A Practical Guide to Spatial Data

Outline

1. **Coordinates, CRS & Reprojection**: The essential foundation.
2. **Types of Spatial Data**: A quick overview of data models.
3. **Vector Data**: Working with points, lines, and polygons.
4. **Raster Data**: Grids, pixels, and surfaces.
5. **Network Data**: Nodes, edges, and connectivity.
6. **Further Learning**: Curated tutorials and resources.

1. Coordinates, CRS & Reprojection

What is a Coordinate Reference System (CRS)?

A CRS defines how coordinates map to a real-world location on Earth. Getting this wrong is the **#1 source of errors** in spatial analysis.

Geographic CRS

- Uses latitude and longitude on a spherical model.
- Units are in degrees.
- Good for global location, bad for measuring distance or area.
- **Example**: `EPSG:4326` (WGS 84)

Projected CRS

- Flattens the Earth onto a 2D surface.
- Units are in meters or feet.
- Essential for accurate distance, area, and buffer calculations.
- **Example**: `EPSG:3857` (Web Mercator), UTM Zones (e.g., `EPSG:32633`)

The Golden Rules of CRS: Rule 1

1. Always Know Your CRS: After loading data, immediately check the `.crs` attribute. If it's `None`, the data is unusable until you define it.

The Golden Rules of CRS: Rule 2

2. NEVER Guess: If the CRS is missing, find it in the metadata or from the data provider. Do not assume it's WGS 84.

The Golden Rules of CRS: Rule 3

3. Use `set_crs` vs. `to_crs` Correctly: - `gdf.set_crs("EPSG:4326")` : **Assigns** a CRS to data that has none. It **does not change the data**. Only use it if you are 100% certain what the CRS should be. - `gdf.to_crs("EPSG:3857")` : **Reprojects** the data, creating new coordinate values. Use this to transform data from one CRS to another.

Vector Reprojection with GeoPandas

This example shows how to load data, check its CRS, and reproject it for area calculations.

```
import geopandas as gpd

# 1. Load the data
gdf = gpd.read_file("data/berlin-neighbourhoods/berlin-neighbourhoods.geojson")

# 2. Always check the CRS
print(f"Original CRS: {gdf.crs}")

# 3. Reproject to a suitable projected CRS for measurements (e.g., UTM Zone 33N)
# This is CRITICAL for accurate area/distance calculations.
gdf_utm = gdf.to_crs("EPSG:32633")
print(f"New CRS: {gdf_utm.crs}")

# 4. Now, you can accurately calculate area in square meters
gdf_utm['area_sqkm'] = gdf_utm.geometry.area / 1_000_000
print(gdf_utm[['neighbourhood', 'area_sqkm']].head())
```

Library Guide: pyproj (1/2)

- What it is: Python bindings to the PROJ library for coordinate reference systems and transformations.
- Why it matters: Accurate, standards-compliant reprojection; the engine behind GeoPandas/Rasterio CRS handling.
- Core features:
  - CRS objects (EPSG, WKT, proj strings), databases, and authority lookups.
  - Transformer pipelines for fast lon/lat <-> projected conversions (NumPy-friendly).
  - Area-of-use and operation metadata to pick the best transform.

Library Guide: pyproj (2/2)

- Best practices:
  - Use `Transformer.from_crs(..., always_xy=True)` to ensure (lon, lat) order.
  - Cache and reuse transformers inside loops; avoid recreating per point.
  - Don’t “fix” data with `set_crs` when you need `to_crs` in GeoPandas (assign vs transform).
- Use it when:
  - You need to transform raw coordinate arrays/tuples without a GeoDataFrame.
  - Building custom pipelines or validating CRS definitions.

Low-Level Reprojection with `pyproj` (setup)

Sometimes you only need to transform a few coordinates. `pyproj` is perfect for this. `geopandas` uses it under the hood.

```
from pyproj import CRS, Transformer

# 1) Define the source and destination CRS
# From WGS 84 (lon/lat) to Web Mercator (meters)
source_crs = CRS("EPSG:4326")
dest_crs = CRS("EPSG:3857")

# 2) Create a transformer
# always_xy=True ensures (lon, lat) -> (x, y) order
transformer = Transformer.from_crs(source_crs, dest_crs, always_xy=True)
```

Low-Level Reprojection with `pyproj` (transform)

```
# 3) Define a lon/lat point (San Diego)
lon, lat = -117.16, 32.71

# 4) Transform the point
x, y = transformer.transform(lon, lat)

print(f"Original (lon, lat): ({lon}, {lat})")
print(f"Projected (x, y): ({x:.2f}, {y:.2f})")
```

CRS & Reprojection: Further Learning

- **Pyproj Documentation**: The library that powers CRS transformations in Python.
  - https://pyproj4.github.io/pyproj/stable/
- **GeoPandas User Guide on CRS**: Practical examples and explanations.
  - https://geopandas.org/en/stable/docs/user_guide/projections.html
- **Pyproj Docs on CRS**: Understanding CRS objects.
  - https://pyproj4.github.io/pyproj/stable/api/crs/crs.html

2. Types of Spatial Data

A Quick Tour of Spatial Data Models

Vector

Represents discrete objects with exact boundaries.

- **Points**: Locations (cities, sensors)
- **Lines**: Paths (rivers, roads)
- **Polygons**: Areas (countries, parks)

Use for: Administrative boundaries, infrastructure, points of interest.

Raster

Represents continuous phenomena on a grid of cells (pixels).

- Each pixel has a value.
- Examples: Elevation, temperature, satellite imagery.

Use for: Environmental data, imagery, surface analysis.

Other important types include **Networks** (for routing) and **Point Clouds** (for 3D data).

3. Vector Data

Working with Points, Lines, and Polygons

Vector data is ideal for representing features with clear, defined shapes.

- **Core Python Libraries**:
  - `geopandas`: The essential tool. A `GeoDataFrame` is a `pandas.DataFrame` with a special `geometry` column.
  - `shapely`: Handles the geometry objects themselves.
  - `fiona`: For reading and writing data, powered by GDAL/OGR.

Library Guide: GeoPandas (1/2)

- What it is: Pandas + geometry column powered by Shapely; high-level vector GIS in Python.
- Why it matters: Read/write common formats, project, join, aggregate, and plot in a few lines.
- Core features:
  - I/O via `read_file` / `to_file` (GeoPackage, Shapefile, GeoJSON…)
  - CRS handling with `.set_crs` and `.to_crs`, spatial joins `sjoin`, overlays.
  - Fast spatial indexing (`.sindex`) and convenient plotting.

Library Guide: GeoPandas (2/3)

- Tips for performance:
  - Read only needed columns (`usecols`) and simplify geometry when appropriate.
  - Use spatial index and coarse bounding boxes before `sjoin` .
  - For very large tables, consider Parquet/Feather exports (geometry-less) and join back later.

Library Guide: GeoPandas (3/3)

- Best practices:
  - Verify `gdf.crs` on load; reproject to an appropriate projected CRS before measuring area/distance.
  - Prefer GeoPackage for robust, single-file outputs; avoid Shapefile’s limitations.
  - Use `predicate=` (e.g., "within", "intersects") in `sjoin` for clarity; pre-filter with bounding boxes.

- Use it when:
  - Performing tabular vector workflows, EDA, and cartography on small-to-medium datasets.

Library Guide: Shapely (1/2)

- What it is: Fundamental geometry engine (points, lines, polygons) and operations.
- Why it matters: Precise geometric predicates and operations underpin almost all vector workflows.
- Core features:
  - Constructive ops (buffer, union, difference, intersection), predicates (contains, intersects).
  - `shapely.ops` utilities (`unary_union`, snapping, linemerge) and prepared geometries for speed.

Library Guide: Shapely (2/2)

- Geometry tips:
  - Use `prepared` geometries for repeated predicate tests against a static geometry.
  - Beware floating-point precision; consider rounding or snapping in workflows sensitive to tiny slivers.

- Best practices:
  - Work in a projected CRS for distance/area/buffers; avoid degree units for metric math.
  - Check and fix invalid geometries (e.g., `buffer(0)` or `make_valid` when available).
  - Simplify cautiously for speed/visualization while preserving topology where needed.

- Use it when:
  - You need per-geometry algorithms or to build custom spatial logic beyond high-level GeoPandas ops.

Library Guide: Fiona (1/2)

- What it is: Pythonic wrapper for GDAL/OGR focused on vector file I/O.
- Why it matters: Fine-grained control over layers, schemas, drivers, and streaming access.
- Core features:
  - Open/read/write specific layers, inspect schemas/CRS, iterate features.
  - Supports many drivers (GeoPackage, Shapefile, etc.).
- Best practices:
  - Use context managers (`with fiona.open(...)`) to handle resources; specify `driver` and `layer` explicitly.
  - Prefer modern formats (GeoPackage) for attributes/CRS fidelity; avoid Shapefile unless required.
- Use it when:
  - You need lower-level control than GeoPandas or to interface with custom driver options.

Library Guide: Fiona (2/2)

- Streaming/large files:
  - Iterate features to avoid loading entire layers into memory.
  - Validate feature schemas and handle encoding explicitly for legacy datasets.

Vector Formats & Trade-offs

- **Common Formats & Trade-offs**:
  - **GeoPackage (`.gpkg`)**: **Best choice**. An open standard, single-file database. Fast, flexible, and robust.
  - **Shapefile (`.shp`)**: Legacy format. Avoid if possible. Multi-file, column name limits, and other issues.
  - **GeoJSON (`.geojson`)**: Good for web applications, text-based, and easy to read. Not efficient for large datasets.

GeoPackage (`.gpkg`)

- Single file, supports multiple layers and spatial indexes. Great default choice.
- Stores CRS and attributes robustly; plays well with `geopandas` and QGIS.

```
import os, geopandas as gpd

os.makedirs("outputs", exist_ok=True)
# Write to a new GeoPackage layer
gdf = gpd.read_file("data/texas/texas.geojson")
gdf.to_file("outputs/data.gpkg", layer="texas", driver="GPKG")

# Read a specific layer back
gpkg = gpd.read_file("outputs/data.gpkg", layer="texas")
print(len(gpkg), gpkg.crs)
```

Shapefile (`.shp`)

- Legacy format with constraints: short field names, multi-file sidecars, encodings.
- Still ubiquitous; treat as an interchange format, not your system of record.

```
import os, geopandas as gpd

os.makedirs("outputs", exist_ok=True)
gdf = gpd.read_file("data/mexico/mexico.geojson")
gdf.to_file("outputs/mexico.shp")  # beware 10-char field truncation

shp = gpd.read_file("outputs/mexico.shp")
print(shp.columns[:5])
```

Validate and fix geometries

```
import geopandas as gpd

gdf = gpd.read_file("data/berlin-neighbourhoods/berlin-neighbourhoods.geojson")
invalid = ~gdf.is_valid
gdf.loc[invalid, "geometry"] = gdf.loc[invalid, "geometry"].buffer(0)
print(f"Fixed {invalid.sum()} invalid geometries")
```

Spatial index & performance

- Use `gdf.sindex` (rtree/pygeos) to prefilter by bounding box.
- Avoid O(N×M) spatial joins by pre-clipping candidates.
- Read only needed columns; write GeoPackage for multi-layer outputs.

Aggregate and dissolve

```
import geopandas as gpd

gdf = gpd.read_file("data/mexico/mexico.geojson")
out = gdf.to_crs("EPSG:6933")  # world cylindrical equal-area
candidates = ["STATE_NAME", "NAME", "state", "ADM1_NAME"]
target = next((c for c in candidates if c in out.columns), None)
if target is None:
  print("No suitable column found to dissolve by; columns:", list(out.columns))
else:
  by_state = out.dissolve(by=target, aggfunc="sum")
  by_state["area_km2"] = by_state.geometry.area / 1_000_000
  print(by_state[["area_km2"]].head())
```

Vector Recipe: Reading, Clipping, and Joining

```
import geopandas as gpd

# Read data
countries = gpd.read_file(gpd.datasets.get_path('naturalearth_lowres'))
cities = gpd.read_file(gpd.datasets.get_path('naturalearth_cities'))

# Clip: Select only cities within the bounding box of Africa
africa = countries[countries['continent'] == 'Africa']
bbox = africa.total_bounds  # [minx, miny, maxx, maxy]
clipped_cities = cities.cx[bbox[0]:bbox[2], bbox[1]:bbox[3]]

# Ensure matching CRS before spatial join
clipped_cities = clipped_cities.to_crs(countries.crs)

# Spatial Join: Find which country each city belongs to
cities_with_country = gpd.sjoin(clipped_cities, countries, how="inner", predicate="within")

print(cities_with_country[['name', 'continent']].head())
```

Vector Data: Further Learning

- **GeoPandas Getting Started**: The official entry point.
  - https://geopandas.org/en/stable/getting_started.html
- **Plotting with GeoPandas**: Create beautiful maps.
  - https://geopandas.org/en/stable/docs/user_guide/mapping.html
- **Tutorial on Spatial Joins**: A fundamental concept explained well.
  - https://geopandas.org/en/stable/gallery/spatial_joins.html
- **Automating GIS Processes**: A great course with many vector data examples.
  - https://autogis-site.readthedocs.io/en/latest/

4. Raster Data

Working with Grids and Pixels

Raster data is a matrix of cells representing values over a continuous surface.

- **Key Properties**:
  - **Resolution**: The size of each pixel (e.g., 30 meters).
  - **Extent**: The bounding box of the entire grid.
  - **Bands**: A raster can have one or more layers (e.g., Red, Green, Blue bands in a color image).
  - `nodata` **value**: A special value assigned to pixels with no data.

- **Core Python Libraries**:
  - `rasterio`: The primary library for reading, writing, and manipulating raster files.
  - `rioxarray`: Integrates `rasterio` with `xarray` for powerful, labeled multi-dimensional array analysis (great for time-series!).

Raster formats in practice

- GeoTIFF: general-purpose workhorse; supports overviews and compression.
- COG (Cloud Optimized GeoTIFF): tiled + overviews for HTTP range requests.
- Bands & dtypes matter (uint8 imagery vs float32 surfaces); mind `nodata`.

Library Guide: Rasterio (1/2)

- What it is: Pythonic raster I/O and processing built on GDAL.
- Why it matters: Robust read/write, windowed access, metadata, CRS/transform handling, masks.
- Core features:
  - `rasterio.open` context manager; read/write windows, profiles, overviews.
  - Affine transforms, CRS, nodata handling; `mask`, `warp.reproject`.
- Best practices:
  - Always use `with rasterio.open(...) as src:`; prefer windowed reads for performance.
  - Match CRS and resolution intentionally; pick resampling by data type (nearest vs bilinear/cubic).
  - Respect `nodata`; convert to float and mask invalids for stats.
- Use it when:
  - Working with GeoTIFF/COG and need performant chunked access or detailed control over raster metadata.

Library Guide: Rasterio (2/2)

- Performance tips:
  - Build overviews and use block-aligned windows for faster reads.
  - Keep profiles consistent when writing derived rasters to avoid implicit conversions.

Library Guide: rioxarray (1/2)

- What it is: Xarray + Rasterio for labeled N-D arrays with geospatial awareness.
- Why it matters: Data-cube style analysis (bands/time) with coordinates and lazy computation (via Dask).
- Core features:
  - `.rio` accessor: `reproject`, `reproject_match`, `clip`, `to_raster`, CRS/transform attributes.
  - Works seamlessly with multi-band/time series; integrates with xarray/dask ecosystem.
- Best practices:
  - Keep grids aligned before arithmetic (`reproject_match`); be explicit about `nodata`.
  - Use Dask-backed arrays for large data and chunk wisely.
- Use it when:
  - You need labeled, multi-dimensional geospatial arrays or time-series raster analysis.

Library Guide: rioxarray (2/2)

- Tips:
  - Use `.rio.write_crs` and `.rio.write_transform` when constructing arrays manually.
  - Export COGs via `rio-cogeo` or write tiled/overviews with Rasterio for cloud workflows.

Rasterio: open and windowed read

```
import os
import rasterio
from rasterio.windows import Window

ras_path = "data/sandiego/sandiego_tracts.tif"
if not os.path.exists(ras_path):
  print("Raster not found:", ras_path)
else:
  with rasterio.open(ras_path) as src:
    window = Window(0, 0, 512, 512)
    arr = src.read(1, window=window)
    print(arr.shape, src.res)
```

Resampling and reprojection notes

- Choose resampling by data type:
  - Nearest: categorical labels
  - Bilinear/Cubic: continuous surfaces
- Reproject with `rioxarray.rio.reproject()` (see earlier slide) or Rasterio's `reproject` API.

Raster Reprojection with `rioxarray`

`rioxarray` simplifies raster reprojection by handling the complex warping calculations for you.

```
import os
import rioxarray

os.makedirs("outputs", exist_ok=True)
ras_path = "data/sandiego/sandiego_tracts.tif"
if not os.path.exists(ras_path):
  print("Raster not found:", ras_path)
else:
  rds = rioxarray.open_rasterio(ras_path)
  rds_3857 = rds.rio.reproject("EPSG:3857")
  print("Reprojected CRS:", rds_3857.rio.crs)
  rds_3857.rio.to_raster("outputs/sandiego_tracts_3857.tif")
```

Raster Recipe: Reading, Masking, and Zonal Stats

This recipe calculates the mean raster value within each polygon.

```
import os
import geopandas as gpd
import rasterio
from rasterio.mask import mask
import numpy as np

ras_path = "data/sandiego/sandiego_tracts.tif"
vec_path = "data/berlin-neighbourhoods/berlin-neighbourhoods.geojson"
if not (os.path.exists(ras_path) and os.path.exists(vec_path)):
  print("Missing input(s):", ras_path, os.path.exists(ras_path), "|", vec_path, os.path.exists(vec_path))
else:
  # Load a raster and vector polygons
  with rasterio.open(ras_path) as src:
    polygons = gpd.read_file(vec_path)

    # IMPORTANT: Ensure polygons are in the same CRS as the raster
    polygons = polygons.to_crs(src.crs)
```

Raster Recipe (continued)

```
  # Iterate through each polygon to perform the mask
  results = []
  for geom in polygons.geometry:
    # Mask the raster with the polygon geometry
    out_image, _ = mask(src, [geom], crop=True)

    # Calculate the mean of the valid (non-nodata) pixels safely
    arr = out_image.astype('float32')
    nodata = src.nodata
    if nodata is not None:
      arr[arr == nodata] = np.nan
    mean_val = np.nanmean(arr)
    results.append(float(mean_val) if np.isfinite(mean_val) else None)

polygons['mean_raster_value'] = results
print(polygons[['neighbourhood', 'mean_raster_value']].head())
```

Raster Data: Further Learning

- **Rasterio Documentation & Cookbook**: Essential reading.
  - https://rasterio.readthedocs.io/en/latest/
- **rioxarray Usage Guide**: For when you need more advanced analysis.
  - https://corteva.github.io/rioxarray/stable/
- **Rasterio Quickstart Guide**: A great starting point for raster processing.
  - https://rasterio.readthedocs.io/en/latest/quickstart.html
- **Cloud-Optimized GeoTIFFs (COGs)**: The future of raster data access.
  - https://www.cogeo.org/

5. Network Data

Working with Nodes, Edges, and Connectivity

Spatial networks are graphs where nodes and/or edges have geographic locations. They are fundamental for routing, accessibility analysis, and logistics.

- **Use Cases**:
  - Calculating the shortest path between two points.
  - Finding all locations reachable within a 15-minute drive (isochrones).
  - Identifying the most critical intersections in a city.

- **Core Python Libraries**:
  - `OSMnx` : The best tool for downloading, analyzing, and visualizing street networks from OpenStreetMap.
  - `NetworkX` : A general-purpose library for graph analysis (centrality, pathfinding, etc.). `OSMnx` uses it under the hood.

Library Guide: OSMnx (1/2)

- What it is: Download, construct, analyze, and visualize street networks from OpenStreetMap using NetworkX.
- Why it matters: One-stop solution for OSM retrieval, graph building, and common transport analyses.
- Core features:
  - `graph_from_place` / `graph_from_polygon`, `plot_graph`, `basic_stats`, `project_graph`.
  - Geocoding, nearest node/edge queries, save/load (GraphML/GeoPackage).

Library Guide: OSMnx (2/2)

- Tips:
  - Persist graphs as GraphML for reuse; consider simplifying/snap-tolerance settings for clean topology.
  - Validate coverage with small plots and `basic_stats` before heavy analyses.

- Best practices:
  - Choose the right `network_type` (drive/walk/bike/all) and project graphs before measuring lengths.
  - Add speeds and compute `travel_time` before routing; cache results to avoid repeated downloads.
  - Be mindful of rate limits and OSM data quality/coverage.

- Use it when:
  - You need OSM-based street networks for routing, accessibility, or urban form metrics.

OSMnx essentials

- Download graphs from OSM, filter by mode (drive/walk/bike), plot in 1 line.
- Compute stats (`basic_stats`), simplify topology, save/load graphs.

```
import osmnx as ox
G = ox.graph_from_place("Kreuzberg, Berlin, Germany", network_type="walk")
ox.plot_graph(G, node_size=0, edge_color="gray")
```

Library Guide: NetworkX (1/2)

- What it is: General-purpose graph theory library for Python.
- Why it matters: Broad suite of algorithms (shortest paths, centrality, connectivity) with flexible graph models.
- Core features:
  - Graph types (Graph/DiGraph/MultiDiGraph), node/edge attributes, generators.
  - Algorithms for paths, flows, clustering, centrality, and more.
- Best practices:
  - Pick the right graph type (e.g., `MultiDiGraph` for street networks with parallel edges/directions).
  - Use generators/iterators for large analyses; consider specialized libraries for massive graphs.
  - Store weights (e.g., length/travel_time) consistently and document units.
- Use it when:
  - You need algorithmic graph analysis independent of data source (beyond OSMnx).

Library Guide: NetworkX (2/2)

- Tips:
  - Use `nx.shortest_path_length(..., weight="travel_time")` to verify weights.
  - For performance-sensitive tasks, look at `igraph`, `graph-tool`, or networkx's algorithms with heuristics (A*).

NetworkX essentials

```
import networkx as nx
G = nx.path_graph(5)
print(nx.shortest_path(G, 0, 4))  # [0,1,2,3,4]
```

Project your graph for metrics

```
import osmnx as ox
G = ox.graph_from_place("Kreuzberg, Berlin, Germany", network_type="drive")
Gp = ox.project_graph(G)  # meters-based CRS for accurate lengths
u, v, k, data = list(Gp.edges(keys=True, data=True))[0]
print(data.get("length"))
```

Routing with travel time

```
import osmnx as ox
G = ox.graph_from_place("Kreuzberg, Berlin, Germany", network_type="drive")
Gp = ox.project_graph(G)

# Assume default speed (km/h) if not present; compute travel_time (seconds)
for _, _, _, d in Gp.edges(keys=True, data=True):
  speed_kmh = d.get("speed_kph", 30)
  length_m = d.get("length", 0)
  d["travel_time"] = length_m / (speed_kmh * (1000/3600))

orig = ox.geocode("Görlitzer Bahnhof, Berlin")
dest = ox.geocode("Kottbusser Tor, Berlin")
orig_n = ox.nearest_nodes(Gp, orig[1], orig[0])
dest_n = ox.nearest_nodes(Gp, dest[1], dest[0])
route = ox.shortest_path(Gp, orig_n, dest_n, weight="travel_time")
```

OSM tags and filters (1/2)

- Use `custom_filter` in `graph_from_place` / `graph_from_polygon` to restrict tags.
- Example: only primary/secondary roads; exclude private/service roads.

OSM tags and filters (2/2)

- Align network type with your analysis (walk vs bike vs drive).
- Validate assumptions by visual inspection and simple stats.

Network Recipe: Get a Street Network with OSMnx

This example downloads the street network for a neighborhood, calculates basic stats, and plots it.

```
import osmnx as ox

# 1. Define a place and download the street network
place_name = "Kreuzberg, Berlin, Germany"
# You can specify network_type as 'drive', 'walk', 'bike', 'all'
G = ox.graph_from_place(place_name, network_type='drive')

# 2. Plot the network
fig, ax = ox.plot_graph(G, node_size=0, edge_linewidth=0.5, edge_color='gray')

# 3. Calculate basic statistics
stats = ox.basic_stats(G)
print(f"Total street length: {stats['street_length_total'] / 1000:.2f} km")
```

Network Recipe (continued)

```
# 4. Find the shortest path (example)
origin = ox.geocode("Görlitzer Bahnhof, Berlin")
destination = ox.geocode("Kottbusser Tor, Berlin")
origin_node = ox.nearest_nodes(G, origin[1], origin[0])
dest_node = ox.nearest_nodes(G, destination[1], destination[0])

route = ox.shortest_path(G, origin_node, dest_node, weight='length')
fig, ax = ox.plot_graph_route(G, route, route_linewidth=6, node_size=0, bgcolor='k')
```

Network Data: Further Learning

- **OSMnx Examples Gallery**: A fantastic collection of tutorials and advanced use cases.
  - https://osmnx.readthedocs.io/en/stable/user-reference/examples.html
- **NetworkX Tutorial**: Learn the fundamentals of graph theory and analysis.
  - https://networkx.org/documentation/stable/tutorial.html
- **Python GIS: Network Analysis**: A practical book combining theory and code.
  - https://pythongis.org/part-iv-network-analysis/

Setup: Python environment

What you'll set up

- A clean Python install (macOS)
- A project-local virtual environment (venv)
- Packages for this deck (GeoPandas, Rasterio, rioxarray, OSMnx, etc.)
- Jupyter for running notebooks
- VS Code as an editor (with Python + Jupyter extensions)

Base requirements (this repo)

These are the core packages in `notebooks/requirements.txt` used across the vector/raster parts of the deck:

```
geopandas
rasterio
rioxarray
xarray
pyproj
pandas
```

Extras (networks, notebooks, utilities) are listed in `notebooks/requirements-extras.txt` and include:

- osmnx, networkx
- jupyter, jupyterlab, ipykernel
- rtree, rasterstats, contextily, matplotlib
- laspy, open3d (optional point clouds)

Install Python on macOS

- Option A (easiest): Install from python.org
  - Download the latest macOS installer and follow prompts
  - After install, open Terminal and check:

```
python3 --version
pip3 --version
```

- Option B (Homebrew):

```
# Install Homebrew if you don't have it (https://brew.sh)
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"

# Then install Python
brew install python
python3 --version
```

Create and activate a virtual environment

```
# Create a project folder (example) and enter it
mkdir -p ~/dev/spatial-work && cd ~/dev/spatial-work

# Create a virtual environment named .venv
python3 -m venv .venv

# Activate it (zsh)
source .venv/bin/activate

# Upgrade pip inside the venv
python -m pip install --upgrade pip
```

You should now see `(.venv)` in your terminal prompt. Use `deactivate` to exit.

Install required packages

- If you're in this repo, install the base packages from the provided requirements file:

```
# From the repo root
source .venv/bin/activate
python -m pip install --upgrade pip setuptools wheel
pip install -r notebooks/requirements.txt
```

- Then add the network + notebook tooling used in this deck:

```
pip install -r notebooks/requirements-extras.txt
```

- Alternatively, install everything explicitly (useful outside this repo):

```
pip install \
  geopandas shapely fiona rasterio rioxarray xarray matplotlib pyproj \
  rasterstats contextily osmnx networkx jupyter jupyterlab ipykernel rtree
```

If you hit build errors, first try: `python -m pip install --upgrade pip setuptools wheel`.

Verify the environment

```
import sys, importlib
pkgs = [
  ("geopandas", "gpd"),
  ("shapely", None),
  ("fiona", None),
  ("rasterio", None),
  ("rioxarray", None),
  ("osmnx", None),
  ("networkx", "nx"),
]
print("Python:", sys.version.split()[0])
for name, alias in pkgs:
    m = importlib.import_module(name)
    ver = getattr(m, "__version__", None)
    print(f"{name}: {ver}")
```

You should see version numbers printed without errors for each package.

Work with Jupyter Notebooks

```
# Start Jupyter Lab (recommended)
jupyter lab

# Or classic Notebook
jupyter notebook
```

- In the UI, ensure the kernel is your venv (Python in `.venv`).
- If the venv kernel isn't listed, install it and retry:

```
pip install ipykernel
python -m ipykernel install --user --name spatial-env --display-name "Python (spatial-env)"
```

Then reopen the notebook and choose the new kernel.

Use VS Code as your editor

- Install VS Code and the extensions:
  - “Python” (Microsoft)
  - “Jupyter” (Microsoft)
- Open the project folder (the repo root)
- Select interpreter: Command Palette → “Python: Select Interpreter” → choose `.venv`
- For notebooks: pick the same kernel in the top-right kernel picker

Optional: Install the `code` command for launching VS Code from Terminal:

```
# VS Code → Command Palette → "Shell Command: Install 'code' command in PATH"
code .
```

Troubleshooting (macOS) 1/2

- Pip build errors for Rasterio/Fiona
  - Ensure Xcode Command Line Tools: `xcode-select --install`
  - Update build tools: `python -m pip install --upgrade pip setuptools wheel`
  - If needed, install libs: `brew install geos proj gdal` then retry `pip install`
  - If `rtree` fails, install spatial index libs: `brew install spatialindex` then `pip install rtree`
- Apple Silicon (M1/M2/M3)
  - Prefer Python from python.org or Homebrew (arm64). Use arm64 Terminal (not Rosetta) for consistency.
  - Wheels now exist for most geospatial packages; if a wheel isn't found, ensure Homebrew's `geos`, `proj`, and `gdal` are installed first.

Troubleshooting (macOS) 2/2

- Jupyter kernel not showing
  - Install the kernel: `python -m ipykernel install --user --name spatial-env --display-name "Python (spatial-env)"`
- Permission or PATH issues
  - Close/reopen Terminal after installs; confirm `python3 --version` and that `(.venv)` is active

Summary & Next Steps

- **CRS is everything**. Always check and correctly manage your coordinate systems.
- **Choose the right data model**: Vector for discrete features, Raster for continuous surfaces.
- **Master the core libraries**: `geopandas` and `rasterio`/`rioxarray` will handle 90% of your tasks.
- **Practice with real data**: Download data for your city or a region of interest and try to replicate these recipes.

Recommended Hands-on Path

1. Complete the **GeoPandas** "Getting Started" guide.
2. Work through the **rasterio** "Cookbook" examples.
3. Use **OSMnx** to analyze the street network of your own neighborhood.