API reference

Map

Make beautiful, interactive maps with Python and Leaflet.js

class **folium.folium.GlobalSwitches**(*no_touch=False*, *disable_3d=False*)

Bases: `Element`

class **folium.folium.Map**(*location: Sequence[float] | None = None*, *width: str | float = '100%'*, *height: str | float = '100%'*, *left: str | float = '0%'*, *top: str | float = '0%'*, *position: str = 'relative'*, *tiles: str | TileLayer | None = 'OpenStreetMap'*, *attr: str | None = None*, *min_zoom: int | None = None*, *max_zoom: int | None = None*, *zoom_start: int = 10*, *min_lat: float = -90*, *max_lat: float = 90*, *min_lon: float = -180*, *max_lon: float = 180*, *max_bounds: bool = False*, *crs: str = 'EPSG3857'*, *control_scale: bool = False*, *prefer_canvas: bool = False*, *no_touch: bool = False*, *disable_3d: bool = False*, *png_enabled: bool = False*, *zoom_control: bool | str = True*, *font_size: str = '1rem'*, ***kwargs: str | float | bool | Sequence | dict | None*)

Bases: `JSCSSMixin`, `Evented`

Create a Map with Folium and Leaflet.js

Generate a base map of given width and height with either default tilesets or a custom tileset URL. Folium has built-in all tilesets available in the `xyzservices` package. For example, you can pass any of the following to the "tiles" keyword:

> - "OpenStreetMap"
> - "CartoDB Positron"
> - "CartoDB Voyager"

Explore more provider names available in `xyzservices` here: https://leaflet-extras.github.io/leaflet-providers/preview/.

You can also pass a custom tileset by passing a `xyzservices.TileProvider` or a Leaflet-style URL to the tiles parameter: `https://{s}.yourtiles.com/{z}/{x}/{y}.png`.

↑ Back to top

Parameters:

- **location** (*tuple or list, default None*) – Latitude and Longitude of Map (Northing, Easting).
- **width** (*pixel int or percentage string (default: '100%')*) – Width of the map.
- **height** (*pixel int or percentage string (default: '100%')*) – Height of the map.
- **tiles** (str or TileLayer or `xyzservices.TileProvider`, default 'OpenStreetMap') – Map tileset to use. Can choose from a list of built-in tiles, pass a `xyzservices.TileProvider`, pass a custom URL, pass a TileLayer object, or pass *None* to create a map without tiles. For more advanced tile layer options, use the *TileLayer* class.
- **min_zoom** (*int, optional, default 0*) – Minimum allowed zoom level for the tile layer that is created. Filled by xyzservices by default.
- **max_zoom** (*int, optional, default 18*) – Maximum allowed zoom level for the tile layer that is created. Filled by xyzservices by default.
- **zoom_start** (*int, default 10*) – Initial zoom level for the map.
- **attr** (*string, default None*) – Map tile attribution; only required if passing custom tile URL.
- **crs** (*str, default 'EPSG3857'*) – Defines coordinate reference systems for projecting geographical points into pixel (screen) coordinates and back. You can use Leaflet's values : * EPSG3857 : The most common CRS for online maps, used by almost all free and commercial tile providers. Uses Spherical Mercator projection. Set in by default in Map's crs option. * EPSG4326 : A common CRS among GIS enthusiasts. Uses simple Equirectangular projection. * EPSG3395 : Rarely used by some commercial tile providers. Uses Elliptical Mercator projection. * Simple : A simple CRS that maps longitude and latitude into x and y directly. May be used for maps of flat surfaces (e.g. game maps). Note that the y axis should still be inverted (going from bottom to top).
- **control_scale** (*bool, default False*) – Whether to add a control scale on the map.
- **prefer_canvas** (*bool, default False*) – Forces Leaflet to use the Canvas back-end (if available) for vector layers instead of SVG. This can increase performance considerably in some cases (e.g. many thousands of circle markers on the map).
- **no_touch** (*bool, default False*) – Forces Leaflet to not use touch events even if it detects them.
- **disable_3d** (*bool, default False*) – Forces Leaflet to not use hardware-accelerated CSS 3D transforms for positioning (which may cause glitches in some rare environments) even if they're supported.
- **zoom_control** (*bool or position string, default True*) – Display zoom controls on the map. The default *True* places it in the top left corner. Other options are 'topleft', 'topright', 'bottomleft' or 'bottomright'.
- **font_size** (*int or float or string (default: '1rem')*) – The font size to use for Leaflet, can either be a number or a string ending in 'rem', 'em', or 'px'.
- ****kwargs** – Additional keyword arguments are passed to Leaflets Map class: https://leafletjs.com/reference.html#map

Return type:

Folium Map Object

Examples

```
>>> m = folium.Map(location=[45.523, -122.675], width=750, height=500)
>>> m = folium.Map(location=[45.523, -122.675], tiles="cartodb positron")
>>> m = folium.Map(
...     location=[45.523, -122.675],
...     zoom_start=2,
...     tiles="https://api.mapbox.com/v4/mapbox.streets/{z}/{x}/{y}.png?access_token=mytoken",
...     attr="Mapbox attribution"
```

```
... )
```

default_css: *List[Tuple[str, str]]*

default_js: *List[Tuple[str, str]]*

fit_bounds(*bounds: Sequence[Sequence[float]]*, *padding_top_left: Sequence[float] | None = None*, *padding_bottom_right: Sequence[float] | None = None*, *padding: Sequence[float] | None = None*, *max_zoom: int | None = None*) → None

Fit the map to contain a bounding box with the maximum zoom level possible.

Parameters:

- **bounds** (*list of (latitude, longitude) points*) – Bounding box specified as two points [southwest, northeast]
- **padding_top_left** (*(x, y) point, default None*) – Padding in the top left corner. Useful if some elements in the corner, such as controls, might obscure objects you're zooming to.
- **padding_bottom_right** (*(x, y) point, default None*) – Padding in the bottom right corner.
- **padding** (*(x, y) point, default None*) – Equivalent to setting both top left and bottom right padding to the same value.
- **max_zoom** (*int, default None*) – Maximum zoom to be used.

Examples

```
>>> m.fit_bounds([[52.193636, -2.221575], [52.636878, -1.139759]])
```

keep_in_front(**args: Layer*) → None

Pass one or multiple layers that must stay in front.

The ordering matters, the last one is put on top.

Parameters:

***args** – Variable length argument list. Any folium object that counts as an overlay. For example FeatureGroup or TileLayer. Does not work with markers, for those use z_index_offset.

↑ Back to top

render(***kwargs*) → None

Renders the HTML representation of the element.

show_in_browser() → None

Display the Map in the default web browser.

UI elements

Classes for drawing maps.

class folium.map.**CustomPane**(*name: str*, *z_index: int | str = 625*, *pointer_events: bool = False*)

Bases: MacroElement

Creates a custom pane to hold map elements.

Behavior is as in https://leafletjs.com/examples/map-panes/

Parameters:

- **name** (*string*) – Name of the custom pane. Other map elements can be added to the pane by specifying the 'pane' kwarg when constructing them.
- **z_index** (*int or string, default 625*) – The z-index that will be associated with the pane, and will determine which map elements lie over/under it. The default (625) corresponds to between markers and tooltips. Default panes and z-indexes can be found at https://leafletjs.com/reference.html#map-pane
- **pointer_events** (*bool, default False*) – Whether or not layers in the pane should interact with the cursor. Setting to False will prevent interfering with pointer events associated with lower layers.

class folium.map.**Evented**

Bases: MacroElement

The base class for Layer and Map

Adds the *on* method for event handling capabilities.

See https://leafletjs.com/reference.html#evented for more in depth documentation. Please note that we have only added the *on(<Object> eventMap)* variant of this method using python keyword arguments.

on(***event_map: JsCode*)

class folium.map.**FeatureGroup**(*name: str | None = None*, *overlay: bool = True*, *control: bool = True*, *show:*

*bool = True, **kwargs: str | float | bool | Sequence | dict | None)*

Bases: `Layer`

Create a FeatureGroup layer ; you can put things in it and handle them as a single layer. For example, you can add a LayerControl to tick/untick the whole group.

Parameters:

- **name** (*str, default None*) – The name of the featureGroup layer. It will be displayed in the LayerControl. If None get_name() will be called to get the technical (ugly) name.
- **overlay** (*bool, default True*) – Whether your layer will be an overlay (ticked with a check box in LayerControls) or a base layer (ticked with a radio button).
- **control** (*bool, default True*) – Whether the layer will be included in LayerControls.
- **show** (*bool, default True*) – Whether the layer will be shown on opening.
- ****kwargs** – Additional (possibly inherited) options. See https://leafletjs.com/reference.html#featuregroup

class folium.map.**FitBounds**(*bounds: Sequence[Sequence[float]], padding_top_left: Sequence[float] | None = None, padding_bottom_right: Sequence[float] | None = None, padding: Sequence[float] | None = None, max_zoom: int | None = None*)

Bases: `MacroElement`

Fit the map to contain a bounding box with the maximum zoom level possible.

Parameters:

- **bounds** (*list of (latitude, longitude) points*) – Bounding box specified as two points [southwest, northeast]
- **padding_top_left** (*(x, y) point, default None*) – Padding in the top left corner. Useful if some elements in the corner, such as controls, might obscure objects you're zooming to.
- **padding_bottom_right** (*(x, y) point, default None*) – Padding in the bottom right corner.
- **padding** (*(x, y) point, default None*) – Equivalent to setting both top left and bottom right padding to the same value.
- **max_zoom** (*int, default None*) – Maximum zoom to be used.

class folium.map.**FitOverlays**(*padding: int = 0, max_zoom: int | None = None, fly: bool = False, fit_on_map_load: bool = True*)

Bases: `MacroElement`

↑ Back to top

Fit the bounds of the maps to the enabled overlays.

Parameters:

- **padding** (*int, default 0*) – Amount of padding in pixels applied in the corners.
- **max_zoom** (*int, optional*) – The maximum possible zoom to use when fitting to the bounds.
- **fly** (*bool, default False*) – Use a smoother, longer animation.
- **fit_on_map_load** (*bool, default True*) – Apply the fit when initially loading the map.

class folium.map.**Icon**(*color: str = 'blue', icon_color: str = 'white', icon: str = 'info-sign', angle: int = 0, prefix: str = 'glyphicon', **kwargs: str | float | bool | Sequence | dict | None*)

Bases: `MacroElement`

Creates an Icon object that will be rendered using Leaflet.awesome-markers.

Parameters:

- **color** (*str, default 'blue'*) –

  The color of the marker. You can use:

  > **['red', 'blue', 'green', 'purple', 'orange', 'darkred',**
  > 'lightred', 'beige', 'darkblue', 'darkgreen', 'cadetblue', 'darkpurple', 'white', 'pink', 'lightblue', 'lightgreen', 'gray', 'black', 'lightgray']

- **icon_color** (*str, default 'white'*) – The color of the drawing on the marker. You can use colors above, or an html color code.
- **icon** (*str, default 'info-sign'*) – The name of the marker sign. See Font-Awesome website to choose yours. Warning : depending on the icon you choose you may need to adapt the *prefix* as well.
- **angle** (*int, default 0*) – The icon will be rotated by this amount of degrees.
- **prefix** (*str, default 'glyphicon'*) – The prefix states the source of the icon. 'fa' for font-awesome or 'glyphicon' for bootstrap 3.
- **https** (*//github.com/lvoogdt/Leaflet.awesome-markers*)

`color_options`

class folium.map.**Layer**(*name: str | None = None, overlay: bool = False, control: bool = True, show: bool = True*)

Bases: `Evented`

An abstract class for everything that is a Layer on the map. It will be used to define whether an object will be included in LayerControls.

Parameters:

- **name** (*string, default None*) – The name of the Layer, as it will appear in LayerControls
- **overlay** (*bool, default False*) – Adds the layer as an optional overlay (True) or the base layer (False).
- **control** (*bool, default True*) – Whether the Layer will be included in LayerControls.
- **show** (*bool, default True*) – Whether the layer will be shown on opening.

`render(**kwargs)`

Renders the HTML representation of the element.

`class folium.map.LayerControl(position: str = 'topright', collapsed: bool = True, autoZIndex: bool = True, draggable: bool = False, **kwargs: str | float | bool | Sequence | dict | None)`

Bases: `MacroElement`

Creates a LayerControl object to be added on a folium map.

This object should be added to a Map object. Only Layer children of Map are included in the layer control.

> **Note**
>
> The LayerControl should be added last to the map. Otherwise, the LayerControl and/or the controlled layers may not appear.

Parameters:

- **position** (*str*) – The position of the control (one of the map corners), can be 'topleft', 'topright', 'bottomleft' or 'bottomright' default: 'topright'
- **collapsed** (*bool, default True*) – If true the control will be collapsed into an icon and expanded on mouse hover or touch.
- **autoZIndex** (*bool, default True*) – If true the control assigns zIndexes in increasing order to all of its layers so that the order is preserved when switching them on/off.
- **draggable** (*bool, default False*) – By default the layer control has a fixed position. Set this argument to True to allow dragging the control around.
- ****kwargs** – Additional (possibly inherited) options. See https://leafletjs.com/reference.html#control-layers

`render(**kwargs) → None`

↑ Back to top

s the HTML representation of the element.

`reset() → None`

`class folium.map.Marker(location: Sequence[float] | None = None, popup: Popup | str | None = None, tooltip: Tooltip | str | None = None, icon: Icon | None = None, draggable: bool = False, **kwargs: str | float | bool | Sequence | dict | None)`

Bases: `MacroElement`

Create a simple stock Leaflet marker on the map, with optional popup text or Vincent visualization.

Parameters:

- **location** (*tuple or list*) – Latitude and Longitude of Marker (Northing, Easting)
- **popup** (*string or folium.Popup, default None*) – Label for the Marker; either an escaped HTML string to initialize folium.Popup or a folium.Popup instance.
- **tooltip** (*str or folium.Tooltip, default None*) – Display a text when hovering over the object.
- **icon** (*Icon plugin*) – the Icon plugin to use to render the marker.
- **draggable** (*bool, default False*) – Set to True to be able to drag the marker around the map.

Return type:

Marker names and HTML in obj.template_vars

Examples

```
>>> Marker(location=[45.5, -122.3], popup="Portland, OR")
>>> Marker(location=[45.5, -122.3], popup=Popup("Portland, OR"))
# If the popup label has characters that need to be escaped in HTML
>>> Marker(
...     location=[45.5, -122.3],
...     popup=Popup("Mom & Pop Arrow Shop >>", parse_html=True),
... )
```

`render() → None`

Renders the HTML representation of the element.

`class folium.map.Popup(html: str | Element | None = None, parse_html: bool = False, max_width: str | int = '100%', show: bool = False, sticky: bool = False, lazy: bool = False, **kwargs: str | float | bool | Sequence | dict | None)`

Bases: `Element`

Create a Popup instance that can be linked to a Layer.

Parameters:

- **html** (*string or Element*) – Content of the Popup.
- **parse_html** (*bool, default False*) – True if the popup is a template that needs to the rendered first.
- **max_width** (*int for pixels or text for percentages, default '100%'*) – The maximal width of the popup.
- **show** (*bool, default False*) – True renders the popup open on page load.
- **sticky** (*bool, default False*) – True prevents map and other popup clicks from closing.
- **lazy** (*bool, default False*) – True only loads the Popup content when clicking on the Marker.

`render(**kwargs) → None`

Renders the HTML representation of the element.

`class folium.map.Tooltip(text: str, style: str | None = None, sticky: bool = True, **kwargs: str | float | bool | Sequence | dict | None)`

Bases: `MacroElement`

Create a tooltip that shows text when hovering over its parent object.

Parameters:

- **text** (*str*) – String to display as a tooltip on the object. If the argument is of a different type it will be converted to str.
- **style** (*str, default None.*) – HTML inline style properties like font and colors. Will be applied to a div with the text in it.
- **sticky** (*bool, default True*) – Whether the tooltip should follow the mouse.
- ****kwargs** – These values will map directly to the Leaflet Options. More info available here: https://leafletjs.com/reference.html#tooltip

`parse_options(kwargs: Dict[str, str | float | bool | Sequence | dict | None]) → Dict[str, str | float | bool | Sequence | dict | None]`

Validate the provided kwargs and return options as json string.

`valid_options: Dict[str, Tuple[Type, ...]]`

Raster Layers

Wraps leaflet TileLayer, WmsTileLayer (TileLayer.WMS), ImageOverlay, and VideoOverlay

↑ Back to top

`class folium.raster_layers.ImageOverlay(image: Any, bounds: Sequence[Sequence[float]], origin: str = 'upper', colormap: Callable | None = None, mercator_project: bool = False, pixelated: bool = True, name: str | None = None, overlay: bool = True, control: bool = True, show: bool = True, **kwargs)`

Bases: `Layer`

Used to load and display a single image over specific bounds of the map, implements ILayer interface.

Parameters:

- **image** (*string, file or array-like object*) – The data you want to draw on the map. * If string, it will be written directly in the output file. * If file, it's content will be converted as embedded in the output file. * If array-like, it will be converted to PNG base64 string and embedded in the output.
- **bounds** (*list*) –

  Image bounds on the map in the form

  [[lat_min, lon_min], [lat_max, lon_max]]
- **opacity** (*float, default Leaflet's default (1.0)*)
- **alt** (*string, default Leaflet's default ('')*)
- **origin** (*['upper' | 'lower'], optional, default 'upper'*) – Place the [0,0] index of the array in the upper left or lower left corner of the axes.
- **colormap** (callable, used only for *mono* image.) – Function of the form [x -> (r,g,b)] or [x -> (r,g,b,a)] for transforming a mono image into RGB. It must output iterables of length 3 or 4, with values between 0 and 1. Hint: you can use colormaps from *matplotlib.cm*.
- **mercator_project** (*bool, default False.*) – Used only for array-like image. Transforms the data to project (longitude, latitude) coordinates to the Mercator projection. Beware that this will only work if *image* is an array-like object. Note that if used the image will be clipped beyond latitude -85 and 85.
- **pixelated** (*bool, default True*) – Sharp sharp/crips (True) or aliased corners (False).
- **name** (*string, default None*) – The name of the Layer, as it will appear in LayerControls
- **overlay** (*bool, default True*) – Adds the layer as an optional overlay (True) or the base layer (False).
- **control** (*bool, default True*) – Whether the Layer will be included in LayerControls.
- **show** (*bool, default True*) – Whether the layer will be shown on opening.
- **https** (*See*)
- **options.**

`render(**kwargs) → None`

Renders the HTML representation of the element.

`class folium.raster_layers.TileLayer(tiles: str | TileProvider = 'OpenStreetMap', min_zoom: int | None = None, max_zoom: int | None = None, max_native_zoom: int | None = None, attr: str | None = None, detect_retina: bool = False, name: str | None = None, overlay: bool = False, control: bool = True, show: bool = True, no_wrap: bool = False, subdomains: str = 'abc', tms: bool = False, opacity: float = 1,`

**kwargs)

Bases: Layer

Create a tile layer to append on a Map.

Parameters:

- **tiles** (str or xyzservices.TileProvider, default 'OpenStreetMap') –

  Map tileset to use. Folium has built-in all tilesets available in the xyzservices package. For example, you can pass any of the following to the "tiles" keyword:

  - "OpenStreetMap"
  - "CartoDB Positron"
  - "CartoDB Voyager"

  Explore more provider names available in xyzservices here: https://leaflet-extras.github.io/leaflet-providers/preview/. You can also pass a custom tileset by passing a xyzservices.TileProvider or a Leaflet-style URL to the tiles parameter: https://{s}.yourtiles.com/{z}/{x}/{y}.png.
- **min_zoom** (*int, optional, default 0*) – Minimum allowed zoom level for this tile layer. Filled by xyzservices by default.
- **max_zoom** (*int, optional, default 18*) – Maximum allowed zoom level for this tile layer. Filled by xyzservices by default.
- **max_native_zoom** (*int, default None*) – The highest zoom level at which the tile server can provide tiles. Filled by xyzservices by default. By setting max_zoom higher than max_native_zoom, you can zoom in past max_native_zoom, tiles will be autoscaled.
- **attr** (*string, default None*) – Map tile attribution; only required if passing custom tile URL.
- **detect_retina** (*bool, default False*) – If true and user is on a retina display, it will request four tiles of half the specified size and a bigger zoom level in place of one to utilize the high resolution.
- **name** (*string, default None*) – The name of the Layer, as it will appear in LayerControls
- **overlay** (*bool, default False*) – Adds the layer as an optional overlay (True) or the base layer (False).
- **control** (*bool, default True*) – Whether the Layer will be included in LayerControls.
- **show** (*bool, default True*) – Whether the layer will be shown on opening. When adding multiple base layers, use this parameter to select which one should be shown when opening the map, by not showing the others.
- **subdomains** (*list of strings, default ['abc']*) – Subdomains of the tile service.
- **tms** (*bool, default False*) – If true, inverses Y axis numbering for tiles (turn this on for TMS services).
- **opacity** (*float, default 1*) – Sets the opacity for the layer.
- ****kwargs** (*additional keyword arguments*) – Other keyword arguments are passed as options to the Leaflet tileLayer object.

↑ Back to top

class folium.raster_layers.VideoOverlay(*video_url: str, bounds: Sequence[Sequence[float]], autoplay: bool = True, loop: bool = True, name: str | None = None, overlay: bool = True, control: bool = True, show: bool = True, **kwargs: str | float | bool | Sequence | dict | None*)

Bases: Layer

Used to load and display a video over the map.

Parameters:

- **video_url** (*str*) – URL of the video
- **bounds** (*list*) –

  Video bounds on the map in the form

  [[lat_min, lon_min], [lat_max, lon_max]]
- **autoplay** (*bool, default True*)
- **loop** (*bool, default True*)
- **name** (*string, default None*) – The name of the Layer, as it will appear in LayerControls
- **overlay** (*bool, default True*) – Adds the layer as an optional overlay (True) or the base layer (False).
- **control** (*bool, default True*) – Whether the Layer will be included in LayerControls.
- **show** (*bool, default True*) – Whether the layer will be shown on opening.
- ****kwargs** – Other valid (possibly inherited) options. See: https://leafletjs.com/reference.html#videooverlay

class folium.raster_layers.WmsTileLayer(*url: str, layers: str, styles: str = '', fmt: str = 'image/jpeg', transparent: bool = False, version: str = '1.1.1', attr: str = '', name: str | None = None, overlay: bool = True, control: bool = True, show: bool = True, **kwargs*)

Bases: Layer

Creates a Web Map Service (WMS) layer.

Parameters:

- **url** (*str*) – The url of the WMS server.
- **layers** (*str*) – Comma-separated list of WMS layers to show.
- **styles** (*str, optional*) – Comma-separated list of WMS styles.
- **fmt** (*str, default 'image/jpeg'*) – The format of the service output. Ex: 'image/png'
- **transparent** (*bool, default False*) – Whether the layer shall allow transparency.
- **version** (*str, default '1.1.1'*) – Version of the WMS service to use.
- **attr** (*str, default ''*) – The attribution of the service. Will be displayed in the bottom right corner.
- **name** (*string, optional*) – The name of the Layer, as it will appear in LayerControls
- **overlay** (*bool, default True*) – Adds the layer as an optional overlay (True) or the base layer (False).

- **control** (*bool, default True*) – Whether the Layer will be included in LayerControls.
- **show** (*bool, default True*) – Whether the layer will be shown on opening.
- ****kwargs** (*additional keyword arguments*) – Passed through to the underlying tileLayer.wms object and can be used for setting extra tileLayer.wms parameters or as extra parameters in the WMS request.
- **https** (*See*)

Vector Layers

Wraps leaflet Polyline, Polygon, Rectangle, Circle, and CircleMarker

class folium.vector_layers.**Circle**(*location: Sequence[float] | None = None*, *radius: float = 50*, *popup: Popup | str | None = None*, *tooltip: Tooltip | str | None = None*, ***kwargs: bool | str | float | None*)

Bases: `Marker`

Class for drawing circle overlays on a map.

It's an approximation and starts to diverge from a real circle closer to the poles (due to projection distortion).

See `folium.vector_layers.path_options()` for the *Path* options.

Parameters:

- **location** (*tuple[float, float]*) – Latitude and Longitude pair (Northing, Easting)
- **popup** (*string or folium.Popup, default None*) – Input text or visualization for object displayed when clicking.
- **tooltip** (*str or folium.Tooltip, default None*) – Display a text when hovering over the object.
- **radius** (*float*) – Radius of the circle, in meters.
- ****kwargs** – Other valid (possibly inherited) options. See: https://leafletjs.com/reference.html#circle

class folium.vector_layers.**CircleMarker**(*location: Sequence[float] | None = None*, *radius: float = 10*, *popup: Popup | str | None = None*, *tooltip: Tooltip | str | None = None*, ***kwargs: bool | str | float | None*)

Bases: `Marker`

A circle of a fixed size with radius specified in pixels.

↑ Back to top

`vector_layers.path_options()` for the *Path* options.

Parameters:

- **location** (*tuple[float, float]*) – Latitude and Longitude pair (Northing, Easting)
- **popup** (*string or folium.Popup, default None*) – Input text or visualization for object displayed when clicking.
- **tooltip** (*str or folium.Tooltip, default None*) – Display a text when hovering over the object.
- **radius** (*float, default 10*) – Radius of the circle marker, in pixels.
- ****kwargs** – Other valid (possibly inherited) options. See: https://leafletjs.com/reference.html#circlemarker

class folium.vector_layers.**PolyLine**(*locations*, *popup=None*, *tooltip=None*, ***kwargs*)

Bases: `BaseMultiLocation`

Draw polyline overlays on a map.

See `folium.vector_layers.path_options()` for the *Path* options.

Parameters:

- **locations** (*list of points (latitude, longitude)*) – Latitude and Longitude of line (Northing, Easting) Pass multiple sequences of coordinates for a multi-polyline.
- **popup** (*str or folium.Popup, default None*) – Input text or visualization for object displayed when clicking.
- **tooltip** (*str or folium.Tooltip, default None*) – Display a text when hovering over the object.
- **smooth_factor** (*float, default 1.0*) – How much to simplify the polyline on each zoom level. More means better performance and smoother look, and less means more accurate representation.
- **no_clip** (*Bool, default False*) – Disable polyline clipping.
- ****kwargs** – Other valid (possibly inherited) options. See: https://leafletjs.com/reference.html#polyline

class folium.vector_layers.**Polygon**(*locations: Iterable[Sequence[float]] | Iterable[Iterable[Sequence[float]]]*, *popup: Popup | str | None = None*, *tooltip: Tooltip | str | None = None*, ***kwargs: bool | str | float | None*)

Bases: `BaseMultiLocation`

Draw polygon overlays on a map.

See `folium.vector_layers.path_options()` for the *Path* options.

Parameters:

- **locations** (*list of points (latitude, longitude)*) –
  - One list of coordinate pairs to define a polygon. You don't have to add a last point equal to the first point.
  - If you pass a list with multiple of those it will make a multi- polygon.
- **popup** (*string or folium.Popup, default None*) – Input text or visualization for object displayed when clicking.
- **tooltip** (*str or folium.Tooltip, default None*) – Display a text when hovering over the object.

- ****kwargs** – Other valid (possibly inherited) options. See: https://leafletjs.com/reference.html#polygon

class folium.vector_layers.**Rectangle**(*bounds: Iterable[Sequence[float]]*, *popup: Popup | str | None = None*, *tooltip: Tooltip | str | None = None*, ***kwargs: bool | str | float | None*)

Bases: `MacroElement`

Draw rectangle overlays on a map.

See `folium.vector_layers.path_options()` for the *Path* options.

Parameters:

- **bounds** (*[(lat1, lon1), (lat2, lon2)]*) – Two lat lon pairs marking the two corners of the rectangle.
- **popup** (*string or folium.Popup, default None*) – Input text or visualization for object displayed when clicking.
- **tooltip** (*str or folium.Tooltip, default None*) – Display a text when hovering over the object.
- ****kwargs** – Other valid (possibly inherited) options. See: https://leafletjs.com/reference.html#rectangle

folium.vector_layers.**path_options**(*line: bool = False*, *radius: float | None = None*, ***kwargs: bool | str | float | None*)

Contains options and constants shared between vector overlays (Polygon, Polyline, Circle, CircleMarker, and Rectangle).

Parameters:

- **stroke** (*Bool, True*) – Whether to draw stroke along the path. Set it to false to disable borders on polygons or circles.
- **color** (*str, '#3388ff'*) – Stroke color.
- **weight** (*int, 3*) – Stroke width in pixels.
- **opacity** (*float, 1.0*) – Stroke opacity.
- **line_cap** (*str, 'round' (lineCap)*) – A string that defines shape to be used at the end of the stroke. https://developer.mozilla.org/en-US/docs/Web/SVG/Attribute/stroke-linecap
- **line_join** (*str, 'round' (lineJoin)*) – A string that defines shape to be used at the corners of the stroke. https://developer.mozilla.org/en-US/docs/Web/SVG/Attribute/stroke-linejoin
- **dash_array** (*str, None (dashArray)*) – A string that defines the stroke dash pattern. Doesn't work on Canvas-powered layers in some old browsers. https://developer.mozilla.org/en-US/docs/Web/SVG/Attribute/stroke-dasharray
- **dash_offset** (*, str, None (dashOffset)*) – A string that defines the distance into the dash pattern to start the dash. Doesn't work on Canvas-powered layers in some old browsers. https://developer.mozilla.org/en-US/docs/Web/SVG/Attribute/stroke-dashoffset
- **fill** (*Bool, False*) – Whether to fill the path with color. Set it to false to disable filling on polygons or circles.
- **fill_color** (str, default to *color* (fillColor)) – Fill color. Defaults to the value of the color option.
- **fill_opacity** (*float, 0.2 (fillOpacity)*) – Fill opacity.
- **fill_rule** (*str, 'evenodd' (fillRule)*) – A string that defines how the inside of a shape is determined. https://developer.mozilla.org/en-US/docs/Web/SVG/Attribute/fill-rule
- **bubbling_mouse_events** (*Bool, True (bubblingMouseEvents)*) – When true a mouse event on this path will trigger the same event on the map (unless L.DomEvent.stopPropagation is used).
- **gradient** (*bool, default None*) – When a gradient on the stroke and fill is available, allows turning it on or off.
- **fill=False.** (*Note that the presence of fill_color will override*)
- **equivalents.** (*This function accepts both snake_case and lowerCamelCase*)
- **https** (*See*)

↑ Back to top

Other map features

Leaflet GeoJson and miscellaneous features.

class folium.features.**Choropleth**(*geo_data: Any*, *data: Any | None = None*, *columns: Sequence[Any] | None = None*, *key_on: str | None = None*, *bins: int | Sequence[float] = 6*, *fill_color: str | None = None*, *nan_fill_color: str = 'black'*, *fill_opacity: float = 0.6*, *nan_fill_opacity: float | None = None*, *line_color: str = 'black'*, *line_weight: float = 1*, *line_opacity: float = 1*, *name: str | None = None*, *legend_name: str = ''*, *overlay: bool = True*, *control: bool = True*, *show: bool = True*, *topojson: str | None = None*, *smooth_factor: float | None = None*, *highlight: bool = False*, *use_jenks: bool = False*, ***kwargs*)

Bases: `FeatureGroup`

Apply a GeoJSON overlay to the map.

Plot a GeoJSON overlay on the base map. There is no requirement to bind data (passing just a GeoJSON plots a single-color overlay), but there is a data binding option to map your columnar data to different feature objects with a color scale.

If data is passed as a Pandas DataFrame, the "columns" and "key-on" keywords must be included, the first to indicate which DataFrame columns to use, the second to indicate the layer in the GeoJSON on which to key the data. The 'columns' keyword does not need to be passed for a Pandas series.

Colors are generated from color brewer (https://colorbrewer2.org/) sequential palettes. By default, linear binning is used between the min and the max of the values. Custom binning can be achieved with the *bins* parameter.

TopoJSONs can be passed as "geo_data", but the "topojson" keyword must also be passed with the reference to the topojson objects to convert. See the topojson.feature method in the TopoJSON API reference: topojson/topojson

Parameters:

- **geo_data** (*string/object*) – URL, file path, or data (json, dict, geopandas, etc) to your GeoJSON geometries

- **data** (*Pandas DataFrame or Series, default None*) – Data to bind to the GeoJSON.
- **columns** (*tuple with two values, default None*) – If the data is a Pandas DataFrame, the columns of data to be bound. Must pass column 1 as the key, and column 2 the values.
- **key_on** (*string, default None*) – Variable in the *geo_data* GeoJSON file to bind the data to. Must start with 'feature' and be in JavaScript objection notation. Ex: 'feature.id' or 'feature.properties.statename'.
- **bins** (*int or sequence of scalars or str, default 6*) – If *bins* is an int, it defines the number of equal-width bins between the min and the max of the values. If *bins* is a sequence, it directly defines the bin edges. For more information on this parameter, have a look at numpy.histogram function.
- **fill_color** (*string, optional*) – Area fill color, defaults to blue. Can pass a hex code, color name, or if you are binding data, one of the following color brewer palettes: 'BuGn', 'BuPu', 'GnBu', 'OrRd', 'PuBu', 'PuBuGn', 'PuRd', 'RdPu', 'YlGn', 'YlGnBu', 'YlOrBr', and 'YlOrRd'.
- **nan_fill_color** (*string, default 'black'*) – Area fill color for nan or missing values. Can pass a hex code, color name.
- **fill_opacity** (*float, default 0.6*) – Area fill opacity, range 0-1.
- **nan_fill_opacity** (*float, default fill_opacity*) – Area fill opacity for nan or missing values, range 0-1.
- **line_color** (*string, default 'black'*) – GeoJSON geopath line color.
- **line_weight** (*int, default 1*) – GeoJSON geopath line weight.
- **line_opacity** (*float, default 1*) – GeoJSON geopath line opacity, range 0-1.
- **legend_name** (*string, default empty string*) – Title for data legend.
- **topojson** (*string, default None*) – If using a TopoJSON, passing "objects.yourfeature" to the topojson keyword argument will enable conversion to GeoJSON.
- **smooth_factor** (*float, default None*) – How much to simplify the polyline on each zoom level. More means better performance and smoother look, and less means more accurate representation. Leaflet defaults to 1.0.
- **highlight** (*boolean, default False*) – Enable highlight functionality when hovering over a GeoJSON area.
- **use_jenks** (*bool, default False*) – Use jenkspy to calculate bins using "natural breaks" (Fisher-Jenks algorithm). This is useful when your data is unevenly distributed.
- **name** (*string, optional*) – The name of the layer, as it will appear in LayerControls
- **overlay** (*bool, default True*) – Adds the layer as an optional overlay (True) or the base layer (False).
- **control** (*bool, default True*) – Whether the Layer will be included in LayerControls.
- **show** (*bool, default True*) – Whether the layer will be shown on opening.

Return type:

GeoJSON data layer in obj.template_vars

Examples

↑ Back to top

```
>>> Choropleth(geo_data="us-states.json", line_color="blue", line_weight=3)
>>> Choropleth(
...     geo_data="geo.json",
...     data=df,
...     columns=["Data 1", "Data 2"],
...     key_on="feature.properties.myvalue",
...     fill_color="PuBu",
...     bins=[0, 20, 30, 40, 50, 60],
... )
>>> Choropleth(geo_data="countries.json", topojson="objects.countries")
>>> Choropleth(
...     geo_data="geo.json",
...     data=df,
...     columns=["Data 1", "Data 2"],
...     key_on="feature.properties.myvalue",
...     fill_color="PuBu",
...     bins=[0, 20, 30, 40, 50, 60],
...     highlight=True,
... )
```

`render(**kwargs) → None`

Render the GeoJson/TopoJson and color scale objects.

`class folium.features.ClickForLatLng(format_str: str | None = None, alert: bool = True)`

Bases: `MacroElement`

When one clicks on a Map that contains a ClickForLatLng, the coordinates of the pointer's position are copied to clipboard.

Parameters:

- **format_str** (*str, default 'lat + "," + lng'*) – The javascript string used to format the text copied to clipboard. eg: format_str = 'lat + "," + lng' >> 46.558860,3.397397 format_str = '"[" + lat + "," + lng + "]"' >> [46.558860,3.397397]
- **alert** (*bool, default True*) – Whether there should be an alert when something has been copied to clipboard.

`class folium.features.ClickForMarker(popup: IFrame | Html | str | None = None)`

Bases: `MacroElement`

When one clicks on a Map that contains a ClickForMarker, a Marker is created at the pointer's position.

Parameters:

popup (*str or IFrame or Html, default None*) – Text to display in the markers' popups. This can also be an Element like IFrame or Html. If None, the popups will display the marker's latitude and longitude. You can include the latitude and longitude with ${lat} and ${lng}.

Examples

```
>>> ClickForMarker("<b>Lat:</b> ${lat}<br /><b>Lon:</b> ${lng}")
```

class folium.features.**ColorLine**(*positions: Iterable[Sequence[float]]*, *colors: Iterable[float]*, *colormap: ColorMap | Sequence[Any] | None = None*, *nb_steps: int = 12*, *weight: int | None = None*, *opacity: float | None = None*, ***kwargs: Any*)

Bases: FeatureGroup

Draw data on a map with specified colors.

Parameters:

- **positions** (*iterable of (lat, lon) pairs*) – The points on the line. Segments between points will be colored.
- **colors** (*iterable of float*) – Values that determine the color of a line segment. It must have length equal to *len(positions)-1*.
- **colormap** (*branca.colormap.Colormap or list or tuple*) – The colormap to use. If a list or tuple of colors is provided, a LinearColormap will be created from it.
- **nb_steps** (*int, default 12*) – The colormap will be discretized to this number of colors.
- **opacity** (*float, default 1*) – Line opacity, scale 0-1
- **weight** (*int, default 2*) – Stroke weight in pixels
- ****kwargs** – Further parameters available. See folium.map.FeatureGroup

Return type:

A ColorLine object that you can *add_to* a Map.

class folium.features.**CustomIcon**(*icon_image: Any*, *icon_size: Tuple[int, int] | None = None*, *icon_anchor: Tuple[int, int] | None = None*, *shadow_image: Any = None*, *shadow_size: Tuple[int, int] | None = None*, *shadow_anchor: Tuple[int, int] | None = None*, *popup_anchor: Tuple[int, int] | None = None*)

Bases: Icon

Create a custom icon, based on an image.

Parameters:

- **icon_image** (*string, file or array-like object*) – The data you want to use as an icon. * If string, it will be written directly in the output file. * If file, it's content will be converted as embedded in the output file. * If array-like, it will be converted to PNG base64 string and embedded in the output.
- **icon_size** (*tuple of 2 int, optional*) – Size of the icon image in pixels.
- **icon_anchor** (*tuple of 2 int, optional*) – The coordinates of the "tip" of the icon (relative to its top left corner). The icon will be aligned so that this point is at the marker's geographical location.
- **shadow_image** (*string, file or array-like object, optional*) – The data for the shadow image. If not specified, no shadow image will be created.
- **shadow_size** (*tuple of 2 int, optional*) – Size of the shadow image in pixels.
- **shadow_anchor** (*tuple of 2 int, optional*) – The coordinates of the "tip" of the shadow relative to its top left corner (the same as icon_anchor if not specified).
- **popup_anchor** (*tuple of 2 int, optional*) – The coordinates of the point from which popups will "open", relative to the icon anchor.

↑ Back to top

class folium.features.**DivIcon**(*html: str | None = None*, *icon_size: Tuple[int, int] | None = None*, *icon_anchor: Tuple[int, int] | None = None*, *popup_anchor: Tuple[int, int] | None = None*, *class_name: str = 'empty'*)

Bases: MacroElement

Represents a lightweight icon for markers that uses a simple *div* element instead of an image.

Parameters:

- **icon_size** (*tuple of 2 int*) – Size of the icon image in pixels.
- **icon_anchor** (*tuple of 2 int*) – The coordinates of the "tip" of the icon (relative to its top left corner). The icon will be aligned so that this point is at the marker's geographical location.
- **popup_anchor** (*tuple of 2 int*) – The coordinates of the point from which popups will "open", relative to the icon anchor.
- **class_name** (*string*) – A custom class name to assign to the icon. Leaflet defaults is 'leaflet-div-icon' which draws a little white square with a shadow. We set it 'empty' in folium.
- **html** (*string*) – A custom HTML code to put inside the div element.
- **https** (*See*)

class folium.features.**GeoJson**(*data: Any*, *style_function: Callable | None = None*, *highlight_function: Callable | None = None*, *popup_keep_highlighted: bool = False*, *name: str | None = None*, *overlay: bool = True*, *control: bool = True*, *show: bool = True*, *smooth_factor: float | None = None*, *tooltip: str | Tooltip | GeoJsonTooltip | None = None*, *embed: bool = True*, *popup: GeoJsonPopup | None = None*, *zoom_on_click: bool = False*, *marker: Circle | CircleMarker | Marker | None = None*, ***kwargs: str | float | bool | Sequence | dict | None*)

Bases: Layer

Creates a GeoJson object for plotting into a Map.

Parameters:

- **data** (*file, dict or str.*) – The GeoJSON data you want to plot. * If file, then data will be read in the file and fully

embedded in Leaflet's JavaScript. * If dict, then data will be converted to JSON and embedded in the JavaScript. * If str, then data will be passed to the JavaScript as-is. * If *__geo_interface__* is available, the *__geo_interface__* dictionary will be serialized to JSON and reprojected if *to_crs* is available.

- **style_function** (*function, default None*) – Function mapping a GeoJson Feature to a style dict.
- **highlight_function** (*function, default None*) – Function mapping a GeoJson Feature to a style dict for mouse events.
- **popup_keep_highlighted** (*bool, default False*) – Whether to keep the highlighting active while the popup is open
- **name** (*string, default None*) – The name of the Layer, as it will appear in LayerControls
- **overlay** (*bool, default True*) – Adds the layer as an optional overlay (True) or the base layer (False).
- **control** (*bool, default True*) – Whether the Layer will be included in LayerControls
- **show** (*bool, default True*) – Whether the layer will be shown on opening.
- **smooth_factor** (*float, default None*) – How much to simplify the polyline on each zoom level. More means better performance and smoother look, and less means more accurate representation. Leaflet defaults to 1.0.
- **tooltip** (*GeoJsonTooltip, Tooltip or str, default None*) – Display a text when hovering over the object. Can utilize the data, see folium.GeoJsonTooltip for info on how to do that.
- **popup** (*GeoJsonPopup, optional*) – Show a different popup for each feature by passing a GeoJsonPopup object.
- **marker** (*Circle, CircleMarker or Marker, optional*) – If your data contains Point geometry, you can format the markers by passing a Circle, CircleMarker or Marker object with your wanted options. The *style_function* and *highlight_function* will also target the marker object you passed.
- **embed** (*bool, default True*) – Whether to embed the data in the html file or not. Note that disabling embedding is only supported if you provide a file link or URL.
- **zoom_on_click** (*bool, default False*) – Set to True to enable zooming in on a geometry when clicking on it.
- ****kwargs** – Keyword arguments are passed to the geoJson object as extra options.

Examples

```
>>> # Providing filename that shall be embedded.
>>> GeoJson("foo.json")
>>> # Providing filename that shall not be embedded.
>>> GeoJson("foo.json", embed=False)
>>> # Providing dict.
>>> GeoJson(json.load(open("foo.json")))
>>> # Providing string.
>>> GeoJson(open("foo.json").read())
```

```
>>> # Provide a style_function that color all states green but Alabama.
>>> style_function = lambda x: {
...     "fillColor": (
...         "#0000ff" if x["properties"]["name"] == "Alabama" else "#00ff00"
...     )
... }
>>> GeoJson(geojson, style_function=style_function)
```

↑ Back to top

`convert_to_feature_collection() → None`

Convert data into a FeatureCollection if it is not already.

`find_identifier() → str`

Find a unique identifier for each feature, create it if needed.

According to the GeoJSON specs a feature:

- MAY have an 'id' field with a string or numerical value.
- MUST have a 'properties' field. The content can be any json object or even null.

`get_geojson_from_web(url: str) → dict`

`process_data(data: Any) → dict`

Convert an unknown data input into a geojson dictionary.

`render(**kwargs) → None`

Renders the HTML representation of the element.

`class folium.features.GeoJsonPopup(fields: Sequence[str], aliases: Sequence[str] | None = None, labels: bool = True, style: str = 'margin: auto;', class_name: str = 'foliumpopup', localize: bool = True, **kwargs: str | float | bool | Sequence | dict | None)`

Bases: `GeoJsonDetail`

Create a popup feature to bind to each element of a GeoJson layer based on its attributes.

Parameters:

- **fields** (*list or tuple.*) – Labels of GeoJson/TopoJson 'properties' or GeoPandas GeoDataFrame columns you'd like to display.
- **aliases** (*list/tuple of strings, same length/order as fields, default None.*) – Optional aliases you'd like to display in the tooltip as field name instead of the keys of *fields*.
- **labels** (*bool, default True.*) – Set to False to disable displaying the field names or aliases.
- **localize** (*bool, default False.*) – This will use JavaScript's .toLocaleString() to format 'clean' values as strings for the user's location; i.e. 1,000,000.00 comma separators, float truncation, etc. Available for most of JavaScript's primitive types (any

data you'll serve into the template).

- **style** (*str, default None.*) – HTML inline style properties like font and colors. Will be applied to a div with the text in it.

Examples

gjson = folium.GeoJson(gdf).add_to(m)

folium.features.GeoJsonPopup(fields=['NAME'],
labels=False).add_to(gjson)

class folium.features.**GeoJsonTooltip**(***fields: Sequence[str], aliases: Sequence[str] | None = None, labels: bool = True, localize: bool = False, style: str | None = None, class_name: str = 'foliumtooltip', sticky: bool = True, **kwargs: str | float | bool | Sequence | dict | None***)

Bases: `GeoJsonDetail`

Create a tooltip that uses data from either geojson or topojson.

Parameters:

- **fields** (*list or tuple.*) – Labels of GeoJson/TopoJson 'properties' or GeoPandas GeoDataFrame columns you'd like to display.
- **aliases** (*list/tuple of strings, same length/order as fields, default None.*) – Optional aliases you'd like to display in the tooltip as field name instead of the keys of *fields*.
- **labels** (*bool, default True.*) – Set to False to disable displaying the field names or aliases.
- **localize** (*bool, default False.*) – This will use JavaScript's .toLocaleString() to format 'clean' values as strings for the user's location; i.e. 1,000,000.00 comma separators, float truncation, etc. Available for most of JavaScript's primitive types (any data you'll serve into the template).
- **style** (*str, default None.*) – HTML inline style properties like font and colors. Will be applied to a div with the text in it.
- **sticky** (*bool, default True*) – Whether the tooltip should follow the mouse.
- ****kwargs** (*Assorted.*) – These values will map directly to the Leaflet Options. More info available here: https://leafletjs.com/reference.html#tooltip

Examples

Provide fields and aliases, with Style. >>> GeoJsonTooltip(... fields=["CNTY_NM", "census-pop-2015", "census-md-income-2015"], ... aliases=["County", "2015 Census Population", "2015 Median Income"], ... localize=True, ... style=(... d-color: grey; color: white; font-family:" ... "courier new; font-size: 24px; padding: 10px;" ...), ...) # Provide fields, with labels off and fixed tooltip positions. >>> GeoJsonTooltip(fields=("CNTY_NM",), labels=False, sticky=False)

↑ Back to top

class folium.features.**LatLngPopup**

Bases: `MacroElement`

When one clicks on a Map that contains a LatLngPopup, a popup is shown that displays the latitude and longitude of the pointer.

class folium.features.**RegularPolygonMarker**(***location: Sequence[float], number_of_sides: int = 4, rotation: int = 0, radius: int = 15, popup: Popup | str | None = None, tooltip: Tooltip | str | None = None, **kwargs: bool | str | float | None***)

Bases: `JSCSSMixin`, `Marker`

Custom markers using the Leaflet Data Vis Framework.

Parameters:

- **location** (*tuple or list*) – Latitude and Longitude of Marker (Northing, Easting)
- **number_of_sides** (*int, default 4*) – Number of polygon sides
- **rotation** (*int, default 0*) – Rotation angle in degrees
- **radius** (*int, default 15*) – Marker radius, in pixels
- **popup** (*string or Popup, optional*) – Input text or visualization for object displayed when clicking.
- **tooltip** (*str or folium.Tooltip, optional*) – Display a text when hovering over the object.
- ****kwargs** – See vector layers path_options for additional arguments.
- **https** (*//humangeo.github.io/leaflet-dvf/*)

default_js: *List[Tuple[str, str]]*

class folium.features.**TopoJson**(***data: Any, object_path: str, style_function: Callable | None = None, name: str | None = None, overlay: bool = True, control: bool = True, show: bool = True, smooth_factor: float | None = None, tooltip: str | Tooltip | None = None***)

Bases: `JSCSSMixin`, `Layer`

Creates a TopoJson object for plotting into a Map.

Parameters:

- **data** (*file, dict or str.*) – The TopoJSON data you want to plot. * If file, then data will be read in the file and fully embedded in Leaflet's JavaScript. * If dict, then data will be converted to JSON and embedded in the JavaScript. * If str, then data will be passed to the JavaScript as-is.
- **object_path** (*str*) – The path of the desired object into the TopoJson structure. Ex: 'objects.myobject'.

- **style_function** (*function, default None*) – A function mapping a TopoJson geometry to a style dict.
- **name** (*string, default None*) – The name of the Layer, as it will appear in LayerControls
- **overlay** (*bool, default False*) – Adds the layer as an optional overlay (True) or the base layer (False).
- **control** (*bool, default True*) – Whether the Layer will be included in LayerControls.
- **show** (*bool, default True*) – Whether the layer will be shown on opening.
- **smooth_factor** (*float, default None*) – How much to simplify the polyline on each zoom level. More means better performance and smoother look, and less means more accurate representation. Leaflet defaults to 1.0.
- **tooltip** (*GeoJsonTooltip, Tooltip or str, default None*) – Display a text when hovering over the object. Can utilize the data, see folium.GeoJsonTooltip for info on how to do that.

Examples

```
>>> # Providing file that shall be embedded.
>>> TopoJson(open("foo.json"), "object.myobject")
>>> # Providing filename that shall not be embedded.
>>> TopoJson("foo.json", "object.myobject")
>>> # Providing dict.
>>> TopoJson(json.load(open("foo.json")), "object.myobject")
>>> # Providing string.
>>> TopoJson(open("foo.json").read(), "object.myobject")
```

```
>>> # Provide a style_function that color all states green but Alabama.
>>> style_function = lambda x: {
...     "fillColor": (
...         "#0000ff" if x["properties"]["name"] == "Alabama" else "#00ff00"
...     )
... }
>>> TopoJson(topo_json, "object.myobject", style_function=style_function)
```

`default_js: List[Tuple[str, str]]`

`get_bounds() → List[List[float]]`

Computes the bounds of the object itself (not including it's children) in the form [[lat_min, lon_min], [lat_max, lon_max]]

`render(**kwargs) → None`

Renders the HTML representation of the element.

↑ Back to top

`style_data() → None`

Applies self.style_function to each feature of self.data.

`class folium.features.Vega(data: Any, width: int | str | None = None, height: int | str | None = None, left: int | str = '0%', top: int | str = '0%', position: str = 'relative')`

Bases: `JSCSSMixin`, `Element`

Creates a Vega chart element.

Parameters:

- **data** (*JSON-like str or object*) – The Vega description of the chart. It can also be any object that has a method *to_json*, so that you can (for instance) provide a *vincent* chart.
- **width** (*int or str, default None*) – The width of the output element. If None, either data['width'] (if available) or '100%' will be used. Ex: 120, '120px', '80%'
- **height** (*int or str, default None*) – The height of the output element. If None, either data['width'] (if available) or '100%' will be used. Ex: 120, '120px', '80%'
- **left** (*int or str, default '0%'*) – The horizontal distance of the output with respect to the parent HTML object. Ex: 120, '120px', '80%'
- **top** (*int or str, default '0%'*) – The vertical distance of the output with respect to the parent HTML object. Ex: 120, '120px', '80%'
- **position** (*str, default 'relative'*) – The *position* argument that the CSS shall contain. Ex: 'relative', 'absolute'

`default_js: List[Tuple[str, str]]`

`render(**kwargs) → None`

Renders the HTML representation of the element.

`class folium.features.VegaLite(data: Any, width: int | str | None = None, height: int | str | None = None, left: int | str = '0%', top: int | str = '0%', position: str = 'relative')`

Bases: `Element`

Creates a Vega-Lite chart element.

Parameters:

- **data** (*JSON-like str or object*) – The Vega-Lite description of the chart. It can also be any object that has a method *to_json*, so that you can (for instance) provide an *Altair* chart.
- **width** (*int or str, default None*) – The width of the output element. If None, either data['width'] (if available) or '100%' will be used. Ex: 120, '120px', '80%'

- **height** (*int or str, default None*) – The height of the output element. If None, either data['width'] (if available) or '100%' will be used. Ex: 120, '120px', '80%'
- **left** (*int or str, default '0%'*) – The horizontal distance of the output with respect to the parent HTML object. Ex: 120, '120px', '80%'
- **top** (*int or str, default '0%'*) – The vertical distance of the output with respect to the parent HTML object. Ex: 120, '120px', '80%'
- **position** (*str, default 'relative'*) – The *position* argument that the CSS shall contain. Ex: 'relative', 'absolute'

`render(**kwargs) → None`

Renders the HTML representation of the element.

property `vegalite_major_version: int | None`

Utilities

class `folium.utilities.JsCode(js_code: str | JsCode)`

Bases: `object`

Wrapper around Javascript code.

class `folium.elements.EventHandler(event: str, handler: JsCode)`

Bases: `MacroElement`

Add javascript event handlers.

Examples

```
>>> import folium
>>> from folium.utilities import JsCode
>>>
>>> m = folium.Map()
>>>
>>> geo_json_data = {
...     "type": "FeatureCollection",
...     "features": [
...         {
...             "type": "Feature",
...             "geometry": {
...                 "type": "Polygon",
...                 "coordinates": [
...                     [
...                         [100.0, 0.0],
...                         [101.0, 0.0],
...                         [101.0, 1.0],
...                         [100.0, 1.0],
...                         [100.0, 0.0],
...                     ]
...                 ],
...             },
...             "properties": {"prop1": {"title": "Somewhere on Sumatra"}},
...         }
...     ],
... }
>>>
>>> g = folium.GeoJson(geo_json_data).add_to(m)
>>>
>>> highlight = JsCode(
...     """
...     function highlight(e) {
...         e.target.original_color = e.layer.options.color;
...         e.target.setStyle({ color: "green" });
...     }
... """
... )
>>>
>>> reset = JsCode(
...     """
...     function reset(e) {
...         e.target.setStyle({ color: e.target.original_color });
...     }
... """
... )
>>>
>>> g.add_child(EventHandler("mouseover", highlight))
>>> g.add_child(EventHandler("mouseout", reset))
```

Plugins

Wrap some of the most popular leaflet external plugins.

class `folium.plugins.AntPath(locations, popup=None, tooltip=None, **kwargs)`

Bases: `JSCSSMixin`, `BaseMultiLocation`

Class for drawing AntPath polyline overlays on a map.

See `folium.vector_layers.path_options()` for the *Path* options.

Parameters:

- **locations** (*list of points (latitude, longitude)*) – Latitude and Longitude of line (Northing, Easting)
- **popup** (*str or folium.Popup, default None*) – Input text or visualization for object displayed when clicking.
- **tooltip** (*str or folium.Tooltip, optional*) – Display a text when hovering over the object.
- ****kwargs** – Polyline and AntPath options. See their Github page for the available parameters.

- **https** (*//github.com/rubenspgcavalcante/leaflet-ant-path/*)

default_js: *List[Tuple[str, str]]*

class **folium.plugins.BeautifyIcon**(*icon=None*, *icon_shape=None*, *border_width=3*, *border_color='#000'*, *text_color='#000'*, *background_color='#FFF'*, *inner_icon_style=''*, *spin=False*, *number=None*, ***kwargs*)

Bases: `JSCSSMixin`, `MacroElement`

Create a BeautifyIcon that can be added to a Marker

Parameters:

- **icon** (*string, default None*) – the Font-Awesome icon name to use to render the marker.
- **icon_shape** (*string, default None*) – the icon shape
- **border_width** (*integer, default 3*) – the border width of the icon
- **border_color** (*string with hexadecimal RGB, default '#000'*) – the border color of the icon
- **text_color** (*string with hexadecimal RGB, default '#000'*) – the text color of the icon
- **background_color** (*string with hexadecimal RGB, default '#FFF'*) – the background color of the icon
- **inner_icon_style** (*string with css styles for the icon, default ''*) – the css styles of the icon
- **spin** (*boolean, default False*) – allow the icon to be spinning.
- **number** (*integer, default None*) – the number of the icon.

Examples

Plugin Website: masajid390/BeautifyMarker >>> BeautifyIcon(... text_color="#000", border_color="transparent", background_color="#FFF" ...).add_to(marker) >>> number_icon = BeautifyIcon(... text_color="#000", ... border_color="transparent", ... background_color="#FFF", ... number=10, ... inner_icon_style="font-size:12px;padding-top:-5px;", ...) >>> Marker(... location=[45.5, -122.3], ... popup=folium.Popup("Portland, OR"), ... icon=number_icon, ...) >>> BeautifyIcon(icon="arrow-down", icon_shape="marker").add_to(marker)

ICON_SHAPE_TYPES

default_css: *List[Tuple[str, str]]*

↑ Back to top

default_js: *List[Tuple[str, str]]*

class **folium.plugins.BoatMarker**(*location*, *popup=None*, *icon=None*, *heading=0*, *wind_heading=None*, *wind_speed=0*, ***kwargs*)

Bases: `JSCSSMixin`, `Marker`

Add a Marker in the shape of a boat.

Parameters:

- **location** (*tuple of length 2, default None*) – The latitude and longitude of the marker. If None, then the middle of the map is used.
- **heading** (*int, default 0*) – Heading of the boat to an angle value between 0 and 360 degrees
- **wind_heading** (*int, default None*) – Heading of the wind to an angle value between 0 and 360 degrees If None, then no wind is represented.
- **wind_speed** (*int, default 0*) – Speed of the wind in knots.
- **https** (*//github.com/thomasbrueggemann/leaflet.boatmarker*)

default_js: *List[Tuple[str, str]]*

class **folium.plugins.CirclePattern**(*width=20*, *height=20*, *radius=12*, *weight=2.0*, *color='#3388ff'*, *fill_color='#3388ff'*, *opacity=0.75*, *fill_opacity=0.5*)

Bases: `JSCSSMixin`, `MacroElement`

Fill Pattern for polygon composed of repeating circles.

Add these to the 'fillPattern' field in GeoJson style functions.

Parameters:

- **width** (*int, default 20*) – Horizontal distance between circles (pixels).
- **height** (*int, default 20*) – Vertical distance between circles (pixels).
- **radius** (*int, default 12*) – Radius of each circle (pixels).
- **weight** (*float, default 2.0*) – Width of outline around each circle (pixels).
- **color** (*string with hexadecimal, RGB, or named color, default "#3388ff"*) – Color of the circle outline.
- **fill_color** (*string with hexadecimal, RGB, or named color, default "#3388ff"*) – Color of the circle interior.
- **opacity** (*float, default 0.75*) – Opacity of the circle outline. Should be between 0 and 1.
- **fill_opacity** (*float, default 0.5*) – Opacity of the circle interior. Should be between 0 and 1.
- **https** (*See*)

default_js: *List[Tuple[str, str]]*

render(**kwargs)

Renders the HTML representation of the element.

class folium.plugins.Draw(export=False, filename='data.geojson', position='topleft', show_geometry_on_click=True, draw_options=None, edit_options=None)

Bases: JSCSSMixin, MacroElement

Vector drawing and editing plugin for Leaflet.

Parameters:

- **export** (*bool, default False*) – Add a small button that exports the drawn shapes as a geojson file.
- **filename** (*string, default 'data.geojson'*) – Name of geojson file
- **position** (*{'topleft', 'topright', 'bottomleft', 'bottomright'}*) – Position of control. See https://leafletjs.com/reference.html#control
- **show_geometry_on_click** (*bool, default True*) – When True, opens an alert with the geometry description on click.
- **draw_options** (*dict, optional*) – The options used to configure the draw toolbar. See http://leaflet.github.io/Leaflet.draw/docs/leaflet-draw-latest.html#drawoptions
- **edit_options** (*dict, optional*) – The options used to configure the edit toolbar. See https://leaflet.github.io/Leaflet.draw/docs/leaflet-draw-latest.html#editpolyoptions

Examples

```
>>> m = folium.Map()
>>> Draw(
...     export=True,
...     filename="my_data.geojson",
...     position="topleft",
...     draw_options={"polyline": {"allowIntersection": False}},
...     edit_options={"poly": {"allowIntersection": False}},
... ).add_to(m)
```

For more info please check https://leaflet.github.io/Leaflet.draw/docs/leaflet-draw-latest.html

default_css: List[Tuple[str, str]]

default_js: List[Tuple[str, str]]

↑ Back to top

render(**kwargs)

Renders the HTML representation of the element.

class folium.plugins.DualMap(location=None, layout='horizontal', **kwargs)

Bases: JSCSSMixin, MacroElement

Create two maps in the same window.

Adding children to this objects adds them to both maps. You can access the individual maps with *DualMap.m1* and *DualMap.m2*.

Uses the Leaflet plugin Sync: jieter/Leaflet.Sync

Parameters:

- **location** (*tuple or list, optional*) – Latitude and longitude of center point of the maps.
- **layout** (*{'horizontal', 'vertical'}*) – Select how the two maps should be positioned. Either horizontal (left and right) or vertical (top and bottom).
- ****kwargs** – Keyword arguments are passed to the two Map objects.

Examples

```
>>> # DualMap accepts the same arguments as Map:
>>> m = DualMap(location=(0, 0), tiles="cartodbpositron", zoom_start=5)
>>> # Add the same marker to both maps:
>>> Marker((0, 0)).add_to(m)
>>> # The individual maps are attributes called `m1` and `m2`:
>>> Marker((0, 1)).add_to(m.m1)
>>> LayerControl().add_to(m)
>>> m.save("map.html")
```

add_child(child, name=None, index=None)

Add object *child* to the first map and store it for the second.

default_js: List[Tuple[str, str]]

fit_bounds(*args, **kwargs)

keep_in_front(*args)

render(***kwargs*)

Renders the HTML representation of the element.

class **folium.plugins.FastMarkerCluster**(*data*, *callback=None*, *options=None*, *name=None*, *overlay=True*, *control=True*, *show=True*, *icon_create_function=None*, ***kwargs*)

Bases: `MarkerCluster`

Add marker clusters to a map using in-browser rendering. Using FastMarkerCluster it is possible to render 000's of points far quicker than the MarkerCluster class.

Be aware that the FastMarkerCluster class passes an empty list to the parent class' __init__ method during initialisation. This means that the add_child method is never called, and no reference to any marker data are retained. Methods such as get_bounds() are therefore not available when using it.

Parameters:

- **data** (*list of list with values*) – List of list of shape [[lat, lon], [lat, lon], etc.] When you use a custom callback you could add more values after the lat and lon. E.g. [[lat, lon, 'red'], [lat, lon, 'blue']]
- **callback** (*string, optional*) – A string representation of a valid Javascript function that will be passed each row in data. See the FasterMarkerCluster for an example of a custom callback.
- **name** (*string, optional*) – The name of the Layer, as it will appear in LayerControls.
- **overlay** (*bool, default True*) – Adds the layer as an optional overlay (True) or the base layer (False).
- **control** (*bool, default True*) – Whether the Layer will be included in LayerControls.
- **show** (*bool, default True*) – Whether the layer will be shown on opening.
- **icon_create_function** (*string, default None*) – Override the default behaviour, making possible to customize markers colors and sizes.
- ****kwargs** – Additional arguments are passed to Leaflet.markercluster options. See Leaflet/Leaflet.markercluster

class **folium.plugins.FeatureGroupSubGroup**(*group*, *name=None*, *overlay=True*, *control=True*, *show=True*)

Bases: `JSCSSMixin`, `Layer`

Creates a Feature Group that adds its child layers into a parent group when added to a map (e.g. through LayerControl). Useful to create nested groups, or cluster markers from multiple overlays. From [0].

↑ Back to top /Leaflet.FeatureGroup.SubGroup

Parameters:

- **group** (*Layer*) – The MarkerCluster or FeatureGroup containing this subgroup.
- **name** (*string, default None*) – The name of the Layer, as it will appear in LayerControls
- **overlay** (*bool, default True*) – Adds the layer as an optional overlay (True) or the base layer (False).
- **control** (*bool, default True*) – Whether the Layer will be included in LayerControls.
- **show** (*bool, default True*) – Whether the layer will be shown on opening.

Examples

Nested groups

```
>>> fg = folium.FeatureGroup()  # Main group
>>> g1 = folium.plugins.FeatureGroupSubGroup(fg, "g1")  # First subgroup of fg
>>> g2 = folium.plugins.FeatureGroupSubGroup(fg, "g2")  # Second subgroup of fg
>>> m.add_child(fg)
>>> m.add_child(g1)
>>> m.add_child(g2)
>>> g1.add_child(folium.Marker([0, 0]))
>>> g2.add_child(folium.Marker([0, 1]))
>>> folium.LayerControl().add_to(m)
```

Multiple overlays part of the same cluster group

```
>>> mcg = folium.plugins.MarkerCluster(
...     control=False
... )  # Marker Cluster, hidden in controls
>>> g1 = folium.plugins.FeatureGroupSubGroup(mcg, "g1")  # First group, in mcg
>>> g2 = folium.plugins.FeatureGroupSubGroup(mcg, "g2")  # Second group, in mcg
>>> m.add_child(mcg)
>>> m.add_child(g1)
>>> m.add_child(g2)
>>> g1.add_child(folium.Marker([0, 0]))
>>> g2.add_child(folium.Marker([0, 1]))
>>> folium.LayerControl().add_to(m)
```

default_js: *List[Tuple[str, str]]*

class **folium.plugins.FloatImage**(*image*, *bottom=75*, *left=75*, ***kwargs*)

Bases: `MacroElement`

Adds a floating image in HTML canvas on top of the map.

Parameters:

- **image** (*str*) – Url to image location. Can also be an inline image using a data URI or a local file using *file://*.

- **bottom** (*int, default 75*) – Vertical position from the bottom, as a percentage of screen height.
- **left** (*int, default 75*) – Horizontal position from the left, as a percentage of screen width.
- ****kwargs** – Additional keyword arguments are applied as CSS properties. For example: *width='300px'*.

class **folium.plugins.Fullscreen**(***position='topleft'***, ***title='Full Screen'***, ***title_cancel='Exit Full Screen'***, ***force_separate_button=False***, ****kwargs***)

Bases: `JSCSSMixin`, `MacroElement`

Adds a fullscreen button to your map.

Parameters:

- **position** (*str*) – change the position of the button can be: 'topleft', 'topright', 'bottomright' or 'bottomleft' default: 'topleft'
- **title** (*str*) – change the title of the button, default: 'Full Screen'
- **title_cancel** (*str*) – change the title of the button when fullscreen is on, default: 'Exit Full Screen'
- **force_separate_button** (*bool, default False*) – force separate button to detach from zoom buttons,
- **https** (*See*)

default_css: List[Tuple[str, str]]

default_js: List[Tuple[str, str]]

class **folium.plugins.Geocoder**(***collapsed: bool = False***, ***position: str = 'topright'***, ***add_marker: bool = True***, ***zoom: int | None = 11***, ***provider: str = 'nominatim'***, ***provider_options: dict = {}***, ****kwargs***)

Bases: `JSCSSMixin`, `MacroElement`

A simple geocoder for Leaflet that by default uses OSM/Nominatim.

Please respect the Nominatim usage policy: https://operations.osmfoundation.org/policies/nominatim/

Parameters:

- **collapsed** (*bool, default False*) – If True, collapses the search box unless hovered/clicked.
- **position** (*str, default 'topright'*) – Choose from 'topleft', 'topright', 'bottomleft' or 'bottomright'.
- **add_marker** (*bool, default True*) – If True, adds a marker on the found location.
- **zoom** (*int, default 11, optional*) – Set zoom level used for displaying the geocode result, note that this only has an effect when add_marker is set to False. Set this to None to preserve the current map zoom level.
- **provider** (*str, default 'nominatim'*) – Defaults to "nominatim", see perliedman/leaflet-control-geocoder for other built-in providers.
- **provider_options** (*dict, default {}*) – For use with specific providers that may require api keys or other parameters.
- **https** (*For all options see*)

default_css: List[Tuple[str, str]]

default_js: List[Tuple[str, str]]

class **folium.plugins.GroupedLayerControl**(***groups***, ***exclusive_groups=True***, ****kwargs***)

Bases: `JSCSSMixin`, `MacroElement`

Create a Layer Control with groups of overlays.

Parameters:

- **groups** (*dict*) –

  A dictionary where the keys are group names and the values are lists of layer objects. e.g. {

  > "Group 1": [layer1, layer2], "Group 2": [layer3, layer4]
  >
  > }

- **exclusive_groups** (*bool, default True*) – Whether to use radio buttons (default) or checkboxes. If you want to use both, use two separate instances of this class.
- ****kwargs** – Additional (possibly inherited) options. See https://leafletjs.com/reference.html#control-layers

default_css: List[Tuple[str, str]]

default_js: List[Tuple[str, str]]

class **folium.plugins.HeatMap**(***data***, ***name=None***, ***min_opacity=0.5***, ***max_zoom=18***, ***radius=25***, ***blur=15***, ***gradient=None***, ***overlay=True***, ***control=True***, ***show=True***, ****kwargs***)

Bases: `JSCSSMixin`, `Layer`

Create a Heatmap layer

↑ Back to top

Parameters:

- **data** (*list of points of the form [lat, lng] or [lat, lng, weight]*) – The points you want to plot. You can also provide a numpy.array of shape (n,2) or (n,3).
- **name** (*string, default None*) – The name of the Layer, as it will appear in LayerControls.
- **min_opacity** (*default 1.*) – The minimum opacity the heat will start at.
- **max_zoom** (*default 18*) – Zoom level where the points reach maximum intensity (as intensity scales with zoom), equals maxZoom of the map by default
- **radius** (*int, default 25*) – Radius of each "point" of the heatmap
- **blur** (*int, default 15*) – Amount of blur
- **gradient** (*dict, default None*) – Color gradient config. e.g. {0.4: 'blue', 0.65: 'lime', 1: 'red'}
- **overlay** (*bool, default True*) – Adds the layer as an optional overlay (True) or the base layer (False).
- **control** (*bool, default True*) – Whether the Layer will be included in LayerControls.
- **show** (*bool, default True*) – Whether the layer will be shown on opening.

`default_js: List[Tuple[str, str]]`

`class folium.plugins.HeatMapWithTime(data, index=None, name=None, radius=15, blur=0.8, min_opacity=0, max_opacity=0.6, scale_radius=False, gradient=None, use_local_extrema=False, auto_play=False, display_index=True, index_steps=1, min_speed=0.1, max_speed=10, speed_step=0.1, position='bottomleft', overlay=True, control=True, show=True)`

Bases: `JSCSSMixin`, `Layer`

Create a HeatMapWithTime layer

Parameters:

- **data** (*list of list of points of the form [lat, lng] or [lat, lng, weight]*) – The points you want to plot. The outer list corresponds to the various time steps in sequential order. (weight is in (0, 1] range and defaults to 1 if not specified for a point)
- **index** (*Index giving the label (or timestamp) of the elements of data. Should have*) – the same length as data, or is replaced by a simple count if not specified.
- **name** (*string, default None*) – The name of the Layer, as it will appear in LayerControls.
- **radius** (*default 15.*) – The radius used around points for the heatmap.
- **blur** (*default 0.8.*) – Blur strength used for the heatmap. Must be between 0 and 1.
- **min_opacity** (*default 0*) – The minimum opacity for the heatmap.
- **max_opacity** (*default 0.6*) – The maximum opacity for the heatmap.
- **scale_radius** (*default False*) – Scale the radius of the points based on the zoom level.
- **gradient** (*dict, default None*) – Match point density values to colors. Color can be a name ('red'), RGB values ('rgb(255,0,0)') or a hex number ('#FF0000').
- **use_local_extrema** (*default False*) – Defines whether the heatmap uses a global extrema set found from the input data OR a local extrema (the maximum and minimum of the currently displayed view).
- **auto_play** (*default False*) – Automatically play the animation across time.
- **display_index** (*default True*) – Display the index (usually time) in the time control.
- **index_steps** (*default 1*) – Steps to take in the index dimension between animation steps.
- **min_speed** (*default 0.1*) – Minimum fps speed for animation.
- **max_speed** (*default 10*) – Maximum fps speed for animation.
- **speed_step** (*default 0.1*) – Step between different fps speeds on the speed slider.
- **position** (*default 'bottomleft'*) – Position string for the time slider. Format: 'bottom/top'+'left/right'.
- **overlay** (*bool, default True*) – Adds the layer as an optional overlay (True) or the base layer (False).
- **control** (*bool, default True*) – Whether the Layer will be included in LayerControls.
- **show** (*bool, default True*) – Whether the layer will be shown on opening.

`default_css: List[Tuple[str, str]]`

`default_js: List[Tuple[str, str]]`

`render(**kwargs)`

Renders the HTML representation of the element.

`class folium.plugins.LocateControl(auto_start=False, **kwargs)`

Bases: `JSCSSMixin`, `MacroElement`

Control plugin to geolocate the user.

This plugins adds a button to the map, and when it's clicked shows the current user device location.

To work properly in production, the connection needs to be encrypted, otherwise browser will not allow users to share their location.

Parameters:

- **auto-start** (*bool, default False*) – When set to True, plugin will be activated on map loading and search for user position. Once user location is founded, the map will automatically centered in using user coordinates.
- ****kwargs** – For possible options, see domoritz/leaflet-locatecontrol

↑ Back to top

Examples

```
>>> m = folium.Map()
# With default settings
>>> LocateControl().add_to(m)
```

With some custom options >>> LocateControl(... position="bottomright", ... strings={"title": "See you current location", "popup": "Your position"}, ...).add_to(m)

For more info check: domoritz/leaflet-locatecontrol

default_css: *List[Tuple[str, str]]*

default_js: *List[Tuple[str, str]]*

class **folium.plugins.MarkerCluster**(***locations***=*None*, ***popups***=*None*, ***icons***=*None*, ***name***=*None*, ***overlay***=*True*, ***control***=*True*, ***show***=*True*, ***icon_create_function***=*None*, ***options***=*None*, ****kwargs***)

Bases: `JSCSSMixin`, `Layer`

Provides Beautiful Animated Marker Clustering functionality for maps.

Parameters:

- **locations** (*list of list or array of shape (n, 2).*) – Data points of the form [[lat, lng]].
- **popups** (*list of length n, default None*) – Popup for each marker, either a Popup object or a string or None.
- **icons** (*list of length n, default None*) – Icon for each marker, either an Icon object or a string or None.
- **name** (*string, default None*) – The name of the Layer, as it will appear in LayerControls
- **overlay** (*bool, default True*) – Adds the layer as an optional overlay (True) or the base layer (False).
- **control** (*bool, default True*) – Whether the Layer will be included in LayerControls.
- **show** (*bool, default True*) – Whether the layer will be shown on opening.
- **icon_create_function** (*string, default None*) – Override the default behaviour, making possible to customize markers colors and sizes.
- **options** (*dict, default None*) – A dictionary with options for Leaflet.markercluster. See Leaflet/Leaflet.markercluster for options.

↑ Back to top

```
>>> icon_create_function = '''
...     function(cluster) {
...     return L.divIcon({html: '<b>' + cluster.getChildCount() + '</b>',
...                       className: 'marker-cluster marker-cluster-small',
...                       iconSize: new L.Point(20, 20)});
...     }
... '''
```

default_css: *List[Tuple[str, str]]*

default_js: *List[Tuple[str, str]]*

class **folium.plugins.MeasureControl**(***position***='topright', ***primary_length_unit***='meters', ***secondary_length_unit***='miles', ***primary_area_unit***='sqmeters', ***secondary_area_unit***='acres', ****kwargs***)

Bases: `JSCSSMixin`, `MacroElement`

Add a measurement widget on the map.

Parameters:

- **position** (*str, default 'topright'*) – Location of the widget.
- **primary_length_unit** (*str, default 'meters'*)
- **secondary_length_unit** (*str, default 'miles'*)
- **primary_area_unit** (*str, default 'sqmeters'*)
- **secondary_area_unit** (*str, default 'acres'*)
- **https** (*See*)

default_css: *List[Tuple[str, str]]*

default_js: *List[Tuple[str, str]]*

class **folium.plugins.MiniMap**(***tile_layer***=*None*, ***position***='bottomright', ***width***=*150*, ***height***=*150*, ***collapsed_width***=*25*, ***collapsed_height***=*25*, ***zoom_level_offset***=*-5*, ***zoom_level_fixed***=*None*, ***center_fixed***=*False*, ***zoom_animation***=*False*, ***toggle_display***=*False*, ***auto_toggle_display***=*False*, ***minimized***=*False*, ****kwargs***)

Bases: `JSCSSMixin`, `MacroElement`

Add a minimap (locator) to an existing map.

Uses the Leaflet plugin by Norkart under BSD 2-Clause "Simplified" License. Norkart/Leaflet-MiniMap

Parameters:

- **tile_layer** (*folium TileLayer object or str, default None*) – Provide a folium TileLayer object or the wanted tiles as string. If not provided it will use the default of 'TileLayer', currently OpenStreetMap.
- **position** (*str, default 'bottomright'*) – The standard Control position parameter for the widget.
- **width** (*int, default 150*) – The width of the minimap in pixels.
- **height** (*int, default 150*) – The height of the minimap in pixels.
- **collapsed_width** (*int, default 25*) – The width of the toggle marker and the minimap when collapsed in pixels.
- **collapsed_height** (*int, default 25*) – The height of the toggle marker and the minimap when collapsed
- **zoom_level_offset** (*int, default -5*) – The offset applied to the zoom in the minimap compared to the zoom of the main map. Can be positive or negative.
- **zoom_level_fixed** (*int, default None*) – Overrides the offset to apply a fixed zoom level to the minimap regardless of the main map zoom. Set it to any valid zoom level, if unset zoom_level_offset is used instead.
- **center_fixed** (*bool, default False*) – Applies a fixed position to the minimap regardless of the main map's view / position. Prevents panning the minimap, but does allow zooming (both in the minimap and the main map). If the minimap is zoomed, it will always zoom around the centerFixed point. You can pass in a LatLng-equivalent object.
- **zoom_animation** (*bool, default False*) – Sets whether the minimap should have an animated zoom. (Will cause it to lag a bit after the movement of the main map.)
- **toggle_display** (*bool, default False*) – Sets whether the minimap should have a button to minimise it.
- **auto_toggle_display** (*bool, default False*) – Sets whether the minimap should hide automatically if the parent map bounds does not fit within the minimap bounds. Especially useful when 'zoomLevelFixed' is set.
- **minimized** (*bool, default False*) – Sets whether the minimap should start in a minimized position.

Examples

```
>>> MiniMap(position="bottomleft")
```

default_css: *List[Tuple[str, str]]*

default_js: *List[Tuple[str, str]]*

class folium.plugins.**MousePosition**(***position***='bottomright', ***separator***=' : ', ***empty_string***='Unavailable', ***lng_first***=False, ***num_digits***=5, ***prefix***='', ***lat_formatter***=None, ***lng_formatter***=None, *****kwargs***)

↑ Back to top

Bases: `JSMixin`, `MacroElement`

Add a field that shows the coordinates of the mouse position.

Uses the Leaflet plugin by Ardhi Lukianto under MIT license. ardhi/Leaflet.MousePosition

Parameters:

- **position** (*str, default 'bottomright'*) – The standard Control position parameter for the widget.
- **separator** (*str, default ' : '*) – Character used to separate latitude and longitude values.
- **empty_string** (*str, default 'Unavailable'*) – Initial text to display.
- **lng_first** (*bool, default False*) – Whether to put the longitude first or not. Set as True to display longitude before latitude.
- **num_digits** (*int, default '5'*) – Number of decimal places included in the displayed longitude and latitude decimal degree values.
- **prefix** (*str, default ''*) – A string to be prepended to the coordinates.
- **lat_formatter** (*str, default None*) – Custom Javascript function to format the latitude value.
- **lng_formatter** (*str, default None*) – Custom Javascript function to format the longitude value.

Examples

```
>>> fmtr = "function(num) {return L.Util.formatNum(num, 3) + ' º ';};"
>>> MousePosition(
...     position="topright",
...     separator=" | ",
...     prefix="Mouse:",
...     lat_formatter=fmtr,
...     lng_formatter=fmtr,
... )
```

default_css: *List[Tuple[str, str]]*

default_js: *List[Tuple[str, str]]*

class folium.plugins.**PolyLineFromEncoded**(***encoded: str***, *****kwargs***)

Bases: `_BaseFromEncoded`

Create PolyLines directly from the encoded string.

Parameters:

- **encoded** (*str*) – The raw encoded string from the Polyline Encoding Algorithm. See: https://developers.google.com/maps/documentation/utilities/polylinealgorithm
- ****kwargs** – Polyline options as accepted by leaflet. See: https://leafletjs.com/reference.html#polyline
- **https** (*Adapted from*)

Examples

```
>>> from folium import Map
>>> from folium.plugins import PolyLineFromEncoded
>>> m = Map()
>>> encoded = r"_p~iF~cn~U_ulLn{vA_mqNvxq`@"
>>> PolyLineFromEncoded(encoded=encoded, color="green").add_to(m)
```

class folium.plugins.**PolyLineOffset**(*locations*, *popup=None*, *tooltip=None*, *offset=0*, ***kwargs*)

Bases: `JSCSSMixin`, `PolyLine`

Add offset capabilities to the PolyLine class.

This plugin adds to folium Polylines the ability to be drawn with a relative pixel offset, without modifying their actual coordinates. The offset value can be either negative or positive, for left- or right-side offset, and remains constant across zoom levels.

See `folium.vector_layers.path_options()` for the *Path* options.

Parameters:

- **locations** (*list of points (latitude, longitude)*) – Latitude and Longitude of line (Northing, Easting)
- **popup** (*str or folium.Popup, default None*) – Input text or visualization for object displayed when clicking.
- **tooltip** (*str or folium.Tooltip, optional*) – Display a text when hovering over the object.
- **offset** (*int, default 0*) – Relative pixel offset to draw a line parallel to an existent one, at a fixed distance.
- ****kwargs** – Polyline options. See their Github page for the available parameters.
- **https** (*See*)

Examples

```
>>> plugins.PolyLineOffset(
...     [[58, -28], [53, -23]], color="#f00", opacity=1, offset=-5
... ).add_to(m)
>>> plugins.PolyLineOffset(
...     [[58, -28], [53, -23]], color="#080", opacity=1, offset=10
... ).add_to(m)
```

default_js: *List[Tuple[str, str]]*

↑ Back to top

class folium.plugins.**PolyLineTextPath**(*polyline*, *text*, *repeat=False*, *center=False*, *below=False*, *offset=0*, *orientation=0*, *attributes=None*, ***kwargs*)

Bases: `JSCSSMixin`, `MacroElement`

Shows a text along a PolyLine.

Parameters:

- **polyline** (*folium.features.PolyLine object*) – The folium.features.PolyLine object to attach the text to.
- **text** (*string*) – The string to be attached to the polyline.
- **repeat** (*bool, default False*) – Specifies if the text should be repeated along the polyline.
- **center** (*bool, default False*) – Centers the text according to the polyline's bounding box
- **below** (*bool, default False*) – Show text below the path
- **offset** (*int, default 0*) – Set an offset to position text relative to the polyline.
- **orientation** (*int, default 0*) – Rotate text to a specified angle.
- **attributes** (*dict*) – Object containing the attributes applied to the text tag. Check valid attributes here: https://developer.mozilla.org/en-US/docs/Web/SVG/Element/text#attributes Example: {'fill': '#007DEF', 'font-weight': 'bold', 'font-size': '24'}
- **https** (*See*)

default_js: *List[Tuple[str, str]]*

class folium.plugins.**PolygonFromEncoded**(*encoded: str*, ***kwargs*)

Bases: `_BaseFromEncoded`

Create Polygons directly from the encoded string.

Parameters:

- **encoded** (*str*) – The raw encoded string from the Polyline Encoding Algorithm. See: https://developers.google.com/maps/documentation/utilities/polylinealgorithm
- ****kwargs** – Polygon options as accepted by leaflet. See: https://leafletjs.com/reference.html#polygon
- **https** (*Adapted from*)

Examples

```
>>> from folium import Map
>>> from folium.plugins import PolygonFromEncoded
>>> m = Map()
>>> encoded = r"w`j~FpxivO}jz@qnnCd}~Bsa{@~f`C`lkH"
>>> PolygonFromEncoded(encoded=encoded).add_to(m)
```

class folium.plugins.**Realtime**(*source: str | dict | JsCode*, *start: bool = True*, *interval: int = 60000*, *get_feature_id: JsCode | str | None = None*, *update_feature: JsCode | str | None = None*, *remove_missing: bool = False*, *container: Layer | None = None*, ***kwargs*)

Bases: `JSCSSMixin`, `MacroElement`

Put realtime data on a Leaflet map: live tracking GPS units, sensor data or just about anything.

Based on: perliedman/leaflet-realtime

Parameters:

- **source** (*str, dict, JsCode*) –
  The source can be one of:
  - a string with the URL to get data from
  - a dict that is passed to javascript's *fetch* function for fetching the data
  - a *folium.JsCode* object in case you need more freedom.
- **start** (*bool, default True*) – Should automatic updates be enabled when layer is added on the map and stopped when layer is removed from the map
- **interval** (*int, default 60000*) – Automatic update interval, in milliseconds
- **get_feature_id** (*str or JsCode, optional*) – A JS function with a geojson *feature* as parameter default returns *feature.properties.id* Function to get an identifier to uniquely identify a feature over time
- **update_feature** (*str or JsCode, optional*) – A JS function with a geojson *feature* as parameter Used to update an existing feature's layer; by default, points (markers) are updated, other layers are discarded and replaced with a new, updated layer. Allows to create more complex transitions, for example, when a feature is updated
- **remove_missing** (*bool, default False*) – Should missing features between updates been automatically removed from the layer
- **container** (*Layer, default GeoJson*) – The container will typically be a *FeatureGroup*, *MarkerCluster* or *GeoJson*, but it can be anything that generates a javascript L.LayerGroup object, i.e. something that has the methods *addLayer* and *removeLayer*.
- **layer** (*Other keyword arguments are passed to the GeoJson*)
- **pass** (*so you can*)
- **style**
- **wrap** (*point_to_layer and/or on_each_feature. Make sure to*)
- **class.** (*Javascript functions in the JsCode*)

↑ Back to top

Examples

```
>>> from folium import JsCode
>>> m = folium.Map(location=[40.73, -73.94], zoom_start=12)
>>> rt = Realtime(
...     "https://raw.githubusercontent.com/python-visualization/folium-example-data/main/subway_stations.geojson",
...     get_feature_id=JsCode("(f) => { return f.properties.objectid; }"),
...     point_to_layer=JsCode(
...         "(f, latlng) => { return L.circleMarker(latlng, {radius: 8, fillOpacity: 0.2})}"
...     ),
...     interval=10000,
... )
>>> rt.add_to(m)
```

default_js: *List[Tuple[str, str]]*

class folium.plugins.**ScrollZoomToggler**

Bases: `MacroElement`

Creates a button for enabling/disabling scroll on the Map.

class folium.plugins.**Search**(*layer*, *search_label=None*, *search_zoom=None*, *geom_type='Point'*, *position='topleft'*, *placeholder='Search'*, *collapsed=False*, ***kwargs*)

Bases: `JSCSSMixin`, `MacroElement`

Adds a search tool to your map.

Parameters:

- **layer** (*GeoJson, TopoJson, FeatureGroup, MarkerCluster class object.*) – The map layer to index in the Search view.
- **search_label** (*str, optional*) – 'properties' key in layer to index Search, if layer is GeoJson/TopoJson.
- **search_zoom** (*int, optional*) – Zoom level to set the map to on match. By default zooms to Polygon/Line bounds and points on their natural extent.
- **geom_type** (*str, default 'Point'*) – Feature geometry type. "Point", "Line" or "Polygon"
- **position** (*str, default 'topleft'*) – Change the position of the search bar, can be: 'topleft', 'topright', 'bottomright' or 'bottomleft',
- **placeholder** (*str, default 'Search'*) – Placeholder text inside the Search box if nothing is entered.
- **collapsed** (*boolean, default False*) – Whether the Search box should be collapsed or not.
- ****kwargs.** – Assorted style options to change feature styling on match. Use the same way as vector layer arguments.
- **https** (*See*)

default_css: *List[Tuple[str, str]]*

default_js: *List[Tuple[str, str]]*

render(**kwargs)

Renders the HTML representation of the element.

test_params(keys)

class folium.plugins.SemiCircle(location, radius, direction=None, arc=None, start_angle=None, stop_angle=None, popup=None, tooltip=None, **kwargs)

Bases: JSCSSMixin, Marker

Add a marker in the shape of a semicircle, similar to the Circle class.

Use (direction and arc) or (start_angle and stop_angle), not both.

Parameters:

- **location** (*tuple[float, float]*) – Latitude and Longitude pair (Northing, Easting)
- **radius** (*float*) – Radius of the circle, in meters.
- **direction** (*int, default None*) – Direction angle in degrees
- **arc** (*int, default None*) – Arc angle in degrees.
- **start_angle** (*int, default None*) – Start angle in degrees
- **stop_angle** (*int, default None*) – Stop angle in degrees.
- **popup** (*str or folium.Popup, optional*) – Input text or visualization for object displayed when clicking.
- **tooltip** (*str or folium.Tooltip, optional*) – Display a text when hovering over the object.
- ****kwargs** – For additional arguments see folium.vector_layers.path_options()
- **https** (*Uses Leaflet plugin*)

default_js: List[Tuple[str, str]]

class folium.plugins.SideBySideLayers(layer_left, layer_right)

Bases: JSCSSMixin, MacroElement

↑ Back to top

deBySideLayers that takes two Layers and adds a sliding control with the leaflet-side-by-side plugin.

Uses the Leaflet leaflet-side-by-side plugin digidem/leaflet-side-by-side

Parameters:

- **layer_left** (*Layer.*) – The left Layer within the side by side control. Must be created and added to the map before being passed to this class.
- **layer_right** (*Layer.*) – The right Layer within the side by side control. Must be created and added to the map before being passed to this class.

Examples

```
>>> sidebyside = SideBySideLayers(layer_left, layer_right)
>>> sidebyside.add_to(m)
```

default_js: List[Tuple[str, str]]

class folium.plugins.StripePattern(angle=0.5, weight=4, space_weight=4, color='#000000', space_color='#ffffff', opacity=0.75, space_opacity=0.0, **kwargs)

Bases: JSCSSMixin, MacroElement

Fill Pattern for polygon composed of alternating lines.

Add these to the 'fillPattern' field in GeoJson style functions.

Parameters:

- **angle** (*float, default 0.5*) – Angle of the line pattern (degrees). Should be between -360 and 360.
- **weight** (*float, default 4*) – Width of the main lines (pixels).
- **space_weight** (*float*) – Width of the alternate lines (pixels).
- **color** (*string with hexadecimal, RGB, or named color, default "#000000"*) – Color of the main lines.
- **space_color** (*string with hexadecimal, RGB, or named color, default "#ffffff"*) – Color of the alternate lines.
- **opacity** (*float, default 0.75*) – Opacity of the main lines. Should be between 0 and 1.
- **space_opacity** (*float, default 0.0*) – Opacity of the alternate lines. Should be between 0 and 1.
- **https** (*See*)

default_js: List[Tuple[str, str]]

render(**kwargs)

Renders the HTML representation of the element.

class folium.plugins.TagFilterButton(data, icon='fa-filter', clear_text='clear',

*filter_on_every_click=True, open_popup_on_hover=False, **kwargs*)

Bases: JSCSSMixin, MacroElement

Creates a Tag Filter Button to filter elements based on criteria (maydemirx/leaflet-tag-filter-button)

This plugin works for multiple element types like Marker, GeoJson and vector layers like PolyLine.

Parameters:

- **data** (*list, of strings.*) – The tags to filter for this filter button.
- **icon** (*string, default 'fa-filter'*) – The icon for the filter button
- **clear_text** (*string, default 'clear'*) – Text of the clear button
- **filter_on_every_click** (*bool, default True*) – if True, the plugin will filter on every click event on checkbox.
- **open_popup_on_hover** (*bool, default False*) – if True, popup that contains tags will be open at mouse hover time

default_css: List[Tuple[str, str]]

default_js: List[Tuple[str, str]]

class folium.plugins.**Terminator**

Bases: JSCSSMixin, MacroElement

Leaflet.Terminator is a simple plug-in to the Leaflet library to overlay day and night regions on maps.

default_js: List[Tuple[str, str]]

class folium.plugins.**TimeSliderChoropleth**(*data, styledict, highlight: bool = False, name=None, overlay=True, control=True, show=True, init_timestamp=0, stroke_opacity=1, stroke_width=0.8, stroke_color='#FFFFFF'*)

Bases: JSCSSMixin, Layer

Create a choropleth with a timeslider for timestamped data.

Visualize timestamped data, allowing users to view the choropleth at different timestamps using a slider.

↑ Back to top

s:

- **data** (*str*) – geojson string
- **styledict** (*dict*) – A dictionary where the keys are the geojson feature ids and the values are dicts of *{time: style_options_dict}*
- **highlight** (*bool, default False*) – Whether to show a visual effect on mouse hover and click.
- **name** (*string, default None*) – The name of the Layer, as it will appear in LayerControls.
- **overlay** (*bool, default False*) – Adds the layer as an optional overlay (True) or the base layer (False).
- **control** (*bool, default True*) – Whether the Layer will be included in LayerControls.
- **show** (*bool, default True*) – Whether the layer will be shown on opening.
- **init_timestamp** (*int, default 0*) – Initial time-stamp index on the slider. Must be in the range *[-L, L-1]*, where *L* is the maximum number of time stamps in *styledict*. For example, use *-1* to initialize the slider to the latest timestamp.

default_js: List[Tuple[str, str]]

class folium.plugins.**Timeline**(*data: dict | str | TextIO, get_interval: JsCode | None = None, **kwargs*)

Bases: GeoJson

Create a layer from GeoJSON with time data to add to a map.

To add time data, you need to do one of the following:

- Add a 'start' and 'end' property to each feature. The start and end can be any comparable item.

Alternatively, you can provide a *get_interval* function.

- This function should be a JsCode object and take as parameter a GeoJson feature and return a dict containing values for 'start', 'end', 'startExclusive' and 'endExcusive' (or false if no data could be extracted from the feature).
- 'start' and 'end' can be any comparable items
- 'startExclusive' and 'endExclusive' should be boolean values.

Parameters:

- **data** (*file, dict or str.*) – The geojson data you want to plot.
- **get_interval** (*JsCode, optional*) –

  Called for each feature, and should return either a time range for the feature or *false*, indicating that it should not be included in the timeline. The time range object should have 'start' and 'end' properties. Optionally, the boolean keys 'startExclusive' and 'endExclusive' allow the interval to be considered exclusive.
  If *get_interval* is not provided, 'start' and 'end' properties are assumed to be present on each feature.

Examples

```
>>> from folium.plugins import Timeline, TimelineSlider
>>> m = folium.Map()
```

```
>>> data = requests.get(
...     "https://earthquake.usgs.gov/earthquakes/feed/v1.0/summary/all_day.geojson"
... ).json()
```

```
>>> timeline = Timeline(
...     data,
...     get_interval=JsCode(
...         '''
...         function (quake) {
...            // earthquake data only has a time, so we'll use that as a "start"
...            // and the "end" will be that + some value based on magnitude
...            // 18000000 = 30 minutes, so a quake of magnitude 5 would show on the
...            // map for 150 minutes or 2.5 hours
...            return {
...                start: quake.properties.time,
...                end: quake.properties.time + quake.properties.mag * 1800000,
...            };
...         };
...     '''
...     ),
... ).add_to(m)
>>> TimelineSlider(
...     auto_play=False,
...     show_ticks=True,
...     enable_keyboard_controls=True,
...     playback_duration=30000,
... ).add_timelines(timeline).add_to(m)
```

Other keyword arguments are passed to the GeoJson layer, so you can pass

style, point_to_layer and/or *on_each_feature*.

default_js

class **folium.plugins.TimelineSlider(*auto_play: bool = True, date_options: str = 'YYYY-MM-DD HH:mm:ss', start: str | int | float | None = None, end: str | int | float | None = None, enable_playback: bool = True, enable_keyboard_controls: bool = False, show_ticks: bool = True, steps: int = 1000, playback_duration: int = 10000, **kwargs*)**

Bases: `JSCSSMixin`, `MacroElement`

Creates a timeline slider for timeline layers.

Parameters:

- **auto_play** (*bool, default True*) – Whether the animation shall start automatically at startup.
- **start** (*str, int or float, default earliest 'start' in GeoJson*) – The beginning/minimum value of the timeline.
- **end** (*str, int or float, default latest 'end' in GeoJSON*) – The end/maximum value of the timeline.
- **date_options** (*str, default "YYYY-MM-DD HH:mm:ss"*) – A format string to render the currently active time in the control.
- **enable_playback** (*bool, default True*) – Show playback controls (i.e. prev/play/pause/next).
- **enable_keyboard_controls** (*bool, default False*) – Allow playback to be controlled using the spacebar (play/pause) and right/left arrow keys (next/previous).
- **show_ticks** (*bool, default True*) – Show tick marks on the slider
- **steps** (*int, default 1000*) – How many steps to break the timeline into. Each step will then be (end-start) / steps. Only affects playback.
- **playback_duration** (*int, default 10000*) – Minimum time, in ms, for the playback to take. Will almost certainly actually take at least a bit longer – after each frame, the next one displays in playback_duration/steps ms, so each frame really takes frame processing time PLUS step time.

Examples

See the documentation for Timeline

add_timelines(args*)**

Add timelines to the control

default_js: *List[Tuple[str, str]]*

render(*kwargs*)**

Renders the HTML representation of the element.

class **folium.plugins.TimestampedGeoJson(*data, transition_time=200, loop=True, auto_play=True, add_last_point=True, period='P1D', min_speed=0.1, max_speed=10, loop_button=False, date_options='YYYY-MM-DD HH:mm:ss', time_slider_drag_update=False, duration=None, speed_slider=True*)**

Bases: `JSCSSMixin`, `MacroElement`

Creates a TimestampedGeoJson plugin from timestamped GeoJSONs to append into a map with Map.add_child.

A geo-json is timestamped if:

- it contains only features of types LineString, MultiPoint, MultiLineString, Polygon and MultiPolygon.
- each feature has a 'times' property with the same length as the coordinates array.
- each element of each 'times' property is a timestamp in ms since epoch, or in ISO string.

Eventually, you may have Point features with a 'times' property being an array of length 1.

↑ Back to top

Parameters:

- **data** (*file, dict or str.*) –

  The timestamped geo-json data you want to plot.

  - If file, then data will be read in the file and fully embedded in Leaflet's javascript.
  - If dict, then data will be converted to json and embedded in the javascript.
  - If str, then data will be passed to the javascript as-is.
- **transition_time** (*int, default 200.*) – The duration in ms of a transition from between timestamps.
- **loop** (*bool, default True*) – Whether the animation shall loop.
- **auto_play** (*bool, default True*) – Whether the animation shall start automatically at startup.
- **add_last_point** (*bool, default True*) – Whether a point is added at the last valid coordinate of a LineString.
- **period** (*str, default "P1D"*) – Used to construct the array of available times starting from the first available time. Format: ISO8601 Duration ex: 'P1M' 1/month, 'P1D' 1/day, 'PT1H' 1/hour, and 'PT1M' 1/minute
- **duration** (*str, default None*) – Period of time which the features will be shown on the map after their time has passed. If None, all previous times will be shown. Format: ISO8601 Duration ex: 'P1M' 1/month, 'P1D' 1/day, 'PT1H' 1/hour, and 'PT1M' 1/minute

Examples

```
>>> TimestampedGeoJson(
...     {
...         "type": "FeatureCollection",
...         "features": [
...             {
...                 "type": "Feature",
...                 "geometry": {
...                     "type": "LineString",
...                     "coordinates": [[-70, -25], [-70, 35], [70, 35]],
...                 },
...                 "properties": {
...                     "times": [1435708800000, 1435795200000, 1435881600000],
...                     "tooltip": "my tooltip text",
...                 },
...             }
...         ],
...     }
... )
```

See socib/Leaflet.TimeDimension for more information.

default_css: *List[Tuple[str, str]]*

↑ Back to top

default_js: *List[Tuple[str, str]]*

render(***kwargs*)

Renders the HTML representation of the element.

class **folium.plugins.TimestampedWmsTileLayers**(*data*, *transition_time=200*, *loop=False*, *auto_play=False*, *period='P1D'*, *time_interval=False*)

Bases: `JSCSSMixin`, `MacroElement`

Creates a TimestampedWmsTileLayer that takes a WmsTileLayer and adds time control with the Leaflet.TimeDimension plugin.

Parameters:

- **data** (*WmsTileLayer.*) – The WmsTileLayer that you want to add time support to. Must be created like a typical WmsTileLayer and added to the map before being passed to this class.
- **transition_time** (*int, default 200.*) – The duration in ms of a transition from between timestamps.
- **loop** (*bool, default False*) – Whether the animation shall loop, default is to reduce load on WMS services.
- **auto_play** (*bool, default False*) – Whether the animation shall start automatically at startup, default is to reduce load on WMS services.
- **period** (*str, default 'P1D'*) – Used to construct the array of available times starting from the first available time. Format: ISO8601 Duration ex: 'P1M' -> 1/month, 'P1D' -> 1/day, 'PT1H' -> 1/hour, and 'PT1M' -> 1/minute Note: this seems to be overridden by the WMS Tile Layer GetCapabilities.

Examples

```
>>> w0 = WmsTileLayer(
...     "http://this.wms.server/ncWMS/wms",
...     name="Test WMS Data",
...     styles="",
...     fmt="image/png",
...     transparent=True,
...     layers="test_data",
...     COLORSCALERANGE="0,10",
... )
>>> w0.add_to(m)
>>> w1 = WmsTileLayer(
...     "http://this.wms.server/ncWMS/wms",
...     name="Test WMS Data",
...     styles="",
...     fmt="image/png",
...     transparent=True,
...     layers="test_data_2",
...     COLORSCALERANGE="0,5",
... )
>>> w1.add_to(m)
>>> # Add WmsTileLayers to time control.
>>> time = TimestampedWmsTileLayers([w0, w1])
```

```
>>> time.add_to(m)
```

See socib/Leaflet.TimeDimension for more information.

default_css: *List[Tuple[str, str]]*

default_js: *List[Tuple[str, str]]*

class **folium.plugins.TreeLayerControl**(*base_tree: dict | list | None = None*, *overlay_tree: dict | list | None = None*, *closed_symbol: str = '+'*, *opened_symbol: str = '-'*, *space_symbol: str = ' '*, *selector_back: bool = False*, *named_toggle: bool = False*, *collapse_all: str = ''*, *expand_all: str = ''*, *label_is_selector: str = 'both'*, ***kwargs*)

Bases: `JSCSSMixin`, `MacroElement`

Create a Layer Control allowing a tree structure for the layers. See jjimenezshaw/Leaflet.Control.Layers.Tree for more information.

Parameters:

- **base_tree** (*dict*) –

  A dictionary defining the base layers. Valid elements are

  children: list

  Array of child nodes for this node. Each node is a dict that has the same valid elements as base_tree.

  label: str

  Text displayed in the tree for this node. It may contain HTML code.

  layer: Layer

  The layer itself. This needs to be added to the map.

  name: str

  Text displayed in the toggle when control is minimized. If not present, label is used. It makes sense only when namedToggle is true, and with base layers.

  radioGroup: str, default ''

  Text to identify different radio button groups. It is used in the name attribute in the radio button. It is used only in the overlays layers (ignored in the base layers), allowing you to have radio buttons instead of checkboxes. See that radio groups cannot be unselected, so create a 'fake' layer (like L.layersGroup([])) if you want to disable it. Default '' (that means checkbox).

  collapsed: bool, default False

  Indicate whether this tree node should be collapsed initially, useful for opening large trees partially based on user input or context.

  selectAllCheckbox: bool or str

  Displays a checkbox to select/unselect all overlays in the sub-tree. In case of being a <str>, that text will be the title (tooltip). When any overlay in the sub-tree is clicked, the checkbox goes into indeterminate state (a dash in the box).

- **overlay_tree** (*dict*) – Similar to baseTree, but for overlays.
- **closed_symbol** (*str, default '+',*) – Symbol displayed on a closed node (that you can click to open).
- **opened_symbol** (*str, default '-',*) – Symbol displayed on an opened node (that you can click to close).
- **space_symbol** (*str, default ' ',*) – Symbol between the closed or opened symbol, and the text.
- **selector_back** (*bool, default False,*) – Flag to indicate if the selector (+ or −) is after the text.
- **named_toggle** (*bool, default False,*) – Flag to replace the toggle image (box with the layers image) with the 'name' of the selected base layer. If the name field is not present in the tree for this layer, label is used. See that you can show a different name when control is collapsed than the one that appears in the tree when it is expanded.
- **collapse_all** (*str, default '',*) – Text for an entry in control that collapses the tree (baselayers or overlays). If empty, no entry is created.
- **expand_all** (*str, default '',*) – Text for an entry in control that expands the tree. If empty, no entry is created
- **label_is_selector** (*str, default 'both',*) – Controls if a label or only the checkbox/radiobutton can toggle layers. If set to *both*, *overlay* or *base* those labels can be clicked on to toggle the layer.
- ****kwargs** – Additional (possibly inherited) options. See https://leafletjs.com/reference.html#control-layers

Examples

```
>>> import folium
>>> from folium.plugins.treelayercontrol import TreeLayerControl
>>> from folium.features import Marker
```

```
>>> m = folium.Map(location=[46.603354, 1.8883335], zoom_start=5)
```

```
>>> marker = Marker([48.8582441, 2.2944775]).add_to(m)
```

```
>>> overlay_tree = {
...     "label": "Points of Interest",
...     "selectAllCheckbox": "Un/select all",
...     "children": [
...         {
...             "label": "Europe",
...             "selectAllCheckbox": True,
...             "children": [
...                 {
```

↑ Back to top

```
...                     "label": "France",
...                     "selectAllCheckbox": True,
...                     "children": [
...                         {"label": "Tour Eiffel", "layer": marker},
...                     ],
...                 }
...             ],
...         }
...     ],
... }
```

```
>>> control = TreeLayerControl(overlay_tree=overlay_tree).add_to(m)
```

default_css: *List[Tuple[str, str]]*

default_js: *List[Tuple[str, str]]*

class **folium.plugins.VectorGridProtobuf**(***url: str***, ***name: str | None*** = *None*, ***options: str | dict | None*** = *None*, ***overlay: bool*** = *True*, ***control: bool*** = *True*, ***show: bool*** = *True*)

Bases: `JSCSSMixin`, `Layer`

Add vector tile layers based on Leaflet/Leaflet.VectorGrid.

Parameters:

- **url** (*str*) – url to tile provider e.g. https://free-{s}.tilehosting.com/data/v3/{z}/{x}/{y}.pbf?token={token}
- **name** (*str, optional*) – Name of the layer that will be displayed in LayerControl
- **options** (*dict or str, optional*) –
  VectorGrid.protobuf options, which you can pass as python dictionary or string. Strings allow plain JavaScript to be passed, therefore allow for conditional styling (see examples).
  Additionally the url might contain any string literals like {token}, or {key} that can be passed as attribute to the options dict and will be substituted.
  Every layer inside the tile layer has to be styled separately.
- **overlay** (*bool, default True*) – Whether your layer will be an overlay (ticked with a check box in LayerControls) or a base layer (ticked with a radio button).
- **control** (*bool, default True*) – Whether the layer will be included in LayerControls.
- **show** (*bool, default True*) – Whether the layer will be shown on opening.

↑ Back to top

Examples

Options as dict:

```
>>> m = folium.Map()
>>> url = "https://free-{s}.tilehosting.com/data/v3/{z}/{x}/{y}.pbf?token={token}"
>>> options = {
...     "subdomain": "tilehosting",
...     "token": "af6P2G9dztAt1F75x7KYt0Hx2DJR052G",
...     "vectorTileLayerStyles": {
...         "layer_name_one": {
...             "fill": True,
...             "weight": 1,
...             "fillColor": "green",
...             "color": "black",
...             "fillOpacity": 0.6,
...             "opacity": 0.6,
...         },
...         "layer_name_two": {
...             "fill": True,
...             "weight": 1,
...             "fillColor": "red",
...             "color": "black",
...             "fillOpacity": 0.6,
...             "opacity": 0.6,
...         },
...     },
... }
```

```
>>> VectorGridProtobuf(url, "layer_name", options).add_to(m)
```

Options as string allows to pass functions

```
>>> m = folium.Map()
>>> url = "https://free-{s}.tilehosting.com/data/v3/{z}/{x}/{y}.pbf?token={token}"
>>> options = '''{
...     "subdomain": "tilehosting",
...     "token": "af6P2G9dztAt1F75x7KYt0Hx2DJR052G",
...     "vectorTileLayerStyles": {
...         all: function(f) {
...             if (f.type === 'parks') {
...                 return {
...                     "fill": true,
...                     "weight": 1,
...                     "fillColor": 'green',
...                     "color": 'black',
...                     "fillOpacity":0.6,
...                     "opacity":0.6
...                 };
...             }
...             if (f.type === 'water') {
...                 return {
...                     "fill": true,
...                     "weight": 1,
...                     "fillColor": 'purple',
...                     "color": 'black',
...                     "fillOpacity":0.6,
...                     "opacity":0.6
```

```
...                 };
...             }
...         }
...     }
... }'''
```

```
>>> VectorGridProtobuf(url, "layer_name", options).add_to(m)
```

For more info, see: https://leaflet.github.io/Leaflet.VectorGrid/vectorgrid-api-docs.html#styling-vectorgrids.

default_js: *List[Tuple[str, str]]*

Folium version 0.16.1.dev76+g2921126e

© Copyright 2013-2024, Rob Story.

Created using Sphinx 7.3.7.

Built with the PyData Sphinx Theme 0.15.3.

Built with the PyData Sphinx Theme 0.15.3.

↑ Back to top