Data mining in geoinformatics

Lecture notes for students

in Cartography and Geoinformatics

Dr. Ungvári Zsuzsanna, PhD
assistant professor
Eötvös Loránd University Budapest, Faculty of Informatics,
Institute of Cartography and Geoinformatics
Budapest, 2025

Contents

Requirements, prerequisites

This course focuses on data mining in spatial datasets. Thus it is different fromIt will not be the same as a standard data mining course.

When designing the course material, I focused on the recent challenges of cartography and geoinformatics. To create visualizations from spatial datasets, you often need some programming to preprocess these data. These tasks are performed with some programming in Python on this course.

To successfully complete this course, you should have basic knowledge of QGIS, Python, and SQL.

Recommended literature

If you want to learn more about data mining, here is a suggested reading.

Pang-Ning Tan, Michael Steinbach, Vipin Kumar: Introduction to Data Mining. Pearson Education Limited, PDF book, 2014.

Reviewed by Dr. Gede Mátyás.

1. Chapter 1. Introduction to data mining

Data mining is a young science: it was brought to life by big databases. Data mining includes all the tools, methods, and algorithms to process large databases and automate certain processing steps. The result allows to get new information and conclusions from the original dataset.

The data mining process can be split to the following subprocesses:

Input data → Pre-processing → Data mining methods → Postprocessing → Information/result

Selection, dimensional reduction, conversion	Visualization, finding and filtering patterns, data analysis

The purpose of pre-processing is to bring data stored in different formats into a single system, making them suitable for further processing. Pre-processing includes cleaning raw dataset, putting it into a common format, reducing noise, removing or correcting errors, filtering the dataset by different conditions.

Data mining includes the processing of the data using mathematical methods. In the post-processing phase, experts share the result as maps or diagrams as well.

Data Science

Data science is an interdisciplinary field that uses scientific methods, algorithms, processes, and systems to extract insights and knowledge from structured and unstructured data. It combines elements of statistics, mathematics, computer science, and domain expertise to analyze and interpret complex data.

Big Data

Big Data refers to extremely large and complex datasets that are difficult to process using traditional data management tools. These datasets are generated at high volume from various sources such as social media, IoT devices, sensors, and online transactions. This term includes not only the large amounts of data, but the hardware and sorfware infrastructure, and even the data processing methods. Big data can be stored in the cloud, or in distributed systems. A distributed system is a network of independent computers that work together as a single system to achieve a common goal. These systems share resources, communicate over a network, and are designed for scalability, reliability, and fault tolerance.

What will we be dealing with?

- **Data extraction from online databases**, with a special focus on pre-processing.
- **Conversion of textual and other formats** (in the case of location-related data) into geospatial data types.
- **Extracting information from pre-processed data** using some basic statistical methods.
- **Post-processing and data visualization.**
- **Programming tasks in Python environment**, including data preparation.
- **Geoprocessing tasks in QGIS environment.**

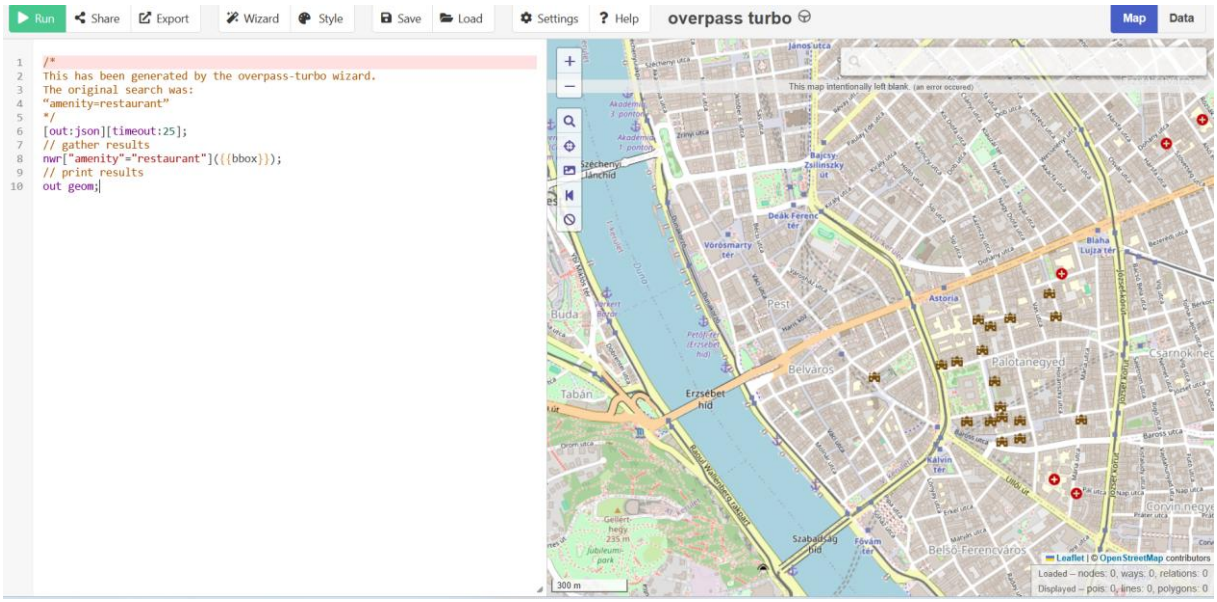
Chapter 2. OVERPASS TURBO API

Overpass Turbo API is a website where users can run queries on the OpenStreetMap (OSM) database. The response is displayed on a map and in a data window, and the result can be downloaded in several formats. Overpass Turbo uses a special query language called Overpass QL (Query Language). Overpass can also work with Overpass XML, but this language is much more complex. Queries sent a HTTP GET requests over the network. The documentation is available here:

https://wiki.openstreetmap.org/wiki/Overpass_turbo

How to begin to use Overpass Turbo API?

Open the Overpass Turbo website from here: https://overpass-turbo.eu/



The website consists of two main parts: on the left, you can see the Code Editor window, and on the right, at map is displayed.
The menu system includes the **Wizard** and **Export** functions. In Overpass Turbo API, queries can be constructed in two ways: you can write a script in **Overpass QL** or use the **Wizard** in parallel.
In the next subchapters, we will explore both options together.

Exporting data

To export the results of queries, use the **Export** menu. You can save the data in **GeoJSON**, **GPX**, **KML**, and **OSM database (XML)** formats.

GeoJSON Format: It is a text-based GIS format.
GeoJSON supports the following geometry types: **Point, LineString, Polygon, MultiPoint, MultiLineString,** and **MultiPolygon**. Geometric objects with additional properties are called **Feature** objects. A single text file can store different geometry types of data.

Here you can read more about the GeoJSON format: https://geojson.org/

An exaple for the file structure:

```
{
  "type": "Feature",
  "geometry": {
    "type": "Point",
    "coordinates": [125.6, 10.1]
  },
  "properties": {
    "name": "Dinagat Islands"
  }
}
```

KML: Keyhole Markup Language (KML) is an XML notation for expressing geographic annotation and visualization within two-dimensional maps and three-dimensional Earth browsers. KML was developed for use with Google Earth. A KML file specifies a set of features (placemarks, images, polylines, polygons, 3D models, textual descriptions, etc.) that can be displayed on maps in geospatial software implementing the KML encoding. Geometries are described by longitude and latitude coordinates. Other data can make a view more specific, such as tilt, heading, or altitude, which together define a "camera view" along with a timestamp or timespan.

An example for KML structure

```
<?xml version="1.0" encoding="UTF-8"?>
<kml xmlns="http://www.opengis.net/kml/2.2">
<Document>
<Placemark>
  <name>New York City</name>
  <description>New York City</description>
  <Point>
    <coordinates>-74.006393,40.714172,0</coordinates>
  </Point>
</Placemark>
</Document>
</kml>
```

GPX: GPS Exchange Format (GPX) is an XML schema designed as a common GPS data format for software applications. It can be used to describe waypoints, tracks, and routes.

An example for GPX format:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<gpx xmlns="http://www.topografix.com/GPX/1/1" version="1.1"
creator="Wikipedia"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.topografix.com/GPX/1/1
http://www.topografix.com/GPX/1/1/gpx.xsd">
 <!-- Comments look like this -->
<wpt lat="52.518611" lon="13.376111">
  <ele>35.0</ele>
  <time>2011-12-31T23:59:59Z</time>
  <name>Reichstag (Berlin)</name>
  <sym>City</sym>
 </wpt>
</gpx>
```

Basics of Overpass QL

Data types in OSM

OSM data can be nodes, ways, and relations (rel). The attributes of these geometry types are called tags. Tags are, in fact, key-value pairs.

Read more about nodes: https://wiki.openstreetmap.org/wiki/Node

Nodes: nodes describe the elementary points that build up any more complex geometry.

Here is an example:

```
<node id="6454986742" lat="46.0725973" lon="18.2051926">
    <tag k="entrance" v="main"/>
```

another example:

```
<node id="1760195778" lat="46.0690616" lon="18.2160408">
    <tag k="amenity" v="restaurant"/>
    <tag k="dog" v="yes"/>
    <tag k="name" v="Szent György Fogadó"/>
    <tag k="toilets:wheelchair" v="yes"/>
    <tag k="website" v="http://www.szentgyorgyfogado.hu/"/>
    <tag k="wheelchair" v="yes"/>
```

Way: A way is one of the fundamental elements of the map. In everyday language, it is a line. A way normally represents a linear feature on the ground (such as a road, wall, or river). Technically, a way is an ordered list of nodes. It normally also has at least one tag or participates in a relation. A way can be open or closed.

https://wiki.openstreetmap.org/wiki/Way

Open way: In an open way (a linear representation of a feature), the first and last node are not identical. Common examples of linear representation with open ways include most roads, streams, and railway lines, because these start at one place and finish at another.

```
<way id="375355485">
    <nd ref="27313594"/>
    <nd ref="4042642645"/>
    <nd ref="3787287460"/>
    <tag k="HU:ed_direction" v="forward"/>
    <tag k="highway" v="primary"/>
    <tag k="maxspeed" v="90"/>
    <tag k="ref" v="71"/>
    <tag k="ref:HU:edid" v="71u2k276m"/>
    <tag k="source:maxspeed" v="HU:rural"/>
    <tag k="surface" v="asphalt"/>
    <tag k="toll:hgv" v="yes"/>
  </way>
```

Closed way: In a closed way, the last node is identical to the first node. A closed way may be interpreted either as a closed polyline (a linear representation of a feature) or as an area, or both, depending on its tags and the tags of containning relations. A typical example is a roundabout.

Area: An area (also polygon) is an enclosed filled area of territory defined as a closed way. Most closed ways are considered to be areas even without an *area=yes* tag. For example, a park, forest area, or building.

```
<way id="143854005">
```

```
    <nd ref="1574148556"/> <nd ref="1574148670"/>
    <nd ref="1574148682"/> <nd ref="2621432173"/>
    <nd ref="2621432172"/> <nd ref="2621432167"/>
    <nd ref="2621432168"/> <nd ref="1574148558"/>
    <nd ref="1574148556"/>
    <tag k="building" v="university"/>
    <tag k="name" v="K-épület"/>
  </way>
```

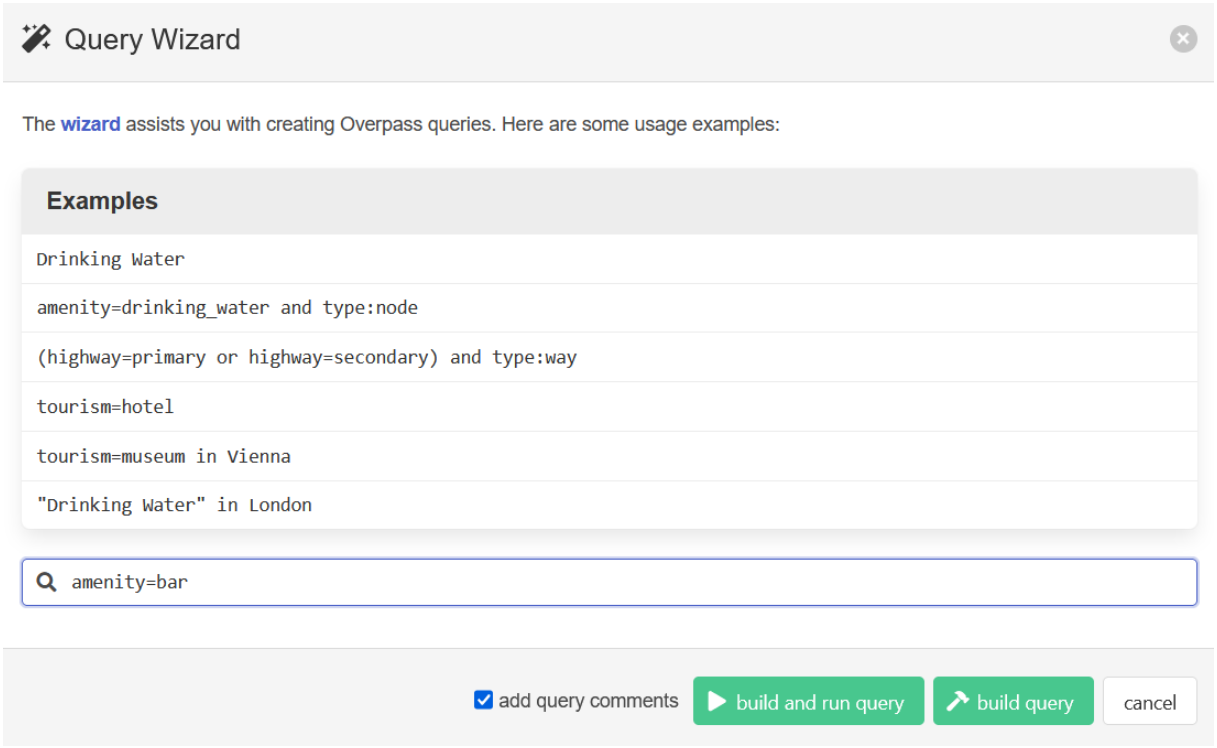
Rel (relation). Relations are structured collections of objects – nodes, ways, and other relations. Relations are groups, for example, bus routes.

https://wiki.openstreetmap.org/wiki/Relation

```
<relation id="9684822">
    <member type="node" ref="1382931855" role="platform"/>
    …
    <member type="node" ref="1382931862" role="platform"/>
    <member type="way" ref="819269604" role=""/>
    …
    <member type="way" ref="819269609" role=""/>
    <tag k="from" v="Kerepes HÉV-állomás"/>
    <tag k="name" v="Helyi járat &quot;A&quot;: Kerepes HÉV-állomás
=&gt; Patkó Csárda =&gt; Iskola =&gt; Berzsenyi uca =&gt;
Szilasliget-Kemping"/>
    <tag k="operator" v="Regio 2007 Kft."/>
    <tag k="public_transport:version" v="2"/>
    <tag k="ref" v="Helyi járat"/>
    <tag k="route" v="bus"/>
    <tag k="to" v="Szilasliget-Kemping"/>
    <tag k="type" v="route"/>
    <tag k="via" v="Patkó Csárda -&gt; Iskola -&gt; Berzsenyi
uca"/>
  </relation>
```

A basic query using Overpass QL

If you want to retrieve data from OSM, you have to write queries. In a query, you usually specify all geometry types in OSM. For example, if you want to know the location of bars in a small area (let's move the view to the city center of Budapest), *"amenity"* is the key, and *"bar"* is the value. You can build the query in the Wizard: amenity=bar. Click on Build and run the query.



If the query is built, its text is available in the code editor. It can be viewed in the following way:

```
(
node["amenity"="bar"]({{bbox}});
way["amenity"="bar"]({{bbox}});
relation["amenity"="bar"]({{bbox}});
);
out;
```

In 2023, NWR was introduced to make shorter statements. So, the query below is same as above, but in a shorter form.

```
nwr["amenity"="bar"]({{bbox}});
out;
```

NWR: A new element in Overpass QL. It includes nodes, ways and relations. Because of its simplicity, I prefer this.

How to give the area of the query?

Let's write a query for restaurants in the Wizard: *amenity=restaur*ant. If you do not define the exact target area, the query will run automatically for the map canvas extent/bounding box.

```
(
  // query part for: "amenity=restaurant"→ This is a comment!
  node["amenity"="restaurant"]({{bbox}});
  way["amenity"="restaurant"]({{bbox}});
  relation["amenity"="restaurant"]({{bbox}});
);
// print results
out body;
```

bbox (*bounding box*). The extent of the canvas is a rectangle. The definition looks like this:

```
({{bbox}});
```

If you want, you can specify a bounding box manually by coordinates. First, provide the southern (bottom) edge, then the western (left) edge, followed by the northern (top) edge, and finally the eastern (right) edge. Separate each coordinate with a comma, and enclose all coordinates in parentheses (round brackets). End the line with a semicolon.

```
(47.4, 18.5, 47.8, 19.3);
```

If the area divided into two parts by the meridian 180°, then divide your query into two parts as well:

```
1   (
2     /*your query here*/(51.0,  7.0, 52.0, 180); /* West side of the antimeridian (up to the positive longitude 180°E) */
3     /*your query here*/(51.0, -180, 52.0, 8.0); /* East side of the antimeridian (from  the negative longitude 180°W) */
4   );
```

Searching radius

In this case, you have to provide a coordinate and a search distance called radius, and the query will run in a circle-shaped area.

```
(around: 2000, 47.4866, 19.0567);
```

Alternatively, you can give a location instead of coordinates. Let's find restaurants within a 500-meter circle of Budapest Deák Ferenc tér 1. In this case, you are performing geocoding and using the *geocodeCoords* parameter.

```
{{radius=500}}
(
node["amenity"="restaurant"](around:{{radius}},{{geocodeCoords:Budapest Deák Ferenc tér 1}});
); out;
```

It is also possible to use an irregular area as a bounding area (mask layer), e.g., a country, a county, or other administrative areas. In this case, you use geocoding again, in the following way:

```
{{geocodeArea:Ercsi}}->.searchArea
node["amenity"="restaurant"](area.searchArea);
```

The *geocodeArea* variable was renamed to *.searchArea*, and it was used as a bounding area.

If you use the Wizard, query the restaurants for Budapest in following way: amenity=restaurant in Budapest.

```
[out:json][timeout:25];
// fetch area "Budapest" to search in
```

```
{{geocodeArea:Budapest}}->.searchArea;
// gather results
nwr["amenity"="bar"](area.searchArea);
// print results
out geom;
```

The OpenStreetMap Wikipedia page

It is important to know the source, where you can browse/check every features. Please visit the website, and study it! This page explains how physical features on the ground, such as roads or buildings, are represented in OpenStreetMap using tags attached to its basic data structures (nodes, ways, and relations). Each tag describes a geographic attribute of the feature being shown by that specific node, way, or relation.

https://wiki.openstreetmap.org/wiki/Map_features

Giving the values of the keys, types of relations

If you search for an exact key–value pair in OSM, use the equal sign.

```
["building"="castle"]
```

In SQL, there is a LIKE operator, which is used in a WHERE clause to search for a specified pattern in a column. In OSM and Overpass QL, there is a very similar way to find patterns in the text. If the value is partially given, for example, you know that the word contains *'bicycle'* (e.g., bicycle_parking, bicycle_repair_station, bicycle_rental), but the other part of the value is not known, or you do not want to specify it exactly. In this case, use amenity:bicycle *or:*

```
["amenity"~"bicycle"]
```

If you want to query all values in a key in Wizard use *amenity=**. In the Overpass QL query, you get simply this:

```
["amenity"]
```

If you want to make an inverted selection for elements, specify a value, and use != operator. Retrive all data except bicycle_rentals in the view of ELTE Lágymányos Campus, where the *amenity!=bicycle_rental.*

```
["amenity"!="bicycle_rental"]
```

Be careful! Use a very small area in the bounding box because many results will be retrieved! The situation is same in the following query. *amenity!~bicycle* returns data from every category for the selected area (except bicyle-related things).

```
["amenity"!~"bicycle"]
```

If you want to query all key and data for the target area, except the values in a given key (e.g. amenity), use this: *amenity!=**,

```
["amenity"!~".*"]
```

The timeout

```
[timeout:180]
```

The maximum allowed time for the query in seconds. If it is exceeded, the query will stop. It is not an obligatory part of the query.

[out:json] or [out:xml]

The query can be opened by *[out:json]***,** and the retrived data will be in JSON format. If you do not add *[out:json]*, the data will retrived in XML.

```
[out:json][timeout:25]; //JSON data format
 (
  // query part for: "amenity=restaurant"
  node["amenity"="restaurant"]({{bbox}});
  way["amenity"="restaurant"]({{bbox}});
  relation["amenity"="restaurant"]({{bbox}});
);
out body;
```


```
{
  "version": 0.6,
  "generator": "Overpass API 0.7.62.5 1bd436f1",
  "osm3s": {
    "timestamp_osm_base": "2025-05-11T19:33:02Z",
    "copyright": "The data included in this document is from www.openstreetmap.org. The data is made available under ODbL."
  },
  "elements": [

{
  "type": "node",
  "id": 5351760340,
  "lat": 47.4978111,
  "lon": 19.0671955,
  "tags": {
    "addr:city": "Budapest",
    "addr:housenumber": "13",
    "addr:postcode": "1072",
    "addr:street": "Klauzál utca",
    "amenity": "restaurant",
    "name": "Barack & Szilva",
    "opening_hours": "Mo-Sa 18:00-24:00",
    "phone": "+36 1 798 8285;+36 30 258 0965",
    "website": "https://barackesszilva.hu/",
    "wheelchair": "yes"
  }
}

  ]
}
```


```
[timeout:25]; //XML data format
 (
  // query part for: "amenity=restaurant"
  node["amenity"="restaurant"]({{bbox}});
  way["amenity"="restaurant"]({{bbox}});
  relation["amenity"="restaurant"]({{bbox}});
);
out body;
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<osm version="0.6" generator="Overpass API 0.7.62.5 1bd436f1">
<note>The data included in this document is from www.openstreetmap.org. The data is made available under ODbL.</note>
<meta osm_base="2025-05-11T19:35:03Z"/>

  <node id="5351760340" lat="47.4978111" lon="19.0671955">
    <tag k="addr:city" v="Budapest"/>
    <tag k="addr:housenumber" v="13"/>
    <tag k="addr:postcode" v="1072"/>
    <tag k="addr:street" v="Klauzál utca"/>
    <tag k="amenity" v="restaurant"/>
    <tag k="name" v="Barack &amp;amp; Szilva"/>
    <tag k="opening_hours" v="Mo-Sa 18:00-24:00"/>
    <tag k="phone" v="+36 1 798 8285;+36 30 258 0965"/>
    <tag k="website" v="https://barackesszilva.hu/"/>
    <tag k="wheelchair" v="yes"/>
  </node>

</osm>

```

Closing the query: out; and its mutations

Every query has to be closed with **out;**. This is the shortest form to close them. If you want to specify how to plot the data in the data view, you can use the following supplements:

out body; Print all information necessary to use the data.

out skel; Print the minimum information necessary for geometry: for nodes: id and coordinates; for ways: id and the ids of its member nodes; for relations: id of the relation, and the id, type, and role of all of its members.

out tags; Print only ids and tags for each element and not coordinates or members.

out ids; Print only the ids of the elements in the set

See more here:

https://wiki.openstreetmap.org/wiki/Overpass_API/Overpass_QL#Output_format_(out:)

Recursivity

The *recurse down* standalone query is written as a single greater than. It takes an input set. It produces a result set. Its result set is composed of:

- all **nodes** that are part of a **way** which appears in the **input** set; plus
- all **nodes and ways** that are members of a **relation** which appears in the **input** set; plus
- all **nodes** that are part of a **way** which appears in the **result** set

In particular, you can change the input and/or result set with the same notation as for the *recurse up* standalone query.

```
>;
```

For further syntax check: https://wiki.openstreetmap.org/wiki/Overpass_API/Overpass_QL

Practice

The following practices work in Overpass Turbo QL, in the Overpass Turbo API. Light blue text hightlights can be written in the Wizard tool. Here is a short description, how to use the Wizard:

https://wiki.openstreetmap.org/wiki/Overpass_turbo/Wizard

There may be some changes in syntax. Please open Overpass Turbo in a new tab of your browser: https://overpass-turbo.eu/

Unit 1. Exercise the different views.

Task 1.1

Set the map view to the centre of Budapest, and find the bars.

The bars are *amenity* features. In Wizard type: *amenity=bar*. In the code editor, you can build the query.

```
(
node["amenity"="bar"]({{bbox}});
way["amenity"="bar"]({{bbox}});
relation["amenity"="bar"]({{bbox}});
);
out;
```

or:

```
nwr["amenity"="bar"]({{bbox}});
out;
```

The viewing area is given in round bracket as a the canvas extent: *({{bbox}});*

Task 1.2

Now, query all bars in Budapest and its sorrounding. The coordinates of the bounding box are: 47.4, 18.7, 47.8, 19.2.

```
nwr["amenity"="bar"] (47.4, 18.7, 47.8, 19.2);
out;
```

Task 1.3

Let's include in the query the availability of the bars with wheelchair. The bounding box is same as in the task 1.2.

In the Wizard, *amenity=bar and wheelchair=yes* have to be written. You search for two tags of one feature, therefore the logical relation is AND.

```
nwr[„amenity”="bar"]
     [wheelchair=yes](47.4, 18.7, 47.8, 19.2);
 out;
```

Task 1.4

Query the bars whose address contains Budapest! *(addr:city)*

In the Wizard: *amenity=bar AND addr:city=Budapest*. You search for two tags of one feature, therefore the logical relation is AND.

```
node["amenity"="bar"]
      ["addr:city"="Budapest"]
     (47.4, 18.7, 47.8, 19.2);
 out;
```

Task 1.5

Look for the bars near you (search radius is 2 km). Your location: 47.4866° és 19.0567°. The bars have to be wheelchair accessible.

```
 nwr["amenity"="bar"]
     ["wheelchair"="yes"]
      (around:2000,47.4866, 19.0567);
  out body;
```

Task 1.6

Look for the bars near you (search radius is 500 m). Your location is in Budapest Deák Ferenc tér 1.

```
[out:json][timeout:25];
{{radius=500}}
 (
node["amenity"="restaurant"](around:{{radius}},{{geocodeCoords:Buda
pest Deák Ferenc tér 1}});
way["amenity"="restaurant"](around:{{radius}},{{geocodeCoords:Budap
est Deák Ferenc tér 1}});
relation["amenity"="restaurant"](around:{{radius}},{{geocodeCoords:
Budapest Deák Ferenc tér 1}});
);
out body;
```

or:

```
{{radius=500}}

nwr["amenity"="restaurant"](around:{{radius}},{{geocodeCoords:Budap
est Deák Ferenc tér 1}});
out body;
```

Task 1.7*

Add a new style to the nodes in Task 1.6. Every nodes has a red fill color, and the label is written to the left of the node. Font-size is 16 pt.

The nodes and the ways can be styled by the MapCSS language. This language has only a few options for styling; therefore you have only one task in this topic. But if you want to read more about MapCSS, please visit:

https://wiki.openstreetmap.org/wiki/MapCSS/Examples

```
 node["amenity"="bar"]
      ["wheelchair"="yes"]
      ["addr:city"="Budapest"]
     (47.4, 18.7, 47.8, 19.2);
{{style:
    node { fill-color: red; color: blue, fill-opacity 1; font-
size:16; text: name; text-position:left; }
}}
 out body;
```

Task 1.8

Query two features at the same time in Budapest city center. These features are the restaurants and bars. Use the OR logical operator!

You have to use a round brackets to group the items that you query.

In the Wizard: *amenity=bar OR amenity=restaurant*.

```
(
  nwr["amenity"="restaurant"]({{bbox}});
  nwr["amenity"="bar"]({{bbox}});
);
out;
```

or:

```
 (
  node["amenity"="restaurant"]({{bbox}});
  way["amenity"="restaurant"]({{bbox}});
  relation["amenity"="restaurant"]({{bbox}});
  node["amenity"="bar"]({{bbox}});
  way["amenity"="bar"]({{bbox}});
  relation["amenity"="bar"]({{bbox}});
);
out body;
```

Unit 2. An exercise on buildings
Task 2.1

Buildings can be retrieved as nodes, ways, and relations. Plot the residential buildings in the city center of Budapest.

In the Wizard: *building=residential*.

```
(
  node["building"="residential"]({{bbox}});
  way["building"="residential"]({{bbox}});
  relation["building"="residential"]({{bbox}});
);
out body;
```

or:

```
nwr["building"="residential"]({{bbox}});
out;
```

Query the university buildings! *building=university or amenity=university*.

```
(
  node["building"="university"]({{bbox}});
  way["building"="university"]({{bbox}});
  relation["building"="university"]({{bbox}});
);
out body;
```

or: (to get the right result, use the recurse up sign!)

```
nwr["building"="university"]({{bbox}});
nwr["amenity"="university"]({{bbox}});
>;
out;
```

Unit 3.
Task 3.1

Query the forest areas in Buda Hills. Set the map view to the western part of Budapest!

In the Wizard: *landuse=forest*

```
(
  node["landuse"="forest"]({{bbox}});
  way["landuse"="forest"]({{bbox}});
  relation["landuse"="forest"]({{bbox}});
);
>;
out body;
```

or:

```
  nwr["landuse"="forest"]({{bbox}});
 >;
out body;
```

Task 3.2

Query the forest areas in the city of Ercsi.

In the Wizard: *landuse=forest in Ercsi*

```
{{geocodeArea:Ercsi}}->.searchArea;
 (
  node["landuse"="forest"](area.searchArea);
  way["landuse"="forest"](area.searchArea);
  relation["landuse"="forest"](area.searchArea);
);
out body;
```

or:

```
{{geocodeArea:Ercsi}}->.searchArea;
  nwr["landuse"="forest"]({{bbox}});
 >;
out body;
```

Task 3.3

Let's find the castles in Nógrád county! Castles can be found both building and in historic categories as well.

building=castle or building historic=castle in Nógrád

```
{{geocodeArea:Nógrád}}->.searchArea;
 (
  node["building"="castle"](area.searchArea);
  way["building"="castle"](area.searchArea);
  relation["building"="castle"](area.searchArea);

  node["historic"="castle"](area.searchArea);
  way["historic"="castle"](area.searchArea);
  relation["historic"="castle"](area.searchArea);
);
out body;
```

or:

```
{{geocodeArea:Nógrád}}->.searchArea;
 (
  nwr["building"="castle"](area.searchArea);
  nwr["historic"="castle"](area.searchArea);
);>;
out body;
```

or:

```
area
  ["boundary"="administrative"]
  ["admin_level"="6"]
  ["name"="Nógrád vármegye"]->.a;
out body;
 (
```

```
  nwr["historic"="castle"](area.a);
  nwr["building"="castle"](area.a);
);>;
out body;
```

Unit 4. Administrative boundaries

Task 4.1

Query the data of Hungary, set the map view to the Carpathian Basin. The data is retrieved in the place tag as nodes (it does not contain the boundary). The tag '*place*' contains the name of the country in several languages.

```
(
  node["place"="country"]["name"="Magyarország"]({{bbox}});
  way["place"="country"]["name"="Magyarország"]({{bbox}});
  relation["place"="country"]["name"="Magyarország"]({{bbox}});
);
out body;
```

Task 4.2

Query all boundaries from the admin_level 2 to 9 in Hungary. Set the map view to Hungary, then zoom in on a county. If you query the administrative regions, it is necessary to known the admin_level parameter value. To get familiar with the admin_levels in Hungary, see the table below.

In the Wizard: *boundary=administrative and admin_level=?*

For Hungary:

Admin_level	**Administrative region**	**Admin_level**	**Administrative region**
2	country border	6	Counties / catpital city borders – NUTS 3
3	-	7	Districts
4	NUTS 1 borders	8	Settlements
5	Regions - NUTS 2	9	Districts of Budapest

For the rest of the world, check the specific country in the following table, where you can see the admin_level numbers and types (from 2 to 11):

https://wiki.openstreetmap.org/wiki/Tag:boundary%3Dadministrative#admin_level=*_Country_specific_values

Let's start with counties (*admin_level: 6*):

```
nwr["boundary"="administrative"]["admin_level"="6"]({{bbox}});
>;
out body;
```

Now, carry on with the *admin_level 7*, these are the districts.

```
(
node["boundary"="administrative"]["admin_level"="7"]({{bbox}});
way["boundary"="administrative"]["admin_level"="7"]({{bbox}});
relation["boundary"="administrative"]["admin_level"="7"]({{bbox}});
);
>;
out body;
```

Settlement can be found at *admin_level 8*.

```
node["boundary"="administrative"]["admin_level"="8"]({{bbox}});
way["boundary"="administrative"]["admin_level"="8"]({{bbox}});
relation["boundary"="administrative"]["admin_level"="8"]({{bbox}});
);
>;
out body;
```

There are two types of regions in Hungary, corresponding to different NUTS levels. There are 3 regions in the *admin_level 4* (West Hungary, East Hungary, Central Region of Hungary) and 7 regions at *admin_level 5*.

```
 (
node["boundary"="administrative"]["admin_level"="5"]({{bbox}});
way["boundary"="administrative"]["admin_level"="5"]({{bbox}});
relation["boundary"="administrative"]["admin_level"="5"]({{bbox}});
);
>;
out body;
```

National borders are represented at *admin_level 2*.

```
(
node["boundary"="administrative"]["admin_level"="2"]({{bbox}});
way["boundary"="administrative"]["admin_level"="2"]({{bbox}});
relation["boundary"="administrative"]["admin_level"="2"]({{bbox}});
);
>;
out body;
```

Plot the border of the county Nógrád!

```
{{geocodeArea:Nógrád}}->.searchArea;
 (

node["boundary"="administrative"]["admin_level"="6"](area.searchAre
a);
way["boundary"="administrative"]["admin_level"="6"](area.searchArea
);
relation["boundary"="administrative"]["admin_level"="6"](area.searc
hArea);
);
>;
out body;
```

or zoom in to Nógrád and plot its boundary!

```
 (
node["boundary"="administrative"]["admin_level"="6"]["name"="Nógrád
vármegye"]({{bbox}});
```

```
way["boundary"="administrative"]["admin_level"="6"]["name"="Nógrád
vármegye"]({{bbox}});
relation["boundary"="administrative"]["admin_level"="6"]["name"="Nó
grád vármegye"]({{bbox}});
);
>;
out body;
```

Let's find the borders of the village Apc. Village borders are at the *admin_level 8.*

Zoom in to East Hungary:

```
(
node["boundary"="administrative"]["admin_level"="8"]["name"="Apc"](
{{bbox}});
way["boundary"="administrative"]["admin_level"="8"]["name"="Apc"]({
{bbox}});
relation["boundary"="administrative"]["admin_level"="8"]["name"="Ap
c"]({{bbox}});
);
>;
out body;
```

or with geocoding:

```
[out:json][timeout:25];
{{geocodeArea:Apc}}->.searchArea;
 (

node["boundary"="administrative"]["admin_level"="8"](area.searchAre
a);
way["boundary"="administrative"]["admin_level"="8"](area.searchArea
);
relation["boundary"="administrative"]["admin_level"="8"](area.searc
hArea);
);
>;
out body;
```

Unit 5.
Task 5.1

Let's find the bars with the 1117 postal code!

```
area["boundary"="postal_code"]["postal_code"="1117"]->.a;
out;
nwr["amenity"="bar"](area.a);
out body;
```

Task 5.2

Let's find the peaks of the Dolomites!

```
area
  [place=region]
```

```
  ["region:type"="mountain_area"]
  ["name:en"="Dolomites"];
out body;

node
  [natural=peak]
  (area);
out body;
```

Task 5.3

Let’s find the peaks of Nógrád vármegye!

```
area
  ["boundary"="administrative"]
  ["admin_level"="6"]
  ["name"="Nógrád vármegye"];
out body;
node
  [natural=peak]
  (area);
out body qt;
```

Task 5.4

Query all settlements from Nógrád vármegye!

```
 [out:json][timeout:25];
{{geocodeArea:Nógrád}}->.searchArea;


nwr["boundary"="administrative"]["admin_level"="8"](area.searchArea
);
>;
out body;
```

Chapter 3. GTFS specification

The General Transit Feed Specification (GTFS) is an Open Standard used to distribute relevant information about transit systems to riders. It allows public transit agencies to publish their transit data in a format that can be consumed by a wide variety of software applications.

GTFS consists of two main parts: GTFS Schedule and GTFS Realtime.

GTFS Realtime

GTFS Realtime (General Transit Feed Specification Realtime) is an extension of the GTFS (General Transit Feed Specification) format that provides real-time public transit data. It allows transit agencies to share live updates about their services.

Key Components of GTFS Realtime:

- Trip Updates – Provides real-time updates on vehicle delays, cancellations, or changes to scheduled trips.
- Vehicle Positions – Gives the current location, speed, and heading of transit vehicles.
- Service Alerts – Communicates disruptions, detours, or other service changes affecting the transit system.

Use Cases

- Public transit apps (like Google Maps, Transit, Moovit) use GTFS Realtime to show live bus/train locations.
- Smart city and transportation analytics platforms use it to monitor transit performance.
- Developers and researchers analyze it for insights into transit efficiency and rider experience. What information can you get from the GTFS?

GTFS Schedule

GTFS Schedule is a feed specification that defines a common format for static public transportation information. It is composed of a collection of simple files, mostly text files (.txt) that are contained in a single ZIP file.

Each file describes a particular aspect of transit information such as stops, routes, trips, etc. At its most basic form, a GTFS Schedule dataset is composed of 7 files: agency.txt, routes.txt, trips.txt, stops.txt, stop_times.txt, calendar.txt and calendar_dates.txt.

The documentation of GTFS:

https://gtfs.org/documentation/overview/

http://developers.google.com/transit/gtfs/

AGENCY.TXT

A járatüzementető adatai.

agency_id	Identifies a transit brand which is often synonymous with a transit agency.
agency_name	Full name of the agency
agency_lang	Two letters language code.
agency_phone	Phone number
agency_email	Email.
agency_fare_url	Website

STOPS.TXT

stop_id	Identifies a location: stop/platform, station, entrance/exit, generic node or boarding area
stop_name	Name of the stop.
stop_lat & stop_lon	Latitude and longitude of the stop
stop_code	Short text or a number that identifies the location for riders.
location_type	0 or blank: stop; 1: station; 2: entrance/exit; 3:generic; node; 4: boarding area
wheelchair_boarding	wheelchair accessibility

ROUTES.TXT

agency_id	Agency ID
route_short_name	Short name of a route. Often a short, abstract identifier (e.g., "32", "100X", "Green") that riders use to identify a route
route_long_name	Full name of a route. This name is generally more descriptive than the route_short_name and often includes the route's destination or stop.
route_id	Route ID
route_color & route_text_color	Printing colors of the routes in a map

TRIPS.TXT

route_id	Foreign key for route ID
trip_id	Tripm ID

service_id	Identifies a set of dates when service is available for one or more routes.
trip_headsign	Text that appears on signage identifying the trip's destination to riders. This field is recommended for all services with headsign text displayed on the vehicle which may be used to distinguish amongst trips in a route.
direction_id	Indicates the direction of travel for a trip. This field should not be used in routing; it provides a way to separate trips by direction when publishing time tables. Valid options are: 0 - Travel in one direction (e.g. outbound travel). 1 - Travel in the opposite direction (e.g. inbound travel).
block_id	Identifies the block to which the trip belongs. A block consists of a single trip or many sequential trips made using the same vehicle, defined by shared service days and block_id. A block_id may have trips with different service days, making distinct blocks.
shape_id	Identifies a geospatial shape describing the vehicle travel path for a trip.
wheelchair_accessible	Wheelchair accessibility

STOP_TIMES.TXT

trip_id	Foreign key for trip ID
stop_id	Foreign key for Stop ID
arrival_time & departure time	Arrival and departure time in the stop
stop_headsign	Text that appears on signage identifying the trip's destination to riders.
stop_sequence	Order of stops
shape_dist_traveled	Actual distance traveled along the associated shape, from the first stop to the stop specified in this record. This field specifies how much of the shape to draw between any two stops during a trip. (in meter)

SHAPES.TXT

shape_id	Id of the shape

`shape_pt_sequence`	Sequence in which the shape points connect to form the shape. Values must increase along the trip but do not need to be consecutive.
`shape_pt_lat` & `shape_pt_lon`	Latitude and longitude of a shape
`shape_dist_traveled`	Actual distance traveled along the shape from the first shape point to the point specified in this record. Used by trip planners to show the correct portion of the shape on a map (in meter).

FEED_INFO.TXT

The file contains information about the dataset itself, rather than the services that the dataset describes. In some cases, the publisher of the dataset is a different entity than any of the agencies.

GTFS text files are essentially tables (similar to those in database management systems). They contain many primary and foreign keys, which connect the files. If you want to work with these text files, it is easiest to import them into QGIS or another database management system (e.g., PostgreSQL). Below, you will see some examples of these solutions.

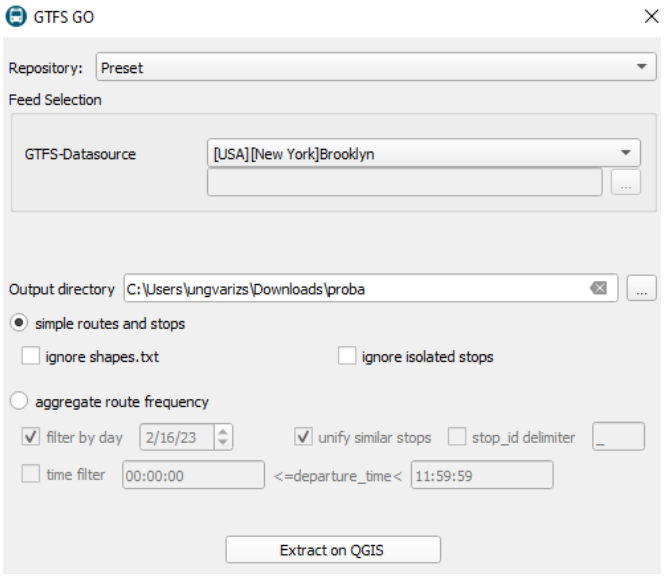
First, get familiar with the tables:

- The *stops* can be found in the *stops.txt* file with coordinates.
- The path or route of the bus/tram/underground railways, etc., can be found in the *shapes* table. According to the *shape_id* field, you can generate the polyline for the path.
- The *routes.txt* file contains the name/number of the route in the *route_short_name* column.
- The "Joker" table is the *trips* table. It connects *routes, shapes*, and *stop_times* tables with foreign keys. The *trip_headsign* field contains the text or label on the front of the bus/tram, etc. You can also use the *route_short_name* from routes.
- *Stop_times* can be connected to *trips*, allowing you to query the arrival and departure times of a route.

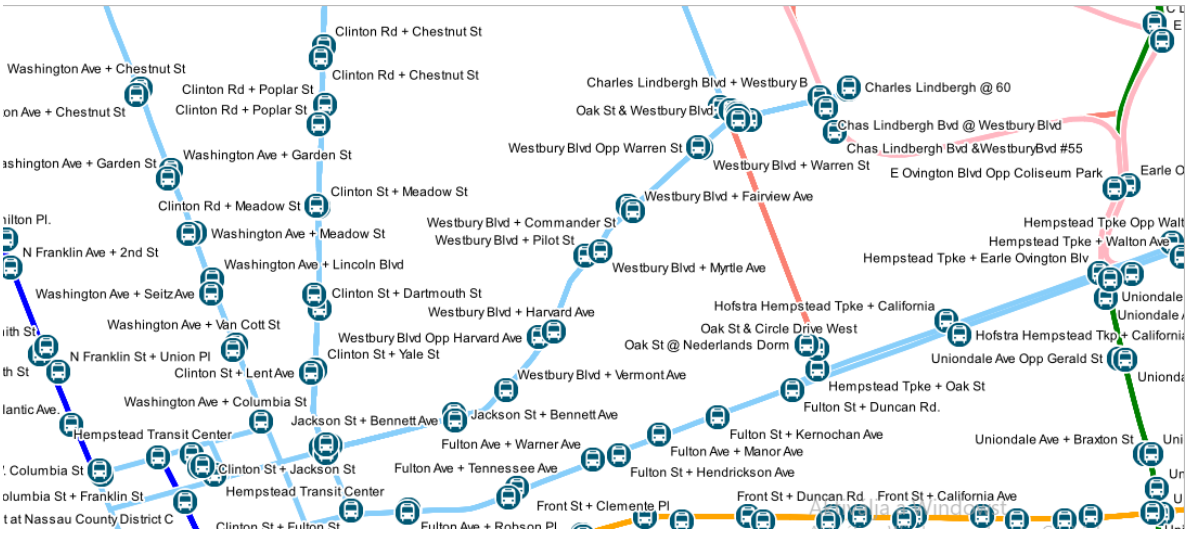
Working with GTFS data in QGIS using GTFS GO

GTFS Go is a QGIS plugin to display paths and stops. First, install this plugin:

1. In QGIS *Plugins* → *Manage and Install Plugins* → *GTFS-GO*→*Install Plugin*
2. Then, open the plugin and load data from a preset repository. For example, use [USA][New York][Brooklyn].
3. Select the GTFS data source, then set an output directory on your computer.



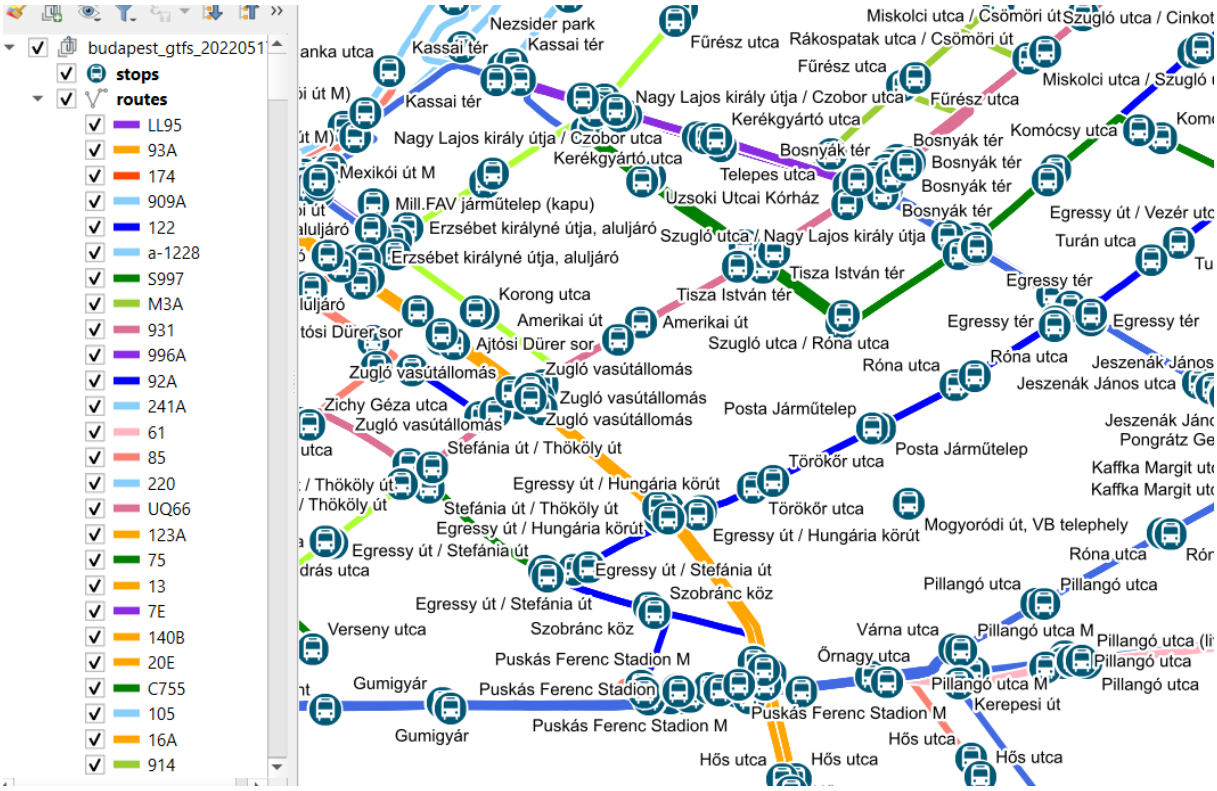
Now, you have the Brooklyn layers, with a stops layer and a routes layer. The stops layer contains the *stop_id* and the name, and routes contains the data from *shapes path layer*, and the *route_short name* field.



What is the projection of the data? It uses geographic coordinates; therefore, the custom CRS of this data is WGS84, EPSG: 4326.

Now, let's load the budapest_gtfs.zip from CANVAS or from this link: https://bkk.hu/gtfs/budapest_gtfs.zip

Select the Local ZIP file option, and find the GTFS. You will get this:



Opening of BKK data in QGIS

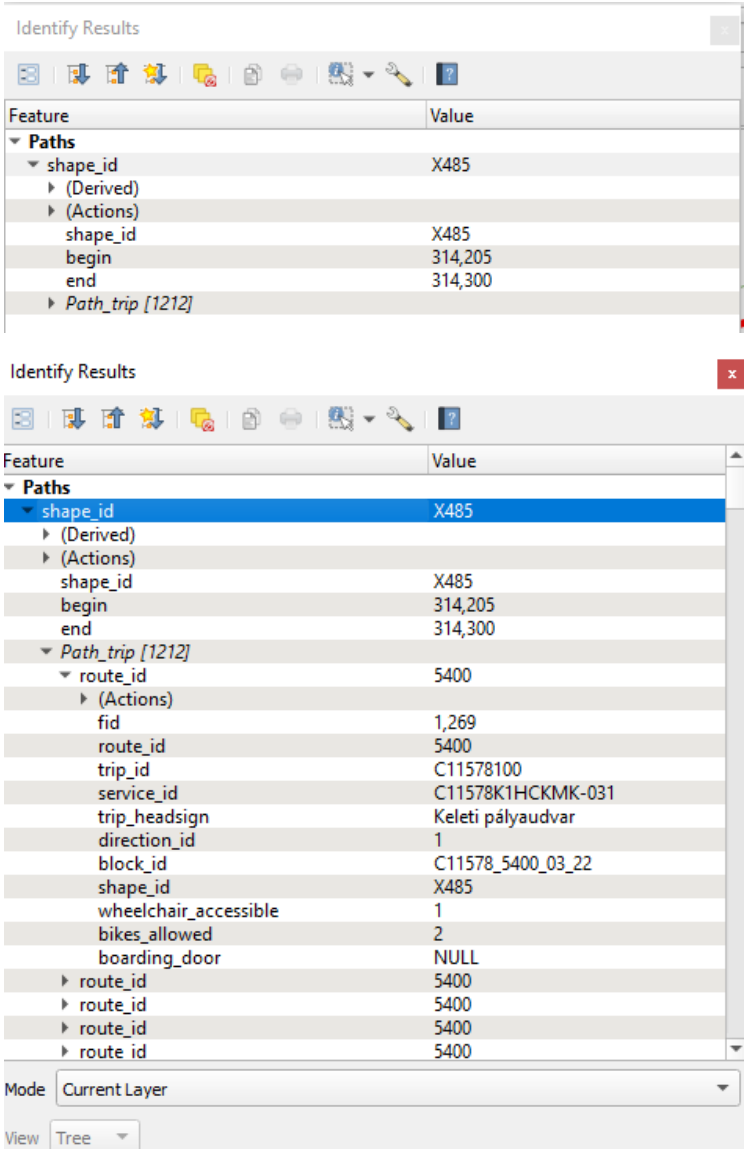
Let's remove all data from QGIS or open a new project. First, import the *stops.txt* and *shapes.txt* files in Data Source Manager as Delimited Text files. The delimiter character is ',' (comma), and the geometry definition is Point coordinates. Set the X field to *shape_pt_lon* and the Y field to *shape_pt_lat*. The projection (CRS) is 4326. Once you are ready, open the Processing menu and find the **Points to Path** algorithm. This algorithm creates polylines from the points of a path. Add the input layer, which is the shapes layer. The Order field specifies the order of points, indicated by the shape_pt_sequence field, and the Group field shows the identification number of lines (this value is the *shape_id*). After running the algorithm, you will get this:

How can I find out, which bus it is and where it goes?

Let's make joins between layers. First, please import the routes.txt, trips.txt, and stop_times.txt files as Delimited Text layers. All settings are the same, except the Geometry Definition is "No geometry." Connect the shapes and the trips tables. The shapes table has the primary key: *shape_id*, and join it to the *trips.shape_id* foreign key.

Shapes layer→*Project* → *Properties* → *Joins*→ Join the table trips with shape_id to target field shape_id. Apply the Joins.

Now, you can join the routes table with routes.route_id to the target column route_id (from the trips table). You can now use the Identify Feature button. If you click on a route, you can read the information in the small window (*route_short_name*, *trip_headsign*).

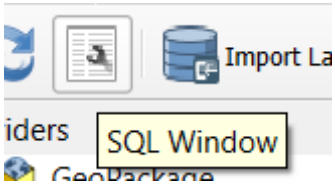


QGIS Database (DB) Manager

[I you can not see it in the menus, activate it in the Plugins→ Manage and install plugins]

Open the DB Manager: *Database* → *DB Manager*.

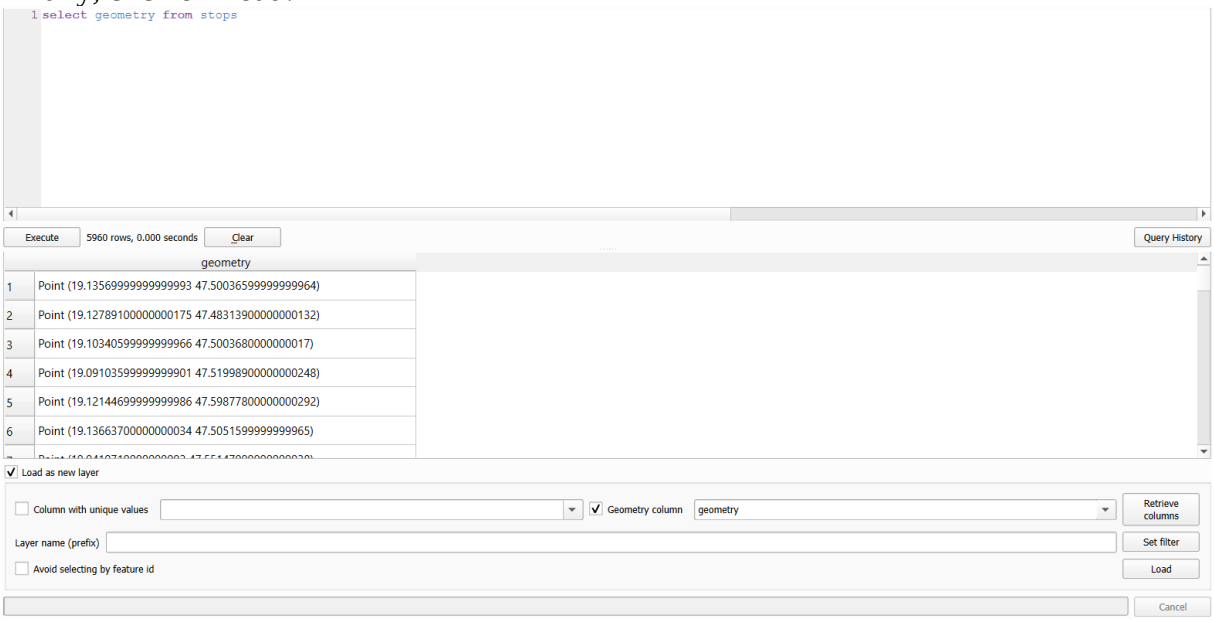
Here on the left, you can see the project layers in the virtual layers tree. Open a new SQL Query.



Let's answer the following questions with queries!

1) Print the stops in 'Örs vezér tere'.
2) How many different name does 'Örs vezér tere' stop have?
3) Display all stops from 'Örs vezér tere' in the QGIS main window!
4) What is the number of the route called 'Csepel-Királyerdő' ? (*trips* and *routes, trip_headsign)*

If you want to show the geometry in the QGIS window, first, put the geometry field after the SELECT statement, then go to Load as New Layer, and set the geometry column (geometry). Finally, click on Load!



Key:

```
1.) select * from stops where stop_name like "Örs vezér tere%"
2.) select stop_name, count(*) from stops where stop_name like "Örs
vezér tere%" group by 1
3.) select stop_name, count(*), geometry from stops where stop_name
like "Örs vezér tere%" group by 1
4.) select * from trips  join routes on
trips.route_id=routes.route_id where trip_headsign='Csepel-
Királyerdő'
```

A database index is a data structure that improves the speed of data retrieval operations on a database table at the cost of additional writes and storage space to maintain the index data structure. Indexes are used to quickly locate data without having to search every row in a

database table every time the table is accessed. Indexes can be created using one or more columns of a database table, providing the basis for both rapid random lookups and efficient access to ordered records. If you do not use indexes in a database, queries run slowly. QGIS does not use indexes on tables; therefore, the next queries will be run in PostgreSQL+PostGIS with DBeaver.

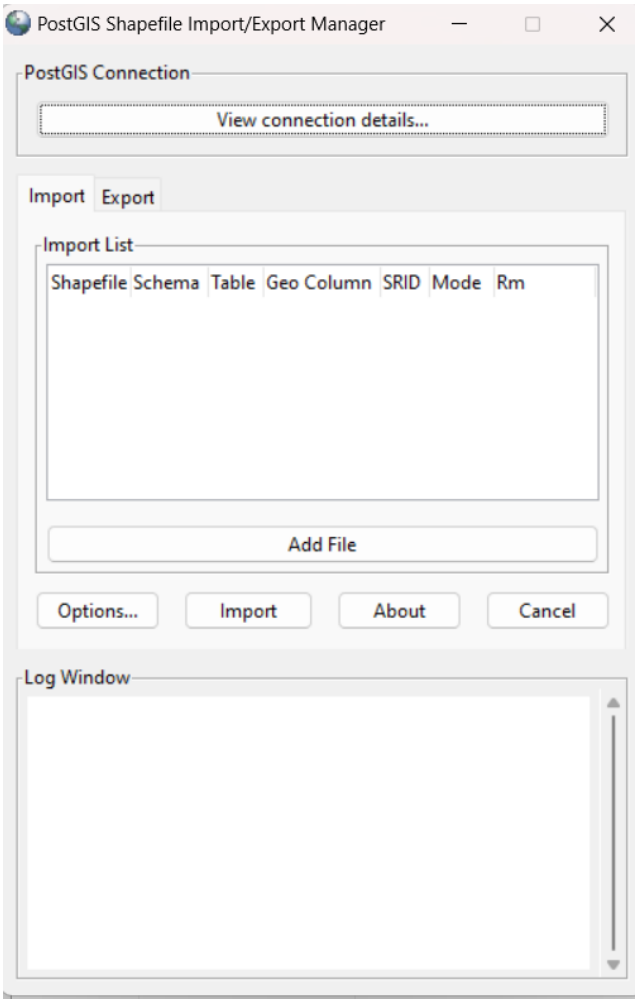
The easiest option to import Shapefiles into PostgreSQL is to use the PostGIS Bundle program. In a Windows environment, it can be easily started from the Start menu. First, please save your data as Shapefiles. If the file does not have geometry, the DBF file can be written. Right-click on the layer name → *Export* → *Save Features As* →*Esri Shapefile*. You will get an error message about missing geometry. Ignore it. Check whether you have the DBF file!

In DBeaver

Create a new empty database. *Right- click on databases*→ *Create an empty database*.

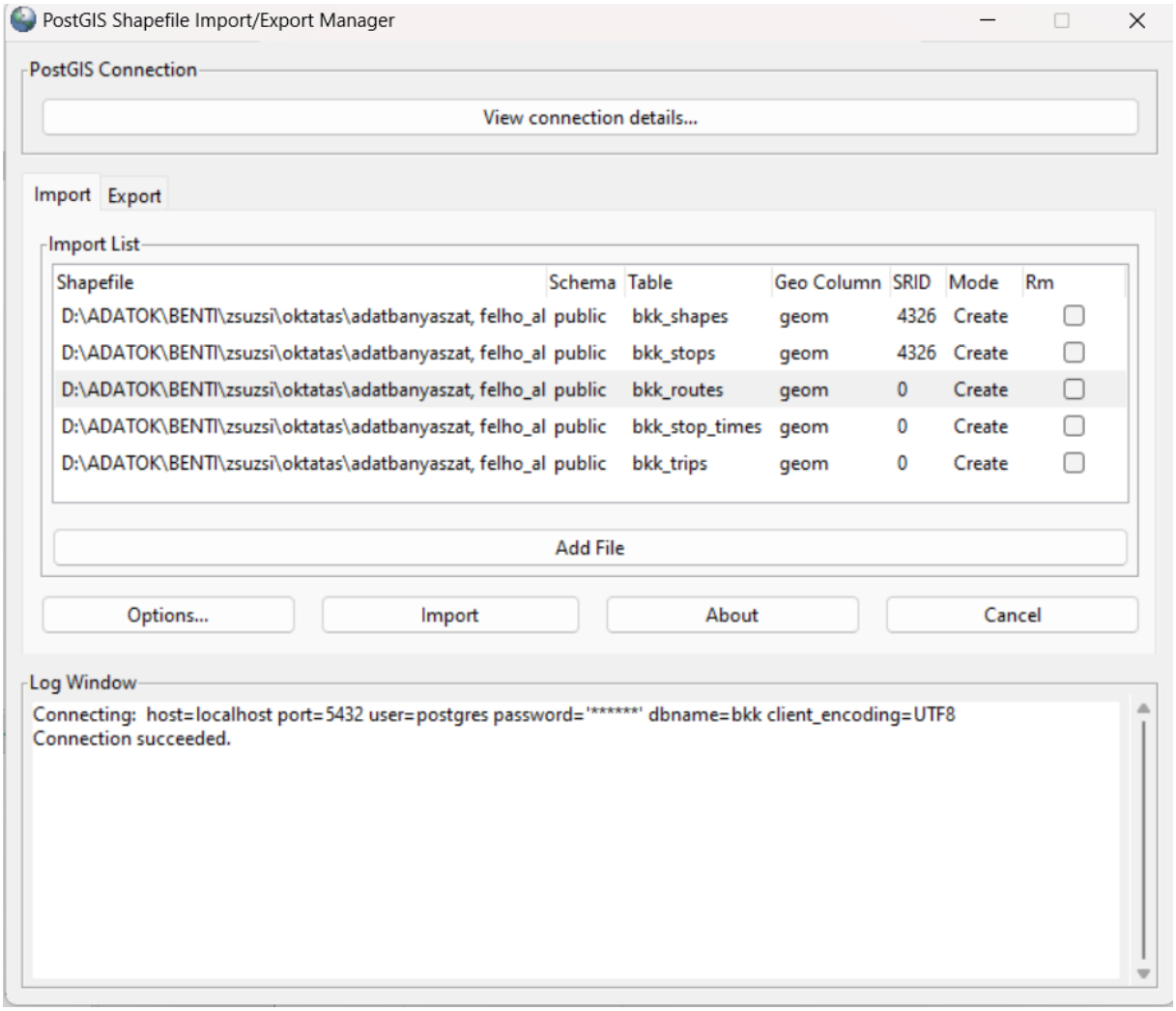
Add the POSTGIS extension to the table.

Open PostGIS Bundle. Set the connection details.



Add the files. Below the white box, you can choose DBF as the file type. Now, you are able to import the data.

Set the SRID to 4326 on *bkk_shapes* and *bkk_stops* tables (They have geometry!). Finally, import them.



Now, you answer the following questions with queries:

5.) Which routes go to ’Göncz Árpád városközpont’? Order in ascending order!
6.) When do buses arrive at ’Egyenes utcai lakótelep’?
7.) Which buses stop at ’Egyenes utcai lakótelep’?
8.) Make a list of the departure times of bus routes ’276E’ at’Egyenes utcai lakótelep’!
9.) Query the route of bus line ’276E’!

Key:

```
5.) select route_short_name from trips  join routes on
trips.route_id=routes.route_id where trip_headsign like 'Göncz
Árpád városközpont%' group by 1 order by 1
6.) select arrival_time from stops join stop_times on
stops.stop_id=stop_times.stop_id where stop_name=’Egyenes utcai
lakótelep’ order by 1
7.) select distinct routes.route_short_name from stop_times join
stops on stops.stop_id=stop_times.stop_id join trips on
trips.trip_id=stop_times.trip_id join routes on
routes.route_id=trips.route_id where stop_name like 'Egyenes utcai
lakótelep%'
```

```
8.) select distinct arrival_time from stop_times join stops on
stops.stop_id=stop_times.stop_id join trips on
trips.trip_id=stop_times.trip_id join routes on
routes.route_id=trips.route_id where stop_name like 'Egyenes utcai
lakótelep%' and route_short_name='276E'
9.) select geometry from shapes_path join trips on
trips.shape_id=shapes_path.shape_id join routes on
routes.route_id=trips.route_id where route_short_name='276E'
```

Chapter 3. Geocoding services

What is geocoding?

Geocoding is the process of assigning coordinates to a given address or geographic name.

What is reverse geocoding?

Reverse geocoding is the process of finding an address or geographic name close to given coordinates.

The result of geocoding and reverse geocoding depends on the accuracy (level of detail) of the source geodatabase. It is worth mentioning that even if the source database is very accurate and detailed, if the given address is not exact or incomplete (e.g., missing district name), the result may not be as good as expected. Let's try to find the following address in OpenStreetMap or Google Maps: Budapest Jókai utca 6.

Budapest Jókai utca 6.

Results:

As you can see, there are several Jókai streets in Budapest. JÓKAI Mór was a popular writer in the 19th century, and therefore his name appears in several street names in Hungary: OpenStreetMap Search:

https://nominatim.openstreetmap.org/ui/search.html?q=Budapest+J%C3%B3kai+utca+6.&exclude_place_ids=71360719%2C173383115%2C24319190%2C259704147%2C226370159

Add the district to the address: Jókai utca 6., Budapest, XVI. kerület OpenStreetMap Search with District:

Jókai utca 6., Budapest, XVI. kerület

https://nominatim.openstreetmap.org/ui/search.html?q=+J%C3%B3kai+utca+6.%2C+Budapest%2C+XVI.+ker%C3%BClet

Now, I found what I wanted.

The address should contain the country, (county or district), city, postal code, street, and house number.

The website of OSM Nominatim

Nominatim works with the OpenStreetMap address database. You can easily try it out on this website. You can send queries here both in simple and structured forms. Simple means the concatenated address. Structured form means that you separate each part of the address into different boxes, such as address, postal code, city, etc. https://nominatim.openstreetmap.org/ui/search.html

How can you geocode in QGIS?

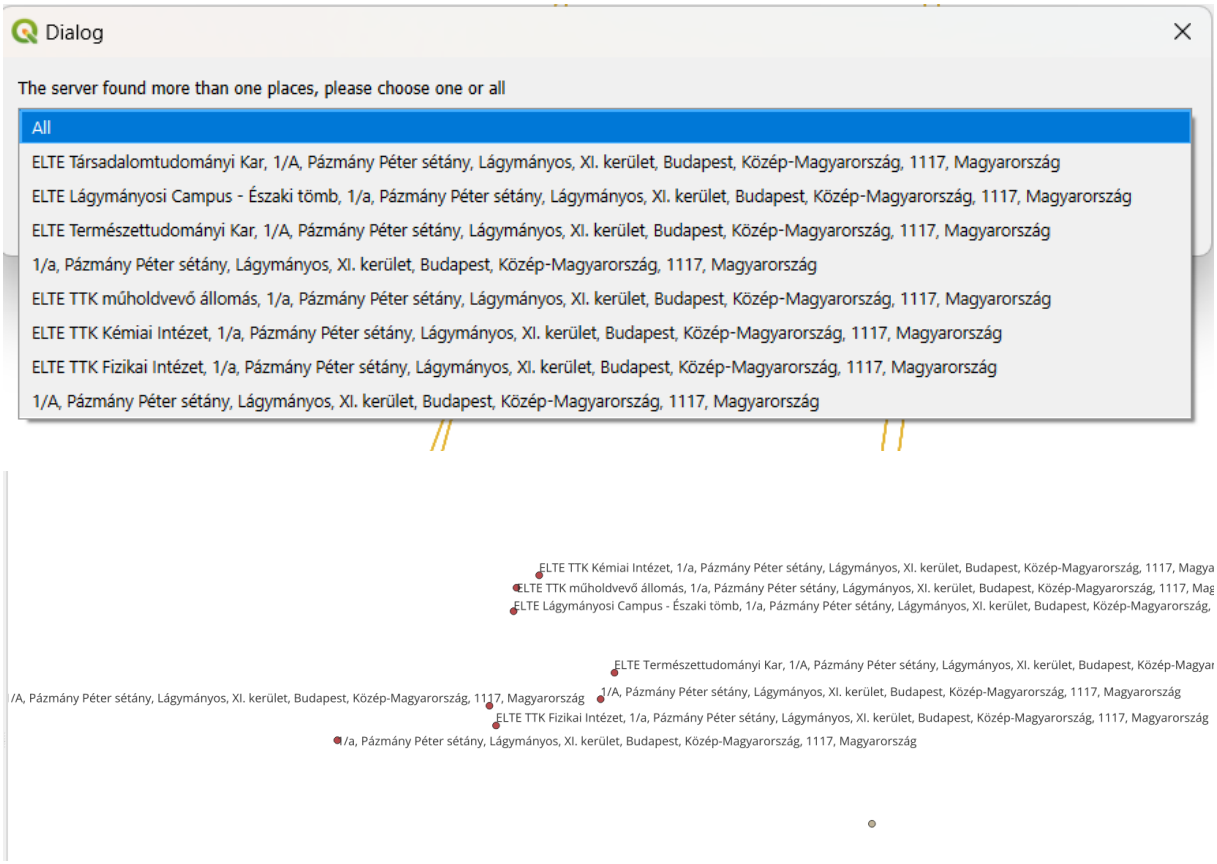
Install 'Geocoding' plugin from the Plugin Repository. *(Plugins →Manage and Install Plugins*).

Open the **Geocoding** plugin.

Enter the address, run the geocoding request, and select the desired result from the list. For example: 1117 Budapest Pázmány Péter sétány 1/a.

There are several addresses; please select one from the set of addresses!

This plugin uses the OpenStreetMap Nominatim geodatabase.



MMQGIS

Install 'MMQGIS' plugin from Plugin Repository. (*Plugins →Manage and Install Plugins*)

I have a dataset with all kindergartens, primary and secondary schools in Hungary. I downloaded this file from the website of the Oktatási Hivatal (Office of Education). https://dari.oktatas.hu/

This quite a big file, so I reduced the size to 100 rows!

MMQGIS is a plugin where you can run geocoding requests. In this plugin, you can read excel tables, and after the geocoding, you will get a new QGIS layer with the points of schools.

But, before you start the geocoding, first redesign the basic Excel table!

1. Reduce the file size, keep only 100 rows.
2. Add a new country field, and fill it with the text 'Magyarország' (Hungary). This helps to increase the success of geocoding.
3. Save the file in CSV format. The delimiter character is comma.
   *(*Set in Excel or *Notepad++* → Search→*Find* → *Replace)*
4. Set the character encoding to UTF-8 in Excel or use Notepad++.

Open *MMQGIS* in the Menu bar → *Geocode* → *Geocode CSV with web service*

Give the following options:

- Address: utca és házszám (street and housenumber)
- City: város (city)
- State: megye (county)
- Country: ország (new country field)

Set the geocoding service to OSM Nominatim. This is a free option. There are other possibilities, like Google Map sor ESRI, but you need activation key (API key) to these geocoder first before you can use them.

The *Output file name* and *not found output list* → it is a list with the addresses, where the geocoding was not succesful

The result is a new layer with the point coordinates of the schools.

Python geoPy

The next part of this task involves using the Python module `geoPy`. Please install it before you begin the work. `geoPy` may have dependencies, so please install them as well.

`geoPy` is a module for geocoding and reverse geocoding in Python. The main advantage of this module is that it allows to use more than 30 geocoding services, including the most important ones like Nominatim, Google Maps, Esri, TomTom, etc. The geocoding request itself is a very short and simple in every case.

Now, you will work with the schools Excel table.

You do not need any preprocessing (please work only with 100 records).

Please install `pandas, openpyxl,` and `GDAL/OGR` module, if you have not done so earlier.

Python `pandas` works with tables, and `openpyxl` is necessary to open XLS or XLSX files with `pandas`.

GeoPy Documentation

https://geopy.readthedocs.io/en/stable/

If you have questions about installation, please read the „Installing modules on Python" Chapter.

Abstract of the task: Read schools.XLSX file. Geocode the full addresses with OSM Nominatim. Create a Shapefile with the following attributes: name of the school, full address. Open this Shapefile in QGIS.

The detailed description of the task:

1. Open the Excel table. Read the file.
2. Concatenate the address cells of the Excel table. Send itt to geocoder
3. Write the answer in a Shapefile. Locations have to be displayed as points.

Import the modules in the code header. GDAL/OGR package is called osgeo. GDAL works with raster data, OGR works with vector data. Use Nominatim for geocoding.

```
from geopy.geocoders import Nominatim
from osgeo import ogr
from osgeo import osr
import pandas as pd
geolocator = Nominatim(user_agent="Myapp")
```

Read *schools.xlsx* table. Import the data into the DataFrame. DataFrame is a special two-dimensional matrix to store table format data. DataFrame allows us easy access to data.

```
df=pd.read_excel('schools.xlsx')
```

Rename all columns according to their header (the first line). This means the index of the columns inherited from the header name.

```
df.rename(columns=df.iloc[0]).drop(df.index[0])
```

Before you begin the geocoding, first create the Shapefile. Define the driver.

```
driver = ogr.GetDriverByName("ESRI Shapefile")
```

Create the Shapefile datasource.

```
ds = driver.CreateDataSource("schools.shp")
```

Give the spatial reference system (WGS84, EPSG: 4326)

```
srs =  osr.SpatialReference()
srs.ImportFromEPSG(4326)
```

Create a layer, define the data type.

```
layer = ds.CreateLayer("points", srs, ogr.wkbPoint)
```

Add the attribute table definition. Create an ID field with integer data type. Create a column with the full address.

```
idField = ogr.FieldDefn("id", ogr.OFTInteger)
layer.CreateField(idField)
featureDefn = layer.GetLayerDefn()
```

Create a *do_geocode* function that performs geocoding. If the geocoder is unavailable, it makes several attempts, before giving up.

```
def do_geocode(address, attempt=1, max_attempts=5):
    try:
```

```
        return geolocator.geocode(address)
    except GeocoderTimedOut:
        if attempt <= max_attempts:
            return do_geocode(address, attempt=attempt+1)
        raise
```

Let's write the for loop, geocode the addresses. First concatenate the full address.

```
for i in range(1,99):
     addr=' Magyarország, '+df.loc[i][COUNTY']+' vármegye
'+df.loc[i][ 'CITY']+', '+str(df.loc[i]['POSTAL_CODE'])+'
'+df.loc[i]['ADDRESS']+' '
```

Call the geocoder.

```
    location = do_geocode(addr)
```

Check the returned address. If you have a result (if the location variable is not empty), write the Shapefile. The location variable contains the latitude, longitude and the full address.

```
if location!=None:
        feature = ogr.Feature(featureDefn) #create the feture
        point = ogr.Geometry(ogr.wkbPoint) #give the geometry type
        pont.AddPoint(location.longitude, location.latitude) #add
the coordinates
        feature.SetGeometry(pont) #add geometry to feature
        feature.SetField("id", i) #set field data
        layer.CreateFeature(feature) #add feauter to layer
ds = None #close the Shapefile
```

Chapter 4: Working with statistical data with pandas and matplotlib modules

This chapter demonstrates how to work with statistical data and how to create diagrams using matplotlib. The source dataset was downloaded from this website:

https://www.ketszintu.hu/publicstat.php

This website contains the results of final exams in Hungarian secondary schools for various subjects including mathematics, biology, Hungarian literature and grammar, chemistry, etc. The exams areconducted at two levels: standard and advanced. Today, we will work with the results of a mathematics exam from 2018. The data are available in CSV format.
https://www.ketszintu.hu/publicstat.php?stat=_2018_1&reszletes=1&eta_id=3&etj_szint=K

Steps:

- First, familiarize yourself with the dataset.
- Second, calculate the average score for the entire country!
    - Sum the points from the 'I. rész' and 'II. rész' sections to get the total score.
    - Use only the data where the student was present at the exam (`vizsgázó megjelent = megjelent`) and where the written score is valid (`írásbeli pontszám != '-'`).
- Print the average scoress for each county!
- Plot a bar chart based on the average scores!
- Save these scoress to a textfile, and create a coropleth map in QGIS using the *hungary.gpkg* dataset.

Let's see the detailed code!

Import the `pandas` and *`matplotlib`* (as plt) modules in the header of your code. Read the CSV file into the DataFrame. Specify the source, the delimiter, the character-encoding and the header line.

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

df = pd.read_csv('matek_2017_2.csv',
delimiter=";",encoding="cp1250", header=0)  #load the data in the
Dataframe
```

Drop the original column index (which is initially an integer index), and set the header as the new index. The Hungarian Math final exam consists of two parts: *'I.rész'* and *'II. rész'*. These columns contain students' scores, but the values are stored as strings. Aggregation functions such as min, max, avg can not process string values. Therefore, you need to convert the strings to integers. First, create a new empty column, as a pandas Series.

```
df.rename(columns=df.iloc[0]).drop(df.index[0])
df['1.rész']=pd.Series(dtype='int')
df['2.rész']=pd.Series(dtype='int')
df['ossz']=pd.Series(dtype='int')
```

Then, fill the Series with numbers. Create a for loop that iterates through all records (length of the DataFrame minus 1). Inside the loop, use an If condition whether the student attended the exam and received a valid score. If so, update the Series with the corresponding values.

After that, you can call the aggregation funtions. Let's try **min**, **max**, and **avg** on the total score (sum of *'I. rész'* and *'II. rész'*), now applied to the entire dataset.

```
v=0;
for i in range(1,len(df)-1):
    if df['vizsgázó részvétele'][i]=='megjelent' and df['írásbeli
pontszám'][i]!='-' :
        v=v+1
        df['1.rész'][i]=int(df['I. rész'][i])
        df['2.rész'][i]=int(df['II. rész'][i])
        df['ossz'][i]=df['1.rész'][i]+df['2.rész'][i]
print(df['ossz'].sum()/v)
print(df['ossz'].aggregate('mean'))
print(df['ossz'].aggregate('max'))
print(df['ossz'].aggregate('min'))
print(df['ossz'].max())
print(df['ossz'].min())
```

If you want to group the data in a table, use the groupby function. Group the data by county (*'intézmény megyéje'*).

```
print (df.groupby('intézmény megyéje')['ossz'].aggregate('mean'))
print (df['1.rész'].sum()/v)
print (df['2.rész'].sum()/v)
```

Now, create a diagram using matplotlib. Store the groupby result in a variable called *df_diagram*. The plot function requires three parameters:

- x-axis → county (*'intézmény megyéje'*)
- y-axis → total score
- diagram type → bar chart.

Use Plt.show() to display the diagram.

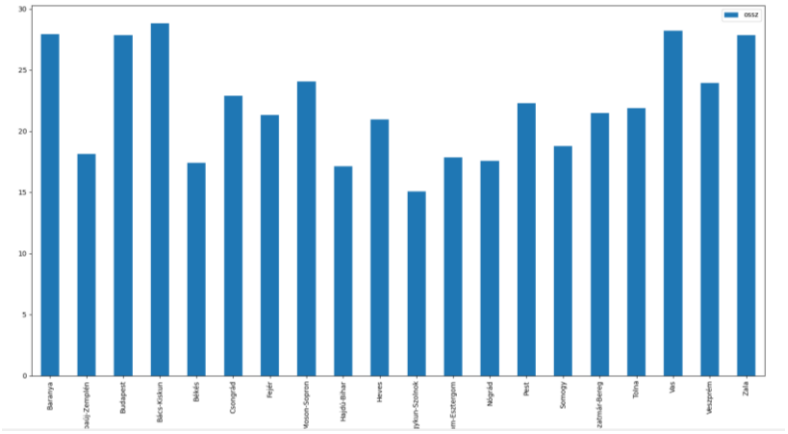
```
df_diagram=df.groupby('intézmény
megyéje')['ossz'].aggregate('mean')
df_diagram.plot(x='intézmény megyéje', y='ossz', kind='bar')
plt.show()
```

Finally, write the total scores by county to a file using the function *to_csv(filename)*.

```
df_diagram.to_csv('ujmatek.csv')
```

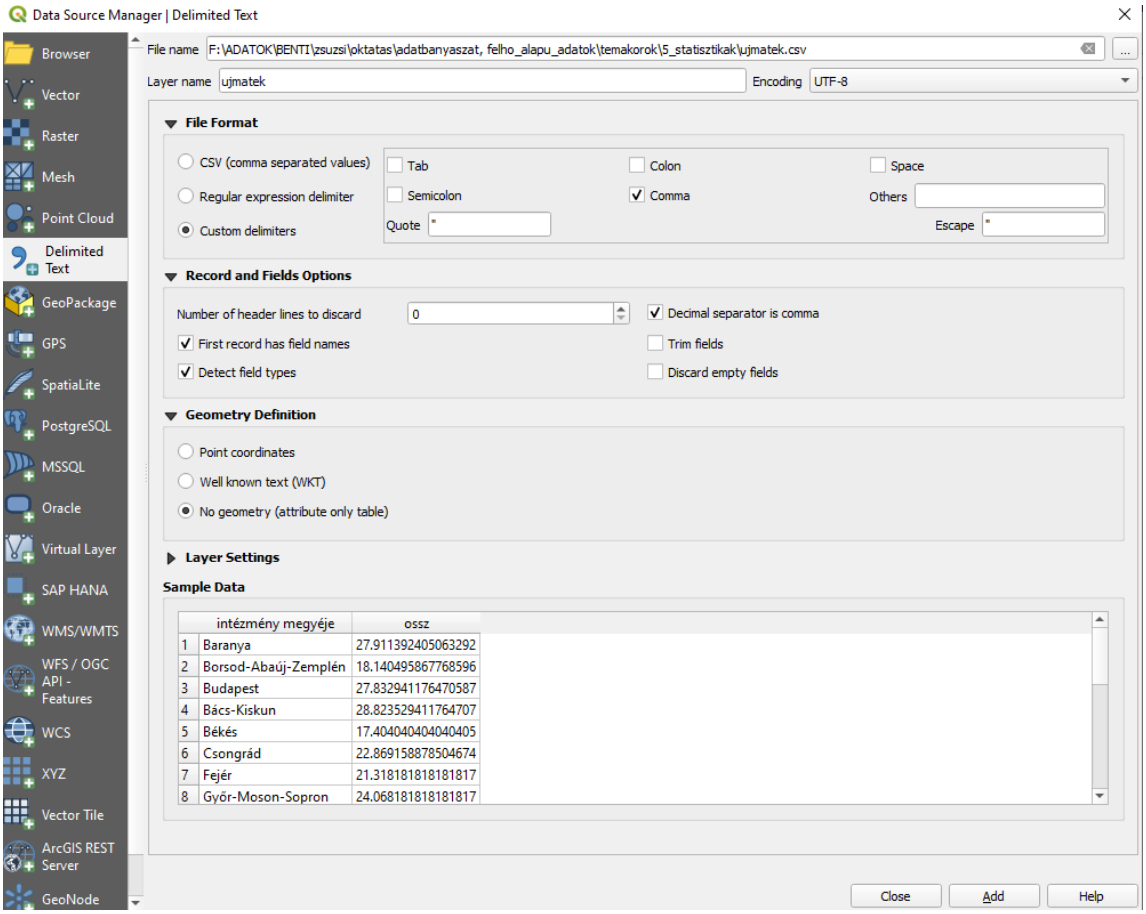
If you want to save the diagram as a file, use *savefig(filename)* function. The file format can be JPG, PNG, PDF, etc.

```
fig = df_diagram.plot(x='intézmény megyéje', y='ossz',
kind='bar').get_figure()
fig.savefig('figure.pdf')
```

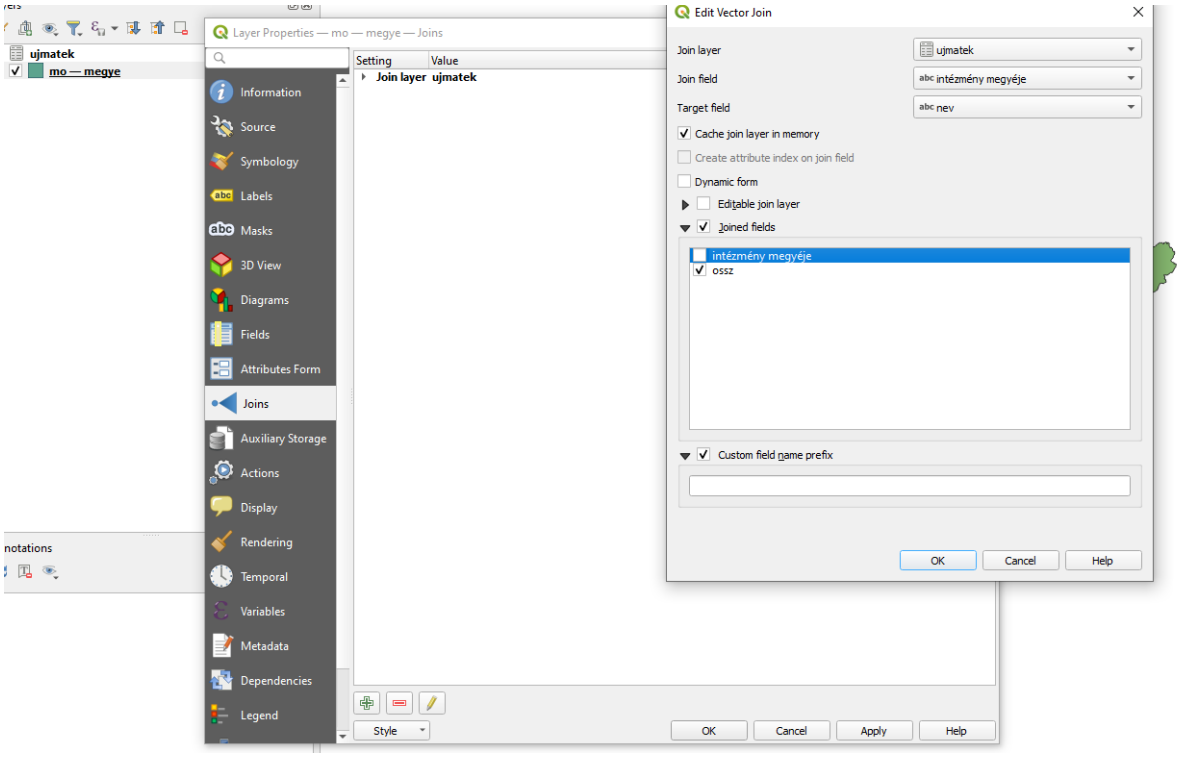


If you have prepared the CSV file, create a choropleth map in QGIS. You need *hungary.gpkg* as a basemap. Open the county layer (megye).

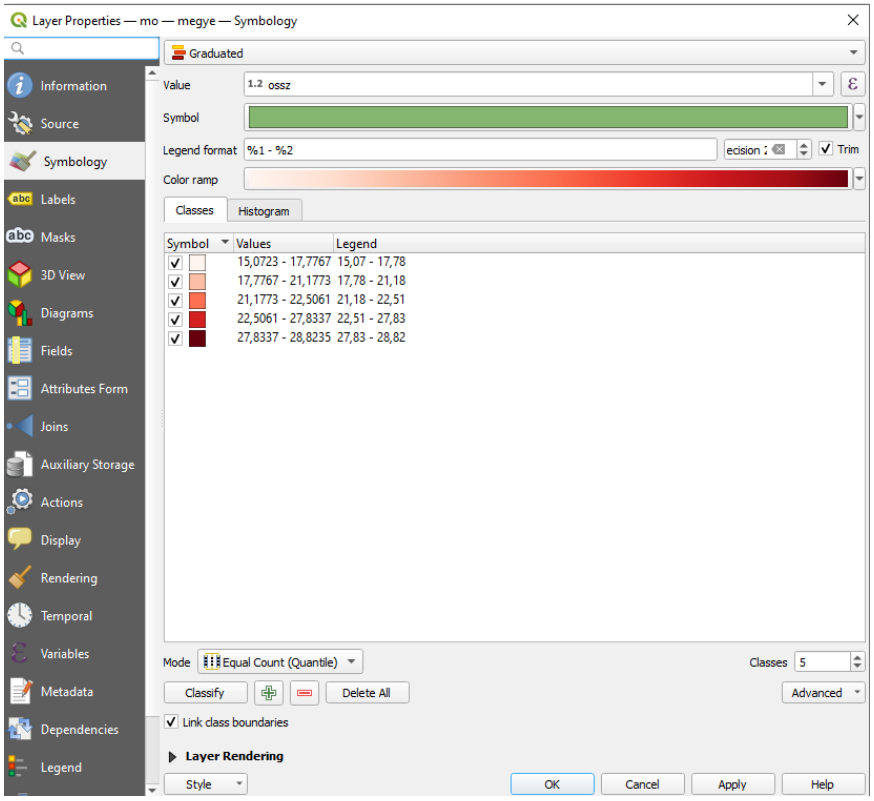
Open the data Source manager and navigate to the Delimited textfile submenu. Import the delimited textfile as an attribute table.

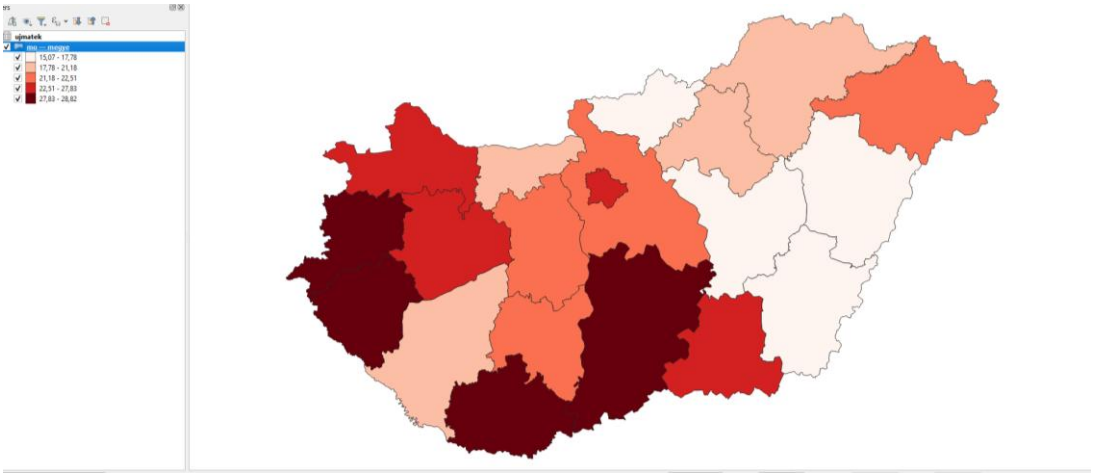


Join this table to the county layer, using the county name as the target field.



Go to the layer symbology, and apply graduated styling. Test different classification methods, create 5 groups. Explain, which classification method works best for this dataset and why?





Homework: If there are values in the column (’szóbeli pontszám’), add them to the total score! What is the new total score? Create a diagram showingthe new average total score by county.

Chapter 5: Working with statistical data with pandas and matplotlib modules: Water level changes

The **OVF Hungarian Hydrological Forecasting Service** continously monitors the water level of lakes and rivers in Hungary. Their website, Hydroinfo (https://www.hydroinfo.hu/), allows everyone to track these water level changes. They also store archived water level data here: https://www.hydroinfo.hu/vituki/archivum/index.html (Központi Hidrológiai Adattár – Cenral Hydrological Archive)

We are currently using the Lake Balaton water level data from 2000 to 2004. During this period, the Carpathian Basin experienced a dry era, which caused the water level of Lake Balaton to drop to extremely low levels. These changes are well illustrated in a diagram. Therefore, we will create a diagram showing the monthly average water levels for the mentioned years.

First, let's check the data format on the website. Please select the *'Balaton átlag'* (=Balaton average) and enter 2000 in the textbox, followed by (2001, 2002, 2003 and 2004).

Észlelt V Í Z Á L L Á S O K Évszám: 2000

adatok jégkóddal /interpolációval/ [cm] Időpont: 7:00 ń 3:00 KEI

Kinyomtatva: 2004-Jún-24 11:42 Vízgyüjtö terület: 5774.0 km2

Állomás kód: 142300 Távolság a torkolattól:

Állomás név: BALATON ATLAG A nullpont magassága:

Vízfolyás: BALATON Érvényes: 2000-Dec-31

Nap	Jan	Feb	Már	Ápr	Máj	Jún	Júl	Aug	Sze	Okt	Nov	Dec
1	110	102	102	111	104	98	85	78	64	59	60	62
2	111	102	102	111	105	98	85	78	65	60	61	62
3	111	101	103	111	104	97	85	78	65	60	60	62
4	111	101	103	111	104	97	84	78	65	60	60	62
5	111	101	104	111	104	97	83	77	64	60	61	62
6	111	101	104	111	104	97	83	76	64	60	61	62
7	110	101	104	111	103	96	83	76	63	59	61	62
8	110	100	104	111	103	96	82	75	63	59	62	62
9	110	100	104	110	103	96	82	74	62	59	62	62
10	110	101	105	109	103	95	82	74	62	59	62	62

This table has a header and footer, which we will disregard. The table contains the days and the months (Jan, Feb, Már…). I saved this table and uploaded it to Canvas. Please download the file Evizallas_balaton.zip and unzip it!

These files contain the water levels, but some preprocessing is required before generating the diagrams. Open the file in Notepad++. The header row should contain the months.

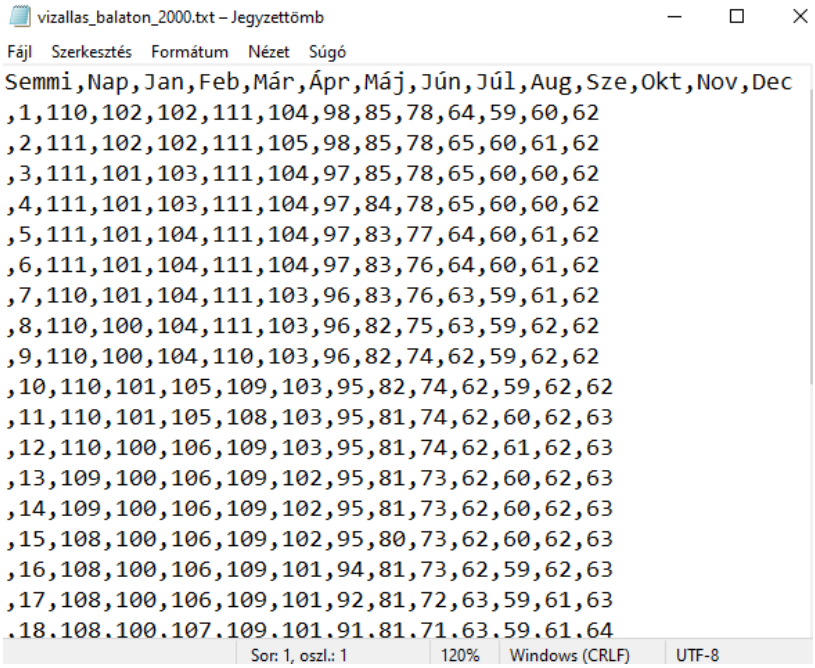
- Remove the empty lines.

- Replace the tabs with commas

- Replace the spaces/double spaces with commas. (Be careful in lines 29,30 and 31!)

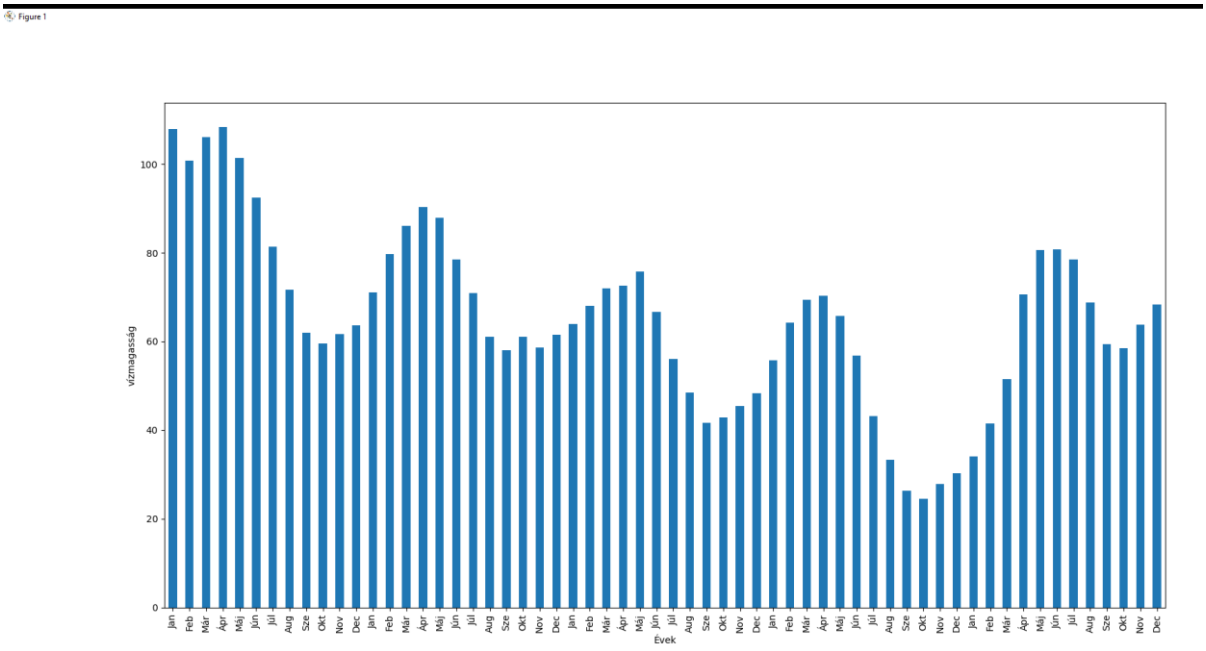
- Add a comma before the day. This will cause a mismatch in the number of columns between the header and the data rows. To fix this, insert a new column at the beginning of the file header (use *"Nothing"* as the header name).

The cleaned file should look like this:



Ensure that the number of columns in the header matches the number of columns in the data rows.

To **create a bar chart** that displays the monthly average water levels for the years 2000–2004, follow these steps:



Import *vizallas_balaton_2000.txt*

- Import the required libraries: `pandas` and `matplotlib`.
- Use *read_csv* function with the following parameters:
  - *utf8* character encoding,
  - header: the first line,
  - delimiter: comma (,).
- Remove the empty columns using *del* command.

```
import pandas
import pandas as pd

#2000-2004
df0=pd.read_csv('vizallas_balaton_2000.txt',
delimiter=",",encoding="utf8", header=0)

df1=pd.read_csv('vizallas_balaton_2001.txt',
delimiter=",",encoding="utf8", header=0)
df2=pd.read_csv('vizallas_balaton_2002.txt',
delimiter=",",encoding="utf8", header=0)
df3=pd.read_csv('vizallas_balaton_2003.txt',
delimiter=",",encoding="utf8", header=0)
df4=pd.read_csv('vizallas_balaton_2004.txt',
delimiter=",",encoding="utf8", header=0)
del df0['Semmi']
del df0['Nap']
del df1['Semmi']
del df1['Nap']
del df2['Semmi']
del df2['Nap']
del df3['Semmi']
del df3['Nap']
del df4['Semmi']
del df4['Nap']
```

Aggregate the data by calculating the mean for each month.

```
df_diagram0=df0.aggregate('mean')
df_diagram1=df1.aggregate('mean')
df_diagram2=df2.aggregate('mean')
df_diagram3=df3.aggregate('mean')
df_diagram4=df4.aggregate('mean')
```

Print the aggregated data.

```
print (df_diagram0)
>>
Jan    107.967742
Feb    100.724138
Már    106.129032
Ápr    108.400000
Máj    101.419355
Jún     92.400000
Júl     81.322581
Aug     71.741935
Sze     62.033333
Okt     59.548387
Nov     61.600000
Dec     63.677419
dtype: float64
```

Combine data from all files:

- For the years 2000 to 2004, concatenate the DataFrames using the *pd.concat()* function.
- First, add the aggregated data from each file to a list.

Plot the bar diagram using the combined data.

```
sumdf1=[df_diagram0,df_diagram1,df_diagram2,df_diagram3,df_diagram4
]
result=pd.concat(sumdf1)
result.plot(xlabel='Évek', ylabel='vízmagasság',kind='bar')
plt.show()
```

Chapter 6. Working with photos metadata (Flickr photos metadata)

Please check this article, focus on images. (page 1, and 17–19.)

Gede Mátyás: Where do tourists go – Visualizing and analysing the Spatial Distribution of Geotagged Photography.

https://www.researchgate.net/publication/262972685_Where_Do_Tourists_Go_Visualizing_and_Analysing_the_Spatial_Distribution_of_Geotagged_Photography**What will we do?**

What will we do?

Flickr is a very popular image sharing website, where users can uploads photographs, and can mark, where the photo was taken. These metadata can be downloaded from Flickr website. We create a grid on an sample area, and count, how many photographs(=points are in the cell). According to the number of points, the cell height will be set in a Google KML format.

Downloading the photo metadata

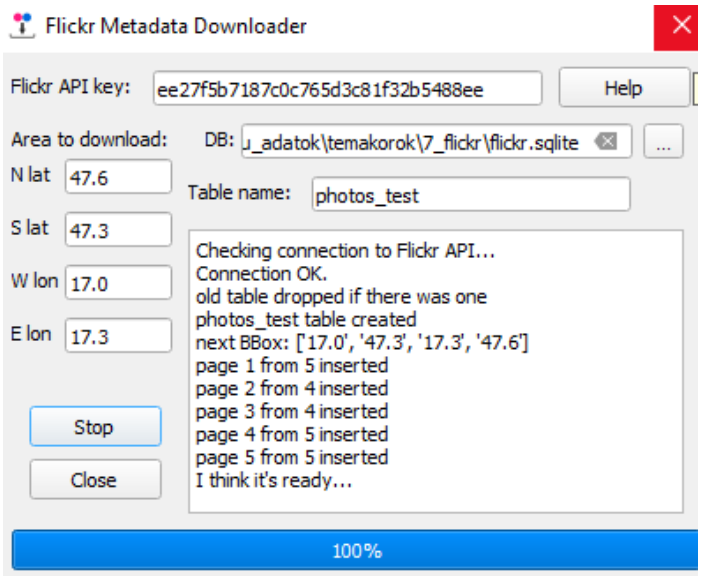
In QGIS go to *Plugins* → *Manage and Install Plugins* → *Install Flickr Metadata Downloader* (FMD).

Open FMD.

FMD works with SQLite layers. Therefore, first, create an empty SQLite file by navigating to *Layer* → *Create Layer* → *New SpatiaLite Layer*. Provide a file name and a default layer name. (Note: We will not use this layer, as FMD will create a new layer. However, this SQLite file is necessary since FMD cannot create the SQLite file itself.)

Start FMD.

Provide your Flickr API key (if you have one) or use the default key. Select the SQLite database file and provide the layer name. Set the coordinates for the bounding box. Finally, start the downloading process.



After downloading, open the photo_test layer. You will see several points indicating where each photo was taken. Make sure to check the attribute table as well!

Generate a grid, count the points in the grid cells

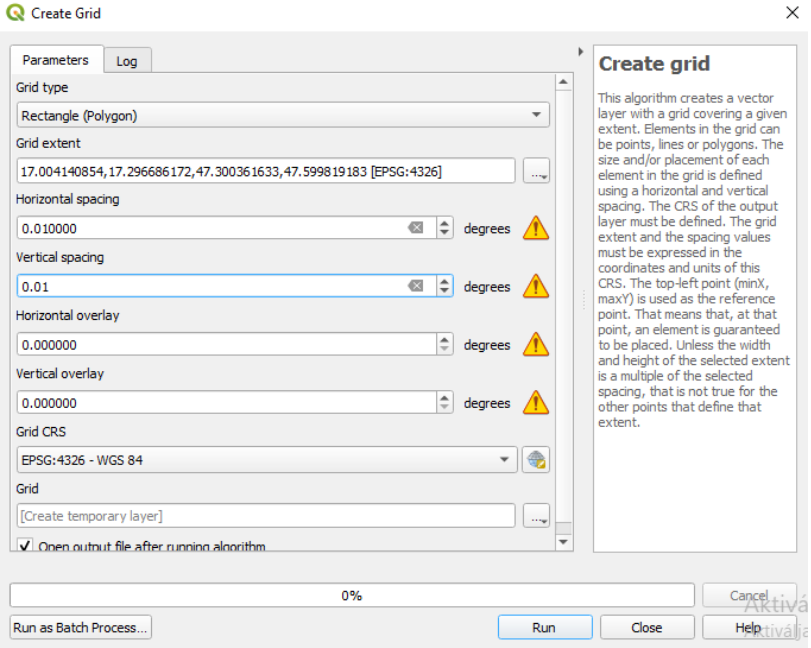
Use the Create Grid tool in the Processing Toolbox to define a grid. The current projection is WGS84 EPSG:4326, therefore the optimal grid size in this area is 0.01°

That means 1x1 km cell size:

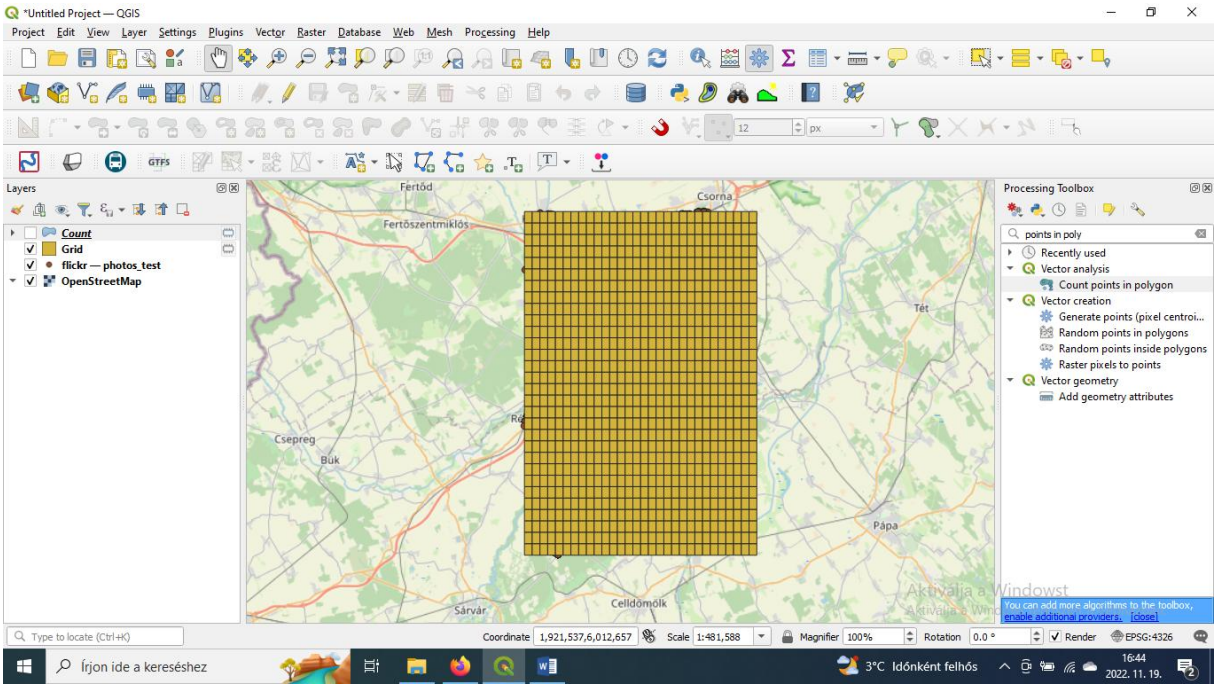
1°~> 111.1 km **0.01°~> 1.1 km**

0.1°~> 11.1 km 0.001°~> 111 m

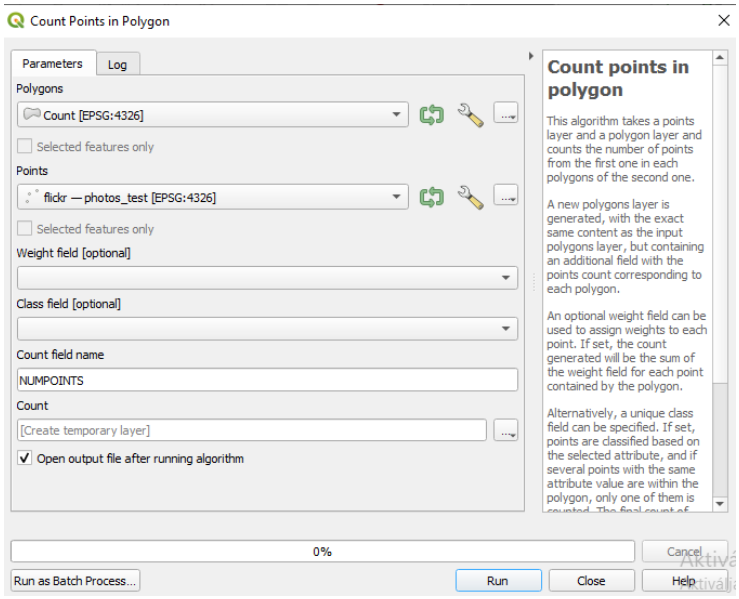
Choose the Rectangle type for the grid. Set the grid's extent to inherit from the **photo_test layer.**



The result:



First, count the points in a polygon using the "Count Points in Polygon" (Processing) tool. This tool writes the number of points into the attribute table. Create a Shapefile layer as the output.



In Python, we will write a program that reads this NUMPOINTS field and the geometry to generate a KML file. The KML will contain spatial bar diagrams.

To do that, we use the OSGEO GDAL/OGR Python modules. Please install them before usage.

First, import the ogr module, which handles vector data. The Osr module is used for projections.

```
from osgeo import ogr
from osgeo import osr
```

The KML file will be written as a text file. First, I define the header and styles.

The bar diagram columns will have the following colors: yellow, red, and brown. In hexadecimal notation: *colors=['ff00ffff', 'ff0000ff', 'ff336699']*.

```
colors=['ff00ffff', 'ff0000ff', 'ff336699']
f = open("pelda.kml", "w")
f.write('<?xml version="1.0" encoding="utf-8" ?><kml
xmlns="http://earth.google.com/kml/2.2"><Document> <name>Pontok a
polyban</name>')
for i in range(0, len(colors)):
    f.write('<Style id="PolyColor'+str(i)+'"> \n ')
    f.write(' <LineStyle> \n ')
    f.write('<width>1.5</width> \n ')
    f.write('</LineStyle> \n ')
    f.write('<PolyStyle>')
    f.write('<color>'+colors[i]+'</color> \n ')
    f.write(' </PolyStyle> \n ')
    f.write(' </Style> \n ')
```

Next, define a driver and open the Shapefile for reading. Retrieve the layer and count the number of features in it.

```
driver = ogr.GetDriverByName("ESRI Shapefile")
dataSource = driver.Open('pointsingrid.shp', 0)
layer = dataSource.GetLayer()
featureCount = layer.GetFeatureCount()
```

Iterate over the features. If NUMPOINTS is null, no column is created. If NUMPOINTS is greater than 0:

-Start writing the placemark.

- Determine the column height. (we use *NUMPOINTS *20* , to emphasize the column size through vertical exaggeration). Finally, set the appropiate style.

```
for feature in layer:
    geom = feature.GetGeometryRef()
    numPoints=feature.GetField("NUMPOINTS")
    if numPoints!=0:

        f.write('<Placemark>')
        if numPoints*20<=20:
            f.write('<styleUrl>#PolyColor0</styleUrl> ')
        elif numPoints*20>20 and numPoints*20<=100:
            f.write('<styleUrl>#PolyColor1</styleUrl> ')
        elif numPoints*20>100:
            f.write('<styleUrl>#PolyColor2</styleUrl> ')
        else:
            pass
```

The polygon is a rectangle (defined by five coordinates, where the first and last are the same) and is extruded to the corresponding height, determined by the NUMPOINTS value.

The *GetPoint()* function provides the coordinates (latitude and longitude) of the point. The third coordinate represents the height.

Finally, close the placemark…

```
          f.write('<Polygon><extrude>1</extrude><tessellate>1
</tessellate><altitudeMode>relativeToGround
      </altitudeMode><outerBoundaryIs><LinearRing><coordinates>')
          geom=feature.GetGeometryRef()
          ring=geom.GetGeometryRef(0)
          points=ring.GetPointCount()

          for p in range(0,points):
              lon= ring.GetPoint(p)[0]
              lat= ring.GetPoint(p)[1]
              f.write(str(lon)+','+str(lat)+','+str(numPoints*20)+'
\n')

f.write('</coordinates></LinearRing></outerBoundaryIs></Polygon></P
lacemark>')
```

… and the files.

```
f.write('</Document></kml>')
f.close()
dataSource.Destroy()
```

Chapter 7. Working with Python OGR module

This chapter introduces the Python GDAL/OGR module. GDAL is a specialized for raster data, while OGR is used for vector data. Why is it important to familiarize youself with this OGR module? Geoinformatics software do not provide processing algorithms for every possible task, and sometimes you encounter unique problems. In such cases, you may need to write scripts to solve them.

The OGR module provides the fundamental tools of geoinformatics software in Python environment, which you can extend with your own functions.

Suggested literature

GDAL/OGR Cookbook

https://pcjericks.github.io/py-gdalogr-cookbook/index.html

GDAL Documentation

https://gdal.org/

Python OGR

https://gdal.org/api/python/osgeo.ogr.html

The first step to use OGR is learning how to handle the basic geometry types. In the next section, we will explore how to read different geometry types including single and multi-geometries. We will use Shapefiles in our examples.

The initial step for reading a file are always the same. Define a driver, which facilitates reading the file. Then, open the file for reading. Retrieve the layer (every Shapefile contains only one layer, but before proceeding, you must store it in a variable). Count number of the features in the layer.

```
from osgeo import ogr
driver = ogr.GetDriverByName("ESRI Shapefile")
dataSource = driver.Open('grid.shp', 0)
layer = dataSource.GetLayer()
featureCount = layer.GetFeatureCount()
```

Reading point features

Iterate over the features. Get geometry using the *GetGeometryRef()* function, and store it in the variable *geom*. Use the *GetField('fieldname')* function to get the attribute data from a field. *GetX()* and *GetY()* functions retrieve the coordinates.

```
for feature in layer:
    geom = feature.GetGeometryRef()
    field1=feature.GetField("p_id")
    lon= geom.GetX()
    lat= geom.GetY()
```

Reading line (LineString) features

Iterate over the features. Get geometry using the *GetGeometryRef()* function, and store it in the variable *geom*. Use the *GetField('fieldname')* function to get the attribute data from a field. Count how many vertices the line has. Write another for loop to iterate over the nodes of the line. Use *GetPoint(node)*, which returns a list. The first element of the list contains the longitude, while the second one contains the latitude.

```
for feature in layer:
    geom = feature.GetGeometryRef()
    field1=feature.GetField("id")
    points=geom.GetPointCount()
    for p in range(0,points):
        lon= geom.GetPoint(p)[0]
        lat= geom.GetPoint(p)[1]
```

Reading polygon features

Iterate over the features. Get geometry using the *GetGeometryRef()* function, and store it in the variable *geom*. Use the *GetField('fieldname')* function to get the attribute data from a field. Count the rings of the polygon using *GetGeometryCount()*. (The first ring is always the outer ring, and if the polygon has holes, these are the inner rings). Read the nodes of each ring using *GetPoint(node)* which returna a list. The first element of the list contains the longitude, while the second one contains the latitude.

```
for feature in layer:
    geom = feature.GetGeometryRef()
    field1=feature.GetField("id")
    for i in range(0,geom.GetGeometryCount()):
        ring=geom.GetGeometryRef(i)
        points=ring.GetPointCount()
        for p in range(0,points):
            lon= ring.GetPoint(p)[0]
            lat= ring.GetPoint(p)[1]
```

Reading MultiPoints features

Iterate over the features. Get geometry using the *GetGeometryRef()* function, and store it in the variable *geom*. Use the *GetField('fieldname')* function to get the attribute data from a field. Count the parts of the geometry, and iterate over them. Using the *GetX()* and *GetY()*, print the coordinates.

```
for feature in layer:
    geom = feature.GetGeometryRef()
    field1=feature.GetField("id")
    numgeom=geom.GetGeometryCount()
    print (numgeom)
    for i in range(0,numgeom):
        lon=geom.GetGeometryRef(i).GetX()
        lat=geom.GetGeometryRef(i).GetY()
```

Readin MultiLineString features

Iterate over the features. Get geometry using the *GetGeometryRef()* function, and store it in the variable *geom*. Use the *GetField('fieldname')* function to get the attribute data from a field. Count the parts of the geometry and iterate over them. Count the number of nodes in the line part. Read the part's geometry and print the latitude and longitude.

```
for feature in layer:
    geom = feature.GetGeometryRef()
    field1=feature.GetField("id")
    numgeom=geom.GetGeometryCount()
    for i in range(0,numgeom):
        line=geom.GetGeometryRef(i)
        points=line.GetPointCount()
        for p in range(0,points):
            lon= line.GetPoint(p)[0]
            lat= line.GetPoint(p)[1]
```

Reading the MultiPolygon features

Iterate over the features. Get geometry using the *GetGeometryRef()* function, and store it in the variable *geom*. Use the *GetField('fieldname')* function to get the attribute data from a field. Count the parts of the geometry using *GetGeometryCount()*, and iterate over them. Count the number of rings, and iterate over them. Create one more for loop to iterate over the ring's geometry. Print the coordinates of each nodes.

In cases where the file contains Polygons and MultiPolygons, we need to add a condition to check the number of geometries. If it consists of one part, it is a Polygon, requiring fewer loops. If it consists of more than one part, it is a MultiPolygon and we use the method mentioned above.

```
for feature in layer:
    geom = feature.GetGeometryRef()
    field1=feature.GetField("id")
    numgeom=geom.GetGeometryCount()
    if numgeom>1:
        for ng in range(0,numgeom):
            poly=geom.GetGeometryRef(ng)
            for i in range(0,poly.GetGeometryCount()):
                ring=poly.GetGeometryRef(i)
                points=ring.GetPointCount()
                for p in range(0,points):
                    lon= ring.GetPoint(p)[0]
                    lat= ring.GetPoint(p)[1]
    else:
        for i in range(0,geom.GetGeometryCount()):
            ring=geom.GetGeometryRef(i)
            points=ring.GetPointCount()

            for p in range(0,points):
                lon= ring.GetPoint(p)[0]
                lat= ring.GetPoint(p)[1]
```

Writing Shapefiles

This chapter demonstrates ho to write point, line and area objects into a file.

Writing points

Read the sample points.shp file and filter the features based on the date field (*p_date*). Write the result in a new Shapefile.

This code is partially similar to the codes in chapter Reading points, therefore I won't explain it detail again. The srs variable defines the coordinate reference system. At the beginning of the program, define the output file. The file driver is a Shapefile driver. Specify the type of the file such as *ogr.wkbPoint*, *ogr.wkbLineString* or *ogr.wkbPolygon*. Create a new field in the file and define the data type in the attribute table (e. g. *OFTInteger*, *OFTString*, *OFTDateTime*). More information about the data types can be found here: https://gdal.org/java/org/gdal/ogr/ogrConstants.html.

```
from osgeo import ogr
from osgeo import osr
driver = ogr.GetDriverByName("ESRI Shapefile")
dataSource = driver.Open('points.shp', 0)
layer = dataSource.GetLayer()
featureCount = layer.GetFeatureCount()

srs =  osr.SpatialReference()
srs.ImportFromEPSG(4326)
ds = driver.CreateDataSource("selected_points.shp")
layeruj = ds.CreateLayer("pontok", srs, ogr.wkbPoint)
dateField = ogr.FieldDefn("datum", ogr.OFTDateTime)
layeruj.CreateField(dateField)
featureDefn = layeruj.GetLayerDefn()
```

Filter the data using the following expression (*p_date>'2010-01-01'*). Create a new feature and define its type. Assign coordinates from the original feature. Set the geometry and attributes of the feature, then append it to the layer. Finally, close the file and complete the writing process.

```
for feature in layer:
    geom = feature.GetGeometryRef()
    field1=feature.GetField("p_date")
    if field1>'2010-01-01':
        lon= geom.GetX()
        lat= geom.GetY()
        feature = ogr.Feature(featureDefn)
        pont = ogr.Geometry(ogr.wkbPoint)
        pont.AddPoint(lon, lat)
        feature.SetGeometry(pont)
        feature.SetField("datum", field1)
        layeruj.CreateFeature(feature)

ds.Destroy()
```

Line Simplification

Read the *lines.shp* file and call the *Simplify(0.01)* funtion on the original features. Simpifying a line means performing automatic shape generalization using the Douglas–Peucker algorithm. The parameter is similar to the unit of the layer's CRS; here, we use degrees. The file reading process is similar to that in the Reading lines Chapter.

```
from osgeo import ogr
from osgeo import osr
driver = ogr.GetDriverByName("ESRI Shapefile")
dataSource = driver.Open('lines.shp', 0)
layer = dataSource.GetLayer()
featureCount = layer.GetFeatureCount()


srs =  osr.SpatialReference()
srs.ImportFromEPSG(4326)
ds = driver.CreateDataSource("simplified_line.shp")
layeruj = ds.CreateLayer("line", srs, ogr.wkbLineString)
idField = ogr.FieldDefn("id", ogr.OFTInteger)
layeruj.CreateField(idField)
featureDefn = layeruj.GetLayerDefn()
```

In this case, use the simplified geometry when writing to the new file (it is unnecessary to read the line node by node). Set the geometry and the attribute data, append the new feature to the layer and finalize the writing process with *Destroy()*.

```
for feature in layer:
    geom = feature.GetGeometryRef()
    newgeom=geom.Simplify(0.01)
    field1=feature.GetField("id")
    feature = ogr.Feature(featureDefn)
    feature.SetGeometry(newgeom)
    feature.SetField("id", field1)
    layeruj.CreateFeature(feature)

ds.Destroy()
```

Writing polygons

Generate the bounding boxes of each line in *lines.shp* and write these polygons into a new Shapefile.

```
from osgeo import ogr
from osgeo import osr
driver = ogr.GetDriverByName("ESRI Shapefile")
dataSource = driver.Open('lines.shp', 0)
layer = dataSource.GetLayer()
featureCount = layer.GetFeatureCount()


srs =  osr.SpatialReference()
srs.ImportFromEPSG(4326)
ds = driver.CreateDataSource("bounding_boxes.shp")
layeruj = ds.CreateLayer("line", srs, ogr.wkbPolygon)
idField = ogr.FieldDefn("id", ogr.OFTInteger)
layeruj.CreateField(idField)
featureDefn = layeruj.GetLayerDefn()
```

Iterate over the features and call the *GetEnvelope()* function, which returns the four bounding coordinates: *[West, East, South, North]*. First, create a LinearRing and add these coordinates to the ring considering the correct order. The first and the last coordinates of the ring should be the same. Next, create a polygon and add this ring to the polygon. Create a new feature from the polygon. and write into the new file.

```
for feature in layer:
    geom = feature.GetGeometryRef()
    env=geom.GetEnvelope()
    ring = ogr.Geometry(ogr.wkbLinearRing)
    ring.AddPoint(env[0],env[3])
    ring.AddPoint(env[0],env[2])
    ring.AddPoint(env[1],env[2])
    ring.AddPoint(env[1],env[3])
    ring.AddPoint(env[0],env[3])
    poly = ogr.Geometry(ogr.wkbPolygon)
    poly.AddGeometry(ring)

    field1=feature.GetField("id")
    feature = ogr.Feature(featureDefn)
    feature.SetGeometry(poly)
    feature.SetField("id", field1)
    layeruj.CreateFeature(feature)

ds.Destroy()
```

Useful links in this topic:

https://www.gis.usu.edu/~chrisg/python/2009/lectures/ospy_slides1.pdf

https://www.gis.usu.edu/~chrisg/python/2009/lectures/ospy_slides2.pdf

https://www.gis.usu.edu/~chrisg/python/2009/lectures/ospy_slides3.pdf

https://www.gis.usu.edu/~chrisg/python/2009/lectures/ospy_slides4.pdf

https://www.gis.usu.edu/~chrisg/python/2009/lectures/ospy_slides5.pdf

https://www.gis.usu.edu/~chrisg/python/2009/lectures/ospy_slides6.pdf

https://www.gis.usu.edu/~chrisg/python/2009/lectures/ospy_slides7.pdf

Chapter 8. The structure of KML files

For more examples and explanations, please visit the official documentation:

https://developers.google.com/kml/documentation

A **KML (Keyhole Markup Language) file** is an XML-based format used to store geographic data and display it in applications like **Google Earth**, **Google Maps**, and **GIS software**. The KML file structured as follows:

```
<property> value </property>
```

All elements have an opening and a closing tag, with the value placed between them.

The header of the file depends on whether we want to use animation or not. However, it is generally recommended to use the longer header, which allows for animation and gx elements.

```
<?xml version="1.0" encoding="UTF-8"?>
<kml xmlns="http://www.opengis.net/kml/2.2"
xmlns:gx="http://www.google.com/kml/ext/2.2">
```

The KML file always contains a header. After the headers, there is a Style definition followed by the geometry. Each geometry is a Placemark element.

In the style definition, always add an id for reference:

```
<Style id="poligongroup1">
<PolyStyle>
        <color>64B40014</color>
        <fill>1</fill>
        <outline>0</outline>
</PolyStyle>
</Style>
```

To apply this style to a **Placemark**, use styleUrl withn the element:

```
<styleUrl>#poligongroup1</styleUrl>
```

KML can contain the following geometries (types of a Placemark): *Point, LineString, Polygon, MultiPoint, MultiLineString, MultiPolygon, GeometryCollection*.

Let's check an example, which contains a polygon element with style definition, name and description. A poupup window will open if the Placemark (feature) has a description.

```
<Placemark>
     <name>Give me a name!</name>
     <description>This is a polygon with a style. </description>
     <styleUrl>#PolyColor0</styleUrl> <Polygon>
     <extrude>1</extrude>
     <tessellate>1</tessellate>
     <altitudeMode>relativeToGround</altitudeMode>
     <outerBoundaryIs>
     <LinearRing>
     <coordinates>17.00414,47.559819,20.0
     17.01414,47.559819,20.0
     17.014140,47.549819,20.0
     17.004140,47.549819,20.0
```

```
      17.004140,47.559819,20.0
      </coordinates>
</LinearRing><
/outerBoundaryIs>
</Polygon>
</Placemark>
```

Let's get familiar with the 3D elements. If the Z coordinate is not 0, the feature can be a 3D element. If we apply the altitudeMode=relativeToGround, the elevation is interpreted above the ground. Extrude=1 means that the feature is connected to the ground. Extrude=0 means the element is floating above the ground. If the altitudeMode is clampToGround the features can not be visualized as 3D elements, only as 2D elements. Other possible values of altitudeMode are: absolute which means the absolute height above the ellipsoid surface, and clampToSeaFloor or relativeToSeaFloor which are very similar to clampToGround and relativeToGround.

When tessellate is set to 1 (true), the geometry follows the terrain and adapts to the Earth's curvature. When tessellate is set to 0 (false), the geometry is drawn as a straight line in 3D space, which may appear to float above the ground.

There are two types of animations: *TimeStamp* and *TimeSpan*.

TimeSpan animation works with beginning and closing dates. These defines that the features appear in the scene.

```
<TimeSpan>
      <begin>1986</begin>
      <end>2023</end>
</TimeSpan>
```

The *gx:track* animation is used to visualize moving object along a path. It requires a timestamps and coordinate pairs. A good example is a hiker moving along a trail.

The timestamp looks like this: year-month-dayThour:minute:secondZ

The *gx:coord* format differs from the regular coordinates: it has no comma between longitude, latitude, and elevation.

```
<Placemark>
<gx:Track>
      <when>2017-01-02T17:00:22Z</when>
      <when>2017-01-02T17:03:22Z</when>
      …
      <gx:coord>19.03814980042824 47.46165145965755 0</gx:coord>
      <gx:coord>19.03837381154784 47.46229927027417 0</gx:coord>
      …
</gx:Track>
      <name>Take a virtual walk together!</name>
      <description> Lets explore the Buda Castle quarter!
</description>
      <Style>
            <IconStyle>
               <Icon><href>traveller.png</href></Icon>
            </IconStyle>
      </Style>
</Placemark>
```

Chapter 9. Installing Python modules from WHEEL files

Installing Python modules depends on the software environment. The easiest way is through PyCharm, where you can install modules directly within the program.

However, sometimes certain modules cannot be installled this way. In such cases, here's how you can manually install them.

Installing Python modules manually

If you have Windows operation system, please open Windows Command Line Prompt (cmd.exe).

Download the WHEEL file from Python Package Index website (https://pypi.org/

) or from this unoffical collection (https://www.lfd.uci.edu/~gohlke/pythonlibs/).

What is a WHEEL file?

A Wheel file (.whl) is a binary package format for Python distributions. It is a faster and more efficient way to install Python packages compared to the traditional source distribution (.tar.gz or .zip).

How to install geoPy from WHEEL file?

1. Download the .whl file and save it in the C:/Users/YOURUSERNAME folder.
2. Navigate tot he folder in the Command Prompt usidng cd command.

```
C:\Users\ungvarizs>
```

3. Install the package using PIP.

```
py -m pip install C:\Users\ungvarizs\Downloads\geopy-2.2.0-py3-
none-any.whl
```

If the new Python module has dependencies, PIP will automatically install them.

Modules covered in this course

`GDAL`: https://pypi.org/project/GDAL/#files

`pandas`: https://pypi.org/project/pandas/#files

`openpyxl`: https://pypi.org/project/openpyxl/

`geoPy`: https://pypi.org/project/geopy/