Home

Contents

GDS4AE - *Geographic Data Science for Applied Economists*

- Dani Arribas-Bel `[@darribas]`
- Diego Puga `[@ProfDiegoPuga]`

> **Note**
>
> A PDF version of this course is available for download **here**

Contact

Dani Arribas-Bel - `D.Arribas-Bel [at] liverpool.ac.uk`
Senior Lecturer in Geographic Data Science
Office 508, Roxby Building,
University of Liverpool - 74 Bedford St S,
Liverpool, L69 7ZT,
United Kingdom.

Diego Puga - diego.puga [at] cemfi.es
Professor
CEMFI,
Casado del Alisal 5,
28014 Madrid,
Spain.

Citation

If you use materials from this resource in your own work, we recommend the following citation:

```
@article{darribas_gds_course,
  author = {Dani Arribas-Bel and Diego Puga},
  title = {Geographic Data Science for Applied Economists},
  year = 2022,
  annote = {\href{https://darribas.org/gds4ae}}
}
```

Overview

This resource provides an introduction to Geographic Data Science for applied economists using Python. It has been designed to be delivered within 15 hours of teaching, split into ten sessions of 1.5h each.

How to follow along

GDS4AE is best followed if you can interactively tinker with its content. To do that, you will need two things:

1. A computer set up with the Jupyter Lab environment and all the required libraries (please see the Software stack part in the Infrastructure section for instructions)
2. A local copy of the materials that you can run on your own computer (see the repository section in the Infrastructure section for instructions)

Blocks have different components:

- 📖 *Ahead of time…*: materials to go on your own ahead of the live session
- 💻 *Hands-on coding*: content for the live session
- 🐾 *Next steps*: a few pointers to continue your journey on the area the block covers

Content

The structure of content is divided in nine blocks:

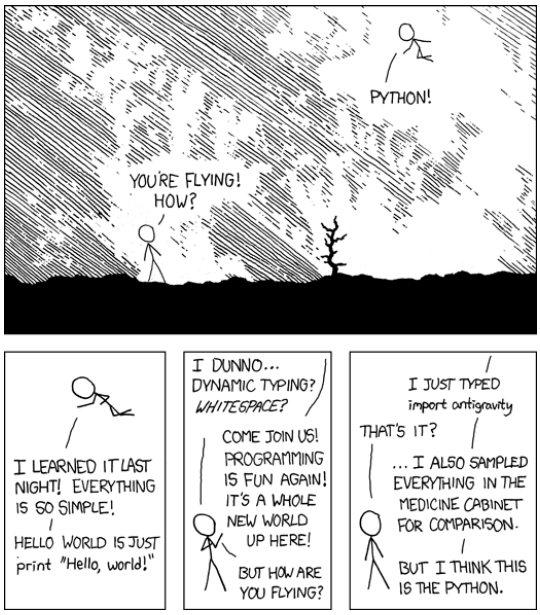
- *Introduction*: get familiar with the computational envirionment of modern data science
- *Spatial Data*: what do spatial data look like in Python?
- *Geovisualisation*: make (good) data maps
- *Spatial Feature Engineering* (Part I and Part II): augment and massage your data using Geography before you feed them into your model
- *OpenStreetMap*: acquire data from the largest geo-table in the world
- *Spatial Networks*: understand and work with spatial graphs
- *Transport Costs*: “getting there” doesn’t always cost the same

Each block has its own section and is designed to be delivered in 1.5 hours approximately. The content of some of these blocks relies on external resources, all of them freely available. When that is the case, enough detail is provided in the to understand how additional material fits in.

Why Python?

There are several reasons why we have made this choice. Many of them are summarised nicely in this article by The Economist (paywalled).:w

Source: XKCD



Data

All the datasets used in this resource is freely available. Some of them have been developed in the context of the resource, others are borrowed from other resources. A full list of the datasets used, together with links to the original source, or to reproducible code to generate the data used is available in the Datasets page.

License

Infrastructure

This page covers a few technical aspects on how the course is built, kept up to date, and how you can create a computational environment to run all the code it includes.

Software stack

This course is best followed if you can not only read its content but also interact with its code and even branch out to write your own code and play on your own. For that, you will need to have installed on your computer a series of interconnected software packages; this is what we call a *stack*.

Instructions on how to install a software stack that allows you to run the materials of this course depend on the operating system you are using. Detailed guides are available for the main systems on the following resource, provided by the Geographic Data Science Lab:

Github repository

All the materials for this course and this website are available on the following Github repository:

If you are interested, you can download a compressed `.zip` file with the most up-to-date version of all the materials, including the HTML for this website at:

Icon made by Freepik from www.flaticon.com

Containerised backend

The course is developed, built and tested using the **`gds_env`**, a containerised platform for Geographic Data Science. You can read more about the **`gds_env`** project at:

Binder

Binder is service that allows you to run scientific projects in the cloud for free. Binder can spin up *"ephemeral"* instances that allow you to run code on the browser without any local setup. It is possible to run the course on Binder by clicking on the button below:

> **Warning**
>
> It is important to note Binder instances are *ephemeral* in the sense that the data and content created in a session is **NOT** saved anywhere and is deleted as soon as the browser tab is closed.

Binder is also the backend this website relies on when you click on the rocket icon (🚀) on a page with code. Remember, you can play with the code interactively but, once you close the tab, all the changes are lost.

Introduction

Geographic Data Science

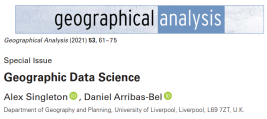
> **Note**
>
> This section is adapted from Block A of the GDS Course [AB19].

Before we learn *how* to do Geographic Data Science or even *why* you would want to do it, let's start with *what* it is. We will rely on two resources:

- First, in this video, Dani Arribas-Bel covers the building blocks at the First Spatial Data Science Conference, organised by CARTO

- Second, *Geographic Data Science*, by Alex Singleton and Dani Arribas-Bel [SAB19]

URL



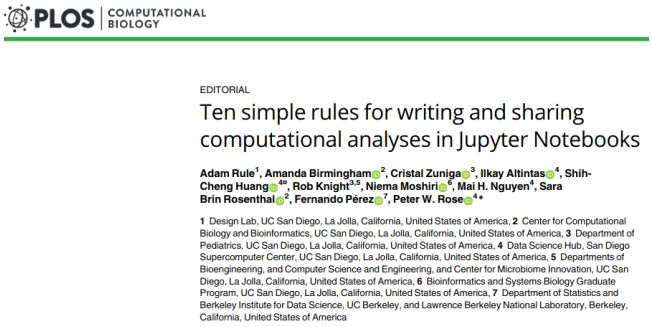
The computational stack

One of the core learning outcomes of this course is to get familiar with the modern computational environment that is used across industry and science to “do” Data Science. In this section, we will learn about ecosystem of concepts and tools that come together to provide the building blocks of much computational work in data science these days.

Source: The Atlantic

- *Ten simple rules for writing and sharing computational analyses in Jupyter Notebooks*, by Adam Rule et al. [RBZ+19]

URL



PLOS COMPUTATIONAL BIOLOGY

EDITORIAL

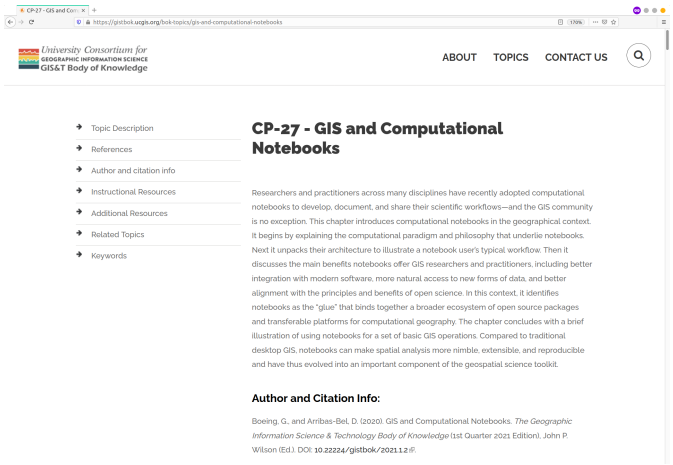
Ten simple rules for writing and sharing computational analyses in Jupyter Notebooks

Adam Rule[1], Amanda Birmingham[2], Cristal Zuniga[3], Ilkay Altintas[4], Shih-Cheng Huang[4¤], Rob Knight[3,5], Niema Moshiri[6], Mai H. Nguyen[4], Sara Brin Rosenthal[2], Fernando Pérez[7], Peter W. Rose[4*]

1 Design Lab, UC San Diego, La Jolla, California, United States of America, **2** Center for Computational Biology and Bioinformatics, UC San Diego, La Jolla, California, United States of America, **3** Department of Pediatrics, UC San Diego, La Jolla, California, United States of America, **4** Data Science Hub, San Diego Supercomputer Center, UC San Diego, La Jolla, California, United States of America, **5** Departments of Bioengineering, and Computer Science and Engineering, and Center for Microbiome Innovation, UC San Diego, La Jolla, California, United States of America, **6** Bioinformatics and Systems Biology Graduate Program, UC San Diego, La Jolla, California, United States of America, **7** Department of Statistics and Berkeley Institute for Data Science, UC Berkeley, and Lawrence Berkeley National Laboratory, Berkeley, California, United States of America

- *GIS and Computational Notebooks*, by Geoff Boeing and Dani Arribas-Bel [BAB20]

URL



University Consortium for Geographic Information Science
GIS&T Body of Knowledge

ABOUT TOPICS CONTACT US

- Topic Description
- References
- Author and citation info
- Instructional Resources
- Additional Resources
- Related Topics
- Keywords

CP-27 - GIS and Computational Notebooks

Researchers and practitioners across many disciplines have recently adopted computational notebooks to develop, document, and share their scientific workflows—and the GIS community is no exception. This chapter introduces computational notebooks in the geographical context. It begins by explaining the computational paradigm and philosophy that underlie notebooks. Next it unpacks their architecture to illustrate a notebook user's typical workflow. Then it discusses the main benefits notebooks offer GIS researchers and practitioners, including better integration with modern software, more natural access to new forms of data, and better alignment with the principles and benefits of open science. In this context, it identifies notebooks as the "glue" that binds together a broader ecosystem of open source packages and transferable platforms for computational geography. The chapter concludes with a brief illustration of using notebooks for a set of basic GIS operations. Compared to traditional desktop GIS, notebooks can make spatial analysis more nimble, extensible, and reproducible and have thus evolved into an important component of the geospatial science toolkit.

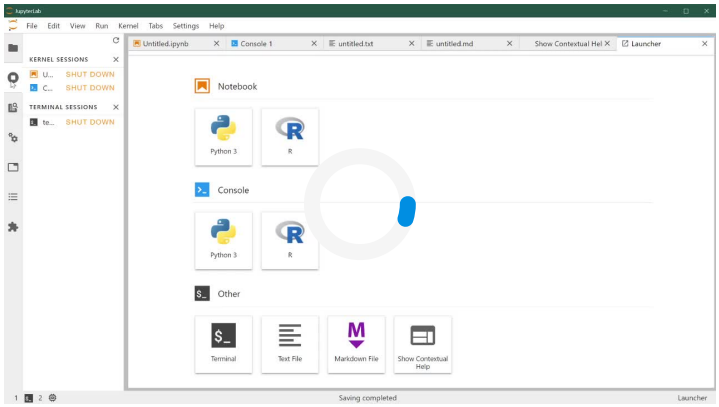
Author and Citation Info:

Boeing, G., and Arribas-Bel, D. (2020). GIS and Computational Notebooks. *The Geographic Information Science & Technology Body of Knowledge* (1st Quarter 2021 Edition). John P. Wilson (Ed.). DOI: 10.22224/gistbok/2021.1.2

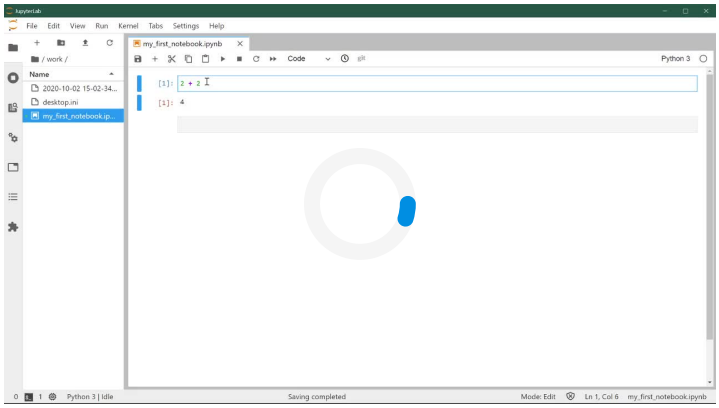
Now we are familiar with the conceptual pillars on top of which we will be working, let's switch gears into a more practical perspective. The following two clips cover the basics of Jupyter Lab, the frontend that glues all the pieces together, and Jupyter Notebooks, the file format, application, and protocol that allows us to record, store and share workflows.

> **Note**
>
> The clips are sourced from Block A of the GDS Course [AB19]

Jupyter Lab



Jupyter Notebooks



Spatial Data

📖 Ahead of time…

This block is all about understanding spatial data, both conceptually and practically. Before your fingers get on the keyboard, the following readings will help you get going and familiar with core ideas:

- Chapter 1 of the GDS Book [RABWng], which provides a conceptual overview of representing Geography in data
- Chapter 3 of the GDS Book [RABWng], a sister chapter with a more applied perspective on how concepts are implemented in computer data structures

Additionally, parts of this block are based and source from Block C in the GDS Course [AB19].

💻 Hands-on coding

(Geographic) tables

```
import pandas
import geopandas
import xarray, rioxarray
import contextily
import matplotlib.pyplot as plt
```

Points

Data
If you want to read more about the data sources behind this dataset, head to the Datasets section

Local files | Online read

Assuming you have the file locally on the path `../data/`:

```
pts = geopandas.read_file("../data/madrid_abb.gpkg")
```

Point geometries from columns

```
pts.info()
```

```
<class 'geopandas.geodataframe.GeoDataFrame'>
RangeIndex: 18399 entries, 0 to 18398
Data columns (total 16 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   price            18399 non-null  object
 1   price_usd        18399 non-null  float64
 2   log1p_price_usd  18399 non-null  float64
 3   accommodates     18399 non-null  int64
 4   bathrooms        18399 non-null  object
 5   bedrooms         18399 non-null  float64
 6   beds             18399 non-null  float64
 7   neighbourhood    18399 non-null  object
 8   room_type        18399 non-null  object
 9   property_type    18399 non-null  object
 10  WiFi             18399 non-null  object
 11  Coffee           18399 non-null  object
 12  Gym              18399 non-null  object
 13  Parking          18399 non-null  object
 14  km_to_retiro     18399 non-null  float64
 15  geometry         18399 non-null  geometry
dtypes: float64(5), geometry(1), int64(1), object(9)
memory usage: 2.2+ MB
```

```
pts.head()
```

	price	price_usd	log1p_price_usd	accommodates	bathrooms	bedroon
0	$60.00	60.0	4.110874	2	1 shared bath	1
1	$31.00	31.0	3.465736	1	1 bath	1
2	$60.00	60.0	4.110874	6	2 baths	3
3	$115.00	115.0	4.753590	4	1.5 baths	2
4	$26.00	26.0	3.295837	1	1 private bath	1

Challenge

Show the top ten values of of `price` and `neighbourhood`

Lines

Local files | Online read

Assuming you have the file locally on the path `../data/`:

```
pts = geopandas.read_file("../data/arturo_streets.gpkg")
```

```
lines.info()
```

```
<class 'geopandas.geodataframe.GeoDataFrame'>
RangeIndex: 66499 entries, 0 to 66498
Data columns (total 9 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   OGC_FID             66499 non-null  object
 1   dm_id               66499 non-null  object
 2   dist_barri          66483 non-null  object
 3   average_quality     66499 non-null  float64
 4   population_density  66499 non-null  float64
 5   X                   66499 non-null  float64
 6   Y                   66499 non-null  float64
 7   value               5465 non-null   float64
 8   geometry            66499 non-null  geometry
dtypes: float64(5), geometry(1), object(3)
memory usage: 4.6+ MB
```

```
lines.loc[0, "geometry"]
```

Challenge

Print descriptive statistics for `population_density` and `average_quality`

Polygons

Local files | Online read

Assuming you have the file locally on the path `../data/`:

```
polys = geopandas.read_file("../data/neighbourhoods.geojson")
```

```
polys.head()
```

	neighbourhood	neighbourhood_group	geometry
0	Palacio	Centro	MULTIPOLYGON (((-3.70584 40.42030, -3.70625 40...
1	Embajadores	Centro	MULTIPOLYGON (((-3.70384 40.41432, -3.70277 40...
2	Cortes	Centro	MULTIPOLYGON (((-3.69796 40.41929, -3.69645 40...
3	Justicia	Centro	MULTIPOLYGON (((-3.69546 40.41898, -3.69645 40...
4	Universidad	Centro	MULTIPOLYGON (((-3.70107 40.42134, -3.70155 40...

```
polys.query("neighbourhood_group == 'Retiro'")
```

	neighbourhood	neighbourhood_group	geometry
13	Pacífico	Retiro	MULTIPOLYGON (((-3.67015 40.40654, -3.67017 40...
14	Adelfas	Retiro	MULTIPOLYGON (((-3.67283 40.39468, -3.67343 40...
15	Estrella	Retiro	MULTIPOLYGON (((-3.66506 40.40647, -3.66512 40...
16	Ibiza	Retiro	MULTIPOLYGON (((-3.66916 40.41796, -3.66927 40...
17	Jerónimos	Retiro	MULTIPOLYGON (((-3.67874 40.40751, -3.67992 40...
18	Niño Jesús	Retiro	MULTIPOLYGON (((-3.66994 40.40850, -3.67012 40...

```
polys.neighbourhood_group.unique()
```

```
array(['Centro', 'Arganzuela', 'Retiro', 'Salamanca', 'Chamartín',
       'Moratalaz', 'Tetuán', 'Chamberí', 'Fuencarral - El Pardo',
       'Moncloa - Aravaca', 'Puente de Vallecas', 'Latina', 'Carabanchel',
       'Usera', 'Ciudad Lineal', 'Hortaleza', 'Villaverde',
       'Villa de Vallecas', 'Vicálvaro', 'San Blas - Canillejas',
       'Barajas'], dtype=object)
```

Challenge

Print the neighborhoods within the “Latina” group

Surfaces

Local files Online read

Assuming you have the file locally on the path `../data/`:

```
sat = xarray.open_rasterio("../data/madrid_scene_s2_10_tc.tif")
```

```
sat
```

xarray.DataArray (**band**: 3, **y**: 3681, **x**: 3129)

[34553547 values with dtype=uint8]

▾ Coordinates:

band	(band)	int64	1 2 3	
x	(x)	float64	4.248e+05 4.248e+05 ... 4.56e+05	
y	(y)	float64	4.499e+06 4.499e+06 ... 4.463e+06	
spatial_ref	()	int64	0	

▾ Attributes:

scale_factor :	1.0
add_offset :	0.0

```
sat.sel(band=1)
```

xarray.DataArray (**y**: 3681, **x**: 3129)

[11517849 values with dtype=uint8]

▾ Coordinates:

band	()	int64	1	
x	(x)	float64	4.248e+05 4.248e+05 ... 4.56e+05	
y	(y)	float64	4.499e+06 4.499e+06 ... 4.463e+06	
spatial_ref	()	int64	0	

▾ Attributes:

scale_factor :	1.0
add_offset :	0.0

```
sat.sel(
    x=slice(430000, 440000),  # x is ascending
    y=slice(4480000, 4470000) # y is descending
)
```

xarray.DataArray (**band**: 3, **y**: 1000, **x**: 1000)

[3000000 values with dtype=uint8]

▾ Coordinates:

band	(band)	int64	1 2 3	
x	(x)	float64	4.3e+05 4.3e+05 ... 4.4e+05 4.4e+05	
y	(y)	float64	4.48e+06 4.48e+06 ... 4.47e+06	
spatial_ref	()	int64	0	

▾ Attributes:

scale_factor :	1.0
add_offset :	0.0

Challenge

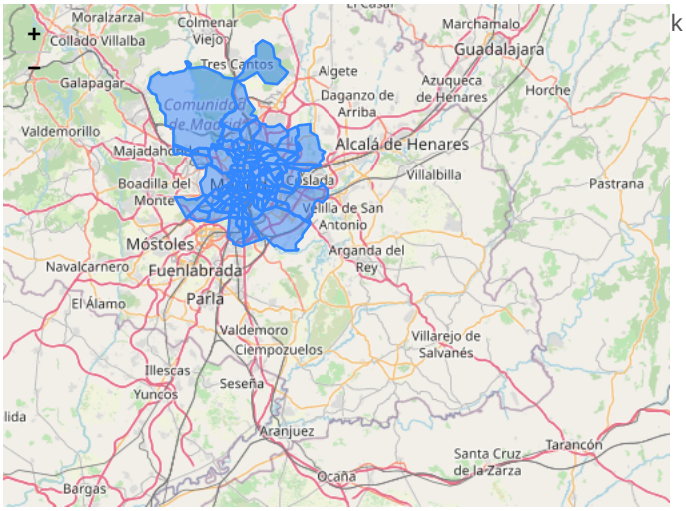
Subset `sat` to band 2 and the section within [444444, 455555] of Easting and [4470000, 4480000] of Northing.

- *How many pixels does it contain?*
- *What if you used bands 1 and 3 instead?*

Visualisation

IMPORTANT
You will need version 0.10.0 or greater of `geopandas` to use `explore`.

```
polys.explore()
```



k

```
polys.plot()
```

```
<AxesSubplot:>
```

_build/jupyter_execute/content/pages/02-Spatial_data_36_1.png

```
ax = lines.plot(linewidth=0.1, color="black")
contextily.add_basemap(ax, crs=lines.crs)
```

_build/jupyter_execute/content/pages/02-Spatial_data_37_0.png

See more basemap options here.

```
ax = pts.plot(color="red", figsize=(12, 12), markersize=0.1)
contextily.add_basemap(
    ax,
    crs = pts.crs,
    source = contextily.providers.CartoDB.DarkMatter
);
```

_build/jupyter_execute/content/pages/02-Spatial_data_39_0.png

```
sat.plot.imshow(figsize=(12, 12))
```

```
<matplotlib.image.AxesImage at 0x7f21c0154100>
```

_build/jupyter_execute/content/pages/02-Spatial_data_40_1.png

IMPORTANT
You will need version 1.1.0 of `contextily` to use label layers. Install it with:

```
pip install \
    -U --no-deps \
    contextily
```

```
f, ax = plt.subplots(1, figsize=(12, 12))
sat.plot.imshow(ax=ax)
contextily.add_basemap(
    ax,
    crs=sat.rio.crs,
    source=contextily.providers.Stamen.TonerLabels,
    zoom=11
);
```

Challenge

Make three plots of `sat`, plotting one single band in each

Spatial operations

(Re-)Projections

```
pts.crs
```

```
<Geographic 2D CRS: EPSG:4326>
Name: WGS 84
Axis Info [ellipsoidal]:
- Lat[north]: Geodetic latitude (degree)
- Lon[east]: Geodetic longitude (degree)
Area of Use:
- name: World.
- bounds: (-180.0, -90.0, 180.0, 90.0)
Datum: World Geodetic System 1984 ensemble
- Ellipsoid: WGS 84
- Prime Meridian: Greenwich
```

```
sat.rio.crs
```

```
CRS.from_epsg(32630)
```

```
pts.to_crs(sat.rio.crs).crs
```

```
<Projected CRS: EPSG:32630>
Name: WGS 84 / UTM zone 30N
Axis Info [cartesian]:
- [east]: Easting (metre)
- [north]: Northing (metre)
Area of Use:
- undefined
Coordinate Operation:
- name: UTM zone 30N
- method: Transverse Mercator
Datum: World Geodetic System 1984
- Ellipsoid: WGS 84
- Prime Meridian: Greenwich
```

```
sat.rio.reproject(pts.crs).rio.crs
```

```
CRS.from_epsg(4326)
```

```
# All into Web Mercator (EPSG:3857)
f, ax = plt.subplots(1, figsize=(12, 12))
## Satellite image
sat.rio.reproject(
    "EPSG:3857"
).plot.imshow(
    ax=ax
)
## Neighbourhoods
polys.to_crs(epsg=3857).plot(
    linewidth=2,
    edgecolor="xkcd:lime",
    facecolor="none",
    ax=ax
)
## Labels
contextily.add_basemap( # No need to reproject
    ax,
    source=contextily.providers.Stamen.TonerLabels,
);
```

Centroids

Note the warning that geometric operations with non-project CRS object result in biases.

```
polys.centroid
```

```
/tmp/ipykernel_104/2101097851.py:1: UserWarning: Geometry is in a geographic CRS. Results
from 'centroid' are likely incorrect. Use 'GeoSeries.to_crs()' to re-project geometries
to a projected CRS before this operation.

  polys.centroid
```

```
0      POINT (-3.71398 40.41543)
1      POINT (-3.70237 40.40925)
2      POINT (-3.69674 40.41485)
3      POINT (-3.69657 40.42367)
4      POINT (-3.70698 40.42568)
                  ...
123    POINT (-3.59135 40.45656)
124    POINT (-3.59723 40.48441)
125    POINT (-3.55847 40.47613)
126    POINT (-3.57889 40.47471)
127    POINT (-3.60718 40.46415)
Length: 128, dtype: geometry
```

```
lines.centroid
```

```
0        POINT (444133.737 4482808.936)
1        POINT (444192.064 4482878.034)
2        POINT (444134.563 4482885.414)
3        POINT (445612.661 4479335.686)
4        POINT (445606.311 4479354.437)
                     ...
66494    POINT (451980.378 4478407.920)
66495    POINT (436975.438 4473143.749)
66496    POINT (442218.600 4478415.561)
66497    POINT (442213.869 4478346.700)
66498    POINT (442233.760 4478278.748)
Length: 66499, dtype: geometry
```

```
ax = polys.plot(color="purple")
polys.centroid.plot(
    ax=ax, color="lime", markersize=1
)
```

```
/tmp/ipykernel_104/1054587808.py:2: UserWarning: Geometry is in a geographic CRS. Results
from 'centroid' are likely incorrect. Use 'GeoSeries.to_crs()' to re-project geometries
to a projected CRS before this operation.

  polys.centroid.plot(
```

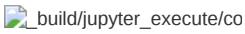
```
<AxesSubplot:>
```

Spatial joins

More information about spatial joins in `geopandas` is available on its documentation page

```
sj = geopandas.sjoin(
    lines,
    polys.to_crs(lines.crs)
)
```

```
sj.info()
```



```
<class 'geopandas.geodataframe.GeoDataFrame'>
Int64Index: 69420 entries, 0 to 66438
Data columns (total 12 columns):
 #   Column               Non-Null Count  Dtype
---  ------               --------------  -----
 0   OGC_FID              69420 non-null  object
 1   dm_id                69420 non-null  object
 2   dist_barri           69414 non-null  object
 3   average_quality      69420 non-null  float64
 4   population_density   69420 non-null  float64
 5   X                    69420 non-null  float64
 6   Y                    69420 non-null  float64
 7   value                5769 non-null   float64
 8   geometry             69420 non-null  geometry
 9   index_right          69420 non-null  int64
 10  neighbourhood        69420 non-null  object
 11  neighbourhood_group  69420 non-null  object
dtypes: float64(5), geometry(1), int64(1), object(5)
memory usage: 6.9+ MB
```

Areas

```
areas = polys.to_crs(
    epsg=25830
).area * 1e-6 # Km2
areas.head()
```

```
0    1.471037
1    1.033253
2    0.592049
3    0.742031
4    0.947616
dtype: float64
```

Distances

```
cemfi = geopandas.tools.geocode(
    "Calle Casado del Alisal, 5, Madrid"
).to_crs(epsg=25830)
cemfi
```

	geometry	address
0	POINT (441477.245 4473939.537)	5, Calle Casado del Alisal, 28014, Calle Casad...

```
polys.to_crs(
    cemfi.crs
).distance(
    cemfi.geometry
)
```

```
/opt/conda/lib/python3.9/site-packages/geopandas/base.py:31: UserWarning: The indices of
the two GeoSeries are different.
  warn("The indices of the two GeoSeries are different.")
```

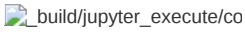
```
0      1491.338749
1              NaN
2              NaN
3              NaN
4              NaN
          ...
123            NaN
124            NaN
125            NaN
126            NaN
127            NaN
Length: 128, dtype: float64
```

```
d2cemfi = polys.to_crs(
    cemfi.crs
).distance(
    cemfi.geometry[0] # NO index
)
d2cemfi.head()
```

```
0    1491.338749
1     565.418135
2     278.121017
3     650.926572
4    1196.771601
dtype: float64
```

Challenge

Give Task III in this block of the GDS course a go



🐾 Next steps

If you are interested in following up on some of the topics explored in this block, the following pointers might be useful:

- Although we have seen here `geopandas` only, all non-geographic operations on geo-tables are really thanks to `pandas`, the workhorse for tabular data in Python. Their official documentation is an excellent first stop. If you prefer a book, McKinney (2012) [McK12] is a great one.
- For more detail on geographic operations on geo-tables, the Geopandas official documentation is a great place to continue the journey.
- Surfaces, as covered here, are really an example of multi-dimensional labelled arrays. The library we use, `xarray` represents the cutting edge for working with these data structures in Python, and their documentation is a great place to wrap your head around how data of this type can be manipulated. For geographic extensions (CRS handling, reprojections, etc.), we have used `rioxarray` under the hood, and its documentation is also well worth checking.

Geovisualisation

📖 Ahead of time…

This block is all about visualising statistical data on top of a geography. Although this task looks simple, there are a few technical and conceptual building blocks that it helps to understand before we try to make our own maps. Aim to complete the following readings by the time we get our hands on the keyboard:

- Block D of the GDS course [AB19], which provides an introduction to choropleths (statistical maps)
- Chapter 5 of the GDS Book [RABWng], discussing choropleths in more detail

💻 Hands-on coding

```
import geopandas
import xarray, rioxarray
import contextily
import seaborn as sns
from pysal.viz import mapclassify as mc
from legendgram import legendgram
import matplotlib.pyplot as plt
import palettable.matplotlib as palmpl
from splot.mapping import vba_choropleth
```

Data

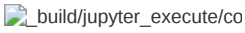
Data
If you want to read more about the data sources behind this dataset, head to the Datasets section

Local files | Online read

Assuming you have the file locally on the path `../data/`:

```
db = geopandas.read_file("../data/cambodia_regional.gpkg")
```

```
db.info()
```



```
<class 'geopandas.geodataframe.GeoDataFrame'>
RangeIndex: 198 entries, 0 to 197
Data columns (total 6 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   adm2_name   198 non-null    object
 1   adm2_altnm  122 non-null    object
 2   motor_mean  198 non-null    float64
 3   walk_mean   198 non-null    float64
 4   no2_mean    198 non-null    float64
 5   geometry    198 non-null    geometry
dtypes: float64(3), geometry(1), object(2)
memory usage: 9.4+ KB
```

We will use the average measurement of nitrogen dioxide (`no2_mean`) by region throughout the block.

To make visualisation a bit easier below, we create an additional column with values rescaled:

```
db["no2_viz"] = db["no2_mean"] * 1e5
```

This way, numbers are larger and will fit more easily on legends:

```
db[["no2_mean", "no2_viz"]].describe()
```

	no2_mean	no2_viz
count	198.000000	198.000000
mean	0.000032	3.236567
std	0.000017	1.743538
min	0.000014	1.377641
25%	0.000024	2.427438
50%	0.000029	2.922031
75%	0.000034	3.390426
max	0.000123	12.323324

Choropleths

A classiffication problem

```
db["no2_viz"].unique().shape
```

```
(198,)
```

```
sns.displot(
    db, x="no2_viz", kde=True, aspect=2
);
```

How to assign colors?

Attention

To build an intuition behind each classification algorithm more easily, we create a helper method (`plot_classi`) that generates a visualisation of a given classification.

Toggle the cell below if you are interested in the code behind it.

- Equal intervals

```
classi = mc.EqualInterval(db["no2_viz"], k=7)
classi
```

```
EqualInterval

   Interval      Count
----------------------
[ 1.38,  2.94] |   103
( 2.94,  4.50] |    80
( 4.50,  6.07] |     6
( 6.07,  7.63] |     1
( 7.63,  9.20] |     3
( 9.20, 10.76] |     0
(10.76, 12.32] |     5
```

- Quantiles

```
classi = mc.Quantiles(db["no2_viz"], k=7)
classi
```

```
Quantiles

   Interval      Count
----------------------
[ 1.38,  2.24] |    29
( 2.24,  2.50] |    28
( 2.50,  2.76] |    28
( 2.76,  3.02] |    28
( 3.02,  3.35] |    28
( 3.35,  3.76] |    28
( 3.76, 12.32] |    29
```

- Fisher-Jenks

```
classi = mc.FisherJenks(db["no2_viz"], k=7)
classi
```

```
FisherJenks

   Interval      Count
----------------------
[ 1.38,  2.06] |    20
( 2.06,  2.69] |    58
( 2.69,  3.30] |    62
( 3.30,  4.19] |    42
( 4.19,  5.64] |     7
( 5.64,  9.19] |     4
( 9.19, 12.32] |     5
```

Now let's dig into the internals of `classi`:

```
classi
```

```
FisherJenks

   Interval      Count
----------------------
[ 1.38,  2.06] |    20
( 2.06,  2.69] |    58
( 2.69,  3.30] |    62
( 3.30,  4.19] |    42
( 4.19,  5.64] |     7
( 5.64,  9.19] |     4
( 9.19, 12.32] |     5
```

```
classi.k
```

```
7
```

```
classi.bins
```

```
array([ 2.05617382,  2.6925931 ,  3.30281182,  4.19124954,  5.63804861,
        9.19190206, 12.32332434])
```

```
classi.yb
```

```
array([2, 3, 3, 1, 1, 2, 1, 1, 1, 0, 0, 3, 2, 1, 1, 1, 3, 1, 1, 1, 2, 0,
       0, 4, 2, 1, 3, 1, 0, 0, 0, 1, 2, 2, 6, 5, 4, 2, 1, 3, 2, 3, 2, 1,
       2, 3, 2, 3, 1, 1, 3, 1, 2, 3, 3, 1, 3, 3, 1, 0, 1, 1, 3, 2, 0, 0,
       2, 1, 0, 0, 0, 2, 0, 1, 3, 3, 3, 2, 3, 2, 3, 1, 2, 3, 1, 1, 1, 1,
       2, 1, 2, 2, 1, 2, 2, 2, 1, 3, 2, 3, 2, 2, 2, 1, 2, 3, 3, 2, 0, 3,
       1, 0, 1, 2, 1, 1, 2, 1, 2, 6, 5, 6, 2, 2, 3, 6, 3, 4, 3, 4, 2, 3,
       0, 2, 5, 6, 4, 5, 2, 2, 2, 1, 1, 1, 2, 1, 2, 3, 3, 2, 2, 2, 3, 2,
       1, 1, 3, 4, 2, 1, 3, 1, 2, 3, 4, 0, 1, 1, 2, 1, 2, 2, 2, 2, 1, 2,
       2, 2, 0, 0, 1, 2, 3, 3, 3, 3, 3, 2, 1, 2, 1, 1, 1, 2, 2, 1, 3, 1])
```

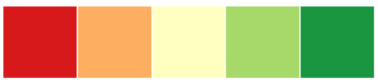
How *many* colors?

Attention

The code used to generate this figure uses more advanced features than planned for this course.
If you want to inspect it, toggle the cell below.

Using the *right* color

For a safe choice, make sure to visit ColorBrewer

-  **Categories**, non-ordered
-  Graduated, **sequential**
-  Graduated, **divergent**

Streamlined

How can we create classifications from data on geo-tables? Two ways:

- Directly within `plot` (only for some algorithms)

```
db.plot(
    "no2_viz", scheme="quantiles", k=7, legend=True
);
```

See this tutorial for more details on fine tuning choropleths manually

> **Challenge**
>
> Create an equal interval map with five bins for `no2_viz`

Manual approach

This is valid for any algorithm and provides much more flexibility at the cost of effort.

```
classi = mc.Quantiles(db["no2_viz"], k=7)
db.assign(
    classes=classi.yb
).plot("classes");
```

Value by alpha mapping

See here for more examples of VBA mapping.

```
db['area_inv'] = 1 / db.to_crs(epsg=5726).area
```

Legendgrams

Legendgrams are a way to more closely connect the statistical characteristics of your data to the map display.

> **Warning**
>
> Legendgrams are *experimental* at the moment so the code is a bit more involved and less stable. Use at your own risk!

Unfold the cell for an example.

Challenge

Give Task I in this block of the GDS course a go.

Choropleths on surfaces

Data
If you want to read more about the data sources behind this dataset, head to the Datasets section

Local files | Online read

Assuming you have the file locally on the path `../data/`:

```
grid = xarray.open_rasterio(
    "../data/cambodia_s5_no2.tif"
).sel(band=1)
```

- (Implicit) continuous equal interval

```
grid.where(
    grid != grid.rio.nodata
).plot(cmap="viridis");
```

```
grid.where(
    grid != grid.rio.nodata
).plot(cmap="viridis", robust=True);
```

- Discrete equal interval

```
grid.where(
    grid != grid.rio.nodata
).plot(cmap="viridis", levels=7)
```

```
<matplotlib.collections.QuadMesh at 0x7f0087843c70>
```

- Combining with `mapclassify`

```
grid_nona = grid.where(
    grid != grid.rio.nodata
)

classi = mc.Quantiles(
    grid_nona.to_series().dropna(), k=7
)

grid_nona.plot(
    cmap="viridis", levels=classi.bins
)
plt.title(classi.name);
```

Challenge

Read the satellite image for Madrid used in the previous section and create three choropleths, one for each band, using the colormaps `Reds`, `Greens`, `Blues`.

Play with different classification algorithms.

- *Do the results change notably?*
- *If so, why do you think that is?*

🐾 Next steps

If you are interested in statistical maps based on classification, here are two recommendations to check out next:

- On the technical side, the documentation for **`mapclassify`** (including its tutorials) provides more detail and illustrates more classification algorithms than those reviewed in this block
- On a more conceptual note, Cynthia Brewer's "Designing better maps" [Bre15] is an excellent blueprint for good map making.

Spatial Feature Engineering (I)

Map Matching

📖 Ahead of time…

Feature Engineering is a common term in machine learning that refers to the processes and transformations involved in turning data from the state in which the modeller access them into what is then fed to a model. This can take several forms, from standardisation of the input data, to the derivation of numeric scores that better describe aspects (*features*) of the data we are using.

Spatial Feature Engineering refers to operations we can use to derive "views" or summaries of our data that we can use in models, *using space* as the key medium to create them.

There is only one reading to complete for this block, Chapter 12 of the GDS Book [RABWng]. The first block of Spatial Feature Engineering in this course loosely follows the first part of the chapter (Map Matching), so focus on this first sections for the block.

💻 Hands-on coding

```
import pandas
import geopandas
import xarray, rioxarray
import contextily
import numpy as np
import matplotlib.pyplot as plt
```

Data

If you want to read more about the data sources behind this dataset, head to the Datasets section

Local files | Online read

Assuming you have the file locally on the path `../data/`:

```
regions = geopandas.read_file("../data/cambodia_regional.gpkg")
cities = geopandas.read_file("../data/cambodian_cities.geojson")
pollution = rioxarray.open_rasterio(
    "../data/cambodia_s5_no2.tif"
).sel(band=1)
friction = rioxarray.open_rasterio(
    "../data/cambodia_2020_motorized_friction_surface.tif"
).sel(band=1)
```

Check both geo-tables and the surface are in the same CRS:

```
(
    regions.crs.to_epsg() ==
    cities.crs.to_epsg() ==
    pollution.rio.crs.to_epsg()
)
```

```
True
```

Polygons to points

In which region is a city?

```
sj = geopandas.sjoin(cities, regions)
```

```
#   City name | Region name
sj[["UC_NM_MN", "adm2_name"]]
```

	UC_NM_MN	adm2_name
0	Sampov Lun	Sampov Lun
1	Khum Pech Chenda	Phnum Proek
2	Poipet	Paoy Paet
3	Sisophon	Serei Saophoan
4	Battambang	Battambang
5	Siem Reap	Siem Reap
6	Sihanoukville	Preah Sihanouk
7	N/A	Trapeang Prasat
8	Kampong Chhnang	Kampong Chhnang
9	Phnom Penh	Tuol Kouk
10	Kampong Cham	Kampong Cham

Challenge

Using the Madrid AirBnb properties and neighbourhoods dataset, can you determine the neighbourhood group of the first ten properties?

Points to polygons

If we were after the number of cities per region, it is a similar approach, with a (`groupby`) twist at the end:

Note

1. We `set_index` to align both tables
2. We `assign` to create a new column

If you want no missing values, you can `fillna(0)` since you *know* missing data are zeros

```
regions.set_index(
    "adm2_name"
).assign(
    city_count=sj.groupby("adm2_name").size()
).info()
```

```
<class 'geopandas.geodataframe.GeoDataFrame'>
Index: 198 entries, Mongkol Borei to Administrative unit not available
Data columns (total 6 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   adm2_altnm  122 non-null    object
 1   motor_mean  198 non-null    float64
 2   walk_mean   198 non-null    float64
 3   no2_mean    198 non-null    float64
 4   geometry    198 non-null    geometry
 5   city_count  11 non-null     float64
dtypes: float64(4), geometry(1), object(1)
memory usage: 10.8+ KB
```

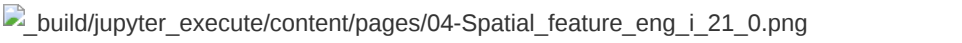
Challenge

Using the Madrid AirBnb properties, can you compute how many properties each neighbourhood group has?

Surface to points

Consider attaching to each city in `cities` the pollution level, as expressed in `pollution`.

The code for generating this figure is a bit more advanced as it fiddles with text, but if you want to explore it you can toggle it on



```
from rasterstats import point_query

city_pollution = point_query(
    cities,
    pollution.values,
    affine=pollution.rio.transform(),
    nodata=pollution.rio.nodata
)
city_pollution
```

```
[3.9397064813333136e-05,
 3.4949825609644426e-05,
 3.825255125820345e-05,
 4.103826573585785e-05,
 3.067677208474005e-05,
 5.108273256655399e-05,
 2.2592785882580366e-05,
 4.050414400882722e-05,
 2.4383652926989897e-05,
 0.0001285838935209779,
 3.258245740282522e-05]
```

And we can map these on the city locations:

```
ax = cities.assign(
    pollution=city_pollution
).plot(
    "pollution",
    cmap="YlOrRd",
    legend=True
)

contextily.add_basemap(
    ax=ax, crs=cities.crs,
);
```

Challenge

Can you calculate the pollution level at the centroid of each Cambodian region in the regional aggregates dataset? how does it compare to their average value?

Surface to polygons

Instead of transferring to points, we want to aggregate all the information in a surface that falls *within* a polygon.

For this case, we will use the motorised friction surface. The question we are asking thus is: *what is the average degree of friction of each region?* Or, in other words: *what regions are harder to get through with motorised transport?*

Again, we can rely on `rasterstats`:

The output is returned from `zonal_stats` as a list of dicts. To make it more manageable, we convert it into a `pandas.DataFrame`.

```
from rasterstats import zonal_stats

regional_friction = pandas.DataFrame(
    zonal_stats(
        regions,
        friction.values,
        affine=friction.rio.transform(),
        nodata=friction.rio.nodata
    ),
    index=regions.index
)
regional_friction.head()
```

	min	max	mean	count
0	0.001200	0.037000	0.006494	979
1	0.001200	0.060000	0.007094	1317
2	0.001200	0.024112	0.006878	324
3	0.001333	0.060000	0.009543	758
4	0.001200	0.060132	0.008619	55

This can then also be mapped onto the polygon geography:

Challenge

Replicate the analysis above to obtain the average friction for each region using the walking surface (`cambodia_2020_walking_friction_surface.tif`).

Surface to surface

If we want to align the `pollution` surface with that of `friction`, we need to resample them to make them "fit on the same frame".

```
pollution.shape
```

```
(138, 152)
```

```
friction.shape
```

```
(574, 636)
```

This involves either moving one surface to the frame of the other one, or both into an entirely new one. For the sake of the illustration, we will do the latter and select a frame that is 300 by 400 pixels. Note this involves stretching (upsampling) `pollution`, while compressing (downsampling) `friction`.

```
# Define dimensions
dimX, dimY = 300, 400
minx, miny, maxx, maxy = pollution.rio.bounds()
# Create XY indices
ys = np.linspace(miny, maxy, dimY)
xs = np.linspace(minx, maxx, dimX)
# Set up placeholder array
canvas = xarray.DataArray(
    np.zeros((dimY, dimX)),
    coords=[ys, xs],
    dims=["y", "x"]
).rio.write_crs(4326) # Add CRS
```

```
cvs_pollution = pollution.rio.reproject_match(canvas)
cvs_friction = friction.rio.reproject_match(canvas)
```

```
cvs_pollution.shape
```

```
(400, 300)
```

```
cvs_pollution.shape == cvs_friction.shape
```

```
True
```

Challenge

Trasfer the `pollution` surface to the frame of `friction`, and viceversa.

Attention

The following methods involve modelling and are thus more sophisticated. Take these as a conceptual introduction with an empirical illustration, but keep in mind there are extense literatures on each of them and these cover some of the simplest cases.

Points to points

See this section of Chapter 12 of the GDS Book [RABWng] for more details on the technique

For this exampe, we will assume that, instead of a surface with pollution values, we only have available a sample of points and we would like to obtain estimates for other locations.

For that we will first generate 100 random points within the extent of `pollution` which we will take as the location of our measurement stations:

Note

The code in this cell contains bits that are a bit more advanced, do not despair if not everything makes sense!

```
np.random.seed(123456)

bb = pollution.rio.bounds()
station_xs = np.random.uniform(bb[0], bb[2], 100)
station_ys = np.random.uniform(bb[1], bb[3], 100)
stations = geopandas.GeoSeries(
    geopandas.points_from_xy(station_xs, station_ys),
    crs="EPSG:4326"
)
```

Our station values come from the `pollution` surface, but we assume we do not have access to the latter, and we would like to obtain estimates for the location of the cities:

We will need the location and the pollution measurements for every station as separate arrays. Before we do that, since we will be calculating distances, we convert our coordinates to a system expressed in metres.

```
stations_mt = stations.to_crs(epsg=5726)
station_xys = np.array(
    [stations_mt.geometry.x, stations_mt.geometry.y]
).T
```

We also need to extract the pollution measurements for each station location:

```
station_measurements = np.array(
    point_query(
        stations,
        pollution.values,
        affine=pollution.rio.transform(),
        nodata=pollution.rio.nodata
    )
)
```

And finally, we will also need the locations of each city expressed in the same coordinate system:

```
cities_mt = cities.to_crs(epsg=5726)
city_xys = np.array(
    [cities_mt.geometry.x, cities_mt.geometry.y]
).T
```

For this illustration, we will use a \(k\)-nearest neighbors regression that estimates the value for each target point (`cities` in our case) as the average weighted by distance of its \(k\) nearest neigbors. In this illustration we will use \(k=10\).

Note how `sklearn` relies only on array data structures, hence why we first had to express all the required information in that format

```
from sklearn.neighbors import KNeighborsRegressor

model = KNeighborsRegressor(
    n_neighbors=10, weights="distance"
).fit(station_xys, station_measurements)
```

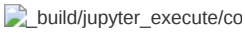
Once we have trained the model, we can use it to obtain predictions for each city location:

```
predictions = model.predict(city_xys)
```

These can be compared with the originally observed values:

```
p2p_comparison = pandas.DataFrame(
    {
        "Observed": city_pollution,
        "Predicted": predictions
    },
    index=cities["UC_NM_MN"]
)
```

```
p2p_comparison
```



	Observed	Predicted
UC_NM_MN		
Sampov Lun	0.000039	0.000027
Khum Pech Chenda	0.000035	0.000025
Poipet	0.000038	0.000030
Sisophon	0.000041	0.000030
Battambang	0.000031	0.000027
Siem Reap	0.000051	0.000027
Sihanoukville	0.000023	0.000019
N/A	0.000041	0.000028
Kampong Chhnang	0.000024	0.000032
Phnom Penh	0.000129	0.000042
Kampong Cham	0.000033	0.000033

> **Challenge**
>
> Replicate the analysis above with \(k=15\) and \(k=5\). *Do results change? Why do you think that is?*

Points to surface

Imagine we do not have a surface like `pollution` but we need it. In this context, if you have measurements from some locations, such as in `stations`, we can use the approach reviewed above to generate a surface. The trick to do this is to realise that we can generate a *uniform* grid of target locations that we can then express as a surface.

We will set as our target locations those of the pixels in the target surface we have seen above:

```
canvas_mt = canvas.rio.reproject(5726)
```

```
xy_pairs = canvas_mt.to_series().index
xys = np.array(
    [
        xy_pairs.get_level_values("x"),
        xy_pairs.get_level_values("y")
    ]
).T
```

To obtain pollution estimates at each location, we can `predict` with `model`:

```
predictions_grid = model.predict(xys)
```

And with these at hand, we can convert them into a surface:

```
predictions_series = pandas.DataFrame(
    {"predictions_grid": predictions_grid}
).join(
    pandas.DataFrame(xys, columns=["x", "y"])
).set_index(["y", "x"])

predictions_surface = xarray.DataArray().from_series(
    predictions_series["predictions_grid"]
).rio.write_crs(canvas_mt.rio.crs)
```

```
f, axs = plt.subplots(1, 2, figsize=(16, 6))

cvs_pollution.where(
    cvs_pollution>0
).plot(ax=axs[0])
axs[0].set_title("Observed")

predictions_surface.where(
    predictions_surface>0
).rio.reproject_match(
    cvs_pollution
).plot(ax=axs[1])
axs[1].set_title("Predicted")

plt.show()
```


_build/jupyter_execute/content/pages/04-Spatial_feature_eng_i_74_0.png

```
f, ax = plt.subplots(1, figsize=(9, 4))
cvs_pollution.where(
    cvs_pollution>0
).plot.hist(
    bins=100, alpha=0.5, ax=ax, label="Observed"
)
predictions_surface.rio.reproject_match(
    cvs_pollution
).plot.hist(
    bins=100, alpha=0.5, ax=ax, color="g", label="predicted"
)
plt.legend()
plt.show()
```


_build/jupyter_execute/content/pages/04-Spatial_feature_eng_i_75_0.png

Room for improvement but, remember this was a rough first pass!

> **Challenge**
>
> Train a model with pollution measurements from each city location and generate a surface from it. *How does the output compare to the one above? Why do you think that is?*

Polygons to polygons

In this final example, we transfer data from a polygon geography to *another* polygon geography. Effectively, we re-apportion values from one set of areas to another based on the extent of shared area.

Our illustration will cover how to move pollution estimates from `regions` into a uniform hexagonal grid we will first create.

> **Important**
>
> This code requires `tobler` 0.7.0 or above

```
import tobler

hex_grid = tobler.util.h3fy(
    regions, resolution=5
)
```

```
/opt/conda/lib/python3.9/site-packages/tobler/util/util.py:151: FutureWarning: Currently,
index_parts defaults to True, but in the future, it will default to False to be
consistent with Pandas. Use `index_parts=True` to keep the current behavior and
True/False to silence the warning.
  source = source.explode()
```

Not that pollution is expressed as an *intesive* (rate) variable. We need to recognise this when specifying the interpolation model:

> **Attention**
>
> This feature requires `tobler` 6.0 or above

```
%%time
pollution_hex = tobler.area_weighted.area_interpolate(
    regions.assign(geometry=regions.buffer(0)).to_crs(epsg=5726),
    hex_grid.to_crs(epsg=5726),
    intensive_variables=["no2_mean"]
)
```

```
CPU times: user 439 ms, sys: 12 ms, total: 451 ms
Wall time: 448 ms
```

And the results look like:

> **Challenge**
>
> Replicate the analytis using `resolution = 4`. *How is the result different? Why?*

🐾 Next steps

If you are interested in learning more about spatial feature engineering through map matching, the following pointers might be useful to delve deeper into specific types of "data transfer":

- The **`datashader`** library is a great option to transfer geo-tables into surfaces, providing tooling to perform these operations in a highly efficient and performant way.
- When aggregating surfaces into geo-tables, the library **`rasterstats`** contains most if not all of the machinery you will need.
- For transfers from polygon to polygon geographies, **`tobler`** is your friend. Its official documentation contains examples for different use cases.

Spatial Feature Engineering (II)

Map Synthesis

📖 Ahead of time…

In this second part of Spatial Feature Engineering, we turn to Map Synthesis. There is only one reading to complete for this block, Chapter 12 of the GDS Book [RABWng]. This block of Spatial Feature Engineering in this course loosely follows the second part of the chapter (Map Synthesis).

💻 Hands-on coding

```
import pandas, geopandas
import numpy as np
import contextily
import tobler
```

> **Data**

If you want to read more about the data sources behind this dataset, head to the Datasets section

Local files | Online read

Assuming you have the file locally on the path `../data/`:

```
pts = geopandas.read_file("../data/madrid_abb.gpkg")
```

We will be working with a modified version of `pts`:

- Since we will require distance calculations, we will switch to the Spanish official projection
- To make calculations in the illustration near-instantaneous, we will work with a smaller (random) sample of Airbnb properties (10% of the total)

```
db = pts.sample(
    frac=0.1, random_state=123
).to_crs(epsg=25830)
```

As you can see in the description, the new CRS is expressed in metres:

```
db.crs
```

```
<Projected CRS: EPSG:25830>
Name: ETRS89 / UTM zone 30N
Axis Info [cartesian]:
- E[east]: Easting (metre)
- N[north]: Northing (metre)
Area of Use:
- name: Europe between 6°W and 0°W: Faroe Islands offshore; Ireland - offshore; Jan Mayen
- offshore; Norway including Svalbard - offshore; Spain - onshore and offshore.
- bounds: (-5.9999999999999, 35.265663028, 1.7053025658242e-13, 80.489344496333)
Coordinate Operation:
- name: UTM zone 30N
- method: Transverse Mercator
Datum: European Terrestrial Reference System 1989 ensemble
- Ellipsoid: GRS 1980
- Prime Meridian: Greenwich
```

Distance buffers

How many Airbnb's are within 500m of each Airbnb?

```
from pysal.lib import weights
```

Using `DistanceBand`, we can build a spatial weights matrix that assigns `1` to each observation within 500m, and `0` otherwise.

```
%%time
w500m = weights.DistanceBand.from_dataframe(
    db, threshold=500, binary=True
)
```

```
CPU times: user 214 ms, sys: 13.8 ms, total: 228 ms
Wall time: 226 ms
```

```
/opt/conda/lib/python3.9/site-packages/libpysal/weights/weights.py:172: UserWarning: The
weights matrix is not fully connected:
 There are 86 disconnected components.
 There are 47 islands with ids: 6878, 16772, 15006, 1336, 3168, 15193, 1043, 5257, 4943,
12849, 10609, 11309, 10854, 10123, 3388, 9380, 10288, 13071, 3523, 15316, 3856, 205,
7720, 10454, 18307, 3611, 12405, 10716, 14813, 15467, 1878, 16597, 14329, 7933, 16215,
13525, 13722, 11932, 14456, 8848, 15197, 8277, 9922, 13072, 13852, 5922, 17151.
  warnings.warn(message)
```

The number of neighbors can be accessed through the `cardinalities` attribute:

```
n_neis = pandas.Series(w500m.cardinalities)
n_neis.head()
```

```
11297    213
2659       5
16242     21
15565      9
14707    159
dtype: int64
```

Challenge

Calculate the number of AirBnb properties within 250m of each other property. *What is the average?*

Distance rings

How many Airbnb's are between 500m and 1km of each Airbnb?

```
%%time
w1km = weights.DistanceBand.from_dataframe(
    db, threshold=1000, binary=True
)
```

```
CPU times: user 575 ms, sys: 31.1 ms, total: 606 ms
Wall time: 602 ms
```

```
/opt/conda/lib/python3.9/site-packages/libpysal/weights/weights.py:172: UserWarning: The
weights matrix is not fully connected:
 There are 20 disconnected components.
 There are 5 islands with ids: 4943, 12849, 15467, 13525, 11932.
  warnings.warn(message)
```

Now, we could do simply a subtraction:

```
n_ring_neis = pandas.Series(w1km.cardinalities) - n_neis
```

Or, if we need to know *which is which*, we can use set operations on weights:

```
w_ring = weights.w_difference(w1km, w500m, constrained=False)
```

```
/opt/conda/lib/python3.9/site-packages/libpysal/weights/weights.py:172: UserWarning: The
weights matrix is not fully connected:
 There are 34 disconnected components.
 There are 23 islands with ids: 3744, 4143, 4857, 4943, 6986, 8345, 8399, 9062, 10592,
10865, 11574, 11613, 11785, 11840, 11932, 12015, 12635, 12714, 12849, 13091, 13317,
13525, 15467.
  warnings.warn(message)
```

And we can confirm they're both the same:

```
(pandas.Series(w_ring.cardinalities) - n_ring_neis).sum()
```

```
0
```

> **Challenge**
>
> Can you create a plot with the following two lines?
>
> - One depicting the average number of properties within a range of 50m, 100m, 250m, 500m, 750m
> - Another one with the *increase* of average neighbors for the same distances above

Cluster membership (points)

We can use the spatial configuration of observations to classify them as part of clusters or not, which can then be encoded, for example, as dummy variables in a model.

These *magic* numbers need to be pre-set and you can play with both `min_pct` (or `min_pts` directly) and `eps` to see how they affect the results (spoiler: a lot!)

```
from sklearn.cluster import DBSCAN

min_pct = 2
min_pts = len(db) * min_pct // 100
eps = 500
```

We will illustrate it with a minimum number of points of `min_pct` % of the sample and a maximum radious of `eps` metres.

```
model = DBSCAN(min_samples=min_pts, eps=eps)
model.fit(
    db.assign(
        x=db.geometry.x
    ).assign(
        y=db.geometry.y
    )[['x', 'y']]
);
```

We will attach the labels to `db` for easy access:

```
db["labels"] = model.labels_
```

We can define boundaries to turn point clusters into polygons if that fits our needs better:

> **Attention**
>
> The code in this cell is a bit more advanced than expected for this course, but is used here as an illustration.

```
from pysal.lib import cg

boundaries = []
cl_ids = [i for i in db["labels"].unique() if i!=-1]
for cl_id in cl_ids:
    sub = db.query(f"labels == {cl_id}")
    cluster_boundaries = cg.alpha_shape_auto(
        np.array(
            [sub.geometry.x, sub.geometry.y]
        ).T,
    )
    boundaries.append(cluster_boundaries)
boundaries = geopandas.GeoSeries(
    boundaries, index=cl_ids, crs=db.crs
)
```

And we can see what the clusters look like:

Challenge

How does the map above change if you require 5% of points instead of 2% for a candidate cluster to be considered so?

Cluster membership (polygons)

We can take a similar approach as above if we have polygon geographies instead of points. Rather than using DBSCAN, here we can rely on local indicators of spatial association (LISAs) to pick up spatial concentrations of high or low values.

For the illustration, we will aggregate the location of Airbnb properties to a regular hexagonal grid, similar to how we generated it when transfering from polygons to polygons. First we create a polygon covering the extent of points:

```
one = geopandas.GeoSeries(
    [cg.alpha_shape_auto(
        np.array(
            [db.geometry.x, db.geometry.y]
        ).T,
    )],
    crs=db.crs
)
```

Then we can tessellate:

```
abb_hex = tobler.util.h3fy(
    one, resolution=8
)
```

```
/opt/conda/lib/python3.9/site-packages/tobler/util/util.py:151: FutureWarning: Currently,
index_parts defaults to True, but in the future, it will default to False to be
consistent with Pandas. Use `index_parts=True` to keep the current behavior and
True/False to silence the warning.
  source = source.explode()
```

And obtain a count of points in each polygon:

```
counts = geopandas.sjoin(
    db, abb_hex
).groupby(
    "index_right"
).size()

abb_hex["count"] = counts
abb_hex["count"] = abb_hex["count"].fillna(0)

abb_hex.plot("count", scheme="fisherjenks");
```

To identify spatial clusters, we rely on `esda`:

```
from pysal.explore import esda
```

```
/opt/conda/lib/python3.9/site-packages/esda/getisord.py:636: SyntaxWarning: "is" with a
literal. Did you mean "=="?
  if __name__ is "__main__":
/opt/conda/lib/python3.9/site-packages/spaghetti/network.py:36: FutureWarning: The next
major release of pysal/spaghetti (2.0.0) will drop support for all ``libpysal.cg``
geometries. This change is a first step in refactoring ``spaghetti`` that is expected to
result in dramatically reduced runtimes for network instantiation and operations. Users
currently requiring network and point pattern input as ``libpysal.cg`` geometries should
prepare for this simply by converting to ``shapely`` geometries.
  warnings.warn(f"{dep_msg}", FutureWarning)
```

And compute the LISA statistics:

```
w = weights.Queen.from_dataframe(abb_hex)
lisa = esda.Moran_Local(abb_hex["count"], w)
```

For a visual inspection of the clusters, `splot`:

```
from pysal.viz import splot
from splot.esda import lisa_cluster
```

```
lisa_cluster(lisa, abb_hex, p=0.01);
```

And, if we want to extract the labels for each polygon, we can do so from the `lisa` object:

```
lisa.q * (lisa.p_sim < 0.01)
```

```
array([0, 0, 1, 0, 0, 3, 0, 0, 0, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 3, 0, 3, 0, 3, 3, 1, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3, 0, 3, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 3, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3,
       0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 3, 0, 3, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 3, 0, 0, 0, 0, 0, 0, 0, 0, 3, 0, 3, 0, 3, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3, 3, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 3, 0, 0, 0, 0, 3, 0, 0, 0, 0, 2, 0, 1, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3, 0, 0, 0, 0, 0, 0,
       0, 0, 3, 0, 0, 0, 3, 0, 0, 0, 0, 0, 3, 0, 3, 0, 0, 0, 0, 3, 3, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3, 0, 3,
       0, 3, 0, 3, 0, 0, 0, 3, 0, 0, 3, 0, 0, 1, 0, 3, 3, 0, 0, 0, 1, 0,
       0, 0, 0, 0, 0, 0, 1, 3, 0, 0, 0, 0, 0, 3, 0, 0, 0])
```

🐾 Next steps

If you want a bit more background into some of the techniques reviewed in this block, the following might be of interest:

- Block E of the GDS Course [AB19] will introduce you to more techniques like the LISAs seen above to explore the spatial dimension of the statistical properties of your data. If you want a more detailed read, Chapter 4 of the GDS Book [RABWng] will do just that.
- Block F of the GDS Course [AB19] will introduce you to more techniques like the LISAs seen above to explore the spatial dimension of the statistical properties of your data. If you want a more detailed read, Chapter 7 of the GDS Book [RABWng] will do just that.
- Block H of the GDS Course [AB19] will introduce you to more techniques for exploring point patterns. If you want a more comprehensive read, Chapter 8 of the GDS Book [RABWng] will do just that.

OpenStreetMap

📖 Ahead of time…

This session is all about OpenStreetMap. To provide an overview of what the project is, whether you have never heard of it or you are somewhat familiar, the followring will set your mind “on course”:

- The following short clip provides a general overview of what OpenStreetMap is

- This recent piece contains several interesting points about how OpenStreetMap is currently being created and some of the implications this model may have.
- Anderson et al. (2019) [ASP19] provides some of the academic underpinnings to the views expressed in Morrison’s piece

💻 Hands-on coding

```
import geopandas
import contextily
from IPython.display import GeoJSON
```

Since some of the query options we will discuss involve pre-defined extents, we will read the Madrid neighbourhoods dataset first:

Local files Online read

Assuming you have the file locally on the path `../data/`:

```
neis = geopandas.read_file("../data/neighbourhoods.geojson")
```

To make some of the examples below *easy* on OpenStreetMap servers, we will single out the smallest neighborhood:

```
areas = neis.to_crs(
    epsg=32630
).area

smallest = neis[areas == areas.min()]
smallest
```

	neighbourhood	neighbourhood_group	geometry
98	Atalaya	Ciudad Lineal	MULTIPOLYGON (((-3.66195 40.46338, -3.66364 40...

```
ax = smallest.plot(
    facecolor="none", edgecolor="blue", linewidth=2
)
contextily.add_basemap(
    ax,
    crs=smallest.crs,
    source=contextily.providers.OpenStreetMap.Mapnik
);
```

osmnx

```
import osmnx as ox
```

Here is a trick to pin all your queries to OpenStreetMap to a specific date, so results are always reproducible, even if the map changes in the meantime.
Tip courtesy of Martin Fleischmann.

> **Tip**
>
> Much of the methods covered here rely on the `osmnx.geometries` module. Check out its reference here

There are two broad areas to keep in mind when querying data on OpenStreetMap through `osmnx`:

- The interface to specify the *extent* of the search
- The *nature* of the entities being queried. Here, the interface relies entirely on OpenStreetMap's tagging system. Given the distributed nature of the project, this is variable, but a good place to start is:

> https://wiki.openstreetmap.org/wiki/Tags

Generally, the interface we will follow involves the following:

```
received_entities = ox.geometries_from_XXX(
    <extent>, tags={<key>: True/<value(s)>}, ...
)
```

The `<extent>` can take several forms:

```
['geometries_from_address',
 'geometries_from_bbox',
 'geometries_from_place',
 'geometries_from_point',
 'geometries_from_polygon',
 'geometries_from_xml']
```

The `tags` follow the official feature spec.

Buildings

```
blgs = ox.geometries_from_polygon(
    smallest.squeeze().geometry, tags={"building": True}
)
```

```
blgs.plot();
```

```
/opt/conda/lib/python3.8/site-packages/ipykernel/ipkernel.py:283: DeprecationWarning:
`should_run_async` will not call `transform_cell` automatically in the future. Please
pass the result to `transformed_cell` argument and any exception that happen during
thetransform in `preprocessing_exc_tuple` in IPython 7.17 and above.
  and should_run_async(code)
```


```
blgs.info()
```

```
<class 'geopandas.geodataframe.GeoDataFrame'>
Int64Index: 115 entries, 0 to 114
Data columns (total 30 columns):
 #   Column            Non-Null Count  Dtype
---  ------            --------------  -----
 0   unique_id         115 non-null    object
 1   osmid             115 non-null    int64
 2   element_type      115 non-null    object
 3   amenity           2 non-null      object
 4   name              2 non-null      object
 5   geometry          115 non-null    geometry
 6   nodes             115 non-null    object
 7   building          115 non-null    object
 8   addr:housenumber  21 non-null     object
 9   addr:postcode     3 non-null      object
 10  addr:street       9 non-null      object
 11  denomination      1 non-null      object
 12  phone             2 non-null      object
 13  religion          1 non-null      object
 14  source            1 non-null      object
 15  source:date       1 non-null      object
 16  url               1 non-null      object
 17  wheelchair        1 non-null      object
 18  building:levels   11 non-null     object
 19  addr:city         8 non-null      object
 20  addr:country      6 non-null      object
 21  country           1 non-null      object
 22  diplomatic        1 non-null      object
 23  name:en           1 non-null      object
 24  name:fr           1 non-null      object
 25  name:ko           1 non-null      object
 26  office            1 non-null      object
 27  target            1 non-null      object
 28  website           1 non-null      object
 29  wikidata          1 non-null      object
dtypes: geometry(1), int64(1), object(28)
memory usage: 27.9+ KB
```

```
/opt/conda/lib/python3.8/site-packages/ipykernel/ipkernel.py:283: DeprecationWarning:
`should_run_async` will not call `transform_cell` automatically in the future. Please
pass the result to `transformed_cell` argument and any exception that happen during
thetransform in `preprocessing_exc_tuple` in IPython 7.17 and above.
  and should_run_async(code)
```

```
blgs.head()
```

	unique_id	osmid	element_type	amenity	name	geometry	
0	way/442595762	442595762	way	NaN	NaN	POLYGON ((-3.66377 40.46317, -3.66363 40.46322...	[44(44(44(44(
1	way/442595763	442595763	way	NaN	NaN	POLYGON ((-3.66394 40.46346, -3.66415 40.46339...	[44(44(44(44(
2	way/442595764	442595764	way	NaN	NaN	POLYGON ((-3.66379 40.46321, -3.66401 40.46314...	[44(44(44(44(
3	way/442595765	442595765	way	NaN	NaN	POLYGON ((-3.66351 40.46356, -3.66294 40.46371...	[44(44(44(44(
4	way/442596830	442596830	way	NaN	NaN	POLYGON ((-3.66293 40.46289, -3.66281 40.46294...	[44(44(44(44(

5 rows × 30 columns

If you want to visit the entity online, you can do so at:

```
https://www.openstreetmap.org/<unique_id>
```

Other polygons

```
park = ox.geometries_from_place(
    "Parque El Retiro, Madrid", tags={"leisure": "park"}
)
```

```
/opt/conda/lib/python3.8/site-packages/ipykernel/ipkernel.py:283: DeprecationWarning:
`should_run_async` will not call `transform_cell` automatically in the future. Please
pass the result to `transformed_cell` argument and any exception that happen during
thetransform in `preprocessing_exc_tuple` in IPython 7.17 and above.
  and should_run_async(code)
```

```
ax = park.plot(
    facecolor="none", edgecolor="blue", linewidth=2
)
contextily.add_basemap(
    ax,
    crs=smallest.crs,
    source=contextily.providers.OpenStreetMap.Mapnik
);
```

```
/opt/conda/lib/python3.8/site-packages/ipykernel/ipkernel.py:283: DeprecationWarning:
`should_run_async` will not call `transform_cell` automatically in the future. Please
pass the result to `transformed_cell` argument and any exception that happen during
thetransform in `preprocessing_exc_tuple` in IPython 7.17 and above.
  and should_run_async(code)
```


_build/jupyter_execute/content/pages/06-OpenStreetMap_31_1.png

Points of interest

Bars around Atocha station:

```
bars = ox.geometries_from_address(
    "Madrid Puerta de Atocha", tags={"amenity": "bar"}, dist=1500
)
```

```
/opt/conda/lib/python3.8/site-packages/ipykernel/ipkernel.py:283: DeprecationWarning:
`should_run_async` will not call `transform_cell` automatically in the future. Please
pass the result to `transformed_cell` argument and any exception that happen during
thetransform in `preprocessing_exc_tuple` in IPython 7.17 and above.
  and should_run_async(code)
```

We can quickly explore with `GeoJSON`:

```
GeoJSON(bars.__geo_interface__)
```

```
/opt/conda/lib/python3.8/site-packages/ipykernel/ipkernel.py:283: DeprecationWarning:
`should_run_async` will not call `transform_cell` automatically in the future. Please
pass the result to `transformed_cell` argument and any exception that happen during
thetransform in `preprocessing_exc_tuple` in IPython 7.17 and above.
  and should_run_async(code)
```

```
<IPython.display.GeoJSON object>
```

And stores within Malasaña:

```
shops = ox.geometries_from_address(
    "Malasaña, Madrid, Spain", # Boundary to search within
    tags={
        "shop": True,
        "landuse": ["retail", "commercial"],
        "building": "retail"
    },
    dist=1000
)
```

We use `geometries_from_place` for delineated areas (“polygonal entities”):

```
cs = ox.geometries_from_place(
    "Madrid, Spain",
    tags={"amenity": "charging_station"}
)
GeoJSON(cs.__geo_interface__)
```

```
/opt/conda/lib/python3.8/site-packages/ipykernel/ipkernel.py:283: DeprecationWarning:
`should_run_async` will not call `transform_cell` automatically in the future. Please
pass the result to `transformed_cell` argument and any exception that happen during
thetransform in `preprocessing_exc_tuple` in IPython 7.17 and above.
  and should_run_async(code)
```

```
<IPython.display.GeoJSON object>
```

Similarly, we can work with location data. For example, searches around a given point:

```
bakeries = ox.geometries_from_point(
    (40.418881103417675, -3.6920446157455444),
    tags={"shop": "bakery", "craft": "bakery"},
    dist=500
)
GeoJSON(bakeries.__geo_interface__)
```

```
/opt/conda/lib/python3.8/site-packages/ipykernel/ipkernel.py:283: DeprecationWarning:
`should_run_async` will not call `transform_cell` automatically in the future. Please
pass the result to `transformed_cell` argument and any exception that happen during
thetransform in `preprocessing_exc_tuple` in IPython 7.17 and above.
  and should_run_async(code)
```

```
<IPython.display.GeoJSON object>
```

Streets

Street data can be obtained as another type of entity, as above; or as a graph object.

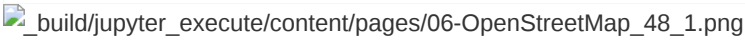
Geo-tables

```
centro = ox.geometries_from_polygon(
    neis.query("neighbourhood == 'Sol'").squeeze().geometry,
    tags={"highway": True}
)
```

```
/opt/conda/lib/python3.8/site-packages/ipykernel/ipkernel.py:283: DeprecationWarning:
`should_run_async` will not call `transform_cell` automatically in the future. Please
pass the result to `transformed_cell` argument and any exception that happen during
thetransform in `preprocessing_exc_tuple` in IPython 7.17 and above.
  and should_run_async(code)
```

We can get a quick peak into what is returned (grey), compared to the region we used for the query:

```
/opt/conda/lib/python3.8/site-packages/ipykernel/ipkernel.py:283: DeprecationWarning:
`should_run_async` will not call `transform_cell` automatically in the future. Please
pass the result to `transformed_cell` argument and any exception that happen during
thetransform in `preprocessing_exc_tuple` in IPython 7.17 and above.
  and should_run_async(code)
```



This however will return all sorts of things:

```
centro.geometry
```

```
/opt/conda/lib/python3.8/site-packages/ipykernel/ipkernel.py:283: DeprecationWarning:
`should_run_async` will not call `transform_cell` automatically in the future. Please
pass the result to `transformed_cell` argument and any exception that happen during
thetransform in `preprocessing_exc_tuple` in IPython 7.17 and above.
  and should_run_async(code)
```

```
0                             POINT (-3.70427 40.41662)
1                             POINT (-3.70802 40.41612)
2                             POINT (-3.70847 40.41677)
3                             POINT (-3.69945 40.41786)
4                             POINT (-3.70054 40.41645)
                              ...
604    LINESTRING (-3.70686 40.41380, -3.70719 40.41369)
605    LINESTRING (-3.70705 40.42021, -3.70680 40.42020)
606    POLYGON ((-3.70948 40.41551, -3.70952 40.41563...
607    POLYGON ((-3.70243 40.41716, -3.70242 40.41714...
608    POLYGON ((-3.70636 40.41475, -3.70635 40.41481...
Name: geometry, Length: 609, dtype: geometry
```

Spatial graphs

This returns clean, processed *graph* objects for the street network:

```
centro_gr = ox.graph_from_polygon(
    neis.query("neighbourhood == 'Sol'").squeeze().geometry,
)
```

```
[i for i in dir(ox) i
```

> **Note**
>
> For more on graph representations of street networks, see block 07

```
centro_gr
```

```
<networkx.classes.multidigraph.MultiDiGraph at 0x7fe6f9033b50>
```

And to visualise it:

```
ox.plot_figure_ground(centro_gr);
```

```
ox.plot_graph_folium(centro_gr)
```

```
/opt/conda/lib/python3.8/site-packages/ipykernel/ipkernel.py:283: DeprecationWarning:
`should_run_async` will not call `transform_cell` automatically in the future. Please
pass the result to `transformed_cell` argument and any exception that happen during
thetransform in `preprocessing_exc_tuple` in IPython 7.17 and above.
  and should_run_async(code)
```

Leaflet (https://leafletjs.com) | © OpenStreetMap (http://www.openstreetmap.org/copyright) contributors © CartoDB (http://cartodb.com/attributions), CartoDB attributions (http://cartodb.com/attributions)

```
/opt/conda/lib/python3
DeprecationWarning: `s
`transform_cell` autor
result to `transformed
happen during thetrans
7.17 and above.
  and should_run_asyn
```

```
['graph_from_address'
 'graph_from_bbox',
 'graph_from_gdfs',
 'graph_from_place',
 'graph_from_point',
 'graph_from_polygon'
 'graph_from_xml']
```

```
[i for i in dir(ox) i
```

```
['plot_graph', 'plot_
'plot_graph_routes']
```

pyrosm

If you are planning to read full collections of OpenStreetMap entities for a given region, `osmnx` might not be the ideal tool. Instead, it is possible to access extracts of regions and read them in full with **`pyrosm`**, which is faster for *these* operations.

More information about the `pyrosm` project is available on its website

```
import pyrosm
```

If you are working on a “popular” place, there are utilities to acquire the data:

```
mad = pyrosm.get_data("Madrid")
mad
```

```
'/tmp/pyrosm/Madrid.osm.pbf'
```

Once downloaded, we can start up the database:

```
mad_osm = pyrosm.OSM(mad)
```

And we can then read parts of all of OpenStreetMap data available for Madrid through queries to `mad_osm`. It is important to note that `pyrosm` will return queries as `GeoDataFrame` objects, but can also interoperate with graph data structures.

Over to you…

The best way to get a hang on OpenStreetMap tags is by playing with it yourself. To facilitate just that, here are some challenges to get you started.

> **Challenges**
>
> - Extract the building footprints for the Sol neighbourhood in `neis`
> - How many music shops does OSM record within 750 metres of Puerta de Alcalá?
> - Are there more restaurants or clothing shops within the polygon that represents the Pacífico neighbourhood in `neis` table?
> - How many bookshops are within a 50m radious of the Paseo de la Castellana? (**NOTE** this one involves extracting the street segment, drawing a buffer and querying OSM for bookshops)

🐾 Next steps

If you found the content in this block useful, the following resources represent some suggestions on where to go next:

- Parts of the block are inspired and informed by Geoff Boeing's excellent course on Urban Data Science
- More in depth content about `osmnx` is available in the official examples collection
- Boeing (2020) [Boe20a] illustrates how OpenStreetMap can be used to analyse urban form (Open Access)

Spatial Networks

📖 Ahead of time…

Thank you very much to Martin Fleischmann for providing support and ideas in the development of this block

In this block we cover some of the analytics you can obtain when you consider street networks as spatial graphs rather than as geo-tables.

- A good example of applying concepts and ideas presented in this block is Boeing (2020) [Boe20b]
- Boeing (2017) [Boe17] provides a general overview on the `osmnx` project
- A brief overview of `momepy`, the package for urban morphometrics, available in Fleischmann (2019) [Fle19]

💻 Hands-on coding

```
import pandas
import geopandas
import momepy
import networkx as nx
import contextily
import matplotlib.pyplot as plt
```

Local files Online read

Assuming you have the file locally on the path `../data/`:

```
db = geopandas.read_file("../data/arturo_streets.gpkg")
```

To make things easier later, we “explode” the table so it is made up of `LINESTRINGS` instead of `MULTILINESTRINGS`:

```
db_tab = db.explode().reset_index()
```

```
db_tab.head()
```

	level_0	level_1	OGC_FID	dm_id	dist_barri	X	Y
0	0	0	1	1	1606	444133.736820	4.482809e+0(
1	1	0	2	2	1606	444192.038205	4.482878e+0(
2	2	0	3	3	1606	444134.537507	4.482885e+0(
3	3	0	4	4	1603	445612.690578	4.479336e+0(
4	4	0	5	5	1603	445606.319326	4.479354e+0(

Analysing street geo-tables

Length

```
length = db_tab.to_crs(
    epsg=32630 # Expressed in metres
).geometry.length
length.head()
```

```
0    118.699481
1     62.210799
2     95.164472
3     23.503065
4     16.090295
dtype: float64
```

Linearity

```
linearity = momepy.Linearity(db_tab).series
linearity.head()
```

```
0    1.000000
1    0.999999
2    1.000000
3    1.000000
4    1.000000
dtype: float64
```

Streets as spatial graphs

From geo-table to spatial graph:

```
db_graph = momepy.gdf_to_nx(db_tab)

db_graph
```

```
<networkx.classes.multigraph.MultiGraph at 0x7f88c5a76bb0>
```

Now `db_graph` is a different animal than `db` that emphasizes *connections* rather than observation attributes.

```
db_graph.is_directed()
```

```
False
```

```
db_graph.is_multigraph()
```

```
True
```

The (first and last) coordinates of each street segment become the ID for each segment in the graph:

```
print(db_tab.loc[0, "geometry"])
```

```
LINESTRING (444096.3161762458 4482762.870216271, 444171.158127317 4482855.001910598)
```

```
l = db_tab.loc[0, "geometry"]
l.coords
```

```
<shapely.coords.CoordinateSequence at 0x7f88a4619670>
```

```
node0a, node0b = edge0 = list(
    db_tab.loc[0, "geometry"].coords
)
edge0
```

```
[(444096.3161762458, 4482762.870216271),
 (444171.15812731703, 4482855.001910598)]
```

We can use those to extract adjacencies to each node:

```
db_graph[node0a]
```

```
AdjacencyView({(444171.15812731703, 4482855.001910598): {0: {'level_0': 0, 'level_1': 0,
'OGC_FID': '1', 'dm_id': '1', 'dist_barri': '1606', 'X': 444133.736820226, 'Y':
4482808.89166328, 'value': nan, 'geometry': <shapely.geometry.linestring.LineString
object at 0x7f88b3e34a00>, 'mm_len': 118.69948078964639}}, (444083.8275243509,
4482747.422611062): {538: {'level_0': 538, 'level_1': 0, 'OGC_FID': '539', 'dm_id':
'539', 'dist_barri': '1606', 'X': 444090.105664431, 'Y': 4482755.13506047, 'value': nan,
'geometry': <shapely.geometry.linestring.LineString object at 0x7f88b43281c0>, 'mm_len':
19.864413729824115}}})
```

We can access edge information for each pair of nodes with a concatenated dict query:

```
db_graph[node0a][node0b]
```

```
AtlasView({0: {'level_0': 0, 'level_1': 0, 'OGC_FID': '1', 'dm_id': '1', 'dist_barri':
'1606', 'X': 444133.736820226, 'Y': 4482808.89166328, 'value': nan, 'geometry':
<shapely.geometry.linestring.LineString object at 0x7f88b3e34a00>, 'mm_len':
118.69948078964639}})
```

```
db_graph[node0a][node0b][0]
```

```
{'level_0': 0,
 'level_1': 0,
 'OGC_FID': '1',
 'dm_id': '1',
 'dist_barri': '1606',
 'X': 444133.736820226,
 'Y': 4482808.89166328,
 'value': nan,
 'geometry': <shapely.geometry.linestring.LineString at 0x7f88b3e34a00>,
 'mm_len': 118.69948078964639}
```

```
db_graph[node0a][node0b][0]["geometry"]
```


_build/jupyter_execute/content/pages/07-Spatial_networks_33_0.svg

If we need all the node IDs:

```
list(
    db_graph.nodes
)[:5] # Limit to the first five elements
```

```
[(444096.3161762458, 4482762.870216271),
 (444171.15812731703, 4482855.001910598),
 (444212.942998509, 4482901.090971609),
 (444097.96831143444, 4482915.825653204),
 (445608.8837672261, 4479346.814511424)]
```

And same for edges:

> **Note**
>
> `edges` returns a triplet with the origin and destination node IDs, *and* the ID of the edge, which is linked to the ID of the segment in the geo-table

```
list(
    db_graph.edges
)[:5] # Limit to the first five elements
```

```
[((444096.3161762458, 4482762.870216271),
  (444171.15812731703, 4482855.001910598),
  0),
 ((444096.3161762458, 4482762.870216271),
  (444083.8275243509, 4482747.422611062),
  538),
 ((444171.15812731703, 4482855.001910598),
  (444212.942998509, 4482901.090971609),
  1),
 ((444171.15812731703, 4482855.001910598),
  (444097.96831143444, 4482915.825653204),
  2),
 ((444212.942998509, 4482901.090971609),
  (444254.9705938099, 4482866.143285849),
  10886)]
```

Or:

```
db_graph.edges[node0a, node0b, 0]
```

```
{'level_0': 0,
 'level_1': 0,
 'OGC_FID': '1',
 'dm_id': '1',
 'dist_barri': '1606',
 'X': 444133.736820226,
 'Y': 4482808.89166328,
 'value': nan,
 'geometry': <shapely.geometry.linestring.LineString at 0x7f88b3e34a00>,
 'mm_len': 118.69948078964639}
```

If you want fast access to adjacencies:

```
db_graph.adj[node0a]
```

```
AdjacencyView({(444171.15812731703, 4482855.001910598): {0: {'level_0': 0, 'level_1': 0,
'OGC_FID': '1', 'dm_id': '1', 'dist_barri': '1606', 'X': 444133.736820226, 'Y':
4482808.89166328, 'value': nan, 'geometry': <shapely.geometry.linestring.LineString
object at 0x7f88b3e34a00>, 'mm_len': 118.69948078964639}}, (444083.8275243509,
4482747.422611062): {538: {'level_0': 538, 'level_1': 0, 'OGC_FID': '539', 'dm_id':
'539', 'dist_barri': '1606', 'X': 444090.105664431, 'Y': 4482755.13506047, 'value': nan,
'geometry': <shapely.geometry.linestring.LineString object at 0x7f88b43281c0>, 'mm_len':
19.864413729824115}}})
```

Analysing graphs

There are *many* ways to extract information and descriptives from a graph. In this section we will explore a few that can tell us important information about the position of a node or edge in the network and about the broader characteristics of sections of the graph.

Degree

Degree tells us the number of neighbors of every edge, that is how many other nodes it is directly connected to.

```
degree = list(db_graph.degree)
degree[:5]
```

```
[((444096.3161762458, 4482762.870216271), 2),
 ((444171.15812731703, 4482855.001910598), 3),
 ((444212.942998509, 4482901.090971609), 3),
 ((444097.96831143444, 4482915.825653204), 3),
 ((445608.8837672261, 4479346.814511424), 2)]
```

Node centrality

Fraction of nodes a node is connected to:

```
nc = pandas.Series(
    nx.degree_centrality(db_graph)
)
nc.head()
```

```
444096.316176  4.482763e+06    0.00004
444171.158127  4.482855e+06    0.00006
444212.942999  4.482901e+06    0.00006
444097.968311  4.482916e+06    0.00006
445608.883767  4.479347e+06    0.00004
dtype: float64
```

```
nc.plot.hist(bins=100, figsize=(6, 3));
```

> **Tip**
>
> Other variations of centrality measures are available in `networkx`. The are computationally demanding but relatively straightforward to calculate using the library. For a few of those, you can check:
>
> - This **`networkx`** example
> - The **`momepy`** documentation on centrality

Meshedness

The messedness of a graph captures the degree of node edge density as compared to that of nodes. Higher meshedness is related to denser, more inter-connected grids.

```
%time meshd = momepy.meshedness(db_graph, distance=500)
```

```
100%|██████████| 49985/49985 [00:57<00:00, 876.55it/s]
```

```
CPU times: user 57.8 s, sys: 132 ms, total: 57.9 s
Wall time: 57.8 s
```

```
meshd.nodes[node0a]
```

```
{'meshedness': 0.058823529411764705}
```

```
pandas.Series(
    {i: meshd.nodes[i]["meshedness"] for i in meshd.nodes}
).plot.hist(bins=100, figsize=(9, 4));
```

Attaching information to street segments

The trick here is to be able to transfer back the information stored as graphs into geo-tables so we can apply everything we already now about manipulating and mapping data in that structure. With `momepy`, we can bring a graph back into a geo-table:

```
nodes = momepy.nx_to_gdf(
    meshd, points=True, lines=False
)
```

```
nodes.head()
```

	meshedness	nodeID	geometry
0	0.058824	1	POINT (444096.316 4482762.870)
1	0.092308	2	POINT (444171.158 4482855.002)
2	0.101449	3	POINT (444212.943 4482901.091)
3	0.065574	4	POINT (444097.968 4482915.826)
4	0.000000	5	POINT (445608.884 4479346.815)

```
ax = nodes.plot(
    "meshedness",
    scheme="fisherjenkssampled",
    markersize=0.1,
    legend=True,
    figsize=(12, 12)
)
contextily.add_basemap(
    ax,
    crs=nodes.crs,
    source=contextily.providers.CartoDB.DarkMatterNoLabels
)
ax.set_title("Meshedness");
```

With other measures index on node IDs, we can use joining machinery in `pandas`:

```
nc.head()
```

```
444096.316176  4.482763e+06    0.00004
444171.158127  4.482855e+06    0.00006
444212.942999  4.482901e+06    0.00006
444097.968311  4.482916e+06    0.00006
445608.883767  4.479347e+06    0.00004
dtype: float64
```

```
degree_tab = pandas.DataFrame(
    degree, columns=["id", "degree"]
)
degree_tab.index = pandas.MultiIndex.from_tuples(
    degree_tab["id"]
)
degree_tab = degree_tab["degree"]
degree_tab.head()
```

```
444096.316176  4.482763e+06    2
444171.158127  4.482855e+06    3
444212.942999  4.482901e+06    3
444097.968311  4.482916e+06    3
445608.883767  4.479347e+06    2
Name: degree, dtype: int64
```

```
net_stats = pandas.DataFrame(
    {"degree": degree_tab, "centrality": nc},
)
net_stats.index.names = ["x", "y"]
net_stats.head()
```

		degree	centrality
x	y		
444096.316176	**4.482763e+06**	2	0.00004
444171.158127	**4.482855e+06**	3	0.00006
444212.942999	**4.482901e+06**	3	0.00006
444097.968311	**4.482916e+06**	3	0.00006
445608.883767	**4.479347e+06**	2	0.00004

```
net_stats_geo = nodes.assign(
    x=nodes.geometry.x
).assign(
    y=nodes.geometry.y
).set_index(
    ["x", "y"]
).join(net_stats)

net_stats_geo.head()
```

		meshedness	nodeID	geometry	degree	ce
x	y					
444096.316176	**4.482763e+06**	0.058824	1	POINT (444096.316 4482762.870)	2	(
444171.158127	**4.482855e+06**	0.092308	2	POINT (444171.158 4482855.002)	3	(
444212.942999	**4.482901e+06**	0.101449	3	POINT (444212.943 4482901.091)	3	(
444097.968311	**4.482916e+06**	0.065574	4	POINT (444097.968 4482915.826)	3	(
445608.883767	**4.479347e+06**	0.000000	5	POINT (445608.884 4479346.815)	2	(

🐾 Next steps

If you found the content in this block useful, the following resources represent some suggestions on where to go next:

- The NetworkX tutorial is a great place to get a better grasp of the data structures we use to represent (spatial) graphs
- Parts of the block benefit from the section on urban networks in Geoff Boeing's excellent course on Urban Data Science
- If you are interested in urban morphometric analysis (the study of the shape of different elements making up cities), the `momepy` library is an excellent reference to absorb, including its user guide

Transport costs

📖 Ahead of time…

💻 Hands-on coding

```
import momepy
import geopandas
import contextily
import xarray, rioxarray
import osmnx as ox
import numpy as np
import matplotlib.pyplot as plt
```

Moving along (street) networks

Local files Online read

Assuming you have the file locally on the path `../data/`:

```
streets = geopandas.read_file("../data/arturo_streets.gpkg")
abbs = geopandas.read_file("../data/madrid_abb.gpkg")
neis = geopandas.read_file("../data/neighbourhoods.geojson")
```

Shortest-path routing

```
import pandana
```

Before building the routing network, we convert to graph and back in `momepy` to "clean" the network and ensure it complies with requirements for routing.

```
%%time
nodes, edges = momepy.nx_to_gdf(
    momepy.gdf_to_nx(
        streets.explode() # We "explode" to avoid multi-part rows
    )
)
nodes = nodes.set_index("nodeID") # Reindex nodes on ID
```

```
CPU times: user 5.3 s, sys: 28.1 ms, total: 5.33 s
Wall time: 5.33 s
```

Once we have nodes and edges "clean" from the graph representation, we can build a `pandana.Network` object we will use for routing:

```
streets_pdn = pandana.Network(
    nodes.geometry.x,
    nodes.geometry.y,
    edges["node_start"],
    edges["node_end"],
    edges[["mm_len"]]
)

streets_pdn
```

```
<pandana.network.Network at 0x7f7c3b9954f0>
```

How do I go from A to B?

For example, from the first Airbnb in the geo-table…

```
first = abbs.loc[[0], :].to_crs(streets.crs)
```

…to Puerta del Sol.

```
import geopy
geopy.geocoders.options.default_user_agent = "gds4ae"
sol = geopandas.tools.geocode(
    "Puerta del Sol, Madrid", geopy.Nominatim
).to_crs(streets.crs)
sol
```

	geometry	address
0	POINT (440247.912 4474264.981)	Puerta del Sol, Sol, Centro, Madrid, Área metr...

First we snap locations to the network:

```
pt_nodes = streets_pdn.get_node_ids(
    [first.geometry.x, sol.geometry.x],
    [first.geometry.y, sol.geometry.y]
)
pt_nodes
```

```
0     3072
1    35732
Name: node_id, dtype: int64
```

Then we can route the shortest path:

```
route_nodes = streets_pdn.shortest_path(
    pt_nodes[0], pt_nodes[1]
)
route_nodes
```

```
array([ 3072,  3477,  8269,  8267,  8268, 18696, 18694,  1433,  1431,
         354,  8176,  8177, 18122, 17477, 16859, 14323, 16858, 17811,
       44796, 41221, 41218, 41222, 41653, 18925, 18929, 48944, 18932,
       21095, 21096, 23220, 15399, 15400, 15401, 47447, 47448, 23277,
       47449, 23260, 23261, 23262, 27952, 27953, 27954, 48328, 11951,
       11950, 11945, 19476, 19477, 27334, 30089, 43295, 11941, 11942,
       11943, 48326, 37485, 48317, 15894, 15891, 15892, 29955, 25454,
        7342, 34992, 23609, 28218, 21649, 21650, 21652, 39076, 25109,
       25103, 25102, 25101, 48519, 47288, 34624, 31188, 29616, 48557,
       22845, 48554, 48556, 40923, 40922, 40924, 48586, 46373, 46372,
       46371, 45676, 45677, 38779, 38778, 19145, 20499, 20498, 20500,
       47738, 42304, 42303, 35731, 35728, 35730, 35732])
```

With this information, we can build the route line manually:

Attention

The code to generate the route involves writing a function and is a bit more advanced than expected for this course. If this looks too complicated, do not despair.
Also, please note this builds a *simplified* line for the route, not one that is based on the original geometries (distance calculations *are* based on the original network).

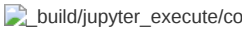
```
from shapely.geometry import LineString

def route_nodes_to_line(nodes, network):
    pts = network.nodes_df.loc[nodes, :]
    s = geopandas.GeoDataFrame(
        {"src_node": [nodes[0]], "tgt_node": [nodes[1]]},
        geometry=[LineString(pts.values)],
        crs=streets.crs
    )
    return s
```

We can calculate the route:

```
route = route_nodes_to_line(route_nodes, streets_pdn)
```

And we get it back as a geo-table (with one row):



```
route
```

	src_node	tgt_node	geometry
0	3072	3477	LINESTRING (442606.507 4478714.516, 442597.100...

If we wanted to obtain the length of the route:

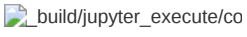
```
route_len = streets_pdn.shortest_path_length(
    pt_nodes[0], pt_nodes[1]
)
round(route_len / 1000, 3) # Dist in Km
```

```
5.514
```

“Third party-driven” routing

How do I go from A to B passing by the “best” buildings?

The overall process is the same; the main difference is, when we build the `Network` object, to replace distance (`mm_len`) with a measure that *combines* distance and building quality. Note that we want to *maximise* building quality, but the routing algorithms use a *minimisation* function. Hence, our composite index will need to reflect that.



The strategy is divided in the following steps:

1. Re-scale distance between 0 and 1
2. Build a measure inverse to building quality in the \([0, 1]\) range
3. Generate a combined measure (`wdist`) by picking a weighting parameter
4. Build a new `Network` object that incorporates `wdist` instead of distance
5. Compute route between the two points of interest

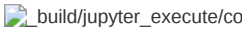
For 1., we can use the scaler in `scikit-learn`:

```
from sklearn.preprocessing import minmax_scale
```

Then generate and attach to `edges` a scaled version of `mm_len`:

```
edges["scaled_dist"] = minmax_scale(edges["mm_len"])
```

We move on to 2., with a similar approach. We will use the negative of the building quality average (`average_quality`):



```
edges["scaled_inv_bquality"] = minmax_scale(
    -edges["average_quality"]
)
```

Taking 1. and 2. into 3. we can build `wdist`. For this example, we will give each dimension the same weight (0.5), but this is at discretion of the researcher.

```
w = 0.5
edges["wdist"] = (
    edges["scaled_dist"] * w +
    edges["scaled_inv_bquality"] * (1-w)
)
```

Now we can recreate the `Network` object based on our new measure (4.) and provide routing. Since it is the same process as with distance, we will do it all in one go:

```
# Build new graph object
w_graph = pandana.Network(
    nodes.geometry.x,
    nodes.geometry.y,
    edges["node_start"],
    edges["node_end"],
    edges[["wdist"]]
)
# Snap locations to their nearest node
pt_nodes = w_graph.get_node_ids(
    [first.geometry.x, sol.geometry.x],
    [first.geometry.y, sol.geometry.y]
)
# Generate route
w_route_nodes = w_graph.shortest_path(
    pt_nodes[0], pt_nodes[1]
)
# Build LineString
w_route = route_nodes_to_line(
    w_route_nodes, w_graph
)
```

Proximity

What is the nearest internet cafe for Airbnb's without WiFi?

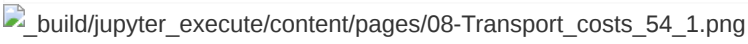
First we identify Airbnb's without WiFi:

```
no_wifi = abbs.query(
    "WiFi == '0'"
).to_crs(streets.crs)
```

Then pull WiFi spots in Madrid from OpenStreetMap:

```
icafes = ox.geometries_from_place(
    "Madrid, Spain", tags={"amenity": "internet_cafe"}
).to_crs(streets.crs).reset_index()
```

```
/opt/conda/lib/python3.8/site-packages/ipykernel/ipkernel.py:283: DeprecationWarning:
`should_run_async` will not call `transform_cell` automatically in the future. Please
pass the result to `transformed_cell` argument and any exception that happen during
thetransform in `preprocessing_exc_tuple` in IPython 7.17 and above.
  and should_run_async(code)
```



The logic for this operation is the following:

1. Add the points of interest (POIs, the internet cafes) to the network object (`streets_pdn`)
2. Find the nearest node to each POI
3. Find the nearest node to each Airbnb without WiFi
4. Connect each Airbnb to its nearest internet cafe

We can add the internet cafes to the network object (1.) with the `set_pois` method:

Note we set `max_items=1` because we are only going to query for the nearest cafe. This will make computations much faster

```
import numpy as np

streets_pdn.set_pois(
    category="Internet cafes",
    maxitems=1,
    maxdist=100000, # 100km so everything is included
    x_col=icafes.geometry.x,
    y_col=icafes.geometry.y,
)
```

```
/opt/conda/lib/python3.8/site-packages/ipykernel/ipkernel.py:283: DeprecationWarning:
`should_run_async` will not call `transform_cell` automatically in the future. Please
pass the result to `transformed_cell` argument and any exception that happen during
thetransform in `preprocessing_exc_tuple` in IPython 7.17 and above.
  and should_run_async(code)
/opt/conda/lib/python3.8/site-packages/pandana/network.py:660: DeprecationWarning: The
default dtype for empty Series will be 'object' instead of 'float64' in a future version.
Specify a dtype explicitly to silence this warning.
  elif isinstance(maxitems, type(pd.Series())):
/opt/conda/lib/python3.8/site-packages/pandana/network.py:668: DeprecationWarning: The
default dtype for empty Series will be 'object' instead of 'float64' in a future version.
Specify a dtype explicitly to silence this warning.
  elif isinstance(maxdist, type(pd.Series())):
```

Once the cafes are added to the network, we can find the nearest one to each node (2.):

Note there are some nodes for which we can't find a nearest cafe. These are related to disconnected parts of the network

```
cafe2nnode = streets_pdn.nearest_pois(
    100000,             # Max distance to look for
    "Internet cafes",   # POIs to look for
    num_pois=1,         # No. of POIs to include
    include_poi_ids=True # Store POI ID
# Then add the internet cafee IDs and name
).join(
    icafes[['osmid', 'name']],
    on="poi1"
# Rename the distance from node to cafe
).rename(
    columns={1: "dist2icafe"}
)
cafe2nnode.info()
```

```
/opt/conda/lib/python3.8/site-packages/ipykernel/ipkernel.py:283: DeprecationWarning:
`should_run_async` will not call `transform_cell` automatically in the future. Please
pass the result to `transformed_cell` argument and any exception that happen during
thetransform in `preprocessing_exc_tuple` in IPython 7.17 and above.
  and should_run_async(code)
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 49985 entries, 1 to 49985
Data columns (total 5 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   dist2icafe  49985 non-null  float64
 1   poi1        49113 non-null  float64
 2   unique_id   49113 non-null  object
 3   osmid       49113 non-null  float64
 4   name        25684 non-null  object
dtypes: float64(3), object(2)
memory usage: 3.3+ MB
```

Note that, to make things easier down the line, we can link `cafe2nnode` to the cafe IDs.

And we can also link Airbnb's to nodes (3.) following a similar approach as we have seen above:

```
abbs_nnode = streets_pdn.get_node_ids(
    no_wifi.geometry.x, no_wifi.geometry.y
)
abbs_nnode.head()
```

```
/opt/conda/lib/python3.8/site-packages/ipykernel/ipkernel.py:283: DeprecationWarning:
`should_run_async` will not call `transform_cell` automatically in the future. Please
pass the result to `transformed_cell` argument and any exception that happen during
thetransform in `preprocessing_exc_tuple` in IPython 7.17 and above.
  and should_run_async(code)
```

```
26      8873
50     10906
62     41159
63     34258
221    32216
Name: node_id, dtype: int64
```

Finally, we can bring together both to find out what is the nearest internet cafe for each Airbnb (4.):

```
abb_icafe = no_wifi[
    ["geometry"]
].assign(
    nnode=abbs_nnode
).join(
    cafe2nnode,
    on="nnode"
)
abb_icafe.head()
```

```
/opt/conda/lib/python3.8/site-packages/ipykernel/ipkernel.py:283: DeprecationWarning:
`should_run_async` will not call `transform_cell` automatically in the future. Please
pass the result to `transformed_cell` argument and any exception that happen during
thetransform in `preprocessing_exc_tuple` in IPython 7.17 and above.
  and should_run_async(code)
```

	geometry	nnode	dist2icafe	poi1	unique_id	osmid
26	POINT (443128.256 4483599.841)	8873	4926.223145	9.0	node/3770327498	3.770327e+09
50	POINT (441885.677 4475916.602)	10906	1876.392944	19.0	node/6922981312	6.922981e+09
62	POINT (440439.640 4476480.771)	41159	1164.812988	17.0	node/5573414444	5.573414e+09
63	POINT (438485.311 4471714.377)	34258	1466.537964	5.0	node/2304484515	2.304485e+09
221	POINT (439941.104 4473117.914)	32216	354.268005	15.0	node/5412144560	5.412145e+09

Accessibility

```
%%time
parks = ox.geometries_from_place(
    "Madrid, Spain", tags={"leisure": "park"}
).to_crs(streets.crs)
```

```
/opt/conda/lib/python3.8/site-packages/ipykernel/ipkernel.py:283: DeprecationWarning:
`should_run_async` will not call `transform_cell` automatically in the future. Please
pass the result to `transformed_cell` argument and any exception that happen during
thetransform in `preprocessing_exc_tuple` in IPython 7.17 and above.
  and should_run_async(code)
```

```
CPU times: user 429 ms, sys: 12 ms, total: 441 ms
Wall time: 437 ms
```

How many parks are within 500m(-euclidean) of an Airbnb?

We draw a radious of 500m around each AirBnb:

```
buffers = geopandas.GeoDataFrame(
    geometry=abbs.to_crs(
        streets.crs
    ).buffer(
        500
    )
)
```

```
/opt/conda/lib/python3.8/site-packages/ipykernel/ipkernel.py:283: DeprecationWarning:
`should_run_async` will not call `transform_cell` automatically in the future. Please
pass the result to `transformed_cell` argument and any exception that happen during
thetransform in `preprocessing_exc_tuple` in IPython 7.17 and above.
  and should_run_async(code)
```

Then intersect it with the location of parks, and count by buffer (ie. Airbnb):

```
park_count = geopandas.sjoin(
    parks, buffers
).groupby(
    "index_right"
).size()
```

How many parks are within 500m(-network) of an Airbnb?

We need to approach this as a calculation *within* the network. The logic of steps thus looks like:

1. Use the aggregation module in `pandana` to count the number of parks within 500m of each node in the network
2. Extract the counts for the nodes nearest to Airbnb properties
3. Assign park counts to each Airbnb

We can set up the aggregate engine (1.). This involves three steps:

a. Obtain nearest node for each park

```
parks_nnode = streets_pdn.get_node_ids(
    parks.centroid.x, parks.centroid.y
)
```

```
/opt/conda/lib/python3.8/site-packages/ipykernel/ipkernel.py:283: DeprecationWarning:
`should_run_async` will not call `transform_cell` automatically in the future. Please
pass the result to `transformed_cell` argument and any exception that happen during
thetransform in `preprocessing_exc_tuple` in IPython 7.17 and above.
  and should_run_async(code)
```

b. Insert the parks' nearest node through `set` so it can be "aggregated"

```
streets_pdn.set(
    parks_nnode, name="Parks"
)
```

```
/opt/conda/lib/python3.8/site-packages/ipykernel/ipkernel.py:283: DeprecationWarning:
`should_run_async` will not call `transform_cell` automatically in the future. Please
pass the result to `transformed_cell` argument and any exception that happen during
thetransform in `preprocessing_exc_tuple` in IPython 7.17 and above.
  and should_run_async(code)
```

c. "Aggregate" for a distance of 500m, effectively counting the number of parks within 500m of each node

```
parks_by_node = streets_pdn.aggregate(
    distance=500, type="count", name="Parks"
)
parks_by_node.head()
```

```
/opt/conda/lib/python3.8/site-packages/ipykernel/ipkernel.py:283: DeprecationWarning:
`should_run_async` will not call `transform_cell` automatically in the future. Please
pass the result to `transformed_cell` argument and any exception that happen during
thetransform in `preprocessing_exc_tuple` in IPython 7.17 and above.
  and should_run_async(code)
```

```
nodeID
1    5.0
2    5.0
3    6.0
4    8.0
5    1.0
dtype: float64
```

At this point, we have the number of parks within 500m of *every* node in the network. To identify those that correspond to each Airbnb (3.), we query those nodes:

```
abbs_xys = abbs.to_crs(streets.crs).geometry
abbs_nnode = streets_pdn.get_node_ids(
    abbs_xys.x, abbs_xys.y
)
```

And use the list to asign the count of the nearest node to each Airbnb:

```
park_count_network = abbs_nnode.map(
    parks_by_node
)
```

```
/opt/conda/lib/python3.8/site-packages/ipykernel/ipkernel.py:283: DeprecationWarning:
`should_run_async` will not call `transform_cell` automatically in the future. Please
pass the result to `transformed_cell` argument and any exception that happen during
thetransform in `preprocessing_exc_tuple` in IPython 7.17 and above.
  and should_run_async(code)
```

For which areas do both differ most?

We can compare the two counts above to explore to what extent the street layout is constraining access to nearby parks.

```
park_comp = geopandas.GeoDataFrame(
    {
        "Euclidean": park_count,
        "Network": park_count_network
    },
    geometry=abbs.geometry,
    crs=abbs.crs
)
```

```
park_comp.plot.scatter("Euclidean", "Network");
```

And, geographically:

Note there are a few cases where there are more network counts than Euclidean. These are due to the slight inaccuracies introduced by calculating network distances from nodes rather than the locations themselves

Moving along surfaces

Local files | Online read

Assuming you have the file locally on the path `../data/`:

```
friction_walk = rioxarray.open_rasterio(
    "../data/cambodia_2020_walking_friction_surface.tif"
)
friction_motor = rioxarray.open_rasterio(
    "../data/cambodia_2020_motorized_friction_surface.tif"
)
cities = geopandas.read_file("../data/cambodian_cities.geojson")
```

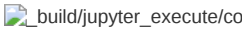
Shortest-path routing

```
import osmnx as ox
```

```
/opt/conda/lib/python3.8/site-packages/ipykernel/ipkernel.py:283: DeprecationWarning:
`should_run_async` will not call `transform_cell` automatically in the future. Please
pass the result to `transformed_cell` argument and any exception that happen during
thetransform in `preprocessing_exc_tuple` in IPython 7.17 and above.
  and should_run_async(code)
```

```
main_roads = ox.geometries_from_place(
    "Cambodia", tags={"highway": "trunk"}
)
```

From the geo-table of roads, we can generate a surface that has a value of 1 on cells where a road crosses, and 0 otherwise (*rasterisation*).



```
from geocube.api.core import make_geocube

roads_surface = make_geocube(
    main_roads.assign(
        one=1
    ).to_crs(epsg=3148),
    measurements=["one"],
    resolution=(500, 500)
)["one"]
```

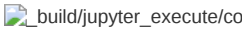
```
/opt/conda/lib/python3.8/site-packages/ipykernel/ipkernel.py:283: DeprecationWarning:
`should_run_async` will not call `transform_cell` automatically in the future. Please
pass the result to `transformed_cell` argument and any exception that happen during
thetransform in `preprocessing_exc_tuple` in IPython 7.17 and above.
  and should_run_async(code)
```

Now we turn it into a binary mask:

```
road_mask = xarray.where(
    roads_surface.isnull(), 0, 1
)
```

```
/opt/conda/lib/python3.8/site-packages/ipykernel/ipkernel.py:283: DeprecationWarning:
`should_run_async` will not call `transform_cell` automatically in the future. Please
pass the result to `transformed_cell` argument and any exception that happen during
thetransform in `preprocessing_exc_tuple` in IPython 7.17 and above.
  and should_run_async(code)
```

```
road_mask
```



xarray.DataArray 'one' (**y**: 745, **x**: 847)

```
array([[0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       ...,
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0]])
```

▾ Coordinates:

y	(y)	float64	1.167e+06 1.168e+06 ... 1.539e+06	
x	(x)	float64	2.348e+05 2.352e+05 ... 6.578e+05	
spatial_ref	()	int64	0	

▸ Attributes: (0)

Then we can generate the routing from, say Phnom Penh to Poipet, using `xarray-spatial`'s A* algorithm:

```
a_star_search?
```

```
Object `a_star_search` not found.
```

```
from xrspatial import a_star_search

# Pull out starting point
start = cities.query(
    "UC_NM_MN == 'Phnom Penh'"
).to_crs(
    epsg=3148
).squeeze().geometry

# Pull out ending point
end = cities.query(
    "UC_NM_MN == 'Poipet'"
).to_crs(
    epsg=3148
).squeeze().geometry

# Routing
route = a_star_search(
    road_mask,          # Road surface
    (start.x, start.y), # Starting point
    (end.x, end.y),     # Destination point
    barriers=[0],       # Cell values that cannot be crossed
    snap_start=True,    # Snap starting point to valid cells
    snap_goal=True      # Snap ending point to valid cells
)
```

```
/opt/conda/lib/python3.8/site-packages/numba/cuda/api.py:112: DeprecationWarning:
`np.float` is a deprecated alias for the builtin `float`. To silence this warning, use
`float` by itself. Doing this will not modify any behavior and is safe. If you
specifically wanted the numpy scalar type, use `np.float64` here.
Deprecated in NumPy 1.20; for more details and guidance:
https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
  def device_array(shape, dtype=np.float, strides=None, order='C', stream=0):
/opt/conda/lib/python3.8/site-packages/numba/cuda/api.py:124: DeprecationWarning:
`np.float` is a deprecated alias for the builtin `float`. To silence this warning, use
`float` by itself. Doing this will not modify any behavior and is safe. If you
specifically wanted the numpy scalar type, use `np.float64` here.
Deprecated in NumPy 1.20; for more details and guidance:
https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
  def managed_array(shape, dtype=np.float, strides=None, order='C', stream=0,
/opt/conda/lib/python3.8/site-packages/numba/cuda/api.py:154: DeprecationWarning:
`np.float` is a deprecated alias for the builtin `float`. To silence this warning, use
`float` by itself. Doing this will not modify any behavior and is safe. If you
specifically wanted the numpy scalar type, use `np.float64` here.
Deprecated in NumPy 1.20; for more details and guidance:
https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
  def pinned_array(shape, dtype=np.float, strides=None, order='C'):
/opt/conda/lib/python3.8/site-packages/numba/cuda/api.py:170: DeprecationWarning:
`np.float` is a deprecated alias for the builtin `float`. To silence this warning, use
`float` by itself. Doing this will not modify any behavior and is safe. If you
specifically wanted the numpy scalar type, use `np.float64` here.
Deprecated in NumPy 1.20; for more details and guidance:
https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
  def mapped_array(shape, dtype=np.float, strides=None, order='C', stream=0,
/opt/conda/lib/python3.8/site-packages/numba/cuda/simulator/cudadrv/devicearray.py:313:
DeprecationWarning: `np.float` is a deprecated alias for the builtin `float`. To silence
this warning, use `float` by itself. Doing this will not modify any behavior and is safe.
If you specifically wanted the numpy scalar type, use `np.float64` here.
Deprecated in NumPy 1.20; for more details and guidance:
https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
  def pinned_array(shape, dtype=np.float, strides=None, order='C'):
/opt/conda/lib/python3.8/site-packages/numba/cuda/simulator/cudadrv/devicearray.py:317:
DeprecationWarning: `np.float` is a deprecated alias for the builtin `float`. To silence
this warning, use `float` by itself. Doing this will not modify any behavior and is safe.
If you specifically wanted the numpy scalar type, use `np.float64` here.
Deprecated in NumPy 1.20; for more details and guidance:
https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
  def managed_array(shape, dtype=np.float, strides=None, order='C'):
```

```
route
```

xarray.DataArray (**y**: 745, **x**: 847)

```
array([[nan, nan, nan, ..., nan, nan, nan],
       [nan, nan, nan, ..., nan, nan, nan],
       [nan, nan, nan, ..., nan, nan, nan],
       ...,
       [nan, nan, nan, ..., nan, nan, nan],
       [nan, nan, nan, ..., nan, nan, nan],
       [nan, nan, nan, ..., nan, nan, nan]])
```

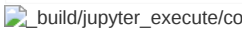
▾ Coordinates:

y	(y)	float64	1.167e+06 1.168e+06 ... 1.539e+06	
x	(x)	float64	2.348e+05 2.352e+05 ... 6.578e+05	
spatial_ref	()	int64	0	

▸ Attributes: (0)

```
/opt/conda/lib/python3
DeprecationWarning: `s
`transform_cell` autor
result to `transformed
happen during thetrans
7.17 and above.
  and should_run_asyn
```

```
<matplotlib.collectio
```



And we can turn the route surface into a line that connects the pixels in the route:

```
from shapely.geometry import LineString

route_line = LineString(
    route.to_series().dropna().reset_index().rename(
        columns={0: "order"}
    ).sort_values(
        "order"
    )[
        ["x", "y"]
    ].values
)
route_line
```

Continuous surfaces

We can use a continuous surface instead of a road network to set the surface cells where it is allowed to “pass”. For example, with the motorised surface, we can explore the degree of friction:

We can determine that, above 0.01, a cell is not available for routing. Using this, we can build a mask similar as above that can be used for generating routes:

```
# Set surface up
surface = friction_motor.where(
    friction_motor!=friction_motor.rio.nodata
).rio.reproject(3148).sel(
    band=1
)
# Swap nodata value for N/A
surface = surface.where(
    surface!=surface.rio.nodata
)
# Turn into binary mask
# Valid for routing if < 0.01
surface = xarray.where(
    surface < 0.01, 1, 0
)
```

```
/opt/conda/lib/python3.8/site-packages/ipykernel/ipkernel.py:283: DeprecationWarning:
`should_run_async` will not call `transform_cell` automatically in the future. Please
pass the result to `transformed_cell` argument and any exception that happen during
thetransform in `preprocessing_exc_tuple` in IPython 7.17 and above.
  and should_run_async(code)
```

```
# Swap order of Y coordinate
surface.coords["y"] = surface.coords["y"][::-1]

# Pull out starting point
start = cities.query(
    "UC_NM_MN == 'Siem Reap'"
).to_crs(
    epsg=3148
).squeeze().geometry

# Pull out ending point
end = cities.query(
    "UC_NM_MN == 'Sihanoukville'"
).to_crs(
    epsg=3148
).squeeze().geometry

# Routing
route = a_star_search(
    surface,            # (Reprojected) road surface
    (start.x, start.y), # Starting point
    (end.x, end.y),     # Destination point
    barriers=[0],       # Cell values that cannot be crossed
    snap_start=True,    # Snap starting point to valid cells
    snap_goal=True      # Snap ending point to valid cells
)

# Swap order of Y coordinate back
surface.coords["y"] = surface.coords["y"][::-1]

# Parse route into line
route_line = LineString(
    route.to_series().dropna().reset_index().rename(
        columns={0: "order"}
    ).sort_values(
        "order"
    )[
        ["x", "y"]
    ].values
)
route_line = geopandas.GeoSeries([route_line])
```

```
/opt/conda/lib/python3.8/site-packages/ipykernel/ipkernel.py:283: DeprecationWarning:
`should_run_async` will not call `transform_cell` automatically in the future. Please
pass the result to `transformed_cell` argument and any exception that happen during
thetransform in `preprocessing_exc_tuple` in IPython 7.17 and above.
  and should_run_async(code)
```


```
/opt/conda/lib/python3.8/site-packages/ipykernel/ipkernel.py:283: DeprecationWarning:
`should_run_async` will not call `transform_cell` automatically in the future. Please
pass the result to `transformed_cell` argument and any exception that happen during
thetransform in `preprocessing_exc_tuple` in IPython 7.17 and above.
  and should_run_async(code)
```


```
f, ax = plt.subplots(1, figsize=(30, 30))
surface.plot(add_colorbar=False)
route_line.plot(
    ax=ax, color="red"
)
cities.to_crs(epsg=3148).plot(
    ax=ax, label="Main cities"
)
plt.show()
```


```
/opt/conda/lib/python3.8/site-packages/ipykernel/ipkernel.py:283: DeprecationWarning:
`should_run_async` will not call `transform_cell` automatically in the future. Please
pass the result to `transformed_cell` argument and any exception that happen during
thetransform in `preprocessing_exc_tuple` in IPython 7.17 and above.
  and should_run_async(code)
```


Over to you…

- Recreate weighted routing using the linearity of street segments. How can you go from A to B avoiding long streets?
- Explore the differences in the output of weighted routing if you change the weight between distance and the additional variable.
- Explore the difference in routing when you vary the friction threshold to allow usage of a cell

🐾 Next steps

If you found the content in this block useful, the following resources represent some suggestions on where to go next:

- The **`pandana`** tutorial and documentation are excellent places to get a more detailed and comprehensive view into the functionality of the library
- More about `xarray-spatial`, the library that provides geospatial techniques on top of surfaces is available at the project's documentation

Datasets

This section covers the datasets required to run the course interactively. For archival reasons, all of those listed here have been mirrored in the repository for this course so, if you have downloaded the course, you already have a local copy of them.

Madrid

Airbnb properties

> **Source**
>
> This dataset has been sourced from the course "Spatial Modelling for Data Scientists". The file imported here corresponds to the **`v0.1.0`** version.

This dataset contains a pre-processed set of properties advertised on the AirBnb website within the region of Madrid (Spain), together with house characteristics.

- 🗃️ Data file **`madrid_abb.gpkg`**
- 🤖 Code used to generate the file **`[URL]`**
- ℹ️ Furhter information **`[URL]`**


This dataset is licensed under a CC0 1.0 Universal Public Domain Dedication.

Airbnb neighbourhoods

> **ℹ Source**
>
> This dataset has been directly sourced from the website Inside Airbnb. The file was imported on February 10th 2021.

This dataset contains neighbourhood boundaries for the city of Madrid, as provided by Inside Airbnb.

- 🗃️ Data file **`neighbourhoods.geojson`**
- ℹ️ Furhter information **`[URL]`**


This dataset is licensed under a CC0 1.0 Universal Public Domain Dedication.

Arturo

This dataset contains the street layout of Madrid as well as scores of habitability, where available, associated with street segments. The data originate from the Arturo Project, by 300,000Km/s, and the available file here is a slimmed down version of their official street layout distributed by the project.

- 🗃️ Data file **`arturo_streets.gpkg`**
- 🤖 Code used to generate the file **`[Page]`**
- ℹ️ Furhter information **`[URL]`**


This dataset is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

Sentinel 2 - 120m mosaic

This dataset contains four scenes for the region of Madrid (Spain) extracted from the Digital Twin Sandbox Sentinel-2 collection, by the SentinelHub. Each scene corresponds to the following dates in 2019:

- January 1st
- April 1st
- July 10th
- November 17th

Each scene includes red, green, blue and near-infrared bands.

- 🗃️ Data files (**`Jan_1st`**, **`Apr_1st`**, **`Jul_10th`**, **`Nov_27th`**)
- 🤖 Code used to generate the file **`[Page]`**
- ℹ️ Furhter information **`[URL]`**


This dataset is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

Sentinel 2 - 10m GHS composite

This dataset contains a scene for the region of Madrid (Spain) extracted from the GHS Composite S2, by the European Commission.

- 🗃️ Data file **`madrid_scene_s2_10_tc.tif`**
- 🤖 Code used to generate the file **`[Page]`**
- ℹ️ Furhter information **`[URL]`**

Cambodia

Pollution

Surface with \(NO_2\) measurements (tropospheric column) information attached from Sentinel 5.

- 🗃️ Data file **`cambodia_s5_no2.tif`**
- 🤖 Code used to generate the file **`[Page]`**
- ℹ️ Furhter information **`[URL]`**

Friction surfaces

This dataset is an extraction of the following two data products by Weiss et al. (2020) [WNVR+20] and distributed through the Malaria Atlas Project:

- Global friction surface enumerating land-based travel walking-only speed without access to motorized transport for a nominal year 2019 (Minutes required to travel one metre)
- Global friction surface enumerating land-based travel speed with access to motorized transport for a nominal year 2019 (Minutes required to travel one metre)

Each is provided on a separate file.

- 🗃️ Data files (**`Motorized`** and **`Walking`**)
- 🤖 Code used to generate the file **`[Page]`**
- ℹ️ Furhter information **`[URL]`**

Regional aggregates

> **Source**
>
> This dataset relies on boundaries from the Humanitarian Data Exchange. The file is provided by the World Food Programme through the Humanitarian Data Exchange and was accessed on February 15th 2021.

Pollution and friction aggregated at Level 2 (municipality) administrative boundaries for Cambodia.

- 🗃️ Data file **`cambodia_regional.gpkg`**
- 🤖 Code used to generate the file **`[Page]`**

Cambodian cities

Extract from the Urban Centre Database (UCDB), version 1.2, of the centroid for Cambodian cities.

- 🗃️ Data file `cambodian_cities.geojson`
- 🤖 Code used to generate the file `[Page]`
- ℹ️ Furhter information `[URL]`

Further Resources

If this course is successful, it will leave you wanting to learn more about using Python for (Geographic) Data Science. See below a few resources that are good “next steps”.

Courses

- The “Automating GIS processes”, by Vuokko Heikinheimo and Henrikki Tenkanen is a great overview of GIS with a modern Python stack:

> https://autogis-site.readthedocs.io/

- The “GDS Course” by Dani Arribas-Bel [AB19] is an introductory level overview of Geographic Data Science, including notebooks, slides and video clips.

> https://darribas.org/gds_course

Books

- *“Python for Geographic Data Analysis”*, by Henrikki Tenkanen, Vuokko Heikinheimo and David Whipp:

> https://pythongis.org/

- *“Geographic Data Science in Python”*, by Sergio J. Rey, Dani Arribas-Bel and Levi J. Wolf:

> https://geographicdata.science

Bibliography

ASP19 Jennings Anderson, Dipto Sarkar, and Leysia Palen. Corporate editors in the evolving landscape of openstreetmap. *ISPRS International Journal of Geo-Information*, 8(5):232, 2019.

AB19 Dani Arribas-Bel. A course on geographic data science. *The Journal of Open Source Education*, 2019. doi:https://doi.org/10.21105/jose.00042.

Boe17 Geoff Boeing. OSMnx: New Methods for Acquiring, Constructing, Analyzing, and Visualizing Complex Street Networks. *Computers, Environment and Urban Systems*, 2017. doi:10.1016/j.compenvurbsys.2017.05.004.

Boe20a Geoff Boeing. Exploring urban form through openstreetmap data: a visual introduction. *arXiv preprint arXiv:2008.12142*, 2020.

Boe20b Geoff Boeing. Off the grid… and back again? the recent evolution of american street network planning and design. *Journal of the American Planning Association*, pages 1–15, 2020.

BAB20 Geoff Boeing and Dani Arribas-Bel. Gis and computational notebooks. In John P. Wilson, editor, *The Geographic Information Science & Technology Body of Knowledge*. UCGIS, 2020.

Bre15 Cynthia Brewer. *Designing better Maps: A Guide for GIS users*. ESRI press, 2015.

Fle19 Martin Fleischmann. Momepy: urban morphology measuring toolkit. *Journal of Open Source Software*, 4(43):1807, 2019.

McK12 Wes McKinney. *Python for data analysis: Data wrangling with Pandas, NumPy, and IPython*. O'Reilly Media, Inc., 2012.

RABWng Sergio J. Rey, Daniel Arribas-Bel, and Levi J. Wolf. *Geographic Data Science with PySAL and the PyData stack*. CRC press, forthcoming.

RBZ+19 Adam Rule, Amanda Birmingham, Cristal Zuniga, Ilkay Altintas, Shih-Cheng Huang, Rob Knight, Niema Moshiri, Mai H Nguyen, Sara Brin Rosenthal, Fernando Pérez, and others. Ten simple rules for writing and sharing computational analyses in jupyter notebooks. *PLoS Comput Biol*, 2019. doi:https://doi.org/10.1371/journal.pcbi.1007007.

SAB19 Alex Singleton and Daniel Arribas-Bel. Geographic data science. *Geographical Analysis*, 2019.

WNVR+20 DJ Weiss, A Nelson, CA Vargas-Ruiz, K Gligorić, S Bavadekar, E Gabrilovich, A Bertozzi-Villa, J Rozier, HS Gibson, T Shekel, and others. Global maps of travel time to healthcare facilities. *Nature Medicine*, 26(12):1835–1838, 2020.

By Dani Arribas-Bel & Diego Puga