Geo-Data and Geo-Data Management

Lecture 3: Vector Data Integration

Yingjing Huang

yingjing.huang@univie.ac.at

2026W

Today's Roadmap

1



Lecture

- Python basics + Week 2 Checkpoint starter
- Spatial join and attribute join (Vienna case)

2



Lab

- Run and inspect the guided Vienna notebook
- Complete the checkpoint

One Cell, One Task

After each calculation or function call, check the new variable right away.

jupyter

In [1]:
```
route_length_m = 120 + 180
print(route_length_m)
```

```
300
```

In [2]:
```
route_length_km = round(route_length_m / 1000, 2)
print(route_length_km)
```

```
0.3
```

Calculate → inspect the result → continue. Each step is easier to debug.

Cells Share Variables and Imports

Once code has run, later cells can use the same variables and packages.

jupyter

In [3]:
```
import pandas as pd
city = "Vienna"
```

In [4]:
```
print(city)
print(pd.__name__)
```

```
Vienna
pandas
```

You do not need to redefine a variable or import a package in every cell.

The Order of Execution Matters

A later cell can only use values that already exist in the running kernel.

jupyter

In [5]:
```
print(place)
```

```
NameError: name 'place' is not defined
```

In [6]:
```
place = "Vienna"
print(place)
```

```
Vienna
```

In [n] shows execution order. Restart clears the session; Run All rebuilds it.

A Variable Gives a Value a Name

Use = to assign a value and print() to inspect it.

jupyter

In [7]:
```
city = "Vienna"
number_of_districts = 4
print(city)
print(number_of_districts)
```

```
Vienna
4
```

Re-running a cell with a new assignment updates the variable.

Numbers, Text and True or False

Different value types support different operations.

jupyter

In [8]:
```
district_id = 2
population = 112675.0
label = "2"
valid = True
print(type(district_id).__name__)
print(type(population).__name__)
print(district_id == label)
```

```
int
float
False
```

Common types are int, float, str and bool. The number 2 differs from "2".

Reading Python Brackets

Look at what the brackets are doing, not just their shape.

jupyter

In [9]:
```
names = ["Innere Stadt", "Wieden"]
xy = (16.361, 48.211)
locations = {"A": xy}
print(names[0])
print(locations["A"])
print(len(names))
```

```
Innere Stadt
(16.361, 48.211)
2
```

[ ] can create a list or access an item. ( ) can group values or call a function.

Lists Store Ordered Values

Each value has a position, and Python starts counting at zero.

jupyter

In [10]:
```
districts = ["Innere Stadt", "Leopoldstadt",
             "Landstraße", "Wieden"]
print(districts[0])
print(districts[3])
print(len(districts))
```

```
Innere Stadt
Wieden
4
```

The first value is at index 0; the fourth value is at index 3.

A Tuple Holds a Coordinate Pair

Keep longitude and latitude together in the correct order.

jupyter

In [11]:
```
xy = (16.3610, 48.2110)
longitude = xy[0]
latitude = xy[1]
print(longitude)
print(latitude)
```

```
16.361
48.211
```

A tuple keeps values together; the first coordinate is longitude (x).

Dictionaries Use Keys and Values

A dictionary finds a value by its name rather than its position.

jupyter

In [12]:
```
stop_xy = {
    "A": (16.3610, 48.2110),
    "B": (16.3660, 48.2100),
    "C": (16.3650, 48.1980),
}
print(stop_xy["A"])
```

```
(16.361, 48.211)
```

"A" is the key. The matching value is a coordinate tuple.

A Dictionary Can Store Tuples

Follow a key to a coordinate pair, then inspect its two values.

jupyter

In [13]:
```
candidate_xy = {
    "A": (16.3610, 48.2110),
    "B": (16.3660, 48.2100),
    "C": (16.3650, 48.1980),
}
xy_A = candidate_xy["A"]
print(xy_A[0], xy_A[1])
```

```
16.361 48.211
```

Dictionary → key "A" → tuple → longitude and latitude.

A Dictionary Can Store Lists of Tuples

One candidate can have a whole route made of several coordinate pairs.

jupyter

In [14]:
```
meeting_xy = (16.3540, 48.2070)
given_route_xy = {
    "A": [meeting_xy, (16.3540, 48.2100),
          (16.3580, 48.2100), candidate_xy["A"]]
}
route_A = given_route_xy["A"]
print(len(route_A))
print(route_A[0])
```

```
4
(16.354, 48.207)
```

Dictionary → key "A" → list of coordinate tuples (a route).

Importing Python Libraries

Import a package once, then use its functions in later cells.

jupyter

In [15]:
```
import pandas as pd
import geopandas as gpd
from shapely.geometry import Point
print(pd.__name__)
print(gpd.__name__)
```

```
pandas
geopandas
```

as defines a short name; from ... import selects something from a module.

Calling Functions and Passing Arguments

Values go into a function; the function returns or displays a result.

jupyter

In [16]:
```
from shapely.geometry import Point
xy_A = (16.3610, 48.2110)
site_A = Point(xy_A)
print(site_A.geom_type)
print(round(3.14159, ndigits=2))
```

```
Point
3.14
```

Point(xy_A) passes an argument. ndigits=2 is a named argument.

Objects Have Attributes and Methods

A dot can inspect an object or call something it knows how to do.

jupyter

In [17]:
```
from shapely.geometry import Point
p = Point(2, 3)
q = Point(5, 7)
print(p.geom_type)
print(p.x, p.y)
print(p.distance(q))
```

```
Point
2.0 3.0
5.0
```

An attribute like p.x is read without ( ); a method like distance() is called.

For Loops Repeat an Action

Indented code runs once for each item in a collection.

jupyter

In [18]:
```
districts = ["Innere Stadt", "Leopoldstadt",
             "Landstraße", "Wieden"]
for name in districts:
    print(name)
print("Done")
```

```
Innere Stadt
Leopoldstadt
Landstraße
Wieden
Done
```

The colon starts the loop; indentation controls what repeats.

If Statements Check Conditions

True or False determines which block of code runs.

jupyter

In [19]:
```
number_of_districts = 4
if number_of_districts == 4:
    print("We have four districts")
else:
    print("Check the data")
```

```
We have four districts
```

Use = to assign a value and == to compare two values.

A pandas DataFrame Is a Table

Columns have names; each row is one record.

jupyter

In [20]:
```
import pandas as pd
df = pd.DataFrame({
    "district": ["Innere Stadt", "Wieden"],
    "population": [16281, 33444]
})
df
```

	district	population
0	Innere Stadt	16281
1	Wieden	33444

The leftmost 0 and 1 are the DataFrame index, not district IDs.

Inspecting and Selecting DataFrames

Read a few rows, choose a column, or select a row by position.

jupyter

In [21]:
```
print(df["district"].iloc[0])
print(df.columns.tolist())
print(len(df))
df.head()
```

```
Innere Stadt
['district', 'population']
2
```

	district	population
0	Innere Stadt	16281
1	Wieden	33444

Explore more: pandas documentation — 10 minutes to pandas

GeoPandas Adds Geometry to a Table

A GeoDataFrame combines ordinary attributes with geographic shapes.

jupyter

In [22]:
```
import geopandas as gpd
from shapely.geometry import Point
site = Point(16.361, 48.211)
sites = gpd.GeoDataFrame(
    {"site": ["A"]}, geometry=[site],
    crs="EPSG:4326"
)
print(sites["site"].iloc[0])
print(sites.geometry.iloc[0].geom_type)
```

```
A
Point
```

Explore more: GeoPandas documentation — Introduction

Find a File Before Reading It

A relative filename is looked up from Python's current working directory.

jupyter

In [23]:
```
from pathlib import Path
data_path = Path("Vienna_Population_2025.csv")
print(data_path.exists())
```

```
True
```

Use pathlib to check a path before trying to read a table.

Read a CSV File with pandas

read_csv() turns a CSV file into a DataFrame; inspect it right away.

jupyter

In [24]:
```
import pandas as pd
population = pd.read_csv("Vienna_Population_2025.csv")
print(len(population))
print(population.columns.tolist())
```

```
4
['district_id', 'population_2025']
```

CSV stores ordinary rows and columns; it does not contain geometry.

Read GeoJSON with GeoPandas

read_file() loads geographic features, including their geometry and CRS.

jupyter

In [25]:
```
import geopandas as gpd
districts = gpd.read_file("Vienna_Districts_1_to_4.geojson")
print(len(districts))
print(districts.crs)
```

```
4
EPSG:4326
```

The same function can read GeoJSON, Shapefile and GeoPackage files.

Read and Write Shapefiles

Shapefiles use several files; keep the matching parts together.

jupyter

In [26]:
```
districts.to_file("practice_districts.shp")
loaded = gpd.read_file("practice_districts.shp")
print(len(loaded), loaded.crs)
```

```
4 EPSG:4326
```

A Shapefile normally needs .shp, .shx, .dbf and .prj together.

Three Geometry Types, One Construction Pattern

A Point is the simplest place to learn how a geometry object is created.

POINT



Meeting place or survey stop

```
Point(16.3600, 48.2050)
```

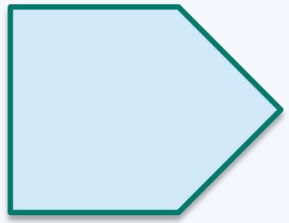
LINESTRING



The planned route

```
LineString([...])
```

POLYGON



The survey zone

```
Polygon([...])
```

Point = location · LineString = given route · Polygon = survey zone

The Checkpoint Gives Us Three Alternatives

A, B, C: minimum zone area 18,000 m²; then shortest given route.

jupyter

In [28]:
```
meeting_xy = (16.3540, 48.2070)
candidate_xy = {
    "A": (16.3610, 48.2110),
    "B": (16.3660, 48.2100),
    "C": (16.3650, 48.1980),
}
print(list(candidate_xy))
print(candidate_xy["A"])
```

```
['A', 'B', 'C']
(16.361, 48.211)
```

These are supplied, synthetic coordinates—not real street routes or survey sites.

Candidate A: Build and Check the Point

Retrieve the coordinate tuple first; turn it into a Point and inspect the object.

jupyter

In [29]:

```
from shapely.geometry import Point
xy_A = candidate_xy["A"]
print(xy_A)
site_A = Point(xy_A)
print(site_A, site_A.geom_type)
```

```
(16.361, 48.211)
POINT (16.361 48.211) Point
```

The tuple contains coordinates. Point(...) creates the geometry object.

The Same Pattern Builds the Route and Zone

LineString and Polygon take the supplied coordinate lists for candidate A.

jupyter

In [30]:
```
from shapely.geometry import LineString, Polygon
route_A = LineString(given_route_xy["A"])
zone_A = Polygon(survey_area_xy["A"])
print(route_A.geom_type, zone_A.geom_type)
print(zone_A.is_valid)
```

```
LineString Polygon
True
```

Retrieve coordinates → construct geometry → check the result.

Candidate A: Keep Geometry and Attributes Together

Three one-row GeoDataFrames keep the Point, route and zone linked to A.

jupyter

In [31]:
```
import geopandas as gpd
sites_A = gpd.GeoDataFrame({"site":["A"]},
                geometry=[site_A], crs="EPSG:4326")
routes_A = gpd.GeoDataFrame({"site":["A"]},
                 geometry=[route_A], crs="EPSG:4326")
zones_A = gpd.GeoDataFrame({"site":["A"]},
                geometry=[zone_A], crs="EPSG:4326")
print(len(sites_A), len(routes_A), len(zones_A))
```

```
1 1 1
```

Each layer has one feature and a known source CRS: EPSG:4326.

Transform the Three Layers Before Measuring

Longitude and latitude are degrees; use EPSG:32633 for metres.

jupyter

In [32]:
```
sites_utm_A = sites_A.to_crs("EPSG:32633")
routes_utm_A = routes_A.to_crs("EPSG:32633")
zones_utm_A = zones_A.to_crs("EPSG:32633")
print(routes_utm_A.crs.to_epsg())
```

```
32633
```

to_crs() transforms coordinates; set_crs() only labels existing coordinates.

Candidate A: Calculate Route Length and Area

The given route is about 880 m; the survey zone covers about 13,215 m².

jupyter

In [33]:

```
route_m_A = routes_utm_A.geometry.length.iloc[0]
area_m2_A = zones_utm_A.geometry.area.iloc[0]
print(round(route_m_A), "m")
print(round(area_m2_A), "m²")
```

```
880 m
13215 m²
```

Use projected geometry: .length gives metres; .area gives square metres.

Candidate A Does Not Meet the Area Rule

13,215 m² is less than the required 18,000 m².

01

Check the zone area

13,215 m² < 18,000 m²

02

Check the route

Given route ≈ 880 m

03

Make the decision

A does not qualify

Repeat the Full Workflow with a For Loop

Repeat every step for A, B and C, starting from the original coordinates.

jupyter

In [34]:
```
for option in candidate_xy:
    point = Point(candidate_xy[option])
    route = LineString(given_route_xy[option])
    zone = Polygon(survey_area_xy[option])
    points = gpd.GeoDataFrame(geometry=[point], crs=4326).to_crs(32633)
    routes = gpd.GeoDataFrame(geometry=[route], crs=4326).to_crs(32633)
    zones = gpd.GeoDataFrame(geometry=[zone], crs=4326).to_crs(32633)
    length = round(routes.length.iloc[0])
    area = round(zones.area.iloc[0])
    print(option, length, area, area >= 18000)
```

```
A  880   13215   False
B  1037  22797   True
C  1502  30072   True
```

For each option: create geometries → build layers → transform CRS → measure.

Put the Full Workflow into a Function

Define the entire workflow once, then call it for each option.

jupyter

In [35]:
```
def measure_option(option):
    point = Point(candidate_xy[option])
    route = LineString(given_route_xy[option])
    zone = Polygon(survey_area_xy[option])
    points = gpd.GeoDataFrame(geometry=[point], crs=4326).to_crs(32633)
    routes = gpd.GeoDataFrame(geometry=[route], crs=4326).to_crs(32633)
    zones = gpd.GeoDataFrame(geometry=[zone], crs=4326).to_crs(32633)
    return round(routes.length.iloc[0]), round(zones.area.iloc[0])

for option in candidate_xy:
    length, area = measure_option(option)
    print(option, length, area, area >= 18000)
```

```
A  880   13215   False
B  1037  22797   True
C  1502  30072   True
```

The function does every step; the loop calls it for A, B and C.

Going Forward, Drawing Maps Is Optional

You can focus on correct geometries, measurements and explanations.

01

Many of you made maps

Great work last week

02

Styling takes practice

Labels · colours · legends

03

Future lab plotting

Optional, not required

Feel free to practise maps—but the geographic reasoning comes first.

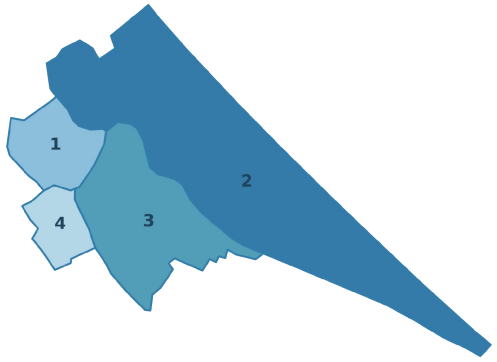
20-MINUTE BREAK

Next: Vector Data Integration in Vienna

Three Datasets, Three Ways of Representing Places

A row means something different in each file. Always inspect that first.

DISTRICT POLYGONS



4 areas

GeoJSON · one row per area

POPULATION TABLE

ID	Population
1	16,281
2	112,675
3	98,881
4	33,444

4 records

CSV · one row per district

FOUNTAIN POSITIONS



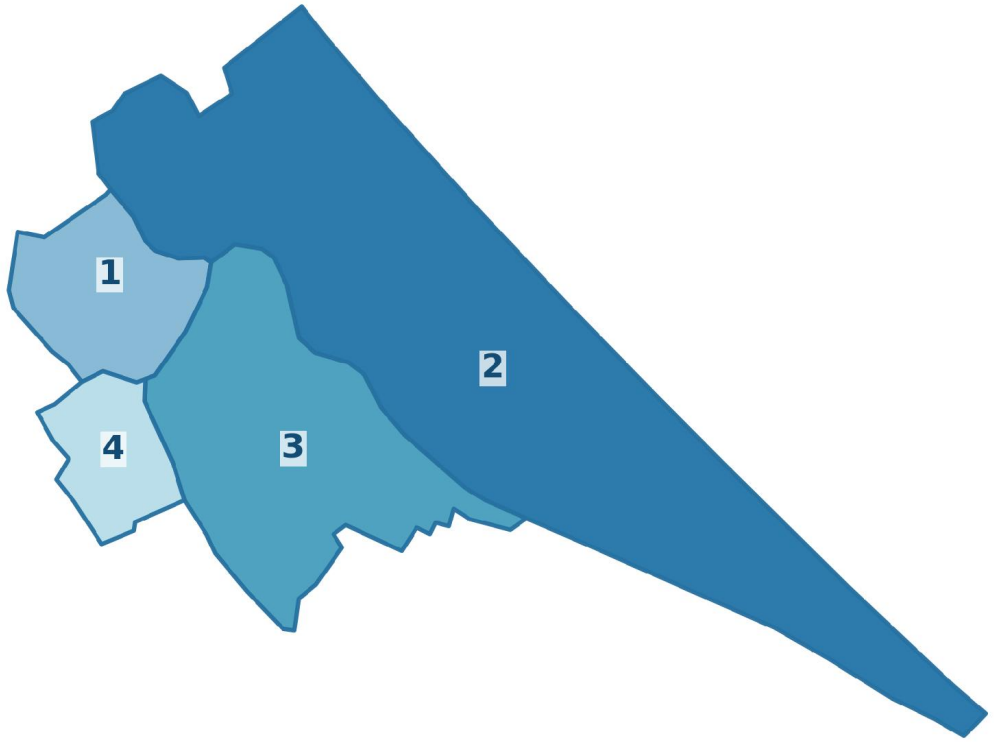
50 locations

CSV · one row per point

Two datasets describe the districts; the third describes individual fountains.

Our Question: Fountains Across Four Vienna Districts

We will combine the data into one four-row district profile.



THE QUESTION

How many public drinking fountains are in each district?

And how does that compare with the district population?

The same four district polygons will appear throughout the lecture.

Our Target: One Row per District

Our output has one row per district, even though its inputs have different row meanings.

District ID	District name	Population	Fountain count
1	Innere Stadt	?	?
2	Leopoldstadt	?	?
3	Landstraße	?	?
4	Wieden	?	?

First add population; then use fountain locations to calculate the counts.

Inside the District GeoJSON File

A feature has properties and geographic geometry—not just ordinary table columns.

GEOJSON • SHORTENED EXCERPT

```
{
  "type": "FeatureCollection",
  "features": [{
    "type": "Feature",
    "properties": {
      "district_id": 1,
      "district_name": "Innere Stadt"
    },
    "geometry": {
      "type": "MultiPolygon",
      "coordinates": [ ... ]
    }
  }, ...]
}
```

WHAT IS STORED?

4 district features

District ID + name

MultiPolygon geometry

Source: Vienna_Districts_1_to_4.geojson · 4 features · abbreviated excerpt

Load District Boundaries with GeoPandas

read_file() converts GeoJSON features into a GeoDataFrame with real geometry objects.

PYTHON • LOAD GEODATAFRAME

```python
import geopandas as gpd
districts = gpd.read_file("Vienna_Districts_1_to_4.geojson")
print(len(districts), districts.crs)
districts[["district_id", "district_name", "geometry"]].head(2)
```

GEODATAFRAME OUTPUT · 4 FEATURES · EPSG:4326

district_id	district_name	geometry
1	Innere Stadt	MULTIPOLYGON (...)
2	Leopoldstadt	MULTIPOLYGON (...)

Check: four rows · MultiPolygon · EPSG:4326

Inside the Population CSV

The file is a simple table: one row per district, with attributes but no geometry.

CSV • ORIGINAL FILE CONTENT

```
district_id,district_name,population_2025
1,Innere Stadt,16281
2,Leopoldstadt,112675
3,Landstraße,98881
4,Wieden,33444
```

WHAT IS STORED?

4 district records

Shared district_id

No geometry column

Shared field: district_id is present in both district and population files.

Read Population with pandas

read_csv() loads an ordinary table as a DataFrame.

PYTHON • LOAD DATAFRAME

```
import pandas as pd
population = pd.read_csv("Vienna_Population_2025.csv")
population.head()
```

DATAFRAME OUTPUT · 4 ROWS × 3 COLUMNS

district_id	district_name	population_2025
1	Innere Stadt	16281
2	Leopoldstadt	112675
3	Landstraße	98881
4	Wieden	33444

Each row is a district record; compare district_id with the GeoDataFrame.

Inside the Fountain CSV

A row represents a facility, not a district; latitude and longitude are ordinary columns.

CSV • FIRST THREE RECORDS (EXCERPT)

```
fountain_id,latitude,longitude,facility_type
TB-7136857,48.21191884,16.35947922,Trinkbrunnen
TB-7136860,48.20932048,16.35870724,Trinkbrunnen
TB-7136867,48.21087308,16.3595992,Trinkbrunnen
...
```

Source: Vienna_Trinkbrunnen_1_to_4.csv · 50 selected records

Read the Fountain CSV as a DataFrame

pd.read_csv() creates a table; longitude and latitude are still ordinary columns.

PYTHON · READ CSV WITH PANDAS

```
fountain_table = pd.read_csv("Vienna_Trinkbrunnen_1_to_4.csv")
fountain_table.head(3)
```

DATAFRAME OUTPUT · 50 ROWS × 4 COLUMNS

fountain_id	latitude	longitude	facility_type
TB-7136857	48.21192	16.35948	Trinkbrunnen
TB-7136860	48.20932	16.35871	Trinkbrunnen
TB-7136867	48.21087	16.35960	Trinkbrunnen

This is a pandas DataFrame: there is no geometry column yet.

Turn the DataFrame into a GeoDataFrame

points_from_xy() creates Point geometry from the longitude and latitude columns.

PYTHON · ADD GEOMETRY TO THE EXISTING TABLE

```
fountains = gpd.GeoDataFrame(
    fountain_table,
    geometry=gpd.points_from_xy(
        fountain_table["longitude"], fountain_table["latitude"]
    ),
    crs="EPSG:4326"
)
fountains[["fountain_id", "geometry"]].head(2)
```

GEODATAFRAME OUTPUT · 50 POINTS

fountain_id	geometry
TB-7136857	POINT (16.35948 48.21192)
TB-7136860	POINT (16.35871 48.20932)

HOW points_from_xy() WORKS

x ← longitude

y ← latitude

CRS: EPSG:4326

The same 50 records now have Point geometries and CRS EPSG:4326.

Check Every Column's Data Type

Check every column, especially the join key and the new fountain geometry.

PYTHON • CHECK ALL COLUMN TYPES

```
print(districts.dtypes)          # GeoDataFrame
print(population.dtypes)         # DataFrame
print(fountains.dtypes)          # GeoDataFrame
```

DISTRICT GEODATAFRAME

district_id	**int32**
district_name	object
geometry	geometry

POPULATION DATAFRAME

district_id	**int64**
district_name	object
population_2025	int64

FOUNTAIN GEODATAFRAME

fountain_id	object
latitude	float64
longitude	float64
facility_type	object
geometry	geometry

District IDs are compatible integers; fountain geometry is ready for the spatial join.

Attribute Join Uses a Shared District ID

The same numeric district_id tells us which records describe the same area.

DISTRICT BOUNDARY TABLE

district_id	district_name
1	Innere Stadt
2	**Leopoldstadt**
3	Landstraße
4	Wieden



POPULATION CSV

district_id	population_2025
1	16,281
2	**112,675**
3	98,881
4	33,444

The shared ID attaches population to the existing district polygons; geometry stays unchanged.

One merge() Attaches Population

Start with the district GeoDataFrame and retain all four left-side records.

PYTHON

```
district_population = districts.merge(
    population[[
        "district_id", "population_2025"
    ]],
    on="district_id",
    how="left"
)
```

OUTPUT: FOUR POLYGONS

District	Population
Innere Stadt	16,281
Leopoldstadt	112,675
Landstraße	98,881
Wieden	33,444

The output remains district-level (4 rows), with one additional population column.

Population Added: Check Our Four District Rows

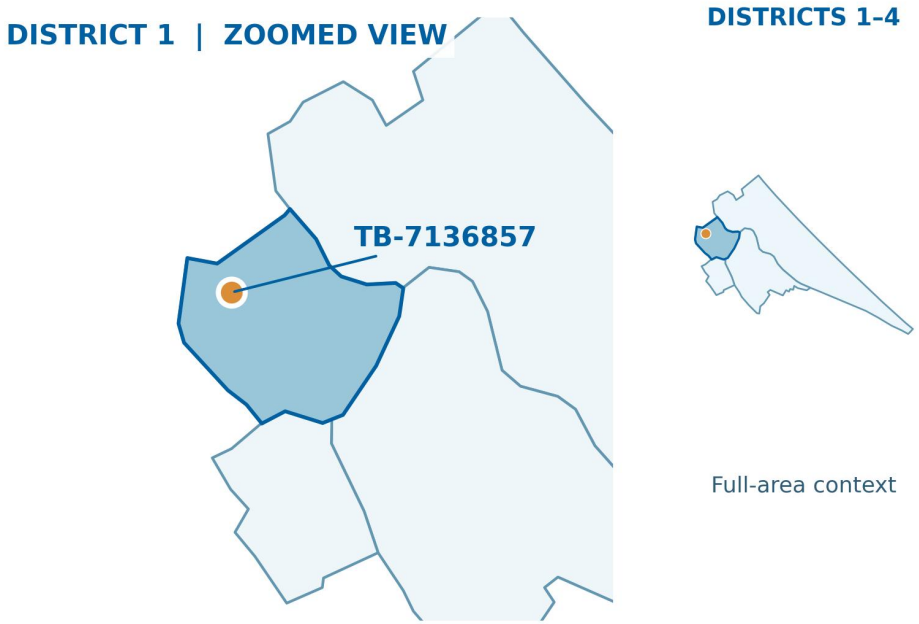
Our first new column is ready. Confirm the join produced four district records.

PYTHON CHECK	OBSERVED OUTPUT	
`len(district_population)`	**4**	District rows
`district_population.district_id.is_unique`	**True**	Unique district IDs
`district_population.population_2025.isna().sum()`	**0**	Missing values

These three checks show that the four-row population merge is structurally consistent.

We Have Population. Where Does This Point Belong?

The next column needs a district ID for every selected fountain Point.



Full-area context

ONE FOUNTAIN POINT

longitude	**16.35948**
latitude	**48.21192**

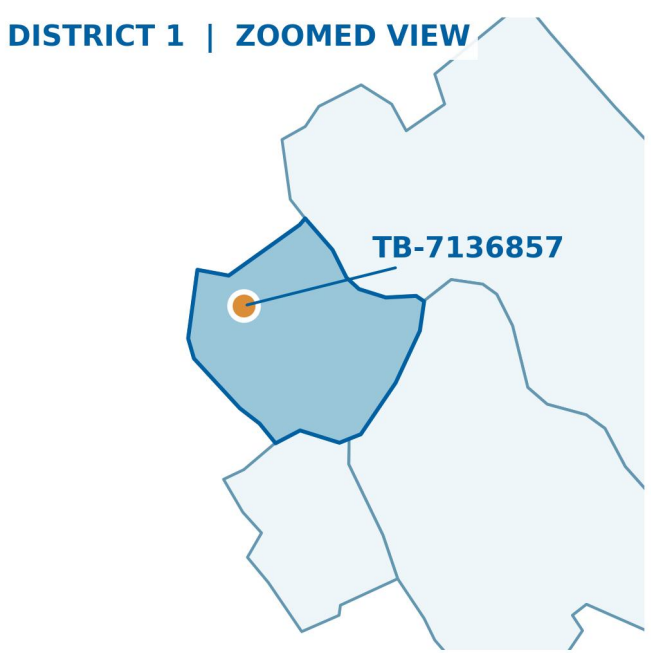
OUR NEXT QUESTION

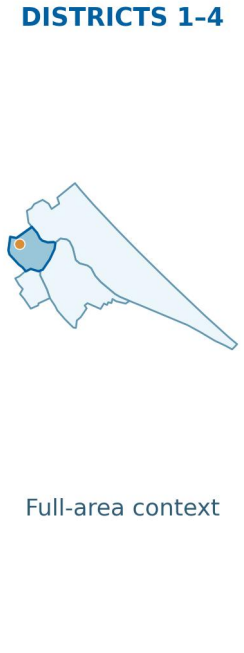
Which district contains this point?

Find its district first, then count it in the right group.

Inspect the Point Geometry We Created

The fountain GeoDataFrame already contains Points; check one before the spatial join.





FIRST FOUNTAIN FROM GEODATAFRAME

fountain_id: TB-7136857

geometry: POINT (16.35948 48.21192)

PYTHON · INSPECT A POINT

```
first = fountains.geometry.iloc[0]
print(first.geom_type)
print(fountains.crs)
print(first.is_valid)

# Point  ·  EPSG:4326  ·  True
```

The Points are ready; the next task is to find which district Polygon contains each one.

Plot Districts and Fountains Together

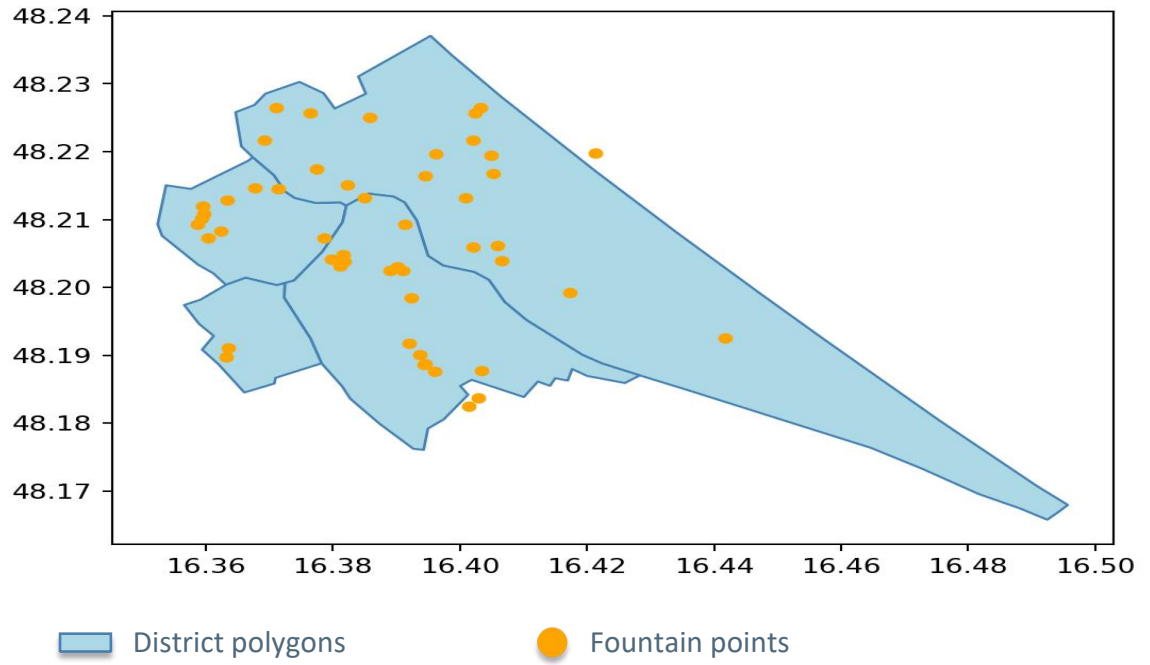
Use two GeoPandas plot() calls on the same axes to visualize both layers.

PYTHON · TWO LAYERS, ONE AXES

```
import matplotlib.pyplot as plt

ax = districts.plot(
    facecolor="lightblue",
    edgecolor="steelblue"
)
fountains.plot(
    ax=ax, color="orange",
markersize=18
)
plt.show()
```

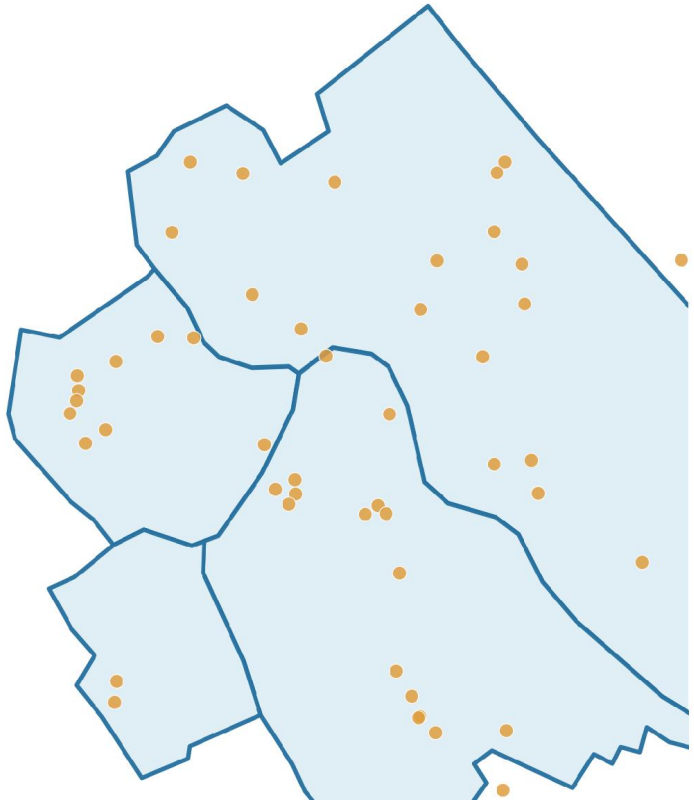
MAP OUTPUT · FOUR DISTRICTS + 50 POINTS



The map is an optional visual check; assigning districts still requires a spatial join.

Spatial Join: Location Becomes a District Attribute

For two layers in EPSG:4326, point-in-polygon answers which district contains a fountain.



GEOGRAPHIC RULE

Point within Polygon

The spatial join adds district information; the original point location stays unchanged.

Use sjoin() to Assign Districts

All 50 fountain locations remain on the left; district polygons supply new columns.

PYTHON

```
assigned = gpd.sjoin(
    fountains,
    districts[[
        "district_id",
        "district_name",
        "geometry"
    ]],
    how="left",
    predicate="within"
)
```



FIRST MATCHED ROWS

fountain_id	district_id
TB-7136857	1
TB-7136860	1
TB-7136867	1
TB-7142614	3
TB-7142659	1

Unlike merge(), sjoin() determines matches using geographic relationships.

What Does One Spatial-Join Row Represent?

The same point rows remain; district_id and district_name are added on matches.

BEFORE: POINT ATTRIBUTES

fountain_id	longitude	latitude
TB-7136857	16.3595	48.2119
TB-7136860	16.3587	48.2093
TB-7136867	16.3596	48.2109
TB-7142614	16.3797	48.2042

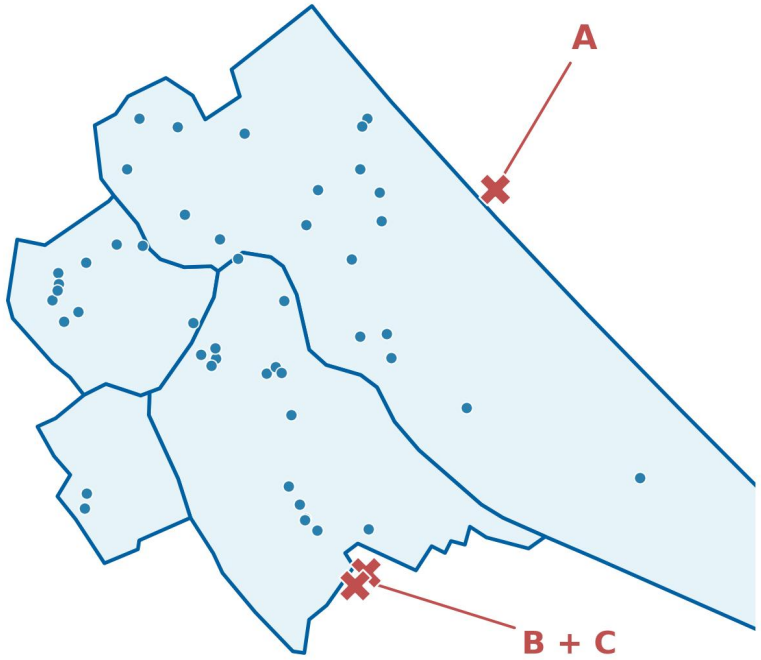
AFTER: POINT + DISTRICT

fountain_id	district_id	district_name
TB-7136857	1	Innere Stadt
TB-7136860	1	Innere Stadt
TB-7136867	1	Innere Stadt
TB-7142614	3	Landstraße

The right-hand table shows selected columns only; existing coordinates and geometry are retained.

Spatial Join Can Leave Points Unmatched

Of 50 selected points, 47 match a polygon and 3 have no within match.



47 MATCHED

Points assigned to a district polygon

3 UNMATCHED

A · TB-7144066
B · TB-7142098
C · TB-7142763

Source labels do not override the spatial result.

No within match in these simplified polygons does not mean the point is invalid.

From Many Fountain Rows to Four District Counts

We have district IDs for individual fountains. Next, count them by district.

ONE ROW PER FOUNTAIN

fountain_id	district_id
TB-7136857	1
TB-7136860	1
TB-7136867	1
…	…



ONE ROW PER DISTRICT

district_id	fountain_count
1	?
2	?
3	?
4	?

GroupBy will fill in the four missing counts on the right.

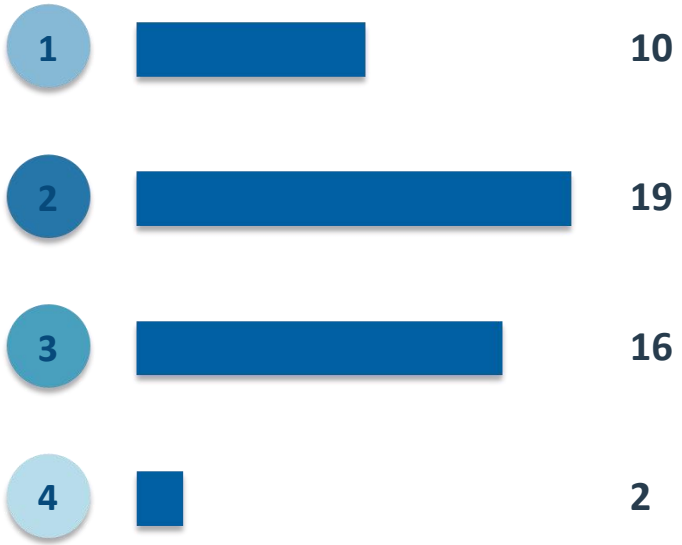
GroupBy Counts Points Sharing the Same District ID

After dropping unmatched points, count how many assigned rows belong to each district.

PYTHON

```
matched = assigned.dropna(
    subset=["district_id"]
)
fountain_counts = (
    matched.groupby("district_id")
    .size()
    .reset_index(name="fountain_count")
)
```

FOUR RESULTING GROUPS



The four counts now add up to the 47 matched fountain Points.

Merge GroupBy Counts Back to the District Polygons

The complete four-row district layer is the frame for the final result.

DISTRICTS + POPULATION

district_id	population_2025
1	16,281
2	112,675
3	98,881
4	33,444

COUNTS FROM GROUPBY

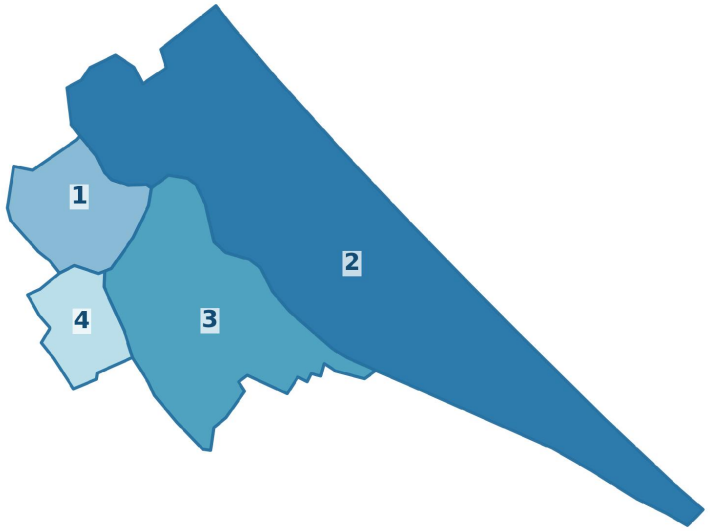
district_id	fountain_count
1	10
2	19
3	16
4	2

PYTHON

```
district_profile = district_population.merge(
    fountain_counts, on="district_id", how="left")
```

One District-Level Profile From Three Sources

One row now describes each district: its geometry, population and matched fountain count.



District	Population	Fountains
1 · Innere Stadt	16,281	10
2 · Leopoldstadt	112,675	19
3 · Landstraße	98,881	16
4 · Wieden	33,444	2

The result can be saved as a GeoDataFrame: four polygon geometries with new attributes.

Data Integration Enables Further Analysis

One combined district profile lets us move from data preparation to new questions.

AFTER THE JOIN

Integrated district profile

Geometry + Population + Fountain counts

WE CAN ALREADY EXPLORE

Fountains per resident?

Use existing counts and population to compare districts.

WITH MORE DATA, WE COULD ASK

How far must people walk?

Add walking routes and more complete fountain data.

Integration enables further analysis — it does not answer these questions by itself.

Attribute Join, Spatial Join, and Aggregation

Three operations: match by an ID, match by location, or summarize records.

Attribute Join

Match tables by a shared ID

`merge()` → Attach population

Spatial Join

Match features by location

`sjoin()` → Assign district IDs

Aggregation (GroupBy)

Group Points and count by district

`groupby().size()` → Count per district

Two joins attach attributes; aggregation summarizes many rows into one result per group.

Guided Lab: Recreate the Vienna Case

Sections 1–5 follow the same three datasets and the same workflow from the lecture.

District polygons
W03_Vienna_Districts.geojson

Population 2025
W03_Vienna_Population_2025.csv

Trinkbrunnen
W03_Vienna_Drinking_Fountains.csv



Your four-row output

District	Population	Count
Innere Stadt	?	?
Leopoldstadt	?	?
Landstraße	?	?
Wieden	?	?

Start with the required Python Basics in Section 0; then work through the Vienna example.

Independent Checkpoint: Public Toilets

Section 6 uses new toilet locations. Apply the same workflow, one small task per cell.

A	**A–B · Prepare**	Read the toilet CSV and create Point geometries.
B	**C–D · Locate**	Use a spatial join to find districts and unmatched points.
C	**E–F · Summarize**	Count the points with GroupBy and merge the results.
D	**G · Interpret**	Use your own results to answer three short data questions.

Optional plots help you explore. Save your notebook with outputs for Moodle.

Next Class: Friday, November 13

NO CLASS

Friday, October 30

Friday, November 6

SEE YOU AGAIN

Friday, November 13

11:30–14:30

Multimedia Mapping-Labor · NIG C0110

LAB TIME