[**NOTE**: you can download an `.ipynb` version of this file here]

Lab 4 - Acquiring data from the web in Python

In this lab, we will interact with a few APIs to get a feel for how they work and how you can make the most of them when trying to access data on the web. To follow this session, you will need to be able to access the following:

- The internet
- A Python installation such as the "Geographic Data Science Stack 2019" in the University of Liverpool computers, or the `gds_env` Docker container in your own machine (see here for instructions)
- A Mapbox API token, which you can access through your Mapbox account

```
%matplotlib inline
from IPython.display import IFrame

import contextily as cx
import geopandas
import json
import matplotlib.pyplot as plt
import mercantile
import pandas
import requests
```

UK Police API

First pass

```
IFrame("https://data.police.uk/docs/", 300, 150)
```

```
    <iframe
        width="300"
        height="150"
        src="https://data.police.uk/docs/"
        frameborder="0"
        allowfullscreen
    ></iframe>
```

Let's look for crime events by location. For example, we can take December of 2019 on a given location (`53.40146` lat and `-2.96459` lon) and build the query:

```
url = ('https://data.police.uk/api/outcomes-at-location?'\
       'date=2019-12&lat=53.40146&lng=-2.96459')
url
```

```
'https://data.police.uk/api/outcomes-at-location?date=2019-12&lat=53.40146&lng=-2.96459'
```

You can check on your browser manually if this works:

```
IFrame(url, 300, 150)
```

```
    <iframe
        width="300"
        height="150"
        src="https://data.police.uk/api/outcomes-at-location?date=2020-01&lat=53.40146&lng=-2.96459"
        frameborder="0"
        allowfullscreen
    ></iframe>
```

Now we'll access it programmatically using `requests`:

```
%time r = requests.get(url)
```

```
# First 100 characters only
r.content[:100]
```

```
b'[{"category":{"code":"awaiting-court-result","name":"Awaiting court outcome"},"date":"2019-12","pers'
```

This is encoded in bytes, we need to decode it into a string and convert it into a data structure that we can work with.

```
crimes = json.loads(r.content)
```

```
type(crimes)
```

```
list
```

```
len(crimes)
```

```
1337
```

```
crimes[0]
```

```
{'category': {'code': 'awaiting-court-result',
  'name': 'Awaiting court outcome'},
```

```
 'date': '2019-12',
 'person_id': None,
 'crime': {'category': 'violent-crime',
  'location_type': 'Force',
  'location': {'latitude': '53.409256',
   'street': {'id': 911941, 'name': 'On or near Norton Street'},
   'longitude': '-2.974566'},
  'context': '',
  'persistent_id': '2fee8abfce463f0fd79844b4d4c630c0aacfdd5b127a9f77192eeeafde9433a7',
  'id': 79311446,
  'location_subtype': 'ROAD',
  'month': '2019-11'}}
```

Now this looks more Pythonic! Let's convert it into a `pandas.DataFrame`:

```
crimes_db = pandas.DataFrame(crimes)
crimes_db.head()
```

category
date
person_id
crime
0
{'code': 'awaiting-court-result', 'name': 'Awa…
2019-12
None
{'category': 'violent-crime', 'location_type':…
1
{'code': 'no-further-action', 'name': 'Investi…
2019-12
None
{'category': 'burglary', 'location_type': 'For…
2
{'code': 'no-further-action', 'name': 'Investi…
2019-12
None
{'category': 'other-theft', 'location_type': '…
3
{'code': 'unable-to-prosecute', 'name': 'Unabl…
2019-12
None
{'category': 'violent-crime', 'location_type':…
4
{'code': 'drugs-possession-warning', 'name': '…
2019-12

None

{‘category’: ‘drugs’, ‘location_type’: ‘Force’…

Now, for some applications this might be fine and we’d be good to go. For mapping however, this does not give us a working location. The reason is that the output from the API is more complex than a flat dictionary.

```
crimes[0]
```

{‘category’: {‘code’: ‘awaiting-court-result’, ‘name’: ‘Awaiting court outcome’}, ‘date’: ‘2019-12’, ‘person_id’: None, ‘crime’: {‘category’: ‘violent-crime’, ‘location_type’: ‘Force’, ‘location’: {‘latitude’: ‘53.409256’, ‘street’: {‘id’: 911941, ‘name’: ‘On or near Norton Street’}, ‘longitude’: ‘-2.974566’}, ‘context’: ’‘, ’persistent_id’: ‘2fee8abfce463f0fd79844b4d4c630c0aacfdd5b127a9f77192eeeafde9433a7’, ‘id’: 79311446, ‘location_subtype’: ‘ROAD’, ‘month’: ‘2019-11’}}

To pull out the lon/lat coordinates, we need to drill deeper into the `location` entry, which is within the `crime` bit:

```
crimes[0]["crime"]["location"]
```

{‘latitude’: ‘53.409256’, ‘street’: {‘id’: 911941, ‘name’: ‘On or near Norton Street’}, ‘longitude’: ‘-2.974566’}

Parsing a single crime event

For convenience, let’s pull out a single crime:

```
cr = crimes[0]
cr
```

{‘category’: {‘code’: ‘awaiting-court-result’, ‘name’: ‘Awaiting court outcome’}, ‘date’: ‘2019-12’, ‘person_id’: None, ‘crime’: {‘category’: ‘violent-crime’, ‘location_type’: ‘Force’, ‘location’: {‘latitude’: ‘53.409256’, ‘street’: {‘id’: 911941, ‘name’: ‘On or near Norton Street’}, ‘longitude’: ‘-2.974566’}, ‘context’: ’‘, ’persistent_id’: ‘2fee8abfce463f0fd79844b4d4c630c0aacfdd5b127a9f77192eeeafde9433a7’, ‘id’: 79311446, ‘location_subtype’: ‘ROAD’, ‘month’: ‘2019-11’}}

```
cr.keys()
```

dict_keys([‘category’, ‘date’, ‘person_id’, ‘crime’])

Let’s say, for every crime event, we want to keep the following attributes:

- Category code

- Crime ID, category, and location (lon/lat, as a geometry)
- Date
- Location type and subtype

Manually, we can extract those bits into a `Series` object:

```
cr_parsed = pandas.Series({
       'category_code': cr['category']['code'],\
       'crime_category': cr['crime']['category'], \
       'cime_id': cr['crime']['id'],\
       'cime_category': cr['crime']['category'], \
       'longitude': float(cr['crime']['location']['longitude']), \
       'latitude': float(cr['crime']['location']['latitude']), \
       'date': cr['crime']['month'], \
       'crime_location_type': cr['crime']['location_type'], \
       'crime_location_subtype': cr['crime']['location_subtype']
                 })
cr_parsed
```

category_code local-resolution crime_category drugs cime_id 80013657 cime_category drugs longitude -2.9654 latitude 53.3885 date 2019-12 crime_location_type Force crime_location_subtype ROAD dtype: object

One way to move into a more automated version is to turn the above into a small function:

```
def parse_crime_event(cr):
    cr_parsed = pandas.Series({
           'category_code': cr['category']['code'],\
           'crime_category': cr['crime']['category'], \
           'cime_id': cr['crime']['id'],\
           'cime_category': cr['crime']['category'], \
           'longitude': float(cr['crime']['location']['longitude']), \
           'latitude': float(cr['crime']['location']['latitude']), \
           'date': cr['crime']['month'], \
           'crime_location_type': cr['crime']['location_type'], \
           'crime_location_subtype': cr['crime']['location_subtype']
                     })
    return cr_parsed
```

Which means we can use it on any crime event with the same structure:

```
parse_crime_event(cr)
```

category_code awaiting-court-result crime_category violent-crime cime_id 79311446 cime_category violent-crime longitude -2.974566 latitude 53.409256 date 2019-11 crime_location_type Force crime_location_subtype ROAD dtype: object

```
parse_crime_event(crimes[10])
```

category_code no-further-action crime_category criminal-damage-arson cime_id 79308783 cime_category criminal-damage-arson longitude -2.975044 latitude 53.389998 date 2019-11 crime_location_type Force crime_location_subtype ROAD dtype: object

To note:

- It works because argument is `cr` and the `cr` object is used throughout
- A function (method) can be applied in any context where the object passed as `cr` (it doesn't need to be named that way) meets the expectations of the method (ie. has all the attributes it expects)

Extending to all crimes in the batch

Armed with the function above, we can apply it to all the crimes in our initial batch. For this, we will use a `for` loop:

```
# Start an empty list to store parsed crimes dynamically
parsed = []

# Loop over each crime event in the list of crimes
for cr in crimes:
    # Parse a single crime
    pc = parse_crime_event(cr)
    # Store the parsed crime into the list created
    parsed.append(pc)

# Conver the list into a DataFrame
parsed = pandas.DataFrame(parsed)


parsed.head()
```

category_code
crime_category
cime_id

cime_category
longitude
latitude
date
crime_location_type
crime_location_subtype
0
awaiting-court-result
violent-crime
79311446
violent-crime
-2.974566
53.409256
2019-11
Force
ROAD
1
no-further-action
burglary
80011089
burglary
-2.975439
53.408602
2019-12
Force
ROAD
2
no-further-action
other-theft
80015947
other-theft
-2.978599
53.409277
2019-12
Force
ROAD
3
unable-to-prosecute
violent-crime
79309151
violent-crime
-2.984887
53.408614
2019-11

Force
ROAD
4
drugs-possession-warning
drugs
80007412
drugs
-2.964395
53.390561
2019-12
Force
ROAD

Now, to complete the job and be able to work with these data in a (web) mapping environment, we need to turn lon/lat coordinates into geometries. For that, we will rely on `geopandas` (see here for more detail).

```
points = geopandas.points_from_xy(parsed["longitude"],
                                  parsed["latitude"]
                                 )
geo_db = geopandas.GeoDataFrame(parsed,
                                geometry=points,
                                crs="EPSG:4326"
                               )

geo_db.plot()
```

<matplotlib.axes._subplots.AxesSubplot at 0x7fb2d50fba90>

Exercise Explore the API for "Stop and search" by area data from the Police API and try to obtain data through it. Specifically try:

1. Request data using the "Specific point" endpoint to pull down a batch of data around the Roxby Building.
2. For the API ninjas out there (ie. optional), try to extract data using the "Custom area" endpoint. For example, create a polygon using `geojson.net` and query data available within the polygon.

Basemaps API

This section will cover the access of basemaps served as tilesets through the standard XYZ protocol. For this, we will use the `contextily`

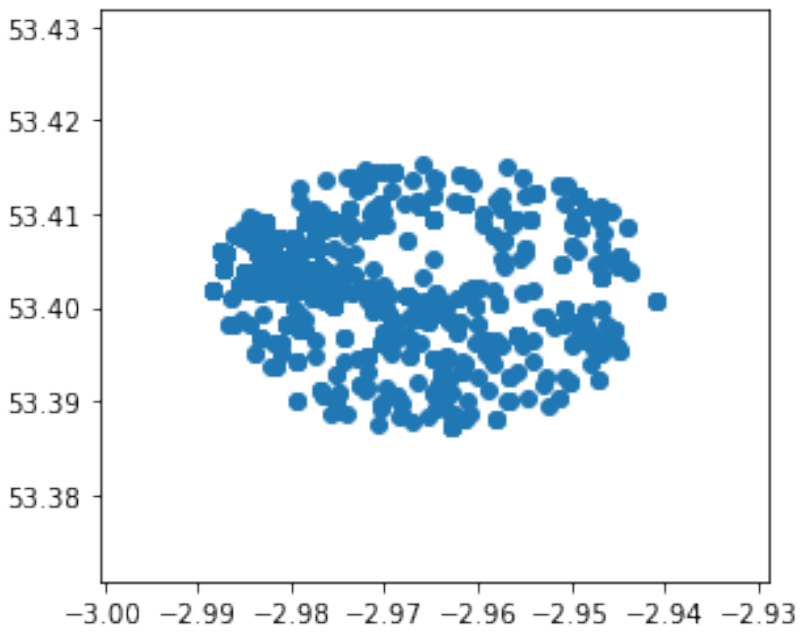

Figure 1: png

library, first as end-users, and then we will peak a bit into its guts to get a better understanding of its inner workings.

The XYZ protocol exposes maps as images for portions of the Earth we will call tiles. The XYZ name stands from the "coordinates" used to locate a given tile. This of the entire planet split up into squares, each of them available with a unique combination of X and Y numbers. Now add a third one (Z) for the zoom level: lower values use less tiles to cover the world, while higher resolution levels (higher Z) will cover progressively smaller areas, but with more detail.

Most XYZ APIs expose tiles directly over HTTP, which means we can access them from the browser. A handy tool in this context is the `mercantile` library, which handles conversions from lon/lat coordinates into tiles. For example, to get the tile XYZ for the coordinates we used above, a zoom level 12, we can run:

```
mercantile.tile(-2.96459, 53.40146, 11)
```

Tile(x=1007, y=663, z=11)

We can access now the Stamen Terrain tile for that tile:

```
tile_url = ("http://tile.stamen.com/terrain/"\
            "11/1007/663.jpg")
tile_url
```

'http://tile.stamen.com/terrain/11/1007/663.jpg'

```
IFrame(tile_url, 300, 300)
```

Now, to make a full basemap of an arbitrary extent at a given zoom level, we would need to identify all the tiles required, collect them, and compose a map that stitches them into a continuous mosaic. Luckily for us, the `contextily` library does exactly that for us. There are two different ways of using it.

First, the easy way. The most straightforward way to make a basemap with `contextily` is to provide a `GeoDataFrame`, plot it, and the add a basemap in a given CRS. For example, let's used the `geo_db` table from the Police API:

```
ax = geo_db.plot(markersize=0.5, color='red')
cx.add_basemap(ax, crs=geo_db.crs)
```

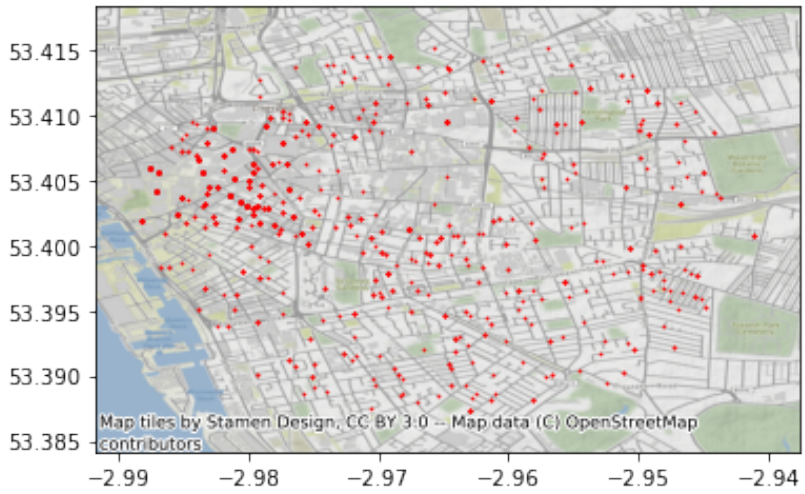

Figure 2: png

Exercise The `contextily.providers` module provides access to several tile providers:

```
cx.providers.keys()
```

dict_keys(['OpenStreetMap', 'OpenSeaMap', 'OpenPtMap', 'OpenTopoMap', 'OpenRailwayMap', 'OpenFireMap', 'SafeCast', 'Thunderforest', 'OpenMapSurfer', 'Hydda', 'MapBox', 'Stamen', 'Esri', 'OpenWeatherMap', 'HERE', 'FreeMapSK', 'MtbMap', 'CartoDB', 'HikeBike', 'BasemapAT', 'nlmaps', 'NASAGIBS', 'NLS', 'JusticeMap', 'Wikimedia', 'GeoportailFrance', 'OneMapSG'])

To swap the default for a particular provider, you need to use the `url` attribute. For example, to use CARTO instead of Stamen:

```
ax = geo_db.plot(markersize=0.5, color='red')
cx.add_basemap(ax,
               crs=geo_db.crs,
               url=cx.providers.CartoDB.Voyager,
               zoom=14
              )
```

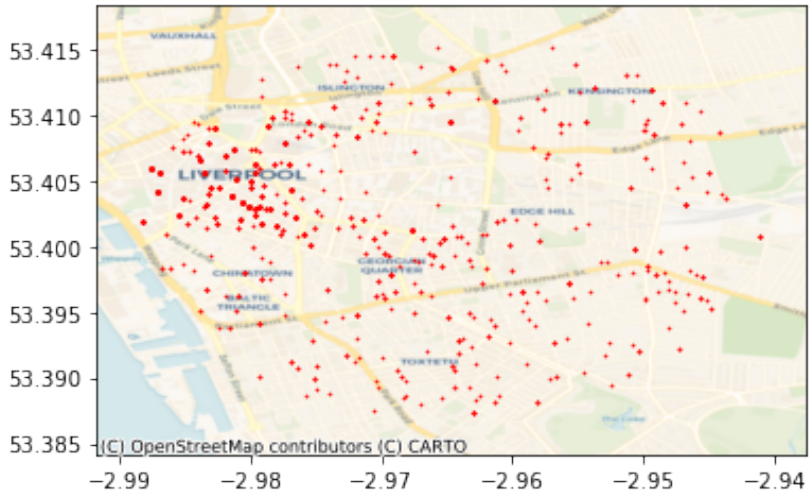

Figure 3: png

Explore the different providers and create similar maps as above using them.

Second, the flexible way. Sometimes, we don't necessarily have a layer to plot under, or we want to have a bit more control over what we actually pull down. For these cases, we'll swap `add_basemap` for `bounds2img`, which allows us to pass an arbitrary bounding box and retrieve a stitched up tile.

For example, we will replicate the basemap above:

```
geo_db.total_bounds
```

array([-2.98822 , 53.387346, -2.941042, 53.415206])

```
w, s, e, n = geo_db.total_bounds
```

With the bounding box, we can pull down the image (and its extent):

```
img, ext = cx.bounds2img(w, s, e, n, ll=True)
```

```
plt.imshow(img, extent=ext)
```

```
<matplotlib.image.AxesImage at 0x7fb2d49da908>
```

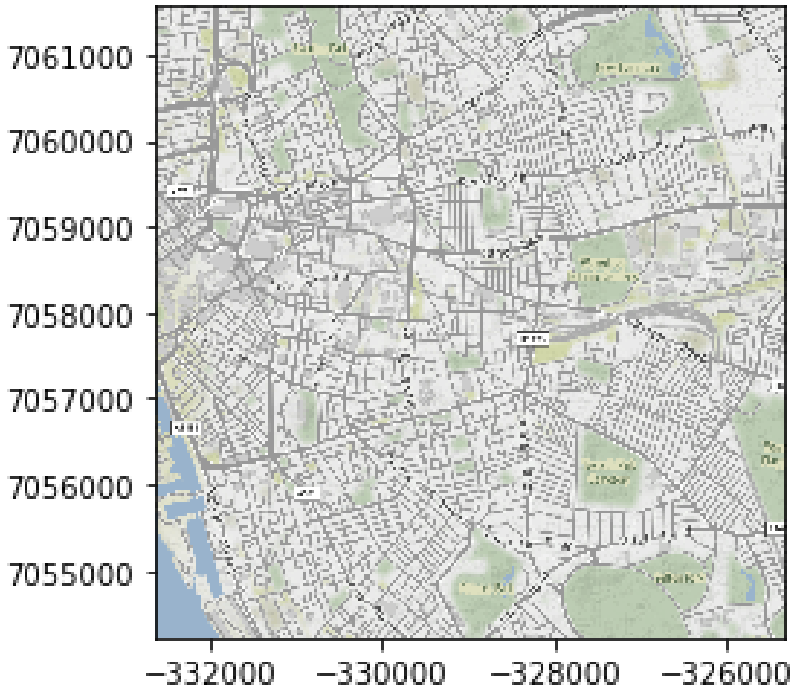

Figure 4: png

A final trick that `contextily` has under its hand is the ability to write a basemap into a `.tiff` file so we can access it in a standard GIS, such as QGIS. The interface is the same as in `bounds2img`, but in this case, we will be calling `bounds2raster`:

```
img, ext = cx.bounds2raster(w,
                            s,
                            e,
                            n,
                            "my_basemap.tif",
                            ll=True
                           )
```

Fire up QGIS and inspect the file we've just created (`my_basemap.tif`).

Directions API

To finish this lab, we will explore an API that allows us to tap into the output of computations that take place in the cloud, rather than a direct database. In particular, we will play witht he Mapbox Directions API. For

---

Exercise Explore the documentation for the isochrone API and try to obtain results. For example, retrieve the area that can be reached within 15 minutes of the Roxby Building.

Exercise (II) Explore the documentation for the geocoding API and try to use it to automatically embed coordinates between addresses.
