gmaps Documentation

Release 0.8.2

Pascal Bugnion

Oct 13, 2018

Contents

CHAPTER 1

Installation

1.1 Installing *jupyter-gmaps* with *conda*

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

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

1.2 Installing *jupyter-gmaps* with *pip*

Make sure that you have enabled *ipywidgets* widgets extensions:

```
$ 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
```

1.3 Installing *jupyter-gmaps* for JupyterLab

To use *jupyter-gmaps* with JupyterLab, you will need to install the jupyter widgets extension for JupyterLab:

```
$ jupyter labextension install @jupyter-widgets/jupyterlab-manager
```

You can then install *jupyter-gmaps* via pip (or conda):

```
$ pip install gmaps
```

Next time you open JupyterLab, you will be prompted to rebuild JupyterLab: this is necessary to include the *jupyter-gmaps* frontend code into your JupyterLab installation. You can also trigger this directly on the command line with:

```
$ jupyter lab build
```

1.4 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 build:nbextension` 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. If you are making many changes to the JavaScript directory, you can run `npm run build:watch` to rebuild on every change.

You should not need to restart the notebook server.

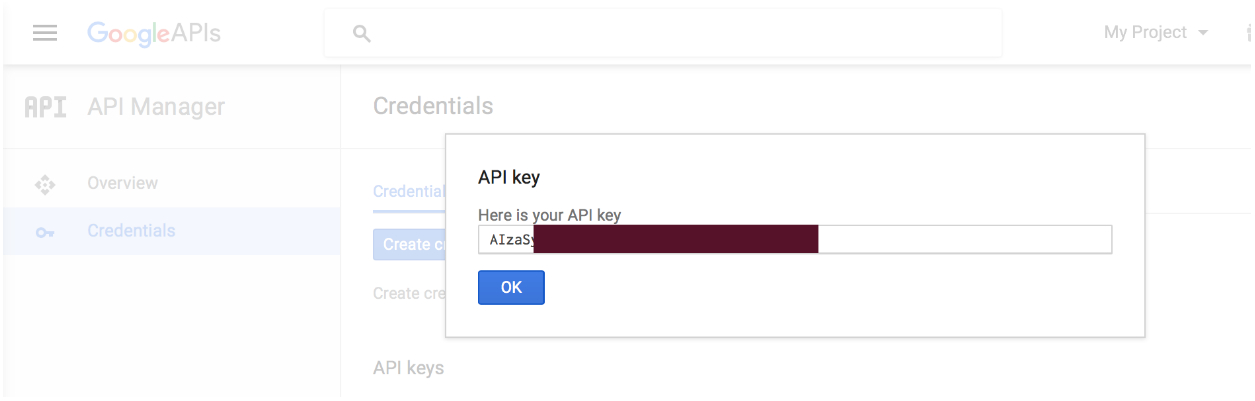
1.5 Source code

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`.



Here is your API key
AIzaSy[redacted]

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
```

(continues on next page)

(continued from previous page)

```
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:

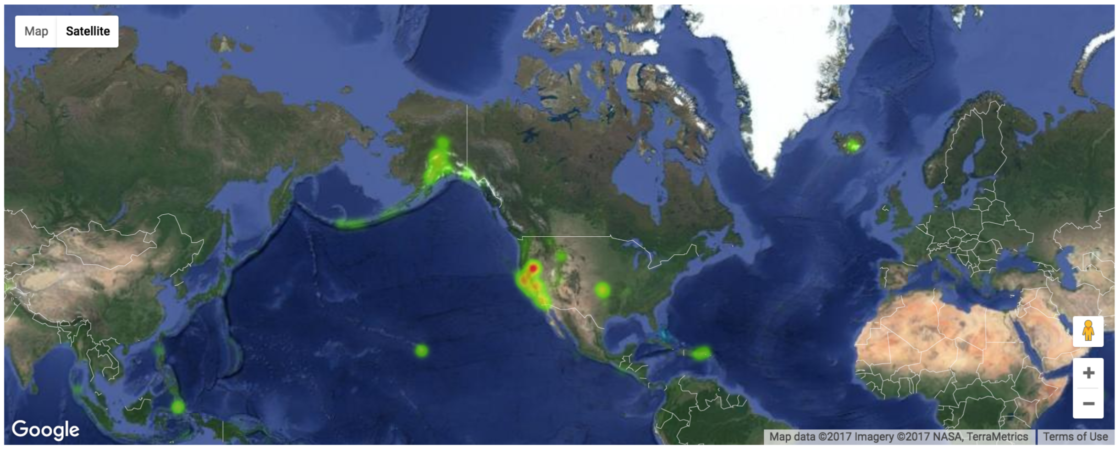
```
locations = earthquake_df[['latitude', 'longitude']]
weights = earthquake_df['magnitude']
fig = gmaps.figure()
fig.add_layer(gmaps.heatmap_layer(locations, weights=weights))
fig
```

```
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.

3.1 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
```

Map
Satellite
Dijon
Besançon
Zürich
Bern
Lucerne
Lausanne
Geneva
Neuchâtel
Biel/Bienne
Fribourg
Thun
Interlaken
Montreux
Sion
Zermatt
Lugano
Locarno
Domodossola
Chalon-sur-Saône
Lons-le-Saunier
Mâcon
Oyonnax
Annemasse
Roanne
Map data ©2017 GeoBasis-DE/BKG (©2009), Google
Terms of Use
Report a map error

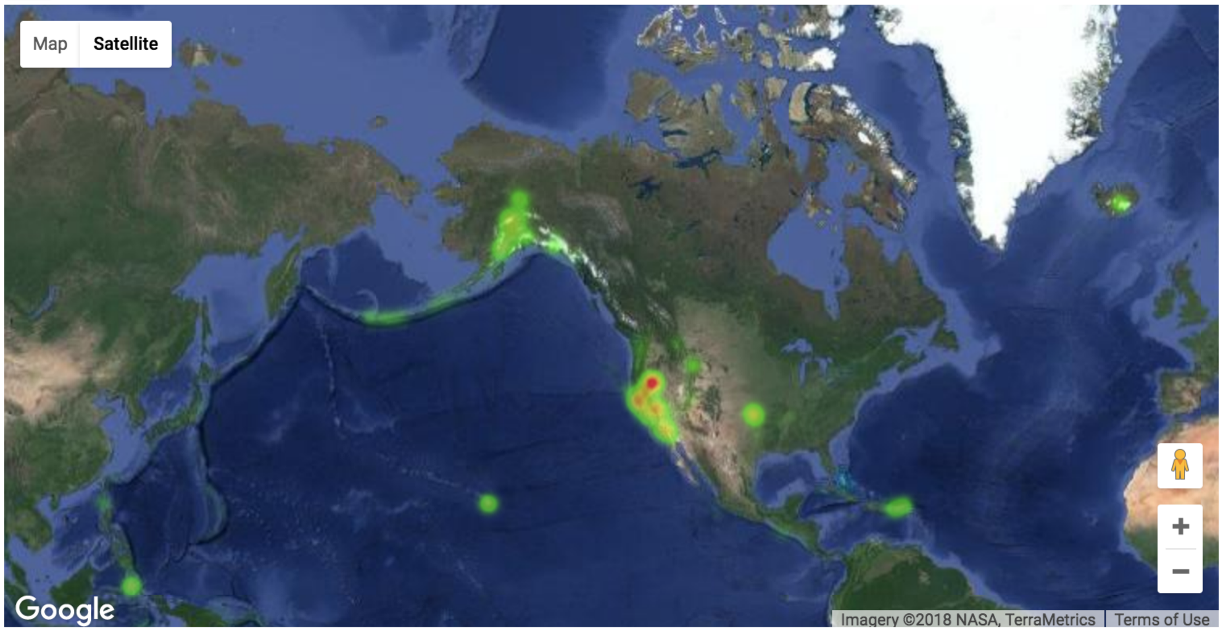
You then add layers on top of the base map. For instance, to add a heatmap layer:

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

fig = gmaps.figure(map_type='SATELLITE')

# 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
```

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

	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

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

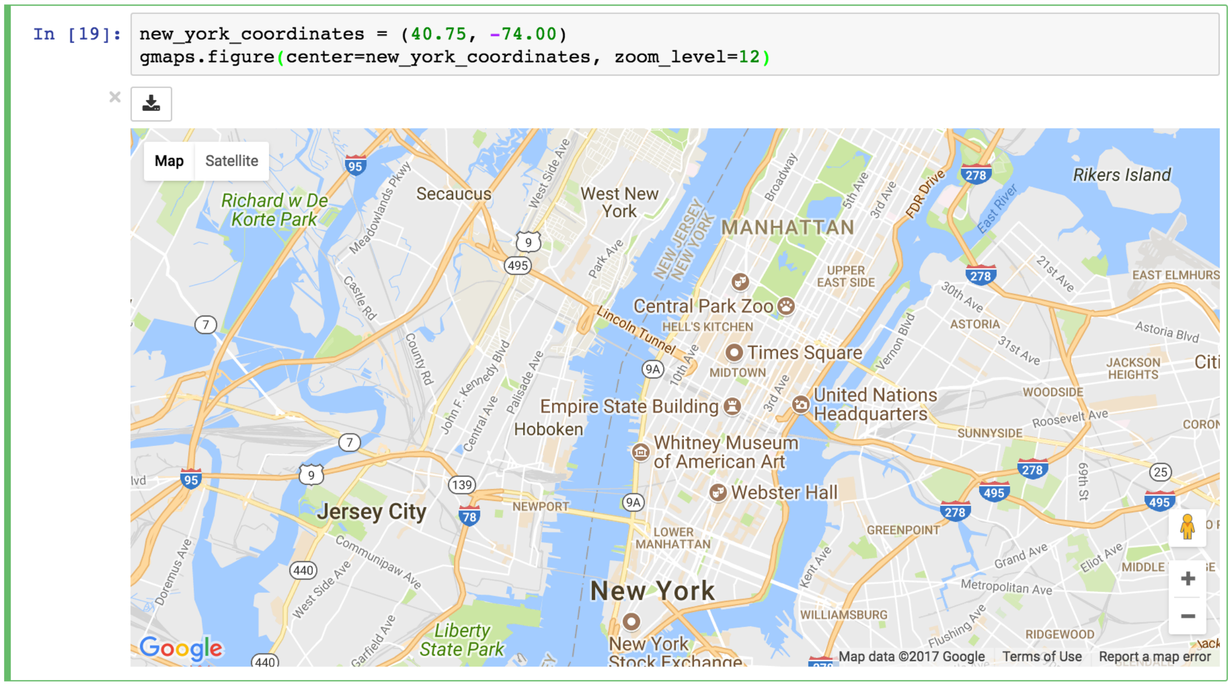
3.2 Base maps

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)
```

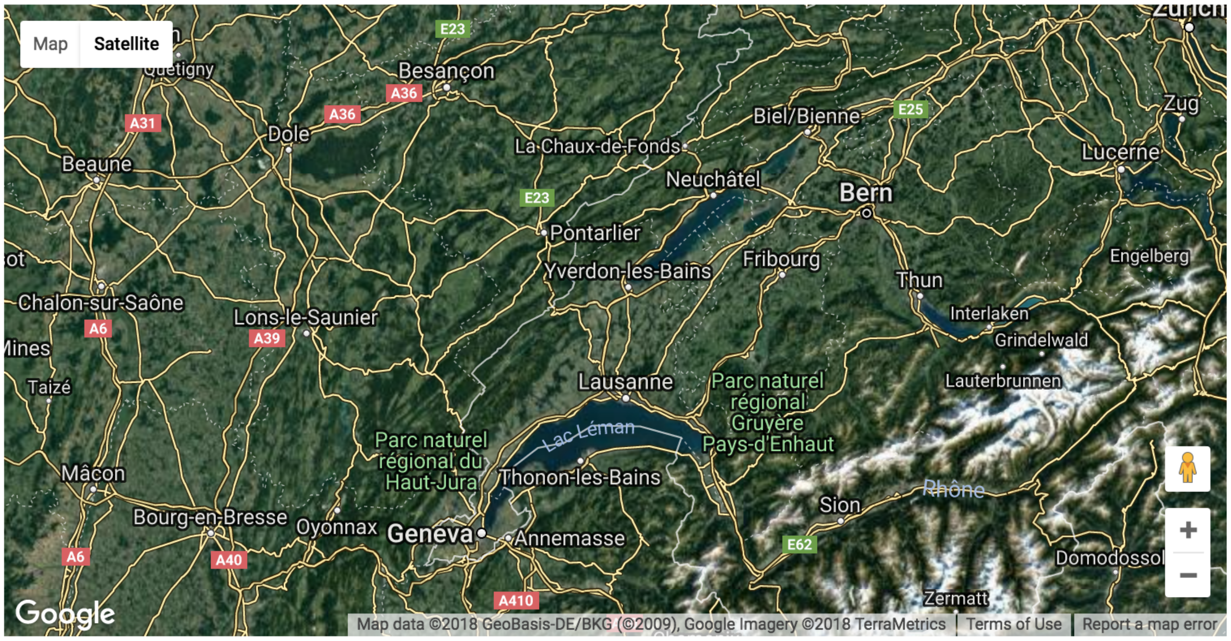


```
In [19]: new_york_coordinates = (40.75, -74.00)
         gmaps.figure(center=new_york_coordinates, zoom_level=12)
```

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.

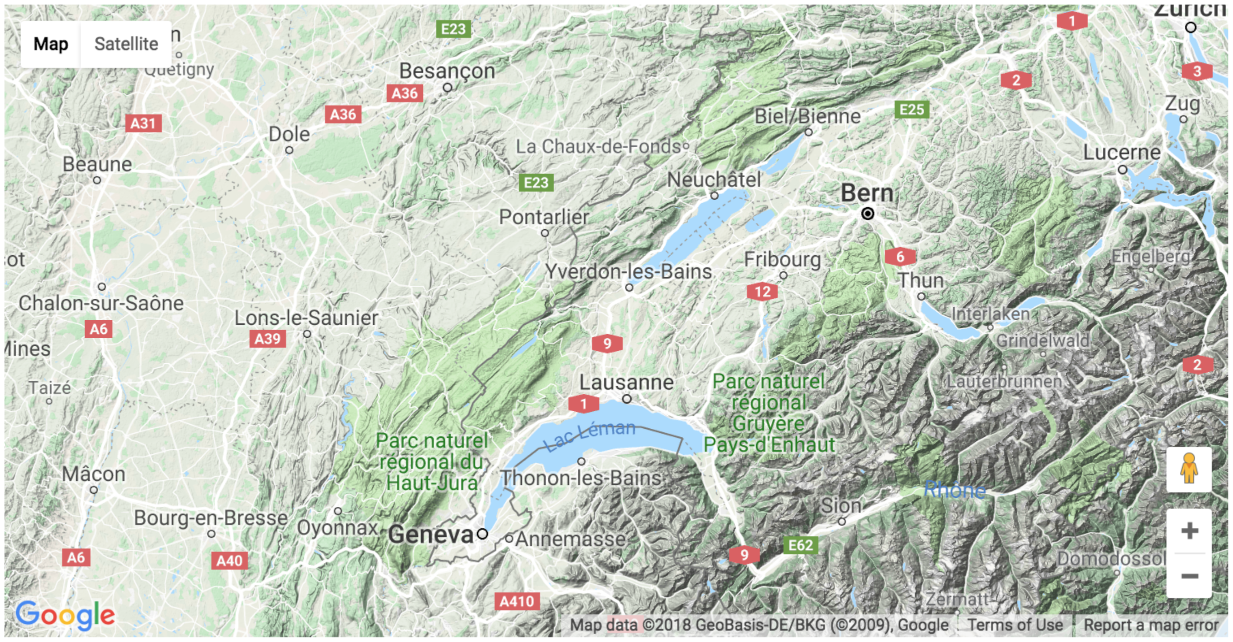
Google maps offers three different base map types. Choose the base map type by setting the `map_type` parameter:

```
gmaps.figure(map_type='HYBRID')
gmaps.figure(map_type='HYBRID')
```



```
gmaps.figure(map_type='TERRAIN')
```

```
gmaps.figure(map_type='TERRAIN')
```



There are four map types available:

- `'ROADMAP'` is the default Google Maps style,

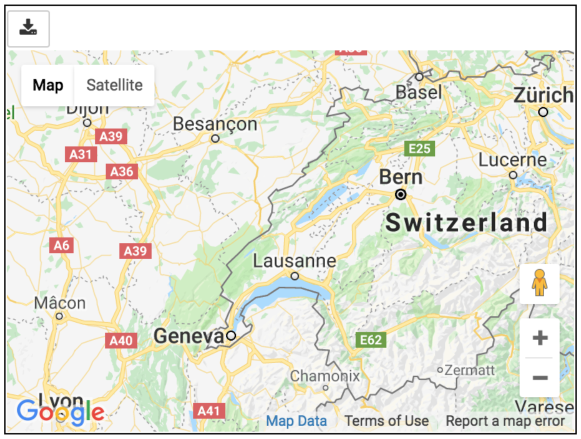
- `'SATELLITE'` is a simple satellite view,
- `'HYBRID'` is a satellite view with common features, such as roads and cities, overlaid,
- `'TERRAIN'` is a map that emphasizes terrain features.

3.3 Customising map width, height and layout

The layout of a map figure is controlled by passing a layout argument. This is a dictionary of properties controlling how the widget is displayed:

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

figure_layout = {
    'width': '400px',
    'height': '400px',
    'border': '1px solid black',
    'padding': '1px'
}
gmaps.figure(layout=figure_layout)
```

```
In [12]: figure_layout = {
             'width': '400px',
             'height': '300px',
             'border': '1px solid black',
             'padding': '1px'
         }
         gmaps.figure(layout=figure_layout)
```



The parameters that you are likely to want to tweak are:

- *width*: controls the figure width. This should be a CSS dimension. For instance, `400px` will create a figure that is 400 pixels wide, while `100%` will create a figure that takes up the output cell's entire width. The default width is `100%`.
- *height*: controls the figure height. This should be a CSS dimension. The default height is `420px`.
- *border*: Place a border around the figure. This should be a valid CSS border.
- *padding*: Gap between the figure and the border. This should be a valid CSS padding. You can either have a single dimension (e.g. `2px`), or a quadruple indicating the padding width for each side (e.g. `1px 2px 1px`

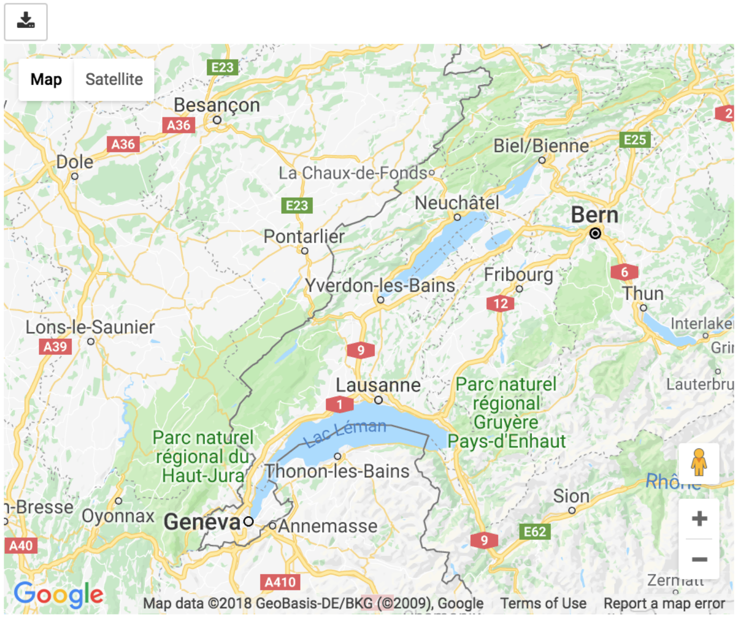
2px). This is 0 by default.

- *margin*: Gap between the border and the figure container. This should be a valid CSS margin. This is 0 by default.

To center a map in an output cell, use a fixed width and set the left and right margins to auto:

```
figure_layout = {'width': '500px', 'margin': '0 auto 0 auto'}
gmaps.figure(layout=figure_layout)
```

```
In [8]: figure_layout = {'width': '500px', 'margin': '0 auto 0 auto'}
        gmaps.figure(layout=figure_layout)
```



3.4 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:

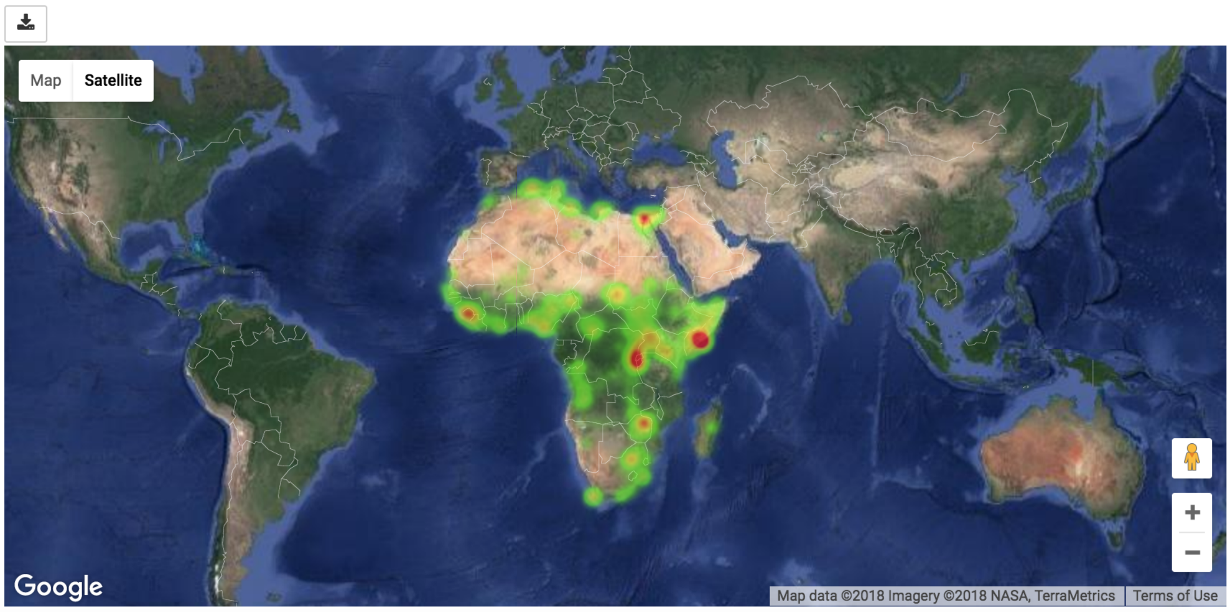
```
import gmaps
import gmaps.datasets
gmaps.configure(api_key='AI...')

locations = gmaps.datasets.load_dataset_as_df('acled_africa')
```

(continues on next page)

(continued from previous page)

```
fig = gmaps.figure(map_type='HYBRID')
heatmap_layer = gmaps.heatmap_layer(locations)
fig.add_layer(heatmap_layer)
fig
```

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



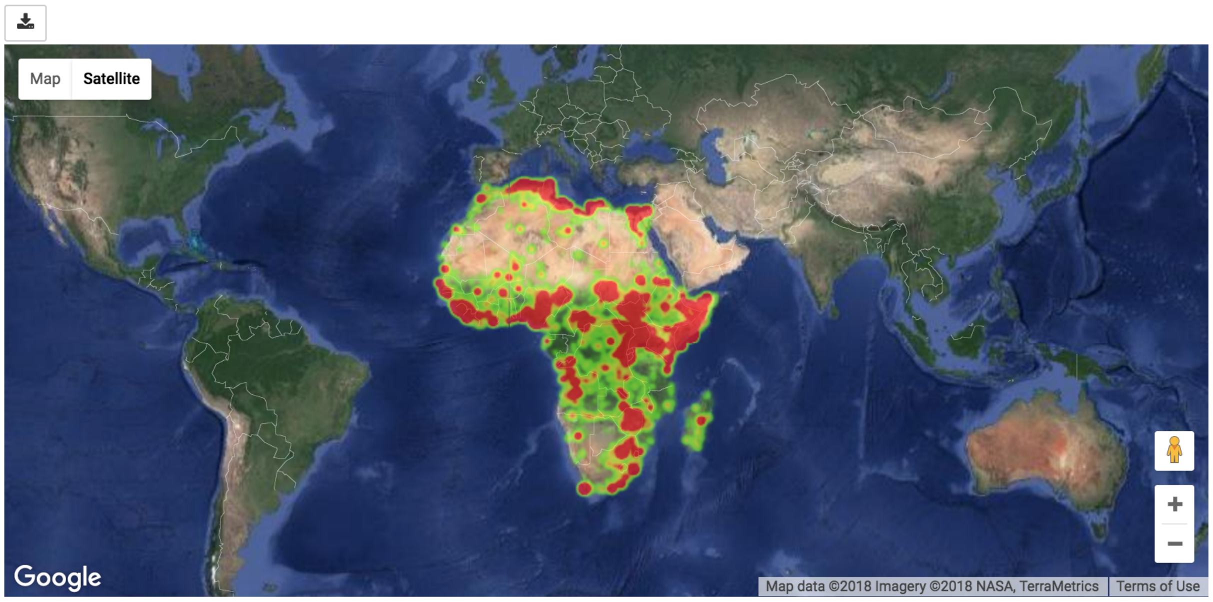
3.4.1 Preventing dissipation on zoom

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:

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



```
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.

3.4.2 Setting the color gradient and opacity

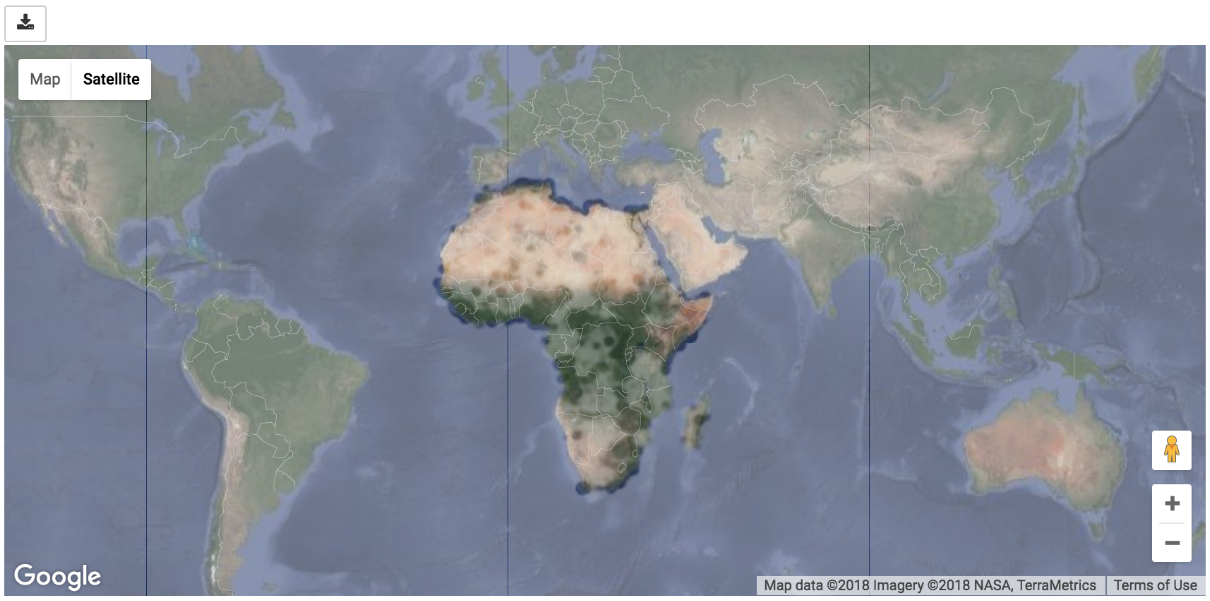
You can set the color gradient of the map by passing in a list of colors. Google maps will interpolate linearly between those colors. You can represent a color as a string denoting the color (the colors allowed by this):

```
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)
]
```

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



```
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
```

3.5 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(
```

(continues on next page)

(continued from previous page)

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

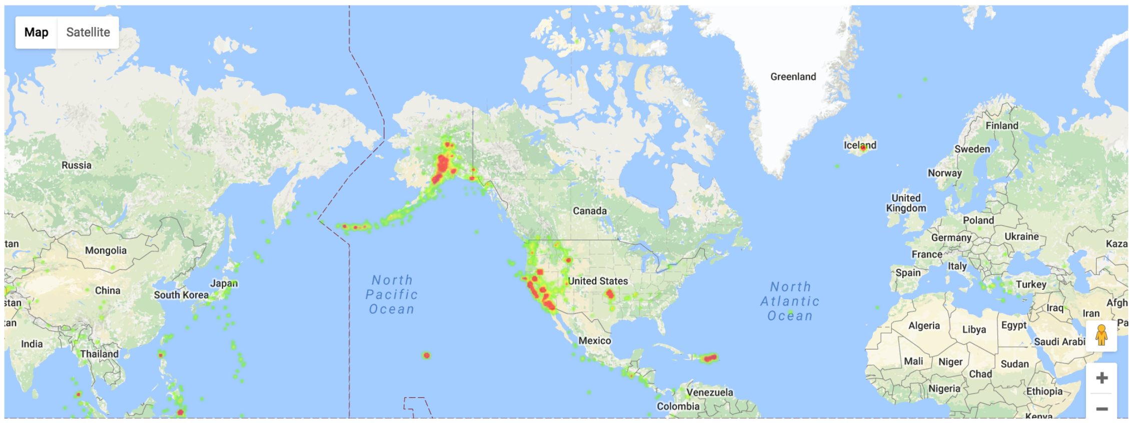
In [24]:
```
import gmaps
import gmaps.datasets
gmaps.configure(api_key="AIz...")
```

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
```



3.6 Markers and symbols

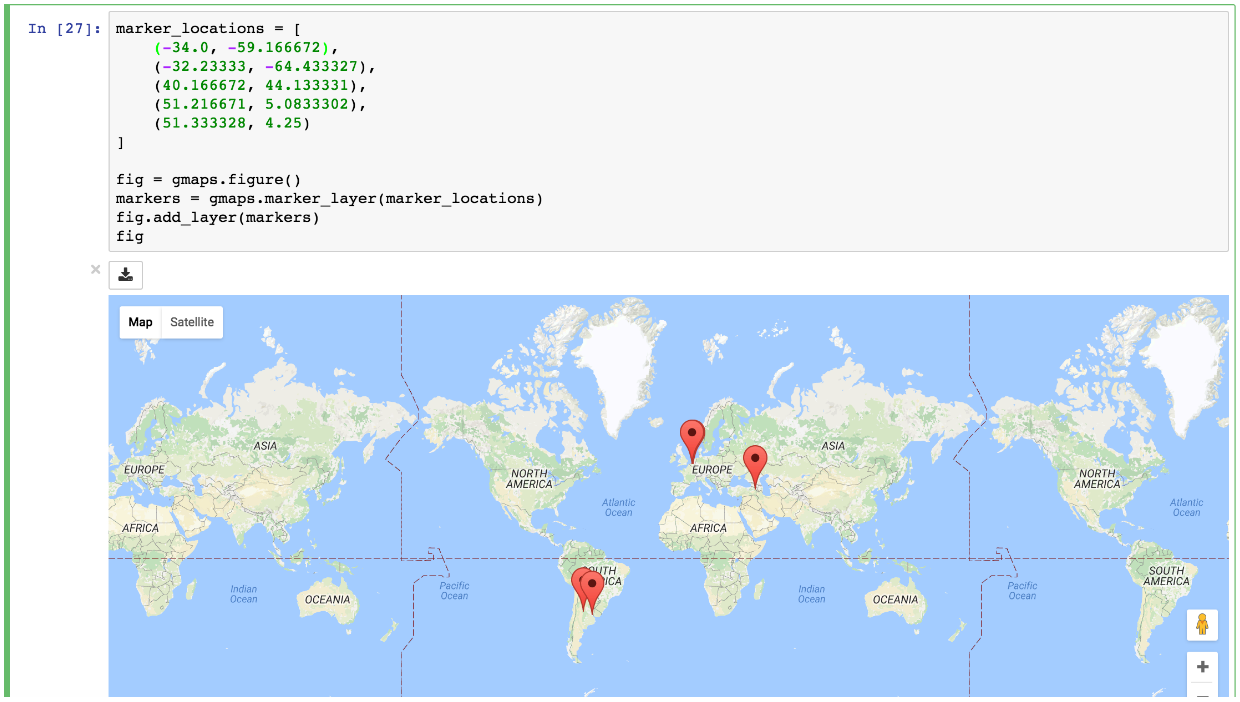
We can add a layer of markers to a Google map. Each marker represents an individual data point:

```
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)
]
```

(continues on next page)

(continued from previous page)

```
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:

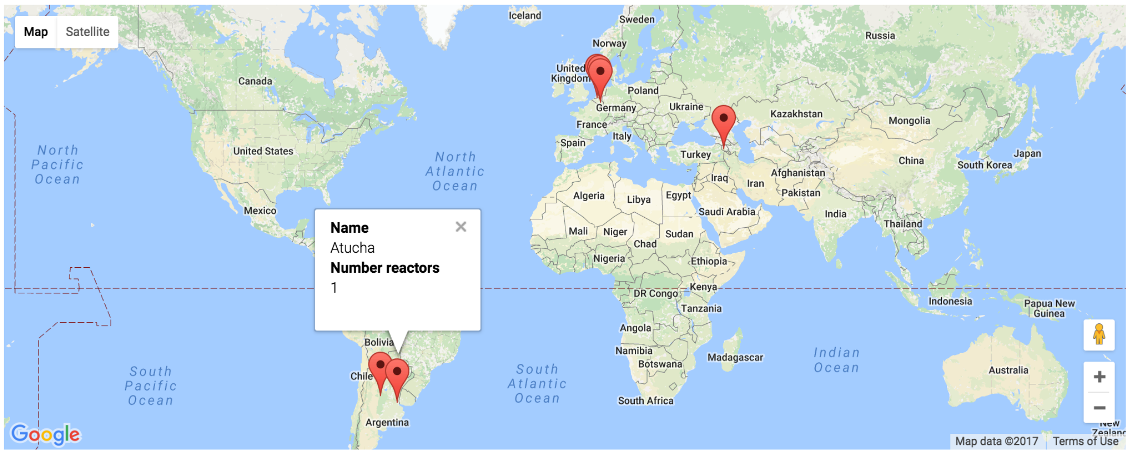
```
import gmaps
gmaps.configure(api_key='AI...')

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)
```

(continues on next page)

(continued from previous page)

```
fig
```

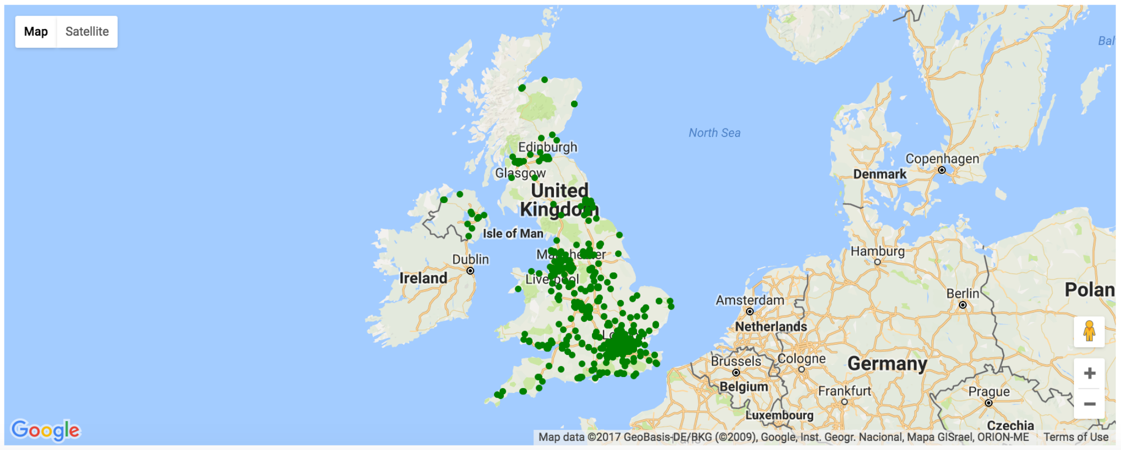
```
In [29]: nuclear_power_plants = [
    {"name": "Atucha", "location": (-34.0, -59.167), "active_reactors": 1},
    {"name": "Embalse", "location": (-32.2333, -64.4333), "active_reactors": 1},
    {"name": "Armenia", "location": (40.167, 44.133), "active_reactors": 1},
    {"name": "Br", "location": (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
```

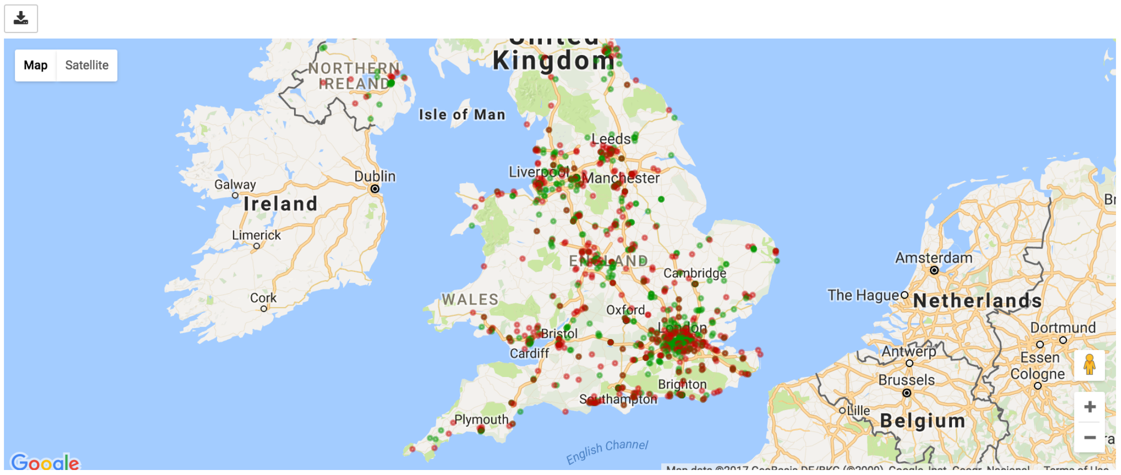
```
In [45]: df = gmaps.datasets.load_dataset_as_df("starbucks_kfc_uk")

         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:

```python
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
```



3.6.1 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.

3.7 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:

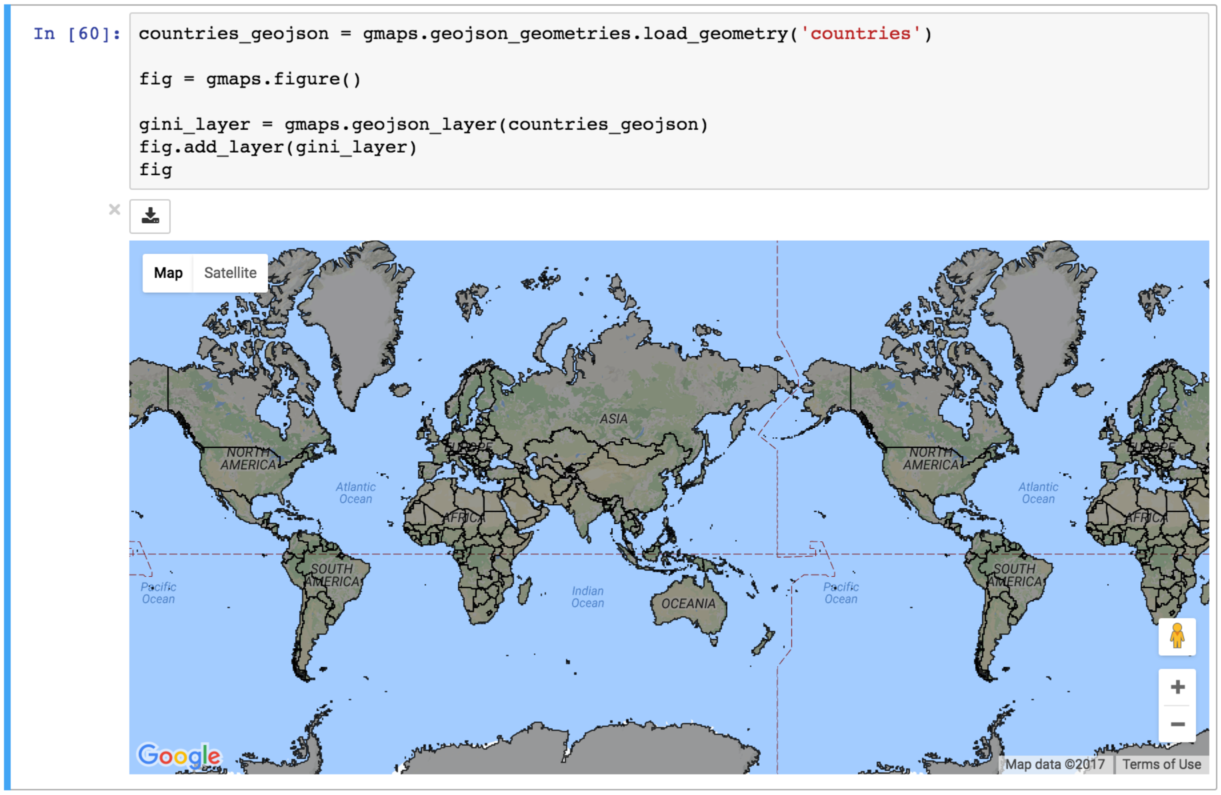
```
import gmaps
import gmaps.geojson_geometries
gmaps.configure(api_key='AIza...')

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

(continues on next page)

(continued from previous page)

```
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
```

(continues on next page)

(continued from previous page)

```
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
```

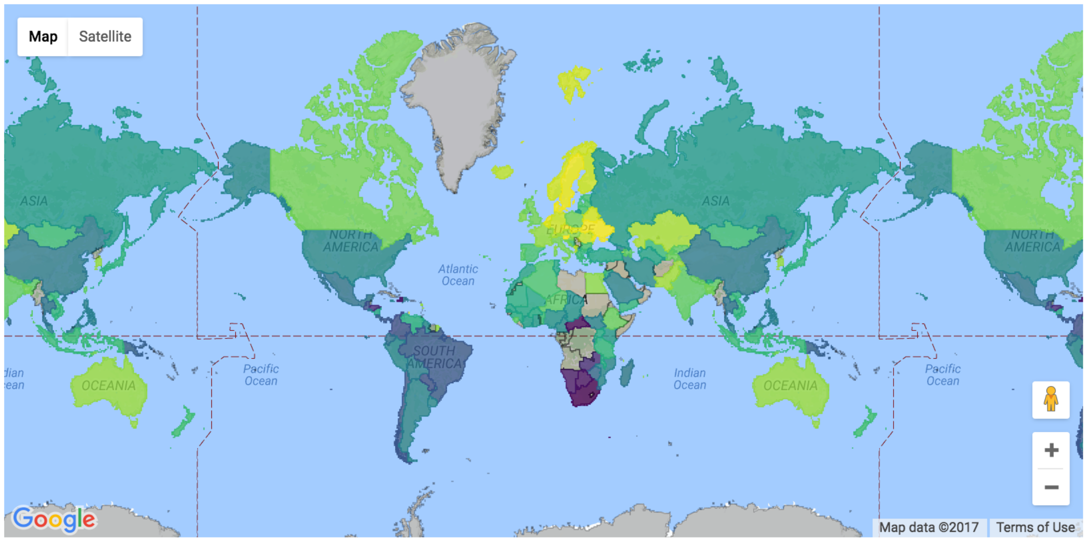
We now need to build an array of colors, one for each country, that we can pass to the GeoJSON layer. The easiest way to do this is to iterate over the array of features in the GeoJSON:

```
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
```



3.7.1 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')

```

(continues on next page)

(continued from previous page)

```
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.

3.7.2 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:

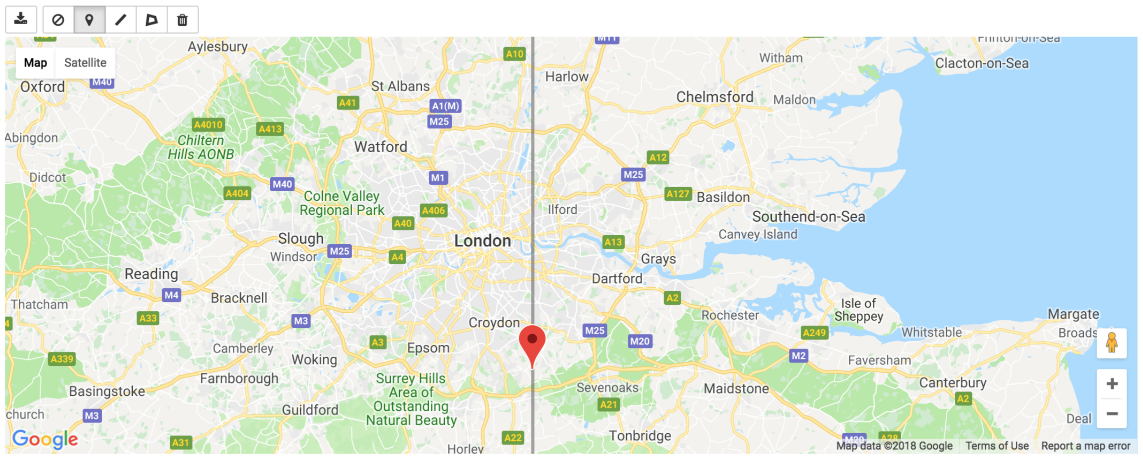
```
import json
import gmaps
gmaps.configure(api_key='AIza...')

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
```

3.8 Drawing markers, lines and polygons

The *drawing layer* lets you draw complex shapes on the map. You can add markers, lines and polygons directly to maps. Let's, for instance, draw the Greenwich meridian and add a marker on Greenwich itself:

```
import gmaps
gmaps.configure(api_key='AIza...')

fig = gmaps.figure(center=(51.5, 0.1), zoom_level=9)

# Features to draw on the map
gmt_meridian = gmaps.Line(
    start=(52.0, 0.0),
    end=(50.0, 0.0),
    stroke_weight=3.0
)
greenwich = gmaps.Marker((51.3, 0.0), info_box_content='Greenwich')

drawing = gmaps.drawing_layer(features=[greenwich, gmt_meridian])
fig.add_layer(drawing)
fig
```

In [31]:
```python
import gmaps
fig = gmaps.figure(center=(51.5, 0.1), zoom_level=9)

# Features to draw on the map
gmt_meridian = gmaps.Line(
    start=(52.0, 0.0),
    end=(50.0, 0.0),
    stroke_weight=3.0
)
greenwich = gmaps.Marker((51.3, 0.0), info_box_content="Greenwich")

drawing = gmaps.drawing_layer(features=[greenwich, gmt_meridian])
fig.add_layer(drawing)
fig
```



Adding the drawing layer to a map displays drawing controls that lets users add arbitrary shapes to the map. This is useful if you want to react to user events (for instance, if you want to run some Python code every time the user adds a marker). This is discussed in the *Reacting to user actions on the map* section.

To hide the drawing controls, pass `show_controls=False` as argument to the drawing layer:

```python
drawing = gmaps.drawing_layer(
    features=[greenwich, gmt_meridian],
    show_controls=False
)
```

Besides lines and markers, you can also draw polygons on the map. This is useful for drawing complex shapes. For instance, we can draw the London congestion charge zone. *jupyter-gmaps* has a built-in dataset with the coordinates of this zone:

```python
import gmaps
import gmaps.datasets

london_congestion_zone_path = gmaps.datasets.load_dataset('london_congestion_zone')
london_congestion_zone_path[:2]
# [(51.530318, -0.123026), (51.530078, -0.123614)]
```

We can draw this on the map with a `gmaps.Polygon`:

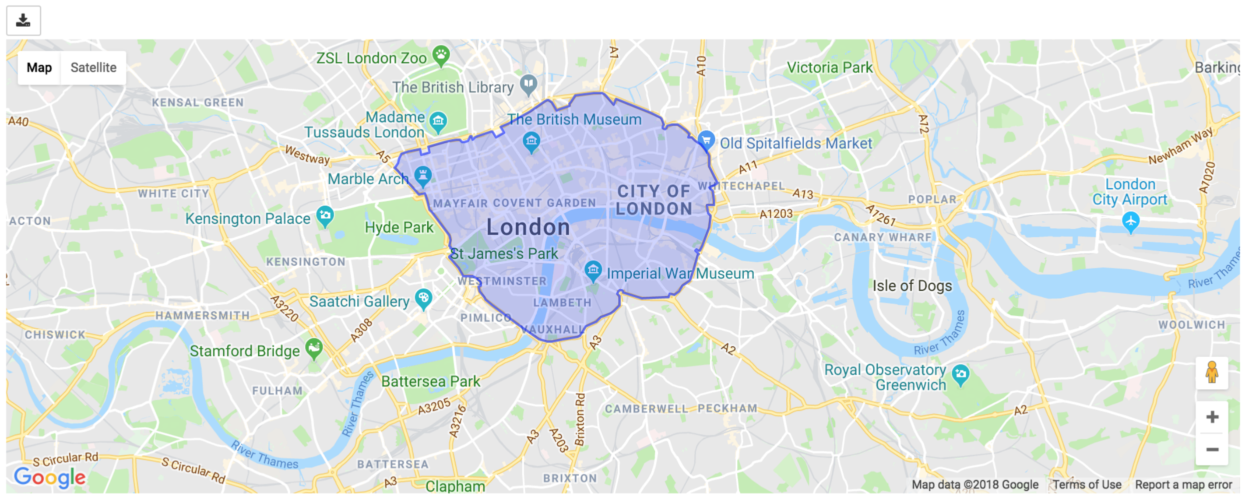
```python
fig = gmaps.figure(center=(51.5, -0.1), zoom_level=12)
london_congestion_zone_polygon = gmaps.Polygon(
    london_congestion_zone_path,
    stroke_color='blue',
    fill_color='blue'
)
drawing = gmaps.drawing_layer(
```

(continues on next page)

(continued from previous page)

```
    features=[london_congestion_zone_polygon],
    show_controls=False
)
fig.add_layer(drawing)
fig
```

```
fig = gmaps.figure(center=(51.5, -0.1), zoom_level=12)
london_congestion_zone_polygon = gmaps.Polygon(
    london_congestion_zone_path,
    stroke_color='blue',
    fill_color='blue'
)
drawing = gmaps.drawing_layer(
    features=[london_congestion_zone_polygon],
    show_controls=False
)
fig.add_layer(drawing)
fig
```



We can pass an arbitrary list of *(latitude, longitude)* pairs to *gmaps.Polygon* to specify complex shapes. For details on how to style polygons, see the `gmaps.Polygon` API documentation.

See the API documentation for `gmaps.drawing_layer()` for an exhaustive list of options for the drawing layer.

3.9 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:

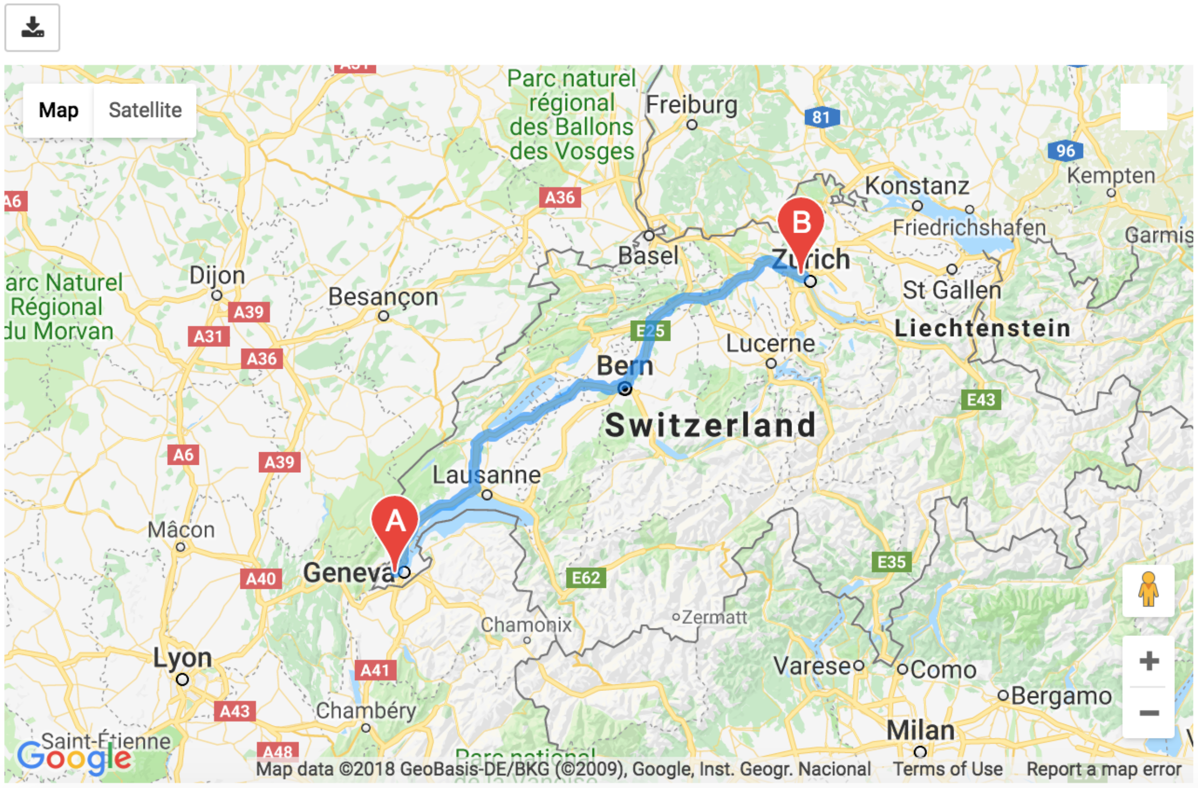
```
import gmaps
import gmaps.datasets
gmaps.configure(api_key='AIza...')

# 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)
```

(continues on next page)

(continued from previous page)

```
fig.add_layer(geneva2zurich)
fig
```

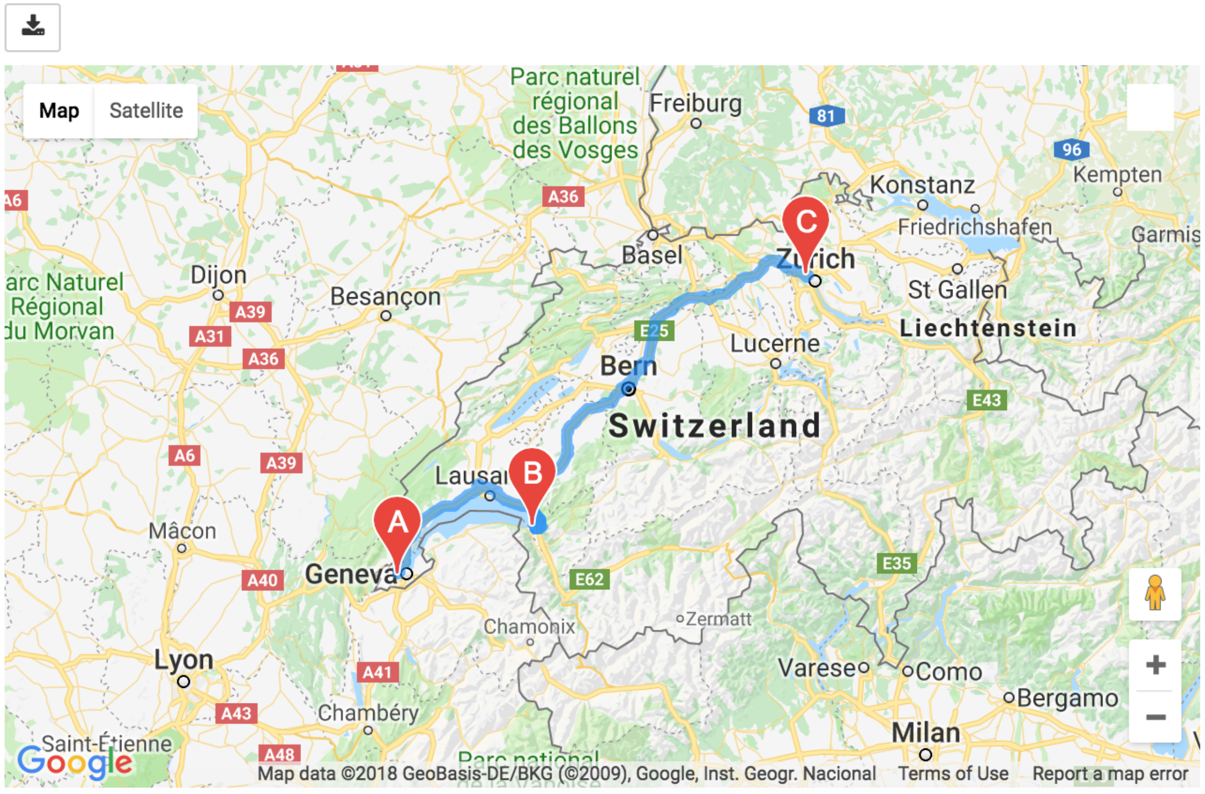
```
geneva = (46.2, 6.1)
montreux = (46.4, 6.9)
zurich = (47.4, 8.5)

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. 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
```

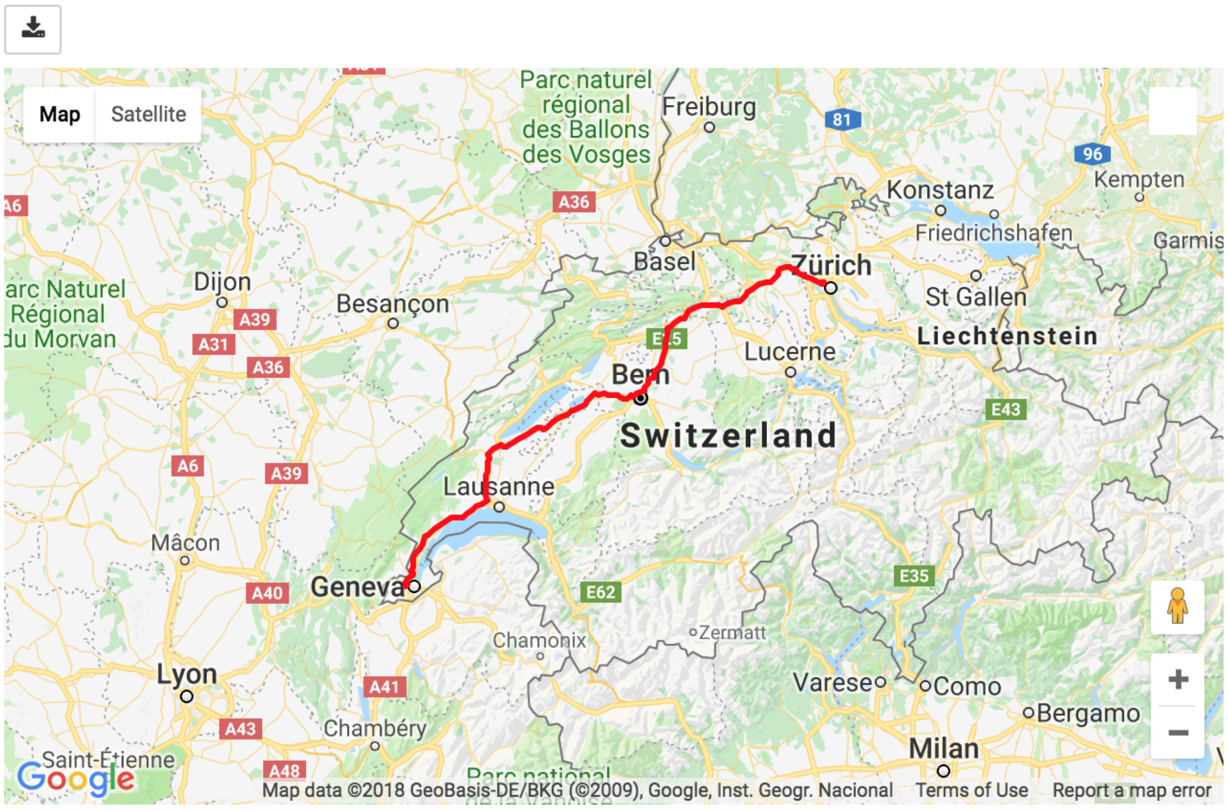
```
fig = gmaps.figure()
geneva2zurich_via_montreux =\
  gmaps.directions_layer(geneva, zurich, waypoints=[montreux])
fig.add_layer(geneva2zurich_via_montreux)
fig
```



You can customise how directions are rendered on the map:

```
fig = gmaps.figure()
geneva2zurich = gmaps.directions_layer(
    geneva, zurich, show_markers=False,
    stroke_color='red', stroke_weight=3.0, stroke_opacity=1.0)
fig.add_layer(geneva2zurich)
fig
```

```
fig = gmaps.figure()
geneva2zurich = gmaps.directions_layer(
    geneva, zurich, show_markers=False,
    stroke_color='red', stroke_weight=3.0, stroke_opacity=1.0)
fig.add_layer(geneva2zurich)
fig
```



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

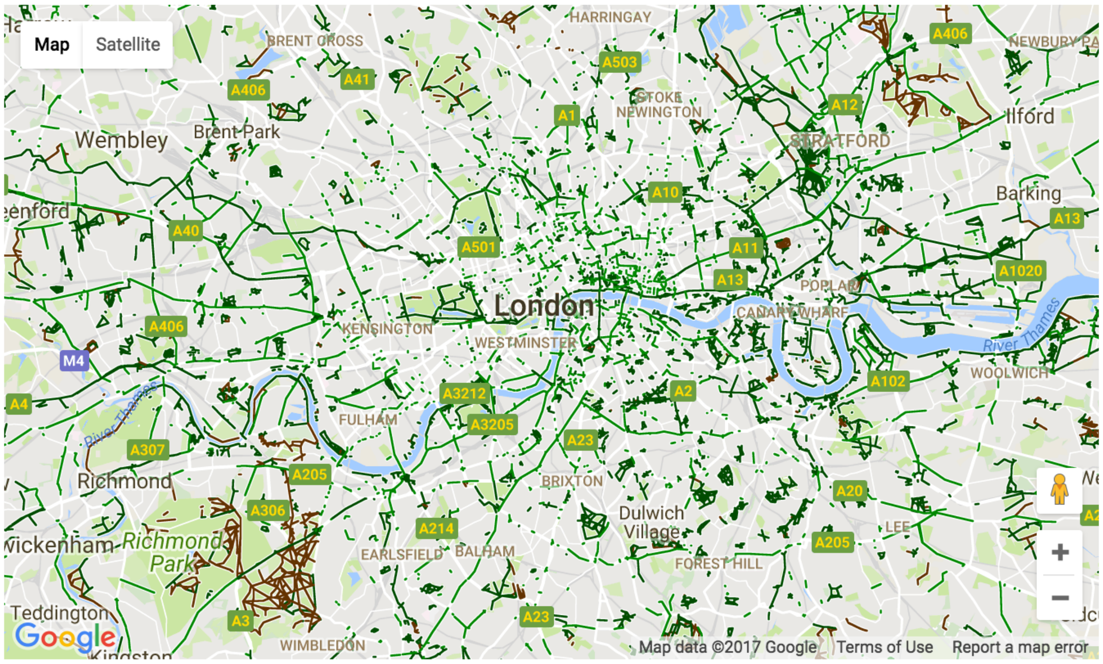
Updating options on the layer object will update the map. This lets you use the directions layer as part of a larger widget application. See the app tutorial for details.

3.10 Bicycling, transit and traffic layers

You can add bicycling, transit and traffic information to a base map. For instance, use *gmaps.bicycling_layer()* to draw cycle lanes. This will also change the style of the base layer to de-emphasize streets which are not cycle-friendly.

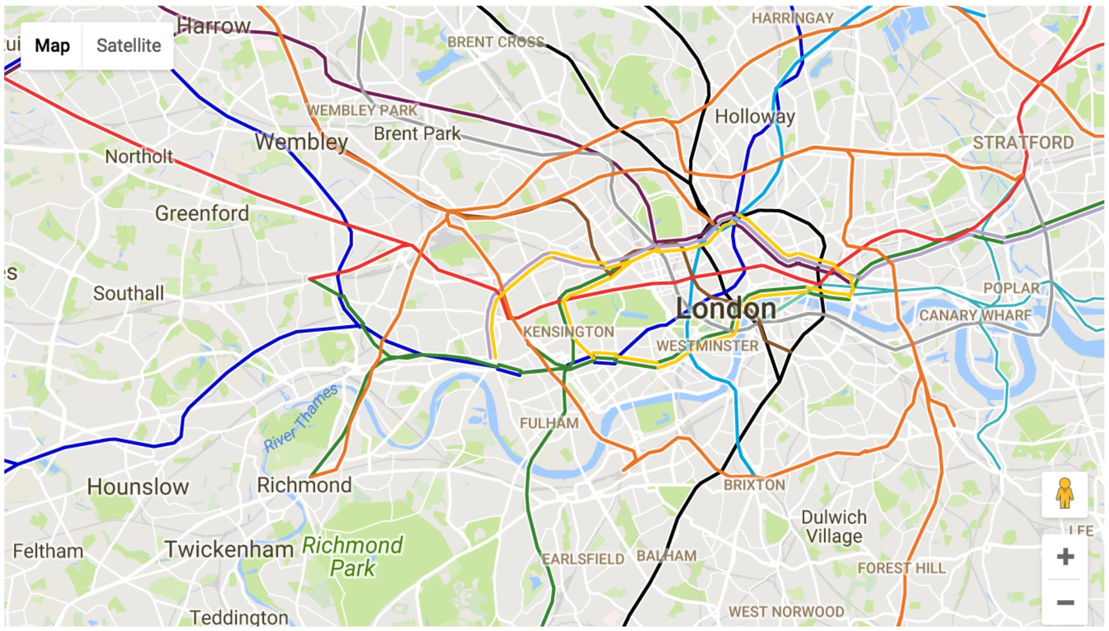
```
import gmaps
gmaps.configure(api_key='AI...')

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

```
In [5]: fig = gmaps.figure(center=(51.5, -0.2), zoom_level=11)
        fig.add_layer(gmaps.bicycling_layer())
        fig
```



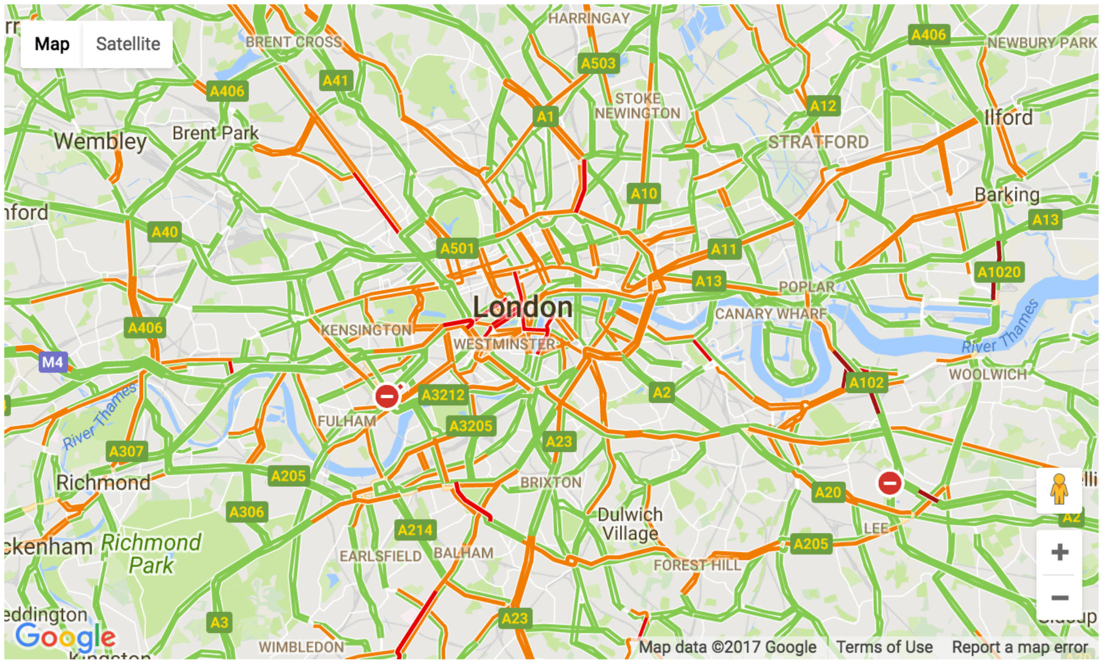
Similarly, the transit layer, available as *gmaps.transit_layer()*, adds information about public transport, where available.

In [4]:
```
fig = gmaps.figure(center=(51.5, -0.2), zoom_level=11)
fig.add_layer(gmaps.transit_layer())
fig
```



The traffic layer, available as *gmaps.traffic_layer()*, adds information about the current state of traffic.

In [5]:
```
fig = gmaps.figure(center=(51.5, -0.2), zoom_level=11)
fig.add_layer(gmaps.traffic_layer())
fig
```



Unlike the other layers, these layers do not take any user data. Thus, *jupyter-gmaps* will not use them to center the map. This means that, if you use these layers by themselves, you will often want to center the figure explicitly, using the `center` and `zoom_level` attributes.

CHAPTER 4

Building applications with *jupyter-gmaps*

You can use *jupyter-gmaps* as a component in a Jupyter widgets application. Jupyter widgets let you embed rich user interfaces

- you can use maps as a way to get user input. The drawing layer lets users draw markers, lines or polygons on the map. We can specify arbitrary Python code that runs whenever a shape is added to the map. As an example, we will build an application where, whenever the user places a marker, we retrieve the address of the marker and write it in a text widget.
- you can use maps as a way to display the result of an external computation. For instance, if you have timestamped geographical data (for instance, you have the date and coordinates of a series of events), you can combine a heatmap with a slider to see how events unfold over time.

4.1 Reacting to user actions on the map

The drawing layer lets us specify Python code to be executed whenever the user adds a feature (like a marker, a line or a polygon) to the map. To demonstrate this, we will build a small application for *reverse geocoding*: when the user places a marker on the map, we will find the address closest to that marker and write it in a text widget. We will use geopy, a wrapper around several geocoding APIs, to calculate the address from the marker's coordinates.

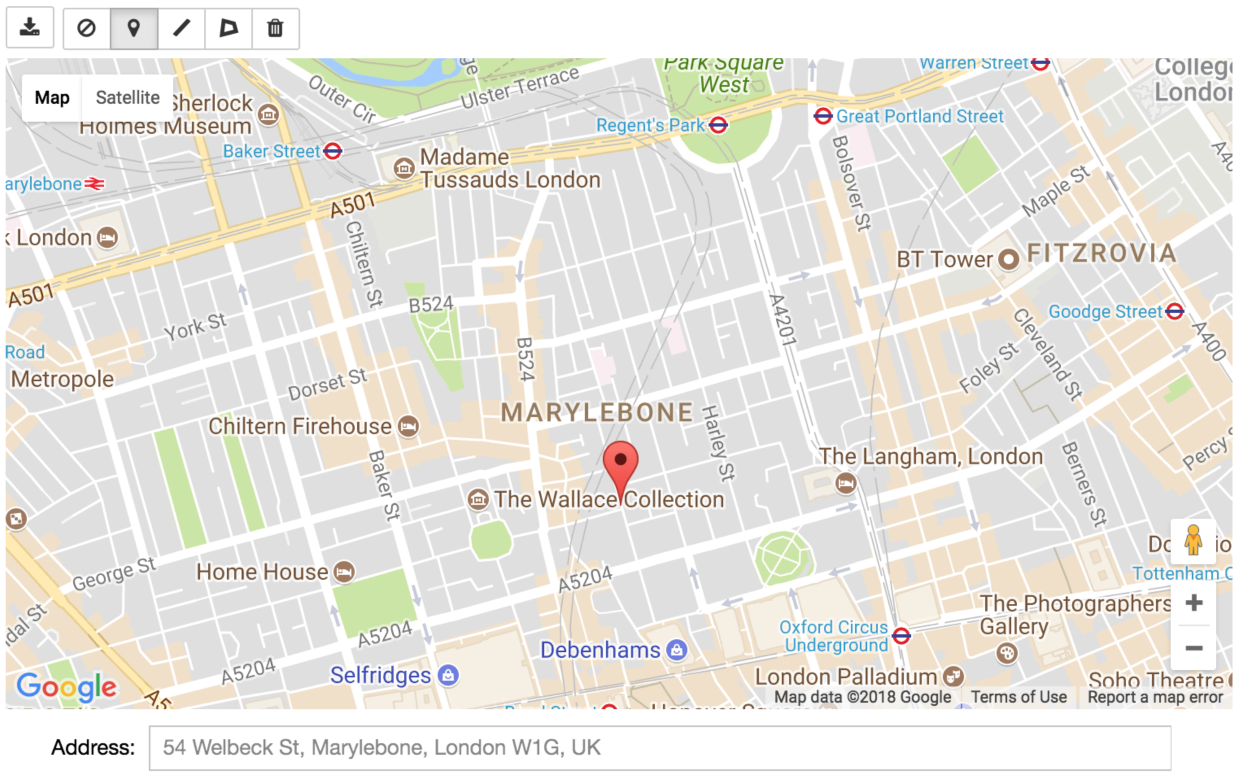
This is the entire code listing:

```python
import ipywidgets as widgets
import geopy
import gmaps

API_KEY = 'AIz...'

gmaps.configure(api_key=API_KEY)

class ReverseGeocoder(object):
    """
    Jupyter widget for finding addresses.
```

(continues on next page)

(continued from previous page)

```
    The user places markers on a map. For each marker,
    we use `geopy` to find the nearest address to that
    marker, and write that address in a text box.
    """

    def __init__(self):
        self._figure = gmaps.figure()
        self._drawing = gmaps.drawing_layer()
        self._drawing.on_new_feature(self._new_feature_callback)
        self._figure.add_layer(self._drawing)
        self._address_box = widgets.Text(
            description='Address: ',
            disabled=True,
            layout={'width': '95%', 'margin': '10px 0 0 0'}
        )
        self._geocoder = geopy.geocoders.GoogleV3(api_key=API_KEY)
        self._container = widgets.VBox([self._figure, self._address_box])

    def _get_location_details(self, location):
        return self._geocoder.reverse(location, exactly_one=True)

    def _clear_address_box(self):
        self._address_box.value = ''

    def _show_address(self, location):
        location_details = self._get_location_details(location)
        if location_details is None:
            self._address_box.value = 'No address found'
        else:
            self._address_box.value = location_details.address

    def _new_feature_callback(self, feature):
        try:
            location = feature.location
        except AttributeError:
            return # Not a marker

        # Clear address box to signify to the user that something is happening
        self._clear_address_box()

        # Remove all markers other than the one that has just been added.
        self._drawing.features = [feature]

        # Compute the address and display it
        self._show_address(location)

    def render(self):
        return self._container

ReverseGeocoder().render()
```



There are several things to note:

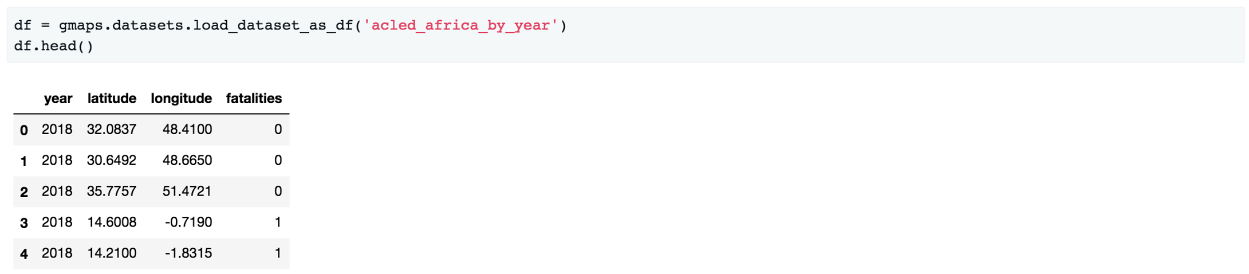
- We wrap the application in a `ReverseGeocoder` class. Wrapping your application in a class (rather than using the notebook's global namespace) helps with encapsulation and lets you instantiate this widget multiple times. Since the flow through widget applications is often more complex than linear data analysis workflows, encapsulation will improve your ability to reason about the code.
- As part of the class constructor, we use *`gmaps.figure()`* to create a figure. We then use *`gmaps.drawing_layer()`* to create a drawing layer, which we add to the figure. We also create a `widgets.Text` widget. This is a text box in which we will write the address. We then wrap our figure and the text box in a single `widgets.VBox`, a widget container that stacks widgets vertically.
- We register a callback on the drawing layer using `.on_new_feature`. The function that we pass in to `.on_new_feature` will get called whenever the user adds a feature to the map. This is the hook that lets us build complex applications on top of the drawing layer: we can run arbitrary Python code when the user adds a marker to the map.
- In the `.on_new_feature` callback, we first check whether the feature that has been added is a marker (the user could, in principle, have added another feature type, like a line, to the map).
- Assuming the feature is a valid marker, we first clear the text widget containing the address. This gives feedback to the user that something is happening.
- We then re-write the `.features` array of the drawing layer, keeping just the marker that the user has just added. This clears previous markers, avoiding clutter on the map.
- We then use geopy to find the adddress. Assuming the address is valid, display it in the text widget.

4.2 Updating data in response to other widgets

Many layers support updating the data without re-rendering the entire map. This is useful for exploring multi-dimensional datasets, especially in conjunction with other widgets.

As an example, we will use the `acled_africa_by_year` dataset, a dataset indexing violence against civilians in Africa. The original dataset is from the ACLED project. The dataset has four columns:

```
import gmaps.datasets

df = gmaps.datasets.load_dataset_as_df('acled_africa_by_year')
df.head()
```



```
df = gmaps.datasets.load_dataset_as_df('acled_africa_by_year')
df.head()
```

	year	latitude	longitude	fatalities
0	2018	32.0837	48.4100	0
1	2018	30.6492	48.6650	0
2	2018	35.7757	51.4721	0
3	2018	14.6008	-0.7190	1
4	2018	14.2100	-1.8315	1

We will build an application that lets the user explore different years via a slider. When the user changes the slider, we display the total number of fatalities for that year, and update a heatmap showing the distribution of conflicts.

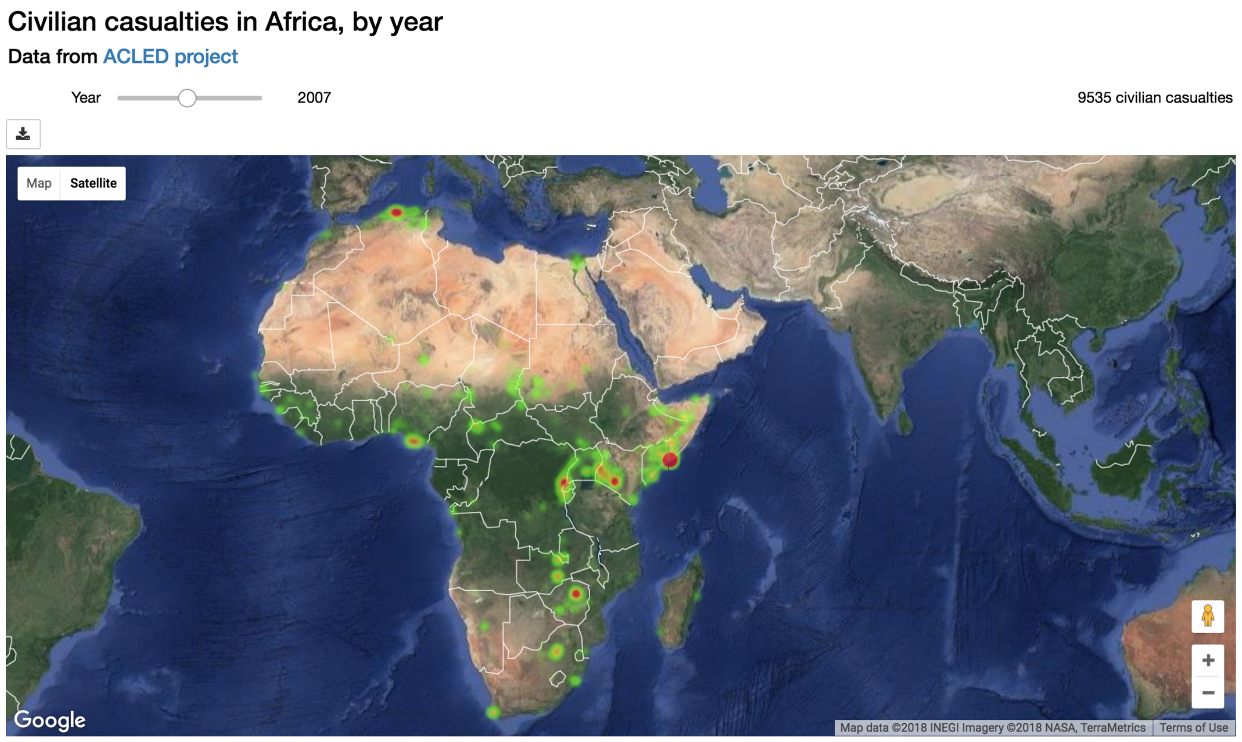
This is the entire code listing:

```
from IPython.display import display
import ipywidgets as widgets

import gmaps
gmaps.configure(api_key='AIza...')

class AcledExplorer(object):
    """
    Jupyter widget for exploring the ACLED dataset.

    The user uses the slider to choose a year. This renders
    a heatmap of civilian victims in that year.
    """

    def __init__(self, df):
        self._df = df
        self._heatmap = None
        self._slider = None
        initial_year = min(self._df['year'])

        title_widget = widgets.HTML(
            '<h3>Civilian casualties in Africa, by year</h3>'
            '<h4>Data from <a href="https://www.acleddata.com/">ACLED project</a></h4>
↪'
        )

        map_figure = self._render_map(initial_year)
        controls = self._render_controls(initial_year)
        self._container = widgets.VBox([title_widget, controls, map_figure])
```

(continues on next page)

(continued from previous page)

```
    def render(self):
        display(self._container)

    def _on_year_change(self, change):
        year = self._slider.value
        self._heatmap.locations = self._locations_for_year(year)
        self._total_box.value = self._total_casualties_text_for_year(year)
        return self._container

    def _render_map(self, initial_year):
        fig = gmaps.figure(map_type='HYBRID')
        self._heatmap = gmaps.heatmap_layer(
            self._locations_for_year(initial_year),
            max_intensity=100,
            point_radius=8
        )
        fig.add_layer(self._heatmap)
        return fig

    def _render_controls(self, initial_year):
        self._slider = widgets.IntSlider(
            value=initial_year,
            min=min(self._df['year']),
            max=max(self._df['year']),
            description='Year',
            continuous_update=False
        )
        self._total_box = widgets.Label(
            value=self._total_casualties_text_for_year(initial_year)
        )
        self._slider.observe(self._on_year_change, names='value')
        controls = widgets.HBox(
            [self._slider, self._total_box],
            layout={'justify_content': 'space-between'}
        )
        return controls

    def _locations_for_year(self, year):
        return self._df[self._df['year'] == year][['latitude', 'longitude']]

    def _total_casualties_for_year(self, year):
        return int(self._df[self._df['year'] == year]['year'].count())

    def _total_casualties_text_for_year(self, year):
        return '{} civilian casualties'.format(self._total_casualties_for_year(year))


AcledExplorer(df).render()
```


Civilian casualties in Africa, by year
Data from ACLED project
Year
2007
9535 civilian casualties
Map
Satellite
Google
Map data ©2018 INEGI Imagery ©2018 NASA, TerraMetrics
Terms of Use

There are several things to note on this:

- We wrap the application in a class to help keep the mutable state encapsulated.
- As part of the class constructor, we use *gmaps.figure()* to create a figure. We add use *gmaps.heatmap_layer()* to create a heatmap, which we add to the figure. The `Heatmap` object returned has a `locations` attribute. Setting this to a new value will automatically update the heatmap.
- We create a slider with `widgets.IntSlider`. In general, *jupyter-gmaps* objects are designed to interact with widgets from ipywidgets. For a full list of available widgets, see the ipywidgets documentation.
- We want to react to changes in the slider: every time the slider moves, we recompute the total number of fatalities and update the data in the heatmap. To react to changes in a widget, we use the `.observe` method on the widget. This lets us specify a callback that gets called whenever a given attribute of the widget changes. We pass the `names="value"` argument to `slider.observe` to only react to changes in the slider's `value` attribute. Note that the callback (`self.render` in our case) needs to take a single argument. It gets passed a dictionary describing the change.
- To build the layout for our application, we use combinations of HBox and VBox widgets.

CHAPTER 5

Exporting maps

5.1 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).

5.2 Exporting to HTML

You can export maps to HTML using the infrastructure provided by *ipywidgets*. For instance, let's export a simple map to HTML:

```python
from ipywidgets.embed import embed_minimal_html
import gmaps

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

fig = gmaps.figure()
embed_minimal_html('export.html', views=[fig])
```

This generates a file, `export.html`, with two (or more) `<script>` tags that contain the widget state. The scripts with tag `<script type="application/vnd.jupyter.widget-view+json">` indicate where the widgets will be placed in the DOM. You can move these around and nest them in other DOM elements to change where the exported maps appear in the DOM.

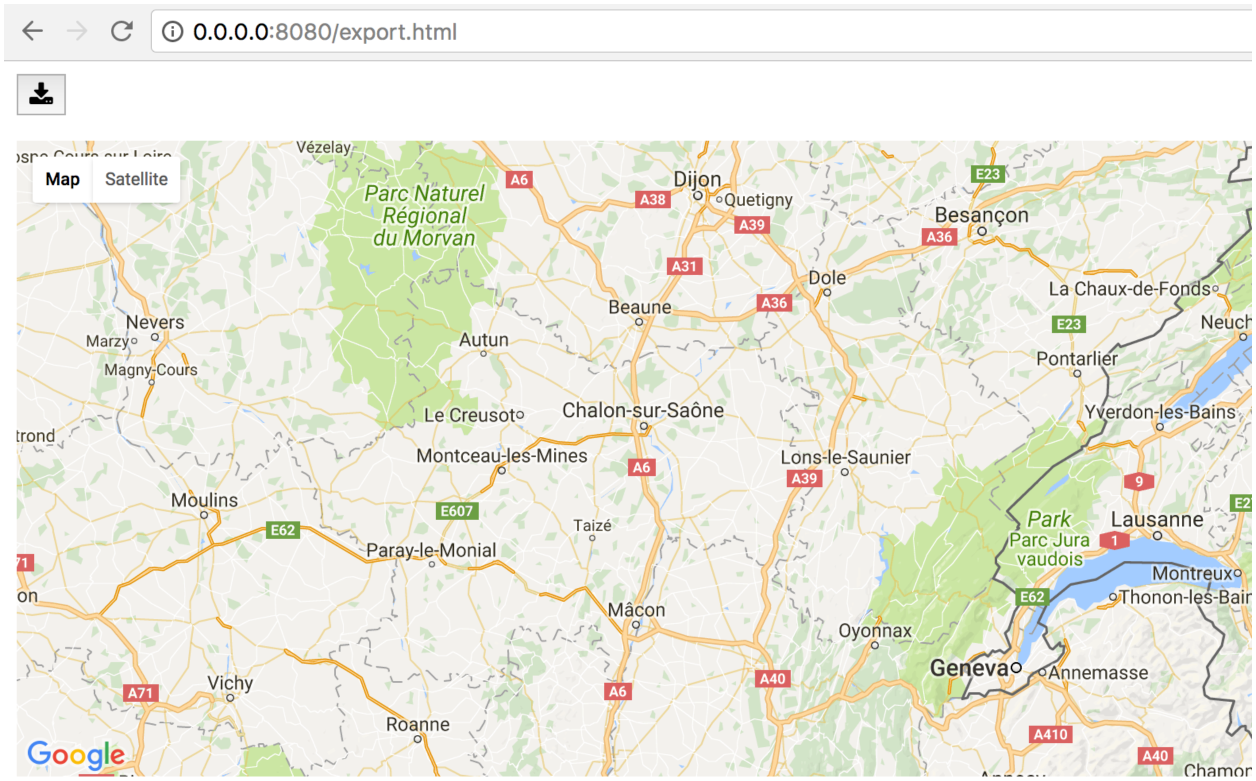
Open `export.html` with a webserver, e.g. by running, if you use Python 3:

```
python -m http.server 8080
```

Or, if you use Python 2:

```
python -m SimpleHTTPServer 8080
```

Navigate to *http://0.0.0.0:8080/export.html* and you should see the export!



The module `ipywidgets.embed` contains other functions for exporting that will give you greater control over what is exported. See the documentation and the source code for more details.

CHAPTER 6

API documentation

6.1 Figures and layers

`gmaps.`**`figure`**(*display_toolbar=True*, *display_errors=True*, *zoom_level=None*, *tilt=45*, *center=None*, *layout=None*, *map_type='ROADMAP'*, *mouse_handling='COOPERATIVE'*)

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.
- **`tilt`** (*`int, optional`*) – Tilt can be either 0 or 45 indicating the tilt angle in degrees. 45-degree imagery is only available for satellite and hybrid map types, and is not available at every location at every zoom level. For locations where 45-degree imagery is not available, Google Maps will automatically fall back to 0 tilt.
- **`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.
- **`map_type`** (*`str, optional`*) – String representing the type of map to show. One of 'ROADMAP' (the classic Google Maps style) 'SATELLITE' (just satellite tiles with no overlay), 'HYBRID' (satellite base tiles but with features such as roads and cities overlaid) and 'TERRAIN' (map showing terrain features). Defaults to 'ROADMAP'.
- **`mouse_handling`** (*`str, optional`*) – String representing how the map captures the page's mouse event. One of 'COOPERATIVE' (scroll events scroll the page without zoom-

ing the map, double clicks or CTRL/CMD+scroll zoom the map), 'GREEDY' (the map captures all scroll events), 'NONE' (the map cannot be zoomed or panned by user gestures) or 'AUTO' (cooperative if the notebook is displayed in an iframe, greedy otherwise). Defaults to 'COOPERATIVE'.

- **layout** (*dict, optional*) – Control the layout of the figure, e.g. its width, height, border etc. For instance, passing `layout={'width': '400px', 'height': '300px'}` will build a figure of fixed width and height. For more in formation on available properties, see the ipywidgets documentation on widget layout.

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)
```

To customise the layout:

```
>>> fig = gmaps.figure(layout={
        'width': '400px',
        'height': '600px',
        'padding': '3px',
        'border': '1px solid black'
})
```

To have a satellite map:

```
>>> fig = gmaps.figure(map_type='HYBRID')
```

`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)
```

(continues on next page)

(continued from previous page)

```
>>> 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)
    ]
```

(continues on next page)

(continued from previous page)

```
>>> 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.`**`drawing_layer`**(*features=None*, *mode='MARKER'*, *show_controls=True*, *marker_options=None*, *line_options=None*, *polygon_options=None*)

Create an interactive drawing layer

Adding a drawing layer to a map allows adding custom shapes, both programatically and interactively (by drawing on the map).

Examples

You can use the drawing layer to add lines, markers and polygons to a map:

```
>>> fig = gmaps.figure()
>>> drawing = gmaps.drawing_layer(features=[
     gmaps.Line((46.23, 5.86), (46.44, 5.24), stroke_weight=3.0),
     gmaps.Marker((46.88, 5.45), label='D'),
     gmaps.Polygon(
         [(46.72, 6.06), (46.48, 6.49), (46.79, 6.91)],
         fill_color='red'
     )
])
>>> fig.add_layer(drawing)
>>> fig
```

You can also use the drawing layer as a way to get user input. The user can draw features on the map. You can then get the list of features programatically.

```
>>> fig = gmaps.figure()
>>> drawing = gmaps.drawing_layer()
>>> fig.add_layer(drawing)
>>> fig
>>> # Now draw on the map
>>> drawing.features
```

(continues on next page)

(continued from previous page)

```
[Marker(location=(46.83, 5.56)),
Marker(location=(46.46, 5.91)),
Line(end=(46.32, 5.98), start=(46.42, 5.12))]
```

You can bind callbacks that are executed when a new feature is added. For instance, you can use geopy to get the address corresponding to markers that you add on the map:

```
API_KEY = "Aiz..."

import gmaps
import geopy

gmaps.configure(api_key=API_KEY)
fig = gmaps.figure()
drawing = gmaps.drawing_layer()

geocoder = geopy.geocoders.GoogleV3(api_key=API_KEY)

def print_address(feature):
    try:
        print(geocoder.reverse(feature.location, exactly_one=True))
    except AttributeError as e:
        # Not a marker
        pass

drawing.on_new_feature(print_feature)
fig.add_layer(drawing)
fig  # display the figure
```

Parameters

- **`features`** (*`list of features, optional`*) – List of features to draw on the map. Features must be one of `gmaps.Marker`, `gmaps.Line` or `gmaps.Polygon`.
- **`marker_options`** (`gmaps.MarkerOptions`, *dict* or *None*, optional) – Options controlling how markers are drawn on the map. Either pass in an instance of `gmaps.MarkerOptions`, or a dictionary with keys *hover_text*, *display_info_box*, *info_box_content*, *label* (or a subset of these). See `gmaps.MarkerOptions` for documentation on possible values. Note that this only affects the initial options of markers added to the map by a user. To customise markers added programatically, pass in the options to the `gmaps.Marker` constructor.
- **`line_options`** (`gmaps.LineOptions`, *dict* or *None*, optional) – Options controlling how new lines are drawn on the map. Either pass in an instance of `gmaps.LineOptions`, or a dictionary with keys *stroke_weight*, *stroke_color*, *stroke_opacity* (or a subset of these). See `gmaps.LineOptions` for documentation on possible values. Note that this only affects the initial options of lines added to the map by a user. To customise lines added programatically, pass in the options to the `gmaps.Line` constructor.
- **`polygon_options`** (`gmaps.PolygonOptions`, *dict* or *None*, optional) – Options controlling how new polygons are drawn on the map. Either pass in an instance of `gmaps.PolygonOptions`, or a dictionary with keys *stroke_weight*, *stroke_color*, *stroke_opacity*, *fill_color*, *fill_opacity* (or a subset of these). See `gmaps.PolygonOptions` for documentation on possible values. Note that this only affects the initial options of polygons added to the map by a user. To customise polygons added programatically, pass in the options to the `gmaps.Polygon` constructor.

- **mode** (*str, optional*) – Initial drawing mode. One of DISABLED, MARKER, LINE, POLYGON or DELETE. Defaults to MARKER if show_controls is True, otherwise defaults to DISABLED.
- **show_controls** (*bool, optional*) – Whether to show the drawing controls in the map toolbar. Defaults to True.

Returns A *gmaps.Drawing* widget.

gmaps.**directions_layer**(*start, end, waypoints=None, avoid_ferries=False, travel_mode='DRIVING', avoid_highways=False, avoid_tolls=False, optimize_waypoints=False, show_markers=True, show_route=True, stroke_color='#0088FF', stroke_weight=6.0, stroke_opacity=0.6*)

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')
```

You can choose to hide the markers, the route or both:

```
>>> directions = gmaps.directions_layer(
        start, end, show_markers=False, show_route=False)
```

Control how the route is displayed by changing the *stroke_color*, *stroke_weight* and *stroke_opacity* attributes.

```
>>> directions = gmaps.directions_layer(
        start, end, stroke_color='red',
        stroke_opacity=1.0, stroke_weight=2.0)
```

You can update parameters on an existing layer. This will automatically update the map:

```
>>> directions.travel_mode = 'DRIVING'
>>> directions.start = (46.4, 6.1)
>>> directions.stroke_color = 'green'
>>> directions.show_markers = False
```

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.
- **show_markers** (*bool, optional*) – If set to False, the markers showing the start, destination and waypoints are explicitly hidden. Defaults to True.
- **show_route** (*bool, optional*) – If set to False, the line indicating the route is explicitly hidden. Defaults to True.
- **stroke_color** (*str or tuple, optional*) – The stroke color of the line indicating the route. 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). Defaults to a blue color: (0, 88, 255)
- **stroke_weight** (*float, optional*) – The width of the line indicating the route. This is a positive float. Defaults to 6.
- **stroke_opacity** (*float, optional*) – The opacity of the stroke color. The opacity should be a float between 0.0 (transparent) and 1.0 (opaque). 0.6 by default.

Returns A *gmaps.Directions* widget.

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
```

`gmaps.`**`traffic_layer`**`(`*auto_refresh=True*`)`

Traffic layer.

Adds information about the current state of traffic to the map. This layer only works at sufficiently high zoom levels, and for regions for which Google Maps has traffic information.

Parameters **`auto_refresh`** (*`bool, optional`*) – Whether the traffic layer refreshes with updated information automatically. This is true by default.

Returns A *`gmaps.Traffic`* widget.

Examples

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

6.2 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.

6.3 Low level widgets

`class` `gmaps.`**`Figure`**`(`**args*, ***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.drawing_layer()`	Create a layer of custom features, and allow users to draw on the map
`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
`gmaps.traffic_layer()`	Create a layer showing current traffic information

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.
- **map_type** (*str, optional*) – String representing the type of map to show. One of 'ROADMAP' (the classic Google Maps style) 'SATELLITE' (just satellite tiles with no overlay), 'HYBRID' (satellite base tiles but with features such as roads and cities overlaid) and 'TERRAIN' (map showing terrain features). Defaults to 'ROADMAP'.
- **mouse_handling** (*str, optional*) – String representing how the map captures the page's mouse event. One of 'COOPERATIVE' (scroll events scroll the page without zooming the map, double clicks or CTRL/CMD+scroll zoom the map), 'GREEDY' (the map captures all scroll events), 'NONE' (the map cannot be zoomed or panned by user gestures) or 'AUTO' (cooperative if the notebook is displayed in an iframe, greedy otherwise). Defaults to 'COOPERATIVE'.

Examples

```
>>> m = gmaps.Map()
>>> 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.Map(initial_viewport=viewport)
```

To have a satellite map:

```
>>> m = gmaps.Map(map_type='HYBRID')
```

You can also change this dynamically:

```
>>> m.map_type = 'TERRAIN'
```

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.

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.
- **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).
- **data** (*list of tuples*) – DEPRECATED. Use *locations* instead. 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).

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)
```

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*.

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** (*list of floats*) – List of non-negative floats corresponding to the importance of each latitude-longitude pair. Must have the same length as *locations*.
- **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).
- **data** (*list of tuples*) – DEPRECATED. Use *locations* and *weights* instead. 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.

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)
```

class gmaps.**Symbol**(*location*, ***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.**MarkerOptions**(***kwargs*)

Style options for a marker

Parameters

- **label** (*string, optional*) – Text to be displayed inside the marker. Google maps only displays the first letter of this string.
- **hover_text** (*string, optional*) – Text to be displayed when a user's mouse is hovering over the marker. If this is set to an empty string, nothing will appear when the user's mouse hovers over a marker.
- **display_info_box** (*bool, optional*) – Whether to display an info box when the user clicks on a marker. Defaults to `True` if `info_box_content` is not an empty string, or `False` otherwise.
- **info_box_content** (*string, optional*) – Content to be displayed in a box above a marker, when the user clicks on it.

class gmaps.**Marker**(*location*, ***kwargs*)

Class representing a marker.

Markers should be added to the map via the *gmaps.marker_layer()* function or the *gmaps.drawing_layer()* function.

Parameters

- **location** (*tuple of floats*) – (latitude, longitude) pair denoting the location of the marker. 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).
- **label** (*string, optional*) – Text to be displayed inside the marker. Google maps only displays the first letter of this string.

- **hover_text** (*string, optional*) – Text to be displayed when a user's mouse is hovering over the marker. If this is set to an empty string, nothing will appear when the user's mouse hovers over a marker.
- **display_info_box** (*bool, optional*) – Whether to display an info box when the user clicks on a marker. Defaults to `True` if `info_box_content` is not an empty string, or `False` otherwise.
- **info_box_content** (*string, optional*) – Content to be displayed in a box above a marker, when the user clicks on it.

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**(*start=None*, *end=None*, *waypoints=None*, ***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)
>>> 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')
```

You can choose to hide the markers, the route or both:

```
>>> directions = gmaps.directions_layer(
        start, end, show_markers=False, show_route=False)
```

Control how the route is displayed by changing the *stroke_color*, *stroke_weight* and *stroke_opacity* attributes.

```
>>> directions = gmaps.directions_layer(
        start, end, stroke_color='red',
        stroke_opacity=1.0, stroke_weight=2.0)
```

You can update parameters on an existing layer. This will automatically update the map:

```
>>> directions.travel_mode = 'DRIVING'
>>> directions.start = (46.4, 6.1)
>>> directions.stroke_color = 'green'
>>> directions.show_markers = False
```

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.
- **show_markers** (*bool, optional*) – If set to False, the markers showing the start, destination and waypoints are explicitly hidden. Defaults to True.
- **show_route** (*bool, optional*) – If set to False, the line indicating the route is explicitly hidden. Defaults to True.
- **stroke_color** (*str or tuple, optional*) – The stroke color of the line indicating the route. 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). Defaults to a blue color: (0, 88, 255)
- **stroke_weight** (*float, optional*) – The width of the line indicating the route. This is a positive float. Defaults to 6.
- **stroke_opacity** (*float, optional*) – The opacity of the stroke color. The opacity should be a float between 0.0 (transparent) and 1.0 (opaque). 0.6 by default.

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
```

class gmaps.**Traffic**(***kwargs*)

Traffic layer

Add this to a `gmaps.Map` or a `gmaps.Figure` instance to add traffic information to the map, where supported.

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

Examples

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

Parameters **auto_refresh** (*bool, optional*) – Whether the traffic layer refreshes with updated information automatically. This is true by default.

class gmaps.**Drawing**(***kwargs*)

Widget for a drawing layer

Add this to a `gmaps.Map` or `gmaps.Figure` instance to let you draw on the map.

You should not need to instantiate this directly. Instead, use the `gmaps.drawing_layer()` factory function.

Examples

You can use the drawing layer to add lines, markers and polygons to a map:

```
>>> fig = gmaps.figure()
>>> drawing = gmaps.drawing_layer(features=[
     gmaps.Line((46.23, 5.86), (46.44, 5.24), stroke_weight=3.0),
     gmaps.Marker((46.88, 5.45), label='D'),
     gmaps.Polygon(
         [(46.72, 6.06), (46.48, 6.49), (46.79, 6.91)],
         fill_color='red'
     )
])
>>> fig.add_layer(drawing)
>>> fig
```

You can also use the drawing layer as a way to get user input. The user can draw features on the map. You can then get the list of features programatically.

```
>>> fig = gmaps.figure()
>>> drawing = gmaps.drawing_layer()
>>> fig.add_layer(drawing)
>>> fig
```

(continues on next page)

(continued from previous page)

```
>>> # Now draw on the map
>>> drawing.features
[Marker(location=(46.83, 5.56)),
Marker(location=(46.46, 5.91)),
Line(end=(46.32, 5.98), start=(46.42, 5.12))]
```

You can bind callbacks that are executed when a new feature is added. For instance, you can use geopy to get the address corresponding to markers that you add on the map:

```
API_KEY = "Aiz..."

import gmaps
import geopy

gmaps.configure(api_key=API_KEY)
fig = gmaps.figure()
drawing = gmaps.drawing_layer()

geocoder = geopy.geocoders.GoogleV3(api_key=API_KEY)

def print_address(feature):
    try:
        print(geocoder.reverse(feature.location, exactly_one=True))
    except AttributeError as e:
        # Not a marker
        pass

drawing.on_new_feature(print_feature)
fig.add_layer(drawing)
fig  # display the figure
```

Parameters

- **features** (*list of features, optional*) – List of features to draw on the map. Features must be one of `gmaps.Marker`, `gmaps.Line` or `gmaps.Polygon`.
- **marker_options** (`gmaps.MarkerOptions`, *dict* or *None*, optional) – Options controlling how markers are drawn on the map. Either pass in an instance of `gmaps.MarkerOptions`, or a dictionary with keys *hover_text*, *display_info_box*, *info_box_content*, *label* (or a subset of these). See `gmaps.MarkerOptions` for documentation on possible values. Note that this only affects the initial options of markers added to the map by a user. To customise markers added programatically, pass in the options to the `gmaps.Marker` constructor.
- **line_options** (`gmaps.LineOptions`, *dict* or *None*, optional) – Options controlling how new lines are drawn on the map. Either pass in an instance of `gmaps.LineOptions`, or a dictionary with keys *stroke_weight*, *stroke_color*, *stroke_opacity* (or a subset of these). See `gmaps.LineOptions` for documentation on possible values. Note that this only affects the initial options of lines added to the map by a user. To customise lines added programatically, pass in the options to the `gmaps.Line` constructor.
- **polygon_options** (`gmaps.PolygonOptions`, *dict* or *None*, optional) – Options controlling how new polygons are drawn on the map. Either pass in an instance of `gmaps.PolygonOptions`, or a dictionary with keys *stroke_weight*, *stroke_color*, *stroke_opacity*, *fill_color*, *fill_opacity* (or a subset of these). See `gmaps.PolygonOptions` for documentation on possible values. Note that this only affects the initial options of polygons

added to the map by a user. To customise polygons added programatically, pass in the options to the `gmaps.Polygon` constructor.

- **mode** (*str, optional*) – Initial drawing mode. One of `DISABLED`, `MARKER`, `LINE`, `POLYGON` or `DELETE`. Defaults to `MARKER` if `toolbar_controls.show_controls` is True, otherwise defaults to `DISABLED`.
- **toolbar_controls** (`gmaps.DrawingControls`, optional) – Widget representing the drawing toolbar.

on_new_feature(*callback*)

Register a callback called when new features are added

Parameters **callback** (*callable*) – Callable to be called when a new feature is added. The callback should take a single argument, the feature that has been added. This can be an instance of `gmaps.Line`, `gmaps.Marker` or `gmaps.Polygon`.

class gmaps.**DrawingControls**(***kwargs*)

Widget for the toolbar snippet representing the drawing controls

Parameters **show_controls** (*bool, optional*) – Whether the drawing controls should be shown. Defaults to True.

class gmaps.**Line**(*start, end, stroke_color='#696969', stroke_weight=2.0, stroke_opacity=0.6*)

Widget representing a single line on a map

Add this line to a map via the `gmaps.drawing_layer()` function, or by passing it directly to the `.features` array of an existing instance of `gmaps.Drawing`.

Examples

```
>>> fig = gmaps.figure()
>>> drawing = gmaps.drawing_layer(features=[
     gmaps.Line((46.44, 5.24), (46.23, 5.86), stroke_color='green'),
     gmaps.Line((48.44, 1.32), (47.13, 3.91), stroke_weight=5.0)
])
>>> fig.add_layer(drawing)
```

You can also add a line to an existing `gmaps.Drawing` instance:

```
>>> fig = gmaps.figure()
>>> drawing = gmaps.drawing_layer()
>>> fig.add_layer(drawing)
>>> fig # display the figure
```

You can now add lines directly on the map:

```
>>> drawing.features = [
     gmaps.Line((46.44, 5.24), (46.23, 5.86), stroke_color='green'),
     gmaps.Line((48.44, 1.32), (47.13, 3.91), stroke_weight=5.0)
]
```

Parameters

- **start** (*tuple of floats*) – (latitude, longitude) pair denoting the start of the line. 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).

- **end** – (latitude, longitude) pair denoting the end of the line. 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).
- **stroke_color** (*str or tuple, optional*) – The stroke color of the line. 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). Defaults to a grey color: (69, 69, 69)
- **stroke_weight** (*float, optional*) – How wide the line is. This is a positive float. Defaults to 2.
- **stroke_opacity** (*float, optional*) – The opacity of the stroke color. The opacity should be a float between 0.0 (transparent) and 1.0 (opaque). 0.6 by default.

class gmaps.**LineOptions**(**args*, ***kwargs*)

Style options for a line

Pass an instance of this class to `gmaps.drawing_layer()` to control the style of new user-drawn lines on the map.

Examples

```
>>> fig = gmaps.figure()
>>> drawing = gmaps.drawing_layer(
        marker_options=gmaps.MarkerOptions(hover_text='some text'),
        line_options=gmaps.LineOptions(stroke_color='red')
    )
>>> fig.add_layer(drawing)
>>> fig # display the figure
```

Parameters

- **stroke_color** (*str or tuple, optional*) – The stroke color of the line. 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). Defaults to a grey color: (69, 69, 69)
- **stroke_weight** (*float, optional*) – How wide the line is. This is a positive float. Defaults to 2.
- **stroke_opacity** (*float, optional*) – The opacity of the stroke color. The opacity should be a float between 0.0 (transparent) and 1.0 (opaque). 0.6 by default.

class gmaps.**Polygon**(*path*, *stroke_color='#696969'*, *stroke_weight=2.0*, *stroke_opacity=0.6*, *fill_color='#696969'*, *fill_opacity=0.2*)

Widget representing a closed polygon on a map

Add this polygon to a map via the `gmaps.drawing_layer()` function, or by passing it directly to the `.features` array of an existing instance of `gmaps.Drawing`.

Examples

```
>>> fig = gmaps.figure()
>>> drawing = gmaps.drawing_layer(features=[
     gmaps.Polygon(
        [(46.72, 6.06), (46.48, 6.49), (46.79, 6.91)],
        stroke_color='red', fill_color=(255, 0, 132)
    )
])
>>> fig.add_layer(drawing)
```

You can also add a polygon to an existing *gmaps.Drawing* instance:

```
>>> fig = gmaps.figure()
>>> drawing = gmaps.drawing_layer()
>>> fig.add_layer(drawing)
>>> fig # display the figure
```

You can now add polygons directly on the map:

```
>>> drawing.features = [
     gmaps.Polygon(
         [(46.72, 6.06), (46.48, 6.49), (46.79, 6.91)]
         stroke_color='red', fill_color=(255, 0, 132)
     )
]
```

Parameters

- **path** (*list of tuples of floats*) – List of (latitude, longitude) pairs denoting each point on the polygon. 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).
- **stroke_color** (*str or tuple, optional*) – The stroke color of the line. 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). Defaults to a grey color: (69, 69, 69)
- **stroke_weight** (*float, optional*) – How wide the line is. This is a positive float. Defaults to 2.
- **stroke_opacity** (*float, optional*) – The opacity of the stroke color. The opacity should be a float between 0.0 (transparent) and 1.0 (opaque). 0.6 by default.
- **fill_color** (*str or tuple, optional*) – The internal color of the polygon. 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). Defaults to a grey color: (69, 69, 69)
- **fill_opacity** (*float, optional*) – The opacity of the fill color. The opacity should be a float between 0.0 (transparent) and 1.0 (opaque). 0.2 by default.

class gmaps.**PolygonOptions**(**args*, ***kwargs*)

Style options for a polygon.

Pass an instance of this class to *gmaps.drawing_layer()* to control the style of new user-drawn polygons on the map.

Examples

```
>>> fig = gmaps.figure()
>>> drawing = gmaps.drawing_layer(
        polygon_options=gmaps.PolygonOptions(
            stroke_color='red', fill_color=(255, 0, 132))
    )
>>> fig.add_layer(drawing)
>>> fig # display the figure
```

Parameters

- **stroke_color** (*str or tuple, optional*) – The stroke color of the line. 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). Defaults to a grey color: (69, 69, 69)
- **stroke_weight** (*float, optional*) – How wide the line is. This is a positive float. Defaults to 2.
- **stroke_opacity** (*float, optional*) – The opacity of the stroke color. The opacity should be a float between 0.0 (transparent) and 1.0 (opaque). 0.6 by default.
- **fill_color** (*str or tuple, optional*) – The internal color of the polygon. 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). Defaults to a grey color: (69, 69, 69)
- **fill_opacity** (*float, optional*) – The opacity of the fill color. The opacity should be a float between 0.0 (transparent) and 1.0 (opaque). 0.2 by default.

6.4 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.

6.5 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
```

6.6 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

exception gmaps.geotraitlets.**InvalidPointException**

exception gmaps.geotraitlets.**InvalidWeightException**

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.**LocationArray**(*trait=None*, *default_value=None*, *minlen=0*, *maxlen=9223372036854775807*, ***kwargs*)

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.**MapType**(*default_value*, ***kwargs*)
String representing a map type

class gmaps.geotraitlets.**MouseHandling**(*default_value*, ***kwargs*)
String representing valid values for mouse handling

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

class gmaps.geotraitlets.**Point**(*default_value=traitlets.Undefined*)
Tuple representing a (latitude, longitude) pair.

class gmaps.geotraitlets.**RgbTuple**(***metadata*)

class gmaps.geotraitlets.**RgbaTuple**(***metadata*)

class gmaps.geotraitlets.**Tilt**(*default_value=traitlets.Undefined*, *allow_none=False*, ***kwargs*)
Integer representing a tilt degree allowed by Google Maps

class gmaps.geotraitlets.**WeightArray**(*trait=None*, *default_value=None*, *minlen=0*, *maxlen=9223372036854775807*, ***kwargs*)

default_value = traitlets.Undefined

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 7

Contributing to jupyter-gmaps

7.1 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.

7.1.1 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.

7.1.2 Installing a development version of gmaps

See the installation instructions for installing a development version.

7.1.3 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.

7.1.4 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.

7.2 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 8

Release notes

8.1 Version 0.8.2

This minor release fixes an issue where the fill color, stroke color or scale were not respected when embedding a map as HTML if they were exactly on the default value in the Python layer (PR 276).

8.2 Version 0.8.1 - 26th September 2018

This minor release:

- adds the tilt option to maps (PR 253).
- fixes passing fill and stroke opacity attributes to heatmap (PR 263).
- fixes an issue on JupyterLab where the color of output cells was changed to grey when built with jupyter-gmaps (PR 268).
- updates the release process to use twine (PR 269).

8.3 Version 0.8.0 - 22nd April 2018

This minor release:

- Changes the directions layer widget to add the *start*, *end* and *waypoints* traitlets. This deprecates the *data* traitlet, scheduled for removal in 0.9.0. (PR 236).
- The directions layer now reacts to changes in start, end and waypoints by re-calculating the route (PR 239)
- The directions layer now supports styling the route (PR 247)
- Errors in the direction layer are now shown in the error box, rather than as an uncatchable exception (PR 242)

- Errors authenticating result in an error message that replaces the map, rather than the cryptic 'Oops, something went wrong' default that Google Maps provides (PR 240)
- Adds style to error box (PR 243)
- Removes the deprecated *data* traitlet from the Heatmap and WeightedHeatmap widgets. See PR 249 for a migration pathway. (PR 249)
- Introduces the Opacity traitlet for encoding stroke and fill opacities (PR 248)

8.4 Version 0.7.4 - 5th April 2018

This minor release:

- allows setting which map type we use (PR 232)
- allows setting how the map interacts with the webpage, in terms of capturing scroll events (PR 232)
- allows setting the style of polygons on the drawing layer (PR 229)
- fixes a bug that stopped the drawing layer from being downloadable as a PNG (PR 227)

8.5 Version 0.7.3 - 11th March 2018

This release: - simplifies setting the width and height for a figure. We now do

not need to explicitly set the width and height of the embedded map (PR 221).

- allows customising the style of lines added to the map in the drawing layer (PR 225).

8.6 Version 0.7.2 - 16th February 2018

This release adds support for JupyterLab (PR 218).

8.7 Version 0.7.1 - 10th February 2018

This minor release:

- Deprecates the *.data* traitlet in heatmaps and weighted heatmaps in favour of *.locations* (for heatmap) and *.locations* and *.weights*. These now have validation, so a user can pass in a dataframe or numpy array (PR 211).
- React to changes in the new *.locations* and *.weights* traitlets to actually update heatmaps dynamically. (PR 212).
- Reduces page load size in documentation by compressing the images (PR 217).

8.8 Version 0.7.0 - 11th November 2017

- This minor release adds a drawing layer, giving the user the ability to add arbitrary lines, markers and polygons to a map. The developer can bind callbacks that are run when a feature is added, allowing the development of complex, widgets- based application on top of jupyter-gmaps (PR 183). - It fixes a bug where the bounds were incorrectly calculated when two longitudes coincided (PR 204). - It fixes a bug where, for single latitudes, the returned bounds could stretch beyond what Google Maps allows (PR 204)

8.9 Version 0.6.2 - 30th October 2017

This minor release:

- fixes a bug that was introduced by shadowing a reserved traitlets method (PR 184)
- migrates the codebase to flake8 3.5.0 (PR 195)

8.10 Version 0.6.1 - 1st September 2017

This is a patch release that is identical to 0.6.0. The dependencies in the conda-forge release of 0.6.0 were badly specified.

8.11 Version 0.6.0 - 26th August 2017

This release:

- PRs 166, 171 and 172 migrate jupyter-gmaps to ipywidgets 7.0.0 (released on the 18th August 2017). This is a breaking change: jupyter-gmaps will not work with ipywidgets 6.x versions.
- PRs 163 and 169 add a layer for displaying bicycling information.
- PRs 165 and 169 add a layer for displaying transit (public transport) information.
- PR 170 adds a layer for displaying traffic information.
- PR 173 improves the layout of the CSS
- PR 173 improves the CSS used for embedding

8.12 Version 0.5.4 - 15th July 2017

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).

8.13 Version 0.5.3 - 8th July 2017

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.

8.14 Version 0.5.2 - 25th June 2017

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.

8.15 Version 0.5.1 - 3rd June 2017

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.

8.16 Version 0.5.0 - 8th May 2017

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.

8.17 Version 0.4.1 - 14th March 2017

- 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).

8.18 Version 0.4.0 - 28th January 2017

- **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).

8.19 Version 0.3.6 - 28th December 2016

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

8.20 Version 0.3.5 - 8th October 2016

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

8.21 Version 0.3.4 - 26th September 2016

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

8.22 Version 0.3.3 - 7th September 2016

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

8.23 Version 0.3.2 - 30th July 2016

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

8.24 Version 0.3.1 - 30th July 2016

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.

8.25 Version 0.3.0 - 14th June 2016

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)

8.26 Version 0.2.2 - 26th March 2016

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

8.27 Version 0.2.1 - 26th March 2016

test release – no changes.

8.28 Version 0.2 - 2nd January 2016

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

8.29 Version 0.1.6 - 8th December 2014

Fixed typo in setup script.

8.30 Version 0.1.5 - 8th December 2014

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.

8.31 Version 0.1.4 - 4th December 2014

Another bugfix release.

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

8.32 Version 0.1.3 - 4th December 2014

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.

8.33 Version 0.1.2 - 4th December 2014

Minor heatmap improvements.

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

8.34 Version 0.1.1 - 2nd December 2014

Bugfix release.

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

8.35 Version 0.1 - 2nd December 2014

Initial release.

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

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index