gmaps Documentation

Release 0.5.5-dev

Pascal Bugnion

Jul 31, 2017

Contents

CHAPTER 1

Installation

Installing *gmaps* with *conda*

The easiest way to install *gmaps* is with *conda*:

```
$ conda install -c conda-forge gmaps
```

Installing *gmaps* with *pip*

If you do not use conda, you can install *gmaps* with pip. The current version of *gmaps* is only tested with *IPython 4.2* or later and *ipywidgets 6.0.0* or later. To upgrade to the latest versions, use:

```
$ pip install -U jupyter
```

Make sure that you have enabled widgets extensions to Jupyter:

```
$ jupyter nbextension enable --py --sys-prefix widgetsnbextension
```

You can then install gmaps with:

```
$ pip install gmaps
```

Then tell Jupyter to load the extension with:

```
$ jupyter nbextension enable --py --sys-prefix gmaps
```

Development version

You must have NPM to install the development version. You can install NPM with your package manager.

We strongly recommend installing *jupyter-gmaps* in a virtual environment (either a conda environment or a virtualenv environment).

Clone the git repository by running:

```
$ git clone https://github.com/pbugnion/gmaps.git
```

For the initial installation, run:

```
$ ./dev-install
```

This installs `gmaps` in editable mode and installs the Javascript components as symlinks.

If you then make changes to the code, you can make those changes available to a running notebook server by:

- restarting the kernel if you have made changes to the Python source code
- running `npm run update` in the `js/` directory and *refreshing* the browser page containing the notebook if you have made changes to the JavaScript source. You do not need to restart the kernel.
- running `npm run update`, refreshing the browser and restarting the kernel if you have made changes to both the Python and JavaScript source.

You should not need to restart the notebook server.

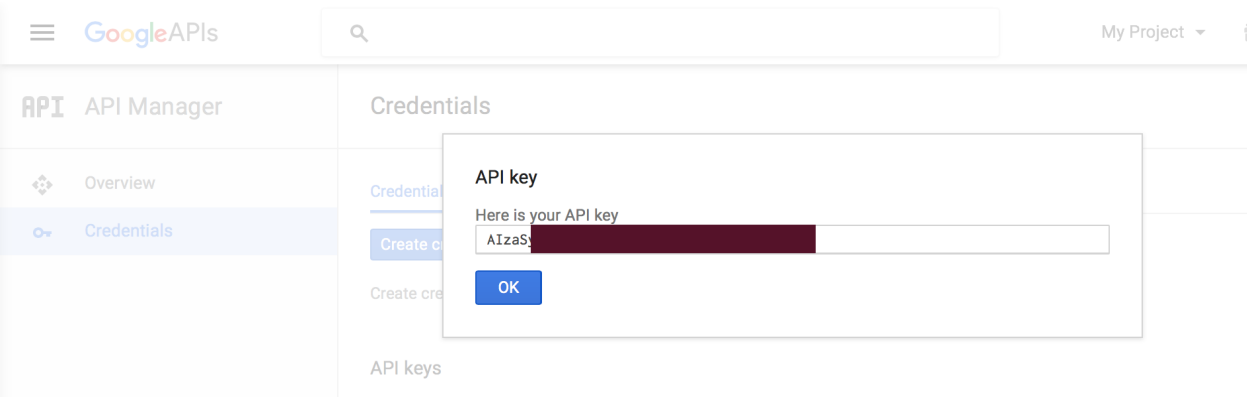
Source code

The *jupyter-gmaps* source is available on GitHub.

CHAPTER 2

Authentication

Most operations on Google Maps require that you tell Google who you are. To authenticate with Google Maps, follow the instructions for creating an API key. You will probably want to create a new project, then click on the *Credentials* section and create a *Browser key*. The API key is a string that starts with the letters `AI`.



You can pass this key to *gmaps* with the `configure` method:

```
gmaps.configure(api_key="AI...")
```

Maps and layers created after the call to `gmaps.configure` will have access to the API key.

You should avoid hard-coding the API key into your Jupyter notebooks. You can use environment variables. Add the following line to your shell start-up file (probably *~/.profile* or *~/.bashrc*):

```
export GOOGLE_API_KEY=AI...
```

Make sure you don't put spaces around the = sign. If you then open a *new* terminal window and type `env` at the command prompt, you should see that your API key. Start a new Jupyter notebook server in a new terminal, and type:

```
import os
import gmaps
```

```
gmaps.configure(api_key=os.environ["GOOGLE_API_KEY"])
```

CHAPTER 3

Getting started

gmaps is a plugin for Jupyter for embedding Google Maps in your notebooks. It is designed as a data visualization tool.

To demonstrate *gmaps*, let's plot the earthquake dataset, included in the package:

```
import gmaps
import gmaps.datasets

gmaps.configure(api_key="AI...") # Fill in with your API key

earthquake_df = gmaps.datasets.load_dataset_as_df("earthquakes")
earthquake_df.head()
```

The earthquake data has three columns: a latitude and longitude indicating the earthquake's epicentre and a weight denoting the magnitude of the earthquake at that point. Let's plot the earthquakes on a Google map:

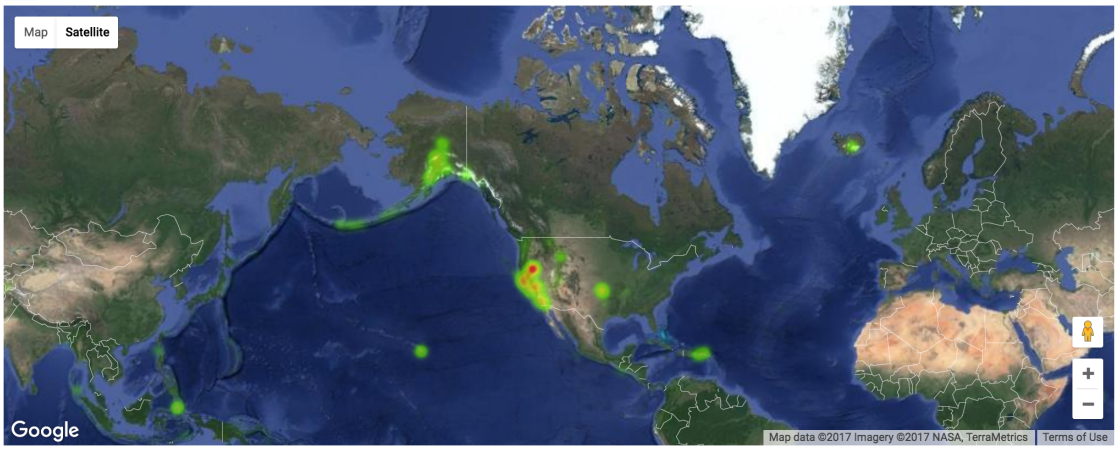
```
locations = earthquake_df[["latitude", "longitude"]]
weights = earthquake_df["magnitude"]
fig = gmaps.figure()
fig.add_layer(gmaps.heatmap_layer(locations, weights=weights))
fig
```

```
In [2]: earthquake_df = gmaps.datasets.load_dataset_as_df("earthquakes")
        earthquake_df.head()
```

Out[2]:

	latitude	longitude	magnitude
0	65.193300	-149.072500	1.70
1	38.791832	-122.780830	2.10
2	38.818001	-122.792168	0.48
3	33.601667	-116.727667	0.78
4	37.378334	-118.520836	3.64

```
In [3]: locations = earthquake_df[["latitude", "longitude"]]
        weights = earthquake_df["magnitude"]
        fig = gmaps.figure()
        fig.add_layer(gmaps.heatmap_layer(locations, weights=weights))
        fig
```



This gives you a fully-fledged Google map. You can zoom in and out, switch to satellite view and even to street view if you really want. The heatmap adjusts as you zoom in and out.

Basic concepts

gmaps is built around the idea of adding layers to a base map. After you've authenticated with Google maps, you start by creating a figure, which contains a base map:

```
import gmaps
gmaps.configure(api_key="AI...")

fig = gmaps.figure()
fig
```

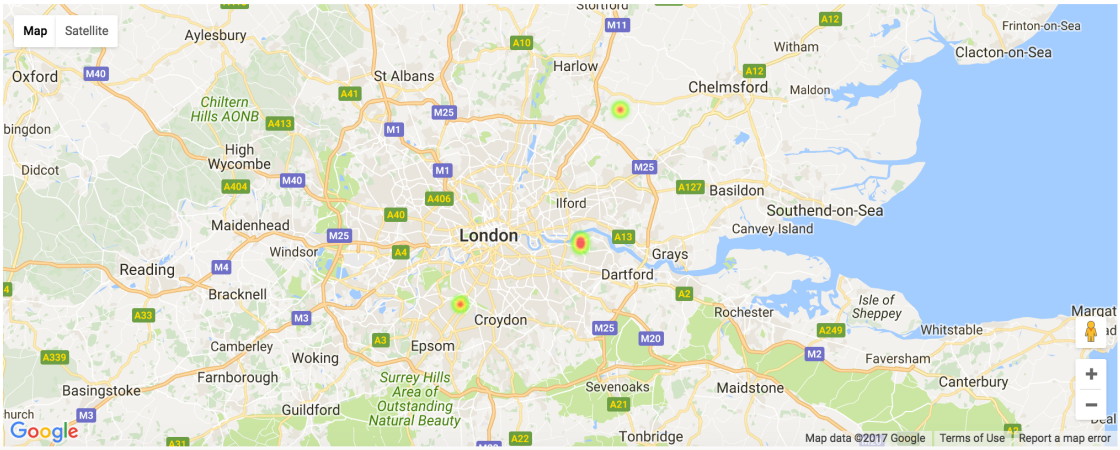
```
In [5]: import gmaps
        gmaps.configure(api_key="AIz...")

In [6]: fig = gmaps.figure()
        fig
```

You then add layers on top of the base map. For instance, to add a heatmap layer:

```
import gmaps
gmaps.configure(api_key="AI...")

fig = gmaps.figure()

# generate some (latitude, longitude) pairs
locations = [(51.5, 0.1), (51.7, 0.2), (51.4, -0.2), (51.49, 0.1)]

heatmap_layer = gmaps.heatmap_layer(locations)
fig.add_layer(heatmap_layer)
fig
```

```
In [5]: import gmaps
        gmaps.configure(api_key="AIz...")
```

```
In [7]: fig = gmaps.figure()

        # generate some (latitude, longitude) pairs
        locations = [(51.5, 0.1), (51.7, 0.2), (51.4, -0.2), (51.49, 0.1)]

        heatmap_layer = gmaps.heatmap_layer(locations)
        fig.add_layer(heatmap_layer)
        fig
```



The *locations* array can either be a list of tuples, as in the example above, a numpy array of shape $N times 2$ or a dataframe with two columns.

Most attributes on the base map and the layers can be set through named arguments in the constructor or as instance attributes once the instance is created. These two constructions are thus equivalent:

```
heatmap_layer = gmaps.heatmap_layer(locations)
heatmap_layer.point_radius = 8
```

and:

```
heatmap_layer = gmaps.heatmap_layer(locations, point_radius=8)
```

The former construction is useful for modifying a map once it has been built. Any change in parameters will propagate to maps in which those layers are included.

Base maps

Your first action with *gmaps* will usually be to build a base map:

```
import gmaps
gmaps.configure(api_key="AI...")

gmaps.figure()
```

This builds an empty map. You can also set the zoom level and map center explicitly:

```
new_york_coordinates = (40.75, -74.00)
gmaps.figure(center=new_york_coordinates, zoom_level=12)
```

```
In [19]: new_york_coordinates = (40.75, -74.00)
         gmaps.figure(center=new_york_coordinates, zoom_level=12)
```

If you do not set the map zoom and center, the viewport will automatically focus on the data as you add it to the map.

Heatmaps

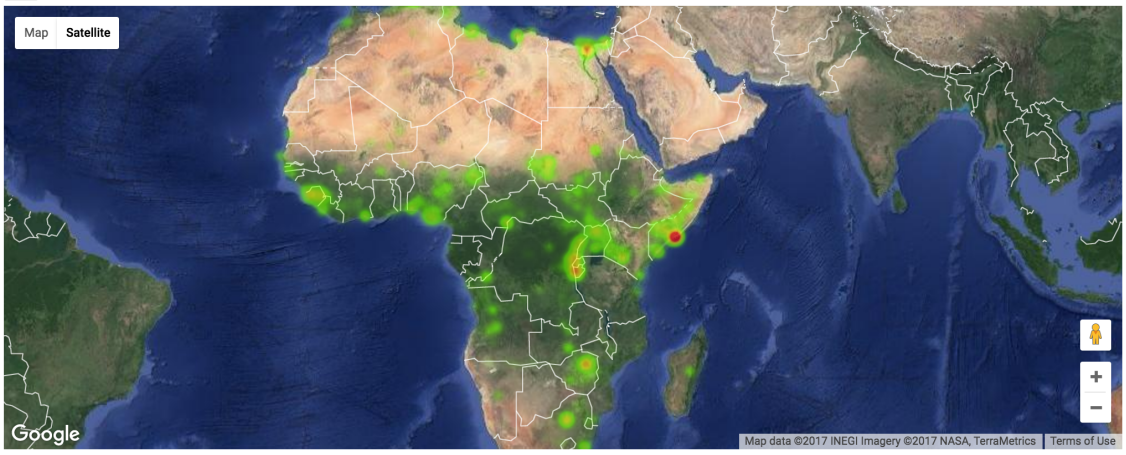
Heatmaps are a good way of getting a sense of the density and clusters of geographical events. They are a powerful tool for making sense of larger datasets. We will use a dataset recording all instances of political violence that occurred in Africa between 1997 and 2015. The dataset comes from the Armed Conflict Location and Event Data Project. This dataset contains about 110,000 rows.

```
import gmaps.datasets

locations = gmaps.datasets.load_dataset_as_df("acled_africa")

locations.head()
# => dataframe with 'longitude' and 'latitude' columns
```

We already know how to build a heatmap layer:

```
import gmaps
import gmaps.datasets
gmaps.configure(api_key="AI...")

locations = gmaps.datasets.load_dataset_as_df("acled_africa")
fig = gmaps.figure()
heatmap_layer = gmaps.heatmap_layer(locations)
fig.add_layer(heatmap_layer)
fig
```

```
In [8]: locations = gmaps.datasets.load_dataset_as_df("acled_africa")
        fig = gmaps.figure()
        heatmap_layer = gmaps.heatmap_layer(locations)
        fig.add_layer(heatmap_layer)
        fig
```



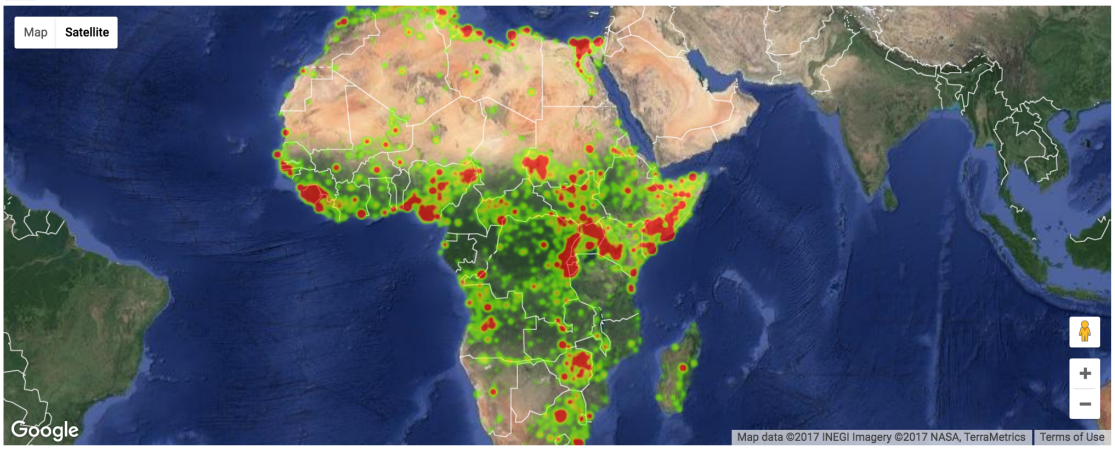
Preventing dissipation on zoom

If you zoom in sufficiently, you will notice that individual points disappear. You can prevent this from happening by controlling the `max_intensity` setting. This caps off the maximum peak intensity. It is useful if your data is strongly peaked. This settings is *None* by default, which implies no capping. Typically, when setting the maximum intensity, you also want to set the `point_radius` setting to a fairly low value. The only good way to find reasonable values for these settings is to tweak them until you have a map that you are happy with.:

```
heatmap_layer.max_intensity = 100
heatmap_layer.point_radius = 5
```

To avoid re-drawing the whole map every time you tweak these settings, you may want to set them in another noteobook cell:

```
In [8]: locations = gmaps.datasets.load_dataset_as_df("acled_africa")
        fig = gmaps.figure()
        heatmap_layer = gmaps.heatmap_layer(locations)
        fig.add_layer(heatmap_layer)
        fig
```




```
In [9]: heatmap_layer.max_intensity = 100
        heatmap_layer.point_radius = 5
```

Google maps also exposes a `dissipating` option, which is true by default. If this is true, the radius of influence of each point is tied to the zoom level: as you zoom out, a given point covers more physical kilometres. If you set it to false, the physical radius covered by each point stays fixed. Your points will therefore either be tiny at high zoom levels or large at low zoom levels.

Setting the color gradient and opacity

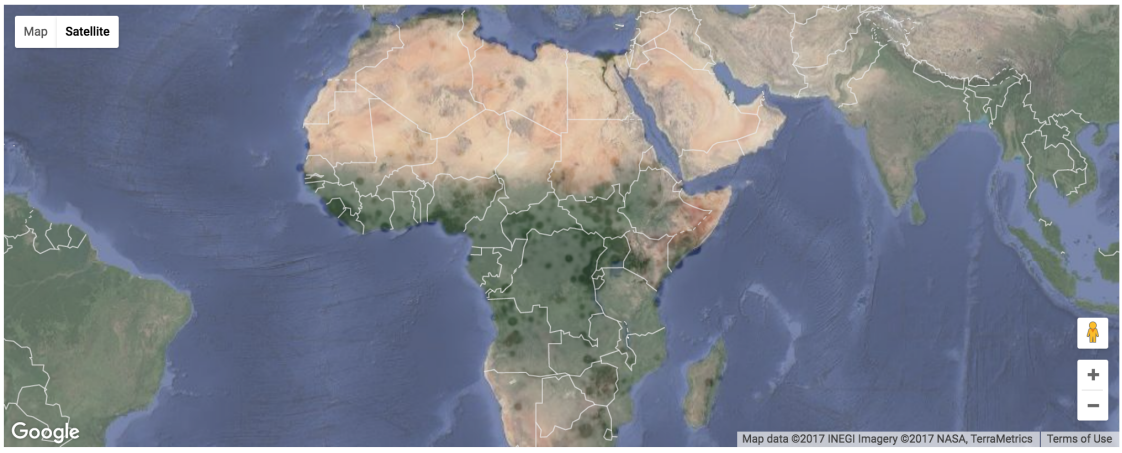
You can set the color gradient of the map by passing in a list of colors. Google maps will interpolate linearly between those colors. You can represent a color as a string denoting the color (the colors allowed by this):

```
heatmap_layer.gradient = [
    'white',
    'silver',
    'gray'
]
```

If you need more flexibility, you can represent colours as an RGB triple or an RGBA quadruple:

```
heatmap_layer.gradient = [
    (200, 200, 200, 0.6),
    (100, 100, 100, 0.3),
    (50, 50, 50, 0.3)
]
```

```
In [20]: locations = gmaps.datasets.load_dataset_as_df("acled_africa")
         fig = gmaps.figure()
         heatmap_layer = gmaps.heatmap_layer(locations)
         fig.add_layer(heatmap_layer)
         fig
```




```
In [21]: heatmap_layer.max_intensity = 100
         heatmap_layer.point_radius = 5
         heatmap_layer.gradient = [
             (200, 200, 200, 0.6),
             (100, 100, 100, 0.3),
             (50, 50, 50, 0.3)
         ]
```

You can also use the `opacity` option to set a single opacity across the entire colour gradient:

```
heatmap_layer.opacity = 0.0 # make the heatmap transparent
```

Weighted heatmaps

By default, heatmaps assume that every row is of equal importance. You can override this by passing weights through the *weights* keyword argument. The *weights* array is an iterable (e.g. a Python list or a Numpy array) or a single pandas series. Weights must all be positive (this is a limitation in Google maps itself).

```
import gmaps
import gmaps.datasets
gmaps.configure(api_key="AI...")

df = gmaps.datasets.load_dataset_as_df("earthquakes")
# dataframe with columns ('latitude', 'longitude', 'magnitude')

fig = gmaps.figure()
heatmap_layer = gmaps.heatmap_layer(
    df[["latitude", "longitude"]], weights=df["magnitude"],
    max_intensity=30, point_radius=3.0
)
fig.add_layer(heatmap_layer)
fig
```

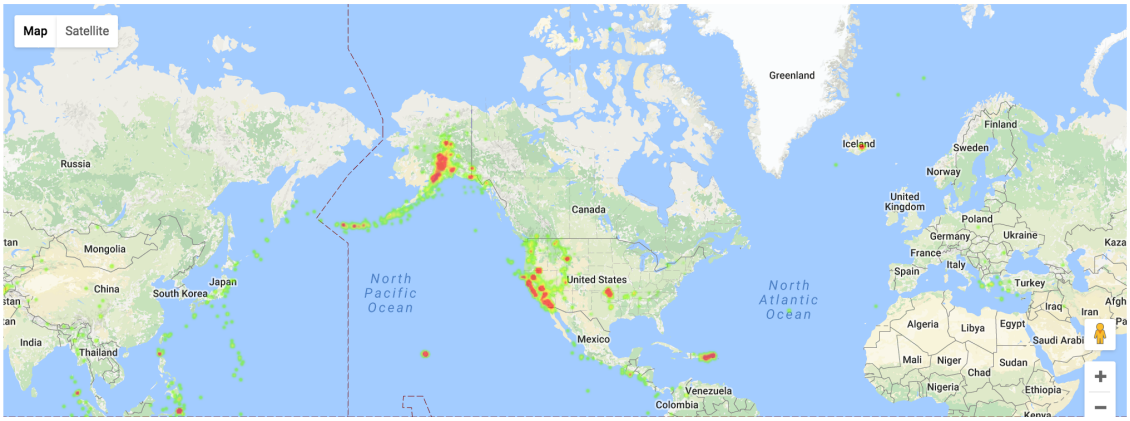
In [24]:
```
import gmaps
import gmaps.datasets
gmaps.configure(api_key="AIz...")
```

In [25]:
```
df = gmaps.datasets.load_dataset_as_df("earthquakes")
df.head()
```

Out[25]:

	latitude	longitude	magnitude
0	65.193300	-149.072500	1.70
1	38.791832	-122.780830	2.10
2	38.818001	-122.792168	0.48
3	33.601667	-116.727667	0.78
4	37.378334	-118.520836	3.64

In [26]:
```
fig = gmaps.figure()
heatmap_layer = gmaps.heatmap_layer(
    df[["latitude", "longitude"]], weights=df["magnitude"],
    max_intensity=30, point_radius=3.0
)
fig.add_layer(heatmap_layer)
fig
```



Markers and symbols

We can add a layer of markers to a Google map. Each marker represents an individual data point:

```
import gmaps
gmaps.configure(api_key="AI...")

marker_locations = [
    (-34.0, -59.166672),
    (-32.23333, -64.433327),
    (40.166672, 44.133331),
    (51.216671, 5.0833302),
    (51.333328, 4.25)
]

fig = gmaps.figure()
markers = gmaps.marker_layer(marker_locations)
fig.add_layer(markers)
fig
```

```
In [27]: marker_locations = [
             (-34.0, -59.166672),
             (-32.23333, -64.433327),
             (40.166672, 44.133331),
             (51.216671, 5.0833302),
             (51.333328, 4.25)
         ]

         fig = gmaps.figure()
         markers = gmaps.marker_layer(marker_locations)
         fig.add_layer(markers)
         fig
```

We can also attach a pop-up box to each marker. Clicking on the marker will bring up the info box. The content of the box can be either plain text or html:

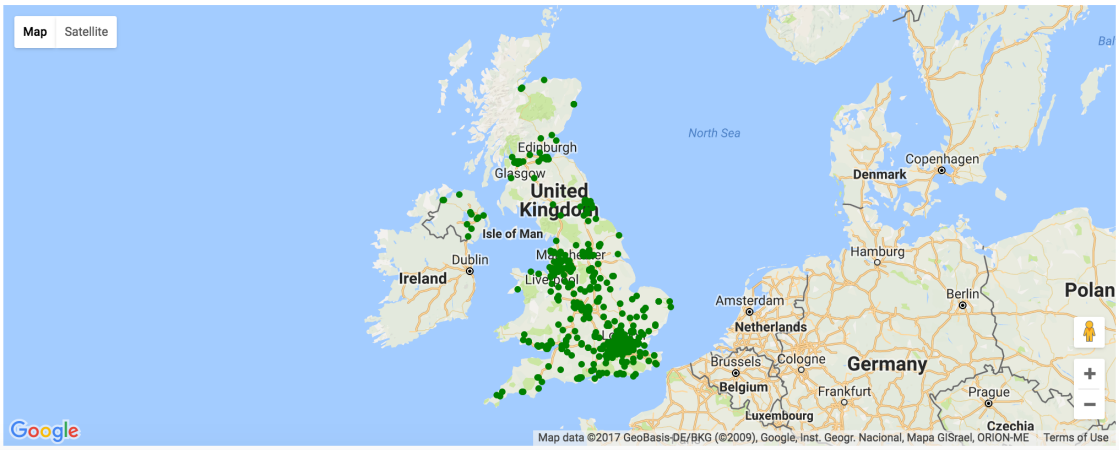
```
import gmaps
gmaps.configure(api_key="AI...")

nuclear_power_plants = [
    {"name": "Atucha", "location": (-34.0, -59.167), "active_reactors": 1},
    {"name": "Embalse", "location": (-32.2333, -64.4333), "active_reactors": 1},
    {"name": "Armenia", "location": (40.167, 44.133), "active_reactors": 1},
    {"name": "Br", "location": (51.217, 5.083), "active_reactors": 1},
    {"name": "Doel", "location": (51.333, 4.25), "active_reactors": 4},
    {"name": "Tihange", "location": (50.517, 5.283), "active_reactors": 3}
]

plant_locations = [plant["location"] for plant in nuclear_power_plants]
info_box_template = """
<dl>
<dt>Name</dt><dd>{name}</dd>
<dt>Number reactors</dt><dd>{active_reactors}</dd>
</dl>
"""
plant_info = [info_box_template.format(**plant) for plant in nuclear_power_plants]

marker_layer = gmaps.marker_layer(plant_locations, info_box_content=plant_info)
fig = gmaps.figure()
fig.add_layer(marker_layer)
fig
```

In [29]:
```
nuclear_power_plants = [
    {"name": "Atucha", "location": (-34.0, -59.167), "active_reactors": 1},
    {"name": "Embalse", "location": (-32.2333, -64.4333), "active_reactors": 1},
    {"name": "Armenia", "location": (40.167, 44.133), "active_reactors": 1},
    {"name": "Br", "location": (51.217, 5.083), "active_reactors": 1},
    {"name": "Doel", "location": (51.333, 4.25), "active_reactors": 4},
    {"name": "Tihange", "location": (50.517, 5.283), "active_reactors": 3}
]

plant_locations = [plant["location"] for plant in nuclear_power_plants]
info_box_template = """
<dl>
<dt>Name</dt><dd>{name}</dd>
<dt>Number reactors</dt><dd>{active_reactors}</dd>
</dl>
"""
plant_info = [info_box_template.format(**plant) for plant in nuclear_power_plants]

marker_layer = gmaps.marker_layer(plant_locations, info_box_content=plant_info)
fig = gmaps.figure()
fig.add_layer(marker_layer)
fig
```

Markers are currently limited to the Google maps style drop icon. If you need to draw more complex shape on maps, use the `symbol_layer` function. Symbols represent each *latitude*, *longitude* pair with a circle whose colour and size you can customize. Let's, for instance, plot the location of every Starbuck's coffee shop in the UK:

```
import gmaps
import gmaps.datasets

gmaps.configure(api_key="AI...")

df = gmaps.datasets.load_dataset_as_df("starbucks_kfc_uk")

starbucks_df = df[df["chain_name"] == "starbucks"]
starbucks_df = starbucks_df[['latitude', 'longitude']]

starbucks_layer = gmaps.symbol_layer(
    starbucks_df, fill_color="green", stroke_color="green", scale=2
)
fig = gmaps.figure()
fig.add_layer(starbucks_layer)
fig
```

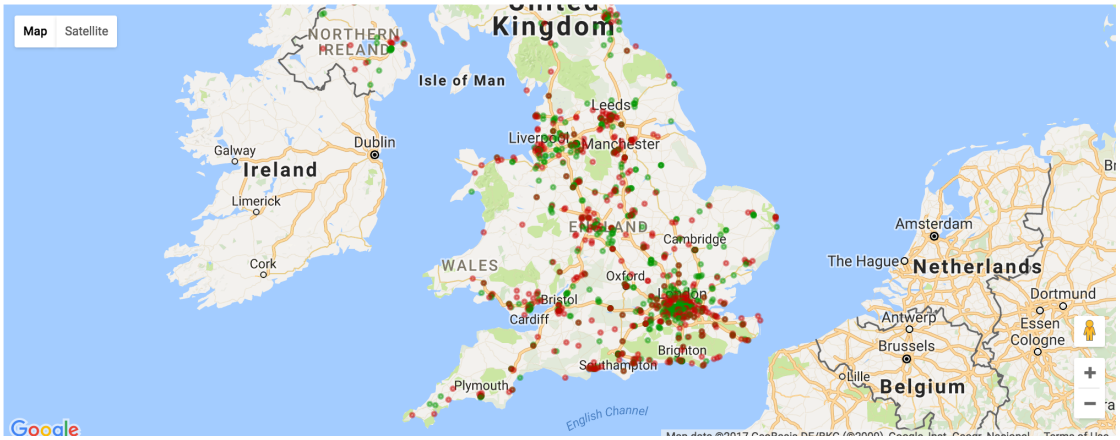
```
In [45]: df = gmaps.datasets.load_dataset_as_df("starbucks_kfc_uk")

         starbucks_df = df[df["chain_name"] == "starbucks"]
         starbucks_df = starbucks_df[['latitude', 'longitude']]

         starbucks_layer = gmaps.symbol_layer(
             starbucks_df, fill_color="green", stroke_color="green", scale=2
         )
         fig = gmaps.figure()
         fig.add_layer(starbucks_layer)
         fig
```



You can have several layers of markers. For instance, we can compare the locations of Starbucks coffee shops and KFC outlets in the UK by plotting both on the same map:

```
import gmaps
import gmaps.datasets

gmaps.configure(api_key="AI...")

df = gmaps.datasets.load_dataset_as_df("starbucks_kfc_uk")

starbucks_df = df[df["chain_name"] == "starbucks"]
starbucks_df = starbucks_df[['latitude', 'longitude']]

kfc_df = df[df["chain_name"] == "kfc"]
kfc_df = kfc_df[['latitude', 'longitude']]


starbucks_layer = gmaps.symbol_layer(
    starbucks_df, fill_color="rgba(0, 150, 0, 0.4)",
    stroke_color="rgba(0, 150, 0, 0.4)", scale=2
)

kfc_layer = gmaps.symbol_layer(
    kfc_df, fill_color="rgba(200, 0, 0, 0.4)",
    stroke_color="rgba(200, 0, 0, 0.4)", scale=2
)

fig = gmaps.figure()
fig.add_layer(starbucks_layer)
fig.add_layer(kfc_layer)
fig
```

```
In [54]: df = gmaps.datasets.load_dataset_as_df("starbucks_kfc_uk")

         starbucks_df = df[df["chain_name"] == "starbucks"]
         starbucks_df = starbucks_df[['latitude', 'longitude']]

         kfc_df = df[df["chain_name"] == "kfc"]
         kfc_df = kfc_df[['latitude', 'longitude']]


         starbucks_layer = gmaps.symbol_layer(
             starbucks_df, fill_color="rgba(0, 150, 0, 0.4)",
             stroke_color="rgba(0, 150, 0, 0.4)", scale=2
         )

         kfc_layer = gmaps.symbol_layer(
             kfc_df, fill_color="rgba(200, 0, 0, 0.4)",
             stroke_color="rgba(200, 0, 0, 0.4)", scale=2
         )

         fig = gmaps.figure()
         fig.add_layer(starbucks_layer)
         fig.add_layer(kfc_layer)
         fig
```



Dataset size limitations

Google maps may become very slow if you try to represent more than a few thousand symbols or markers. If you have a larger dataset, you should either consider subsampling or use heatmaps.

GeoJSON layer

We can add GeoJSON to a map. This is very useful when we want to draw chloropleth maps.

You can either load data from your own GeoJSON file, or you can load one of the GeoJSON geometries bundled with *gmaps*. Let's start with the latter. We will create a map of the GINI coefficient (a measure of inequality) for every country in the world.

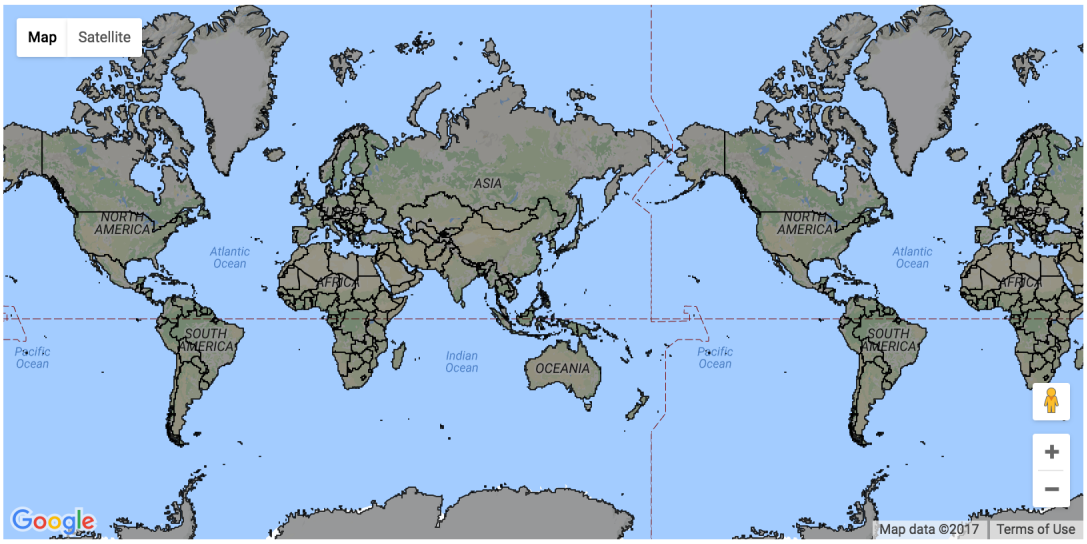
Let's start by just plotting the raw GeoJSON:

```
import gmaps
import gmaps.geojson_geometries
gmaps.configure(api_key="AIza...")

countries_geojson = gmaps.geojson_geometries.load_geometry('countries')

fig = gmaps.figure()
```

```
gini_layer = gmaps.geojson_layer(countries_geojson)
fig.add_layer(gini_layer)
fig
```

This just plots the country boundaries on top of a Google map.

```
In [60]: countries_geojson = gmaps.geojson_geometries.load_geometry('countries')

         fig = gmaps.figure()

         gini_layer = gmaps.geojson_layer(countries_geojson)
         fig.add_layer(gini_layer)
         fig
```



Next, we want to colour each country by a colour derived from its GINI index. We first need to map from each item in the GeoJSON document to a GINI value. GeoJSON documents are organised as a collection of *features*, each of which has the keys *geometry* and *properties*. For instance, for our countries:

```
>>> print(len(geojson['features']))
217 # corresponds to 217 distinct countries and territories
>>> print(geojson['features'][0])
{
  'type': 'Feature'
  'geometry': {'coordinates': [ ... ], 'type': 'Polygon'},
  'properties': {'ISO_A3': u'AFG', 'name': u'Afghanistan'}
}
```

As we can see, *properties* encodes meta-information about the feature, like the country name. We will use this name to look up a GINI value for that country and translate that into a colour. We can download a list of GINI coefficients for (nearly) every country using the *gmaps.datasets* module (you could load your own data here):

```
import gmaps.datasets
rows = gmaps.datasets.load_dataset('gini') # 'rows' is a list of tuples
country2gini = dict(rows) # dictionary mapping 'country' -> gini coefficient
print(country2gini['United Kingdom'])
# 32.4
```

We can now use the `country2gini` dictionary to map each country to a color. We will use a Matplotlib colormap to map from our GINI floats to a color that makes sense on a linear scale. We will use the Viridis colorscale:

```
from matplotlib.cm import viridis
from matplotlib.colors import to_hex

# We will need to scale the GINI values to lie between 0 and 1
min_gini = min(country2gini.values())
max_gini = max(country2gini.values())
gini_range = max_gini - min_gini

def calculate_color(gini):
    """
    Convert the GINI coefficient to a color
    """
    # make gini a number between 0 and 1
    normalized_gini = (gini - min_gini) / gini_range

    # invert gini so that high inequality gives dark color
    inverse_gini = 1.0 - normalized_gini

    # transform the gini coefficient to a matplotlib color
    mpl_color = viridis(inverse_gini)

    # transform from a matplotlib color to a valid CSS color
    gmaps_color = to_hex(mpl_color, keep_alpha=False)

    return gmaps_color
```

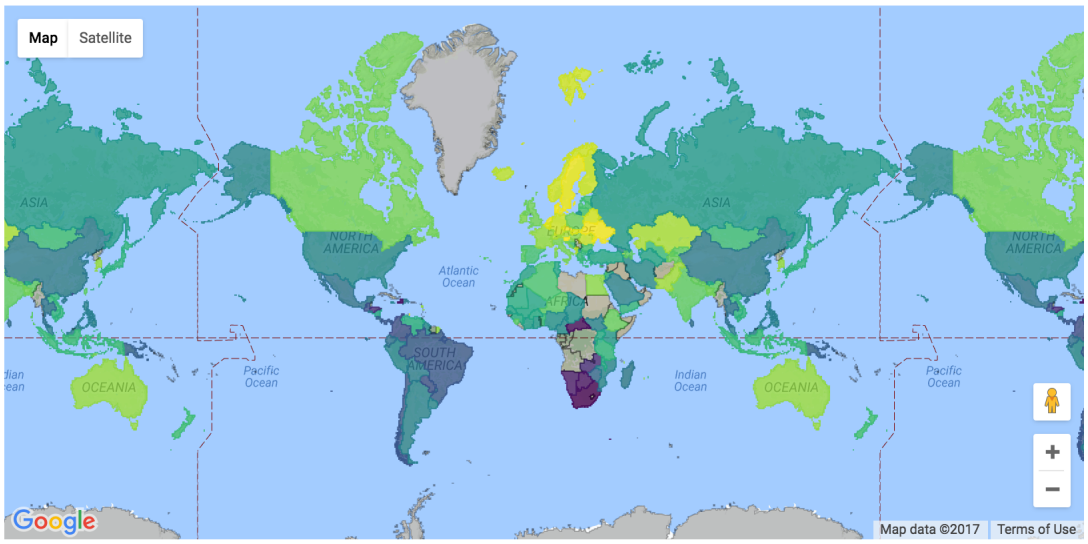
We now need to build an array of colors, one for each country, that we can pass to the GeoJSON layer. The easiest way to do this is to iterate over the array of features in the GeoJSON:

```
colors = []
for feature in countries_geojson['features']:
    country_name = feature['properties']['name']
    try:
        gini = country2gini[country_name]
        color = calculate_color(gini)
    except KeyError:
        # no GINI for that country: return default color
        color = (0, 0, 0, 0.3)
    colors.append(color)
```

We can now pass our array of colors to the GeoJSON layer:

```
fig = gmaps.figure()
gini_layer = gmaps.geojson_layer(
    countries_geojson,
    fill_color=colors,
    stroke_color=colors,
    fill_opacity=0.8)
fig.add_layer(gini_layer)
fig
```

```
In [62]: fig = gmaps.figure()
         gini_layer = gmaps.geojson_layer(
             countries_geojson,
             fill_color=colors,
             stroke_color=colors,
             fill_opacity=0.8)
         fig.add_layer(gini_layer)
         fig
```



GeoJSON geometries bundled with Gmaps

Finding appropriate GeoJSON geometries can be painful. To mitigate this somewhat, *gmaps* comes with its own set of curated GeoJSON geometries:

```
>>> import gmaps.geojson_geometries
>>> gmaps.geojson_geometries.list_geometries()
['brazil-states',
'england-counties',
'us-states',
'countries',
'india-states',
'us-counties',
'countries-high-resolution']

>>> gmaps.geojson_geometries.geometry_metadata('brazil-states')
{'description': 'US county boundaries',
 'source': 'http://eric.clst.org/Stuff/USGeoJSON'}
```

Use the *load_geometry* function to get the GeoJSON object:

```
import gmaps
import gmaps.geojson_geometries
gmaps.configure(api_key="AIza...")

countries_geojson = gmaps.geojson_geometries.load_geometry('brazil-states')

fig = gmaps.figure()
```

```
geojson_layer = gmaps.geojson_layer(countries_geojson)
fig.add_layer(geojson_layer)
fig
```

New geometries would greatly enhance the usability of *jupyter-gmaps*. Refer to this issue on GitHub for information on how to contribute a geometry.

Loading your own GeoJSON

So far, we have only considered visualizing GeoJSON geometries that come with *jupyter-gmaps*. Most of the time, though, you will want to load your own geometry. Use the standard library json module for this:

```
import json
import gmaps
gmaps.configure(api_key="AIza...")

with open("my_geojson_geometry.json") as f:
    geometry = json.load(f)

fig = gmaps.figure()
geojson_layer = gmaps.geojson_layer(geometry)
fig.add_layer(geojson_layer)
fig
```

Directions layer

gmaps supports drawing routes based on the Google maps directions service. At the moment, this only supports directions between points denoted by latitude and longitude:

```
import gmaps
import gmaps.datasets
gmaps.configure(api_key="AIza...")

# Latitude-longitude pairs
geneva = (46.2, 6.1)
montreux = (46.4, 6.9)
zurich = (47.4, 8.5)

fig = gmaps.figure()
geneva2zurich = gmaps.directions_layer(geneva, zurich)
fig.add_layer(geneva2zurich)
fig
```

In [63]:
```
geneva = (46.2, 6.1)
montreux = (46.4, 6.9)
zurich = (47.4, 8.5)

fig = gmaps.figure()
geneva2zurich = gmaps.directions_layer(geneva, zurich)
fig.add_layer(geneva2zurich)
fig
```

You can also pass waypoints and customise the directions request. You can pass up to 23 waypoints, and waypoints are not supported when the travel mode is `'TRANSIT'` (this is a limitation of the Google Maps directions service):

```
fig = gmaps.figure()
geneva2zurich_via_montreux = gmaps.directions_layer(
        geneva, zurich, waypoints=[montreux],
        travel_mode='BICYCLING')
fig.add_layer(geneva2zurich_via_montreux)
fig
```

```
In [4]: fig = gmaps.figure()
geneva2zurich_via_montreux =\
  gmaps.directions_layer(
      geneva, zurich, waypoints=[montreux], travel_mode='BICYCLING')
fig.add_layer(geneva2zurich_via_montreux)
fig
```

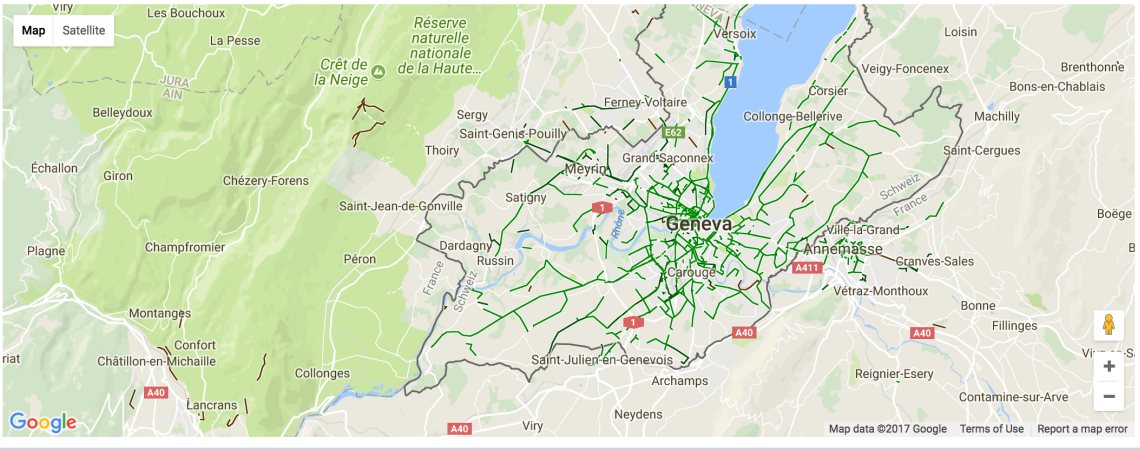
The full list of options is given as part of the documentation for the *gmaps.directions_layer()*.

Bicycling layer

You can add a bicycling layer to a map to draw cycle lanes. This will also change the style of the base layer to de-emphasize streets which are not cycle-friendly.

```
import gmaps
gmaps.configure(api_key="AI...")

fig = gmaps.figure()
fig.add_layer(gmaps.bicycling_layer())
fig
```

```
In [2]: fig = gmaps.figure()
fig.add_layer(gmaps.bicycling_layer())
fig
```



CHAPTER 4

Exporting maps

Exporting to PNG

You can save maps to PNG by clicking the *Download* button in the toolbar. This will download a static copy of the map.

This feature suffers from some know issues:

- there is no way to set the quality of the rendering at present,
- you cannot export maps that contain a *Directions* layer (see the issue on Github for details).

Exporting to HTML

You can export maps to HTML using the infrastructure provided by *ipywidgets*. In the menu of your notebook, click *Widgets* > *Embed widgets*. This will open a modal containing some HTML. The HTML is composed of several script tags:

```
<!-- Script tags that need to go into the head of the document -->
<script src="https://unpkg.com/jupyter-js-widgets@~2.1.4/dist/embed.js"></script>

<script type="application/vnd.jupyter.widget-state+json">
    // State of the widgets
</script>

<!--
    Script tags that should be embedded where the views should go.
    There should be one tag per cell containing a widget
-->
<script type="application/vnd.jupyter.widget-view+json">
  {
  "model_id": "c05e9b0ca0dd405295d2adde29776c95"
  }
```

```
</script>
<script type="application/vnd.jupyter.widget-view+json">
  {
  "model_id": "ea6c1d9f7d1c4065818fafa1ed9125aa"
  }
</script>
```

The first two define the state of the widgets and should be embedded in the head of the HTML document. The remaining tags describe the views. There will be one for each cell containing a widget in the notebook. You should paste these into the DOM elements that need to hold the views.

Additionally, you will need to import *jupyter-gmaps*, JQuery and the Bootstrap CSS in the head of your HTML document. You can use this template to make sure you have all the imports:

```
<html>
    <head>
        <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/
↪css/bootstrap.min.css">
        <script src="https://code.jquery.com/jquery-3.2.1.min.js"></script>

        {{ paste first two script tags here }}

    </head>
    <body>
        <div>
            {{ paste script tags defining the views here }}
        </div>
    </body>
</html>
```

Thus, a valid HTML document containing a single map would look like this (the API key has been redacted, but apart from that, this will work out of the box):

```
<html>
    <head>
        <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/
↪css/bootstrap.min.css">
        <script src="https://code.jquery.com/jquery-3.2.1.min.js"></script>

        <script src="https://unpkg.com/jupyter-js-widgets@~2.1.4/dist/embed.js"></
↪script>
        <script type="application/vnd.jupyter.widget-state+json">
        {
            "version_major": 1,
            "version_minor": 0,
            "state": {
                "acd855c3e79c4100bf23ac682b97fef6": {
                    "model_name": "LayoutModel",
                    "model_module": "jupyter-js-widgets",
                    "model_module_version": "~2.1.4",
                    "state": {
                        "_model_module_version": "~2.1.4",
                        "height": "400px",
                        "_view_module_version": "~2.1.4",
                        "align_self": "stretch"
                    }
                },
                "e6333a5e4408424fa1d13bafe32e3ec8": {
```

```
                    "model_name": "PlainmapModel",
                    "model_module": "jupyter-gmaps",
                    "model_module_version": "*",
                    "state": {
                        "layers": [],
                        "_dom_classes": [],
                        "msg_throttle": 1,
                        "_model_module_version": "*",
                        "_view_module_version": "*",
                        "data_bounds": [
                            [
                                46.2,
                                6.1
                            ],
                            [
                                47.2,
                                7.1
                            ]
                        ],
                        "layout": "IPY_MODEL_acd855c3e79c4100bf23ac682b97fef6",
                        "configuration": {
                            "api_key": "AIza_FILL_ME_IN"
                        }
                    }
                }
            }
        }
        </script>

    </head>

    <body>
        <h1>GMaps embedding example</h1>
        <div id="widget-embedded-here">
            <script type="application/vnd.jupyter.widget-view+json">
            {
                "model_id": "e6333a5e4408424fa1d13bafe32e3ec8"
            }
            </script>
        </div>
    </body>
</html>
```

CHAPTER 5

API documentation

Figures and layers

gmaps.**figure**(*display_toolbar=True*, *display_errors=True*, *zoom_level=None*, *center=None*)

Create a gmaps figure

This returns a *Figure* object to which you can add data layers.

Parameters

- **display_toolbar** (*boolean, optional*) – Boolean denoting whether to show the toolbar. Defaults to True.
- **display_errors** (*boolean, optional*) – Boolean denoting whether to show errors that arise in the client. Defaults to True.
- **zoom_level** (*int, optional*) – Integer between 0 and 21 indicating the initial zoom level. High values are more zoomed in. By default, the zoom level is chosen to fit the data passed to the map. If specified, you must also specify the map center.
- **center** (*tuple, optional*) – Latitude-longitude pair determining the map center. By default, the map center is chosen to fit the data passed to the map. If specified, you must also specify the zoom level.

Returns A *gmaps.Figure* widget.

Examples

```
>>> import gmaps
>>> gmaps.configure(api_key="AI...")
>>> fig = gmaps.figure()
>>> locations = [(46.1, 5.2), (46.2, 5.3), (46.3, 5.4)]
>>> fig.add_layer(gmaps.heatmap_layer(locations))
```

You can also explicitly specify the intiial map center and zoom:

```
>>> fig = gmaps.figure(center=(46.0, -5.0), zoom_level=8)
```

gmaps.**heatmap_layer**(*locations*, *weights=None*, *max_intensity=None*, *dissipating=True*, *point_radius=None*, *opacity=0.6*, *gradient=None*)

Create a heatmap layer.

This returns a *gmaps.Heatmap* or a *gmaps.WeightedHeatmap* object that can be added to a *gmaps.Figure* to draw a heatmap. A heatmap shows the density of points in or near a particular area.

To set the parameters, pass them to the constructor or set them on the `Heatmap` object after construction:

```
>>> heatmap = gmaps.heatmap_layer(locations, max_intensity=10)
```

or:

```
>>> heatmap = gmaps.heatmap_layer(locations)
>>> heatmap.max_intensity = 10
```

Examples

```
>>> fig = gmaps.figure()
>>> locations = [(46.1, 5.2), (46.2, 5.3), (46.3, 5.4)]
>>> heatmap = gmaps.heatmap_layer(locations)
>>> heatmap.max_intensity = 2
>>> heatmap.point_radius = 3
>>> heatmap.gradient = ['white', 'gray']
>>> fig.add_layer(heatmap)
```

Parameters

- **locations** (*iterable of latitude, longitude pairs*) – Iterable of (latitude, longitude) pairs denoting a single point. Latitudes are expressed as a float between -90 (corresponding to 90 degrees south) and +90 (corresponding to 90 degrees north). Longitudes are expressed as a float between -180 (corresponding to 180 degrees west) and +180 (corresponding to 180 degrees east). This can be passed in as either a list of tuples, a two-dimensional numpy array or a pandas dataframe with two columns, in which case the first one is taken to be the latitude and the second one is taken to be the longitude.
- **weights** (*iterable of floats, optional*) – Iterable of weights of the same length as *locations*. All the weights must be positive.
- **max_intensity** (*float, optional*) – Strictly positive floating point number indicating the numeric value that corresponds to the hottest colour in the heatmap gradient. Any density of points greater than that value will just get mapped to the hottest colour. Setting this value can be useful when your data is sharply peaked. It is also useful if you find that your heatmap disappears as you zoom in.
- **point_radius** (*int, optional*) – Number of pixels for each point passed in the data. This determines the "radius of influence" of each data point.
- **dissipating** (*bool, optional*) – Whether the radius of influence of each point changes as you zoom in or out. If *dissipating* is True, the radius of influence of each point increases as you zoom out and decreases as you zoom in. If False, the radius of influence remains the same. Defaults to True.
- **opacity** (*float, optional*) – The opacity of the heatmap layer. Defaults to 0.6.

- **gradient** (*list of colors, optional*) – The color gradient for the heatmap. This must be specified as a list of colors. Google Maps then interpolates linearly between those colors. Colors can be specified as a simple string, e.g. 'blue', as an RGB tuple, e.g. (100, 0, 0), or as an RGBA tuple, e.g. (100, 0, 0, 0.5).

Returns A *gmaps.Heatmap* or a *gmaps.WeightedHeatmap* widget.

gmaps.**symbol_layer**(*locations*, *hover_text=''*, *fill_color=None*, *fill_opacity=1.0*, *stroke_color=None*, *stroke_opacity=1.0*, *scale=3*, *info_box_content=None*, *display_info_box=None*)

Symbol layer

Add this layer to a *gmaps.Figure* instance to draw symbols on the map. A symbol will be drawn on the map for each point in the `locations` argument.

Examples

```
>>> fig = gmaps.figure()
>>> locations = [
        (-34.0, -59.166672),
        (-32.23333, -64.433327),
        (40.166672, 44.133331),
        (51.216671, 5.0833302),
        (51.333328, 4.25)
    ]
>>> symbols = gmaps.symbol_layer(
        locations, fill_color="red", stroke_color="red")
>>> fig.add_layer(symbols)
```

You can set a list of information boxes, which will be displayed when the user clicks on a marker.

```
>>> list_of_infoboxes = [
        "Simple string info box",
        "<a href='http://example.com'>HTML content</a>"
    ]
>>> symbol_layer = gmaps.symbol_layer(
            locations, info_box_content=list_of_infoboxes)
```

You can also set text that appears when someone's mouse hovers over a point:

```
>>> names = ["Atucha", "Embalse", "Armenia", "BR", "Doel"]
>>> symbol_layer = gmaps.symbol_layer(locations, hover_text=names)
```

Apart from `locations`, which must be an iterable of (latitude, longitude) pairs, the arguments can be given as either a list of the same length as `locations`, or a single value. If given as a single value, this value will be broadcast to every marker. Thus, these two calls are equivalent:

```
>>> symbols = gmaps.symbol_layer(
        locations, fill_color=["red"]*len(locations))
>>> symbols = gmaps.symbol_layer(
        locations, fill_color="red")
```

The former is useful for passing different colours to different symbols.

```
>>> colors = ["red", "green", "blue", "black", "white"]
>>> symbols = gmaps.symbol_layer(
        locations, fill_color=colors, stroke_color=colors)
```

Parameters

- **locations** (*list of tuples*) – List of (latitude, longitude) pairs denoting a single point. Latitudes are expressed as a float between -90 (corresponding to 90 degrees south) and +90 (corresponding to 90 degrees north). Longitudes are expressed as a float between -180 (corresponding to 180 degrees west) and +180 (corresponding to 180 degrees east).
- **hover_text** (*string or list of strings, optional*) – Text to be displayed when a user's mouse is hovering over a marker. This can be either a single string, in which case it will be applied to every marker, or a list of strings, in which case it must be of the same length as *locations*. If this is set to an empty string, nothing will appear when the user's mouse hovers over a symbol.
- **fill_color** (*single color or list of colors, optional*) – The fill color of the symbol. This can be specified as a single color, in which case the same color will apply to every symbol, or as a list of colors, in which case it must be the same length as `locations`. Colors can be specified as a simple string, e.g. 'blue', as an RGB tuple, e.g. (100, 0, 0), or as an RGBA tuple, e.g. (100, 0, 0, 0.5).
- **fill_opacity** (*float or list of floats, optional*) – The opacity of the fill color. The opacity should be a float between 0.0 (transparent) and 1.0 (opaque), or a list of floats. 1.0 by default.
- **stroke_color** (*single color or list of colors, optional*) – The stroke color of the symbol. This can be specified as a single color, in which case the same color will apply to every symbol, or as a list of colors, in which case it must be the same length as `locations`. Colors can be specified as a simple string, e.g. 'blue', as an RGB tuple, e.g. (100, 0, 0), or as an RGBA tuple, e.g. (100, 0, 0, 0.5).
- **stroke_opacity** (*float or list of floats, optional*) – The opacity of the stroke color. The opacity should be a float between 0.0 (transparent) and 1.0 (opaque), or a list of floats. 1.0 by default.
- **scale** (*integer or list of integers, optional*) – How large the marker is. This can either be a single integer, in which case the same scale will be applied to every marker, or it must be an iterable of the same length as `locations`. The scale must be greater than 1. This defaults to 3.
- **info_box_content** (*string or list of strings, optional*) – Content to be displayed when user clicks on a marker. This should either be a single string, in which case the same content will apply to every marker, or a list of strings of the same length of the *locations* list.
- **display_info_box** (*boolean or list of booleans, optional*) – Whether to display an info box when the user clicks on a symbol. This should either be a single boolean value, in which case it will be applied to every symbol, or a list of boolean values of the same length as the *locations* list. The default value is True for any symbols for which *info_box_content* is set, and False otherwise.

`gmaps.`**marker_layer**(*locations*, *hover_text=''*, *label=''*, *info_box_content=None*, *display_info_box=None*)

Marker layer

Add this layer to a *gmaps.Figure* instance to draw markers corresponding to specific locations on the map. A marker will be drawn on the map for each point in the `locations` argument.

Examples

```
>>> fig = gmaps.figure()
>>> locations = [
        (-34.0, -59.166672),
```

```
        (-32.23333, -64.433327),
        (40.166672, 44.133331),
        (51.216671, 5.0833302),
        (51.333328, 4.25)
    ]
>>> markers = gmaps.marker_layer(locations)
>>> fig.add_layer(markers)
```

Parameters

- **locations** (*list of tuples*) – List of (latitude, longitude) pairs denoting a single point. Latitudes are expressed as a float between -90 (corresponding to 90 degrees south) and +90 (corresponding to 90 degrees north). Longitudes are expressed as a float between -180 (corresponding to 180 degrees west) and +180 (corresponding to 180 degrees east).
- **hover_text** (*string or list of strings, optional*) – Text to be displayed when a user's mouse is hovering over a marker. This can be either a single string, in which case it will be applied to every marker, or a list of strings, in which case it must be of the same length as *locations*. If this is set to an empty string, nothing will appear when the user's mouse hovers over a marker.
- **label** (*string or list of strings, optional*) – Text to be displayed inside the marker. Google maps only displays the first letter of whatever string is passed to the marker. This can be either a single string, in which case every marker will receive the same label, or a list of strings, in which case it must be of the same length as *locations*.
- **info_box_content** (*string or list of strings, optional*) – Content to be displayed when user clicks on a marker. This should either be a single string, in which case the same content will apply to every marker, or a list of strings of the same length of the *locations* list.
- **display_info_box** (*boolean or list of booleans, optional*) – Whether to display an info box when the user clicks on a marker. This should either be a single boolean value, in which case it will be applied to every marker, or a list of boolean values of the same length as the *locations* list. The default value is True for any markers for which *info_box_content* is set, and False otherwise.

gmaps.**geojson_layer**(*geojson*, *fill_color=None*, *fill_opacity=0.4*, *stroke_color=None*, *stroke_opacity=0.8*, *stroke_weight=1.0*)

GeoJSON layer

Add this layer to a *gmaps.Figure* instance to render GeoJSON.

Examples

Let's start by fetching some GeoJSON. We could have loaded it from file, but let's load it from a URL instead. You will need *requests*.

```
>>> import json
>>> import requests
>>> countries_string = requests.get(
    "https://raw.githubusercontent.com/johan/world.geo.json/master/countries.geo.
↪json"
).content
>>> countries = json.loads(countries_string)
```

```
>>> import gmaps
>>> gmaps.configure(api_key="AI...")
```

```
>>> fig = gmaps.figure()
>>> geojson = gmaps.geojson_layer(countries)
>>> fig.add_layer(geojson)
>>> fig
```

We can pass style options into the layer. Let's assign a random color to each country:

```
>>> import random
>>> colors = [
    random.choice(['red', 'green', 'blue', 'purple', 'yellow', 'teal'])
    for country in countries['features']
]
>>> geojson = gmaps.geojson_layer(countries, fill_color=colors)
```

Finally, let's also make our colors more transparent and decrease the stroke weight.

```
>>> geojson = gmaps.geojson_layer(
        countries, fill_color=colors, fill_opacity=0.2, stroke_weight=1)
```

Parameters

- **geojson** (*dict*) – A Python dictionary containing a GeoJSON feature collection. If you have a GeoJSON file, you will need to load it using json.load.
- **fill_color** (*single color or list of colors, optional*) – The fill color of the symbol. This can be specified as a single color, in which case the same color will apply to every symbol, or as a list of colors, in which case it must be the same length as `locations`. Colors can be specified as a simple string, e.g. 'blue', as an RGB tuple, e.g. (100, 0, 0), or as an RGBA tuple, e.g. (100, 0, 0, 0.5).
- **fill_opacity** (*float or list of floats, optional*) – The opacity of the fill color. The opacity should be a float between 0.0 (transparent) and 1.0 (opaque), or a list of floats. 0.4 by default.
- **stroke_color** (*single color or list of colors, optional*) – The stroke color of the symbol. This can be specified as a single color, in which case the same color will apply to every symbol, or as a list of colors, in which case it must be the same length as `locations`. Colors can be specified as a simple string, e.g. 'blue', as an RGB tuple, e.g. (100, 0, 0), or as an RGBA tuple, e.g. (100, 0, 0, 0.5).
- **stroke_opacity** (*float or list of floats, optional*) – The opacity of the stroke color. The opacity should be a float between 0.0 (transparent) and 1.0 (opaque), or a list of floats. 0.8 by default.
- **stroke_weight** (*float or list of floats, optional*) – The width, in pixels, of the stroke. Useful values range from 0.0 (corresponding to no stroke) to about 20, corresponding to a very fat brush. 3.0 by default.

gmaps.**directions_layer**(*start*, *end*, *waypoints=None*, *avoid_ferries=False*, *travel_mode='DRIVING'*, *avoid_highways=False*, *avoid_tolls=False*, *optimize_waypoints=False*)

Create a directions layer.

Add this layer to a *gmaps.Figure* instance to draw directions on the map.

Examples

```
>>> fig = gmaps.figure()
>>> start = (46.2, 6.1)
>>> end = (47.4, 8.5)
```

```
>>> directions = gmaps.directions_layer(start, end)
>>> fig.add_layer(directions)
>>> fig
```

You can also add waypoints on the route:

```
>>> waypoints = [(46.4, 6.9), (46.9, 8.0)]
>>> directions = gmaps.directions_layer(start, end, waypoints=waypoints)
```

You can choose the travel mode:

```
>>> directions = gmaps.directions_layer(start, end, travel_mode='WALKING')
```

Parameters

- **start** (*2-element tuple*) – (Latitude, longitude) pair denoting the start of the journey.
- **end** (*2-element tuple*) – (Latitude, longitude) pair denoting the end of the journey.
- **waypoints** (*List of 2-element tuples, optional*) – Iterable of (latitude, longitude) pair denoting waypoints. Google maps imposes a limitation on the total number of waypoints. This limit is currently 23. You cannot use waypoints when the travel_mode is `'TRANSIT'`.
- **travel_mode** (*str, optional*) – Choose the mode of transport. One of `'BICYCLING'`, `'DRIVING'`, `'WALKING'` or `'TRANSIT'`. A travel mode of `'TRANSIT'` indicates public transportation. Defaults to `'DRIVING'`.
- **avoid_ferries** (*bool, optional*) – Avoid ferries where possible.
- **avoid_highways** (*bool, optional*) – Avoid highways where possible.
- **avoid_tolls** (*bool, optional*) – Avoid toll roads where possible.
- **optimize_waypoints** (*bool, optional*) – If set to true, will attempt to re-order the supplied intermediate waypoints to minimize overall cost of the route.

gmaps.**bicycling_layer**()

Bicycling layer.

Adds cycle routes and decreases the weight of main routes on the map.

Returns A *gmaps.Bicycling* widget.

Examples

```
>>> fig = gmaps.figure()
>>> fig.add_layer(gmaps.bicycling_layer())
```

gmaps.**transit_layer**()

Transit layer.

Adds information about public transport lines to the map. This only affects region for which Google has public transport information.

Returns A *gmaps.Transit* widget.

Examples

```
# map centered on London
>>> fig = gmaps.figure(center=(51.5, -0.2), zoom_level=11)
>>> fig.add_layer(gmaps.transit_layer())
>>> fig
```

Utility functions

gmaps.**configure**(*api_key=None*)
Configure access to the GoogleMaps API.

Parameters **api_key** – String denoting the key to use when accessing Google maps, or None to not pass an API key.

gmaps.locations.**locations_to_list**(*locations*)
Convert from a generic iterable of locations to a list of tuples

Layer widgets only accepts lists of tuples, but we want the user to be able to pass in any reasonable iterable. We therefore need to convert the iterable passed in.

Low level widgets

class gmaps.**Figure**(***kwargs*)
Figure widget

This is the base widget for a Figure. Prefer instantiating instances of `Figure` using the *gmaps.figure()* factory method.

add_layer(*layer*)
Add a data layer to this figure.

Parameters **layer** – a *gmaps* layer.

Examples

```
>>> f = figure()
>>> fig.add_layer(gmaps.heatmap_layer(locations))
```

See also:

layer creation functions

gmaps.heatmap_layer() Create a heatmap layer

gmaps.symbol_layer() Create a layer of symbols

gmaps.marker_layer() Create a layer of markers

gmaps.geojson_layer() Create a GeoJSON layer

gmaps.directions_layer() Create a layer with directions

gmaps.bicycling_layer() Create a layer showing cycle routes

gmaps.transit_layer() Create a layer showing public transport

class gmaps.**Map**(***kwargs*)
Base map class

Instances of this act as a base map on which you can add additional layers.

You should use the *gmaps.figure()* factory method to instiate a figure, rather than building this class directly.

Parameters **initial_viewport** – Define the initial zoom level and map centre. You should construct this using one of the static methods on *gmaps.InitialViewport*. By default, the map is centered on the data.

Examples

```
>>> m = gmaps.figure()
>>> m.add_layer(gmaps.heatmap_layer(locations))
```

To explicitly set the initial map zoom and center:

```
>>> zoom_level = 8
>>> center = (20.0, -10.0)
>>> viewport = InitialViewport.from_zoom_center(zoom_level, center)
>>> m = gmaps.figure(initial_viewport=viewport)
```

class gmaps.**InitialViewport**(***metadata*)

Traitlet defining the initial viewport for a map.

static **from_data_bounds**()

Create a viewport centered on the map's data.

Most of the time, you should rely on the defaults provided by the *gmaps.figure()* factory method, rather than creating a viewport yourself.

Examples

```
>>> m = gmaps.Map(initial_viewport=InitialViewport.from_data_bounds())
```

static **from_zoom_center**(*zoom_level*, *center*)

Create a viewport by explicitly setting the zoom and center

Most of the time, you should rely on the defaults provided by the *gmaps.figure()* factory method, rather than creating a viewport yourself.

Parameters

- **zoom_level** (*int*) – The zoom level for the map. A value between 0 (zoomed out) and 21 (zoomed in). Note that the highest zoom levels are only available in some regions of the world (e.g. cities).
- **center** (*tuple of floats*) – (Latitude, longitude) pair denoting the map center.

Examples

```
>>> zoom_level = 8
>>> center = (20.0, -10.0)
>>> viewport = InitialViewport.from_zoom_center(zoom_level, center)
>>> m = gmaps.figure(initial_viewport=viewport)
```

class gmaps.**Heatmap**(***kwargs*)

Heatmap layer.

Add this to a Map instance to draw a heatmap. A heatmap shows the density of points in or near a particular area.

You should not instantiate this directly. Instead, use the *gmaps.heatmap_layer()* factory function.

Examples

```
>>> fig = gmaps.figure()
>>> locations = [(46.1, 5.2), (46.2, 5.3), (46.3, 5.4)]
>>> heatmap = gmaps.heatmap_layer(locations)
>>> heatmap.max_intensity = 2
>>> heatmap.point_radius = 3
>>> heatmap.gradient = ['white', 'gray']
>>> fig.add_layer(heatmap_layer)
```

Parameters

- **data** (*list of tuples*) – List of (latitude, longitude) pairs denoting a single point. Latitudes are expressed as a float between -90 (corresponding to 90 degrees south) and +90 (corresponding to 90 degrees north). Longitudes are expressed as a float between -180 (corresponding to 180 degrees west) and 180 (corresponding to 180 degrees east).
- **max_intensity** (*float, optional*) – Strictly positive floating point number indicating the numeric value that corresponds to the hottest colour in the heatmap gradient. Any density of points greater than that value will just get mapped to the hottest colour. Setting this value can be useful when your data is sharply peaked. It is also useful if you find that your heatmap disappears as you zoom in.
- **point_radius** (*int, optional*) – Number of pixels for each point passed in the data. This determines the "radius of influence" of each data point.
- **dissipating** (*bool, optional*) – Whether the radius of influence of each point changes as you zoom in or out. If *dissipating* is True, the radius of influence of each point increases as you zoom out and decreases as you zoom in. If False, the radius of influence remains the same. Defaults to True.
- **opacity** (*float, optional*) – The opacity of the heatmap layer. Defaults to 0.6.
- **gradient** (*list of colors, optional*) – The color gradient for the heatmap. This must be specified as a list of colors. Google Maps then interpolates linearly between those colors. Colors can be specified as a simple string, e.g. 'blue', as an RGB tuple, e.g. (100, 0, 0), or as an RGBA tuple, e.g. (100, 0, 0, 0.5).

class gmaps.**WeightedHeatmap**(***kwargs*)

Heatmap with weighted points.

Add this layer to a `Map` instance to draw a heatmap. Unlike the plain Heatmap layer, which assumes that all points should have equal weight, this layer lets you specifiy different weights for points.

You should not instantiate this directly. Instead, use the *gmaps.heatmap_layer()* factory function, passing in a parameter for *weights*.

Examples

```
>>> fig = gmaps.figure()
>>> locations = [(46.1, 5.2), (46.2, 5.3), (46.3, 5.4)]
>>> weights = [0.5, 0.2, 0.8]
>>> heatmap = gmaps.heatmap_layer(locations, weights=weights)
>>> heatmap.max_intensity = 2
>>> fig.add_layer(heatmap_layer)
```

Parameters

- **data** (*list of tuples*) – List of (latitude, longitude, weight) triples for a single point. Latitudes are expressed as a float between -90 (corresponding to 90 degrees south) and +90 (corresponding to 90 degrees north). Longitudes are expressed as a float between -180

(corresponding to 180 degrees west) and +180 (corresponding to 180 degrees east). Weights must be non-negative.

- **max_intensity** (*float, optional*) – Strictly positive floating point number indicating the numeric value that corresponds to the hottest colour in the heatmap gradient. Any density of points greater than that value will just get mapped to the hottest colour. Setting this value can be useful when your data is sharply peaked. It is also useful if you find that your heatmap disappears as you zoom in.
- **point_radius** (*int, optional*) – Number of pixels for each point passed in the data. This determines the "radius of influence" of each data point.
- **dissipating** (*bool, optional*) – Whether the radius of influence of each point changes as you zoom in or out. If *dissipating* is True, the radius of influence of each point increases as you zoom out and decreases as you zoom in. If False, the radius of influence remains the same. Defaults to True.
- **opacity** (*float, optional*) – The opacity of the heatmap layer. Defaults to 0.6.
- **gradient** (*list of colors, optional*) – The color gradient for the heatmap. This must be specified as a list of colors. Google Maps then interpolates linearly between those colors. Colors can be specified as a simple string, e.g. 'blue', as an RGB tuple, e.g. (100, 0, 0), or as an RGBA tuple, e.g. (100, 0, 0, 0.5).

class gmaps.**Symbol**(***kwargs*)
Class representing a single symbol.

Symbols are like markers, but the point is represented by an SVG symbol, rather than the default inverted droplet. Symbols should be added to the map via the 'Symbols' widget.

class gmaps.**Marker**(***kwargs*)
Class representing a marker.

Markers should be added to the map via the 'Markers' widget.

class gmaps.**Markers**(***kwargs*)
A collection of markers or symbols.

class gmaps.**GeoJsonFeature**(***kwargs*)
Widget for a single GeoJSON feature.

Prefer to use the *geojson_layer* function to construct these, rather than making them explicitly.

class gmaps.**GeoJson**(***kwargs*)
Widget for a collection of GeoJSON features.

Prefer to use the *geojson_layer* function to construct this, rather than making them explicitly.

Use the *features* attribute on this class to change the style of the features in this layer.

class gmaps.**Directions**(***kwargs*)
Directions layer.

Add this to a *gmaps.Figure* instance to draw directions.

Use the *gmaps.directions_layer()* factory function to instantiate this class, rather than the constructor.

Examples

```
>>> fig = gmaps.figure()
>>> start = (46.2, 6.1)
>>> end = (47.4, 8.5)
>>> waypoints = [(52.37403, 4.88969)]
```

```
>>> directions_layer = gmaps.directions_layer(start, end, waypoints)
>>> fig.add_layer(directions_layer)
```

There is a limitation in the number of waypoints allowed by Google (currently 23). If it fails to return directions, a `DirectionsServiceException` is raised.

```
>>> directions_layer = gmaps.Directions(data=data*10)
Traceback (most recent call last):
    ...
DirectionsServiceException: No directions returned: MAX WAYPOINTS EXCEEDED
```

Parameters

- **data** (*list of tuples of length >= 2*) – List of (latitude, longitude) pairs denoting a single point. The first pair denotes the starting point and the last pair denote the end of the route. Latitudes are expressed as a float between -90 (corresponding to 90 degrees south) and +90 (corresponding to 90 degrees north). Longitudes are expressed as a float between -180 (corresponding to 180 degrees west) and 180 (corresponding to 180 degrees east).
- **travel_mode** (*str, optional*) – Choose the mode of transport. One of `'BICYCLING'`, `'DRIVING'`, `'WALKING'` or `'TRANSIT'`. A travel mode of `'TRANSIT'` indicates public transportation. Defaults to `'DRIVING'`.
- **avoid_ferries** (*bool, optional*) – Avoids ferries where possible.
- **avoid_highways** (*bool, optional*) – Avoids highways where possible.
- **avoid_tolls** (*bool, optional*) – Avoids toll roads where possible.
- **optimize_waypoints** (*bool, optional*) – Attempt to re-order the supplied intermediate waypoints to minimize overall cost of the route.

class gmaps.**Bicycling**(***kwargs*)

Bicycling layer.

Add this to a *gmaps.Map* or *gmaps.Figure* instance to add cycling routes.

You should not instantiate this directly. Instead, use the *gmaps.bicycling_layer()* factory function.

Examples

```
>>> fig = gmaps.figure()
>>> fig.add_layer(gmaps.bicycling_layer())
```

class gmaps.**Transit**(***kwargs*)

Transit layer.

Add this to a *gmaps.Map* or a *gmaps.Figure* instance to add transit (public transport) information. This only affects regions for which Google has transit information.

You should not instantiate this directly. Instead, use the *gmaps.transit_layer()* factory function.

Examples

```
# map centered on London
>>> fig = gmaps.figure(center=(51.5, -0.2), zoom_level=11)
>>> fig.add_layer(gmaps.transit_layer())
>>> fig
```

Datasets

gmaps.datasets.**list_datasets**()
List of datasets available

gmaps.datasets.**dataset_metadata**(*dataset_name*)
Information about the dataset

This returns a dictionary containing a 'description', a list of the dataset headers and optionally information about the dataset source.

Examples

```
>>> dataset_metadata("earthquakes")
{'description': 'Taxi pickup location data in San Francisco',
 'headers': ['latitude', 'longitude']}
```

gmaps.datasets.**load_dataset**(*dataset_name*)
Fetch a dataset, returning an array of tuples.

gmaps.datasets.**load_dataset_as_df**(*dataset_name*)
Fetch a dataset, returning a pandas dataframe.

GeoJSON geometries

gmaps.geojson_geometries.**list_geometries**()
List of GeoJSON geometries available

gmaps.geojson_geometries.**geometry_metadata**(*geometry_name*)
Information about the geometry.

This returns a dictionary containing a 'description'.

Examples

```
>>> geometry_metadata("countries")
{'description': 'Map of world countries'}
```

gmaps.geojson_geometries.**load_geometry**(*geometry_name*)
Fetch a geometry.

Returns A python dictionary containing the geometry.

Examples

```
>>> import gmaps
>>> import gmaps.geojson_geometries
>>> gmaps.configure(api_key="AIza...")
>>> countries_geojson = gmaps.geojson_geometries.load_geometry('countries')
```

```
>>> fig = gmaps.figure()
>>> gini_layer = gmaps.geojson_layer(countries_geojson)
>>> fig.add_layer(gini_layer)
>>> fig
```

Traitlets

class gmaps.geotraitlets.**ColorAlpha**(*default_value=traitlets.Undefined*, *allow_none=False*, ***metadata*)

Trait representing a color that can be passed to Google maps.

This is either a string like 'blue' or '#aabbcc' or an RGB tuple like (100, 0, 250) or an RGBA tuple like (100, 0, 250, 0.5).

validate(*obj*, *value*)

Verifies that 'value' is a string or tuple and converts it to a value like 'rgb(x,y,z)'

class gmaps.geotraitlets.**ColorString**(*default_value=traitlets.Undefined*, *allow_none=False*, *read_only=None*, *help=None*, *config=None*, ***kwargs*)

A string holding a color recognized by Google Maps.

Apparently Google Maps accepts 'all CSS3 colors, including RGBA, [...] except for extended named colors and HSL(A) values'.

Using *this <https://www.w3.org/TR/css3-color/#html4>* page for reference.

default_value = traitlets.Undefined

class gmaps.geotraitlets.**Latitude**(*default_value=traitlets.Undefined*, *allow_none=False*, ***kwargs*)

Float representing a latitude

Latitude values must be between -90 and 90.

default_value = traitlets.Undefined

class gmaps.geotraitlets.**Longitude**(*default_value=traitlets.Undefined*, *allow_none=False*, ***kwargs*)

Float representing a longitude

Longitude values must be between -180 and 180.

default_value = traitlets.Undefined

class gmaps.geotraitlets.**Point**(*default_value*)

Tuple representing a (latitude, longitude) pair.

class gmaps.geotraitlets.**ZoomLevel**(*default_value=traitlets.Undefined*, *allow_none=False*, ***kwargs*)

Integer representing a zoom value allowed by Google Maps

default_value = traitlets.Undefined

CHAPTER 6

Contributing to jupyter-gmaps

Contributing

We want to start by thanking you for using Jupyter-gmaps. We very much appreciate all of the users who catch bugs, contribute enhancements and features or add to the documentation. Every contribution is meaningful, so thank you for participating.

How to contribute

Code contributions are more than welcome. Take a look at the issue tracker, specially issues labelled as *beginner-friendly*. These are issues which have a lot of impact on the project, but don't require understanding the entire codebase.

Beyond code, the following contributions will make *gmaps* a better project:

- additional datasets related to geographical data. The data needs to be clean, of reasonable size (ideally not more than 1MB), and should be clearly related to geography.
- additional GeoJSON geometries. These should be clean and reasonably small (ideally 1-3MB).
- Examples of you using Jupyter-gmaps. If you've used gmaps and have an artefact to show for it (a blogpost or an image), I'm very happy to put a link in the documentation.

Installing a development version of gmaps

See the installation instructions for installing a development version.

Testing

We use nose for unit testing. Run `nosetests` in the root directory of the project to run all the tests, or in a specific directory to just run the tests in that directory.

Guidelines

Workflow

We loosely follow the git workflow used in numpy development. Features should be developed in separate branches and merged into the master branch when complete.

Code

Please follow the PEP8 conventions for formatting and indenting code and for variable names.

How to release jupyter-gmaps

This is a set of instructions for releasing to Pypi. The release process is somewhat automated with an invoke task file. You will need *invoke* installed.

- Run `invoke prerelease <version>`, where `version` is the version number of the release candidate. If you are aiming to release version `0.5.0`, this will be `0.5.0-rc1`. This will automatically bump the version numbers and upload the release to Pypi and NPM. Unfortunately, Pypi does not recognize this as a pre-release, and therefore gives it more precendence than the previous, stable release. To correct this, go to the gmaps page on Pypi, then go to the *releases* tab and manually hide that release and un-hide the previous one.
- Verify that you can install the new version and that it works correctly with `pip install gmaps==<new version>` and `jupyter nbextension enable --py --sys-prefix gmaps`. It's best to verify the installation on a clean virtual machine (rather than just in a new environment) since installation is more complex than for pure Python packages.
- If the manual installation tests failed, fix the issue and repeat the previous steps with `rc2` etc. If installing worked, proceed to the next steps.
- Run `invoke release <version>`, where `version` is the version number of the release (e.g. `0.5.0`). You will be prompted to enter a changelog.
- Verify that the new version is available by running `pip install gmaps` in a new virtual environment.
- Run `invoke postrelease <version>`, where `version` is the version number of the new release. This will commit the changes in version, add an annotated tag from the changelog and push the changes to Github. It will then change the version back to a `-dev` version.
- Run `invoke release_conda <version>` to release the new version to conda-forge.

CHAPTER 7

Release notes

Version 0.5.4

This release:

- Fixes a bug where bounds were incorrectly calculated for the case where there was a single point in the data (PR 160).
- Allows setting the travel mode in the directions layer (PR 157).
- Fixes the release script to use a fork of the conda-forge feedstock (PR 156).

Version 0.5.3

This release adds two minor features:

- The directions layer can be customised, in particular how the route is calculated ([PR 153](https://github.com/pbugnion/gmaps/pull/153))
- The user can explicitly set the map zoom and center ([PR 154](https://github.com/pbugnion/gmaps/pull/154))

It also makes the following non-breaking changes:

- Refactor JS to use ES6 classes.

Version 0.5.3

This release adds two minor features:

- The directions layer can be customised, in particular how the route is calculated ([PR 153](https://github.com/pbugnion/gmaps/pull/153))

- The user can explicitly set the map zoom and center ([PR 154](https://github.com/pbugnion/gmaps/pull/154))

It also makes the following non-breaking changes:

- Refactor JS to use ES6 classes.

Version 0.5.2

This is a bugfix release.

- Bounds are now calculated correctly when there are multiple layers (PR 148).
- Latitude bounds cannot exceed the maximum allowed by Google Maps (PR 149).
- Alpha values of 1.0 are now allowed.

Version 0.5.1

This patch release:

- fixes flakiness downloading images as PNGs (issue 129).
- adds an error box view for errors that come up in the frontend.

It adds improvements to the development workflow:

- License is included in the source to facilicate deployment to conda-forge
- Facilitate installation in dev mode.
- Automation of release process.

Version 0.5.0

This release:

- introduces a new Figure widget that wraps a toolbar and a map
- adds the ability to export maps to PNG
- fixes bugs and outdated dependencies that prevented embedding maps in rendered HTML.

Version 0.4.1

- Add a GeoJSON layer (PRs #106 and #115)
- Add the *geojson_geometries* module for bundling GeoJSON geometries with *jupyter-gmaps* (PR #111).
- Minor improvements to README and compatibility guide.
- Support for Python 3.6 (PR #107).

Version 0.4.0

- **Add factory functions to make creating layers easier. Instead of creating widgets directly, the widgets are instantiated thro**
  - passing arbitrary iterables to the factory function (issue #66)
  - passing more complex sets of options (issue #65)
- The directions interface is now a first class layer (issue #64)
- A regression whereby the API documentation wasn't building on readthedocs is now fixed (PR #105).

Version 0.3.6

- Adds info boxes to the marker and symbol layers (PR #98).

Version 0.3.5

- Bugfix in deprecated heatmap method (PR #89).

Version 0.3.4

- Add marker and symbol layer (PR #78)
- Fix bug involving incorrect latitude bound calculation.

Version 0.3.3

- Improve automatic bounds calculations for heatmaps (PR #84)

Version 0.3.2

- Allow setting heatmap options (issues #74)
- Basic unit tests for traitlets, mixins and datasets
- Continuous integration with Travis CI.

Version 0.3.1

Fix release to allow injecting Google maps API keys. Google maps now mandates API keys, so this release provides a way to pass in a key (issue #61).

This release also includes a fix for having multiple layers on the same map.

Version 0.3.0

Complete re-write of gmaps to work with IPython 4.2 and ipywidgets 5.x. This release is at feature parity with the previous release, but the interface differs:

- Maps are now built up from a base to which we add layers.
- Heatmaps and weighted heatmaps are now layers that can be added to the base map.
- Add the acled_africa dataset to demonstrate heatmaps with a substantial amount of data.
- Now fits into the Jupyter installation convention for widget extensions.
- Add sphinx documentation
- Remove example notebooks (these may be added back in a later release)

Version 0.2.2

- Remove dependency on Numpy
- Fix broken datasets example (issue #52)

Version 0.2.1

test release – no changes.

Version 0.2

- IPython 4.0 compatibility
- Python 3 compatibility
- Drop IPython 2.x compatibility

Version 0.1.6

Fixed typo in setup script.

Version 0.1.5

Weighted heatmaps and datasets

- Added possibility of including weights in heatmap data.
- Added a datasets module to allow new users to play around with data without having to find their own dataset.

Version 0.1.4

Another bugfix release.

- Fixed a bug that arose when using heatmap with default values of some of the parameters.

Version 0.1.3

Bugfix release.

- Fixed a bug that arose when using the heatmap with IPython2.3 in the previous release. The bug was caused by the slightly different traitlets API between the two IPython versions.

Version 0.1.2

Minor heatmap improvements.

- Exposed the 'maxIntensity' and 'radius' options for heatmaps.

Version 0.1.1

Bugfix release.

- Ensures the notebook extensions are actually included in the source distribution.

Version 0.1

Initial release.

- Allows plotting heatmaps from a list / array of pairs of longitude, latitude floats on top of a Google Map.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

g

Index

V

W

Z