Lecture 7: Introduction to Spatial Data

Contents

Lecture Outline

Lecture Learning Objectives

- Describe the difference between vector and raster data.
- Load vector data into `geopandas`.
- Plot vector data using the `geopandas` method `.plot()`.
- Wrangle vector data using `geopandas` functions, methods, and attributes like `gpd.sjoin()`, `gpd.clip()`, `.length()`, `.buffer()`, etc.
- Import data from OpenStreetMap using `osmnx`.
- Read raster data with `rasterio`.
- Describe at a high level why coordinate reference systems (CRS) are important and identify the CRS of a `geopandas` object using the `.crs` attribute and reproject it to another CRS using the `.to_crs()` method.

Imports

```
import rasterio
import keplergl
import numpy as np
import osmnx as ox
import pandas as pd
import geopandas as gpd
import matplotlib.pyplot as plt
import folium
import warnings
import matplotlib
import mapclassify
import json
import base64
import IPython


plt.style.use('ggplot')
plt.rcParams.update({'font.size': 16, 'axes.labelweight': 'bold', 'figure.figs
```

1.Introduction to spatial data

- Spatial data (or “geospatial” data) are data with location information

- Just as temporal data (time series) contained temporal dependence, spatial data looks to understand, leverage, and model spatial dependence
- We'll be focusing on three main tasks related to spatial data:
    1. Wrangling
    2. Visualizing
    3. Modelling
- There are two main ways of representing spatial data in computers:
    1. Vector format
    2. Raster format
- To get you excited, here's a cool 3D map of UBC that we'll build together next lecture:

```
warnings.filterwarnings("ignore")

ubc = (ox.features_from_place("University of British Columbia, Canada", tags={
         .loc[:, ["geometry"]]                # just keep the geometry column
        # .query("geometry.type == 'Polygon'")
         .assign(Label="Building Footprints")  # assign a label for later use
         .reset_index(drop=True)              # reset to 0 integer indexing
      )
```

```
warnings.filterwarnings("ignore")

# # load a customized keplergl config
import json
ubc_config = json.load(open('keplergl/ubc_config.json', 'r'))

# # Import ubc height data from shapefile
ubc_bldg_heights = gpd.read_file("data-spatial/ubc-building-footprints-2009")

# # Join ubc height with ubc geom data from osmnx
ubc_bldg_heights = (gpd.sjoin(ubc.drop(columns='Label'), ubc_bldg_heights[["hg
                        .drop(columns="index_right")
                        .rename(columns={"hgt_agl": "Height"})
                        .reset_index()
                        .dissolve(by="index", aggfunc="mean")  # dissolve is li
                    )
# ubc_bldg_heights
# # # Create keplergl plot
ubc_height_map = keplergl.KeplerGl(height=500,config=ubc_config)
ubc_height_map.add_data(data=ubc_bldg_heights.copy(), name="Building heights")

ubc_height_map = keplergl.KeplerGl(height=500,config=ubc_config)
ubc_height_map.add_data(data=ubc_bldg_heights.copy(), name="Building heights")

# ubc_height_map
```

```
User Guide: https://docs.kepler.gl/docs/keplergl-jupyter
User Guide: https://docs.kepler.gl/docs/keplergl-jupyter
```

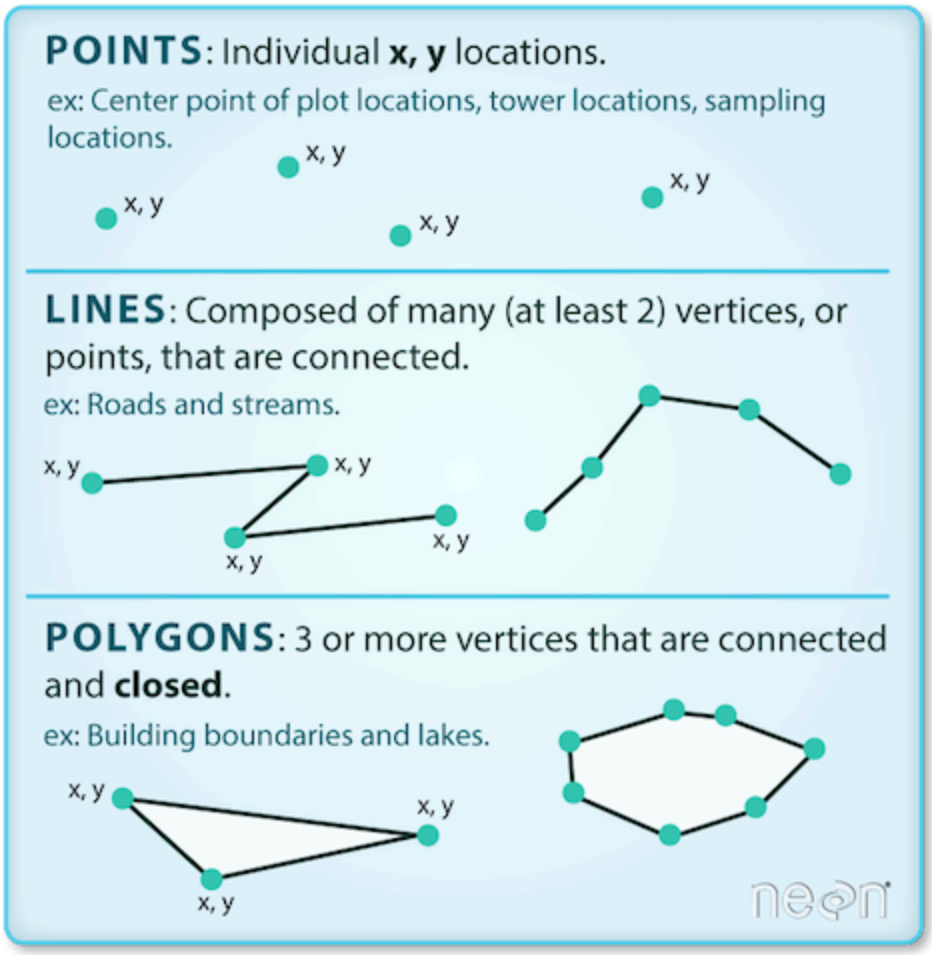
```
# # # # render keplergl in jupyter lab
# orig_html = str(ubc_height_map._repr_html_(),'utf-8')
# b64d_html = base64.b64encode(orig_html.encode('utf-8')).decode('utf-8')
# framed_html = f'<iframe src="data:text/html;base64,{b64d_html}" style="width
# IPython.display.HTML(framed_html)
```

2. Working with vector data

2.1. Vector data

- Vector data is a very intuitive and common spatial data format and the one we'll focus on most this week

- Vector data is simply a collection of discrete locations (`(x, y)` values) called "vertices" that define one of three shapes:
    1. **Point**: a single `(x, y)` point. Like the location of your house.
    2. **Line**: two or more connected `(x, y)` points. Like a road.
    3. **Polygon**: three or more `(x, y)` points connected and closed. Like a lake, or the border of a country.



> Source: National Ecological Observatory Network.

- Vector data is most commonly stored in a "shapefile"
- A shapefile is actually composed of 3 required files with the same prefix (here, `spatial-data`) but different extensions:
    1. `spatial-data.shp`: main file that stores records of each shape geometries
    2. `spatial-data.shx`: index of how the geometries in the main file relate to one-another

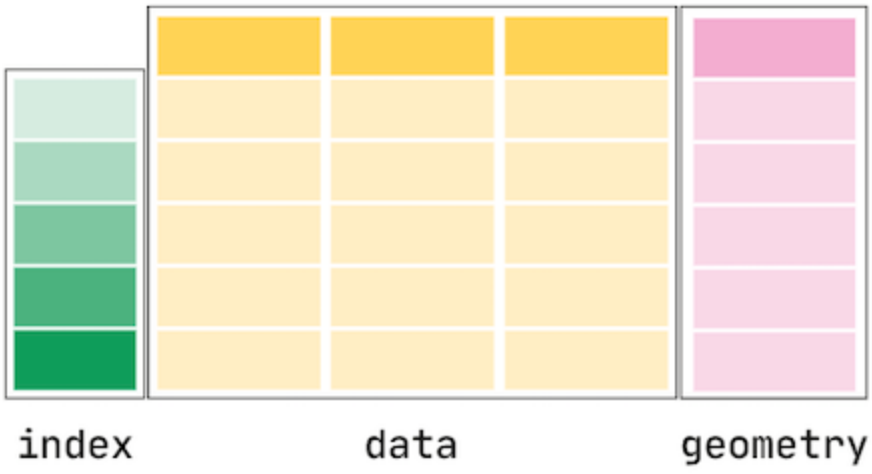
3. `spatial-data.dbf`: attributes of each record

- There are other optional files that may also be part of a shapefile but we won't worry about them for now
- Each shapefile can only contain one type of shape. For example, the descriptions for a house (point), a road (line), and a postal code area (polygon) would be stored in three separate shapefiles.

> There are other file-types for storing vector data too like `geojson`. These files can generally be imported into Python using the same methods and packages we use below.

2.2. Geopandas

- geopandas is the main library we'll be using to work with vector data in Python
- It's built off `shapely` (which is *the* Python library for working with geometric objects in Python) and `pandas`
- Like `pandas`, `geopandas` provides two key classes for spatial data manipulation:
  1. `GeoSeries`: just like a `pandas` series but stores geometries like points, lines, polygons (we'll see that shortly)
  2. `GeoDataFrame`: just like a `pandas` dataframe with one or more columns of regular series and one or more columns of `geoseries`.
- Along with those new classes, we also have a variety of geopstial wrangling methods which we'll explore in this lecture
- Here's a schematic of a `GeoDataFrame`:



> Source: GeoPandas Documentation.

- We usually import `geopandas` using the alias `gpd`:

```
import geopandas as gpd
```

2.3. Loading data

- Let's take a look at loading in a shapefile to a `GeoDataFrame` now
- I downloaded a shapefile of Candian provinces from statcan in a `data-spatial/provinces` shapefile that looks like this:

```
provinces
├── provinces.dbf
├── provinces.shp
├── provinces.shx
└── provinces.prj  # this contains projection information which I'll dicuss la
```

- We can read our shapefile using `gpd.read_file()`:

```
provinces = gpd.read_file("data-spatial/provinces")  # note that I point to th
provinces = provinces.to_crs("EPSG:4326")    # I'll explain this later, I'm co
provinces
```

	PRUID	PRNAME	PRENAME	PRFNAME	PREABBR	PRFABBR	
0	10	Newfoundland and Labrador / Terre-Neuve-et-Lab...	Newfoundland and Labrador	Terre-Neuve-et-Labrador	N.L.	T.-N.-L.	MU
1	11	Prince Edward Island / Île-du-Prince-Édouard	Prince Edward Island	Île-du-Prince-Édouard	P.E.I.	Î.-P.-É.	MU
2	12	Nova Scotia / Nouvelle-Écosse	Nova Scotia	Nouvelle-Écosse	N.S.	N.-É.	MU
3	13	New Brunswick / Nouveau-Brunswick	New Brunswick	Nouveau-Brunswick	N.B.	N.-B.	MU
4	24	Quebec / Québec	Quebec	Québec	Que.	Qc	MU
5	35	Ontario	Ontario	Ontario	Ont.	Ont.	MU
6	46	Manitoba	Manitoba	Manitoba	Man.	Man.	MU
7	47	Saskatchewan	Saskatchewan	Saskatchewan	Sask.	Sask.	
8	48	Alberta	Alberta	Alberta	Alta.	Alb.	((- -,
9	59	British Columbia / Colombie-Britannique	British Columbia	Colombie-Britannique	B.C.	C.-B.	MU -1

	PRUID	PRNAME	PRENAME	PRFNAME	PREABBR	PRFABBR	
10	60	Yukon	Yukon	Yukon	Y.T.	Yn	MU
11	61	Northwest Territories / Territoires du Nord-Ouest	Northwest Territories	Territoires du Nord-Ouest	N.W.T.	T.N.-O.	MU
12	62	Nunavut	Nunavut	Nunavut	Nvt.	Nt	MU

```
type(provinces)
```

```
geopandas.geodataframe.GeoDataFrame
```

- Because `geopandas` is built off `pandas`, our `GeoDataFrame` inherits most of the same functionality as a regular dataframe
- For example, let's try the `.info()` method:

```
provinces.info()
```

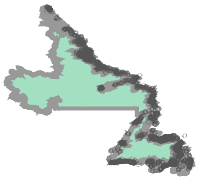
```
<class 'geopandas.geodataframe.GeoDataFrame'>
RangeIndex: 13 entries, 0 to 12
Data columns (total 7 columns):
 #   Column    Non-Null Count  Dtype   
---  ------    --------------  -----   
 0   PRUID     13 non-null     object  
 1   PRNAME    13 non-null     object  
 2   PRENAME   13 non-null     object  
 3   PRFNAME   13 non-null     object  
 4   PREABBR   13 non-null     object  
 5   PRFABBR   13 non-null     object  
 6   geometry  13 non-null     geometry
dtypes: geometry(1), object(6)
memory usage: 856.0+ bytes
```

- Note we have 5 columns of dtype "object" which typically means "strings" in `pandas`, and we have our one "geometry" column which contains vector data - polygons in this

case. Well `MULTIPOLYGON`s which just means multiple polygons together, for example British Columbia has a lot of little islands, so to make a boundary of it, we need multiple polygons

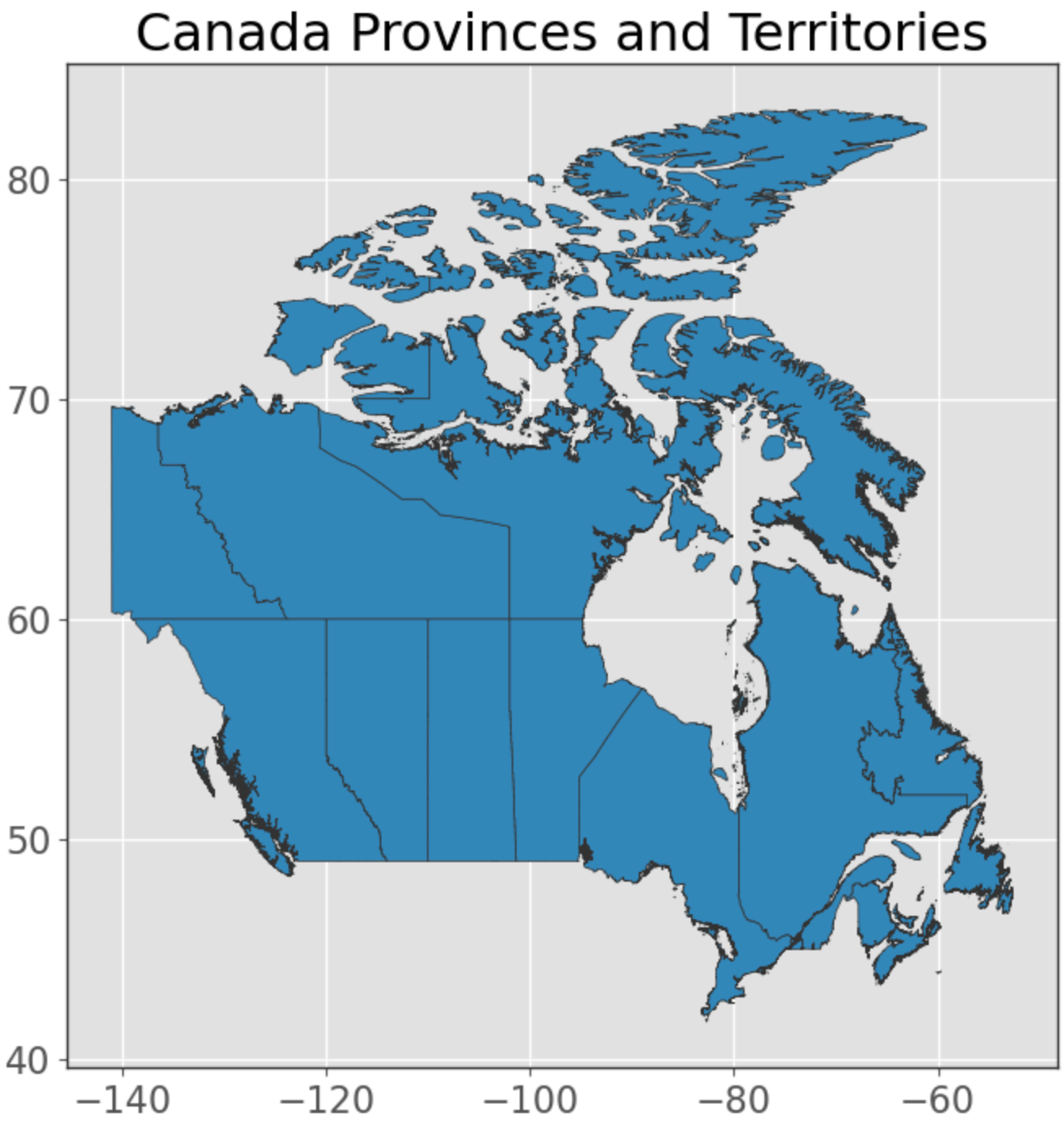
- We can easily look at one of these shapes in Jupyter:

```
provinces["geometry"][0]
```



- I think that's Newfoundland and Labrador but it's hard to see and not very useful, a proper plot would be better!
- Luckily, `geopandas` has built-in plotting functionality (just like `pandas`) which is useful for making quick plots to verify your data:

```
provinces.plot(edgecolor="0.2", figsize=(10, 8))
plt.title("Canada Provinces and Territories");
```

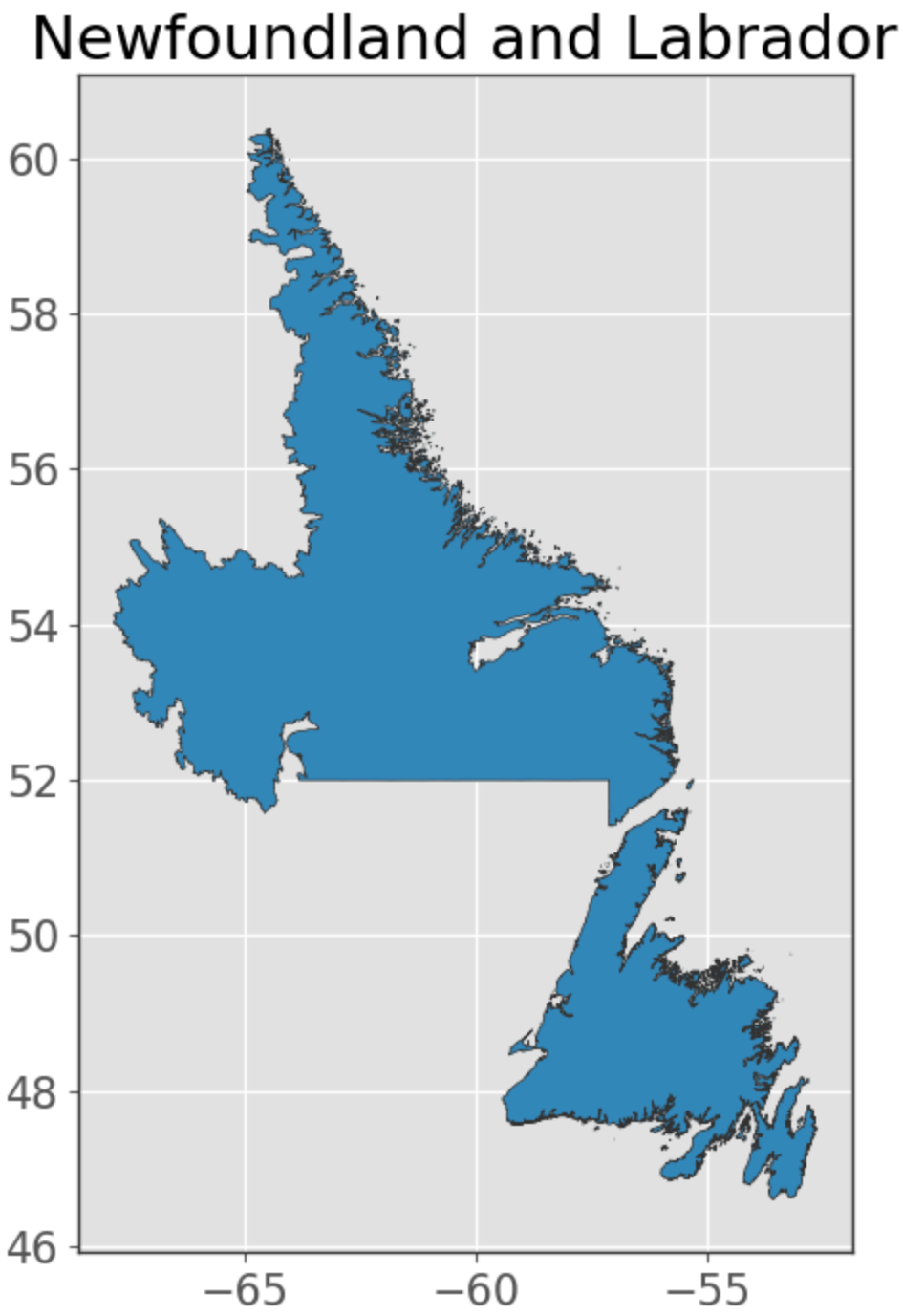


- That looks like Canada to me!
- We can also index our `GeoDataFrame` just like a regular dataframe:

```
provinces.iloc[[0]]
```

	PRUID	PRNAME	PRENAME	PRFNAME	PREABBR	PRFABBR	g
0	10	Newfoundland and Labrador / Terre-Neuve-et-Lab...	Newfoundland and Labrador	Terre-Neuve-et-Labrador	N.L.	T.-N.-L.	MULTIP (((-5 5 -57

```
name = provinces.iloc[0]["PRENAME"]
provinces.iloc[[0]].plot(edgecolor="0.2", figsize=(10, 8))
plt.title(name);
```



- Let's filter our dataframe for only British Columbia using the helpful `pandas` method `.query()`:

```
province = "British Columbia"
bc = provinces.query("PRENAME == @province").copy()
bc
```

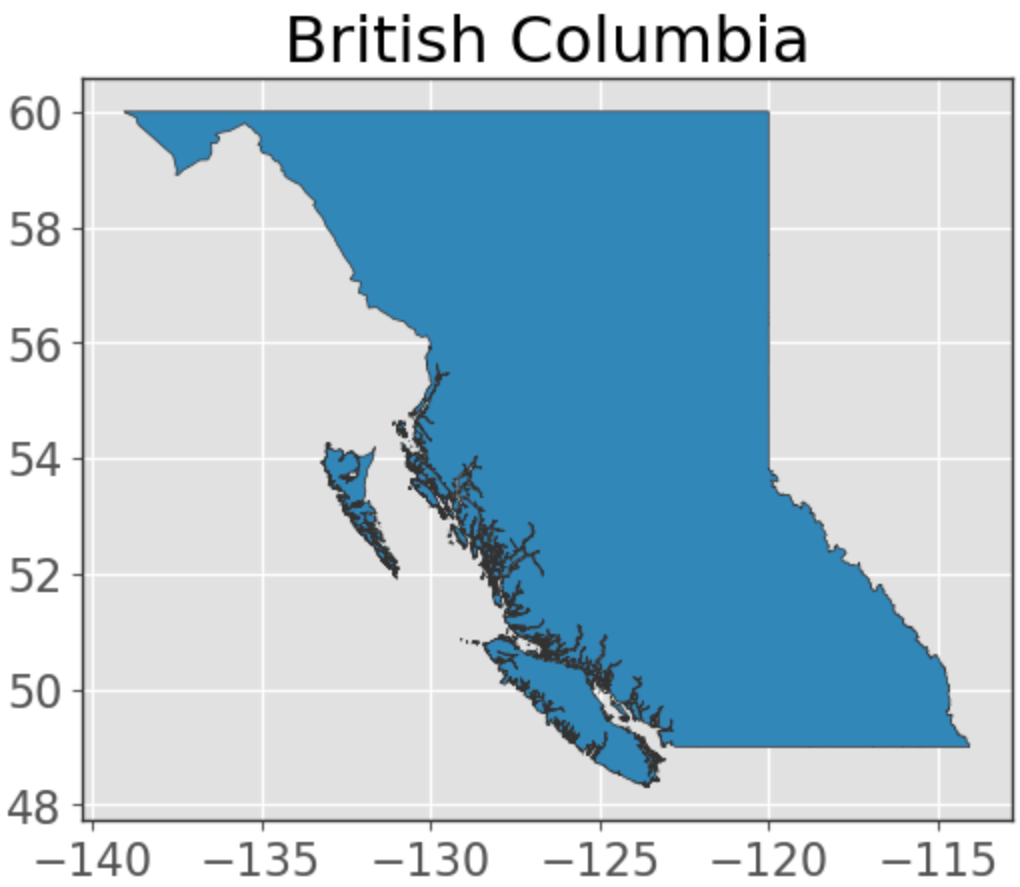
	PRUID	PRNAME	PRENAME	PRFNAME	PREABBR	PRFABBR	geome
9	59	British Columbia / Colombie-Britannique	British Columbia	Colombie-Britannique	B.C.	C.-B.	MULTIPOLYG (((-13 60.000 -135.3875 6(

- Now, let's do some simple manipulation to clean up our dataframe. All these methods should be familiar to you:

```
bc = (bc.loc[:, ["PRENAME", "geometry"]]
        .rename(columns={"PRNAME": "Province"})
        .reset_index(drop=True)
      )
bc
```

	PRENAME	geometry
0	British Columbia	MULTIPOLYGON (((-135.4 60.00006, -135.3875 60....

```
bc.plot(edgecolor="0.2")
plt.title("British Columbia");
```



2.3. Making data

- Typically, you'll be loading data from a file like we did above (or using an API as we'll do later in this lecture)
- But we can also create our own vector data
- Let's create some "points" for BC's biggest cities in a regular dataframe:

```
cities = pd.DataFrame(
    {"City": ["Vancouver", "Victoria", "Kelowna"],
     "Population": [2_463_431, 397_237, 222_162],
     "Latitude": [49.260833, 48.428333, 49.888056],
     "Longitude": [-123.113889, -123.364722, -119.495556],
    }
)
cities
```

	City	Population	Latitude	Longitude
0	Vancouver	2463431	49.260833	-123.113889
1	Victoria	397237	48.428333	-123.364722
2	Kelowna	222162	49.888056	-119.495556

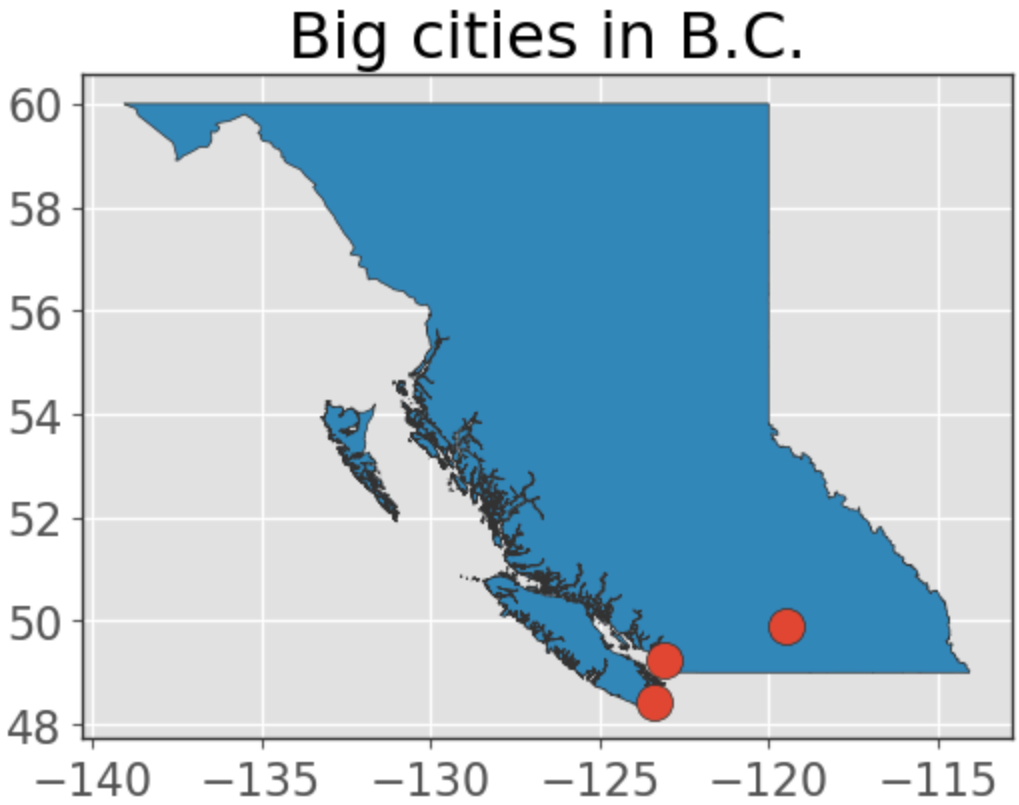
- We can coerce that data into a `GeoDataFrame` using `gpd.GeoDataFrame()` and by using the function `gpd.points_from_xy()` to change our "Latitude" and "Longitude" columns to geometries:

```
cities = gpd.GeoDataFrame(cities,
                          crs="EPSG:4326",  # I'll talk about this later
                          geometry=gpd.points_from_xy(cities["Longitude"], cit
cities
```

	City	Population	Latitude	Longitude	geometry
0	Vancouver	2463431	49.260833	-123.113889	POINT (-123.11389 49.26083)
1	Victoria	397237	48.428333	-123.364722	POINT (-123.36472 48.42833)
2	Kelowna	222162	49.888056	-119.495556	POINT (-119.49556 49.88806)

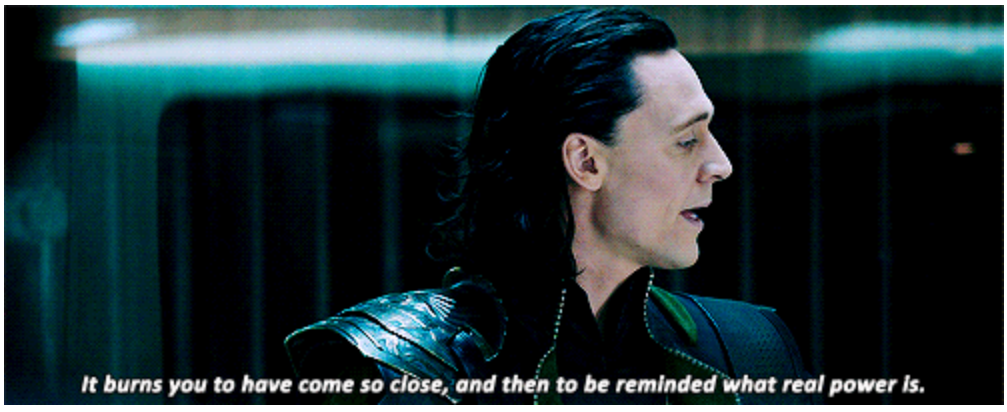
- Let's plot those points on our map:

```
ax = bc.plot(edgecolor="0.2")
cities.plot(ax=ax, markersize=170, edgecolor="0.2")
plt.title("Big cities in B.C.");
```



2.4. Loading from Open Street Map

- So we can read vector data from a file and we can create our own, but let's see what real power feels like



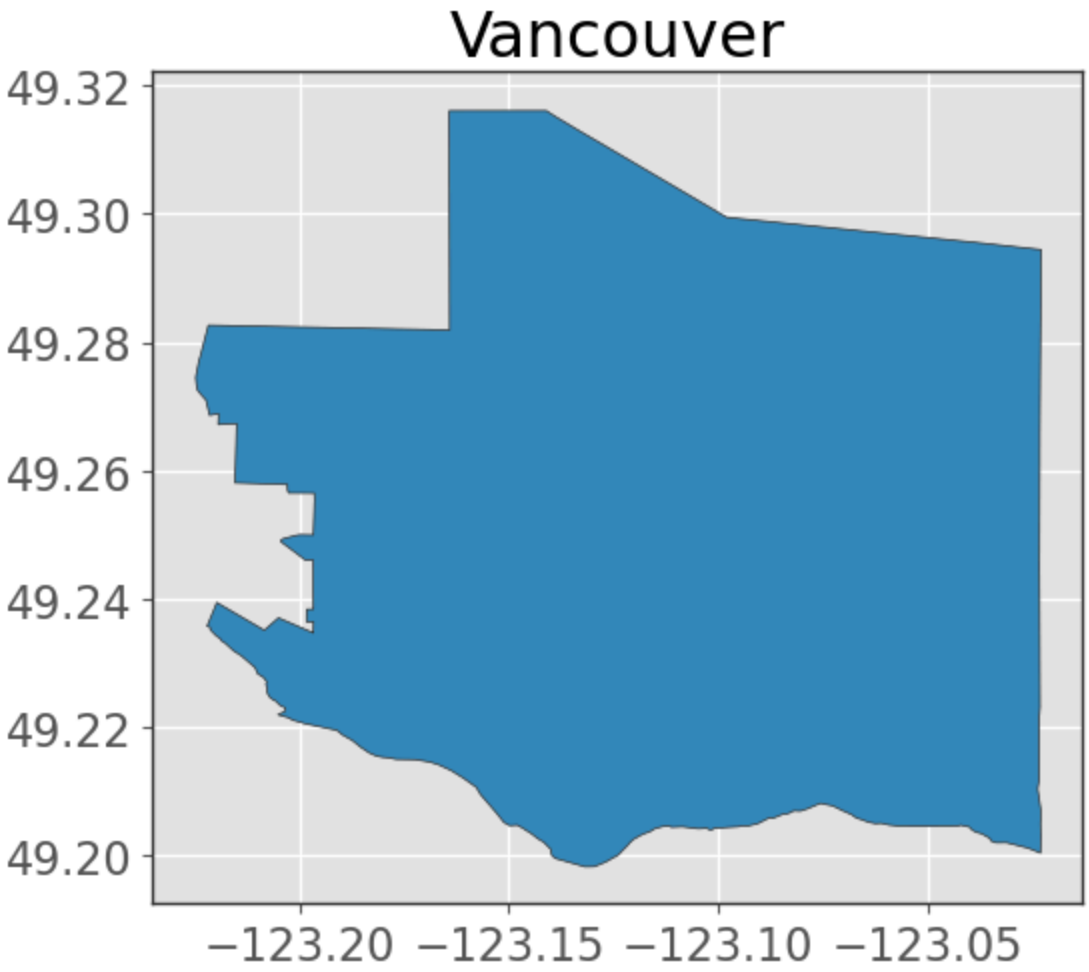
- Often it will be helpful to obtain data from an online source using an API. The most relevant "online source" here is OpenStreetMap (OSM), which is like the Wikipedia of geospatial data (think world map, road networks, bike networks, buidling heights, sandy coastlines, you name it)
- There are plenty of Python APIs for getting data from OSM but by far the best I've come across is osmnx:

```
conda install -c conda-forge osmnx
```

- `osmnx` provides an easy-to-use API to query OSM data. I usually import it with the alias `ox`. Let's get a polygon of Vancouver now using the function `ox.geocode_to_gdf()`

> By default `osmnx` caches responses locally in a folder `cache` so that you can quickly access data again without needing to call the API. You can turn this behaviour off if you wish.

```
import osmnx as ox

vancouver = ox.geocode_to_gdf("Vancouver, Canada")
vancouver.plot(edgecolor="0.2")
plt.title("Vancouver");
```

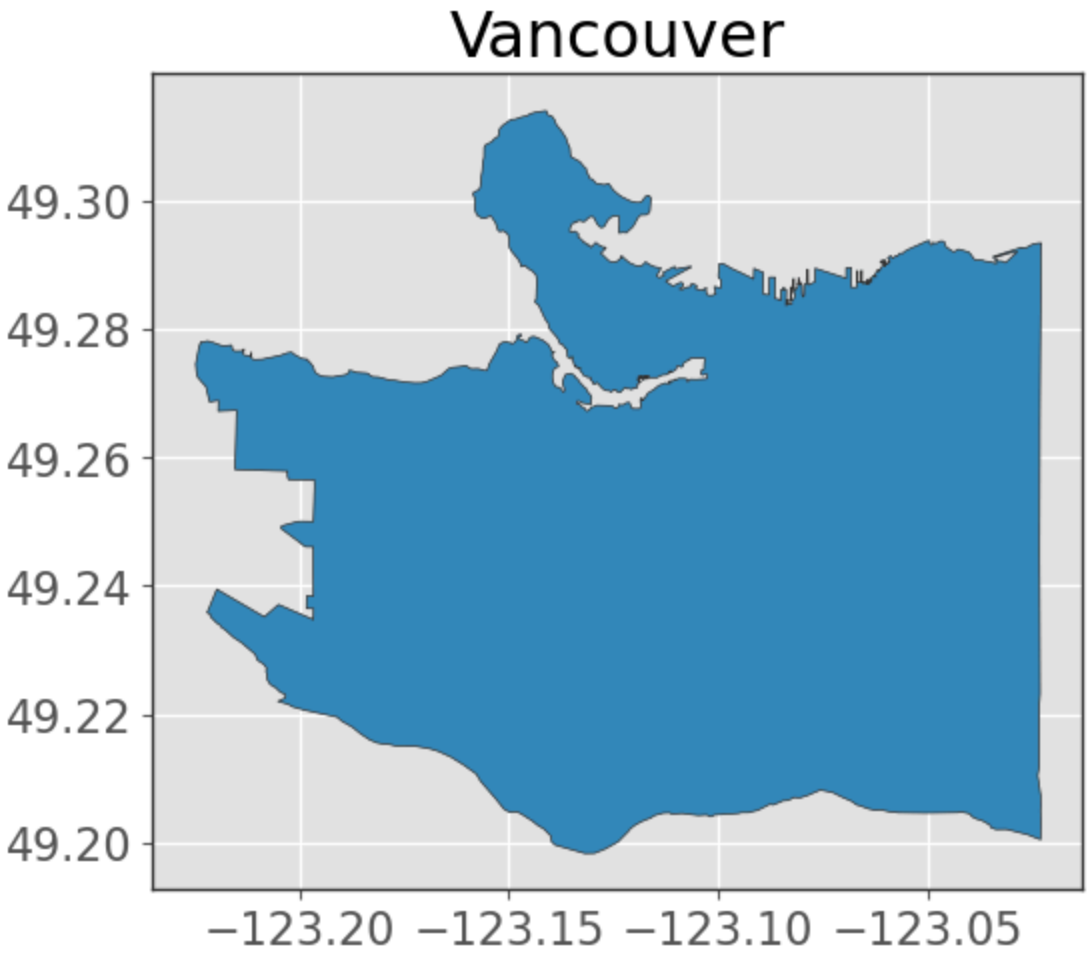


- It's certainly Vancouver, but it looks a bit blocky. It might be a bit low resolution, or someone just decided this was the best way to encapsulate "Vancouver". Either way, let's

use this polygon to “clip” a section of our higher-resolution provinces data (which is the official shapefile downloaded from statcan)

- This is the first geometric wrangling operation we’ll see. I’ll show some more later, but think of “clipping” as passing a top layer of cookie dough (the map above), over a bottom layer cookiecutter (our high-resolution provinces data) to get a shape out:

```
van_bc = gpd.clip(bc, vancouver)
van_bc.plot(edgecolor="0.2")
plt.title("Vancouver");
```

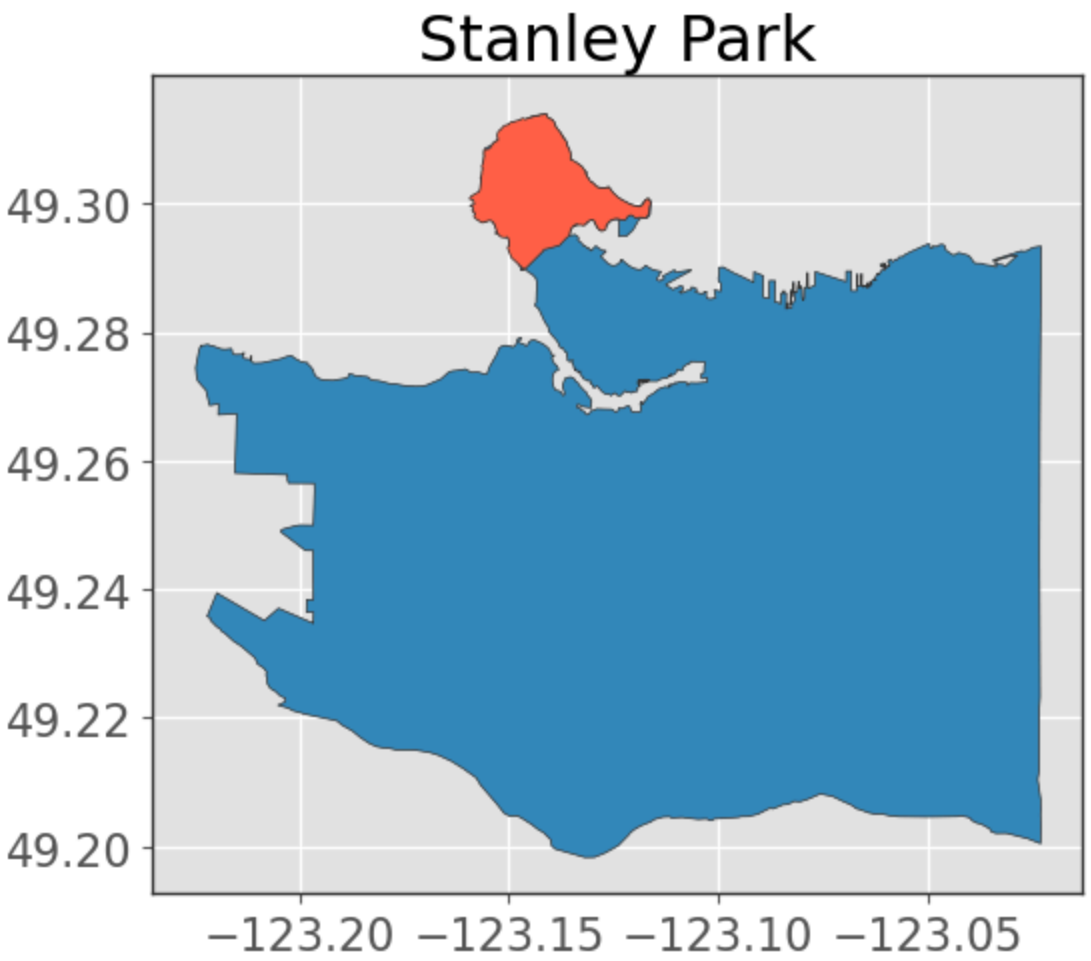


- That looks better! Now we can clearly see Stanley Park!
- Speaking of which, let’s get a polygon of Stanley Park:

```
stanley_park = ox.geocode_to_gdf("Stanley Park, Vancouver")
```

- And plot it on our map:

```
ax = van_bc.plot(edgecolor="0.2")
stanley_park.plot(ax=ax, edgecolor="0.2", color="tomato")
plt.title("Stanley Park");
```



- Okay let's do one last cool thing! Let's use `osmnx` to get the bicycle network within Stanley Park!
- We can get networks like road, rail, bike, etc., using the function `ox.graph_from_place()`:

```
import warnings
warnings.filterwarnings('ignore')

bike_network = ox.graph_from_place("Stanley Park, Vancouver",
                                   network_type="bike")
bike_network
```

```
<networkx.classes.multidigraph.MultiDiGraph at 0x15f3c65b0>
```

- As you can see, this returns a different object, a networkx `MultiDiGraph` which is a structure for holding network/graph-like objects such as road networks
- We're not interested in graph operations, we are just interested in geometries, so we'll convert this to a `GeoDataFrame` using the function `ox.graph_to_gdfs()`:

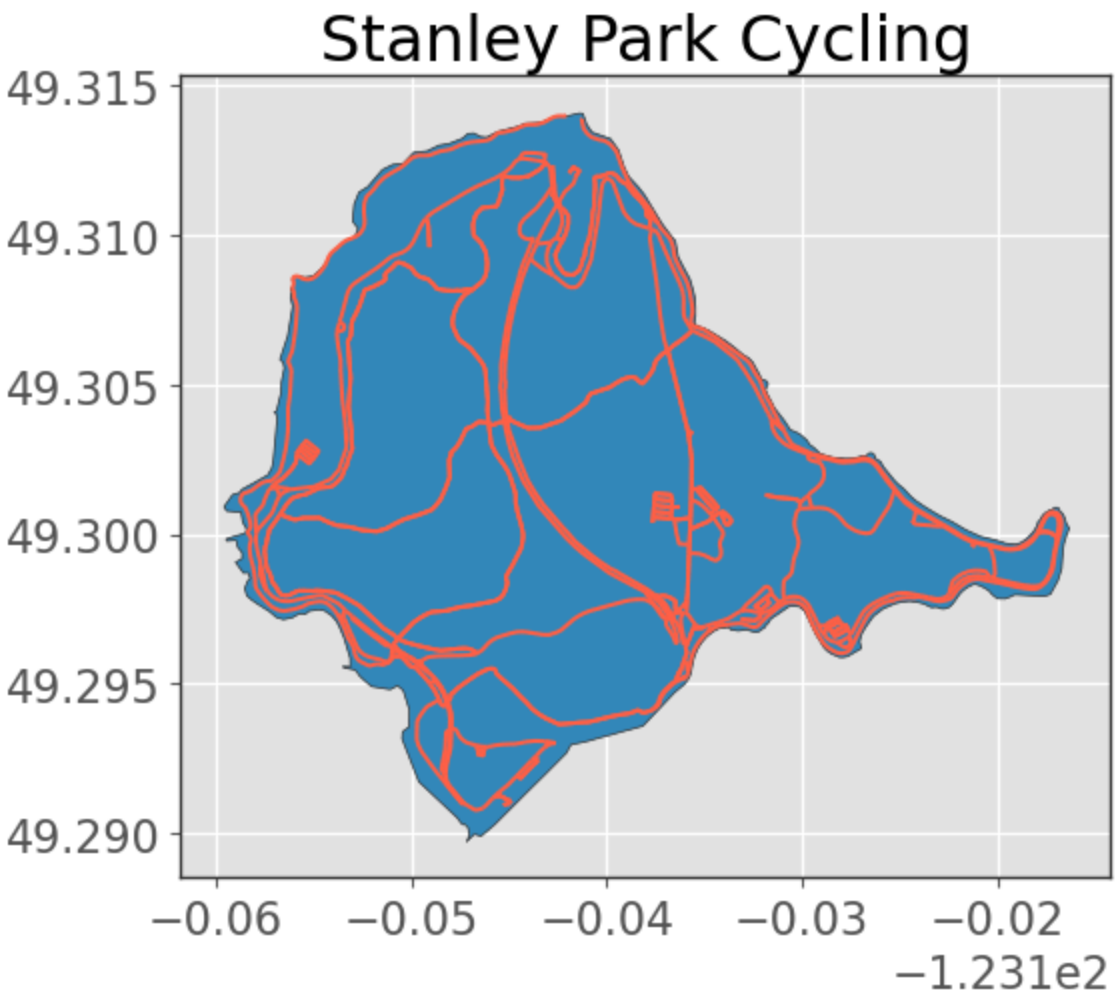
```
bike_network = (ox.graph_to_gdfs(bike_network, nodes=False)
                  .reset_index(drop=True)
                  .loc[:, ["name", "length", "bridge", "geometry"]]
                )
bike_network
```

	name	length	bridge	geometry
0	Stanley Park Causeway	172.973720	yes	LINESTRING (-123.13729 49.29773, -123.13722 49...
1	Stanley Park Causeway	53.210950	NaN	LINESTRING (-123.13729 49.29773, -123.13733 49...
2	NaN	8.066392	NaN	LINESTRING (-123.13211 49.29737, -123.13212 49...
3	Stanley Park Drive	80.487997	NaN	LINESTRING (-123.13211 49.29737, -123.13199 49...
4	NaN	91.002930	NaN	LINESTRING (-123.13211 49.29737, -123.13212 49...
...	...	...	...	...
429	Stanley Park Drive	3.520613	NaN	LINESTRING (-123.11786 49.30008, -123.11788 49...
430	NaN	8.783788	NaN	LINESTRING (-123.11786 49.30008, -123.11792 49...
431	Bridle Path	3.136844	NaN	LINESTRING (-123.15096 49.2962, -123.15095 49....
432	NaN	9.123747	NaN	LINESTRING (-123.15096 49.2962, -123.15086 49....
433	Bridle Path	19.616686	NaN	LINESTRING (-123.15096 49.2962, -123.15098 49....

434 rows × 4 columns

- Now let's plot this bike network on Stanley Park!

```
ax = stanley_park.plot(edgecolor="0.2")
bike_network.plot(ax=ax, edgecolor="0.2", color="tomato")
plt.title("Stanley Park Cycling");
```



You can also create an interactive map using `.explore()`

```
bike_network.explore()
```

- So cool!

2.5. Basic wrangling

- Awesome! So we did some cool mapping and data manipulation above, but what's it all for? Why would you want to do any of this?
- Well we can now start calculating and manipulating this spatial data to answer questions. Let's answer some now.

Question 1: What is the total length of bike lanes in Stanley Park?

- Well, our `GeoDataFrame` actually already has this information, it came for free from our original `ox.graph_from_place()` function and is in the "length" column:

```
total_length = bike_network["length"].sum()  # total length in m
print(f"Total bike lane length: {total_length / 1000:.0f}km")
```

```
Total bike lane length: 58km
```

- But even if we didn't have this column, we could still calculate lengths based on our line geometries and the `.length` attribute:

```
bike_network["geometry"]
```

```
0      LINESTRING (-123.13729 49.29773, -123.13722 49...
1      LINESTRING (-123.13729 49.29773, -123.13733 49...
2      LINESTRING (-123.13211 49.29737, -123.13212 49...
3      LINESTRING (-123.13211 49.29737, -123.13199 49...
4      LINESTRING (-123.13211 49.29737, -123.13212 49...
                           ...
429    LINESTRING (-123.11786 49.30008, -123.11788 49...
430    LINESTRING (-123.11786 49.30008, -123.11792 49...
431    LINESTRING (-123.15096 49.2962, -123.15095 49....
432    LINESTRING (-123.15096 49.2962, -123.15086 49....
433    LINESTRING (-123.15096 49.2962, -123.15098 49....
Name: geometry, Length: 434, dtype: geometry
```

```
bike_network["geometry"].length
```

```
0      0.001740
1      0.000636
2      0.000073
3      0.001021
4      0.001151
         ...
429    0.000035
430    0.000108
431    0.000029
432    0.000115
433    0.000180
Length: 434, dtype: float64
```

- What's that warning? More on that a bit later, but it's telling us that our coordinate system has units of degrees - not linear units like meters which would be better for calculating distances
- I'm going to convert my geometries to a coordinate systems based on linear units (meters). I'll specify the projection "EPSG:3347" (Lambert projection) which is the one used by statcan:

```
bike_network = bike_network.to_crs("EPSG:3347")
bike_network["geometry"].length
```

```
0      172.901620
1       53.246522
2        8.058879
3       80.580937
4       91.110800
          ...
429      3.518654
430      8.791850
431      3.134422
432      9.133630
433     19.599690
Length: 434, dtype: float64
```

```
total_length = bike_network["geometry"].length.sum()
print(f"Total bike lane length: {total_length / 1000:.0f}km")
```

```
Total bike lane length: 59km
```

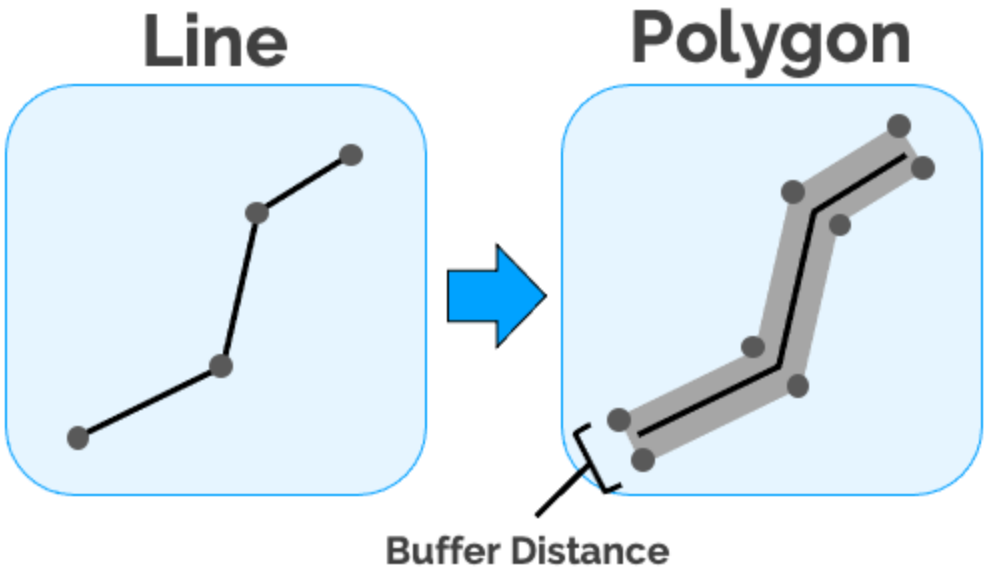
- Close to the same as we got before! Nice!

Question 2: What percentage of the area of Stanley Park is bike lanes?

- This is a tougher one!
- First let's calculate the "area" of our bike lanes. We need to make an assumption about the width of our lanes. This City of Vancouver planning document suggests the bike lanes around Stanley Park are about 3m so let's go with that.
- I'm going to use the `.buffer()` method to turn the lines of my bike network into polygons with a specified width (3m in our cases). Because my `bike_network` data is in linear meters units now (remember, I changed the projection), I'm also going to convert our Stanley Park map to that projection ("EPSG:3347"):

```
stanley_park = stanley_park.to_crs("EPSG:3347")
```

- Now let's "buffer" our bike lanes to be be 3m wide polygons
- "Buffer" just mean to add some area around the object

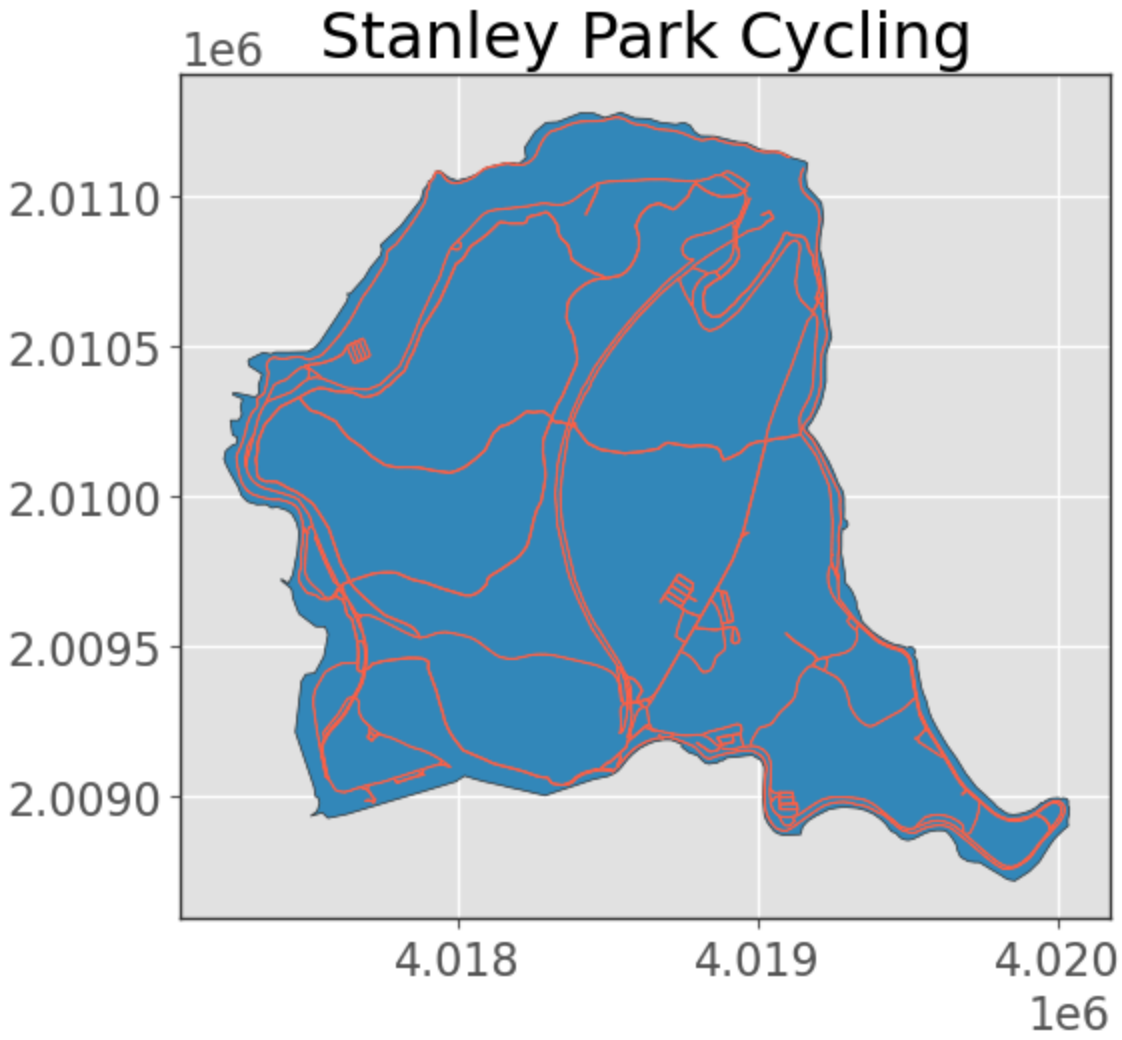



```
width = 3  # desired with of bike lanes in meters
bike_network["geometry"] = bike_network.buffer(distance=width / 2)  # note tha
bike_network
```

	name	length	bridge	geometry
0	Stanley Park Causeway	172.973720	yes	POLYGON ((4018560.899 2009366.392, 4018561.108...
1	Stanley Park Causeway	53.210950	NaN	POLYGON ((4018556.97 2009381.431, 4018555.748 ...
2	NaN	8.066392	NaN	POLYGON ((4018874.157 2009173.346, 4018874.22 ...
3	Stanley Park Drive	80.487997	NaN	POLYGON ((4018880.901 2009163.536, 4018883.956...
4	NaN	91.002930	NaN	POLYGON ((4018870.301 2009159.558, 4018867.565...
...	...	...	...	...
429	Stanley Park Drive	3.520613	NaN	POLYGON ((4019925.158 2008938.688, 4019925.04 ...
430	NaN	8.783788	NaN	POLYGON ((4019922.505 2008940.618, 4019918.681...
431	Bridle Path	3.136844	NaN	POLYGON ((4017604.407 2009700.263, 4017604.512...
432	NaN	9.123747	NaN	POLYGON ((4017609.166 2009689.722, 4017609.231...
433	Bridle Path	19.616686	NaN	POLYGON ((4017601.792 2009692.945, 4017600.763...

434 rows × 4 columns

```
ax = stanley_park.plot(edgecolor="0.2")
bike_network.plot(ax=ax, edgecolor="tomato")
plt.title("Stanley Park Cycling");
```



- Now we can calculate the area using the `.area` attribute
- Note that `geopandas` is smart enough to know that you probably want to calculate these spatial attributes on the `geometry` column, so you don't actually have to index that particular column (if you have multiple geometry columns, then you can choose which one is "active" and acted on by default using the `.set_geometry()` method):

```
bike_network_area = bike_network.area.sum()
print(f"Bike path area: {bike_network_area:.0f} m2")
```

```
Bike path area: 178516 m2
```

- If you think about it, that should be roughly similar to if we just multiplied our `total_length` from "Question 1" by 3 (the width of the bike paths):

```
total_length * 3
```

```
175517.46024714972
```

- Well that's a nice sanity check!
- Now we just need the area of Stanley Park and we can calculate our ratio:

```
stanley_park_area = stanley_park.area
print(f"{bike_network_area / stanley_park_area[0] * 100:.2f}% of Stanley Park
```

```
4.47% of Stanley Park is bike paths.
```

Question 3: What FSA in Vancouver has the most bike lanes (by length)?

- An FSA is a "forward sortation area", basically a group of postal codes that all start with the same first 3 letters
- So to answer this question, we need two things:
  1. FSA polygons (available on statcan here)
  2. The bike network for all of Vancouver
- I have already downloaded the above shapefile of FSAs for all of Canada. We'll load it in and then clip it using our Vancouver polygon:

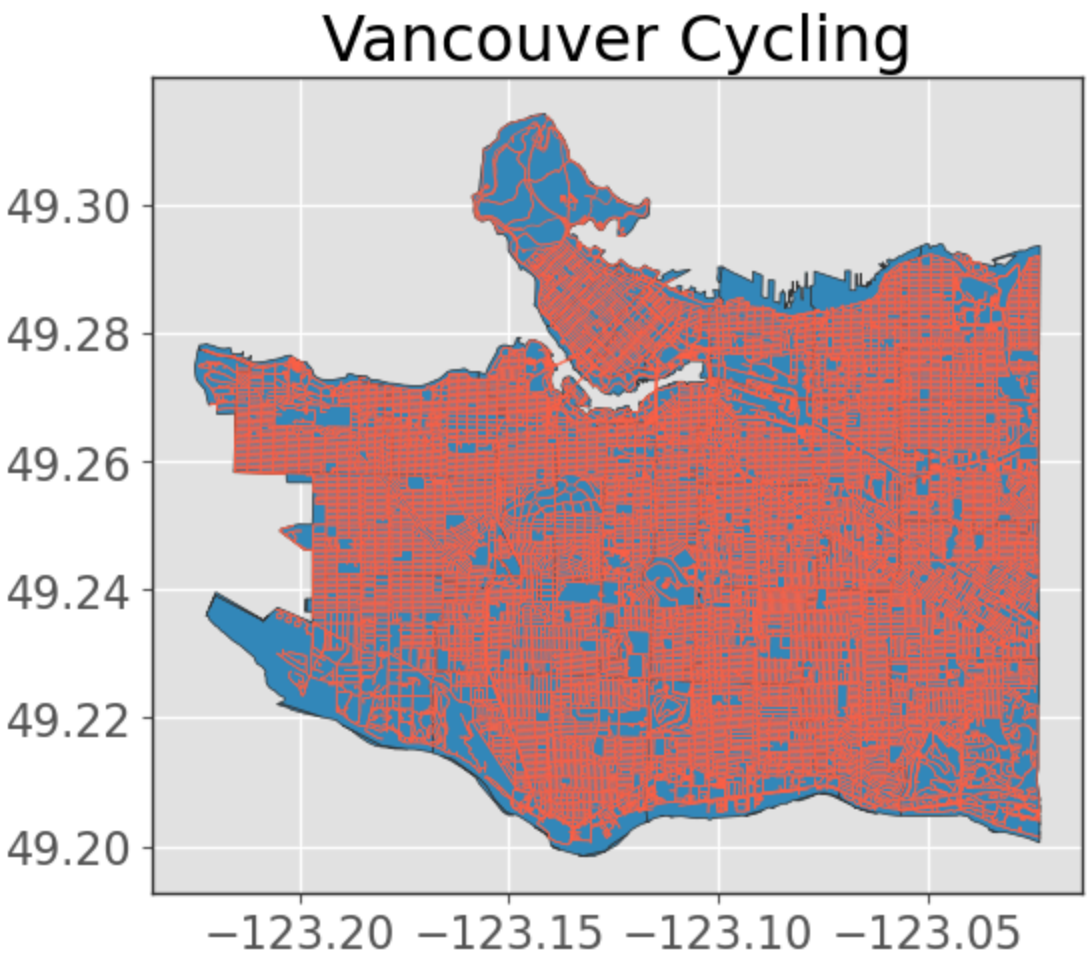
```
fsa = gpd.read_file("data-spatial/fsa")
fsa = fsa.to_crs("EPSG:4326")
van_fsa = gpd.clip(fsa, vancouver)
```

- Now let's get the Vancouver bike network using `osmnx`:

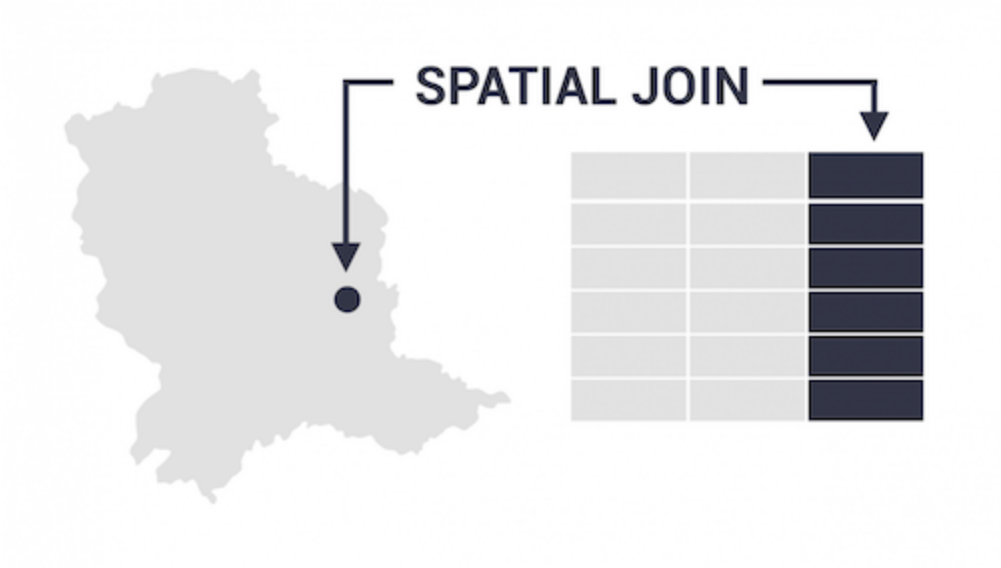
```
van_bike_network = ox.graph_from_place("Vancouver", network_type="bike")
van_bike_network = (ox.graph_to_gdfs(van_bike_network, nodes=False)
                    .reset_index(drop=True)
                    .loc[:, ["name", "length", "bridge", "geometry"]]
                   )
```

- Let's take a look at our data so far:

```
ax = van_fsa.plot(edgecolor="0.2")
van_bike_network.plot(ax=ax, edgecolor="tomato", linewidth=0.5)
plt.title("Vancouver Cycling");
```



- Okay so how do we work out the total length of bike lanes in each FSA?
- We need to do a spatial join, which joins two geometries based on their locations. In the plot below, we'll join the column of the dark dot to the column(s) of the grey polygon because the dark dot is contained within the spatial region of the polygon.



> Source: GISGeography.

- We can do a spatial join with `gpd.sjoin()` (it's just like joining in base `pandas`). There are different options for the argument `predicate` which allow you how to control the join. Below I'll specify "contain", meaning I only want to join when a bike lane is fully contained within an FSA (you can read more about `predicatevan_bike_network` in the documentation);

```
joined_data = gpd.sjoin(van_fsa, van_bike_network, how="inner", predicate="con
joined_data.head()
```

	CFSAUID	PRUID	PRNAME	geometry	index_right	name	lengtl
1614	V6P	59	British Columbia / Colombie-Britannique	POLYGON ((-123.11703 49.20387, -123.11732 49.2...	27740	Hudson Street	48.17556
1614	V6P	59	British Columbia / Colombie-Britannique	POLYGON ((-123.11703 49.20387, -123.11732 49.2...	48963	Hudson Street	48.17556
1614	V6P	59	British Columbia / Colombie-Britannique	POLYGON ((-123.11703 49.20387, -123.11732 49.2...	27741	West 77th Avenue	37.92880
1614	V6P	59	British Columbia / Colombie-Britannique	POLYGON ((-123.11703 49.20387, -123.11732 49.2...	59792	West 77th Avenue	37.92880
1614	V6P	59	British Columbia / Colombie-Britannique	POLYGON ((-123.11703 49.20387, -123.11732 49.2...	59791	NaN	48.38372

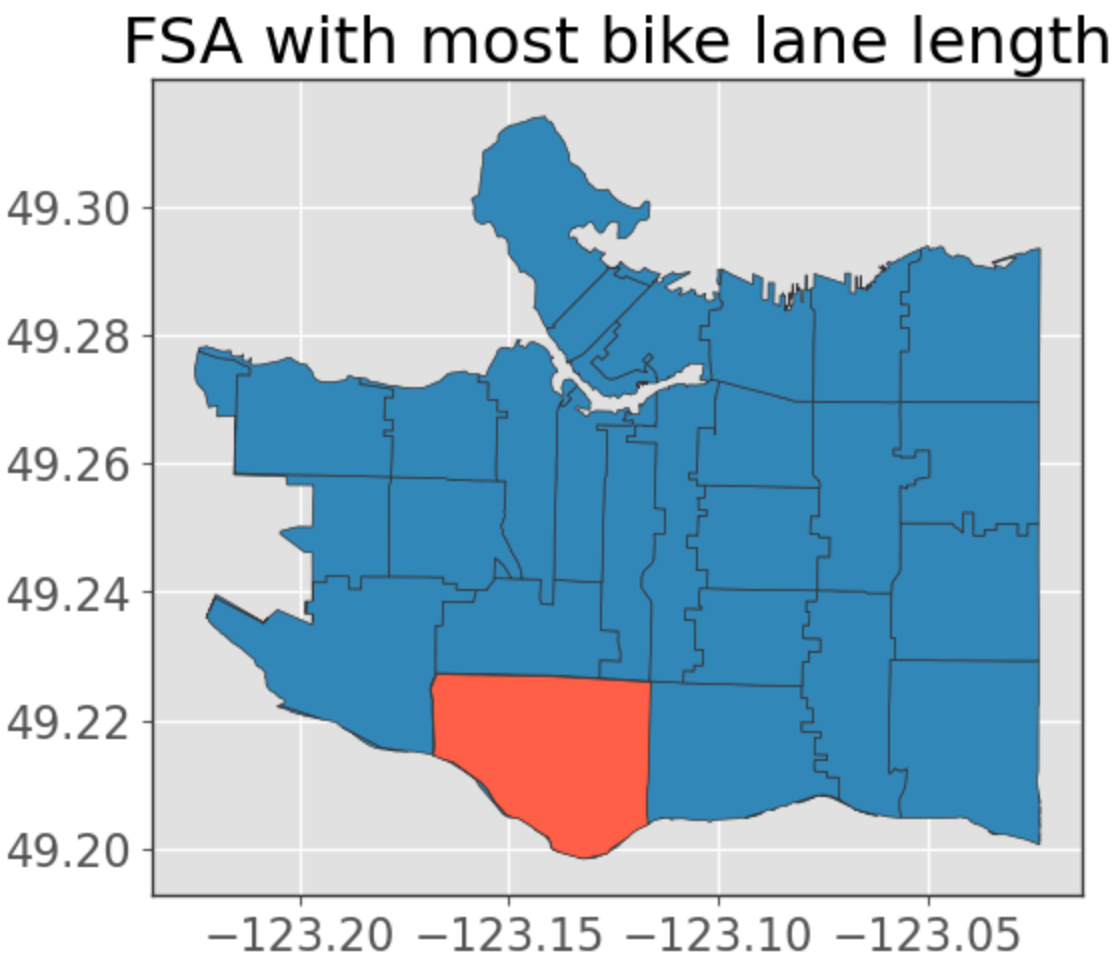
- Now we just need to `.groupby()`:

```
(joined_data[["CFSAUID", "length"]].groupby(by="CFSAUID")
                                   .sum()
                                   .sort_values("length", ascending=False)
                                   .head()
)
```

	length
CFSAUID	
V6P	359336.640531
V5R	281176.237261
V5X	271991.854461
V5S	269586.673042
V5K	249186.081774

- We see that “V6P” has the largest length of bike lanes, what FSA is that?

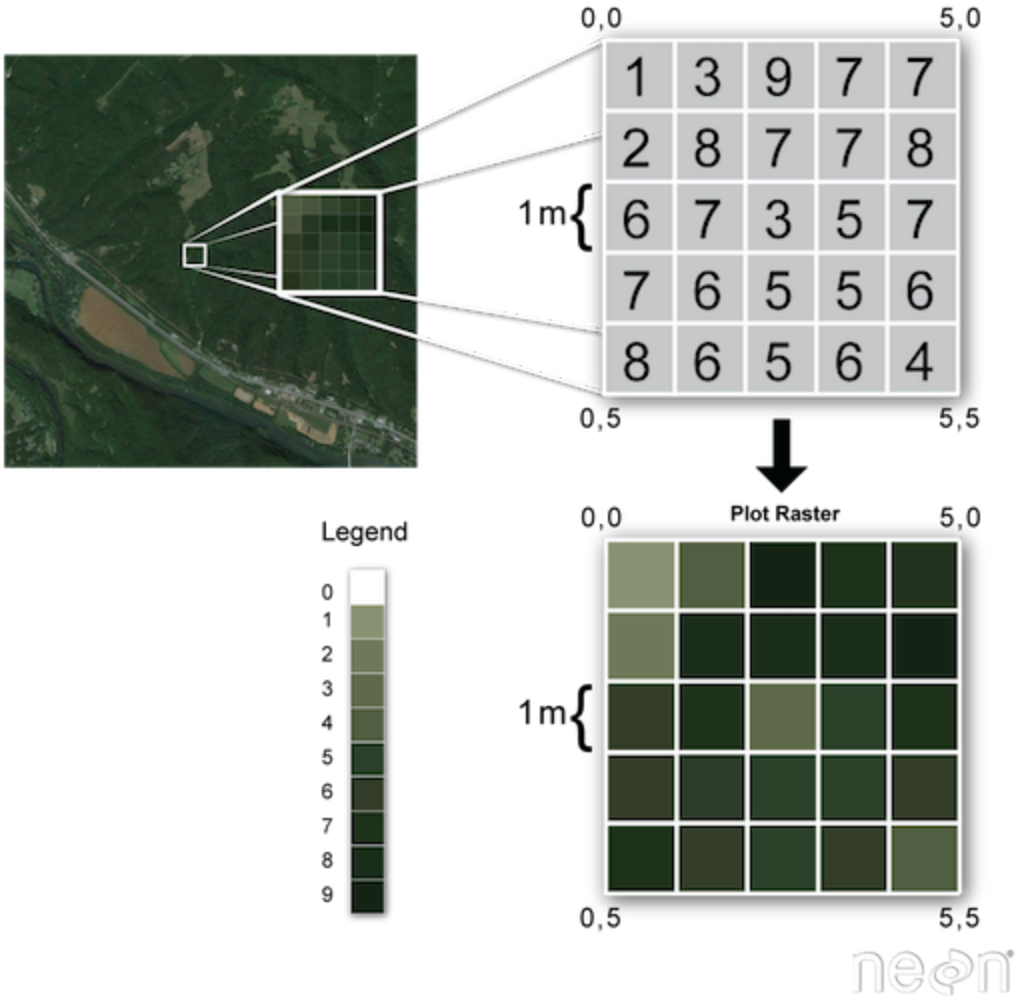
```
ax = van_fsa.plot(edgecolor="0.2")
van_fsa.query("CFSAUID == 'V6P'").plot(ax=ax, edgecolor="0.2", color="tomato")
plt.title("FSA with most bike lane length");
```



- This FSA includes the neighbourhoods Kerrisdale, Marpole, Oakridge, South Vancouver and West Side. It also features a part of the Arbutus Greenway!

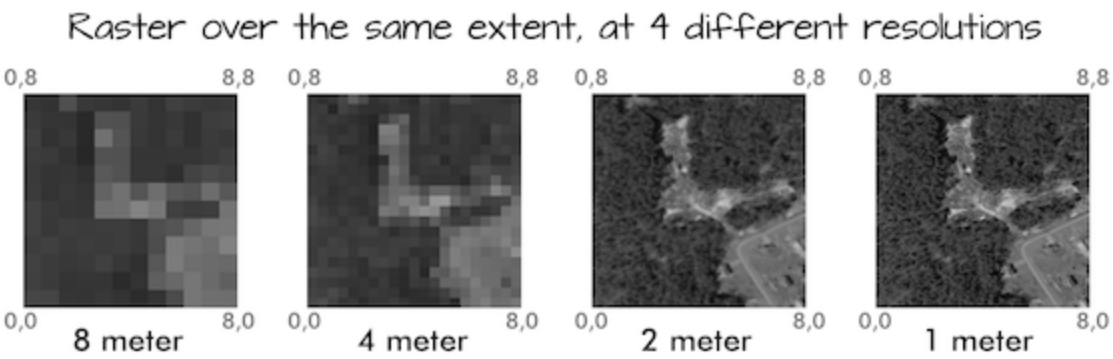
3. Working with raster data

- Unlike vector data (geometric objects like points, lines, polygons), raster data is a matrix of values of “pixels” (also called “cells”)
- Each cell represents a small area and contains a value representing some information



> Source: National Ecological Observatory Network.

- Raster data is just like digital image data you look at on your computer, except that now, each pixel represents a spatial region
- The “resolution” of a raster is the area that each pixel represents. A 1 meter resolution raster means that each pixel represents a 1 m x 1 m area on the ground.
- However, when we say “high resolution” we often mean, a low value of resolution for each pixel, i.e., 1 meter resolution is higher than 8 meter resolution as exmplified by the image below:



> Source: National Ecological Observatory Network.

- Like vector data, there are different file formats for raster data. The most common is GeoTIFF (`.tif`), which is essentially an image file with georeferencing information embedded within it.
- Raster data is used for a variety of problems, common examples include satellite imagery and digital elevation models. Those things are gettings a bit outside the scope of this course but let's briefly look at some raster data below
- The core packge for working with raster data in Python is rasterio

```
import rasterio
```

- I have a satellite image raster file of part of UBC in my data folder which I downloaded from the Abacus Data Network
- Let's load it in with `rasterio`:

```
dataset = rasterio.open("/Users/katieburak/Desktop/MDS/481E_5456N/481E_5456N.t
```

- We can start to investigate things like the width and height (in pixels/cells) of the raster:

```
print(f"Width: {dataset.width} pixels")
print(f"Height: {dataset.height} pixels")
```

```
Width: 10000 pixels
Height: 10000 pixels
```

- Raster data often have “bands” similar to “channels” in colour images you saw back in DSCI 572
- This example has 4 bands (in order: red, blue, green, infrared):

```
dataset.count
```

```
4
```

- We could import the first band as a numpy array using:

```
band1 = dataset.read(1)
band1
```

```
array([[ 70,  67,  52, ...,  88,  86,  85],
       [ 64,  60,  53, ...,  89,  88,  92],
       [ 67,  63,  62, ...,  93, 104, 107],
       ...,
       [ 58,  56,  54, ...,  52,  28,  20],
       [ 55,  56,  55, ...,  55,  30,  25],
       [ 54,  57,  59, ...,  30,  19,  29]], dtype=uint8)
```

- But it’s easier just to see the image
- First I’m going to “downsample” my raster (reduce the resolution by increasing the cell size) to reduce the size of the data and speed things up a bit:

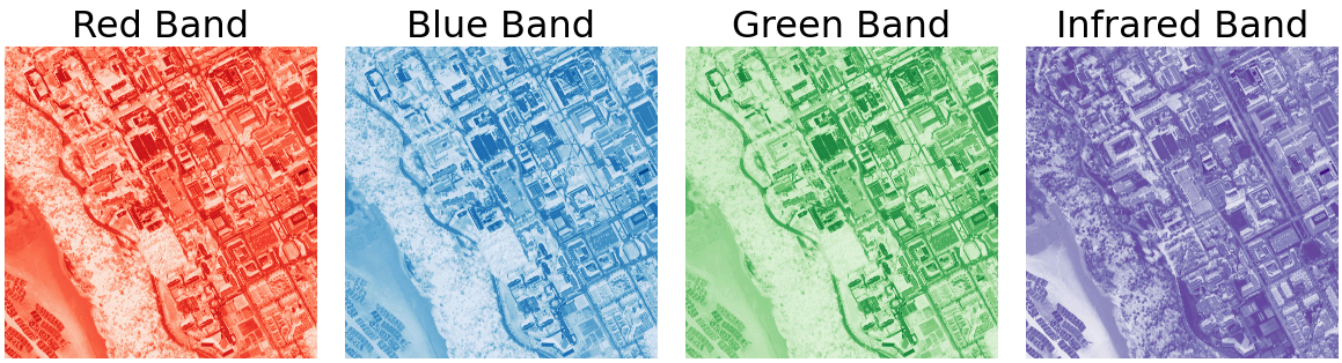
```
rescale_factor = 0.5
scaled_data = dataset.read(out_shape=(dataset.count, int(dataset.height * resc
                                      int(dataset.width * rescale_factor))
                           )
print(f"Scaled shape: {scaled_data.shape}")
```

```
Scaled shape: (4, 5000, 5000)
```

- As our data is just numpy array(s), we can plot it with the `matplotlib` function `plt.imshow()`:

```

```

```
fig, ax = plt.subplots(1, 4, figsize=(12, 4))
cmaps = ["Reds", "Blues", "Greens", "Purples"]
bands = ["Red Band", "Blue Band", "Green Band", "Infrared Band"]
for band in [0, 1, 2, 3]:
    ax[band].imshow(scaled_data[band, :, :], cmap=cmaps[band])
    ax[band].set_title(bands[band])
    ax[band].axis("off")
plt.tight_layout();
```



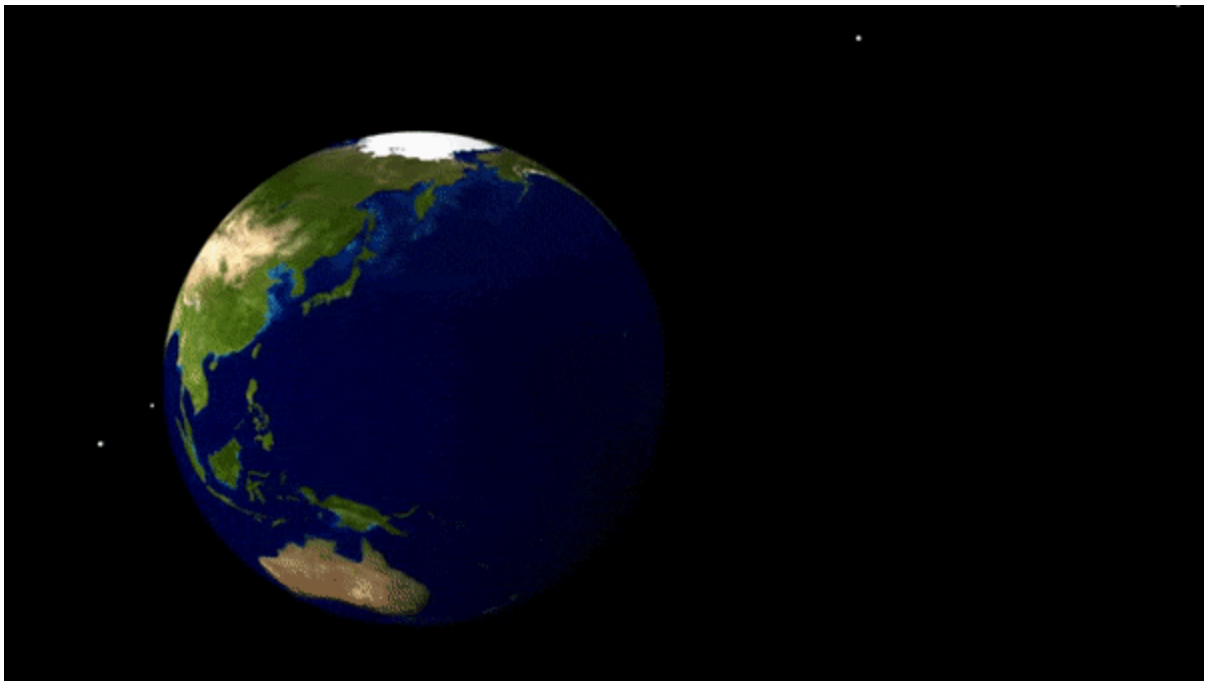
- Of course, it looks more realistic using all channels:

```
plt.figure(figsize=(5, 5))
plt.imshow(np.moveaxis(scaled_data, 0, -1)[:, :, :3])
plt.axis("off")
plt.title("All bands");
```

- `rasterio` has lots of advanced fucntionality for manipulating and plotting raster data if you find the need to. Check out the documentation.

4. Coordinate reference systems

- Generally speaking a coordinate reference system is how we project the 3D surface of the Earth onto a 2D shape for easy viewing



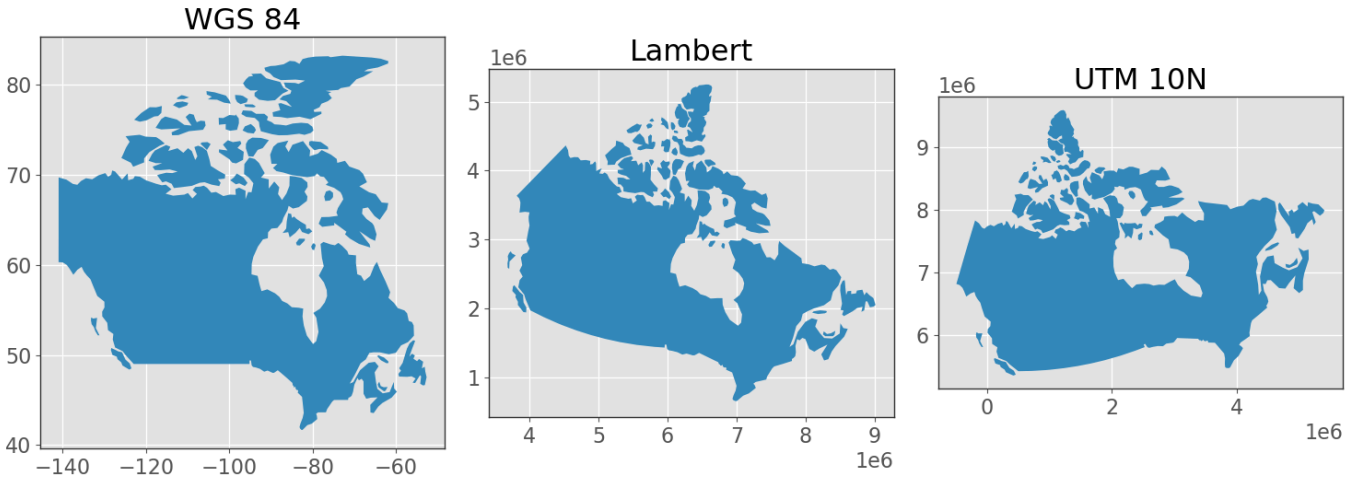
- There are many different projections and they are typically identified by an EPSG code
- No projection is perfect (it's impossible to perfectly flatten a 3d sphere) and each comprises on minimzing the distortion of shapes, distances, and areas of the Earth
- All you need to know is that some projections are:
  - in **angular units** (degrees of latitude and longitude) and are good for locating places on Earth, for making global maps, and minimizing shape distortion. The most common is WGS 84 ("EPSG:4326")
  - in **linear units** (e.g., meters) and are good for measuring distances. Most common is UTM which splits the Earth into different linear regions, the code for the region encompassing British Columbia is "EPSG:32610"
- But many countries/regions use other specific projections which minimize distortion of that specific area. For example Statistics Canada uses the Lambert projection for Canada ("EPSG:3347")
- Much of the time, you won't have to worry too much about this, or it will be determined/specified by your project
- I downloaded a shapefile of countries from Natural Earth in a `data-spatial/countries` shapefile.
- Let's take a quick look at some different projections for Canada:

```
countries = gpd.read_file("data-spatial/countries")
countries.head()
```

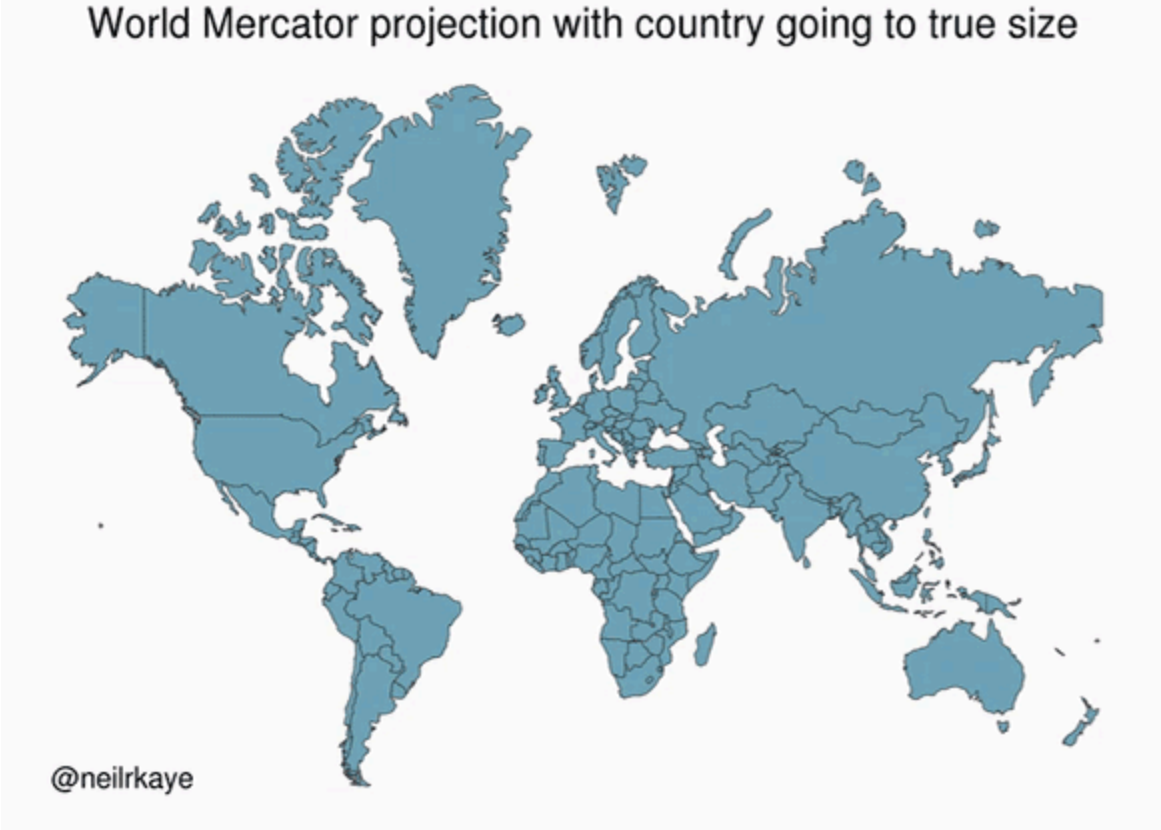
	featurecla	scalerank	LABELRANK	SOVEREIGNT	SOV_A3	ADM0_DIF	LEVE
0	Admin-0 country	1	6	Fiji	FJI	0	
1	Admin-0 country	1	3	United Republic of Tanzania	TZA	0	
2	Admin-0 country	1	7	Western Sahara	SAH	0	
3	Admin-0 country	1	2	Canada	CAN	0	
4	Admin-0 country	1	2	United States of America	US1	1	

5 rows × 169 columns

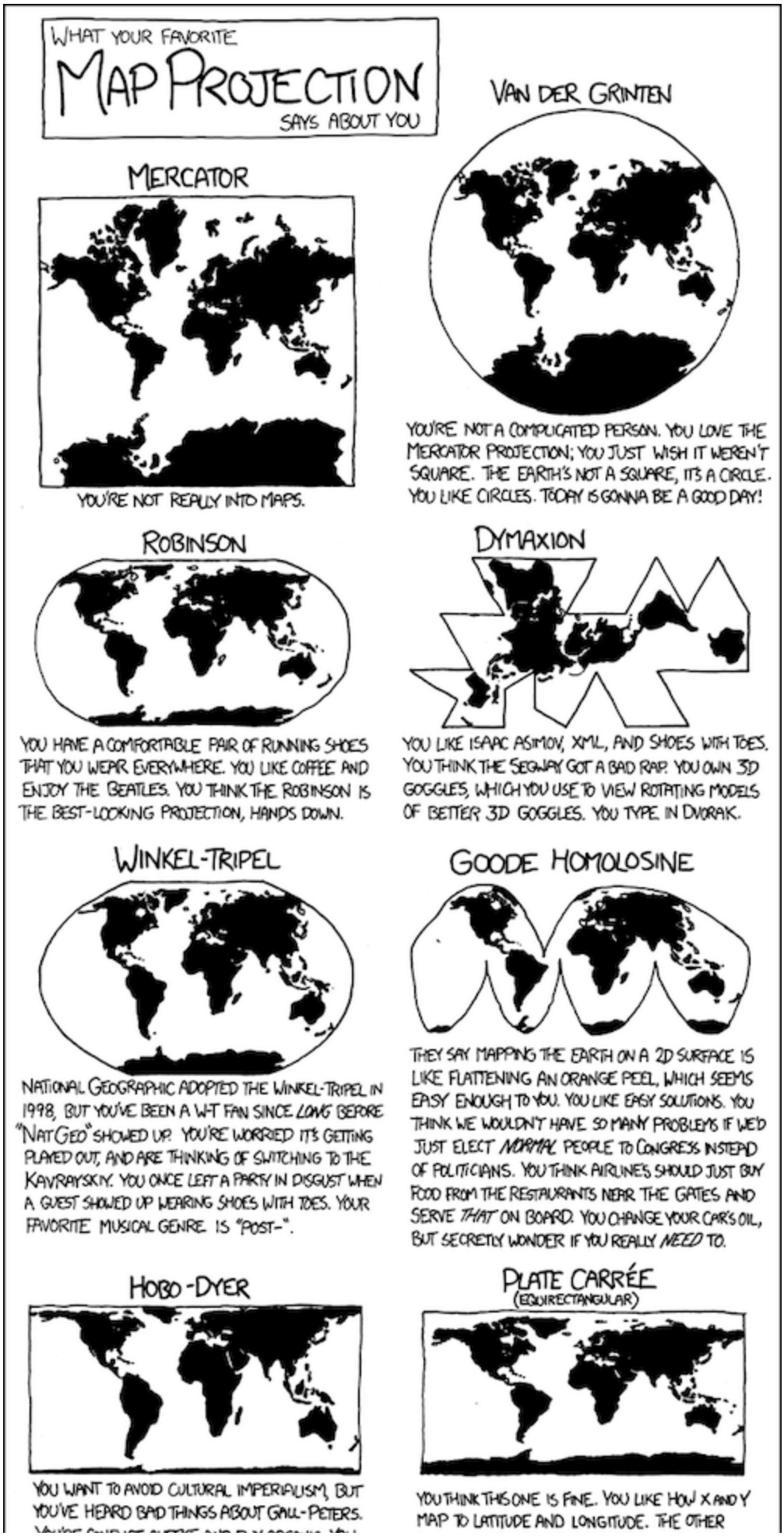
```
canada = countries.query("SOVEREIGNT == 'Canada'")

# Plot
fig, axs = plt.subplots(1, 3, figsize=(15, 12))
crs_list = [("WGS 84", "EPSG:4326"), ("Lambert", "EPSG:3347"), ("UTM 10N", "EP
for n, (name, epsg) in enumerate(crs_list):
    canada.to_crs(epsg).plot(ax=axs[n])
    axs[n].set_title(name)
plt.tight_layout();
```

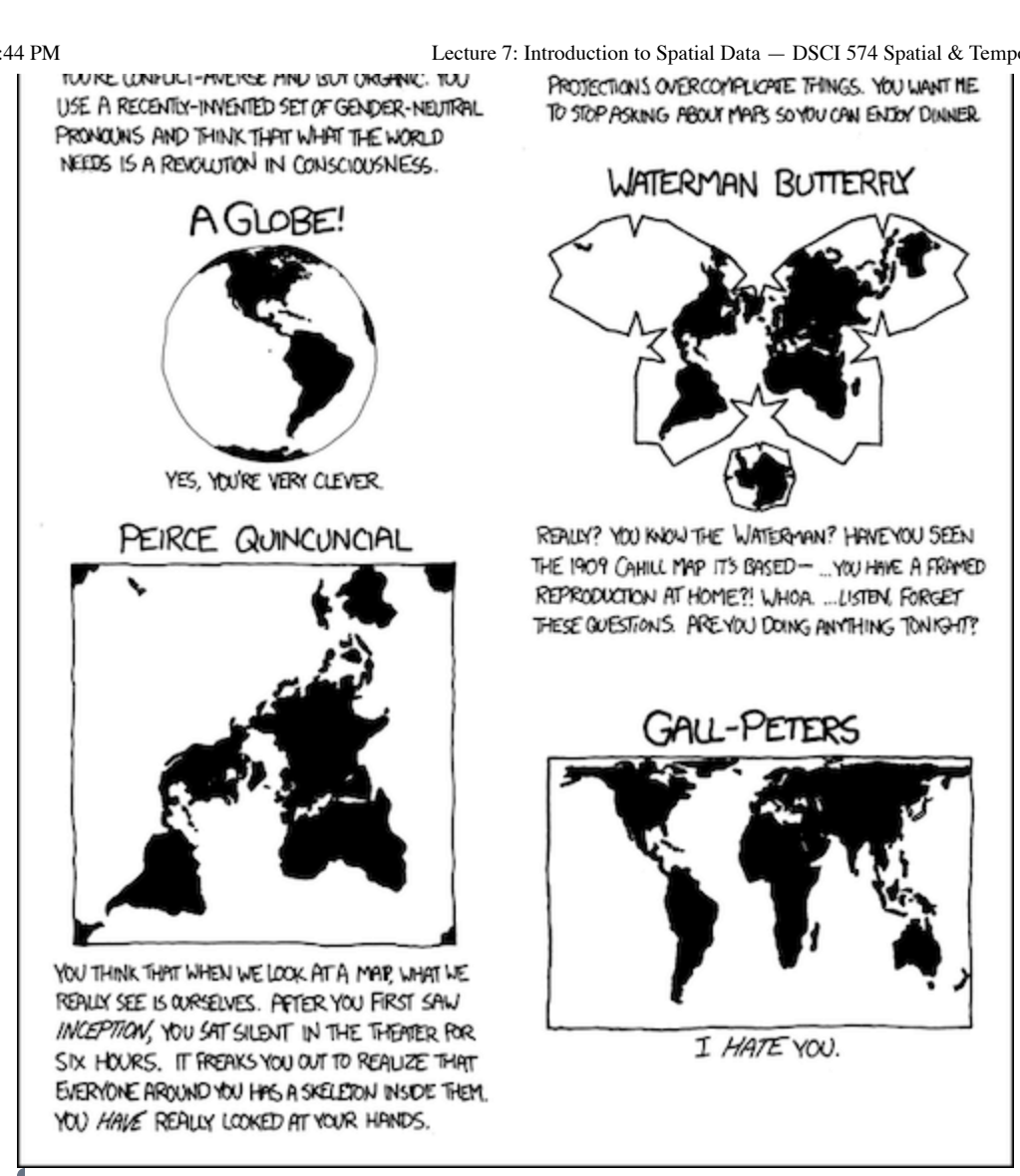


True or False: Africa is larger than US, China, and Europe combined



- Here's a classic xkcd comic to finish off:


WHAT YOUR FAVORITE
MAP PROJECTION
SAYS ABOUT YOU
MERCATOR
YOU'RE NOT REALLY INTO MAPS.
VAN DER GRINTEN
YOU'RE NOT A COMPLICATED PERSON. YOU LOVE THE MERCATOR PROJECTION; YOU JUST WISH IT WEREN'T SQUARE. THE EARTH'S NOT A SQUARE, IT'S A CIRCLE. YOU LIKE CIRCLES. TODAY IS GONNA BE A GOOD DAY!
ROBINSON
YOU HAVE A COMFORTABLE PAIR OF RUNNING SHOES THAT YOU WEAR EVERYWHERE. YOU LIKE COFFEE AND ENJOY THE BEATLES. YOU THINK THE ROBINSON IS THE BEST-LOOKING PROJECTION, HANDS DOWN.
DYMAXION
YOU LIKE ISAAC ASIMOV, XML, AND SHOES WITH TOES. YOU THINK THE SEGWAY GOT A BAD RAP. YOU OWN 3D GOGGLES, WHICH YOU USE TO VIEW ROTATING MODELS OF BETTER 3D GOGGLES. YOU TYPE IN DVORAK.
WINKEL-TRIPEL
NATIONAL GEOGRAPHIC ADOPTED THE WINKEL-TRIPEL IN 1998, BUT YOU'VE BEEN A WT FAN SINCE LONG BEFORE "NAT GEO" SHOWED UP. YOU'RE WORRIED IT'S GETTING PLAYED OUT, AND ARE THINKING OF SWITCHING TO THE KAVRAYSKIY. YOU ONCE LEFT A PARTY IN DISGUST WHEN A GUEST SHOWED UP WEARING SHOES WITH TOES. YOUR FAVORITE MUSICAL GENRE IS "POST-".
GOODE HOMOLOSINE
THEY SAY MAPPING THE EARTH ON A 2D SURFACE IS LIKE FLATTENING AN ORANGE PEEL, WHICH SEEMS EASY ENOUGH TO YOU. YOU LIKE EASY SOLUTIONS. YOU THINK WE WOULDN'T HAVE SO MANY PROBLEMS IF WE'D JUST ELECT NORMAL PEOPLE TO CONGRESS INSTEAD OF POLITICIANS. YOU THINK AIRLINES SHOULD JUST BUY FOOD FROM THE RESTAURANTS NEAR THE GATES AND SERVE THAT ON BOARD. YOU CHANGE YOUR CAR'S OIL, BUT SECRETLY WONDER IF YOU REALLY NEED TO.
HOBO-DYER
YOU WANT TO AVOID CULTURAL IMPERIALISM, BUT YOU'VE HEARD BAD THINGS ABOUT GALL-PETERS.
PLATE CARRÉE
(EQUIRECTANGULAR)
YOU THINK THIS ONE IS FINE. YOU LIKE HOW X AND Y MAP TO LATITUDE AND LONGITUDE. THE OTHER


YOU'RE CONFLICT-AVERSE AND SOY ORGANIC. YOU USE A RECENTLY-INVENTED SET OF GENDER-NEUTRAL PRONOUNS AND THINK THAT WHAT THE WORLD NEEDS IS A REVOLUTION IN CONSCIOUSNESS.
A GLOBE!
YES, YOU'RE VERY CLEVER.
PEIRCE QUINCUNCIAL
YOU THINK THAT WHEN WE LOOK AT A MAP, WHAT WE REALLY SEE IS OURSELVES. AFTER YOU FIRST SAW INCEPTION, YOU SAT SILENT IN THE THEATER FOR SIX HOURS. IT FREAKS YOU OUT TO REALIZE THAT EVERYONE AROUND YOU HAS A SKELETON INSIDE THEM. YOU HAVE REALLY LOOKED AT YOUR HANDS.
PROJECTIONS OVERCOMPLICATE THINGS. YOU WANT ME TO STOP ASKING ABOUT MAPS SO YOU CAN ENJOY DINNER.
WATERMAN BUTTERFLY
REALLY? YOU KNOW THE WATERMAN? HAVE YOU SEEN THE 1909 CAHILL MAP IT'S BASED— ...YOU HAVE A FRAMED REPRODUCTION AT HOME?! WHOA. ...LISTEN, FORGET THESE QUESTIONS. ARE YOU DOING ANYTHING TONIGHT?
GALL-PETERS
I HATE YOU.

Source: xkcd.