condition];
```

For the example presented, the command will take the following form:

```
SELECT ProductId, Product.Name, Product.ProducerId,  
Product.ProducerId, Producer.Name, Producer.Address  
FROM Product FULL OUTER JOIN Producer ON  
Product.ProducerId=Producer.Producer.Id;
```



Task 11

1. List the names of all customers and their booking costs greater than 1000.
2. List the city of residence of all customers who rented a car in the period 12-20.07.2020, and the engine power of the rental car is not more than 120, sorting by the highest rental cost.
3. * List the number of rental cars with a daily rental cost of 300 or more by grouping cars by engine power, sorting from the smallest.
4. * List the sum of the costs of all bookings that were made in the period July 14-18, 2020.
5. * List:
 - a. average amount of money spent by each customer - column naming: Average_reservations_price
 - b. the number of rented cars for each customer, taking into account only those customers who have rented at least two cars - column nomenclature: Number of rented cars
 - c. name and surname of the client - nomenclature of the columns: Name, Surname
 - d. sorting by the largest number of rental cars. All with one query.

Answer

1)

```
SELECT c.name, b.total_amount
FROM bookings b
JOIN clients c ON c.client_id = b.client_id
WHERE b.total_amount > 1000;
```

2)

```
SELECT c.city, b.total_amount
FROM clients c
JOIN bookings b ON c.client_id = b.client_id
JOIN cars r ON b.car_id = r.car_id
WHERE b.start_date >= '2020-07-12'
AND b.end_date <= '2020-07-20'
AND r.horse_power <= 120
ORDER BY b.total_amount DESC;
```

3)

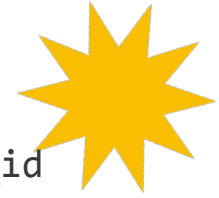
```
SELECT COUNT(b.car_id)
FROM bookings b
JOIN cars r ON b.car_id = r.car_id
WHERE r.price_per_day >= 300
GROUP BY r.horse_power
ORDER BY r.horse_power;
```

4)

```
SELECT SUM(total_amount)
FROM bookings
WHERE start_date >= '2020-07-14'
AND end_date <= '2020-07-18';
```

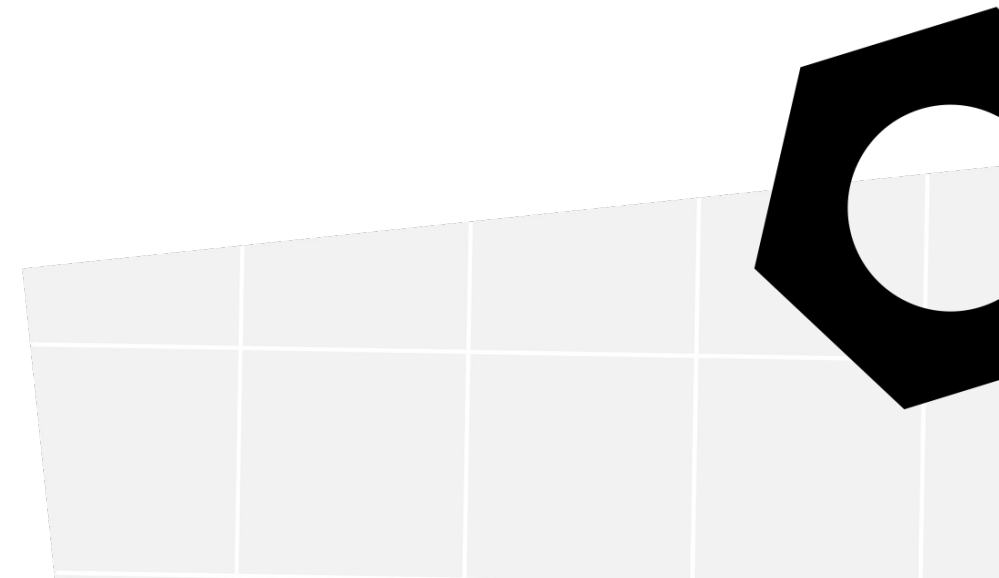
5)

```
SELECT AVG(b.total_amount) AS
Average_reservations_price,
COUNT(b.car_id) AS Amount_of_rented_cars,
c.name AS Imie, c.surname AS Surname
FROM bookings b
JOIN clients c ON c.client_id = b.client_id
GROUP BY b.client_id
HAVING Amount_of_rented_cars >= 2
ORDER BY Amount_of_rented_cars DESC;
```

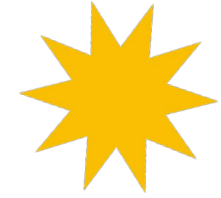




Grouping data

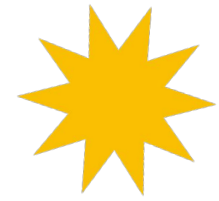


STATISTICAL AND AGGREGATING FUNCTIONS



The SQL language provides a set of many statistical and aggregation functions, including:

Function	Description
AVG	Calculating the average number from the values in the query
COUNT	Calculating the amount of numbers in the query
MIN	Calculating the minimal value from the query
MAX	Calculating the maximal value from the query



COUNT

This function returns the total number of records obtained from the query. The * operator is an alias for all lines. The query will result in a table with a column named COUNT (*) and one record for the total number of records in the parent table.

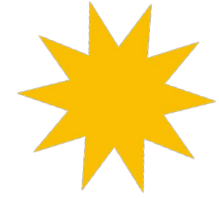
```
SELECT COUNT(*)  
FROM Product;
```

In order to change the name of the result column, you can use the alias:

```
SELECT COUNT(*) AS 'Amount of products'  
FROM Product;
```

As part of the query, you can also specify a condition that should ultimately reduce the number of records searched:

```
SELECT COUNT(*) AS 'Amount of products'  
FROM Product  
WHERE Name='Macbook';
```



CALCULATION OF THE AVERAGE

This function allows you to calculate the average of the value returned in the query:

```
SELECT AVG(Price) AS 'Average price of the computer'  
FROM Product
```

In addition, as standard, as for each SELECT query, we can enter the WHERE clause, which will return the result based on records that meet the condition specified in the query:

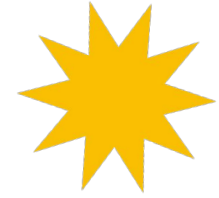
```
SELECT AVG(Price) AS Average price of the product'  
FROM Product  
WHERE Price > 10;
```


MIN AND MAX

These functions allow you to search for a minimum and maximum value from the data returned in the query:

```
SELECT MIN(Price) AS 'The cheapest  
computer'  
FROM Computer  
WHERE ProducerId=10;
```

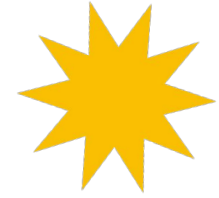
```
SELECT MAX(Price) AS 'The most  
expensive computer'  
FROM Computer  
WHERE ProducerId=4;
```



SUM

This function allows you to return the sum of specific values based on the obtained results:

```
SELECT SUM(Price) AS `Total value of APPLE computers` FROM Computer  
WHERE ProducId=4;
```



GROUPING OF INQUIRY RESULTS

The GROUP BY clause allows us to group query results within a selected column or multiple columns.

Thanks to this clause, it is possible to return, for example, the number of products for each manufacturer in the database. The result of the query will be a new table containing the number of records corresponding to the number of manufacturers with correlated products in the database.

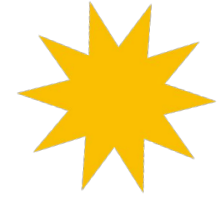
The next one represents the use of the GROUP BY clause with an additional condition:

```
SELECT column1, column2, ..., columnN
FROM table1, table2, ..., tableN
WHERE conditions
GROUP BY column1, column2, ..., columnN;
```

```
SELECT COUNT(*)
FROM Product
GROUP BY ProducerId;
```

```
SELECT MIN(Price), Max(Price)
FROM computer
WHERE Processor='Intel I9'
GROUP BY ProducerId;
```

GROUPING CONDITIONS



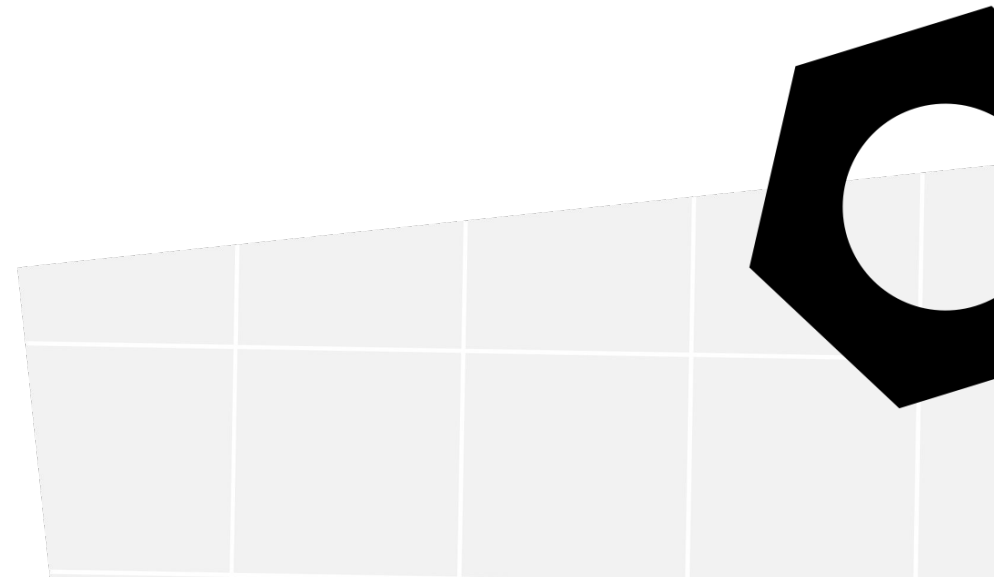
The HAVING clause allows you to limit the results of group queries:

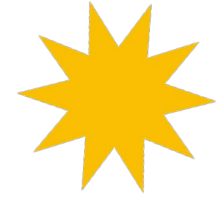
```
SELECT column1, column2, ..., columnN
FROM table1, table2, ..., tableN
WHERE conditions_where
GROUP BY column1, column2, ..., columnN
HAVING conditions_having;
```

```
SELECT SUM(Price)
FROM Computer
WHERE Processor='Intel I9'
GROUP BY ProducerId
HAVING COUNT(*) > 40;
```



SQLAlchemy

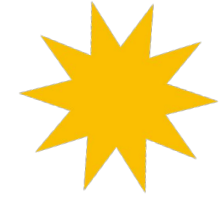




SQLAlchemy

An open source programming library written in the Python programming language for working with databases.

SQLAlchemy

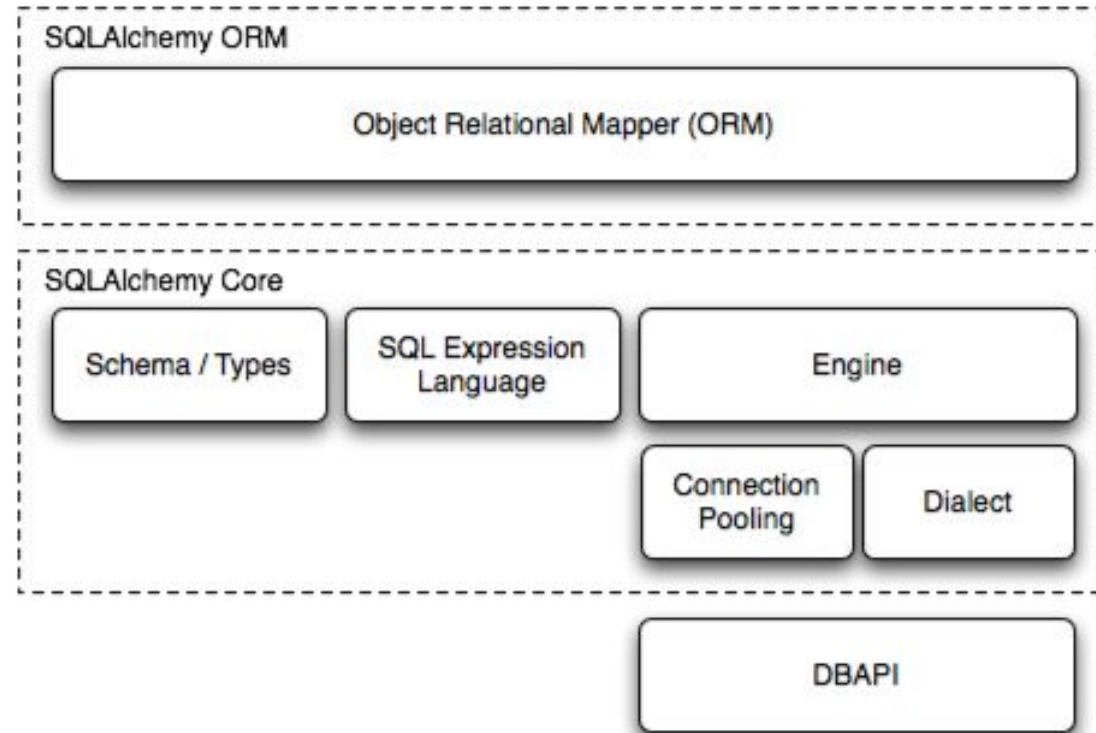


SQLAlchemy

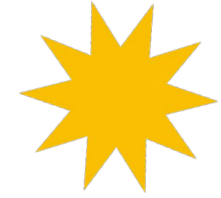
It consists of two parts:

- Core
 - It gives unified Python API access to different types of databases.
 - It also allows you to conveniently use Python expressions in SQL queries.
- ORM (Object-Relational Mapping)
 - An optional component that allows to describe the tables in the database with classes.
 - It also allows you to manipulate data through operations on objects (without writing SQL queries by the programmer himself).

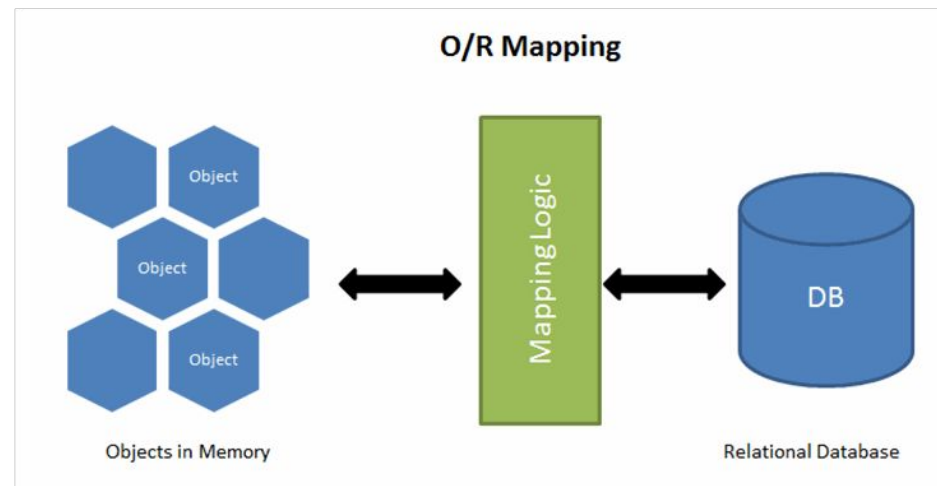
At the moment, the following databases are supported: SQLite, Postgresql, MySQL, Oracle, MS-SQL, Firebird and Sybase.

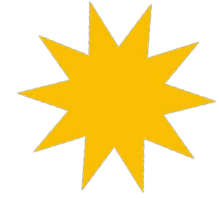


ORM



ORM stands for "Object-Relational Mapping". So it is about changing tabular data (relations in a database) into objects, or the other way around. It is a modern approach to the issue of cooperation with a database, using the philosophy of object-oriented programming.





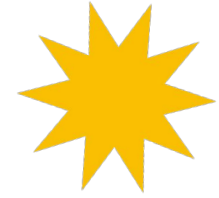
SQL or ORM queries?

ORM queries

- Perfect for simpler queries and working with multiple dialects at the same time
- Access to many open-source libraries enabling e.g. data serialization

SQL queries

- Perfect for more complex database operations
- It uses the full potential of the database engine



Database connection

To connect to the selected database, we need to use the **create_engine** function:

This function creates an 'Engine' class object based on the given URL.

This function creates an 'Engine' class object based on the given URL

```
create_engine('dialect+driver://username:password@host:port/database')
```

Example:

```
create_engine('postgresql+pg8000://user:password@localhost:3306/car_rental', echo = True)
```

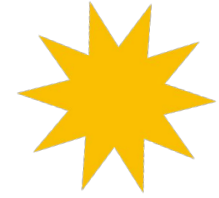
The echo flag enables displaying / logging of activities, queries that SQLAlchemy will perform on the given database.

- dialect – a type of sql dialect (qlite, mysql, postgresql, oracle, or mssql)
- driver – name of the DBAPI (Python Database API) used to connect to the database. If it is not given, the interpreter will use the default for the given dialect

SQLite file databases require a path, you can also create volatile databases in RAM (they disappear after the script is finished).

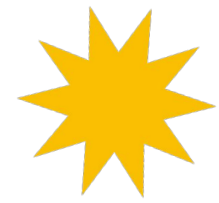
For databases such as MySQL or Postgresql, we provide the host and database name and optionally other parameters

Standard queries



To execute any query, just use the execute method:

```
mysql_db = create_engine('mysql://login:password@localhost/db_name', echo=True)
s = mysql_db.execute("SHOW TABLES;")
print list(s)
```



ORM basics

In the case of using ORM, each table we want to work with must be described with a class. Here is an example:

First we give the name of the table in the database with `__tablename__`.

Then we list the table columns by their names. The `Column` class takes a variety of arguments, including field types, such as text is `String` and numeric is `Integer`. For example, MySQL translates into `VARCHAR` and `INTEGER` fields.

Optionally, we can define methods such as `__init__` or `__repr__`.

```
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy import Column, Integer, String
```

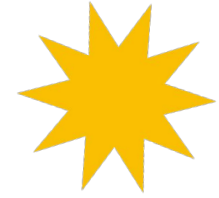
```
# base for table classes
Base = declarative_base()
```

```
# sample class mapping table from database
class User(Base):
    __tablename__ = 'users'
```

```
# fields and their types
    id = Column(Integer, primary_key=True)
    name = Column(String)
    fullname = Column(String)
    password = Column(String)
```

```
    def __init__(self, name, fullname, password):
        self.name = name
        self.fullname = fullname
        self.password = password
```

```
    def __repr__(self):
        return "<User('%s', '%s', '%s')>" % (self.name,
self.fullname, self.password)
```



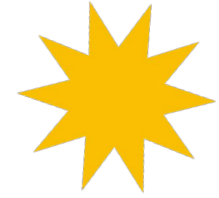
Creating tables

We have a class that is not yet linked to any real table in the existing database. Our "User" class is table metadata (metadata) in SQLAlchemy terminology. We can operate on this metadata with Base.metadata. For example, if tables do not exist, they must be created:

```
engine =  
create_engine('sqlite:///memory:',  
echo=True)
```

```
Base.metadata.create_all(engine)
```

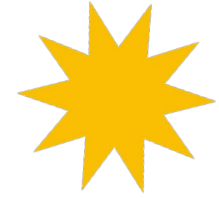
Google Colab environment initialization



```
!apt-get install mysql-server > /dev/null
!service mysql start
!mysql -e "ALTER USER 'root'@'localhost' IDENTIFIED WITH mysql_native_password BY 'root'"
!pip -q install PyMySQL
```

```
from sqlalchemy import create_engine
```

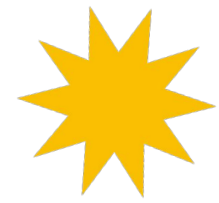
```
engine = create_engine("mysql+pymysql://root:root@/")
engine.execute("CREATE DATABASE car_rental") #create db
engine.execute("USE car_rental") # select new db
```



Task 12

1. Create cars, clients, bookings tables according to the guidelines (without relations):
 - a. cars: car_id(int, pk), producer(str), model(str), year(int), horse_power(int), price_per_day(int)
 - b. clients: client_id(int, pk), name(str), surname(str), address(str), city(str)
 - c. bookings: booking_id(int, pk), client_id(int), car_id(int), start_date(date), end_date(date), total_amount(int)
2. * Do the same with a SQL query.

Answer



```
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy import Column, String, Integer, Date

engine = create_engine("mysql+pymysql://root:root@/", echo = True)

engine.execute("CREATE DATABASE car_rental") #create db
engine.execute("USE car_rental") # select new db

_____
eng = create_engine("mysql+pymysql://root:root@/car_rental")
base = declarative_base()

class Cars(base):
    __tablename__ = 'cars'

    car_id = Column(Integer, primary_key=True, autoincrement=True)
    producer = Column(String(30), nullable=False)
    model = Column(String(30), nullable=False)
    year = Column(Integer, nullable=False)
    horse_power = Column(Integer, nullable=False)
    price_per_day = Column(Integer, nullable=False)
```

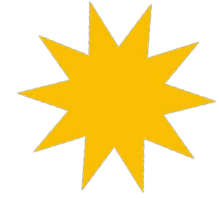
```
class Clients(base):
    __tablename__ = 'clients'

    client_id = Column(Integer, primary_key=True, autoincrement=True)
    name = Column(String(30), nullable=False)
    surname = Column(String(30), nullable=False)
    address = Column(String(30), nullable=False)
    city = Column(String(30), nullable=False)

class Bookings(base):
    __tablename__ = 'bookings'

    booking_id = Column(Integer, primary_key=True, autoincrement=True)
    client_id = Column(Integer, nullable=False)
    car_id = Column(Integer, nullable=False)
    start_date = Column(Date, nullable=False)
    end_date = Column(Date, nullable=False)
    total_amount = Column(Integer, nullable=False)

base.metadata.create_all(eng)
```

Adding records

To save the record, we need to handle the session and pass it within the object.

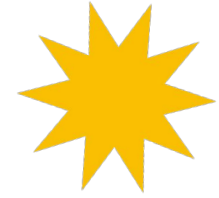
```
from sqlalchemy.orm import sessionmaker

engine = create_engine('sqlite:///memory:', echo=True)

Base.metadata.create_all(engine)

# creating a session for a given database:
Session = sessionmaker(bind=engine)
session = Session()

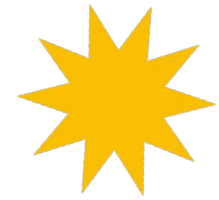
ed_user = User('James', 'James Hajto', 'pas(s)y')
session.add(ed_user)
# performing the operation
session.commit()
print (ed_user.id)
```



Task 13

1. Complete the clients and cars tables with the following data:
 - a. clients: 'Andrew', 'Novak', 'Front Street 43', 'New York'
 - b. cars: 'Seat', 'Leon', 2016, 80, 200
2. * Add your data.
3. * Add with SQL query:
 - a. 'Andrew', 'Jones', 'Back Street 43', 'Los Angeles'
 - b. 'Opel', 'Vectra', 2010, 240, 70000000

Answer



```
from sqlalchemy.orm import sessionmaker
```

```
Session = sessionmaker(bind=eng)
```

```
session = Session()
```

```
client_1 = Clients(name='Jan', surname='Smith', address='Front Street 12', city='Los Angeles')
```

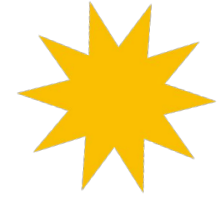
```
car_1 = Cars(producer='Seat', model='Leon', year=2016, horse_power=80, price_per_day=200)
```

```
session.add(client_1)
```

```
session.add(car_1)
```

```
session.commit()
```

Queries



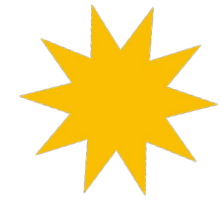
We make inquiries on the session instance, for example:

```
for instance in session.query(User).order_by(User.id):  
    print(instance)
```

or

using SELECT

```
s = select([users]).order_by(users)  
for row in eng.execute(s):  
    print(row)
```



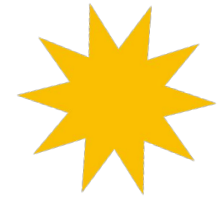
Task 14

1. Check if the data added in the previous task has been saved in the database – add a piece of code that will list all the data from the cars and clients tables.
2. *Check the same with an SQL query.
3. * Update the Cars, Clients, Bookings classes so that the display of data with the function print () is clearer, for example like below (we don't have a reservation added yet):

```
<Client: id=1, name=John, surname=Smith, address=Front Street 12,city=New York>
```

```
<Car: id=1, producer=Seat, model=Leon, year=2016, horse_power=80, price_per_day=200>
```

Answer



```
...
from sqlalchemy.orm import sessionmaker

Session = sessionmaker(bind=eng)
session = Session()

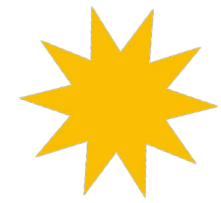
for client in session.query(Clients).all():
    print(client)

for car in session.query(Cars).all():
    print(car)
```

```
3) (Example)
class Cars(base):
    __tablename__ = 'cars'

    car_id = Column(Integer, primary_key=True, autoincrement=True)
    producer = Column(String(30), nullable=False)
    model = Column(String(30), nullable=False)
    year = Column(Integer, nullable=False)
    horse_power = Column(Integer, nullable=False)
    price_per_day = Column(Integer, nullable=False)

    def __repr__(self):
        return f'<Car: id={self.car_id}, producer={self.producer},
model={self.model}, year={self.year}, ' \
            f'horse_power={self.horse_power},
price_per_day={self.price_per_day}>'
```



Task 15

1. We need your data to be able to carry out a series of inquiries. Write a function `insert_data` that will take arguments:
 - a. `base` – the base class of the model we want to create an instance of
 - b. `params` – a dictionary with specific parameters of the created object
2. Test the function on the given data:

```
clients = [
    {'name': 'Jan', 'surname': 'Kowalski', 'address': 'ul. Florianska 12', 'city': 'Krakow'},
    {'name': 'Andrzej', 'surname': 'Nowak', 'address': 'ul. Saska 43', 'city': 'Wroclaw'},
    {'name': 'Michal', 'surname': 'Taki', 'address': 'os. Srodkowe 12', 'city': 'Poznan'},
    {'name': 'Pawel', 'surname': 'Ktory', 'address': 'ul. Stara 11', 'city': 'Gdynia'},
    {'name': 'Anna', 'surname': 'Inna', 'address': 'os. Srednie 1', 'city': 'Gniezno'},
    {'name': 'Alicja', 'surname': 'Panna', 'address': 'os. Duze 33', 'city': 'Torun'},
    {'name': 'Damian', 'surname': 'Papa', 'address': 'ul. Skosna 66', 'city': 'Warszawa'},
    {'name': 'Marek', 'surname': 'Troska', 'address': 'os. Male 90', 'city': 'Radom'},
    {'name': 'Jakub', 'surname': 'Klos', 'address': 'os. Polskie 19', 'city': 'Wadowice'},
    {'name': 'Lukasz', 'surname': 'Lis', 'address': 'os. Podlaskie 90', 'city': 'Bialystok'}]

cars = [
    {'producer': 'Seat', 'model': 'Leon', 'year': 2016, 'horse_power': 80, 'price_per_day': 200},
    {'producer': 'Toyota', 'model': 'Avensis', 'year': 2014, 'horse_power': 72, 'price_per_day': 100},
    {'producer': 'Mercedes', 'model': 'CLK', 'year': 2018, 'horse_power': 190, 'price_per_day': 400},
    {'producer': 'Hyundai', 'model': 'Coupe', 'year': 2014, 'horse_power': 165, 'price_per_day': 300},
    {'producer': 'Dacia', 'model': 'Logan', 'year': 2015, 'horse_power': 103, 'price_per_day': 150},
    {'producer': 'Saab', 'model': '95', 'year': 2012, 'horse_power': 140, 'price_per_day': 140},
    {'producer': 'BMW', 'model': 'E36', 'year': 2007, 'horse_power': 110, 'price_per_day': 80},
    {'producer': 'Fiat', 'model': 'Panda', 'year': 2016, 'horse_power': 77, 'price_per_day': 190},
    {'producer': 'Honda', 'model': 'Civic', 'year': 2019, 'horse_power': 130, 'price_per_day': 360},
    {'producer': 'Volvo', 'model': 'XC70', 'year': 2013, 'horse_power': 180, 'price_per_day': 280}]

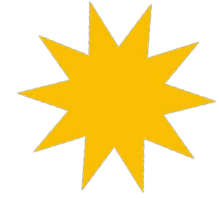
bookings = [
    {'client_id': 3, 'car_id': 3, 'start_date': '2020-07-06', 'end_date': '2020-07-08', 'total_amount': 400},
    {'client_id': 6, 'car_id': 10, 'start_date': '2020-07-10', 'end_date': '2020-07-16', 'total_amount': 1680},
    {'client_id': 4, 'car_id': 5, 'start_date': '2020-07-11', 'end_date': '2020-07-14', 'total_amount': 450},
    {'client_id': 5, 'car_id': 4, 'start_date': '2020-07-11', 'end_date': '2020-07-13', 'total_amount': 600},
    {'client_id': 7, 'car_id': 3, 'start_date': '2020-07-12', 'end_date': '2020-07-14', 'total_amount': 800},
    {'client_id': 5, 'car_id': 7, 'start_date': '2020-07-14', 'end_date': '2020-07-17', 'total_amount': 240},
    {'client_id': 3, 'car_id': 8, 'start_date': '2020-07-14', 'end_date': '2020-07-16', 'total_amount': 380},
    {'client_id': 5, 'car_id': 9, 'start_date': '2020-07-15', 'end_date': '2020-07-18', 'total_amount': 1080},
    {'client_id': 6, 'car_id': 10, 'start_date': '2020-07-16', 'end_date': '2020-07-20', 'total_amount': 1120},
    {'client_id': 8, 'car_id': 1, 'start_date': '2020-07-16', 'end_date': '2020-07-19', 'total_amount': 600},
    {'client_id': 9, 'car_id': 2, 'start_date': '2020-07-16', 'end_date': '2020-07-21', 'total_amount': 500},
    {'client_id': 10, 'car_id': 6, 'start_date': '2020-07-17', 'end_date': '2020-07-19', 'total_amount': 280},
    {'client_id': 1, 'car_id': 9, 'start_date': '2020-07-17', 'end_date': '2020-07-19', 'total_amount': 720},
    {'client_id': 3, 'car_id': 7, 'start_date': '2020-07-18', 'end_date': '2020-07-21', 'total_amount': 240},
    {'client_id': 5, 'car_id': 4, 'start_date': '2020-07-18', 'end_date': '2020-07-22', 'total_amount': 1200}]
```

Answer



```
...
def insert_data(session, base, params):
    session.add(base(**params))
    session.commit()

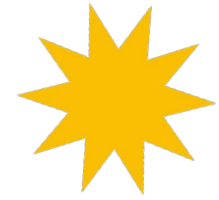
for client in clients:
    insert_data(session, Clients, client)
for car in cars:
    insert_data(session, Cars, car)
for booking in bookings:
    insert_data(session, Bookings, booking)
```

Task 16

1. List all reservations for the client with id = 3. Try both with query () and select ().
2. * List all cars rented by the client with id = 5. Cars may be repeated, we mean the history of rentals. Try both with select ().

Answer



1)

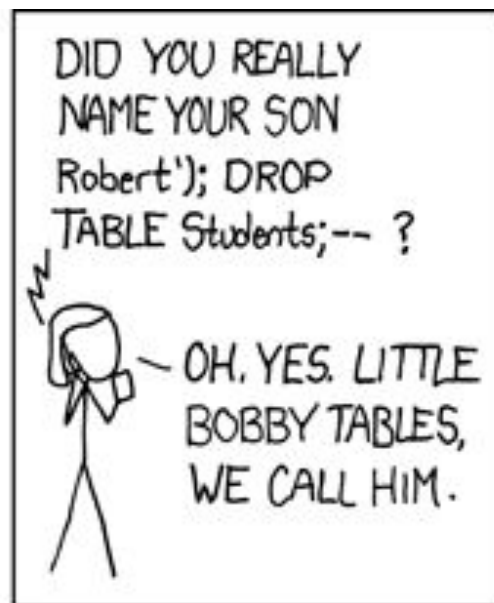
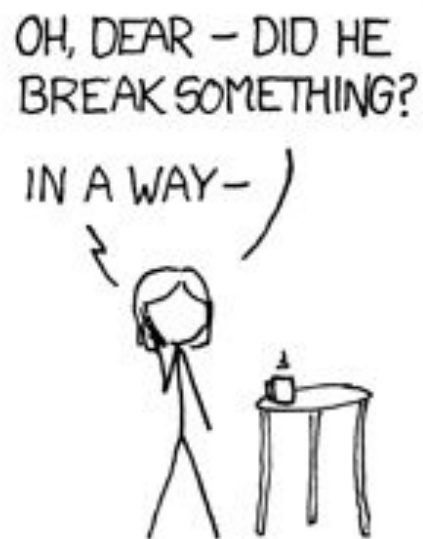
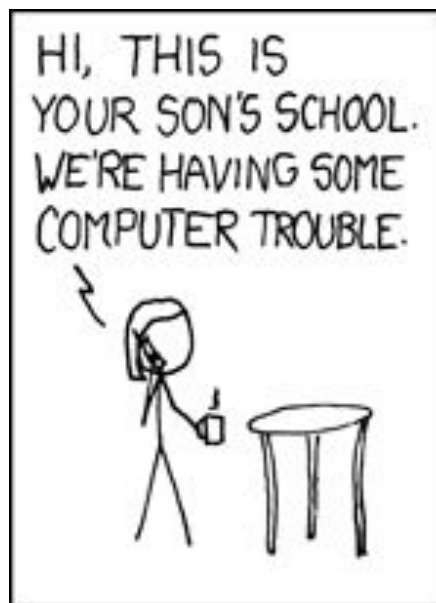
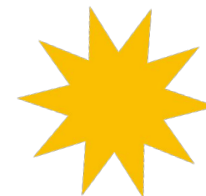
```
...
#1
result = session.query(Bookings).filter(Bookings.client_id == 3)
for booking in result:
    print(booking)
#2
from sqlalchemy.sql import select

conn = eng.connect()
s = select([Bookings]).where(Bookings.client_id == 3)
result = conn.execute(s).fetchall()
print(result)
```

2)

```
from sqlalchemy.sql import select
from sqlalchemy import join

j = join(Bookings, Cars, Bookings.car_id == Cars.car_id)
s = select([Cars]).select_from(j).where(Bookings.client_id == 5)
result = conn.execute(s)
for car in result:
    print(car)
```



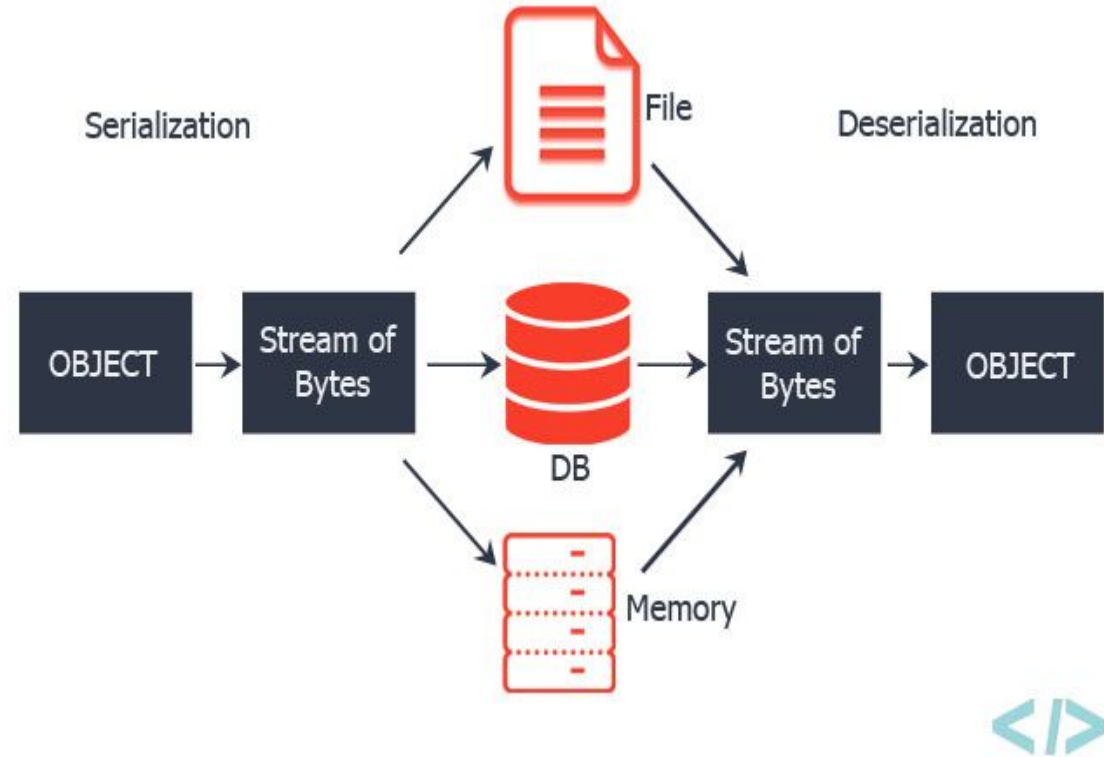


File serialization

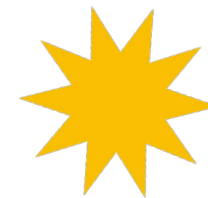
Serialization

The process of transforming objects, i.e. instances of specific classes, into a serial form, i.e. into a stream of bytes, while maintaining the current state of the object.

The serialized object can be persisted to a disk file, sent to another process or computer over the network. The process opposite to serialization is deserialization.



Data

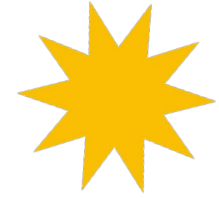


Flat

```
{  
  "Type" : "A",  
  "field1": "value1",  
  "field2": "value2",  
  "field3": "value3"  
}
```

Nested

```
{  
  "A"  
    {  
      "field1": "value1",  
      "field2": "value2",  
      "field3": "value3"  
    }  
}
```



Pickle(nested data)

Pickle enables you to serialize and deserialize the structure and properties of objects.

- "Pickling" – This is the process by which a Python object is converted into a sequence of bytes
- "Unpickling" – This is the reverse operation

Example:

```
>>> grades = { 'Alice': 89, 'Bob': 72, 'Charles': 87 }
```

```
#Use dumps to convert the object to a serialized string
```

```
>>> serial_grades = pickle.dumps(grades)
```

```
b'\x80\x03}q\x00(X\x05\x00\x00\x00Aliceq\x01K\u0003\x00\x00\x00Bobq\x02KH\u0007\x00\x00\x00Charlesq\x03KWu.'
```

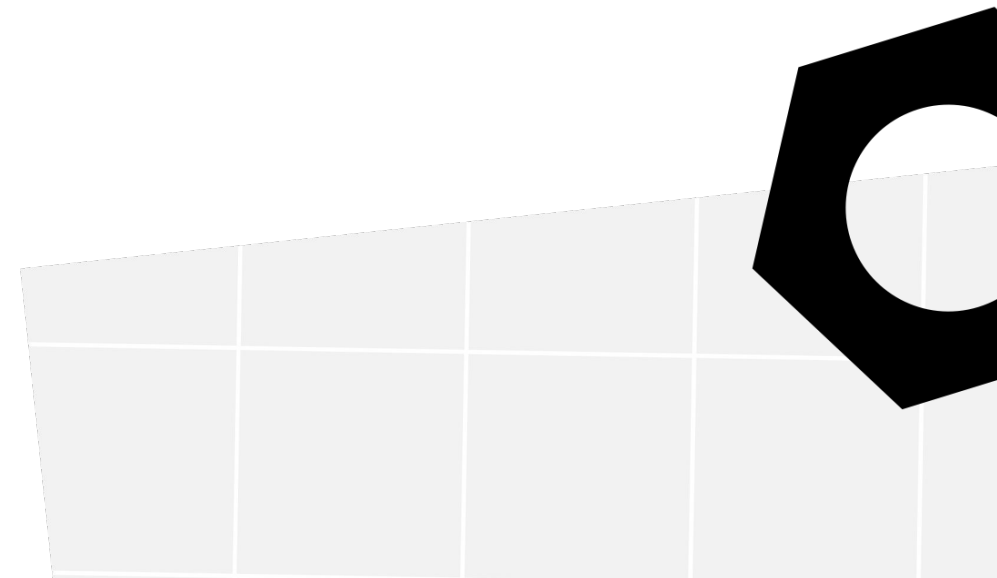
```
#Use loads to de-serialize an object
```

```
>>> received_grades = pickle.loads(serial_grades)
```

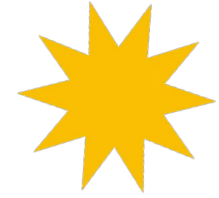
```
{'Alice': 89, 'Bob': 72, 'Charles': 87}
```



Pandas Profiling



Pandas Profiling



Generates an automatic report based on the provided data frame enabling EDA (Exploratory Data Analysis) to be carried out

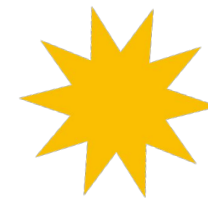
Basic statistics are calculated for each column, and everything is presented in the form of an interactive HTML report:

- column type
- unique and missing values
- quantiles, minimum, maximum, range
- mean, standard deviation
- the most common values
- histogram

and many others!



Creating a report



Creating a report is very easy, as you can see on the right.

For large datasets, we recommend that you disable the option of creating interaction graphs. To do this, set the appropriate flag.

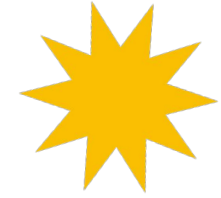
The product is constantly evolving, which you can follow at the link below:

<https://github.com/pandas-profiling/pandas-profiling>

```
from pandas_profiling import ProfileReport

# Generate the Profiling Report
profile = ProfileReport(df,
                        title="Titanic Dataset",
                        html={'style': {'full_width':
True}},
                        sort="None")

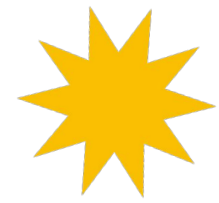
# Save to file
profile.to_file('report.html')
```



Overview

In the first basic view, we can see the general statistics and determine the number of individual types in the data frame

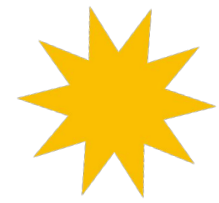
Overview	Warnings 11	Reproduction
Dataset statistics		Variable types
Number of variables	12	CAT 6
Number of observations	891	NUM 5
Missing cells	866	BOOL 1
Missing cells (%)	8.1%	
Duplicate rows	0	
Duplicate rows (%)	0.0%	
Total size in memory	83.7 KiB	
Average record size in memory	96.1 B	



Warnings

Pandas profiling also offers displaying information about potential problems with the dataset, such as multiple missing values, only unique values, or mostly zero values.

Warnings	
Ticket has a high cardinality: 681 distinct values	High cardinality
Cabin has a high cardinality: 147 distinct values	High cardinality
Age has 177 (19.9%) missing values	Missing
Cabin has 687 (77.1%) missing values	Missing
Ticket is uniformly distributed	Uniform
Cabin is uniformly distributed	Uniform
PassengerId has unique values	Unique
Name has unique values	Unique
SibSp has 608 (68.2%) zeros	Zeros



Column

Each column is analyzed separately here, on the right there is a histogram showing the distribution of values in a given column, in the center - basic statistics, and on the left, any warnings. In order to display more information, click on 'toggle details'

Variables

PassengerId

Real number ($\mathbb{R}_{\geq 0}$)

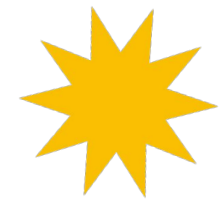
UNIQUE

Distinct	891
Distinct (%)	100.0%
Missing	0
Missing (%)	0.0%
Infinite	0
Infinite (%)	0.0%

Mean	446
Minimum	1
Maximum	891
Zeros	0
Zeros (%)	0.0%
Memory size	7.0 KiB

0200400600800

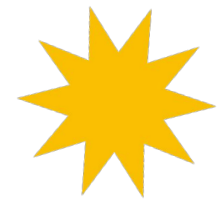
Toggle details



Details

The details contain virtually all interesting information about the data, and in addition to statistics, it is also possible to view the most common values or extreme values.

Statistics	Histogram	Common values	Extreme values	
Quantile statistics		Descriptive statistics		
Minimum	1		Standard deviation	257.353842
5-th percentile	45.5		Coefficient of variation (CV)	0.5770265516
Q1	223.5		Kurtosis	-1.2
median	446		Mean	446
Q3	668.5		Median Absolute Deviation (MAD)	223
95-th percentile	846.5		Skewness	0
Maximum	891		Sum	397386
Range	890		Variance	66231
Interquartile range (IQR)	445		Monotocity	Strictly increasing



Categorical values

Pandas profiling also deals with different types of variables, both numerical and categorical as you can see below:

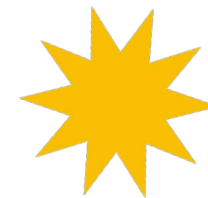
Sex
Categorical

Distinct	2
Distinct (%)	0.2%
Missing	0
Missing (%)	0.0%
Memory size	7.0 KiB

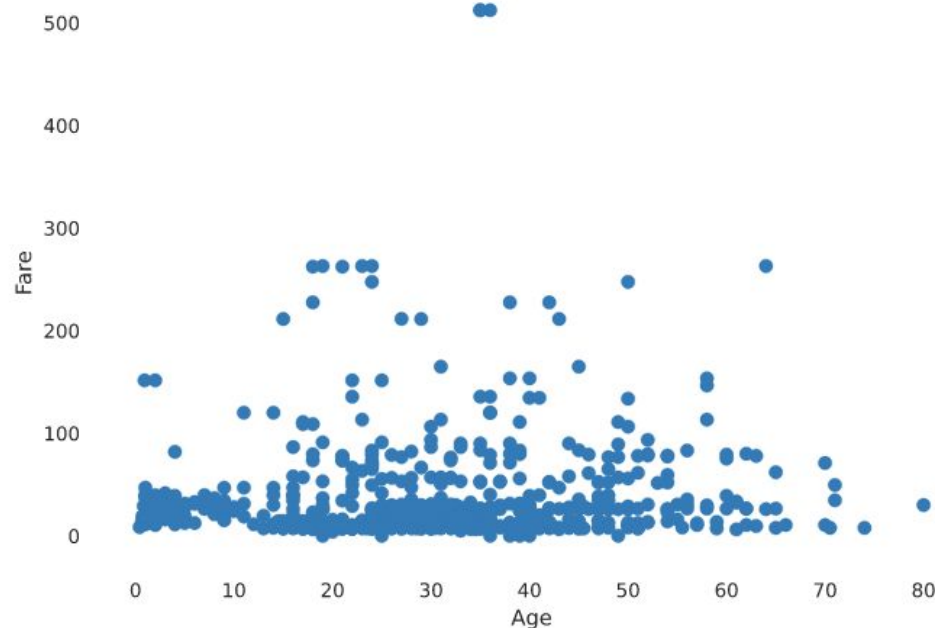


Toggle details

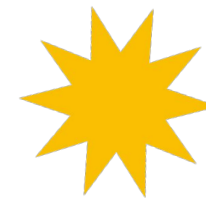
Interactions



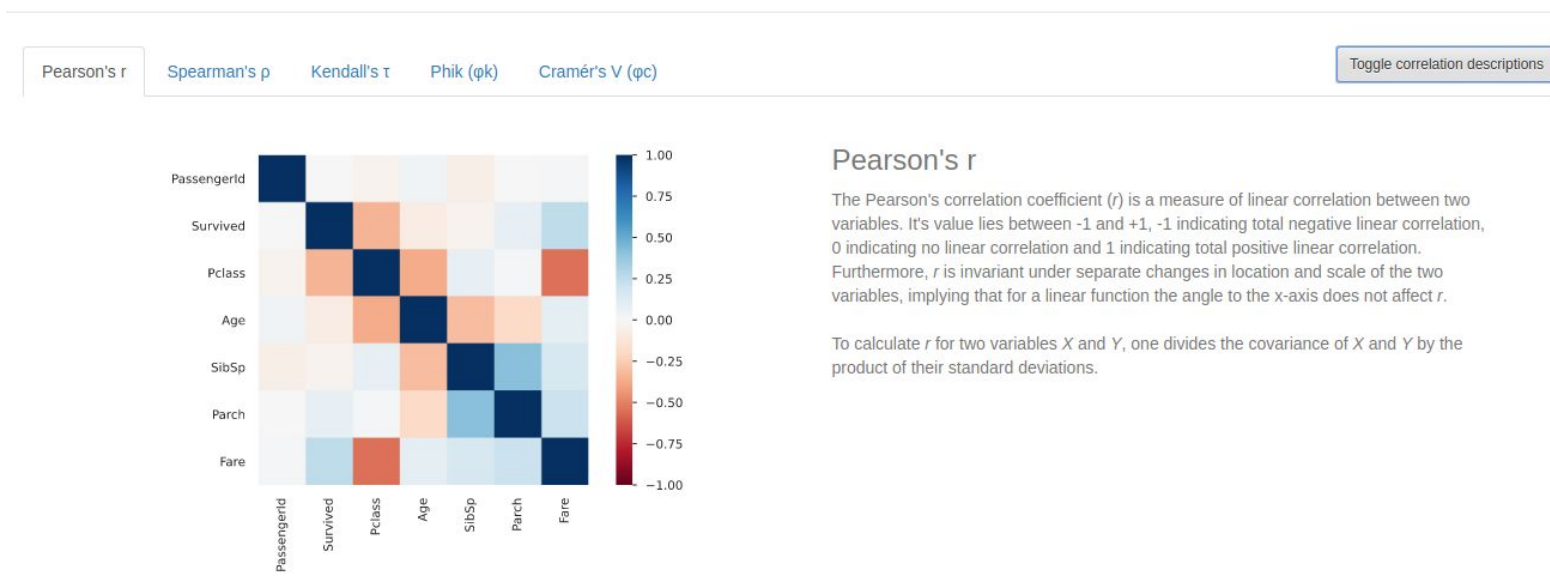
In the further part of the report you can find charts defining the relationships between particular numerical values. Thanks to them, you can easily notice certain phenomena. For example, there were two people on board, whose ticket was almost twice as expensive as the rest of the passengers.

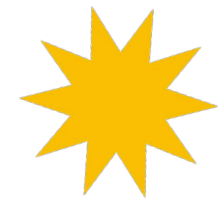


Correlations



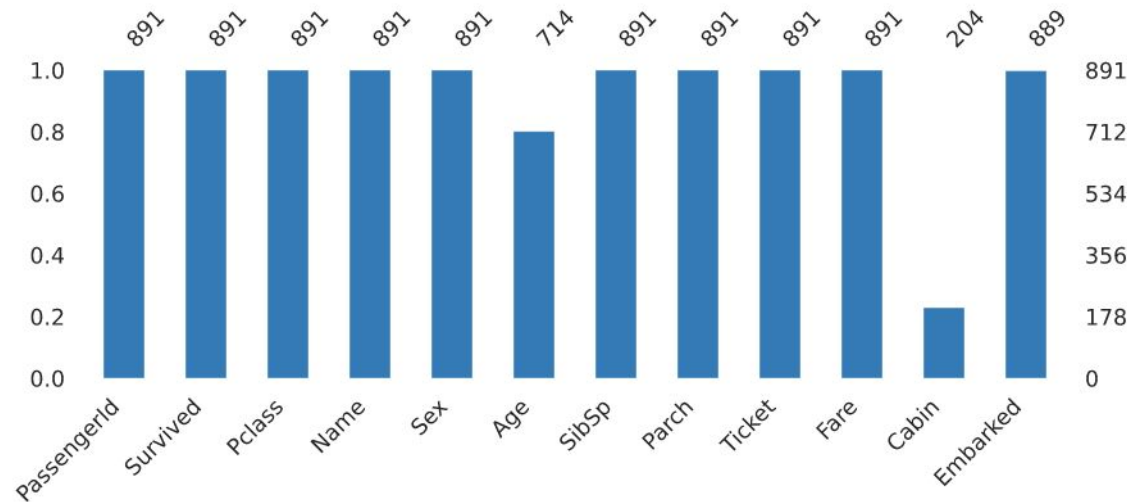
In the further part there are all possible correlations, together with the available description (after clicking on 'toggle correlation descriptions'). We can immediately notice that the ticket fee correlates with the information that someone has survived.





Missing values and "sample"

At the very end, there is information about missing values and an illustrative sample of data in tabular form.

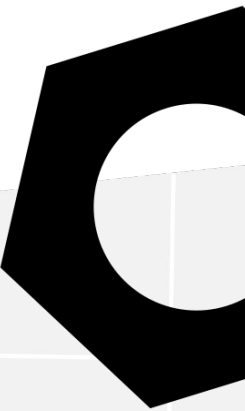


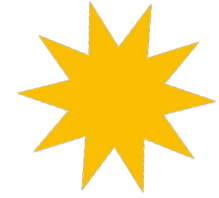
First rows

	PassengerId	Survived	Pclass	Name	Sex	Age
0	1	0	3	Braund, Mr. Owen Harris	male	22.0
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Thayer)	female	38.0
2	3	1	3	Heikkinen, Miss. Laina	female	26.0
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0
4	5	0	3	Allen, Mr. William Henry	male	35.0
5	6	0	3	Moran, Mr. James	male	NaN
6	7	0	1	McCarthy, Mr. Timothy J	male	54.0



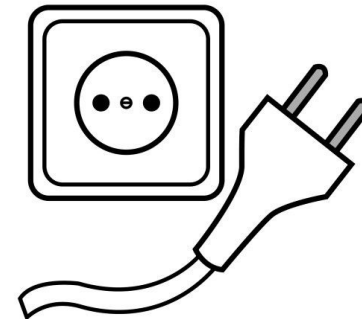
API – connectivity between applications



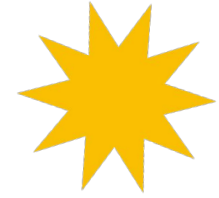


What is API?

API (Application Programming Interface) stands for an interface intended for programmers creating applications. Most large companies create such interfaces for their customers as well as for internal use. You've probably used the API without even knowing it, such as commenting on a Facebook post on a third-party website, paying for trendy pink shoes at an exclusive online store using your PayPal account, or even using the store's search engine or using Google Maps to check its location..



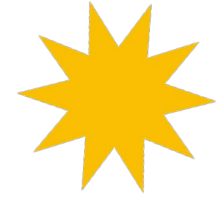
What can be downloaded from the API?



The API can be used primarily to retrieve up-to-date and historical information, for example about:

- weather
- traffic
- flights
- the latest news
- tweets
- maps
- movies
- many many others

You have to remember that most of the good APIs are paid, or have a free trial / demo version. We always need to have an API key generated for us.



A simple example – weather data

We will use the OpenWeathermap service: <https://openweathermap.org>

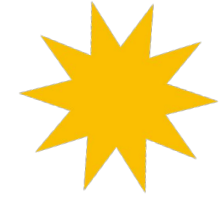
To obtain the key, you must register – the key should be sent to the e-mail.

As part of the free version, we can make 60 queries per minute, check the current weather and its forecasts up to 7 days for the whole world.

Data is returned as JSON or XML.

Very good documentation is attached to the API:

<https://openweathermap.org/current>

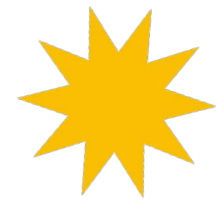


JSON?

JSON (JavaScript Object Notation) is an open format for saving data structures. Its purpose is most often data exchange between applications. JSON consists of attribute-value pairs and array data types. JSON notation is similar to JavaScript objects or Python dictionaries.

Its advantages are popularity, simplicity of operation, brevity of the syntax, and as the data is written to the text – after formatting it is readable for humans. JSON can be an alternative to XML or CSV. Save JSON files with the .json extension. The JSON logo is a Mobius torus.

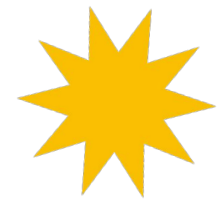
```
{  
  "title" : "This Is What You Came For",  
  "artist" : "Calvin Harris",  
  "length" : "3:41",  
  "released" : "2016.04.29"  
}
```



XML?

XML (Extensible Markup Language) – a universal markup language designed to represent various data in a structured way. It is platform independent, which enables easy document exchange between heterogeneous systems and has significantly contributed to the popularity of this language in the Internet age.

```
<?xml version="1.0" encoding="iso-8859-8" standalone="yes" ?>
<CURRENCIES>
  <LAST_UPDATE>2004-07-29</LAST_UPDATE>
  <CURRENCY>
    <NAME>dollar</NAME>
    <UNIT>1</UNIT>
    <CURRENCYCODE>USD</CURRENCYCODE>
    <COUNTRY>USA</COUNTRY>
    <RATE>4.527</RATE>
    <CHANGE>0.044</CHANGE>
  </CURRENCY>
  <CURRENCY>
    <NAME>euro</NAME>
    <UNIT>1</UNIT>
    <CURRENCYCODE>EUR</CURRENCYCODE>
    <COUNTRY>European Monetary Union</COUNTRY>
    <RATE>5.4417</RATE>
    <CHANGE>-0.013</CHANGE>
  </CURRENCY>
</CURRENCIES>
```

A simple example – weather data

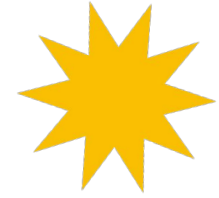
requests – library for API polling

```
import requests

url = 'http://api.openweathermap.org/data/2.5/weather?q=Wroclaw&appid=41e78d83e31864183bc6d74812722732&units=metric'

response = requests.get(url)
```

We ask the API about the current weather for Wrocław in metric units.



A simple example – weather data

```
print(response) #will display the status
```

```
# displaying the status and perhaps an error code
```

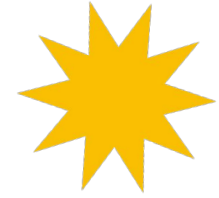
```
print(f"Request returned {response.status_code} : '{response.reason}'")
```

```
# reading the data from the response (parsing as json
```

```
payload = response.json()
```

```
<Response [200]>
```

```
Request returned 200 : 'OK'
```



A simple example – weather data

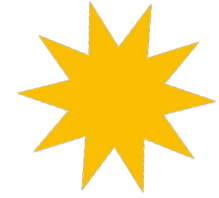
Displaying data:

```
print(payload)
```

```
{'coord': {'lon': 17.03, 'lat': 51.1}, 'weather': [{'id': 803, 'main': 'Clouds', 'description':
```

Prettier format – pprint library

Displaying keys: payload.keys()



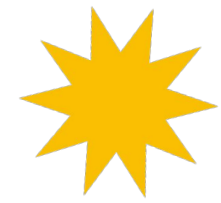
A simple example – weather data

Viewing historical data as part of the free API:

<https://openweathermap.org/api/one-call-api#history>

Weather forecast:

<https://openweathermap.org/api/one-call-api>

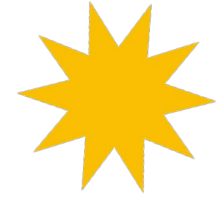


A simple example – weather data

Saving data from API to the dataframe:

```
import pandas as pd
df = pd.DataFrame(columns=['datetime', 'temp', 'pressure'])
for i in range(len(payload['hourly'])):
    df.loc[i]=[pd.to_datetime(payload['hourly'][i]['dt'],unit='s'), payload['hourly'][i]['temp'], payload['hourly'][i]['pressure']]
print(df)
```

	datetime	temp	pressure
0	2020-10-15 13:00:00	19.72	1011
1	2020-10-15 14:00:00	19.68	1012
2	2020-10-15 15:00:00	19.85	1013
3	2020-10-15 16:00:00	18.55	1014
4	2020-10-15 17:00:00	16.19	1015
5	2020-10-15 18:00:00	15.86	1015
6	2020-10-15 19:00:00	15.96	1016
7	2020-10-15 20:00:00	15.72	1016

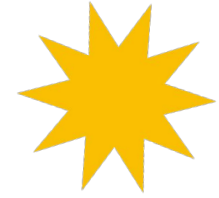


A simple example – weather data

Save to file and download from Google Colabulatory:

```
from google.colab import files  
df.to_csv('weather.csv')  
files.download('weather.csv')
```

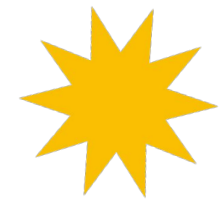
Maybe more interesting – stock market data



You can use many APIs available. I recommend yahoo-fin:
http://theautomatic.net/yahoo_fin-documentation/

```
!pip install yahoo-fin  
!pip install requests-html
```

```
import yahoo_fin.stock_info as yh
```



Maybe more interesting – stock market data

The biggest increases on a given day:

```
response = yh.get_day_gainers()
pp.pprint(response)
```

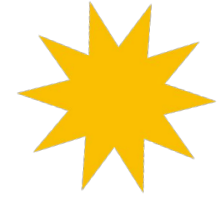
	Symbol	Name	...	Market Cap	PE Ratio (TTM)
0	CIT	CIT Group Inc.	...	2.386000e+09	NaN
1	NAV	Navistar International Corporation	...	4.268000e+09	NaN
2	FLEX	Flex Ltd.	...	7.220000e+09	80.98
3	RAZFF	Razer Inc.	...	2.778000e+09	NaN
4	BMI	Badger Meter, Inc.	...	2.341000e+09	50.89
..
64	CGNX	Cognex Corporation	...	1.200300e+10	86.71
65	MCRB	Seres Therapeutics, Inc.	...	3.000000e+09	NaN
66	VLVLY	AB Volvo (publ)	...	4.190200e+10	13.54
67	ACAD	ACADIA Pharmaceuticals Inc.	...	6.991000e+09	NaN
68	CHWY	Chewy, Inc.	...	2.753400e+10	NaN



Maybe more interesting – stock market data

Objective: Get netflix stock data for the last 30 days via API. Calculate the average opening price, the average of the high and low prices of the day, and calculate the percentage increase (or decrease) of the company's shares during the month.

Maybe more interesting – stock market data



```
response = yh.get_data('nflx', start_date = '01/09/2020', end_date = '30/09/2020', index_as_date = True,
interval = 'id')
print("Average value of open: ", response.open.mean())
print("Average highest daily value: " response.high.mean())
print("Average lowest daily value: " response.low.mean())
print("Percentage increase in the company's shares: ", (response.open[-1]-response.open[0] * 100, "%")
```

Average value of open: 427.887157836247

Average highest daily value: 436.24497277098277

Average lowest daily value: 419.83683168171535

Percentage increase in the company's shares: 43.12865497076023 %