OSMnx Documentation

Release 1.0.1

Geoff Boeing

Jan 22, 2021

CONTENTS

1 Installation **3**

2 Usage **5**

3 User reference **7**
- 3.1 User reference 7
- 3.2 Internals reference 42

4 Support **95**

5 License **97**

6 Indices **99**

Python Module Index **101**

Index **103**

OSMnx is a Python package that lets you download spatial data from OpenStreetMap and model, project, visualize, and analyze real-world street networks. You can download and model walkable, drivable, or bikeable urban networks with a single line of Python code then easily analyze and visualize them. You can just as easily download and work with other infrastructure types, amenities/points of interest, building footprints, elevation data, street bearings/orientations, and speed/travel time.

If you use OSMnx in your work, please cite the journal article:

Boeing, G. 2017. OSMnx: New Methods for Acquiring, Constructing, Analyzing, and Visualizing Complex Street Networks. *Computers, Environment and Urban Systems* 65, 126-139. doi:10.1016/j.compenvurbsys.2017.05.004

CHAPTER

ONE

INSTALLATION

You can install OSMnx with conda:

```
conda config --prepend channels conda-forge
conda create -n ox --strict-channel-priority osmnx
```

If you want other packages, such as `jupyterlab`, installed in this environment as well, just add their names after `osmnx` above. See the conda documentation for further details. To upgrade OSMnx to a newer release, just remove the conda environment you created and then create a new one again, as above. Don't just run "conda update" or you could get package conflicts.

You can also run OSMnx + Jupyter directly from its official Docker container, or you can install OSMnx via pip if you already have all of its dependencies installed and fully tested on your system. Note: installing the dependencies with pip is nontrivial. If you don't know *exactly* what you're doing, just use conda as described above.

CHAPTER

TWO

USAGE

To get started with sample code and usage examples/demos, see the examples GitHub repo and read the user reference.

OSMnx is built on top of GeoPandas, NetworkX, and matplotlib and interacts with OpenStreetMap's APIs to:

- Download and model street networks or other networked infrastructure anywhere in the world with a single line of code
- Download any other spatial geometries, place boundaries, building footprints, or points of interest as a GeoDataFrame
- Download by city name, polygon, bounding box, or point/address + network distance
- Download drivable, walkable, bikeable, or all street networks
- Download node elevations and calculate edge grades (inclines)
- Impute missing speeds and calculate graph edge travel times
- Simplify and correct the network's topology to clean-up nodes and consolidate intersections
- Fast map-matching of points, routes, or trajectories to nearest graph edges or nodes
- Save networks to disk as shapefiles, GeoPackages, and GraphML
- Save/load street network to/from a local .osm XML file
- Conduct topological and spatial analyses to automatically calculate dozens of indicators
- Calculate and visualize street bearings and orientations
- Calculate and visualize shortest-path routes that minimize distance, travel time, elevation, etc
- Visualize street networks as a static map or interactive Leaflet web map
- Visualize travel distance and travel time with isoline and isochrone maps
- Plot figure-ground diagrams of street networks and building footprints

OSMnx geocodes place names and addresses with the OpenStreetMap Nominatim API. Using OSMnx's `geometries` module, you can retrieve any geospatial objects (such as building footprints, grocery stores, schools, public parks, transit stops, etc) from the OpenStreetMap Overpass API as a GeoPandas GeoDataFrame. Using OSMnx's `graph` module, you can retrieve any spatial network data (such as streets, paths, canals, etc) from the Overpass API and model them as NetworkX MultiDiGraphs.

OSMnx automatically processes network topology from the original raw OpenStreetMap data such that nodes represent intersections/dead-ends and edges represent the street segments that link them. MultiDiGraphs are nonplanar directed graphs with possible self-loops and parallel edges. Thus, a one-way street will be represented with a single directed edge from node u to node v, but a bidirectional street will be represented with two reciprocal directed edges (with identical geometries): one from node u to node v and another from v to u, to represent both possible directions of

flow. OSMnx can convert a MultiDiGraph to a MultiGraph if you prefer an undirected representation of the network. It can also convert a MultiDiGraph to/from GeoPandas node and edge GeoDataFrames.

Usage examples and demonstrations of these features are in the examples GitHub repo. More feature development details are in the change log. Read the journal article for further technical details. Package usage is detailed in the user reference.

CHAPTER

THREE

USER REFERENCE

3.1 User reference

User reference for the OSMnx package.

This guide covers usage of all public modules and functions. Every function can be accessed via *ox.module_name.function_name()* and the vast majority of them can also be accessed directly via *ox.function_name()* as a shortcut. Only a few less-common functions are accessible only via *ox.module_name.function_name()*.

3.1.1 osmnx.bearing module

Calculate graph edge bearings.

osmnx.bearing.**add_edge_bearings**(*G*, *precision=1*)

Add *bearing* attributes to all graph edges.

Calculate the compass bearing from origin node to destination node for each edge in the directed graph then add each bearing as a new edge attribute. Bearing represents angle in degrees (clockwise) between north and the direction from the origin node to the destination node.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **precision** (*int*) – decimal precision to round bearing

Returns **G** – graph with edge bearing attributes

Return type networkx.MultiDiGraph

osmnx.bearing.**get_bearing**(*origin_point*, *destination_point*)

Calculate the bearing between two lat-lng points.

Each argument tuple should represent (lat, lng) as decimal degrees. Bearing represents angle in degrees (clockwise) between north and the direction from the origin point to the destination point.

Parameters

- **origin_point** (*tuple*) – (lat, lng)
- **destination_point** (*tuple*) – (lat, lng)

Returns **bearing** – the compass bearing in decimal degrees from the origin point to the destination point

Return type float

3.1.2 osmnx.distance module

Calculate distances and shortest paths and find nearest node/edge(s) to point(s).

osmnx.distance.**euclidean_dist_vec**(*y1*, *x1*, *y2*, *x2*)
Calculate Euclidean distances between points.

Vectorized function to calculate the Euclidean distance between two points' coordinates or between arrays of points' coordinates. For most accurate results, use projected coordinates rather than decimal degrees.

Parameters

- **y1** (*float or np.array of float*) – first point's y coordinate
- **x1** (*float or np.array of float*) – first point's x coordinate
- **y2** (*float or np.array of float*) – second point's y coordinate
- **x2** (*float or np.array of float*) – second point's x coordinate

Returns **dist** – distance or array of distances from (x1, y1) to (x2, y2) in coordinates' units

Return type float or np.array of float

osmnx.distance.**get_nearest_edge**(*G*, *point*, *return_geom=False*, *return_dist=False*)
Find the nearest edge to a point by minimum Euclidean distance.

For best results, both G and point should be projected.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **point** (*tuple*) – the (lat, lng) or (y, x) point for which we will find the nearest edge in the graph
- **return_geom** (*bool*) – Optionally return the geometry of the nearest edge
- **return_dist** (*bool*) – Optionally return the distance in graph's coordinates' units between the point and the nearest edge

Returns Graph edge unique identifier as a tuple of (u, v, key). Or a tuple of (u, v, key, geom) if return_geom is True. Or a tuple of (u, v, key, dist) if return_dist is True. Or a tuple of (u, v, key, geom, dist) if return_geom and return_dist are True.

Return type tuple

osmnx.distance.**get_nearest_edges**(*G*, *X*, *Y*, *method=None*, *dist=0.0001*)
Find the nearest edge to each point in a list of points.

Pass in points as separate lists of X and Y coordinates. The 'kdtree' method is by far the fastest with large data sets, but only finds approximate nearest edges if working in unprojected coordinates like lat-lng (it precisely finds the nearest edge if working in projected coordinates). The 'balltree' method is second fastest with large data sets, but it is precise if working in unprojected coordinates like lat-lng. As a rule of thumb, if you have a small graph just use method=None. If you have a large graph with lat-lng coordinates, use method='balltree'. If you have a large graph with projected coordinates, use method='kdtree'. Note that if you are working in units of lat-lng, the X vector corresponds to longitude and the Y vector corresponds to latitude. The method creates equally distanced points along the edges of the network. Then, these points are used in a kdTree or BallTree search to identify which is nearest. Note that this method will not give exact perpendicular point along the edge, but the smaller the *dist* parameter, the closer (but slower) the solution will be.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph

- **X** (*list-like*) – the longitudes or x coordinates for which we will find the nearest edge in the graph. For projected graphs use the projected coordinates, usually in meters.
- **Y** (*list-like*) – the latitudes or y coordinates for which we will find the nearest edge in the graph. For projected graphs use the projected coordinates, usually in meters.
- **method** (*string {None, 'kdtree', 'balltree'}*) – Which method to use for finding nearest edge to each point. If None, we manually find each edge one at a time using get_nearest_edge. If 'kdtree' we use scipy.spatial.cKDTree for very fast euclidean search. Recommended for projected graphs. If 'balltree', we use sklearn.neighbors.BallTree for fast haversine search. Recommended for unprojected graphs.
- **dist** (*float*) – spacing length along edges. Units are the same as the graph's geometries. The smaller the value, the more points are created.

Returns **ne** – array of edge IDs representing the edge nearest to each point in the passed-in list of points. Edge IDs are represented by u, v, key where u and v the node IDs of the nodes the edge links.

Return type np.array

osmnx.distance.**get_nearest_node**(*G*, *point*, *method='haversine'*, *return_dist=False*)

Find the nearest node to a point.

Return the graph node nearest to some (lat, lng) or (y, x) point and optionally the distance between the node and the point. This function can use either the haversine formula or Euclidean distance.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **point** (*tuple*) – The (lat, lng) or (y, x) point for which we will find the nearest node in the graph
- **method** (*string {'haversine', 'euclidean'}*) – Which method to use for calculating distances to find nearest node. If 'haversine', graph nodes' coordinates must be in units of decimal degrees. If 'euclidean', graph nodes' coordinates must be projected.
- **return_dist** (*bool*) – Optionally also return the distance (in meters if haversine, or graph node coordinate units if euclidean) between the point and the nearest node

Returns Nearest node ID or optionally a tuple of (node ID, dist), where dist is the distance (in meters if haversine, or graph node coordinate units if euclidean) between the point and nearest node

Return type int or tuple of (int, float)

osmnx.distance.**get_nearest_nodes**(*G*, *X*, *Y*, *method=None*)

Find the nearest node to each point in a list of points.

Pass in points as separate lists of X and Y coordinates. The 'kdtree' method is by far the fastest with large data sets, but only finds approximate nearest nodes if working in unprojected coordinates like lat-lng (it precisely finds the nearest node if working in projected coordinates). The 'balltree' method is second fastest with large data sets but it is precise if working in unprojected coordinates like lat-lng.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **X** (*list-like*) – the longitudes or x coordinates for which we will find the nearest node in the graph
- **Y** (*list-like*) – the latitudes or y coordinates for which we will find the nearest node in the graph

- **method** (*string {None, 'kdtree', 'balltree'}*) – Which method to use for finding the nearest node to each point. If None, we manually find each node one at a time using utils.get_nearest_node and haversine. If 'kdtree' we use scipy.spatial.cKDTree for very fast euclidean search. If 'balltree', we use sklearn.neighbors.BallTree for fast haversine search.

Returns **nn** – array of node IDs representing the node nearest to each point in the passed-in list of points

Return type np.array

osmnx.distance.**great_circle_vec**(*lat1*, *lng1*, *lat2*, *lng2*, *earth_radius=6371009*)

Calculate great-circle distances between points.

Vectorized function to calculate the great-circle distance between two points' coordinates or between arrays of points' coordinates using the haversine formula. Expects coordinates in decimal degrees.

Parameters

- **lat1** (*float or np.array of float*) – first point's latitude coordinate
- **lng1** (*float or np.array of float*) – first point's longitude coordinate
- **lat2** (*float or np.array of float*) – second point's latitude coordinate
- **lng2** (*float or np.array of float*) – second point's longitude coordinate
- **earth_radius** (*int or float*) – radius of earth in units in which distance will be returned (default is meters)

Returns **dist** – distance or array of distances from (lat1, lng1) to (lat2, lng2) in units of earth_radius

Return type float or np.array

osmnx.distance.**k_shortest_paths**(*G*, *orig*, *dest*, *k*, *weight='length'*)

Get k shortest paths from origin node to destination node.

See also *shortest_path* to get just the one shortest path.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **orig** (*int*) – origin node ID
- **dest** (*int*) – destination node ID
- **k** (*int*) – number of shortest paths to get
- **weight** (*string*) – edge attribute to minimize when solving shortest paths. default is edge length in meters.

Returns a generator of k shortest paths ordered by total weight. each path is a list of node IDs.

Return type generator

osmnx.distance.**shortest_path**(*G*, *orig*, *dest*, *weight='length'*)

Get shortest path from origin node to destination node.

See also *k_shortest_paths* to get multiple shortest paths.

This function is a convenience wrapper around networkx.shortest_path. For more functionality or different algorithms, use networkx directly.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph

- **orig** (*int*) – origin node ID
- **dest** (*int*) – destination node ID
- **weight** (*string*) – edge attribute to minimize when solving shortest path. default is edge length in meters.

Returns **path** – list of node IDs consituting the shortest path

Return type list

3.1.3 osmnx.downloader module

Interact with the OSM APIs.

osmnx.downloader.**nominatim_request**(*params*, *request_type='search'*, *pause=1*, *error_pause=60*)

Send a HTTP GET request to the Nominatim API and return JSON response.

Parameters

- **params** (*OrderedDict*) – key-value pairs of parameters
- **request_type** (*string {"search", "reverse", "lookup"}*) – which Nominatim API endpoint to query
- **pause** (*int*) – how long to pause before request, in seconds. per the nominatim usage policy: "an absolute maximum of 1 request per second" is allowed
- **error_pause** (*int*) – how long to pause in seconds before re-trying request if error

Returns **response_json**

Return type dict

osmnx.downloader.**overpass_request**(*data*, *pause=None*, *error_pause=60*)

Send a HTTP POST request to the Overpass API and return JSON response.

Parameters

- **data** (*OrderedDict*) – key-value pairs of parameters
- **pause** (*int*) – how long to pause in seconds before request, if None, will query API status endpoint to find when next slot is available
- **error_pause** (*int*) – how long to pause in seconds (in addition to *pause*) before re-trying request if error

Returns **response_json**

Return type dict

3.1.4 osmnx.elevation module

Get node elevations and calculate edge grades.

osmnx.elevation.**add_edge_grades**(*G*, *add_absolute=True*, *precision=3*)

Add *grade* attribute to each graph edge.

Get the directed grade (ie, rise over run) for each edge in the graph and add it to the edge as an attribute. Nodes must have *elevation* attributes to use this function.

See also the *add_node_elevations* function.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **add_absolute** (*bool*) – if True, also add absolute value of grade as *grade_abs* attribute
- **precision** (*int*) – decimal precision to round grade values

Returns G – graph with edge *grade* (and optionally *grade_abs*) attributes

Return type networkx.MultiDiGraph

osmnx.elevation.**add_node_elevations**(*G*, *api_key*, *max_locations_per_batch=350*, *pause_duration=0.02*, *precision=3*)

Add *elevation* (meters) attribute to each node.

Uses the Google Maps Elevation API by default, but you can configure this to a different provider via ox.config()

See also the *add_edge_grades* function.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **api_key** (*string*) – your google maps elevation API key, or equivalent if using a different provider
- **max_locations_per_batch** (*int*) – max number of coordinate pairs to submit in each API call (if this is too high, the server will reject the request because its character limit exceeds the max)
- **pause_duration** (*float*) – time to pause between API calls
- **precision** (*int*) – decimal precision to round elevation

Returns G – graph with node elevation attributes

Return type networkx.MultiDiGraph

3.1.5 osmnx.folium module

Create interactive Leaflet web maps of graphs and routes via folium.

osmnx.folium.**plot_graph_folium**(*G*, *graph_map=None*, *popup_attribute=None*, *tiles='cartodbpositron'*, *zoom=1*, *fit_bounds=True*, *edge_color=None*, *edge_width=None*, *edge_opacity=None*, ***kwargs*)

Plot a graph as an interactive Leaflet web map.

Note that anything larger than a small city can produce a large web map file that is slow to render in your browser.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **graph_map** (*folium.folium.Map*) – if not None, plot the graph on this preexisting folium map object
- **popup_attribute** (*string*) – edge attribute to display in a pop-up when an edge is clicked
- **tiles** (*string*) – name of a folium tileset
- **zoom** (*int*) – initial zoom level for the map

- **fit_bounds** (*bool*) – if True, fit the map to the boundaries of the graph's edges
- **edge_color** (*string*) – deprecated, do not use, use kwargs instead
- **edge_width** (*numeric*) – deprecated, do not use, use kwargs instead
- **edge_opacity** (*numeric*) – deprecated, do not use, use kwargs instead
- **kwargs** – keyword arguments to pass to folium.PolyLine(), see folium docs for options (for example *color="#333333", weight=5, opacity=0.7*)

Returns

Return type folium.folium.Map

osmnx.folium.**plot_route_folium**(*G*, *route*, *route_map=None*, *popup_attribute=None*, *tiles='cartodbpositron'*, *zoom=1*, *fit_bounds=True*, *route_color=None*, *route_width=None*, *route_opacity=None*, ***kwargs*)

Plot a route as an interactive Leaflet web map.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **route** (*list*) – the route as a list of nodes
- **route_map** (*folium.folium.Map*) – if not None, plot the route on this preexisting folium map object
- **popup_attribute** (*string*) – edge attribute to display in a pop-up when an edge is clicked
- **tiles** (*string*) – name of a folium tileset
- **zoom** (*int*) – initial zoom level for the map
- **fit_bounds** (*bool*) – if True, fit the map to the boundaries of the route's edges
- **route_color** (*string*) – deprecated, do not use, use kwargs instead
- **route_width** (*numeric*) – deprecated, do not use, use kwargs instead
- **route_opacity** (*numeric*) – deprecated, do not use, use kwargs instead
- **kwargs** – keyword arguments to pass to folium.PolyLine(), see folium docs for options (for example *color="#cc0000", weight=5, opacity=0.7*)

Returns

Return type folium.folium.Map

3.1.6 osmnx.geocoder module

Geocode queries and create GeoDataFrames of place boundaries.

osmnx.geocoder.**geocode**(*query*)

Geocode a query string to (lat, lng) with the Nominatim geocoder.

Parameters **query** (*string*) – the query string to geocode

Returns **point** – the (lat, lng) coordinates returned by the geocoder

Return type tuple

osmnx.geocoder.**geocode_to_gdf**(*query*, *which_result=None*, *by_osmid=False*, *buffer_dist=None*)

Retrieve place(s) by name or ID from the Nominatim API as a GeoDataFrame.

You can query by place name or OSM ID. If querying by place name, the query argument can be a string or structured dict, or a list of such strings/dicts to send to geocoder. You can instead query by OSM ID by setting *by_osmid=True*. In this case, geocode_to_gdf treats the query argument as an OSM ID (or list of OSM IDs) for Nominatim lookup rather than text search. OSM IDs must be prepended with their types: node (N), way (W), or relation (R), in accordance with the Nominatim format. For example, *query=["R2192363", "N240109189", "W427818536"]*.

If query argument is a list, then which_result should be either a single value or a list with the same length as query. The queries you provide must be resolvable to places in the Nominatim database. The resulting GeoDataFrame's geometry column contains place boundaries if they exist in OpenStreetMap.

Parameters

- **query** (*string or dict or list*) – query string(s) or structured dict(s) to geocode
- **which_result** (*int*) – which geocoding result to use. if None, auto-select the first (Multi)Polygon or raise an error if OSM doesn't return one. to get the top match regardless of geometry type, set which_result=1
- **by_osmid** (*bool*) – if True, handle query as an OSM ID for lookup rather than text search
- **buffer_dist** (*float*) – distance to buffer around the place geometry, in meters

Returns **gdf** – a GeoDataFrame with one row for each query

Return type geopandas.GeoDataFrame

3.1.7 osmnx.geometries module

Download geospatial entities' geometries and attributes from OpenStreetMap.

Retrieve points of interest, building footprints, or any other objects from OSM, including their geometries and attribute data, and construct a GeoDataFrame of them.

osmnx.geometries.**geometries_from_address**(*address*, *tags*, *dist=1000*)

Create GeoDataFrame of OSM entities within some distance N, S, E, W of address.

Parameters

- **address** (*string*) – the address to geocode and use as the central point around which to get the geometries
- **tags** (*dict*) – Dict of tags used for finding objects in the selected area. Results returned are the union, not intersection of each individual tag. Each result matches at least one given tag. The dict keys should be OSM tags, (e.g., *building*, *landuse*, *highway*, etc) and the dict values should be either *True* to retrieve all items with the given tag, or a string to get a single tag-value combination, or a list of strings to get multiple values for the given tag. For example, *tags = {'building': True}* would return all building footprints in the area. *tags = {'amenity':True, 'landuse':['retail','commercial'], 'highway':'bus_stop'}* would return all amenities, landuse=retail, landuse=commercial, and highway=bus_stop.
- **dist** (*numeric*) – distance in meters

Returns **gdf**

Return type geopandas.GeoDataFrame

Notes

You can configure the Overpass server timeout, memory allocation, and other custom settings via ox.config().

osmnx.geometries.**geometries_from_bbox**(*north*, *south*, *east*, *west*, *tags*)
Create a GeoDataFrame of OSM entities within a N, S, E, W bounding box.

Parameters

- **north** (*float*) – northern latitude of bounding box
- **south** (*float*) – southern latitude of bounding box
- **east** (*float*) – eastern longitude of bounding box
- **west** (*float*) – western longitude of bounding box
- **tags** (*dict*) – Dict of tags used for finding objects in the selected area. Results returned are the union, not intersection of each individual tag. Each result matches at least one given tag. The dict keys should be OSM tags, (e.g., *building*, *landuse*, *highway*, etc) and the dict values should be either *True* to retrieve all items with the given tag, or a string to get a single tag-value combination, or a list of strings to get multiple values for the given tag. For example, *tags = {'building': True}* would return all building footprints in the area. *tags = {'amenity':True, 'landuse':['retail','commercial'], 'highway':'bus_stop'}* would return all amenities, landuse=retail, landuse=commercial, and highway=bus_stop.

Returns gdf

Return type geopandas.GeoDataFrame

Notes

You can configure the Overpass server timeout, memory allocation, and other custom settings via ox.config().

osmnx.geometries.**geometries_from_place**(*query*, *tags*, *which_result=None*, *buffer_dist=None*)
Create a GeoDataFrame of OSM entities within the boundaries of a place.

Parameters

- **query** (*string or dict or list*) – the query or queries to geocode to get place boundary polygon(s)
- **tags** (*dict*) – Dict of tags used for finding objects in the selected area. Results returned are the union, not intersection of each individual tag. Each result matches at least one given tag. The dict keys should be OSM tags, (e.g., *building*, *landuse*, *highway*, etc) and the dict values should be either *True* to retrieve all items with the given tag, or a string to get a single tag-value combination, or a list of strings to get multiple values for the given tag. For example, *tags = {'building': True}* would return all building footprints in the area. *tags = {'amenity':True, 'landuse':['retail','commercial'], 'highway':'bus_stop'}* would return all amenities, landuse=retail, landuse=commercial, and highway=bus_stop.
- **which_result** (*int*) – which geocoding result to use. if None, auto-select the first (Multi)Polygon or raise an error if OSM doesn't return one.
- **buffer_dist** (*float*) – distance to buffer around the place geometry, in meters

Returns gdf

Return type geopandas.GeoDataFrame

Notes

You can configure the Overpass server timeout, memory allocation, and other custom settings via ox.config().

osmnx.geometries.**geometries_from_point**(*center_point*, *tags*, *dist=1000*)
Create GeoDataFrame of OSM entities within some distance N, S, E, W of a point.

Parameters

- **center_point** (*tuple*) – the (lat, lng) center point around which to get the geometries
- **tags** (*dict*) – Dict of tags used for finding objects in the selected area. Results returned are the union, not intersection of each individual tag. Each result matches at least one given tag. The dict keys should be OSM tags, (e.g., *building*, *landuse*, *highway*, etc) and the dict values should be either *True* to retrieve all items with the given tag, or a string to get a single tag-value combination, or a list of strings to get multiple values for the given tag. For example, *tags = {'building': True}* would return all building footprints in the area. *tags = {'amenity':True, 'landuse':['retail','commercial'], 'highway':'bus_stop'}* would return all amenities, landuse=retail, landuse=commercial, and highway=bus_stop.
- **dist** (*numeric*) – distance in meters

Returns gdf

Return type geopandas.GeoDataFrame

Notes

You can configure the Overpass server timeout, memory allocation, and other custom settings via ox.config().

osmnx.geometries.**geometries_from_polygon**(*polygon*, *tags*)
Create GeoDataFrame of OSM entities within boundaries of a (multi)polygon.

Parameters

- **polygon** (*shapely.geometry.Polygon or shapely.geometry.MultiPolygon*) – geographic boundaries to fetch geometries within
- **tags** (*dict*) – Dict of tags used for finding objects in the selected area. Results returned are the union, not intersection of each individual tag. Each result matches at least one given tag. The dict keys should be OSM tags, (e.g., *building*, *landuse*, *highway*, etc) and the dict values should be either *True* to retrieve all items with the given tag, or a string to get a single tag-value combination, or a list of strings to get multiple values for the given tag. For example, *tags = {'building': True}* would return all building footprints in the area. *tags = {'amenity':True, 'landuse':['retail','commercial'], 'highway':'bus_stop'}* would return all amenities, landuse=retail, landuse=commercial, and highway=bus_stop.

Returns gdf

Return type geopandas.GeoDataFrame

Notes

You can configure the Overpass server timeout, memory allocation, and other custom settings via ox.config().

osmnx.geometries.**geometries_from_xml**(*filepath*, *polygon=None*, *tags=None*)
Create a GeoDataFrame of OSM entities in an OSM-formatted XML file.

Because this function creates a GeoDataFrame of geometries from an OSM-formatted XML file that has already been downloaded (i.e. no query is made to the Overpass API) the polygon and tags arguments are not required. If they are not supplied to the function, geometries_from_xml() will return geometries for all of the tagged elements in the file. If they are supplied they will be used to filter the final GeoDataFrame.

Parameters

- **filepath** (*string or pathlib.Path*) – path to file containing OSM XML data
- **polygon** (*shapely.geometry.Polygon*) – optional geographic boundary to filter objects
- **tags** (*dict*) – optional dict of tags for filtering objects from the XML. Results returned are the union, not intersection of each individual tag. Each result matches at least one given tag. The dict keys should be OSM tags, (e.g., *building*, *landuse*, *highway*, etc) and the dict values should be either *True* to retrieve all items with the given tag, or a string to get a single tag-value combination, or a list of strings to get multiple values for the given tag. For example, *tags = {'building': True}* would return all building footprints in the area. *tags = {'amenity':True, 'landuse':['retail','commercial'], 'highway':'bus_stop'}* would return all amenities, landuse=retail, landuse=commercial, and highway=bus_stop.

Returns **gdf**

Return type geopandas.GeoDataFrame

3.1.8 osmnx.graph module

Graph creation functions.

osmnx.graph.**graph_from_address**(*address*, *dist=1000*, *dist_type='bbox'*, *network_type='all_private'*, *simplify=True*, *retain_all=False*, *truncate_by_edge=False*, *return_coords=False*, *clean_periphery=True*, *custom_filter=None*)
Create a graph from OSM within some distance of some address.

Parameters

- **address** (*string*) – the address to geocode and use as the central point around which to construct the graph
- **dist** (*int*) – retain only those nodes within this many meters of the center of the graph
- **dist_type** (*string {"network", "bbox"}*) – if "bbox", retain only those nodes within a bounding box of the distance parameter. if "network", retain only those nodes within some network distance from the center-most node.
- **network_type** (*string {"all_private", "all", "bike", "drive", "drive_service", "walk"}*) – what type of street network to get if custom_filter is None
- **simplify** (*bool*) – if True, simplify graph topology with the *simplify_graph* function
- **retain_all** (*bool*) – if True, return the entire graph even if it is not connected. otherwise, retain only the largest weakly connected component.

- **truncate_by_edge** (*bool*) – if True, retain nodes outside bounding box if at least one of node's neighbors is within the bounding box
- **return_coords** (*bool*) – optionally also return the geocoded coordinates of the address
- **clean_periphery** (*bool,*) – if True, buffer 500m to get a graph larger than requested, then simplify, then truncate it to requested spatial boundaries
- **custom_filter** (*string*) – a custom ways filter to be used instead of the network_type presets e.g., '["power"~"line"]' or '["highway"~"motorway|trunk"]'. Also pass in a network_type that is in settings.bidirectional_network_types if you want graph to be fully bidirectional.

Returns

Return type networkx.MultiDiGraph or optionally (networkx.MultiDiGraph, (lat, lng))

Notes

You can configure the Overpass server timeout, memory allocation, and other custom settings via ox.config().

osmnx.graph.**graph_from_bbox**(*north*, *south*, *east*, *west*, *network_type='all_private'*, *simplify=True*, *retain_all=False*, *truncate_by_edge=False*, *clean_periphery=True*, *custom_filter=None*)

Create a graph from OSM within some bounding box.

Parameters

- **north** (*float*) – northern latitude of bounding box
- **south** (*float*) – southern latitude of bounding box
- **east** (*float*) – eastern longitude of bounding box
- **west** (*float*) – western longitude of bounding box
- **network_type** (*string {"all_private", "all", "bike", "drive", "drive_service", "walk"}*) – what type of street network to get if custom_filter is None
- **simplify** (*bool*) – if True, simplify graph topology with the *simplify_graph* function
- **retain_all** (*bool*) – if True, return the entire graph even if it is not connected. otherwise, retain only the largest weakly connected component.
- **truncate_by_edge** (*bool*) – if True, retain nodes outside bounding box if at least one of node's neighbors is within the bounding box
- **clean_periphery** (*bool*) – if True, buffer 500m to get a graph larger than requested, then simplify, then truncate it to requested spatial boundaries
- **custom_filter** (*string*) – a custom ways filter to be used instead of the network_type presets e.g., '["power"~"line"]' or '["highway"~"motorway|trunk"]'. Also pass in a network_type that is in settings.bidirectional_network_types if you want graph to be fully bidirectional.

Returns G

Return type networkx.MultiDiGraph

Notes

You can configure the Overpass server timeout, memory allocation, and other custom settings via ox.config().

osmnx.graph.**graph_from_place**(*query*, *network_type='all_private'*, *simplify=True*, *retain_all=False*, *truncate_by_edge=False*, *which_result=None*, *buffer_dist=None*, *clean_periphery=True*, *custom_filter=None*)

Create graph from OSM within the boundaries of some geocodable place(s).

The query must be geocodable and OSM must have polygon boundaries for the geocode result. If OSM does not have a polygon for this place, you can instead get its street network using the graph_from_address function, which geocodes the place name to a point and gets the network within some distance of that point.

If OSM does have polygon boundaries for this place but you're not finding it, try to vary the query string, pass in a structured query dict, or vary the which_result argument to use a different geocode result. If you know the OSM ID of the place, you can retrieve its boundary polygon using the geocode_to_gdf function, then pass it to the graph_from_polygon function.

Parameters

- **query** (*string or dict or list*) – the query or queries to geocode to get place boundary polygon(s)
- **network_type** (*string {"all_private", "all", "bike", "drive", "drive_service", "walk"}*) – what type of street network to get if custom_filter is None
- **simplify** (*bool*) – if True, simplify graph topology with the *simplify_graph* function
- **retain_all** (*bool*) – if True, return the entire graph even if it is not connected. otherwise, retain only the largest weakly connected component.
- **truncate_by_edge** (*bool*) – if True, retain nodes outside boundary polygon if at least one of node's neighbors is within the polygon
- **which_result** (*int*) – which geocoding result to use. if None, auto-select the first (Multi)Polygon or raise an error if OSM doesn't return one.
- **buffer_dist** (*float*) – distance to buffer around the place geometry, in meters
- **clean_periphery** (*bool*) – if True, buffer 500m to get a graph larger than requested, then simplify, then truncate it to requested spatial boundaries
- **custom_filter** (*string*) – a custom ways filter to be used instead of the network_type presets e.g., '["power"~"line"]' or '["highway"~"motorway|trunk"]'. Also pass in a network_type that is in settings.bidirectional_network_types if you want graph to be fully bidirectional.

Returns G

Return type networkx.MultiDiGraph

Notes

You can configure the Overpass server timeout, memory allocation, and other custom settings via ox.config().

osmnx.graph.**graph_from_point**(*center_point*, *dist=1000*, *dist_type='bbox'*, *network_type='all_private'*, *simplify=True*, *retain_all=False*, *truncate_by_edge=False*, *clean_periphery=True*, *custom_filter=None*)

Create a graph from OSM within some distance of some (lat, lng) point.

Parameters

- **center_point** (*tuple*) – the (lat, lng) center point around which to construct the graph
- **dist** (*int*) – retain only those nodes within this many meters of the center of the graph, with distance determined according to dist_type argument
- **dist_type** (*string {"network", "bbox"}*) – if "bbox", retain only those nodes within a bounding box of the distance parameter. if "network", retain only those nodes within some network distance from the center-most node.
- **network_type** (*string, {"all_private", "all", "bike", "drive", "drive_service", "walk"}*) – what type of street network to get if custom_filter is None
- **simplify** (*bool*) – if True, simplify graph topology with the *simplify_graph* function
- **retain_all** (*bool*) – if True, return the entire graph even if it is not connected. otherwise, retain only the largest weakly connected component.
- **truncate_by_edge** (*bool*) – if True, retain nodes outside bounding box if at least one of node's neighbors is within the bounding box
- **clean_periphery** (*bool,*) – if True, buffer 500m to get a graph larger than requested, then simplify, then truncate it to requested spatial boundaries
- **custom_filter** (*string*) – a custom ways filter to be used instead of the network_type presets e.g., '["power"~"line"]' or '["highway"~"motorway|trunk"]'. Also pass in a network_type that is in settings.bidirectional_network_types if you want graph to be fully bidirectional.

Returns G

Return type networkx.MultiDiGraph

Notes

You can configure the Overpass server timeout, memory allocation, and other custom settings via ox.config().

osmnx.graph.**graph_from_polygon**(*polygon*, *network_type='all_private'*, *simplify=True*, *retain_all=False*, *truncate_by_edge=False*, *clean_periphery=True*, *custom_filter=None*)

Create a graph from OSM within the boundaries of some shapely polygon.

Parameters

- **polygon** (*shapely.geometry.Polygon or shapely.geometry.MultiPolygon*) – the shape to get network data within. coordinates should be in unprojected latitude-longitude degrees (EPSG:4326).
- **network_type** (*string {"all_private", "all", "bike", "drive", "drive_service", "walk"}*) – what type of street network to get if custom_filter is None

- **simplify** (*bool*) – if True, simplify graph topology with the *simplify_graph* function
- **retain_all** (*bool*) – if True, return the entire graph even if it is not connected. otherwise, retain only the largest weakly connected component.
- **truncate_by_edge** (*bool*) – if True, retain nodes outside boundary polygon if at least one of node's neighbors is within the polygon
- **clean_periphery** (*bool*) – if True, buffer 500m to get a graph larger than requested, then simplify, then truncate it to requested spatial boundaries
- **custom_filter** (*string*) – a custom ways filter to be used instead of the network_type presets e.g., '["power"~"line"]' or '["highway"~"motorway|trunk"]'. Also pass in a network_type that is in settings.bidirectional_network_types if you want graph to be fully bidirectional.

Returns G

Return type networkx.MultiDiGraph

Notes

You can configure the Overpass server timeout, memory allocation, and other custom settings via ox.config().

osmnx.graph.**graph_from_xml**(*filepath*, *bidirectional=False*, *simplify=True*, *retain_all=False*)

Create a graph from data in a .osm formatted XML file.

Parameters

- **filepath** (*string or pathlib.Path*) – path to file containing OSM XML data
- **bidirectional** (*bool*) – if True, create bi-directional edges for one-way streets
- **simplify** (*bool*) – if True, simplify graph topology with the *simplify_graph* function
- **retain_all** (*bool*) – if True, return the entire graph even if it is not connected. otherwise, retain only the largest weakly connected component.

Returns G

Return type networkx.MultiDiGraph

3.1.9 osmnx.io module

Serialize graphs to/from files on disk.

osmnx.io.**load_graphml**(*filepath*, *node_dtypes=None*, *edge_dtypes=None*)

Load an OSMnx-saved GraphML file from disk.

Converts the node/edge attributes to appropriate data types, which can be customized if needed by passing in *node_dtypes* or *edge_dtypes* arguments.

Parameters

- **filepath** (*string or pathlib.Path*) – path to the GraphML file
- **node_dtypes** (*dict*) – dict of node attribute names:types to convert values' data types
- **edge_dtypes** (*dict*) – dict of edge attribute names:types to convert values' data types

Returns G

Return type networkx.MultiDiGraph

osmnx.io.**save_graph_geopackage**(*G*, *filepath=None*, *encoding='utf-8'*, *directed=False*)
Save graph nodes and edges to disk as layers in a GeoPackage file.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **filepath** (*string or pathlib.Path*) – path to the GeoPackage file including extension. if None, use default data folder + graph.gpkg
- **encoding** (*string*) – the character encoding for the saved file
- **directed** (*bool*) – if False, save one edge for each undirected edge in the graph but retain original oneway and to/from information as edge attributes; if True, save one edge for each directed edge in the graph

Returns

Return type None

osmnx.io.**save_graph_shapefile**(*G*, *filepath=None*, *encoding='utf-8'*, *directed=False*)
Save graph nodes and edges to disk as ESRI shapefiles.

The shapefile format is proprietary and outdated. Whenever possible, you should use the superior GeoPackage file format instead via the save_graph_geopackage function.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **filepath** (*string or pathlib.Path*) – path to the shapefiles folder (no file extension). if None, use default data folder + graph_shapefile
- **encoding** (*string*) – the character encoding for the saved files
- **directed** (*bool*) – if False, save one edge for each undirected edge in the graph but retain original oneway and to/from information as edge attributes; if True, save one edge for each directed edge in the graph

Returns

Return type None

osmnx.io.**save_graph_xml**(*data*, *filepath=None*, *node_tags=['highway']*, *node_attrs=['id', 'timestamp', 'uid', 'user', 'version', 'changeset', 'lat', 'lon']*, *edge_tags=['highway', 'lanes', 'maxspeed', 'name', 'oneway']*, *edge_attrs=['id', 'timestamp', 'uid', 'user', 'version', 'changeset']*, *oneway=False*, *merge_edges=True*, *edge_tag_aggs=None*)
Do not use: deprecated. Use osm_xml.save_graph_xml instead.

Parameters

- **data** (*networkx multi(di)graph OR a length 2 iterable of nodes/edges*) – geopandas GeoDataFrames
- **filepath** (*string or pathlib.Path*) – path to the .osm file including extension. if None, use default data folder + graph.osm
- **node_tags** (*list*) – osm node tags to include in output OSM XML
- **node_attrs** (*list*) – osm node attributes to include in output OSM XML
- **edge_tags** (*list*) – osm way tags to include in output OSM XML
- **edge_attrs** (*list*) – osm way attributes to include in output OSM XML
- **oneway** (*bool*) – the default oneway value used to fill this tag where missing

- **merge_edges** (*bool*) – if True merges graph edges such that each OSM way has one entry and one entry only in the OSM XML. Otherwise, every OSM way will have a separate entry for each node pair it contains.
- **edge_tag_aggs** (*list of length-2 string tuples*) – useful only if merge_edges is True, this argument allows the user to specify edge attributes to aggregate such that the merged OSM way entry tags accurately represent the sum total of their component edge attributes. For example, if the user wants the OSM way to have a "length" attribute, the user must specify *edge_tag_aggs=[('length', 'sum')]* in order to tell this method to aggregate the lengths of the individual component edges. Otherwise, the length attribute will simply reflect the length of the first edge associated with the way.

Returns

Return type None

osmnx.io.**save_graphml**(*G*, *filepath=None*, *gephi=False*, *encoding='utf-8'*)

Save graph to disk as GraphML file.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **filepath** (*string or pathlib.Path*) – path to the GraphML file including extension. if None, use default data folder + graph.graphml
- **gephi** (*bool*) – if True, give each edge a unique key/id to work around Gephi's interpretation of the GraphML specification
- **encoding** (*string*) – the character encoding for the saved file

Returns

Return type None

3.1.10 osmnx.osm_xml module

Read/write .osm formatted XML files.

osmnx.osm_xml.**save_graph_xml**(*data*, *filepath=None*, *node_tags=['highway']*, *node_attrs=['id', 'timestamp', 'uid', 'user', 'version', 'changeset', 'lat', 'lon']*, *edge_tags=['highway', 'lanes', 'maxspeed', 'name', 'oneway']*, *edge_attrs=['id', 'timestamp', 'uid', 'user', 'version', 'changeset']*, *oneway=False*, *merge_edges=True*, *edge_tag_aggs=None*)

Save graph to disk as an OSM-formatted XML .osm file.

This function exists only to allow serialization to the .osm file format for applications that require it, and has constraints to conform to that. To save/load full-featured OSMnx graphs to/from disk for later use, use the save_graphml and load_graphml functions instead.

Note: for large networks this function can take a long time to run. Before using this function, make sure you configured OSMnx as described in the example below when you created the graph.

Example

```
>>> import osmnx as ox
>>> utn = ox.settings.useful_tags_node
>>> oxna = ox.settings.osm_xml_node_attrs
>>> oxnt = ox.settings.osm_xml_node_tags
>>> utw = ox.settings.useful_tags_way
>>> oxwa = ox.settings.osm_xml_way_attrs
>>> oxwt = ox.settings.osm_xml_way_tags
>>> utn = list(set(utn + oxna + oxnt))
>>> utw = list(set(utw + oxwa + oxwt))
>>> ox.config(all_oneway=True, useful_tags_node=utn, useful_tags_way=utw)
>>> G = ox.graph_from_place('Piedmont, CA, USA', network_type='drive')
>>> ox.save_graph_xml(G, filepath='./data/graph1.osm')
```

Parameters

- **data** (*networkx multi(di)graph OR a length 2 iterable of nodes/edges*) – geopandas GeoDataFrames
- **filepath** (*string or pathlib.Path*) – path to the .osm file including extension. if None, use default data folder + graph.osm
- **node_tags** (*list*) – osm node tags to include in output OSM XML
- **node_attrs** (*list*) – osm node attributes to include in output OSM XML
- **edge_tags** (*list*) – osm way tags to include in output OSM XML
- **edge_attrs** (*list*) – osm way attributes to include in output OSM XML
- **oneway** (*bool*) – the default oneway value used to fill this tag where missing
- **merge_edges** (*bool*) – if True merges graph edges such that each OSM way has one entry and one entry only in the OSM XML. Otherwise, every OSM way will have a separate entry for each node pair it contains.
- **edge_tag_aggs** (*list of length-2 string tuples*) – useful only if merge_edges is True, this argument allows the user to specify edge attributes to aggregate such that the merged OSM way entry tags accurately represent the sum total of their component edge attributes. For example, if the user wants the OSM way to have a "length" attribute, the user must specify *edge_tag_aggs=[('length', 'sum')]* in order to tell this method to aggregate the lengths of the individual component edges. Otherwise, the length attribute will simply reflect the length of the first edge associated with the way.

Returns

Return type None

3.1.11 osmnx.plot module

Plot spatial geometries, street networks, and routes.

osmnx.plot.**get_colors**(*n*, *cmap='viridis'*, *start=0.0*, *stop=1.0*, *alpha=1.0*, *return_hex=False*)

Get *n* evenly-spaced colors from a matplotlib colormap.

Parameters

- **n** (*int*) – number of colors
- **cmap** (*string*) – name of a matplotlib colormap
- **start** (*float*) – where to start in the colorspace
- **stop** (*float*) – where to end in the colorspace
- **alpha** (*float*) – opacity, the alpha channel for the RGBa colors
- **return_hex** (*bool*) – if True, convert RGBa colors to HTML-like hexadecimal RGB strings. if False, return colors as (R, G, B, alpha) tuples.

Returns color_list

Return type list

osmnx.plot.**get_edge_colors_by_attr**(*G*, *attr*, *num_bins=None*, *cmap='viridis'*, *start=0*, *stop=1*, *na_color='none'*, *equal_size=False*)

Get colors based on edge attribute values.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **attr** (*string*) – name of a numerical edge attribute
- **num_bins** (*int*) – if None, linearly map a color to each value. otherwise, assign values to this many bins then assign a color to each bin.
- **cmap** (*string*) – name of a matplotlib colormap
- **start** (*float*) – where to start in the colorspace
- **stop** (*float*) – where to end in the colorspace
- **na_color** (*string*) – what color to assign edges with missing attr values
- **equal_size** (*bool*) – ignored if num_bins is None. if True, bin into equal-sized quantiles (requires unique bin edges). if False, bin into equal-spaced bins.

Returns **edge_colors** – series labels are edge IDs (u, v, key) and values are colors

Return type pandas.Series

osmnx.plot.**get_node_colors_by_attr**(*G*, *attr*, *num_bins=None*, *cmap='viridis'*, *start=0*, *stop=1*, *na_color='none'*, *equal_size=False*)

Get colors based on node attribute values.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **attr** (*string*) – name of a numerical node attribute
- **num_bins** (*int*) – if None, linearly map a color to each value. otherwise, assign values to this many bins then assign a color to each bin.
- **cmap** (*string*) – name of a matplotlib colormap

- **start** (*float*) – where to start in the colorspace
- **stop** (*float*) – where to end in the colorspace
- **na_color** (*string*) – what color to assign nodes with missing attr values
- **equal_size** (*bool*) – ignored if num_bins is None. if True, bin into equal-sized quantiles (requires unique bin edges). if False, bin into equal-spaced bins.

Returns **node_colors** – series labels are node IDs and values are colors

Return type pandas.Series

osmnx.plot.**plot_figure_ground**(*G=None*, *address=None*, *point=None*, *dist=805*, *network_type='drive_service'*, *street_widths=None*, *default_width=4*, *figsize=(8, 8)*, *edge_color='w'*, *smooth_joints=True*, ***pg_kwargs*)

Plot a figure-ground diagram of a street network.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph, must be unprojected
- **address** (*string*) – address to geocode as the center point if G is not passed in
- **point** (*tuple*) – center point if address and G are not passed in
- **dist** (*numeric*) – how many meters to extend north, south, east, west from center point
- **network_type** (*string*) – what type of street network to get
- **street_widths** (*dict*) – dict keys are street types and values are widths to plot in pixels
- **default_width** (*numeric*) – fallback width in pixels for any street type not in street_widths
- **figsize** (*numeric*) – (width, height) of figure, should be equal
- **edge_color** (*string*) – color of the edges' lines
- **smooth_joints** (*bool*) – if True, plot nodes same width as streets to smooth line joints and prevent cracks between them from showing
- **pg_kwargs** – keyword arguments to pass to plot_graph

Returns **fig, ax** – matplotlib figure, axis

Return type tuple

osmnx.plot.**plot_footprints**(*gdf*, *ax=None*, *figsize=(8, 8)*, *color='orange'*, *alpha=None*, *bgcolor='#111111'*, *bbox=None*, *save=False*, *show=True*, *close=False*, *filepath=None*, *dpi=600*)

Plot a GeoDataFrame of geospatial entities' footprints.

Parameters

- **gdf** (*geopandas.GeoDataFrame*) – GeoDataFrame of footprints (shapely Polygons and MultiPolygons)
- **ax** (*axis*) – if not None, plot on this preexisting axis
- **figsize** (*tuple*) – if ax is None, create new figure with size (width, height)
- **color** (*string*) – color of the footprints
- **alpha** (*float*) – opacity of the footprints
- **bgcolor** (*string*) – background color of the plot

- **bbox** (*tuple*) – bounding box as (north, south, east, west). if None, will calculate from the spatial extents of the geometries in gdf
- **save** (*bool*) – if True, save the figure to disk at filepath
- **show** (*bool*) – if True, call pyplot.show() to show the figure
- **close** (*bool*) – if True, call pyplot.close() to close the figure
- **filepath** (*string*) – if save is True, the path to the file. file format determined from extension. if None, use settings.imgs_folder/image.png
- **dpi** (*int*) – if save is True, the resolution of saved file

Returns **fig, ax** – matplotlib figure, axis

Return type tuple

osmnx.plot.**plot_graph**(*G*, *ax=None*, *figsize=(8, 8)*, *bgcolor='#111111'*, *node_color='w'*, *node_size=15*, *node_alpha=None*, *node_edgecolor='none'*, *node_zorder=1*, *edge_color='#999999'*, *edge_linewidth=1*, *edge_alpha=None*, *show=True*, *close=False*, *save=False*, *filepath=None*, *dpi=300*, *bbox=None*)

Plot a graph.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **ax** (*matplotlib axis*) – if not None, plot on this preexisting axis
- **figsize** (*tuple*) – if ax is None, create new figure with size (width, height)
- **bgcolor** (*string*) – background color of plot
- **node_color** (*string or list*) – color(s) of the nodes
- **node_size** (*int*) – size of the nodes: if 0, then skip plotting the nodes
- **node_alpha** (*float*) – opacity of the nodes, note: if you passed RGBA values to node_color, set node_alpha=None to use the alpha channel in node_color
- **node_edgecolor** (*string*) – color of the nodes' markers' borders
- **node_zorder** (*int*) – zorder to plot nodes: edges are always 1, so set node_zorder=0 to plot nodes below edges
- **edge_color** (*string or list*) – color(s) of the edges' lines
- **edge_linewidth** (*float*) – width of the edges' lines: if 0, then skip plotting the edges
- **edge_alpha** (*float*) – opacity of the edges, note: if you passed RGBA values to edge_color, set edge_alpha=None to use the alpha channel in edge_color
- **show** (*bool*) – if True, call pyplot.show() to show the figure
- **close** (*bool*) – if True, call pyplot.close() to close the figure
- **save** (*bool*) – if True, save the figure to disk at filepath
- **filepath** (*string*) – if save is True, the path to the file. file format determined from extension. if None, use settings.imgs_folder/image.png
- **dpi** (*int*) – if save is True, the resolution of saved file
- **bbox** (*tuple*) – bounding box as (north, south, east, west). if None, will calculate from spatial extents of plotted geometries.

Returns **fig, ax** – matplotlib figure, axis

Return type tuple

osmnx.plot.**plot_graph_route**(*G*, *route*, *route_color='r'*, *route_linewidth=4*, *route_alpha=0.5*, *orig_dest_size=100*, *ax=None*, ***pg_kwargs*)

Plot a route along a graph.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **route** (*list*) – route as a list of node IDs
- **route_color** (*string*) – color of the route
- **route_linewidth** (*int*) – width of the route line
- **route_alpha** (*float*) – opacity of the route line
- **orig_dest_size** (*int*) – size of the origin and destination nodes
- **ax** (*matplotlib axis*) – if not None, plot route on this preexisting axis instead of creating a new fig, ax and drawing the underlying graph
- **pg_kwargs** – keyword arguments to pass to plot_graph

Returns **fig, ax** – matplotlib figure, axis

Return type tuple

osmnx.plot.**plot_graph_routes**(*G*, *routes*, *route_colors='r'*, ***pgr_kwargs*)

Plot several routes along a graph.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **routes** (*list*) – routes as a list of lists of node IDs
- **route_colors** (*string or list*) – if string, 1 color for all routes. if list, the colors for each route.
- **pgr_kwargs** – keyword arguments to pass to plot_graph_route

Returns **fig, ax** – matplotlib figure, axis

Return type tuple

3.1.12 osmnx.projection module

Project spatial geometries and spatial networks.

osmnx.projection.**project_gdf**(*gdf*, *to_crs=None*, *to_latlong=False*)

Project a GeoDataFrame from its current CRS to another.

If to_crs is None, project to the UTM CRS for the UTM zone in which the GeoDataFrame's centroid lies. Otherwise project to the CRS defined by to_crs. The simple UTM zone calculation in this function works well for most latitudes, but may not work for some extreme northern locations like Svalbard or far northern Norway.

Parameters

- **gdf** (*geopandas.GeoDataFrame*) – the GeoDataFrame to be projected
- **to_crs** (*string or pyproj.CRS*) – if None, project to UTM zone in which gdf's centroid lies, otherwise project to this CRS
- **to_latlong** (*bool*) – if True, project to settings.default_crs and ignore to_crs

Returns **gdf_proj** – the projected GeoDataFrame

Return type geopandas.GeoDataFrame

osmnx.projection.**project_geometry**(*geometry*, *crs=None*, *to_crs=None*, *to_latlong=False*)

Project a shapely geometry from its current CRS to another.

If to_crs is None, project to the UTM CRS for the UTM zone in which the geometry's centroid lies. Otherwise project to the CRS defined by to_crs.

Parameters

- **geometry** (*shapely.geometry.Polygon or shapely.geometry.MultiPolygon*) – the geometry to project
- **crs** (*string or pyproj.CRS*) – the starting CRS of the passed-in geometry. if None, it will be set to settings.default_crs
- **to_crs** (*string or pyproj.CRS*) – if None, project to UTM zone in which geometry's centroid lies, otherwise project to this CRS
- **to_latlong** (*bool*) – if True, project to settings.default_crs and ignore to_crs

Returns **geometry_proj, crs** – the projected geometry and its new CRS

Return type tuple

osmnx.projection.**project_graph**(*G*, *to_crs=None*)

Project graph from its current CRS to another.

If to_crs is None, project the graph to the UTM CRS for the UTM zone in which the graph's centroid lies. Otherwise, project the graph to the CRS defined by to_crs.

Parameters

- **G** (*networkx.MultiDiGraph*) – the graph to be projected
- **to_crs** (*string or pyproj.CRS*) – if None, project graph to UTM zone in which graph centroid lies, otherwise project graph to this CRS

Returns **G_proj** – the projected graph

Return type networkx.MultiDiGraph

3.1.13 osmnx.settings module

Global settings that can be configured by user with utils.config().

3.1.14 osmnx.simplification module

Simplify, correct, and consolidate network topology.

osmnx.simplification.**consolidate_intersections**(*G*, *tolerance=10*, *rebuild_graph=True*, *dead_ends=False*, *reconnect_edges=True*)

Consolidate intersections comprising clusters of nearby nodes.

Merges nearby nodes and returns either their centroids or a rebuilt graph with consolidated intersections and reconnected edge geometries. The tolerance argument should be adjusted to approximately match street design standards in the specific street network, and you should always use a projected graph to work in meaningful and consistent units like meters.

When rebuild_graph=False, it uses a purely geometrical (and relatively fast) algorithm to identify "geometrically close" nodes, merge them, and return just the merged intersections' centroids. When rebuild_graph=True, it uses a topological (and slower but more accurate) algorithm to identify "topologically close" nodes, merge them, then rebuild/return the graph. Returned graph's node IDs represent clusters rather than osmids. Refer to nodes' osmid_original attributes for original osmids. If multiple nodes were merged together, the osmid_original attribute is a list of merged nodes' osmids.

Divided roads are often represented by separate centerline edges. The intersection of two divided roads thus creates 4 nodes, representing where each edge intersects a perpendicular edge. These 4 nodes represent a single intersection in the real world. A similar situation occurs with roundabouts and traffic circles. This function consolidates nearby nodes by buffering them to an arbitrary distance, merging overlapping buffers, and taking their centroid.

Parameters

- **G** (*networkx.MultiDiGraph*) – a projected graph
- **tolerance** (*float*) – nodes are buffered to this distance (in graph's geometry's units) and subsequent overlaps are dissolved into a single node
- **rebuild_graph** (*bool*) – if True, consolidate the nodes topologically, rebuild the graph, and return as networkx.MultiDiGraph. if False, consolidate the nodes geometrically and return the consolidated node points as geopandas.GeoSeries
- **dead_ends** (*bool*) – if False, discard dead-end nodes to return only street-intersection points
- **reconnect_edges** (*bool*) – ignored if rebuild_graph is not True. if True, reconnect edges and their geometries in rebuilt graph to the consolidated nodes and update edge length attributes; if False, returned graph has no edges (which is faster if you just need topologically consolidated intersection counts).

Returns if rebuild_graph=True, returns MultiDiGraph with consolidated intersections and reconnected edge geometries. if rebuild_graph=False, returns GeoSeries of shapely Points representing the centroids of street intersections

Return type networkx.MultiDiGraph or geopandas.GeoSeries

osmnx.simplification.**simplify_graph**(*G*, *strict=True*, *remove_rings=True*)

Simplify a graph's topology by removing interstitial nodes.

Simplifies graph topology by removing all nodes that are not intersections or dead-ends. Create an edge directly between the end points that encapsulate them, but retain the geometry of the original edges, saved as a new *geometry* attribute on the new edge. Note that only simplified edges receive a *geometry* attribute. Some of the resulting consolidated edges may comprise multiple OSM ways, and if so, their multiple attribute values are stored as a list.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **strict** (*bool*) – if False, allow nodes to be end points even if they fail all other rules but have incident edges with different OSM IDs. Lets you keep nodes at elbow two-way intersections, but sometimes individual blocks have multiple OSM IDs within them too.
- **remove_rings** (*bool*) – if True, remove isolated self-contained rings that have no endpoints

Returns **G** – topologically simplified graph, with a new *geometry* attribute on each simplified edge

Return type networkx.MultiDiGraph

3.1.15 osmnx.speed module

Calculate graph edge speeds and travel times.

osmnx.speed.**add_edge_speeds**(*G*, *hwy_speeds=None*, *fallback=None*, *precision=1*)

Add edge speeds (km per hour) to graph as new *speed_kph* edge attributes.

Imputes free-flow travel speeds for all edges based on mean *maxspeed* value of edges, per highway type. For highway types in graph that have no *maxspeed* value on any edge, function assigns the mean of all *maxspeed* values in graph.

This mean-imputation can obviously be imprecise, and the caller can override it by passing in *hwy_speeds* and/or *fallback* arguments that correspond to local speed limit standards.

If edge *maxspeed* attribute has "mph" in it, value will automatically be converted from miles per hour to km per hour. Any other speed units should be manually converted to km per hour prior to running this function, otherwise there could be unexpected results. If "mph" does not appear in the edge's maxspeed attribute string, then function assumes kph, per OSM guidelines: https://wiki.openstreetmap.org/wiki/Map_Features/Units

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **hwy_speeds** (*dict*) – dict keys = OSM highway types and values = typical speeds (km per hour) to assign to edges of that highway type for any edges missing speed data. Any edges with highway type not in *hwy_speeds* will be assigned the mean preexisting speed value of all edges of that highway type.
- **fallback** (*numeric*) – default speed value (km per hour) to assign to edges whose highway type did not appear in *hwy_speeds* and had no preexisting speed values on any edge
- **precision** (*int*) – decimal precision to round speed_kph

Returns **G** – graph with speed_kph attributes on all edges

Return type networkx.MultiDiGraph

osmnx.speed.**add_edge_travel_times**(*G*, *precision=1*)

Add edge travel time (seconds) to graph as new *travel_time* edge attributes.

Calculates free-flow travel time along each edge, based on *length* and *speed_kph* attributes. Note: run *add_edge_speeds* first to generate the *speed_kph* attribute. All edges must have *length* and *speed_kph* attributes and all their values must be non-null.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **precision** (*int*) – decimal precision to round travel_time

Returns **G** – graph with travel_time attributes on all edges

Return type networkx.MultiDiGraph

3.1.16 osmnx.stats module

Calculate geometric and topological network measures.

osmnx.stats.**basic_stats**(*G*, *area=None*, *clean_intersects=False*, *tolerance=15*, *circuity_dist='gc'*)

Calculate basic descriptive geometric and topological stats for a graph.

For an unprojected lat-lng graph, tolerance and graph units should be in degrees, and circuity_dist should be 'gc'. For a projected graph, tolerance and graph units should be in meters (or similar) and circuity_dist should be 'euclidean'.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **area** (*numeric*) – the land area of this study site, in square meters. must be greater than 0. if None, will skip all density-based metrics.
- **clean_intersects** (*bool*) – if True, calculate consolidated intersections count (and density, if area is provided) via consolidate_intersections function
- **tolerance** (*numeric*) – tolerance value passed along if clean_intersects=True, see consolidate_intersections function documentation for details and usage
- **circuity_dist** (*string*) – 'gc' or 'euclidean', how to calculate straight-line distances for circuity measurement; use former for lat-lng networks and latter for projected networks

Returns

stats – network measures containing the following elements (some keys may not be present, based on the arguments passed into the function):

- n = number of nodes in the graph
- m = number of edges in the graph
- k_avg = average node degree of the graph
- **intersection_count = number of intersections in graph, that is,** nodes with >1 physical street connected to them
- **streets_per_node_avg = how many physical streets (edges in the** undirected representation of the graph) connect to each node (ie, intersection or dead-end) on average (mean)
- **streets_per_node_counts = dict with keys of number of physical** streets connecting to a node, and values of number of nodes with this count
- **streets_per_node_proportion = dict, same as previous, but as a** proportion of the total, rather than counts
- edge_length_total = sum of all edge lengths in graph, in meters
- edge_length_avg = mean edge length in the graph, in meters
- **street_length_total = sum of all edges in the undirected** representation of the graph
- **street_length_avg = mean edge length in the undirected** representation of the graph, in meters
- **street_segments_count = number of edges in the undirected** representation of the graph
- node_density_km = n divided by area in square kilometers
- **intersection_density_km = intersection_count divided by area in** square kilometers
- **edge_density_km = edge_length_total divided by area in square** kilometers

- **street_density_km = street_length_total divided by area in square** kilometers
- **circuity_avg = edge_length_total divided by the sum of the great** circle distances between the nodes of each edge
- **self_loop_proportion = proportion of edges that have a single node** as its endpoints (ie, the edge links nodes u and v, and u==v)
- **clean_intersection_count = number of intersections in street** network, merging complex ones into single points
- **clean_intersection_density_km = clean_intersection_count divided** by area in square kilometers

Return type dict

osmnx.stats.**extended_stats**(*G*, *connectivity=False*, *anc=False*, *ecc=False*, *bc=False*, *cc=False*)

Calculate extended topological measures for a graph.

Many of these algorithms have an inherently high time complexity. Global topological analysis of large complex networks is extremely time consuming and may exhaust computer memory. Consider using function arguments to not run metrics that require computation of a full matrix of paths if they will not be needed.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **connectivity** (*bool*) – if True, calculate node and edge connectivity
- **anc** (*bool*) – if True, calculate average node connectivity
- **ecc** (*bool*) – if True, calculate shortest paths, eccentricity, and topological metrics that use eccentricity
- **bc** (*bool*) – if True, calculate node betweenness centrality
- **cc** (*bool*) – if True, calculate node closeness centrality

Returns

stats – dictionary of network measures containing the following elements (some only calculated/returned optionally, based on passed parameters):

- avg_neighbor_degree
- avg_neighbor_degree_avg
- avg_weighted_neighbor_degree
- avg_weighted_neighbor_degree_avg
- degree_centrality
- degree_centrality_avg
- clustering_coefficient
- clustering_coefficient_avg
- clustering_coefficient_weighted
- clustering_coefficient_weighted_avg
- pagerank
- pagerank_max_node
- pagerank_max

- pagerank_min_node
- pagerank_min
- node_connectivity
- node_connectivity_avg
- edge_connectivity
- eccentricity
- diameter
- radius
- center
- periphery
- closeness_centrality
- closeness_centrality_avg
- betweenness_centrality
- betweenness_centrality_avg

Return type dict

3.1.17 osmnx.truncate module

Truncate graph by distance, bounding box, or polygon.

osmnx.truncate.**truncate_graph_bbox**(*G*, *north*, *south*, *east*, *west*, *truncate_by_edge=False*, *retain_all=False*, *quadrat_width=0.05*, *min_num=3*)

Remove every node in graph that falls outside a bounding box.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **north** (*float*) – northern latitude of bounding box
- **south** (*float*) – southern latitude of bounding box
- **east** (*float*) – eastern longitude of bounding box
- **west** (*float*) – western longitude of bounding box
- **truncate_by_edge** (*bool*) – if True, retain nodes outside bounding box if at least one of node's neighbors is within the bounding box
- **retain_all** (*bool*) – if True, return the entire graph even if it is not connected. otherwise, retain only the largest weakly connected component.
- **quadrat_width** (*numeric*) – passed on to intersect_index_quadrats: the linear length (in degrees) of the quadrats with which to cut up the geometry (default = 0.05, approx 4km at NYC's latitude)
- **min_num** (*int*) – passed on to intersect_index_quadrats: the minimum number of linear quadrat lines (e.g., min_num=3 would produce a quadrat grid of 4 squares)

Returns **G** – the truncated graph

Return type networkx.MultiDiGraph

osmnx.truncate.**truncate_graph_dist**(*G*, *source_node*, *max_dist=1000*, *weight='length'*, *retain_all=False*)

Remove every node farther than some network distance from source_node.

This function can be slow for large graphs, as it must calculate shortest path distances between source_node and every other graph node.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **source_node** (*int*) – the node in the graph from which to measure network distances to other nodes
- **max_dist** (*int*) – remove every node in the graph greater than this distance from the source_node (along the network)
- **weight** (*string*) – how to weight the graph when measuring distance (default 'length' is how many meters long the edge is)
- **retain_all** (*bool*) – if True, return the entire graph even if it is not connected. otherwise, retain only the largest weakly connected component.

Returns **G** – the truncated graph

Return type networkx.MultiDiGraph

osmnx.truncate.**truncate_graph_polygon**(*G*, *polygon*, *retain_all=False*, *truncate_by_edge=False*, *quadrat_width=0.05*, *min_num=3*)

Remove every node in graph that falls outside a (Multi)Polygon.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **polygon** (*shapely.geometry.Polygon or shapely.geometry.MultiPolygon*) – only retain nodes in graph that lie within this geometry
- **retain_all** (*bool*) – if True, return the entire graph even if it is not connected. otherwise, retain only the largest weakly connected component.
- **truncate_by_edge** (*bool*) – if True, retain nodes outside boundary polygon if at least one of node's neighbors is within the polygon
- **quadrat_width** (*numeric*) – passed on to intersect_index_quadrats: the linear length (in degrees) of the quadrats with which to cut up the geometry (default = 0.05, approx 4km at NYC's latitude)
- **min_num** (*int*) – passed on to intersect_index_quadrats: the minimum number of linear quadrat lines (e.g., min_num=3 would produce a quadrat grid of 4 squares)

Returns **G** – the truncated graph

Return type networkx.MultiDiGraph

3.1.18 osmnx.utils module

General utility functions.

osmnx.utils.**citation**()
Print the OSMnx package's citation information.

Boeing, G. 2017. OSMnx: New Methods for Acquiring, Constructing, Analyzing, and Visualizing Complex Street Networks. Computers, Environment and Urban Systems, 65(126-139). https://doi.org/10.1016/j.compenvurbsys.2017.05.004

Returns

Return type None

osmnx.utils.**config**(*all_oneway=False*, *bidirectional_network_types=['walk']*, *cache_folder='./cache'*, *cache_only_mode=False*, *data_folder='./data'*, *default_accept_language='en'*, *default_access='["access"!~"private"]'*, *default_crs='epsg:4326'*, *default_referer='OSMnx Python package (https://github.com/gboeing/osmnx)'*, *default_user_agent='OSMnx Python package (https://github.com/gboeing/osmnx)'*, *elevation_provider='google'*, *imgs_folder='./images'*, *log_console=False*, *log_file=False*, *log_filename='osmnx'*, *log_level=20*, *log_name='OSMnx'*, *logs_folder='./logs'*, *max_query_area_size=2500000000*, *memory=None*, *nominatim_endpoint='https://nominatim.openstreetmap.org/'*, *nominatim_key=None*, *osm_xml_node_attrs=['id', 'timestamp', 'uid', 'user', 'version', 'changeset', 'lat', 'lon']*, *osm_xml_node_tags=['highway']*, *osm_xml_way_attrs=['id', 'timestamp', 'uid', 'user', 'version', 'changeset']*, *osm_xml_way_tags=['highway', 'lanes', 'maxspeed', 'name', 'oneway']*, *overpass_endpoint='http://overpass-api.de/api'*, *overpass_settings='[out:json][timeout:{timeout}]{maxsize}'*, *timeout=180*, *use_cache=True*, *useful_tags_node=['ref', 'highway']*, *useful_tags_way=['bridge', 'tunnel', 'oneway', 'lanes', 'ref', 'name', 'highway', 'maxspeed', 'service', 'access', 'area', 'landuse', 'width', 'est_width', 'junction']*)
Configure OSMnx by setting the default global settings' values.

Any parameters not passed by the caller are (re-)set to their original default values.

Parameters

- **all_oneway** (*bool*) – Only use if specifically saving to .osm XML file with save_graph_xml function. if True, forces all ways to be loaded as oneway ways, preserving the original order of nodes stored in the OSM way XML.
- **bidirectional_network_types** (*list*) – network types for which a fully bidirectional graph will be created
- **cache_folder** (*string or pathlib.Path*) – path to folder in which to save/load HTTP response cache
- **data_folder** (*string or pathlib.Path*) – path to folder in which to save/load graph files by default
- **cache_only_mode** (*bool*) – If True, download network data from Overpass then raise a CacheOnlyModeInterrupt error for user to catch. This prevents graph building from taking place and instead just saves OSM response data to cache. Useful for sequentially caching lots of raw data (as you can only query Overpass one request at a time) then using the cache to quickly build many graphs simultaneously with multiprocessing.
- **default_accept_language** (*string*) – HTTP header accept-language
- **default_access** (*string*) – default filter for OSM "access" key

- **default_crs** (*string*) – default coordinate reference system to set when creating graphs
- **default_referer** (*string*) – HTTP header referer
- **default_user_agent** (*string*) – HTTP header user-agent
- **elevation_provider** (*string {"google", "airmap"}*) – the API provider to use for adding node elevations
- **imgs_folder** (*string or pathlib.Path*) – path to folder in which to save plot images by default
- **log_file** (*bool*) – if True, save log output to a file in logs_folder
- **log_filename** (*string*) – name of the log file, without file extension
- **log_console** (*bool*) – if True, print log output to the console (terminal window)
- **log_level** (*int*) – one of Python's logger.level constants
- **log_name** (*string*) – name of the logger
- **logs_folder** (*string or pathlib.Path*) – path to folder in which to save log files
- **max_query_area_size** (*int*) – maximum area for any part of the geometry in meters: any polygon bigger than this will get divided up for multiple queries to API (default 50km x 50km)
- **memory** (*int*) – Overpass server memory allocation size for the query, in bytes. If None, server will use its default allocation size. Use with caution.
- **nominatim_endpoint** (*string*) – the API endpoint to use for nominatim queries
- **nominatim_key** (*string*) – your API key, if you are using an endpoint that requires one
- **osm_xml_node_attrs** (*list*) – node attributes for saving .osm XML files with save_graph_xml function
- **osm_xml_node_tags** (*list*) – node tags for saving .osm XML files with save_graph_xml function
- **osm_xml_way_attrs** (*list*) – edge attributes for saving .osm XML files with save_graph_xml function
- **osm_xml_way_tags** (*list*) – edge tags for for saving .osm XML files with save_graph_xml function
- **overpass_endpoint** (*string*) – the API endpoint to use for overpass queries
- **overpass_settings** (*string*) – Settings string for overpass queries. For example, to query historical OSM data as of a certain date: *'[out:json][timeout:90][date:"2019-10-28T19:20:00Z"]'*. Use with caution.
- **timeout** (*int*) – the timeout interval for the HTTP request and for API to use while running the query
- **use_cache** (*bool*) – if True, cache HTTP responses locally instead of calling API repeatedly for the same request
- **useful_tags_node** (*list*) – OSM "node" tags to add as graph node attributes, when present

- **useful_tags_way** (*list*) – OSM "way" tags to add as graph edge attributes, when present

Returns

Return type None

osmnx.utils.**log**(*message*, *level=None*, *name=None*, *filename=None*)
Write a message to the logger.

This logs to file and/or prints to the console (terminal), depending on the current configuration of settings.log_file and settings.log_console.

Parameters

- **message** (*string*) – the message to log
- **level** (*int*) – one of Python's logger.level constants
- **name** (*string*) – name of the logger
- **filename** (*string*) – name of the log file, without file extension

Returns

Return type None

osmnx.utils.**ts**(*style='datetime'*, *template=None*)
Get current timestamp as string.

Parameters

- **style** (*string {"datetime", "date", "time"}*) – format the timestamp with this built-in template
- **template** (*string*) – if not None, format the timestamp with this template instead of one of the built-in styles

Returns **ts** – the string timestamp

Return type string

3.1.19 osmnx.utils_geo module

Geospatial utility functions.

osmnx.utils_geo.**bbox_from_point**(*point*, *dist=1000*, *project_utm=False*, *return_crs=False*)
Create a bounding box from a (lat, lng) center point.

Create a bounding box some distance in each direction (north, south, east, and west) from the center point and optionally project it.

Parameters

- **point** (*tuple*) – the (lat, lng) center point to create the bounding box around
- **dist** (*int*) – bounding box distance in meters from the center point
- **project_utm** (*bool*) – if True, return bounding box as UTM-projected coordinates
- **return_crs** (*bool*) – if True, and project_utm=True, return the projected CRS too

Returns (north, south, east, west) or (north, south, east, west, crs_proj)

Return type tuple

osmnx.utils_geo.**bbox_to_poly**(*north*, *south*, *east*, *west*)
Convert bounding box coordinates to shapely Polygon.

Parameters

- **north** (*float*) – northern coordinate
- **south** (*float*) – southern coordinate
- **east** (*float*) – eastern coordinate
- **west** (*float*) – western coordinate

Returns

Return type shapely.geometry.Polygon

osmnx.utils_geo.**redistribute_vertices**(*geom*, *dist*)
Redistribute the vertices on a projected LineString or MultiLineString.

The distance argument is only approximate since the total distance of the linestring may not be a multiple of the preferred distance. This function works on only (Multi)LineString geometry types.

Parameters

- **geom** (*shapely.geometry.LineString or shapely.geometry.MultiLineString*) – a Shapely geometry (should be projected)
- **dist** (*float*) – spacing length along edges. Units are same as the geom: degrees for unprojected geometries and meters for projected geometries. The smaller the dist value, the more points are created.

Returns the redistributed vertices as a list if geom is a LineString or MultiLineString if geom is a MultiLineString

Return type list or shapely.geometry.MultiLineString

osmnx.utils_geo.**round_geometry_coords**(*geom*, *precision*)
Round the coordinates of a shapely geometry to some decimal precision.

Parameters

- **geom** (*shapely.geometry.geometry {Point, MultiPoint, LineString, MultiLineString, Polygon, MultiPolygon}*) – the geometry to round the coordinates of
- **precision** (*int*) – decimal precision to round coordinates to

Returns

Return type shapely.geometry.geometry

3.1.20 osmnx.utils_graph module

Graph utility functions.

osmnx.utils_graph.**add_edge_lengths**(*G*, *precision=3*)
Add *length* attribute (in meters) to each edge.

Calculated via great-circle distance between each edge's incident nodes, so ensure graph is in unprojected coordinates. Graph should be unsimplified to get accurate distances. Note: this function is run by all the *graph.graph_from_x* functions automatically to add *length* attributes to all edges.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **precision** (*int*) – decimal precision to round lengths

Returns **G** – graph with edge length attributes

Return type networkx.MultiDiGraph

osmnx.utils_graph.**count_streets_per_node**(*G*, *nodes=None*)

Count how many physical street segments connect to each node in a graph.

This function uses an undirected representation of the graph and special handling of self-loops to accurately count physical streets rather than directed edges. Note: this function is automatically run by all the *graph.graph_from_x* functions prior to truncating the graph to the requested boundaries, to add accurate *street_count* attributes to each node even if some of its neighbors are outside the requested graph boundaries.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **nodes** (*list*) – which node IDs to get counts for. if None, use all graph nodes, otherwise calculate counts only for these node IDs

Returns **streets_per_node** – counts of how many physical streets connect to each node, with keys = node ids and values = counts

Return type dict

osmnx.utils_graph.**get_digraph**(*G*, *weight='length'*)

Convert MultiDiGraph to DiGraph.

Chooses between parallel edges by minimizing *weight* attribute value. Note: see also *get_undirected* to convert MultiDiGraph to MultiGraph.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **weight** (*string*) – attribute value to minimize when choosing between parallel edges

Returns

Return type networkx.DiGraph

osmnx.utils_graph.**get_largest_component**(*G*, *strongly=False*)

Get subgraph of G's largest weakly/strongly connected component.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **strongly** (*bool*) – if True, return the largest strongly instead of weakly connected component

Returns **G** – the largest connected component subgraph of the original graph

Return type networkx.MultiDiGraph

osmnx.utils_graph.**get_route_edge_attributes**(*G*, *route*, *attribute=None*, *minimize_key='length'*, *retrieve_default=None*)

Get a list of attribute values for each edge in a path.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **route** (*list*) – list of nodes IDs constituting the path

- **attribute** (*string*) – the name of the attribute to get the value of for each edge. If None, the complete data dict is returned for each edge.
- **minimize_key** (*string*) – if there are parallel edges between two nodes, select the one with the lowest value of minimize_key
- **retrieve_default** (*Callable[Tuple[Any, Any], Any]*) – function called with the edge nodes as parameters to retrieve a default value, if the edge does not contain the given attribute (otherwise a *KeyError* is raised)

Returns **attribute_values** – list of edge attribute values

Return type list

osmnx.utils_graph.**get_undirected**(*G*)

Convert MultiDiGraph to undirected MultiGraph.

Maintains parallel edges only if their geometries differ. Note: see also *get_digraph* to convert MultiDiGraph to DiGraph.

Parameters **G** (*networkx.MultiDiGraph*) – input graph

Returns

Return type networkx.MultiGraph

osmnx.utils_graph.**graph_from_gdfs**(*gdf_nodes*, *gdf_edges*, *graph_attrs=None*)

Convert node and edge GeoDataFrames to a MultiDiGraph.

This function is the inverse of *graph_to_gdfs* and is designed to work in conjunction with it. However, you can convert arbitrary node and edge GeoDataFrames as long as gdf_nodes is uniquely indexed by *osmid* and gdf_edges is uniquely multi-indexed by *u*, *v*, *key* (following normal MultiDiGraph structure). This allows you to load any node/edge shapefiles or GeoPackage layers as GeoDataFrames then convert them to a MultiDiGraph for graph analysis.

Parameters

- **gdf_nodes** (*geopandas.GeoDataFrame*) – GeoDataFrame of graph nodes uniquely indexed by osmid
- **gdf_edges** (*geopandas.GeoDataFrame*) – GeoDataFrame of graph edges uniquely multi-indexed by u, v, key
- **graph_attrs** (*dict*) – the new G.graph attribute dict. if None, use crs from gdf_edges as the only graph-level attribute (gdf_edges must have crs attribute set)

Returns **G**

Return type networkx.MultiDiGraph

osmnx.utils_graph.**graph_to_gdfs**(*G*, *nodes=True*, *edges=True*, *node_geometry=True*, *fill_edge_geometry=True*)

Convert a MultiDiGraph to node and/or edge GeoDataFrames.

This function is the inverse of *graph_from_gdfs*.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **nodes** (*bool*) – if True, convert graph nodes to a GeoDataFrame and return it
- **edges** (*bool*) – if True, convert graph edges to a GeoDataFrame and return it
- **node_geometry** (*bool*) – if True, create a geometry column from node x and y data

- **fill_edge_geometry** (*bool*) – if True, fill in missing edge geometry fields using nodes u and v

Returns gdf_nodes or gdf_edges or tuple of (gdf_nodes, gdf_edges)

Return type geopandas.GeoDataFrame or tuple

osmnx.utils_graph.**remove_isolated_nodes**(*G*)

Remove from a graph all nodes that have no incident edges.

Parameters **G** (*networkx.MultiDiGraph*) – graph from which to remove isolated nodes

Returns **G** – graph with all isolated nodes removed

Return type networkx.MultiDiGraph

3.2 Internals reference

This is the complete OSMnx internals reference, including private internal functions. If you are looking for the user reference to OSMnx's public-facing API, you can find it here.

3.2.1 osmnx.bearing module

Calculate graph edge bearings.

osmnx.bearing.**add_edge_bearings**(*G*, *precision=1*)

Add *bearing* attributes to all graph edges.

Calculate the compass bearing from origin node to destination node for each edge in the directed graph then add each bearing as a new edge attribute. Bearing represents angle in degrees (clockwise) between north and the direction from the origin node to the destination node.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **precision** (*int*) – decimal precision to round bearing

Returns **G** – graph with edge bearing attributes

Return type networkx.MultiDiGraph

osmnx.bearing.**get_bearing**(*origin_point*, *destination_point*)

Calculate the bearing between two lat-lng points.

Each argument tuple should represent (lat, lng) as decimal degrees. Bearing represents angle in degrees (clockwise) between north and the direction from the origin point to the destination point.

Parameters

- **origin_point** (*tuple*) – (lat, lng)
- **destination_point** (*tuple*) – (lat, lng)

Returns **bearing** – the compass bearing in decimal degrees from the origin point to the destination point

Return type float

3.2.2 osmnx.distance module

Calculate distances and shortest paths and find nearest node/edge(s) to point(s).

osmnx.distance.**euclidean_dist_vec**(*y1*, *x1*, *y2*, *x2*)
Calculate Euclidean distances between points.

Vectorized function to calculate the Euclidean distance between two points' coordinates or between arrays of points' coordinates. For most accurate results, use projected coordinates rather than decimal degrees.

Parameters

- **y1** (*float or np.array of float*) – first point's y coordinate
- **x1** (*float or np.array of float*) – first point's x coordinate
- **y2** (*float or np.array of float*) – second point's y coordinate
- **x2** (*float or np.array of float*) – second point's x coordinate

Returns **dist** – distance or array of distances from (x1, y1) to (x2, y2) in coordinates' units

Return type float or np.array of float

osmnx.distance.**get_nearest_edge**(*G*, *point*, *return_geom=False*, *return_dist=False*)
Find the nearest edge to a point by minimum Euclidean distance.

For best results, both G and point should be projected.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **point** (*tuple*) – the (lat, lng) or (y, x) point for which we will find the nearest edge in the graph
- **return_geom** (*bool*) – Optionally return the geometry of the nearest edge
- **return_dist** (*bool*) – Optionally return the distance in graph's coordinates' units between the point and the nearest edge

Returns Graph edge unique identifier as a tuple of (u, v, key). Or a tuple of (u, v, key, geom) if return_geom is True. Or a tuple of (u, v, key, dist) if return_dist is True. Or a tuple of (u, v, key, geom, dist) if return_geom and return_dist are True.

Return type tuple

osmnx.distance.**get_nearest_edges**(*G*, *X*, *Y*, *method=None*, *dist=0.0001*)
Find the nearest edge to each point in a list of points.

Pass in points as separate lists of X and Y coordinates. The 'kdtree' method is by far the fastest with large data sets, but only finds approximate nearest edges if working in unprojected coordinates like lat-lng (it precisely finds the nearest edge if working in projected coordinates). The 'balltree' method is second fastest with large data sets, but it is precise if working in unprojected coordinates like lat-lng. As a rule of thumb, if you have a small graph just use method=None. If you have a large graph with lat-lng coordinates, use method='balltree'. If you have a large graph with projected coordinates, use method='kdtree'. Note that if you are working in units of lat-lng, the X vector corresponds to longitude and the Y vector corresponds to latitude. The method creates equally distanced points along the edges of the network. Then, these points are used in a kdTree or BallTree search to identify which is nearest. Note that this method will not give exact perpendicular point along the edge, but the smaller the *dist* parameter, the closer (but slower) the solution will be.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph

- **X** (*list-like*) – the longitudes or x coordinates for which we will find the nearest edge in the graph. For projected graphs use the projected coordinates, usually in meters.
- **Y** (*list-like*) – the latitudes or y coordinates for which we will find the nearest edge in the graph. For projected graphs use the projected coordinates, usually in meters.
- **method** (*string {None, 'kdtree', 'balltree'}*) – Which method to use for finding nearest edge to each point. If None, we manually find each edge one at a time using get_nearest_edge. If 'kdtree' we use scipy.spatial.cKDTree for very fast euclidean search. Recommended for projected graphs. If 'balltree', we use sklearn.neighbors.BallTree for fast haversine search. Recommended for unprojected graphs.
- **dist** (*float*) – spacing length along edges. Units are the same as the graph's geometries. The smaller the value, the more points are created.

Returns **ne** – array of edge IDs representing the edge nearest to each point in the passed-in list of points. Edge IDs are represented by u, v, key where u and v the node IDs of the nodes the edge links.

Return type np.array

osmnx.distance.**get_nearest_node**(*G*, *point*, *method='haversine'*, *return_dist=False*)

Find the nearest node to a point.

Return the graph node nearest to some (lat, lng) or (y, x) point and optionally the distance between the node and the point. This function can use either the haversine formula or Euclidean distance.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **point** (*tuple*) – The (lat, lng) or (y, x) point for which we will find the nearest node in the graph
- **method** (*string {'haversine', 'euclidean'}*) – Which method to use for calculating distances to find nearest node. If 'haversine', graph nodes' coordinates must be in units of decimal degrees. If 'euclidean', graph nodes' coordinates must be projected.
- **return_dist** (*bool*) – Optionally also return the distance (in meters if haversine, or graph node coordinate units if euclidean) between the point and the nearest node

Returns Nearest node ID or optionally a tuple of (node ID, dist), where dist is the distance (in meters if haversine, or graph node coordinate units if euclidean) between the point and nearest node

Return type int or tuple of (int, float)

osmnx.distance.**get_nearest_nodes**(*G*, *X*, *Y*, *method=None*)

Find the nearest node to each point in a list of points.

Pass in points as separate lists of X and Y coordinates. The 'kdtree' method is by far the fastest with large data sets, but only finds approximate nearest nodes if working in unprojected coordinates like lat-lng (it precisely finds the nearest node if working in projected coordinates). The 'balltree' method is second fastest with large data sets but it is precise if working in unprojected coordinates like lat-lng.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **X** (*list-like*) – the longitudes or x coordinates for which we will find the nearest node in the graph
- **Y** (*list-like*) – the latitudes or y coordinates for which we will find the nearest node in the graph

- **method** (*string {None, 'kdtree', 'balltree'}*) – Which method to use for finding the nearest node to each point. If None, we manually find each node one at a time using utils.get_nearest_node and haversine. If 'kdtree' we use scipy.spatial.cKDTree for very fast euclidean search. If 'balltree', we use sklearn.neighbors.BallTree for fast haversine search.

Returns **nn** – array of node IDs representing the node nearest to each point in the passed-in list of points

Return type np.array

osmnx.distance.**great_circle_vec**(*lat1*, *lng1*, *lat2*, *lng2*, *earth_radius=6371009*)

Calculate great-circle distances between points.

Vectorized function to calculate the great-circle distance between two points' coordinates or between arrays of points' coordinates using the haversine formula. Expects coordinates in decimal degrees.

Parameters

- **lat1** (*float or np.array of float*) – first point's latitude coordinate
- **lng1** (*float or np.array of float*) – first point's longitude coordinate
- **lat2** (*float or np.array of float*) – second point's latitude coordinate
- **lng2** (*float or np.array of float*) – second point's longitude coordinate
- **earth_radius** (*int or float*) – radius of earth in units in which distance will be returned (default is meters)

Returns **dist** – distance or array of distances from (lat1, lng1) to (lat2, lng2) in units of earth_radius

Return type float or np.array

osmnx.distance.**k_shortest_paths**(*G*, *orig*, *dest*, *k*, *weight='length'*)

Get k shortest paths from origin node to destination node.

See also *shortest_path* to get just the one shortest path.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **orig** (*int*) – origin node ID
- **dest** (*int*) – destination node ID
- **k** (*int*) – number of shortest paths to get
- **weight** (*string*) – edge attribute to minimize when solving shortest paths. default is edge length in meters.

Returns a generator of k shortest paths ordered by total weight. each path is a list of node IDs.

Return type generator

osmnx.distance.**shortest_path**(*G*, *orig*, *dest*, *weight='length'*)

Get shortest path from origin node to destination node.

See also *k_shortest_paths* to get multiple shortest paths.

This function is a convenience wrapper around networkx.shortest_path. For more functionality or different algorithms, use networkx directly.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph

- **orig** (*int*) – origin node ID
- **dest** (*int*) – destination node ID
- **weight** (*string*) – edge attribute to minimize when solving shortest path. default is edge length in meters.

Returns **path** – list of node IDs consituting the shortest path

Return type list

3.2.3 osmnx.downloader module

Interact with the OSM APIs.

osmnx.downloader.**_create_overpass_query**(*polygon_coord_str*, *tags*)
Create an overpass query string based on passed tags.

Parameters

- **polygon_coord_str** (*list*) – list of lat lng coordinates
- **tags** (*dict*) – dict of tags used for finding elements in the selected area

Returns **query**

Return type string

osmnx.downloader.**_get_http_headers**(*user_agent=None*, *referer=None*, *accept_language=None*)
Update the default requests HTTP headers with OSMnx info.

Parameters

- **user_agent** (*string*) – the user agent string, if None will set with OSMnx default
- **referer** (*string*) – the referer string, if None will set with OSMnx default
- **accept_language** (*string*) – make accept-language explicit e.g. for consistent nominatim result sorting

Returns **headers**

Return type dict

osmnx.downloader.**_get_osm_filter**(*network_type*)
Create a filter to query OSM for the specified network type.

Parameters **network_type** (*string {"all_private", "all", "bike", "drive", "drive_service", "walk"}*) – what type of street network to get

Returns

Return type string

osmnx.downloader.**_get_pause**(*recursive_delay=5*, *default_duration=60*)
Get a pause duration from the Overpass API status endpoint.

Check the Overpass API status endpoint to determine how long to wait until next slot is available.

Parameters

- **recursive_delay** (*int*) – how long to wait between recursive calls if the server is currently running a query
- **default_duration** (*int*) – if fatal error, fall back on returning this value

Returns pause

Return type int

osmnx.downloader.**_make_overpass_polygon_coord_strs**(*polygon*)
Subdivide query polygon and return list of coordinate strings.

Project to utm, divide polygon up into sub-polygons if area exceeds a max size (in meters), project back to lat-lng, then get a list of polygon(s) exterior coordinates

Parameters **polygon** (*shapely.geometry.Polygon or shapely.geometry.MultiPolygon*) – geographic boundaries to fetch the OSM geometries within

Returns **polygon_coord_strs** – list of exterior coordinate strings for smaller sub-divided polygons

Return type list

osmnx.downloader.**_make_overpass_settings**()
Make settings string to send in Overpass query.

Returns

Return type string

osmnx.downloader.**_osm_geometries_download**(*polygon*, *tags*)
Retrieve non-networked elements within boundary from the Overpass API.

Parameters

- **polygon** (*shapely.geometry.Polygon*) – boundaries to fetch elements within
- **tags** (*dict*) – dict of tags used for finding elements in the selected area

Returns **response_jsons** – list of JSON responses from the Overpass server

Return type list

osmnx.downloader.**_osm_network_download**(*polygon*, *network_type*, *custom_filter*)
Retrieve networked ways and nodes within boundary from the Overpass API.

Parameters

- **polygon** (*shapely.geometry.Polygon or shapely.geometry.MultiPolygon*) – boundary to fetch the network ways/nodes within
- **network_type** (*string*) – what type of street network to get if custom_filter is None
- **custom_filter** (*string*) – a custom ways filter to be used instead of the network_type presets

Returns **response_jsons** – list of JSON responses from the Overpass server

Return type list

osmnx.downloader.**_osm_place_download**(*query*, *by_osmid=False*, *limit=1*, *polygon_geojson=1*)
Retrieve a place from the Nominatim API.

Parameters

- **query** (*string or dict*) – query string or structured query dict
- **by_osmid** (*bool*) – if True, handle query as an OSM ID for lookup rather than text search
- **limit** (*int*) – max number of results to return
- **polygon_geojson** (*int*) – retrieve the place's geometry from the API, 0=no, 1=yes

Returns **response_json** – JSON response from the Nominatim server

Return type dict

osmnx.downloader.**_retrieve_from_cache**(*url*, *check_remark=False*)
Retrieve a HTTP response JSON object from the cache, if it exists.

Parameters

- **url** (*string*) – the URL of the request
- **check_remark** (*string*) – if True, only return filepath if cached response does not have a remark key indicating a server warning

Returns **response_json** – cached response for url if it exists in the cache, otherwise None

Return type dict

osmnx.downloader.**_save_to_cache**(*url*, *response_json*, *sc*)
Save a HTTP response JSON object to a file in the cache folder.

Function calculates the checksum of url to generate the cache file's name. If the request was sent to server via POST instead of GET, then URL should be a GET-style representation of request. Response is only saved to a cache file if settings.use_cache is True, response_json is not None, and sc = 200.

Users should always pass OrderedDicts instead of dicts of parameters into request functions, so the parameters remain in the same order each time, producing the same URL string, and thus the same hash. Otherwise the cache will eventually contain multiple saved responses for the same request because the URL's parameters appeared in a different order each time.

Parameters

- **url** (*string*) – the URL of the request
- **response_json** (*dict*) – the JSON response
- **sc** (*int*) – the response's HTTP status code

Returns

Return type None

osmnx.downloader.**_url_in_cache**(*url*)
Determine if a URL's response exists in the cache.

Calculates the checksum of url to determine the cache file's name.

Parameters **url** (*string*) – the URL to look for in the cache

Returns **filepath** – path to cached response for url if it exists, otherwise None

Return type pathlib.Path

osmnx.downloader.**nominatim_request**(*params*, *request_type='search'*, *pause=1*, *error_pause=60*)
Send a HTTP GET request to the Nominatim API and return JSON response.

Parameters

- **params** (*OrderedDict*) – key-value pairs of parameters
- **request_type** (*string {"search", "reverse", "lookup"}*) – which Nominatim API endpoint to query
- **pause** (*int*) – how long to pause before request, in seconds. per the nominatim usage policy: "an absolute maximum of 1 request per second" is allowed
- **error_pause** (*int*) – how long to pause in seconds before re-trying request if error

Returns **response_json**

Return type dict

osmnx.downloader.**overpass_request**(*data*, *pause=None*, *error_pause=60*)
Send a HTTP POST request to the Overpass API and return JSON response.

Parameters

- **data** (*OrderedDict*) – key-value pairs of parameters
- **pause** (*int*) – how long to pause in seconds before request, if None, will query API status endpoint to find when next slot is available
- **error_pause** (*int*) – how long to pause in seconds (in addition to *pause*) before re-trying request if error

Returns response_json

Return type dict

3.2.4 osmnx.elevation module

Get node elevations and calculate edge grades.

osmnx.elevation.**add_edge_grades**(*G*, *add_absolute=True*, *precision=3*)
Add *grade* attribute to each graph edge.

Get the directed grade (ie, rise over run) for each edge in the graph and add it to the edge as an attribute. Nodes must have *elevation* attributes to use this function.

See also the *add_node_elevations* function.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **add_absolute** (*bool*) – if True, also add absolute value of grade as *grade_abs* attribute
- **precision** (*int*) – decimal precision to round grade values

Returns **G** – graph with edge *grade* (and optionally *grade_abs*) attributes

Return type networkx.MultiDiGraph

osmnx.elevation.**add_node_elevations**(*G*, *api_key*, *max_locations_per_batch=350*, *pause_duration=0.02*, *precision=3*)
Add *elevation* (meters) attribute to each node.

Uses the Google Maps Elevation API by default, but you can configure this to a different provider via ox.config()

See also the *add_edge_grades* function.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **api_key** (*string*) – your google maps elevation API key, or equivalent if using a different provider
- **max_locations_per_batch** (*int*) – max number of coordinate pairs to submit in each API call (if this is too high, the server will reject the request because its character limit exceeds the max)
- **pause_duration** (*float*) – time to pause between API calls
- **precision** (*int*) – decimal precision to round elevation

Returns **G** – graph with node elevation attributes

Return type networkx.MultiDiGraph

3.2.5 osmnx.folium module

Create interactive Leaflet web maps of graphs and routes via folium.

osmnx.folium.**_make_folium_polyline**(*geom*, *popup_val=None*, ***kwargs*)

Turn LineString geometry into a folium PolyLine with attributes.

Parameters

- **geom** (*shapely LineString*) – geometry of the line
- **popup_val** (*string*) – text to display in pop-up when a line is clicked, if None, no popup
- **kwargs** – keyword arguments to pass to folium.PolyLine()

Returns **pl**

Return type folium.PolyLine

osmnx.folium.**_plot_folium**(*gdf*, *m*, *popup_attribute*, *tiles*, *zoom*, *fit_bounds*, ***kwargs*)

Plot a GeoDataFrame of LineStrings on a folium map object.

Parameters

- **gdf** (*geopandas.GeoDataFrame*) – a GeoDataFrame of LineString geometries and attributes
- **m** (*folium.folium.Map or folium.FeatureGroup*) – if not None, plot on this preexisting folium map object
- **popup_attribute** (*string*) – attribute to display in pop-up on-click, if None, no popup
- **tiles** (*string*) – name of a folium tileset
- **zoom** (*int*) – initial zoom level for the map
- **fit_bounds** (*bool*) – if True, fit the map to gdf's boundaries
- **kwargs** – keyword arguments to pass to folium.PolyLine()

Returns **m**

Return type folium.folium.Map

osmnx.folium.**plot_graph_folium**(*G*, *graph_map=None*, *popup_attribute=None*, *tiles='cartodbpositron'*, *zoom=1*, *fit_bounds=True*, *edge_color=None*, *edge_width=None*, *edge_opacity=None*, ***kwargs*)

Plot a graph as an interactive Leaflet web map.

Note that anything larger than a small city can produce a large web map file that is slow to render in your browser.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **graph_map** (*folium.folium.Map*) – if not None, plot the graph on this preexisting folium map object

- **popup_attribute** (*string*) – edge attribute to display in a pop-up when an edge is clicked
- **tiles** (*string*) – name of a folium tileset
- **zoom** (*int*) – initial zoom level for the map
- **fit_bounds** (*bool*) – if True, fit the map to the boundaries of the graph's edges
- **edge_color** (*string*) – deprecated, do not use, use kwargs instead
- **edge_width** (*numeric*) – deprecated, do not use, use kwargs instead
- **edge_opacity** (*numeric*) – deprecated, do not use, use kwargs instead
- **kwargs** – keyword arguments to pass to folium.PolyLine(), see folium docs for options (for example *color="#333333", weight=5, opacity=0.7*)

Returns

Return type folium.folium.Map

osmnx.folium.**plot_route_folium**(*G*, *route*, *route_map=None*, *popup_attribute=None*, *tiles='cartodbpositron'*, *zoom=1*, *fit_bounds=True*, *route_color=None*, *route_width=None*, *route_opacity=None*, ***kwargs*)

Plot a route as an interactive Leaflet web map.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **route** (*list*) – the route as a list of nodes
- **route_map** (*folium.folium.Map*) – if not None, plot the route on this preexisting folium map object
- **popup_attribute** (*string*) – edge attribute to display in a pop-up when an edge is clicked
- **tiles** (*string*) – name of a folium tileset
- **zoom** (*int*) – initial zoom level for the map
- **fit_bounds** (*bool*) – if True, fit the map to the boundaries of the route's edges
- **route_color** (*string*) – deprecated, do not use, use kwargs instead
- **route_width** (*numeric*) – deprecated, do not use, use kwargs instead
- **route_opacity** (*numeric*) – deprecated, do not use, use kwargs instead
- **kwargs** – keyword arguments to pass to folium.PolyLine(), see folium docs for options (for example *color="#cc0000", weight=5, opacity=0.7*)

Returns

Return type folium.folium.Map

3.2.6 osmnx.geocoder module

Geocode queries and create GeoDataFrames of place boundaries.

osmnx.geocoder.**_geocode_query_to_gdf**(*query*, *which_result*, *by_osmid*)
Geocode a single place query to a GeoDataFrame.

Parameters

- **query** (*string or dict*) – query string or structured dict to geocode
- **which_result** (*int*) – which geocoding result to use. if None, auto-select the first (Multi)Polygon or raise an error if OSM doesn't return one. to get the top match regardless of geometry type, set which_result=1
- **by_osmid** (*bool*) – if True, handle query as an OSM ID for lookup rather than text search

Returns **gdf** – a GeoDataFrame with one row containing the result of geocoding

Return type geopandas.GeoDataFrame

osmnx.geocoder.**_get_first_polygon**(*results*, *query*)
Choose first result of geometry type (Multi)Polygon from list of results.

Parameters

- **results** (*list*) – list of results from downloader._osm_place_download
- **query** (*str*) – the query string or structured dict that was geocoded

Returns **result** – the chosen result

Return type dict

osmnx.geocoder.**geocode**(*query*)
Geocode a query string to (lat, lng) with the Nominatim geocoder.

Parameters **query** (*string*) – the query string to geocode

Returns **point** – the (lat, lng) coordinates returned by the geocoder

Return type tuple

osmnx.geocoder.**geocode_to_gdf**(*query*, *which_result=None*, *by_osmid=False*, *buffer_dist=None*)
Retrieve place(s) by name or ID from the Nominatim API as a GeoDataFrame.

You can query by place name or OSM ID. If querying by place name, the query argument can be a string or structured dict, or a list of such strings/dicts to send to geocoder. You can instead query by OSM ID by setting *by_osmid=True*. In this case, geocode_to_gdf treats the query argument as an OSM ID (or list of OSM IDs) for Nominatim lookup rather than text search. OSM IDs must be prepended with their types: node (N), way (W), or relation (R), in accordance with the Nominatim format. For example, *query=["R2192363", "N240109189", "W427818536"]*.

If query argument is a list, then which_result should be either a single value or a list with the same length as query. The queries you provide must be resolvable to places in the Nominatim database. The resulting GeoDataFrame's geometry column contains place boundaries if they exist in OpenStreetMap.

Parameters

- **query** (*string or dict or list*) – query string(s) or structured dict(s) to geocode
- **which_result** (*int*) – which geocoding result to use. if None, auto-select the first (Multi)Polygon or raise an error if OSM doesn't return one. to get the top match regardless of geometry type, set which_result=1
- **by_osmid** (*bool*) – if True, handle query as an OSM ID for lookup rather than text search

- **buffer_dist** (*float*) – distance to buffer around the place geometry, in meters

Returns **gdf** – a GeoDataFrame with one row for each query

Return type geopandas.GeoDataFrame

3.2.7 osmnx.geometries module

Download geospatial entities' geometries and attributes from OpenStreetMap.

Retrieve points of interest, building footprints, or any other objects from OSM, including their geometries and attribute data, and construct a GeoDataFrame of them.

osmnx.geometries.**_assemble_multipolygon_component_polygons**(*element*, *geometries*)
Assemble a MultiPolygon from its component LineStrings and Polygons.

The OSM wiki suggests an algorithm for assembling multipolygon geometries https://wiki.openstreetmap.org/wiki/Relation:multipolygon/Algorithm. This method takes a simpler approach relying on the accurate tagging of component ways with 'inner' and 'outer' roles as required on this page https://wiki.openstreetmap.org/wiki/Relation:multipolygon.

Parameters

- **element** (*dict*) – element type "relation" from overpass response JSON
- **geometries** (*dict*) – dict containing all linestrings and polygons generated from OSM ways

Returns **geometry** – a single MultiPolygon object

Return type shapely.geometry.MultiPolygon

osmnx.geometries.**_buffer_invalid_geometries**(*gdf*)
Buffer any invalid geometries remaining in the GeoDataFrame.

Invalid geometries in the GeoDataFrame (which may accurately reproduce invalid geometries in OpenStreetMap) can cause the filtering to the query polygon and other subsequent geometric operations to fail. This function logs the ids of the invalid geometries and applies a buffer of zero to try to make them valid.

Note: the resulting geometries may differ from the originals - please check them against OpenStreetMap

Parameters **gdf** (*geopandas.GeoDataFrame*) – a GeoDataFrame with possibly invalid geometries

Returns **gdf** – the GeoDataFrame with .buffer(0) applied to invalid geometries

Return type geopandas.GeoDataFrame

osmnx.geometries.**_create_gdf**(*response_jsons*, *polygon*, *tags*)
Parse JSON responses from the Overpass API to a GeoDataFrame.

Note: the *polygon* and *tags* arguments can both be *None* and the GeoDataFrame will still be created but it won't be filtered at the end i.e. the final GeoDataFrame will contain all tagged geometries in the *response_jsons*.

Parameters

- **response_jsons** (*list*) – list of JSON responses from from the Overpass API
- **polygon** (*shapely.geometry.Polygon*) – geographic boundary used for filtering the final GeoDataFrame
- **tags** (*dict*) – dict of tags used for filtering the final GeoDataFrame

Returns **gdf** – GeoDataFrame of geometries and their associated tags

Return type geopandas.GeoDataFrame

osmnx.geometries.**_filter_gdf_by_polygon_and_tags**(*gdf*, *polygon*, *tags*)

Filter the GeoDataFrame to the requested bounding polygon and tags.

Filters GeoDataFrame to query polygon and tags. Removes columns of all NaNs (that held values only in rows removed by the filters). Resets the index of GeoDataFrame, writing it into a new column called 'unique_id'.

Parameters

- **gdf** (*geopandas.GeoDataFrame*) – the GeoDataFrame to filter
- **polygon** (*shapely.geometry.Polygon*) – polygon defining the boundary of the requested area
- **tags** (*dict*) – the tags requested

Returns **gdf** – final filtered GeoDataFrame

Return type geopandas.GeoDataFrame

osmnx.geometries.**_is_closed_way_a_polygon**(*element*, *polygon_features={'aeroway': {'polygon': 'blocklist', 'values': ['taxiway']}, 'amenity': {'polygon': 'all'}, 'area': {'polygon': 'all'}, 'area:highway': {'polygon': 'all'}, 'barrier': {'polygon': 'passlist', 'values': ['city_wall', 'ditch', 'hedge', 'retaining_wall', 'spikes']}, 'boundary': {'polygon': 'all'}, 'building': {'polygon': 'all'}, 'building:part': {'polygon': 'all'}, 'craft': {'polygon': 'all'}, 'golf': {'polygon': 'all'}, 'highway': {'polygon': 'passlist', 'values': ['services', 'rest_area', 'escape', 'elevator']}, 'historic': {'polygon': 'all'}, 'indoor': {'polygon': 'all'}, 'landuse': {'polygon': 'all'}, 'leisure': {'polygon': 'all'}, 'man_made': {'polygon': 'blocklist', 'values': ['cutline', 'embankment', 'pipeline']}, 'military': {'polygon': 'all'}, 'natural': {'polygon': 'blocklist', 'values': ['coastline', 'cliff', 'ridge', 'arete', 'tree_row']}, 'office': {'polygon': 'all'}, 'place': {'polygon': 'all'}, 'power': {'polygon': 'passlist', 'values': ['plant', 'substation', 'generator', 'transformer']}, 'public_transport': {'polygon': 'all'}, 'railway': {'polygon': 'passlist', 'values': ['station', 'turntable', 'roundhouse', 'platform']}, 'ruins': {'polygon': 'all'}, 'shop': {'polygon': 'all'}, 'tourism': {'polygon': 'all'}, 'waterway': {'polygon': 'passlist', 'values': ['riverbank', 'dock', 'boatyard', 'dam']}}*)

Determine whether a closed OSM way represents a Polygon, not a LineString.

Closed OSM ways may represent LineStrings (e.g. a roundabout or hedge round a field) or Polygons (e.g. a building footprint or land use area) depending on the tags applied to them.

The starting assumption is that it is not a polygon, however any polygon type tagging will return a polygon unless explicitly tagged with area:no.

It is possible for a single closed OSM way to have both LineString and Polygon type tags (e.g. both barrier=fence and landuse=agricultural). OSMnx will return a single Polygon for elements tagged in this way. For more information see: https://wiki.openstreetmap.org/wiki/One_feature,_one_OSM_element)

Parameters

- **element** (*dict*) – closed element type "way" from overpass response JSON
- **polygon_features** (*dict*) – dict of tag keys with associated values and block-list/passlist

Returns **is_polygon** – True if the tags are for a polygon type geometry

Return type bool

osmnx.geometries.**_parse_node_to_coords**(*element*)
Parse coordinates from a node in the overpass response.

The coords are only used to create LineStrings and Polygons.

Parameters **element** (*dict*) – element type "node" from overpass response JSON

Returns **coords** – dict of latitude/longitude coordinates

Return type dict

osmnx.geometries.**_parse_node_to_point**(*element*)
Parse point from a tagged node in the overpass response.

The points are geometries in their own right.

Parameters **element** (*dict*) – element type "node" from overpass response JSON

Returns **point** – dict of OSM ID, OSM element type, tags and geometry

Return type dict

osmnx.geometries.**_parse_relation_to_multipolygon**(*element*, *geometries*)
Parse multipolygon from OSM relation (type:MultiPolygon).

See more information about relations from OSM documentation: http://wiki.openstreetmap.org/wiki/Relation

Parameters

- **element** (*dict*) – element type "relation" from overpass response JSON
- **geometries** (*dict*) – dict containing all linestrings and polygons generated from OSM ways

Returns **multipolygon** – dict of tags and geometry for a single multipolygon

Return type dict

osmnx.geometries.**_parse_way_to_linestring_or_polygon**(*element*, *coords*, *polygon_features={'aeroway': {'polygon': 'blocklist', 'values': ['taxiway']}, 'amenity': {'polygon': 'all'}, 'area': {'polygon': 'all'}, 'area:highway': {'polygon': 'all'}, 'barrier': {'polygon': 'passlist', 'values': ['city_wall', 'ditch', 'hedge', 'retaining_wall', 'spikes']}, 'boundary': {'polygon': 'all'}, 'building': {'polygon': 'all'}, 'building:part': {'polygon': 'all'}, 'craft': {'polygon': 'all'}, 'golf': {'polygon': 'all'}, 'highway': {'polygon': 'passlist', 'values': ['services', 'rest_area', 'escape', 'elevator']}, 'historic': {'polygon': 'all'}, 'indoor': {'polygon': 'all'}, 'landuse': {'polygon': 'all'}, 'leisure': {'polygon': 'all'}, 'man_made': {'polygon': 'blocklist', 'values': ['cutline', 'embankment', 'pipeline']}, 'military': {'polygon': 'all'}, 'natural': {'polygon': 'blocklist', 'values': ['coastline', 'cliff', 'ridge', 'arete', 'tree_row']}, 'office': {'polygon': 'all'}, 'place': {'polygon': 'all'}, 'power': {'polygon': 'passlist', 'values': ['plant', 'substation', 'generator', 'transformer']}, 'public_transport': {'polygon': 'all'}, 'railway': {'polygon': 'passlist', 'values': ['station', 'turntable', 'roundhouse', 'platform']}, 'ruins': {'polygon': 'all'}, 'shop': {'polygon': 'all'}, 'tourism': {'polygon': 'all'}, 'waterway': {'polygon': 'passlist', 'values': ['riverbank', 'dock', 'boatyard', 'dam']}}*)

Parse open LineString, closed LineString or Polygon from OSM 'way'.

Please see https://wiki.openstreetmap.org/wiki/Overpass_turbo/Polygon_Features for more information on which tags should be parsed to polygons

Parameters

- **element** (*dict*) – element type "way" from overpass response JSON
- **coords** (*dict*) – dict of node IDs and their latitude/longitude coordinates
- **polygon_features** (*dict*) – dict for determining whether closed ways are LineStrings or Polygons

Returns **linestring_or_polygon** – dict of OSM ID, OSM element type, nodes, tags and geometry

Return type dict

osmnx.geometries.**_subtract_inner_polygons_from_outer_polygons**(*element*, *outer_polygons*, *inner_polygons*)

Subtract inner polygons from outer polygons.

Creates a Polygon or MultiPolygon with holes.

Parameters

- **element** (*dict*) – element type "relation" from overpass response JSON
- **outer_polygons** (*list*) – list of outer polygons that are part of a multipolygon
- **inner_polygons** (*list*) – list of inner polygons that are part of a multipolygon

Returns **geometry** – a single Polygon or MultiPolygon

Return type shapely.geometry.Polygon or shapely.geometry.MultiPolygon

osmnx.geometries.**geometries_from_address**(*address*, *tags*, *dist=1000*)

Create GeoDataFrame of OSM entities within some distance N, S, E, W of address.

Parameters

- **address** (*string*) – the address to geocode and use as the central point around which to get the geometries
- **tags** (*dict*) – Dict of tags used for finding objects in the selected area. Results returned are the union, not intersection of each individual tag. Each result matches at least one given tag. The dict keys should be OSM tags, (e.g., *building*, *landuse*, *highway*, etc) and the dict values should be either *True* to retrieve all items with the given tag, or a string to get a single tag-value combination, or a list of strings to get multiple values for the given tag. For example, *tags = {'building': True}* would return all building footprints in the area. *tags = {'amenity':True, 'landuse':['retail','commercial'], 'highway':'bus_stop'}* would return all amenities, landuse=retail, landuse=commercial, and highway=bus_stop.
- **dist** (*numeric*) – distance in meters

Returns **gdf**

Return type geopandas.GeoDataFrame

Notes

You can configure the Overpass server timeout, memory allocation, and other custom settings via ox.config().

osmnx.geometries.**geometries_from_bbox**(*north*, *south*, *east*, *west*, *tags*)

Create a GeoDataFrame of OSM entities within a N, S, E, W bounding box.

Parameters

- **north** (*float*) – northern latitude of bounding box
- **south** (*float*) – southern latitude of bounding box
- **east** (*float*) – eastern longitude of bounding box
- **west** (*float*) – western longitude of bounding box

- **tags** (*dict*) – Dict of tags used for finding objects in the selected area. Results returned are the union, not intersection of each individual tag. Each result matches at least one given tag. The dict keys should be OSM tags, (e.g., *building*, *landuse*, *highway*, etc) and the dict values should be either *True* to retrieve all items with the given tag, or a string to get a single tag-value combination, or a list of strings to get multiple values for the given tag. For example, *tags = {'building': True}* would return all building footprints in the area. *tags = {'amenity':True, 'landuse':['retail','commercial'], 'highway':'bus_stop'}* would return all amenities, landuse=retail, landuse=commercial, and highway=bus_stop.

Returns gdf

Return type geopandas.GeoDataFrame

Notes

You can configure the Overpass server timeout, memory allocation, and other custom settings via ox.config().

osmnx.geometries.**geometries_from_place**(*query*, *tags*, *which_result=None*, *buffer_dist=None*)
Create a GeoDataFrame of OSM entities within the boundaries of a place.

Parameters

- **query** (*string or dict or list*) – the query or queries to geocode to get place boundary polygon(s)
- **tags** (*dict*) – Dict of tags used for finding objects in the selected area. Results returned are the union, not intersection of each individual tag. Each result matches at least one given tag. The dict keys should be OSM tags, (e.g., *building*, *landuse*, *highway*, etc) and the dict values should be either *True* to retrieve all items with the given tag, or a string to get a single tag-value combination, or a list of strings to get multiple values for the given tag. For example, *tags = {'building': True}* would return all building footprints in the area. *tags = {'amenity':True, 'landuse':['retail','commercial'], 'highway':'bus_stop'}* would return all amenities, landuse=retail, landuse=commercial, and highway=bus_stop.
- **which_result** (*int*) – which geocoding result to use. if None, auto-select the first (Multi)Polygon or raise an error if OSM doesn't return one.
- **buffer_dist** (*float*) – distance to buffer around the place geometry, in meters

Returns gdf

Return type geopandas.GeoDataFrame

Notes

You can configure the Overpass server timeout, memory allocation, and other custom settings via ox.config().

osmnx.geometries.**geometries_from_point**(*center_point*, *tags*, *dist=1000*)
Create GeoDataFrame of OSM entities within some distance N, S, E, W of a point.

Parameters

- **center_point** (*tuple*) – the (lat, lng) center point around which to get the geometries
- **tags** (*dict*) – Dict of tags used for finding objects in the selected area. Results returned are the union, not intersection of each individual tag. Each result matches at least one given tag. The dict keys should be OSM tags, (e.g., *building*, *landuse*, *highway*, etc) and the dict values should be either *True* to retrieve all items with the given tag, or a string to get a single tag-value combination, or a list of strings to get multiple values for the given tag. For

example, *tags = {'building': True}* would return all building footprints in the area. *tags = {'amenity':True, 'landuse':['retail','commercial'], 'highway':'bus_stop'}* would return all amenities, landuse=retail, landuse=commercial, and highway=bus_stop.

- **dist** (*numeric*) – distance in meters

Returns gdf

Return type geopandas.GeoDataFrame

Notes

You can configure the Overpass server timeout, memory allocation, and other custom settings via ox.config().

osmnx.geometries.**geometries_from_polygon**(*polygon*, *tags*)
Create GeoDataFrame of OSM entities within boundaries of a (multi)polygon.

Parameters

- **polygon** (*shapely.geometry.Polygon or shapely.geometry.MultiPolygon*) – geographic boundaries to fetch geometries within
- **tags** (*dict*) – Dict of tags used for finding objects in the selected area. Results returned are the union, not intersection of each individual tag. Each result matches at least one given tag. The dict keys should be OSM tags, (e.g., *building*, *landuse*, *highway*, etc) and the dict values should be either *True* to retrieve all items with the given tag, or a string to get a single tag-value combination, or a list of strings to get multiple values for the given tag. For example, *tags = {'building': True}* would return all building footprints in the area. *tags = {'amenity':True, 'landuse':['retail','commercial'], 'highway':'bus_stop'}* would return all amenities, landuse=retail, landuse=commercial, and highway=bus_stop.

Returns gdf

Return type geopandas.GeoDataFrame

Notes

You can configure the Overpass server timeout, memory allocation, and other custom settings via ox.config().

osmnx.geometries.**geometries_from_xml**(*filepath*, *polygon=None*, *tags=None*)
Create a GeoDataFrame of OSM entities in an OSM-formatted XML file.

Because this function creates a GeoDataFrame of geometries from an OSM-formatted XML file that has already been downloaded (i.e. no query is made to the Overpass API) the polygon and tags arguments are not required. If they are not supplied to the function, geometries_from_xml() will return geometries for all of the tagged elements in the file. If they are supplied they will be used to filter the final GeoDataFrame.

Parameters

- **filepath** (*string or pathlib.Path*) – path to file containing OSM XML data
- **polygon** (*shapely.geometry.Polygon*) – optional geographic boundary to filter objects
- **tags** (*dict*) – optional dict of tags for filtering objects from the XML. Results returned are the union, not intersection of each individual tag. Each result matches at least one given tag. The dict keys should be OSM tags, (e.g., *building*, *landuse*, *highway*, etc) and the dict values should be either *True* to retrieve all items with the given tag, or a string to get a single tag-value combination, or a list of strings to get multiple values for the given tag. For example, *tags = {'building': True}* would return all building footprints in the area. *tags =*

{'amenity':True, 'landuse':['retail','commercial'], 'highway':'bus_stop'} would return all amenities, landuse=retail, landuse=commercial, and highway=bus_stop.

Returns **gdf**

Return type geopandas.GeoDataFrame

3.2.8 osmnx.graph module

Graph creation functions.

osmnx.graph.**_add_paths**(*G*, *paths*, *bidirectional=False*)
Add a list of paths to the graph as edges.

Parameters

- **G** (*networkx.MultiDiGraph*) – graph to add paths to
- **paths** (*list*) – list of paths' tag:value attribute data dicts
- **bidirectional** (*bool*) – if True, create bi-directional edges for one-way streets

Returns

Return type None

osmnx.graph.**_convert_node**(*element*)
Convert an OSM node element into the format for a networkx node.

Parameters **element** (*dict*) – an OSM node element

Returns **node**

Return type dict

osmnx.graph.**_convert_path**(*element*)
Convert an OSM way element into the format for a networkx path.

Parameters **element** (*dict*) – an OSM way element

Returns **path**

Return type dict

osmnx.graph.**_create_graph**(*response_jsons*, *retain_all=False*, *bidirectional=False*)
Create a networkx MultiDiGraph from Overpass API responses.

Adds length attributes in meters (great-circle distance between endpoints) to all of the graph's (pre-simplified, straight-line) edges via the *utils_graph.add_edge_lengths* function.

Parameters

- **response_jsons** (*list*) – list of dicts of JSON responses from from the Overpass API
- **retain_all** (*bool*) – if True, return the entire graph even if it is not connected. otherwise, retain only the largest weakly connected component.
- **bidirectional** (*bool*) – if True, create bi-directional edges for one-way streets

Returns **G**

Return type networkx.MultiDiGraph

osmnx.graph.**_is_path_one_way**(*path*, *bidirectional*, *oneway_values*)
Determine if a path of nodes allows travel in only one direction.

Parameters

- **path** (*dict*) – a path's tag:value attribute data
- **bidirectional** (*bool*) – whether this is a bi-directional network type
- **oneway_values** (*set*) – the values OSM uses in its 'oneway' tag to denote True

Returns

Return type bool

osmnx.graph.**_is_path_reversed**(*path*, *reversed_values*)
Determine if the order of nodes in a path should be reversed.

Parameters

- **path** (*dict*) – a path's tag:value attribute data
- **reversed_values** (*set*) – the values OSM uses in its 'oneway' tag to denote travel can only occur in the opposite direction of the node order

Returns

Return type bool

osmnx.graph.**_parse_nodes_paths**(*response_json*)
Construct dicts of nodes and paths from an Overpass response.

Parameters **response_json** (*dict*) – JSON response from the Overpass API

Returns **nodes, paths** – dicts' keys = osmid and values = dict of attributes

Return type tuple of dicts

osmnx.graph.**graph_from_address**(*address*, *dist=1000*, *dist_type='bbox'*, *network_type='all_private'*, *simplify=True*, *retain_all=False*, *truncate_by_edge=False*, *return_coords=False*, *clean_periphery=True*, *custom_filter=None*)
Create a graph from OSM within some distance of some address.

Parameters

- **address** (*string*) – the address to geocode and use as the central point around which to construct the graph
- **dist** (*int*) – retain only those nodes within this many meters of the center of the graph
- **dist_type** (*string {"network", "bbox"}*) – if "bbox", retain only those nodes within a bounding box of the distance parameter. if "network", retain only those nodes within some network distance from the center-most node.
- **network_type** (*string {"all_private", "all", "bike", "drive", "drive_service", "walk"}*) – what type of street network to get if custom_filter is None
- **simplify** (*bool*) – if True, simplify graph topology with the *simplify_graph* function
- **retain_all** (*bool*) – if True, return the entire graph even if it is not connected. otherwise, retain only the largest weakly connected component.
- **truncate_by_edge** (*bool*) – if True, retain nodes outside bounding box if at least one of node's neighbors is within the bounding box
- **return_coords** (*bool*) – optionally also return the geocoded coordinates of the address
- **clean_periphery** (*bool,*) – if True, buffer 500m to get a graph larger than requested, then simplify, then truncate it to requested spatial boundaries

- **custom_filter** (*string*) – a custom ways filter to be used instead of the network_type presets e.g., '["power"~"line"]' or '["highway"~"motorway|trunk"]'. Also pass in a network_type that is in settings.bidirectional_network_types if you want graph to be fully bidirectional.

Returns

Return type networkx.MultiDiGraph or optionally (networkx.MultiDiGraph, (lat, lng))

Notes

You can configure the Overpass server timeout, memory allocation, and other custom settings via ox.config().

osmnx.graph.**graph_from_bbox**(*north*, *south*, *east*, *west*, *network_type='all_private'*, *simplify=True*, *retain_all=False*, *truncate_by_edge=False*, *clean_periphery=True*, *custom_filter=None*)

Create a graph from OSM within some bounding box.

Parameters

- **north** (*float*) – northern latitude of bounding box
- **south** (*float*) – southern latitude of bounding box
- **east** (*float*) – eastern longitude of bounding box
- **west** (*float*) – western longitude of bounding box
- **network_type** (*string {"all_private", "all", "bike", "drive", "drive_service", "walk"}*) – what type of street network to get if custom_filter is None
- **simplify** (*bool*) – if True, simplify graph topology with the *simplify_graph* function
- **retain_all** (*bool*) – if True, return the entire graph even if it is not connected. otherwise, retain only the largest weakly connected component.
- **truncate_by_edge** (*bool*) – if True, retain nodes outside bounding box if at least one of node's neighbors is within the bounding box
- **clean_periphery** (*bool*) – if True, buffer 500m to get a graph larger than requested, then simplify, then truncate it to requested spatial boundaries
- **custom_filter** (*string*) – a custom ways filter to be used instead of the network_type presets e.g., '["power"~"line"]' or '["highway"~"motorway|trunk"]'. Also pass in a network_type that is in settings.bidirectional_network_types if you want graph to be fully bidirectional.

Returns G

Return type networkx.MultiDiGraph

Notes

You can configure the Overpass server timeout, memory allocation, and other custom settings via ox.config().

osmnx.graph.**graph_from_place**(*query*, *network_type='all_private'*, *simplify=True*, *retain_all=False*, *truncate_by_edge=False*, *which_result=None*, *buffer_dist=None*, *clean_periphery=True*, *custom_filter=None*)

Create graph from OSM within the boundaries of some geocodable place(s).

The query must be geocodable and OSM must have polygon boundaries for the geocode result. If OSM does not have a polygon for this place, you can instead get its street network using the graph_from_address function, which geocodes the place name to a point and gets the network within some distance of that point.

If OSM does have polygon boundaries for this place but you're not finding it, try to vary the query string, pass in a structured query dict, or vary the which_result argument to use a different geocode result. If you know the OSM ID of the place, you can retrieve its boundary polygon using the geocode_to_gdf function, then pass it to the graph_from_polygon function.

Parameters

- **query** (*string or dict or list*) – the query or queries to geocode to get place boundary polygon(s)
- **network_type** (*string {"all_private", "all", "bike", "drive", "drive_service", "walk"}*) – what type of street network to get if custom_filter is None
- **simplify** (*bool*) – if True, simplify graph topology with the *simplify_graph* function
- **retain_all** (*bool*) – if True, return the entire graph even if it is not connected. otherwise, retain only the largest weakly connected component.
- **truncate_by_edge** (*bool*) – if True, retain nodes outside boundary polygon if at least one of node's neighbors is within the polygon
- **which_result** (*int*) – which geocoding result to use. if None, auto-select the first (Multi)Polygon or raise an error if OSM doesn't return one.
- **buffer_dist** (*float*) – distance to buffer around the place geometry, in meters
- **clean_periphery** (*bool*) – if True, buffer 500m to get a graph larger than requested, then simplify, then truncate it to requested spatial boundaries
- **custom_filter** (*string*) – a custom ways filter to be used instead of the network_type presets e.g., '["power"~"line"]' or '["highway"~"motorway|trunk"]'. Also pass in a network_type that is in settings.bidirectional_network_types if you want graph to be fully bidirectional.

Returns G

Return type networkx.MultiDiGraph

Notes

You can configure the Overpass server timeout, memory allocation, and other custom settings via ox.config().

osmnx.graph.**graph_from_point**(*center_point*, *dist=1000*, *dist_type='bbox'*, *network_type='all_private'*, *simplify=True*, *retain_all=False*, *truncate_by_edge=False*, *clean_periphery=True*, *custom_filter=None*)

Create a graph from OSM within some distance of some (lat, lng) point.

Parameters

- **center_point** (*tuple*) – the (lat, lng) center point around which to construct the graph
- **dist** (*int*) – retain only those nodes within this many meters of the center of the graph, with distance determined according to dist_type argument
- **dist_type** (*string {"network", "bbox"}*) – if "bbox", retain only those nodes within a bounding box of the distance parameter. if "network", retain only those nodes within some network distance from the center-most node.
- **network_type** (*string, {"all_private", "all", "bike", "drive", "drive_service", "walk"}*) – what type of street network to get if custom_filter is None
- **simplify** (*bool*) – if True, simplify graph topology with the *simplify_graph* function
- **retain_all** (*bool*) – if True, return the entire graph even if it is not connected. otherwise, retain only the largest weakly connected component.
- **truncate_by_edge** (*bool*) – if True, retain nodes outside bounding box if at least one of node's neighbors is within the bounding box
- **clean_periphery** (*bool,*) – if True, buffer 500m to get a graph larger than requested, then simplify, then truncate it to requested spatial boundaries
- **custom_filter** (*string*) – a custom ways filter to be used instead of the network_type presets e.g., '["power"~"line"]' or '["highway"~"motorway|trunk"]'. Also pass in a network_type that is in settings.bidirectional_network_types if you want graph to be fully bidirectional.

Returns G

Return type networkx.MultiDiGraph

Notes

You can configure the Overpass server timeout, memory allocation, and other custom settings via ox.config().

osmnx.graph.**graph_from_polygon**(*polygon*, *network_type='all_private'*, *simplify=True*, *retain_all=False*, *truncate_by_edge=False*, *clean_periphery=True*, *custom_filter=None*)

Create a graph from OSM within the boundaries of some shapely polygon.

Parameters

- **polygon** (*shapely.geometry.Polygon or shapely.geometry.MultiPolygon*) – the shape to get network data within. coordinates should be in unprojected latitude-longitude degrees (EPSG:4326).
- **network_type** (*string {"all_private", "all", "bike", "drive", "drive_service", "walk"}*) – what type of street network to get if custom_filter is None

- **simplify** (*bool*) – if True, simplify graph topology with the *simplify_graph* function
- **retain_all** (*bool*) – if True, return the entire graph even if it is not connected. otherwise, retain only the largest weakly connected component.
- **truncate_by_edge** (*bool*) – if True, retain nodes outside boundary polygon if at least one of node's neighbors is within the polygon
- **clean_periphery** (*bool*) – if True, buffer 500m to get a graph larger than requested, then simplify, then truncate it to requested spatial boundaries
- **custom_filter** (*string*) – a custom ways filter to be used instead of the network_type presets e.g., '["power"~"line"]' or '["highway"~"motorway|trunk"]'. Also pass in a network_type that is in settings.bidirectional_network_types if you want graph to be fully bidirectional.

Returns G

Return type networkx.MultiDiGraph

Notes

You can configure the Overpass server timeout, memory allocation, and other custom settings via ox.config().

osmnx.graph.**graph_from_xml**(*filepath*, *bidirectional=False*, *simplify=True*, *retain_all=False*)
Create a graph from data in a .osm formatted XML file.

Parameters

- **filepath** (*string or pathlib.Path*) – path to file containing OSM XML data
- **bidirectional** (*bool*) – if True, create bi-directional edges for one-way streets
- **simplify** (*bool*) – if True, simplify graph topology with the *simplify_graph* function
- **retain_all** (*bool*) – if True, return the entire graph even if it is not connected. otherwise, retain only the largest weakly connected component.

Returns G

Return type networkx.MultiDiGraph

3.2.9 osmnx.io module

Serialize graphs to/from files on disk.

osmnx.io.**_convert_edge_attr_types**(*G*, *dtypes=None*)
Convert graph edges' attributes using a dict of data types.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **dtypes** (*dict*) – dict of edge attribute names:types

Returns G

Return type networkx.MultiDiGraph

osmnx.io.**_convert_node_attr_types**(*G*, *dtypes=None*)
Convert graph nodes' attributes using a dict of data types.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **dtypes** (*dict*) – dict of node attribute names:types

Returns G

Return type networkx.MultiDiGraph

osmnx.io.**_stringify_nonnumeric_cols**(*gdf*)
Make every non-numeric GeoDataFrame column (besides geometry) a string.

This allows proper serializing via Fiona of GeoDataFrames with mixed types such as strings and ints in the same column.

Parameters **gdf** (*geopandas.GeoDataFrame*) – gdf to stringify non-numeric columns of

Returns **gdf** – gdf with non-numeric columns stringified

Return type geopandas.GeoDataFrame

osmnx.io.**load_graphml**(*filepath*, *node_dtypes=None*, *edge_dtypes=None*)
Load an OSMnx-saved GraphML file from disk.

Converts the node/edge attributes to appropriate data types, which can be customized if needed by passing in *node_dtypes* or *edge_dtypes* arguments.

Parameters

- **filepath** (*string or pathlib.Path*) – path to the GraphML file
- **node_dtypes** (*dict*) – dict of node attribute names:types to convert values' data types
- **edge_dtypes** (*dict*) – dict of edge attribute names:types to convert values' data types

Returns G

Return type networkx.MultiDiGraph

osmnx.io.**save_graph_geopackage**(*G*, *filepath=None*, *encoding='utf-8'*, *directed=False*)
Save graph nodes and edges to disk as layers in a GeoPackage file.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **filepath** (*string or pathlib.Path*) – path to the GeoPackage file including extension. if None, use default data folder + graph.gpkg
- **encoding** (*string*) – the character encoding for the saved file
- **directed** (*bool*) – if False, save one edge for each undirected edge in the graph but retain original oneway and to/from information as edge attributes; if True, save one edge for each directed edge in the graph

Returns

Return type None

osmnx.io.**save_graph_shapefile**(*G*, *filepath=None*, *encoding='utf-8'*, *directed=False*)
Save graph nodes and edges to disk as ESRI shapefiles.

The shapefile format is proprietary and outdated. Whenever possible, you should use the superior GeoPackage file format instead via the save_graph_geopackage function.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph

- **filepath** (*string or pathlib.Path*) – path to the shapefiles folder (no file extension). if None, use default data folder + graph_shapefile
- **encoding** (*string*) – the character encoding for the saved files
- **directed** (*bool*) – if False, save one edge for each undirected edge in the graph but retain original oneway and to/from information as edge attributes; if True, save one edge for each directed edge in the graph

Returns

Return type None

osmnx.io.**save_graph_xml**(*data*, *filepath=None*, *node_tags=['highway']*, *node_attrs=['id', 'timestamp', 'uid', 'user', 'version', 'changeset', 'lat', 'lon']*, *edge_tags=['highway', 'lanes', 'maxspeed', 'name', 'oneway']*, *edge_attrs=['id', 'timestamp', 'uid', 'user', 'version', 'changeset']*, *oneway=False*, *merge_edges=True*, *edge_tag_aggs=None*)

Do not use: deprecated. Use osm_xml.save_graph_xml instead.

Parameters

- **data** (*networkx multi(di)graph OR a length 2 iterable of nodes/edges*) – geopandas GeoDataFrames
- **filepath** (*string or pathlib.Path*) – path to the .osm file including extension. if None, use default data folder + graph.osm
- **node_tags** (*list*) – osm node tags to include in output OSM XML
- **node_attrs** (*list*) – osm node attributes to include in output OSM XML
- **edge_tags** (*list*) – osm way tags to include in output OSM XML
- **edge_attrs** (*list*) – osm way attributes to include in output OSM XML
- **oneway** (*bool*) – the default oneway value used to fill this tag where missing
- **merge_edges** (*bool*) – if True merges graph edges such that each OSM way has one entry and one entry only in the OSM XML. Otherwise, every OSM way will have a separate entry for each node pair it contains.
- **edge_tag_aggs** (*list of length-2 string tuples*) – useful only if merge_edges is True, this argument allows the user to specify edge attributes to aggregate such that the merged OSM way entry tags accurately represent the sum total of their component edge attributes. For example, if the user wants the OSM way to have a "length" attribute, the user must specify *edge_tag_aggs=[('length', 'sum')]* in order to tell this method to aggregate the lengths of the individual component edges. Otherwise, the length attribute will simply reflect the length of the first edge associated with the way.

Returns

Return type None

osmnx.io.**save_graphml**(*G*, *filepath=None*, *gephi=False*, *encoding='utf-8'*)

Save graph to disk as GraphML file.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **filepath** (*string or pathlib.Path*) – path to the GraphML file including extension. if None, use default data folder + graph.graphml

- **gephi** (*bool*) – if True, give each edge a unique key/id to work around Gephi's interpretation of the GraphML specification
- **encoding** (*string*) – the character encoding for the saved file

Returns

Return type None

3.2.10 osmnx.osm_xml module

Read/write .osm formatted XML files.

class osmnx.osm_xml.**_OSMContentHandler**

SAX content handler for OSM XML.

Used to build an Overpass-like response JSON object in self.object. For format notes, see http://wiki.openstreetmap.org/wiki/OSM_XML#OSM_XML_file_format_notes and http://overpass-api.de/output_formats.html#json

endElement(*name*)

Signals the end of an element in non-namespace mode.

The name parameter contains the name of the element type, just as with the startElement event.

startElement(*name*, *attrs*)

Signals the start of an element in non-namespace mode.

The name parameter contains the raw XML 1.0 name of the element type as a string and the attrs parameter holds an instance of the Attributes class containing the attributes of the element.

osmnx.osm_xml.**_append_edges_xml_tree**(*root*, *gdf_edges*, *edge_attrs*, *edge_tags*, *edge_tag_aggs*, *merge_edges*)

Append edges to an XML tree.

Parameters

- **root** (*ElementTree.Element*) – xml tree
- **gdf_edges** (*geopandas.GeoDataFrame*) – GeoDataFrame of graph edges
- **edge_attrs** (*list*) – osm way attributes to include in output OSM XML
- **edge_tags** (*list*) – osm way tags to include in output OSM XML
- **edge_tag_aggs** (*list of length-2 string tuples*) – useful only if merge_edges is True, this argument allows the user to specify edge attributes to aggregate such that the merged OSM way entry tags accurately represent the sum total of their component edge attributes. For example, if the user wants the OSM way to have a "length" attribute, the user must specify *edge_tag_aggs=[('length', 'sum')]* in order to tell this method to aggregate the lengths of the individual component edges. Otherwise, the length attribute will simply reflect the length of the first edge associated with the way.
- **merge_edges** (*bool*) – if True merges graph edges such that each OSM way has one entry and one entry only in the OSM XML. Otherwise, every OSM way will have a separate entry for each node pair it contains.

Returns **root** – xml tree with edges appended

Return type ElementTree.Element

osmnx.osm_xml.**_append_nodes_xml_tree**(*root*, *gdf_nodes*, *node_attrs*, *node_tags*)

Append nodes to an XML tree.

Parameters

- **root** (*ElementTree.Element*) – xml tree
- **gdf_nodes** (*geopandas.GeoDataFrame*) – GeoDataFrame of graph nodes
- **node_attrs** (*list*) – osm way attributes to include in output OSM XML
- **node_tags** (*list*) – osm way tags to include in output OSM XML

Returns **root** – xml tree with nodes appended

Return type ElementTree.Element

osmnx.osm_xml.**_get_unique_nodes_ordered_from_way**(*df_way_edges*)
Recover original node order from df of edges associated w/ single OSM way.

Parameters **df_way_edges** (*pandas.DataFrame*) – Dataframe containing columns 'u' and 'v' corresponding to origin/destination nodes.

Returns **unique_ordered_nodes** – An ordered list of unique node IDs. Note: If the edges do not all connect (e.g. [(1, 2), (2,3), (10, 11), (11, 12), (12, 13)]), then this method will return only those nodes associated with the largest component of connected edges, even if subsequent connected chunks are contain more total nodes. This is done to ensure a proper topological representation of nodes in the XML way records because if there are unconnected components, the sorting algorithm cannot recover their original order. We would not likely ever encounter this kind of disconnected structure of nodes within a given way, but it is not explicitly forbidden in the OSM XML design schema.

Return type list

osmnx.osm_xml.**_overpass_json_from_file**(*filepath*)
Read OSM XML from file and return Overpass-like JSON.

Parameters **filepath** (*string or pathlib.Path*) – path to file containing OSM XML data

Returns

Return type OSMContentHandler object

osmnx.osm_xml.**save_graph_xml**(*data*, *filepath=None*, *node_tags=['highway']*, *node_attrs=['id', 'timestamp', 'uid', 'user', 'version', 'changeset', 'lat', 'lon']*, *edge_tags=['highway', 'lanes', 'maxspeed', 'name', 'oneway']*, *edge_attrs=['id', 'timestamp', 'uid', 'user', 'version', 'changeset']*, *oneway=False*, *merge_edges=True*, *edge_tag_aggs=None*)
Save graph to disk as an OSM-formatted XML .osm file.

This function exists only to allow serialization to the .osm file format for applications that require it, and has constraints to conform to that. To save/load full-featured OSMnx graphs to/from disk for later use, use the save_graphml and load_graphml functions instead.

Note: for large networks this function can take a long time to run. Before using this function, make sure you configured OSMnx as described in the example below when you created the graph.

Example

```
>>> import osmnx as ox
>>> utn = ox.settings.useful_tags_node
>>> oxna = ox.settings.osm_xml_node_attrs
>>> oxnt = ox.settings.osm_xml_node_tags
>>> utw = ox.settings.useful_tags_way
>>> oxwa = ox.settings.osm_xml_way_attrs
>>> oxwt = ox.settings.osm_xml_way_tags
>>> utn = list(set(utn + oxna + oxnt))
>>> utw = list(set(utw + oxwa + oxwt))
>>> ox.config(all_oneway=True, useful_tags_node=utn, useful_tags_way=utw)
>>> G = ox.graph_from_place('Piedmont, CA, USA', network_type='drive')
>>> ox.save_graph_xml(G, filepath='./data/graph1.osm')
```

Parameters

- **data** (*networkx multi(di)graph OR a length 2 iterable of nodes/edges*) – geopandas GeoDataFrames
- **filepath** (*string or pathlib.Path*) – path to the .osm file including extension. if None, use default data folder + graph.osm
- **node_tags** (*list*) – osm node tags to include in output OSM XML
- **node_attrs** (*list*) – osm node attributes to include in output OSM XML
- **edge_tags** (*list*) – osm way tags to include in output OSM XML
- **edge_attrs** (*list*) – osm way attributes to include in output OSM XML
- **oneway** (*bool*) – the default oneway value used to fill this tag where missing
- **merge_edges** (*bool*) – if True merges graph edges such that each OSM way has one entry and one entry only in the OSM XML. Otherwise, every OSM way will have a separate entry for each node pair it contains.
- **edge_tag_aggs** (*list of length-2 string tuples*) – useful only if merge_edges is True, this argument allows the user to specify edge attributes to aggregate such that the merged OSM way entry tags accurately represent the sum total of their component edge attributes. For example, if the user wants the OSM way to have a "length" attribute, the user must specify *edge_tag_aggs=[('length', 'sum')]* in order to tell this method to aggregate the lengths of the individual component edges. Otherwise, the length attribute will simply reflect the length of the first edge associated with the way.

Returns

Return type None

3.2.11 osmnx.plot module

Plot spatial geometries, street networks, and routes.

osmnx.plot.**_config_ax**(*ax*, *crs*, *bbox*, *padding*)
Configure axis for display.

Parameters

- **ax** (*matplotlib axis*) – the axis containing the plot
- **crs** (*dict or string or pyproj.CRS*) – the CRS of the plotted geometries
- **bbox** (*tuple*) – bounding box as (north, south, east, west)
- **padding** (*float*) – relative padding to add around the plot's bbox

Returns **ax** – the configured/styled axis

Return type matplotlib axis

osmnx.plot.**_get_colors_by_value**(*vals*, *num_bins*, *cmap*, *start*, *stop*, *na_color*, *equal_size*)
Map colors to the values in a series.

Parameters

- **vals** (*pandas.Series*) – series labels are node/edge IDs and values are attribute values
- **num_bins** (*int*) – if None, linearly map a color to each value. otherwise, assign values to this many bins then assign a color to each bin.
- **cmap** (*string*) – name of a matplotlib colormap
- **start** (*float*) – where to start in the colorspace
- **stop** (*float*) – where to end in the colorspace
- **na_color** (*string*) – what color to assign to missing values
- **equal_size** (*bool*) – ignored if num_bins is None. if True, bin into equal-sized quantiles (requires unique bin edges). if False, bin into equal-spaced bins.

Returns **color_series** – series labels are node/edge IDs and values are colors

Return type pandas.Series

osmnx.plot.**_save_and_show**(*fig*, *ax*, *save=False*, *show=True*, *close=True*, *filepath=None*, *dpi=300*)
Save a figure to disk and/or show it, as specified by args.

Parameters

- **fig** (*figure*) – matplotlib figure
- **ax** (*axis*) – matplotlib axis
- **save** (*bool*) – if True, save the figure to disk at filepath
- **show** (*bool*) – if True, call pyplot.show() to show the figure
- **close** (*bool*) – if True, call pyplot.close() to close the figure
- **filepath** (*string*) – if save is True, the path to the file. file format determined from extension. if None, use settings.imgs_folder/image.png
- **dpi** (*int*) – if save is True, the resolution of saved file

Returns **fig, ax** – matplotlib figure, axis

Return type tuple

osmnx.plot.**get_colors**(*n*, *cmap='viridis'*, *start=0.0*, *stop=1.0*, *alpha=1.0*, *return_hex=False*)
Get *n* evenly-spaced colors from a matplotlib colormap.

Parameters

- **n** (*int*) – number of colors
- **cmap** (*string*) – name of a matplotlib colormap
- **start** (*float*) – where to start in the colorspace
- **stop** (*float*) – where to end in the colorspace
- **alpha** (*float*) – opacity, the alpha channel for the RGBa colors
- **return_hex** (*bool*) – if True, convert RGBa colors to HTML-like hexadecimal RGB strings. if False, return colors as (R, G, B, alpha) tuples.

Returns color_list

Return type list

osmnx.plot.**get_edge_colors_by_attr**(*G*, *attr*, *num_bins=None*, *cmap='viridis'*, *start=0*, *stop=1*, *na_color='none'*, *equal_size=False*)
Get colors based on edge attribute values.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **attr** (*string*) – name of a numerical edge attribute
- **num_bins** (*int*) – if None, linearly map a color to each value. otherwise, assign values to this many bins then assign a color to each bin.
- **cmap** (*string*) – name of a matplotlib colormap
- **start** (*float*) – where to start in the colorspace
- **stop** (*float*) – where to end in the colorspace
- **na_color** (*string*) – what color to assign edges with missing attr values
- **equal_size** (*bool*) – ignored if num_bins is None. if True, bin into equal-sized quantiles (requires unique bin edges). if False, bin into equal-spaced bins.

Returns **edge_colors** – series labels are edge IDs (u, v, key) and values are colors

Return type pandas.Series

osmnx.plot.**get_node_colors_by_attr**(*G*, *attr*, *num_bins=None*, *cmap='viridis'*, *start=0*, *stop=1*, *na_color='none'*, *equal_size=False*)
Get colors based on node attribute values.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **attr** (*string*) – name of a numerical node attribute
- **num_bins** (*int*) – if None, linearly map a color to each value. otherwise, assign values to this many bins then assign a color to each bin.
- **cmap** (*string*) – name of a matplotlib colormap
- **start** (*float*) – where to start in the colorspace
- **stop** (*float*) – where to end in the colorspace

- **na_color** (*string*) – what color to assign nodes with missing attr values
- **equal_size** (*bool*) – ignored if num_bins is None. if True, bin into equal-sized quantiles (requires unique bin edges). if False, bin into equal-spaced bins.

Returns **node_colors** – series labels are node IDs and values are colors

Return type pandas.Series

osmnx.plot.**plot_figure_ground**(*G=None*, *address=None*, *point=None*, *dist=805*, *network_type='drive_service'*, *street_widths=None*, *default_width=4*, *figsize=(8, 8)*, *edge_color='w'*, *smooth_joints=True*, ***pg_kwargs*)

Plot a figure-ground diagram of a street network.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph, must be unprojected
- **address** (*string*) – address to geocode as the center point if G is not passed in
- **point** (*tuple*) – center point if address and G are not passed in
- **dist** (*numeric*) – how many meters to extend north, south, east, west from center point
- **network_type** (*string*) – what type of street network to get
- **street_widths** (*dict*) – dict keys are street types and values are widths to plot in pixels
- **default_width** (*numeric*) – fallback width in pixels for any street type not in street_widths
- **figsize** (*numeric*) – (width, height) of figure, should be equal
- **edge_color** (*string*) – color of the edges' lines
- **smooth_joints** (*bool*) – if True, plot nodes same width as streets to smooth line joints and prevent cracks between them from showing
- **pg_kwargs** – keyword arguments to pass to plot_graph

Returns **fig, ax** – matplotlib figure, axis

Return type tuple

osmnx.plot.**plot_footprints**(*gdf*, *ax=None*, *figsize=(8, 8)*, *color='orange'*, *alpha=None*, *bgcolor='#111111'*, *bbox=None*, *save=False*, *show=True*, *close=False*, *filepath=None*, *dpi=600*)

Plot a GeoDataFrame of geospatial entities' footprints.

Parameters

- **gdf** (*geopandas.GeoDataFrame*) – GeoDataFrame of footprints (shapely Polygons and MultiPolygons)
- **ax** (*axis*) – if not None, plot on this preexisting axis
- **figsize** (*tuple*) – if ax is None, create new figure with size (width, height)
- **color** (*string*) – color of the footprints
- **alpha** (*float*) – opacity of the footprints
- **bgcolor** (*string*) – background color of the plot
- **bbox** (*tuple*) – bounding box as (north, south, east, west). if None, will calculate from the spatial extents of the geometries in gdf

- **save** (*bool*) – if True, save the figure to disk at filepath
- **show** (*bool*) – if True, call pyplot.show() to show the figure
- **close** (*bool*) – if True, call pyplot.close() to close the figure
- **filepath** (*string*) – if save is True, the path to the file. file format determined from extension. if None, use settings.imgs_folder/image.png
- **dpi** (*int*) – if save is True, the resolution of saved file

Returns **fig, ax** – matplotlib figure, axis

Return type tuple

osmnx.plot.**plot_graph**(*G*, *ax=None*, *figsize=(8, 8)*, *bgcolor='#111111'*, *node_color='w'*, *node_size=15*, *node_alpha=None*, *node_edgecolor='none'*, *node_zorder=1*, *edge_color='#999999'*, *edge_linewidth=1*, *edge_alpha=None*, *show=True*, *close=False*, *save=False*, *filepath=None*, *dpi=300*, *bbox=None*)

Plot a graph.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **ax** (*matplotlib axis*) – if not None, plot on this preexisting axis
- **figsize** (*tuple*) – if ax is None, create new figure with size (width, height)
- **bgcolor** (*string*) – background color of plot
- **node_color** (*string or list*) – color(s) of the nodes
- **node_size** (*int*) – size of the nodes: if 0, then skip plotting the nodes
- **node_alpha** (*float*) – opacity of the nodes, note: if you passed RGBA values to node_color, set node_alpha=None to use the alpha channel in node_color
- **node_edgecolor** (*string*) – color of the nodes' markers' borders
- **node_zorder** (*int*) – zorder to plot nodes: edges are always 1, so set node_zorder=0 to plot nodes below edges
- **edge_color** (*string or list*) – color(s) of the edges' lines
- **edge_linewidth** (*float*) – width of the edges' lines: if 0, then skip plotting the edges
- **edge_alpha** (*float*) – opacity of the edges, note: if you passed RGBA values to edge_color, set edge_alpha=None to use the alpha channel in edge_color
- **show** (*bool*) – if True, call pyplot.show() to show the figure
- **close** (*bool*) – if True, call pyplot.close() to close the figure
- **save** (*bool*) – if True, save the figure to disk at filepath
- **filepath** (*string*) – if save is True, the path to the file. file format determined from extension. if None, use settings.imgs_folder/image.png
- **dpi** (*int*) – if save is True, the resolution of saved file
- **bbox** (*tuple*) – bounding box as (north, south, east, west). if None, will calculate from spatial extents of plotted geometries.

Returns **fig, ax** – matplotlib figure, axis

Return type tuple

osmnx.plot.**plot_graph_route**(*G*, *route*, *route_color='r'*, *route_linewidth=4*, *route_alpha=0.5*, *orig_dest_size=100*, *ax=None*, ***pg_kwargs*)

Plot a route along a graph.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **route** (*list*) – route as a list of node IDs
- **route_color** (*string*) – color of the route
- **route_linewidth** (*int*) – width of the route line
- **route_alpha** (*float*) – opacity of the route line
- **orig_dest_size** (*int*) – size of the origin and destination nodes
- **ax** (*matplotlib axis*) – if not None, plot route on this preexisting axis instead of creating a new fig, ax and drawing the underlying graph
- **pg_kwargs** – keyword arguments to pass to plot_graph

Returns **fig, ax** – matplotlib figure, axis

Return type tuple

osmnx.plot.**plot_graph_routes**(*G*, *routes*, *route_colors='r'*, ***pgr_kwargs*)

Plot several routes along a graph.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **routes** (*list*) – routes as a list of lists of node IDs
- **route_colors** (*string or list*) – if string, 1 color for all routes. if list, the colors for each route.
- **pgr_kwargs** – keyword arguments to pass to plot_graph_route

Returns **fig, ax** – matplotlib figure, axis

Return type tuple

3.2.12 osmnx.projection module

Project spatial geometries and spatial networks.

osmnx.projection.**project_gdf**(*gdf*, *to_crs=None*, *to_latlong=False*)

Project a GeoDataFrame from its current CRS to another.

If to_crs is None, project to the UTM CRS for the UTM zone in which the GeoDataFrame's centroid lies. Otherwise project to the CRS defined by to_crs. The simple UTM zone calculation in this function works well for most latitudes, but may not work for some extreme northern locations like Svalbard or far northern Norway.

Parameters

- **gdf** (*geopandas.GeoDataFrame*) – the GeoDataFrame to be projected
- **to_crs** (*string or pyproj.CRS*) – if None, project to UTM zone in which gdf's centroid lies, otherwise project to this CRS
- **to_latlong** (*bool*) – if True, project to settings.default_crs and ignore to_crs

Returns **gdf_proj** – the projected GeoDataFrame

Return type geopandas.GeoDataFrame

osmnx.projection.**project_geometry**(*geometry*, *crs=None*, *to_crs=None*, *to_latlong=False*)

Project a shapely geometry from its current CRS to another.

If to_crs is None, project to the UTM CRS for the UTM zone in which the geometry's centroid lies. Otherwise project to the CRS defined by to_crs.

Parameters

- **geometry** (*shapely.geometry.Polygon or shapely.geometry.MultiPolygon*) – the geometry to project
- **crs** (*string or pyproj.CRS*) – the starting CRS of the passed-in geometry. if None, it will be set to settings.default_crs
- **to_crs** (*string or pyproj.CRS*) – if None, project to UTM zone in which geometry's centroid lies, otherwise project to this CRS
- **to_latlong** (*bool*) – if True, project to settings.default_crs and ignore to_crs

Returns **geometry_proj, crs** – the projected geometry and its new CRS

Return type tuple

osmnx.projection.**project_graph**(*G*, *to_crs=None*)

Project graph from its current CRS to another.

If to_crs is None, project the graph to the UTM CRS for the UTM zone in which the graph's centroid lies. Otherwise, project the graph to the CRS defined by to_crs.

Parameters

- **G** (*networkx.MultiDiGraph*) – the graph to be projected
- **to_crs** (*string or pyproj.CRS*) – if None, project graph to UTM zone in which graph centroid lies, otherwise project graph to this CRS

Returns **G_proj** – the projected graph

Return type networkx.MultiDiGraph

3.2.13 osmnx.settings module

Global settings that can be configured by user with utils.config().

3.2.14 osmnx.simplification module

Simplify, correct, and consolidate network topology.

osmnx.simplification.**_build_path**(*G*, *endpoint*, *endpoint_successor*, *endpoints*)

Build a path of nodes from one endpoint node to next endpoint node.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **endpoint** (*int*) – the endpoint node from which to start the path
- **endpoint_successor** (*int*) – the successor of endpoint through which the path to the next endpoint will be built
- **endpoints** (*set*) – the set of all nodes in the graph that are endpoints

Returns **path** – the first and last items in the resulting path list are endpoint nodes, and all other items are interstitial nodes that can be removed subsequently

Return type list

osmnx.simplification.**_consolidate_intersections_rebuild_graph**(*G*, *tolerance=10*, *reconnect_edges=True*)

Consolidate intersections comprising clusters of nearby nodes.

Merge nodes and return a rebuilt graph with consolidated intersections and reconnected edge geometries.

The tolerance argument should be adjusted to approximately match street design standards in the specific street network, and you should always use a projected graph to work in meaningful and consistent units like meters.

Returned graph's node IDs represent clusters rather than osmids. Refer to nodes' osmid_original attributes for original osmids. If multiple nodes were merged together, the osmid_original attribute is a list of merged nodes' osmids.

Parameters

- **G** (*networkx.MultiDiGraph*) – a projected graph
- **tolerance** (*float*) – nodes are buffered to this distance (in graph's geometry's units) and subsequent overlaps are dissolved into a single node
- **reconnect_edges** (*bool*) – ignored if rebuild_graph is not True. if True, reconnect edges and their geometries in rebuilt graph to the consolidated nodes and update edge length attributes; if False, returned graph has no edges (which is faster if you just need topologically consolidated intersection counts).

Returns **H** – a rebuilt graph with consolidated intersections and reconnected edge geometries

Return type networkx.MultiDiGraph

osmnx.simplification.**_get_paths_to_simplify**(*G*, *strict=True*)

Generate all the paths to be simplified between endpoint nodes.

The path is ordered from the first endpoint, through the interstitial nodes, to the second endpoint.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **strict** (*bool*) – if False, allow nodes to be end points even if they fail all other rules but have edges with different OSM IDs

Yields **path_to_simplify** (*list*)

osmnx.simplification.**_is_endpoint**(*G*, *node*, *strict=True*)

Is node a true endpoint of an edge.

Return True if the node is a "real" endpoint of an edge in the network, otherwise False. OSM data includes lots of nodes that exist only as points to help streets bend around curves. An end point is a node that either: 1) is its own neighbor, ie, it self-loops. 2) or, has no incoming edges or no outgoing edges, ie, all its incident edges point inward or all its incident edges point outward. 3) or, it does not have exactly two neighbors and degree of 2 or 4. 4) or, if strict mode is false, if its edges have different OSM IDs.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **node** (*int*) – the node to examine
- **strict** (*bool*) – if False, allow nodes to be end points even if they fail all other rules but have edges with different OSM IDs

Returns

Return type bool

osmnx.simplification.**_merge_nodes_geometric**(*G*, *tolerance*, *chunk=True*)

Geometrically merge nodes within some distance of each other.

If chunk=True, it sorts the nodes GeoSeries by geometry x and y values (to make unary_union faster), then buffers by tolerance. Next it divides the nodes GeoSeries into n-sized chunks, where n = the square root of the number of nodes. Then it runs unary_union on each chunk, and then runs unary_union on the resulting unary unions. This is much faster on large graphs (n>100000) because of how unary_union's runtime scales with vertex count. But chunk=False is usually faster on small and medium sized graphs. This hacky method will hopefully be made obsolete when shapely becomes vectorized by incorporating the pygeos codebase.

Parameters

- **G** (*networkx.MultiDiGraph*) – a projected graph
- **tolerance** (*float*) – nodes are buffered to this distance (in graph's geometry's units) and subsequent overlaps are dissolved into a single polygon
- **chunk** (*bool*) – if True, divide nodes into geometrically sorted chunks to improve the speed of unary_union operation by running it on each chunk and then running it on the results of those runs

Returns **merged_nodes** – the merged overlapping polygons of the buffered nodes

Return type GeoSeries

osmnx.simplification.**consolidate_intersections**(*G*, *tolerance=10*, *rebuild_graph=True*, *dead_ends=False*, *reconnect_edges=True*)

Consolidate intersections comprising clusters of nearby nodes.

Merges nearby nodes and returns either their centroids or a rebuilt graph with consolidated intersections and reconnected edge geometries. The tolerance argument should be adjusted to approximately match street design standards in the specific street network, and you should always use a projected graph to work in meaningful and consistent units like meters.

When rebuild_graph=False, it uses a purely geometrical (and relatively fast) algorithm to identify "geometrically close" nodes, merge them, and return just the merged intersections' centroids. When rebuild_graph=True, it uses a topological (and slower but more accurate) algorithm to identify "topologically close" nodes, merge them, then rebuild/return the graph. Returned graph's node IDs represent clusters rather than osmids. Refer to nodes' osmid_original attributes for original osmids. If multiple nodes were merged together, the osmid_original attribute is a list of merged nodes' osmids.

Divided roads are often represented by separate centerline edges. The intersection of two divided roads thus creates 4 nodes, representing where each edge intersects a perpendicular edge. These 4 nodes represent a single intersection in the real world. A similar situation occurs with roundabouts and traffic circles. This function consolidates nearby nodes by buffering them to an arbitrary distance, merging overlapping buffers, and taking their centroid.

Parameters

- **G** (*networkx.MultiDiGraph*) – a projected graph
- **tolerance** (*float*) – nodes are buffered to this distance (in graph's geometry's units) and subsequent overlaps are dissolved into a single node
- **rebuild_graph** (*bool*) – if True, consolidate the nodes topologically, rebuild the graph, and return as networkx.MultiDiGraph. if False, consolidate the nodes geometrically and return the consolidated node points as geopandas.GeoSeries

- **dead_ends** (*bool*) – if False, discard dead-end nodes to return only street-intersection points
- **reconnect_edges** (*bool*) – ignored if rebuild_graph is not True. if True, reconnect edges and their geometries in rebuilt graph to the consolidated nodes and update edge length attributes; if False, returned graph has no edges (which is faster if you just need topologically consolidated intersection counts).

Returns if rebuild_graph=True, returns MultiDiGraph with consolidated intersections and reconnected edge geometries. if rebuild_graph=False, returns GeoSeries of shapely Points representing the centroids of street intersections

Return type networkx.MultiDiGraph or geopandas.GeoSeries

osmnx.simplification.**simplify_graph**(*G*, *strict=True*, *remove_rings=True*)

Simplify a graph's topology by removing interstitial nodes.

Simplifies graph topology by removing all nodes that are not intersections or dead-ends. Create an edge directly between the end points that encapsulate them, but retain the geometry of the original edges, saved as a new *geometry* attribute on the new edge. Note that only simplified edges receive a *geometry* attribute. Some of the resulting consolidated edges may comprise multiple OSM ways, and if so, their multiple attribute values are stored as a list.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **strict** (*bool*) – if False, allow nodes to be end points even if they fail all other rules but have incident edges with different OSM IDs. Lets you keep nodes at elbow two-way intersections, but sometimes individual blocks have multiple OSM IDs within them too.
- **remove_rings** (*bool*) – if True, remove isolated self-contained rings that have no endpoints

Returns **G** – topologically simplified graph, with a new *geometry* attribute on each simplified edge

Return type networkx.MultiDiGraph

3.2.15 osmnx.speed module

Calculate graph edge speeds and travel times.

osmnx.speed.**_clean_maxspeed**(*value*, *convert_mph=True*)

Clean a maxspeed string and convert mph to kph if necessary.

Parameters

- **value** (*string*) – an OSM way maxspeed value
- **convert_mph** (*bool*) – if True, convert mph to kph

Returns **value_clean**

Return type string

osmnx.speed.**_collapse_multiple_maxspeed_values**(*value*)

Collapse a list of maxspeed values into its mean value.

Parameters **value** (*list or string*) – an OSM way maxspeed value, or a list of them

Returns **mean_value** – an integer representation of the mean value in the list, converted to kph if original value was in mph.

Return type int

osmnx.speed.**add_edge_speeds**(*G*, *hwy_speeds=None*, *fallback=None*, *precision=1*)
Add edge speeds (km per hour) to graph as new *speed_kph* edge attributes.

Imputes free-flow travel speeds for all edges based on mean *maxspeed* value of edges, per highway type. For highway types in graph that have no *maxspeed* value on any edge, function assigns the mean of all *maxspeed* values in graph.

This mean-imputation can obviously be imprecise, and the caller can override it by passing in *hwy_speeds* and/or *fallback* arguments that correspond to local speed limit standards.

If edge *maxspeed* attribute has "mph" in it, value will automatically be converted from miles per hour to km per hour. Any other speed units should be manually converted to km per hour prior to running this function, otherwise there could be unexpected results. If "mph" does not appear in the edge's maxspeed attribute string, then function assumes kph, per OSM guidelines: https://wiki.openstreetmap.org/wiki/Map_Features/Units

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **hwy_speeds** (*dict*) – dict keys = OSM highway types and values = typical speeds (km per hour) to assign to edges of that highway type for any edges missing speed data. Any edges with highway type not in *hwy_speeds* will be assigned the mean preexisting speed value of all edges of that highway type.
- **fallback** (*numeric*) – default speed value (km per hour) to assign to edges whose highway type did not appear in *hwy_speeds* and had no preexisting speed values on any edge
- **precision** (*int*) – decimal precision to round speed_kph

Returns **G** – graph with speed_kph attributes on all edges

Return type networkx.MultiDiGraph

osmnx.speed.**add_edge_travel_times**(*G*, *precision=1*)
Add edge travel time (seconds) to graph as new *travel_time* edge attributes.

Calculates free-flow travel time along each edge, based on *length* and *speed_kph* attributes. Note: run *add_edge_speeds* first to generate the *speed_kph* attribute. All edges must have *length* and *speed_kph* attributes and all their values must be non-null.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **precision** (*int*) – decimal precision to round travel_time

Returns **G** – graph with travel_time attributes on all edges

Return type networkx.MultiDiGraph

3.2.16 osmnx.stats module

Calculate geometric and topological network measures.

osmnx.stats.**basic_stats**(*G*, *area=None*, *clean_intersects=False*, *tolerance=15*, *circuity_dist='gc'*)
Calculate basic descriptive geometric and topological stats for a graph.

For an unprojected lat-lng graph, tolerance and graph units should be in degrees, and circuity_dist should be 'gc'. For a projected graph, tolerance and graph units should be in meters (or similar) and circuity_dist should be 'euclidean'.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **area** (*numeric*) – the land area of this study site, in square meters. must be greater than 0. if None, will skip all density-based metrics.
- **clean_intersects** (*bool*) – if True, calculate consolidated intersections count (and density, if area is provided) via consolidate_intersections function
- **tolerance** (*numeric*) – tolerance value passed along if clean_intersects=True, see consolidate_intersections function documentation for details and usage
- **circuity_dist** (*string*) – 'gc' or 'euclidean', how to calculate straight-line distances for circuity measurement; use former for lat-lng networks and latter for projected networks

Returns

stats – network measures containing the following elements (some keys may not be present, based on the arguments passed into the function):

- n = number of nodes in the graph
- m = number of edges in the graph
- k_avg = average node degree of the graph
- **intersection_count = number of intersections in graph, that is,** nodes with >1 physical street connected to them
- **streets_per_node_avg = how many physical streets (edges in the** undirected representation of the graph) connect to each node (ie, intersection or dead-end) on average (mean)
- **streets_per_node_counts = dict with keys of number of physical** streets connecting to a node, and values of number of nodes with this count
- **streets_per_node_proportion = dict, same as previous, but as a** proportion of the total, rather than counts
- edge_length_total = sum of all edge lengths in graph, in meters
- edge_length_avg = mean edge length in the graph, in meters
- **street_length_total = sum of all edges in the undirected** representation of the graph
- **street_length_avg = mean edge length in the undirected** representation of the graph, in meters
- **street_segments_count = number of edges in the undirected** representation of the graph
- node_density_km = n divided by area in square kilometers
- **intersection_density_km = intersection_count divided by area in** square kilometers
- **edge_density_km = edge_length_total divided by area in square** kilometers
- **street_density_km = street_length_total divided by area in square** kilometers
- **circuity_avg = edge_length_total divided by the sum of the great** circle distances between the nodes of each edge
- **self_loop_proportion = proportion of edges that have a single node** as its endpoints (ie, the edge links nodes u and v, and u==v)
- **clean_intersection_count = number of intersections in street** network, merging complex ones into single points
- **clean_intersection_density_km = clean_intersection_count divided** by area in square kilometers

Return type dict

osmnx.stats.**extended_stats**(*G*, *connectivity=False*, *anc=False*, *ecc=False*, *bc=False*, *cc=False*)

Calculate extended topological measures for a graph.

Many of these algorithms have an inherently high time complexity. Global topological analysis of large complex networks is extremely time consuming and may exhaust computer memory. Consider using function arguments to not run metrics that require computation of a full matrix of paths if they will not be needed.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **connectivity** (*bool*) – if True, calculate node and edge connectivity
- **anc** (*bool*) – if True, calculate average node connectivity
- **ecc** (*bool*) – if True, calculate shortest paths, eccentricity, and topological metrics that use eccentricity
- **bc** (*bool*) – if True, calculate node betweenness centrality
- **cc** (*bool*) – if True, calculate node closeness centrality

Returns

stats – dictionary of network measures containing the following elements (some only calculated/returned optionally, based on passed parameters):

- avg_neighbor_degree
- avg_neighbor_degree_avg
- avg_weighted_neighbor_degree
- avg_weighted_neighbor_degree_avg
- degree_centrality
- degree_centrality_avg
- clustering_coefficient
- clustering_coefficient_avg
- clustering_coefficient_weighted
- clustering_coefficient_weighted_avg
- pagerank
- pagerank_max_node
- pagerank_max
- pagerank_min_node
- pagerank_min
- node_connectivity
- node_connectivity_avg
- edge_connectivity
- eccentricity
- diameter
- radius

- center
- periphery
- closeness_centrality
- closeness_centrality_avg
- betweenness_centrality
- betweenness_centrality_avg

Return type dict

3.2.17 osmnx.truncate module

Truncate graph by distance, bounding box, or polygon.

osmnx.truncate.**truncate_graph_bbox**(*G*, *north*, *south*, *east*, *west*, *truncate_by_edge=False*, *retain_all=False*, *quadrat_width=0.05*, *min_num=3*)

Remove every node in graph that falls outside a bounding box.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **north** (*float*) – northern latitude of bounding box
- **south** (*float*) – southern latitude of bounding box
- **east** (*float*) – eastern longitude of bounding box
- **west** (*float*) – western longitude of bounding box
- **truncate_by_edge** (*bool*) – if True, retain nodes outside bounding box if at least one of node's neighbors is within the bounding box
- **retain_all** (*bool*) – if True, return the entire graph even if it is not connected. otherwise, retain only the largest weakly connected component.
- **quadrat_width** (*numeric*) – passed on to intersect_index_quadrats: the linear length (in degrees) of the quadrats with which to cut up the geometry (default = 0.05, approx 4km at NYC's latitude)
- **min_num** (*int*) – passed on to intersect_index_quadrats: the minimum number of linear quadrat lines (e.g., min_num=3 would produce a quadrat grid of 4 squares)

Returns **G** – the truncated graph

Return type networkx.MultiDiGraph

osmnx.truncate.**truncate_graph_dist**(*G*, *source_node*, *max_dist=1000*, *weight='length'*, *retain_all=False*)

Remove every node farther than some network distance from source_node.

This function can be slow for large graphs, as it must calculate shortest path distances between source_node and every other graph node.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **source_node** (*int*) – the node in the graph from which to measure network distances to other nodes

- **max_dist** (*int*) – remove every node in the graph greater than this distance from the source_node (along the network)
- **weight** (*string*) – how to weight the graph when measuring distance (default 'length' is how many meters long the edge is)
- **retain_all** (*bool*) – if True, return the entire graph even if it is not connected. otherwise, retain only the largest weakly connected component.

Returns G – the truncated graph

Return type networkx.MultiDiGraph

osmnx.truncate.**truncate_graph_polygon**(*G*, *polygon*, *retain_all=False*, *truncate_by_edge=False*, *quadrat_width=0.05*, *min_num=3*)

Remove every node in graph that falls outside a (Multi)Polygon.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **polygon** (*shapely.geometry.Polygon or shapely.geometry.MultiPolygon*) – only retain nodes in graph that lie within this geometry
- **retain_all** (*bool*) – if True, return the entire graph even if it is not connected. otherwise, retain only the largest weakly connected component.
- **truncate_by_edge** (*bool*) – if True, retain nodes outside boundary polygon if at least one of node's neighbors is within the polygon
- **quadrat_width** (*numeric*) – passed on to intersect_index_quadrats: the linear length (in degrees) of the quadrats with which to cut up the geometry (default = 0.05, approx 4km at NYC's latitude)
- **min_num** (*int*) – passed on to intersect_index_quadrats: the minimum number of linear quadrat lines (e.g., min_num=3 would produce a quadrat grid of 4 squares)

Returns G – the truncated graph

Return type networkx.MultiDiGraph

3.2.18 osmnx.utils module

General utility functions.

osmnx.utils.**_get_logger**(*level=None*, *name=None*, *filename=None*)

Create a logger or return the current one if already instantiated.

Parameters

- **level** (*int*) – one of Python's logger.level constants
- **name** (*string*) – name of the logger
- **filename** (*string*) – name of the log file, without file extension

Returns logger

Return type logging.logger

osmnx.utils.**citation**()

Print the OSMnx package's citation information.

Boeing, G. 2017. OSMnx: New Methods for Acquiring, Constructing, Analyzing, and Visualizing Complex Street Networks. Computers, Environment and Urban Systems, 65(126-139). https://doi.org/10.1016/j.compenvurbsys.2017.05.004

Returns

Return type None

osmnx.utils.**config**(*all_oneway=False*, *bidirectional_network_types=['walk']*, *cache_folder='./cache'*, *cache_only_mode=False*, *data_folder='./data'*, *default_accept_language='en'*, *default_access='["access"!~"private"]'*, *default_crs='epsg:4326'*, *default_referer='OSMnx Python package (https://github.com/gboeing/osmnx)'*, *default_user_agent='OSMnx Python package (https://github.com/gboeing/osmnx)'*, *elevation_provider='google'*, *imgs_folder='./images'*, *log_console=False*, *log_file=False*, *log_filename='osmnx'*, *log_level=20*, *log_name='OSMnx'*, *logs_folder='./logs'*, *max_query_area_size=2500000000*, *memory=None*, *nominatim_endpoint='https://nominatim.openstreetmap.org/'*, *nominatim_key=None*, *osm_xml_node_attrs=['id', 'timestamp', 'uid', 'user', 'version', 'changeset', 'lat', 'lon']*, *osm_xml_node_tags=['highway']*, *osm_xml_way_attrs=['id', 'timestamp', 'uid', 'user', 'version', 'changeset']*, *osm_xml_way_tags=['highway', 'lanes', 'maxspeed', 'name', 'oneway']*, *overpass_endpoint='http://overpass-api.de/api'*, *overpass_settings='[out:json][timeout:{timeout}]{maxsize}'*, *timeout=180*, *use_cache=True*, *useful_tags_node=['ref', 'highway']*, *useful_tags_way=['bridge', 'tunnel', 'oneway', 'lanes', 'ref', 'name', 'highway', 'maxspeed', 'service', 'access', 'area', 'landuse', 'width', 'est_width', 'junction']*)

Configure OSMnx by setting the default global settings' values.

Any parameters not passed by the caller are (re-)set to their original default values.

Parameters

- **all_oneway** (*bool*) – Only use if specifically saving to .osm XML file with save_graph_xml function. if True, forces all ways to be loaded as oneway ways, preserving the original order of nodes stored in the OSM way XML.
- **bidirectional_network_types** (*list*) – network types for which a fully bidirectional graph will be created
- **cache_folder** (*string or pathlib.Path*) – path to folder in which to save/load HTTP response cache
- **data_folder** (*string or pathlib.Path*) – path to folder in which to save/load graph files by default
- **cache_only_mode** (*bool*) – If True, download network data from Overpass then raise a CacheOnlyModeInterrupt error for user to catch. This prevents graph building from taking place and instead just saves OSM response data to cache. Useful for sequentially caching lots of raw data (as you can only query Overpass one request at a time) then using the cache to quickly build many graphs simultaneously with multiprocessing.
- **default_accept_language** (*string*) – HTTP header accept-language
- **default_access** (*string*) – default filter for OSM "access" key
- **default_crs** (*string*) – default coordinate reference system to set when creating graphs
- **default_referer** (*string*) – HTTP header referer
- **default_user_agent** (*string*) – HTTP header user-agent

- **elevation_provider** (*string {"google", "airmap"}*) – the API provider to use for adding node elevations
- **imgs_folder** (*string or pathlib.Path*) – path to folder in which to save plot images by default
- **log_file** (*bool*) – if True, save log output to a file in logs_folder
- **log_filename** (*string*) – name of the log file, without file extension
- **log_console** (*bool*) – if True, print log output to the console (terminal window)
- **log_level** (*int*) – one of Python's logger.level constants
- **log_name** (*string*) – name of the logger
- **logs_folder** (*string or pathlib.Path*) – path to folder in which to save log files
- **max_query_area_size** (*int*) – maximum area for any part of the geometry in meters: any polygon bigger than this will get divided up for multiple queries to API (default 50km x 50km)
- **memory** (*int*) – Overpass server memory allocation size for the query, in bytes. If None, server will use its default allocation size. Use with caution.
- **nominatim_endpoint** (*string*) – the API endpoint to use for nominatim queries
- **nominatim_key** (*string*) – your API key, if you are using an endpoint that requires one
- **osm_xml_node_attrs** (*list*) – node attributes for saving .osm XML files with save_graph_xml function
- **osm_xml_node_tags** (*list*) – node tags for saving .osm XML files with save_graph_xml function
- **osm_xml_way_attrs** (*list*) – edge attributes for saving .osm XML files with save_graph_xml function
- **osm_xml_way_tags** (*list*) – edge tags for for saving .osm XML files with save_graph_xml function
- **overpass_endpoint** (*string*) – the API endpoint to use for overpass queries
- **overpass_settings** (*string*) – Settings string for overpass queries. For example, to query historical OSM data as of a certain date: *'[out:json][timeout:90][date:"2019-10-28T19:20:00Z"]'*. Use with caution.
- **timeout** (*int*) – the timeout interval for the HTTP request and for API to use while running the query
- **use_cache** (*bool*) – if True, cache HTTP responses locally instead of calling API repeatedly for the same request
- **useful_tags_node** (*list*) – OSM "node" tags to add as graph node attributes, when present
- **useful_tags_way** (*list*) – OSM "way" tags to add as graph edge attributes, when present

Returns

Return type None

osmnx.utils.**log**(*message*, *level=None*, *name=None*, *filename=None*)
Write a message to the logger.

This logs to file and/or prints to the console (terminal), depending on the current configuration of settings.log_file and settings.log_console.

Parameters

- **message** (*string*) – the message to log
- **level** (*int*) – one of Python's logger.level constants
- **name** (*string*) – name of the logger
- **filename** (*string*) – name of the log file, without file extension

Returns

Return type None

osmnx.utils.**ts**(*style='datetime'*, *template=None*)
Get current timestamp as string.

Parameters

- **style** (*string {"datetime", "date", "time"}*) – format the timestamp with this built-in template
- **template** (*string*) – if not None, format the timestamp with this template instead of one of the built-in styles

Returns **ts** – the string timestamp

Return type string

3.2.19 osmnx.utils_geo module

Geospatial utility functions.

osmnx.utils_geo.**_consolidate_subdivide_geometry**(*geometry*, *max_query_area_size=None*)
Consolidate and subdivide some geometry.

Consolidate a geometry into a convex hull, then subdivide it into smaller sub-polygons if its area exceeds max size (in geometry's units). Configure the max size via max_query_area_size in the settings module.

Parameters

- **geometry** (*shapely.geometry.Polygon or shapely.geometry.MultiPolygon*) – the geometry to consolidate and subdivide
- **max_query_area_size** (*int*) – maximum area for any part of the geometry in meters: any polygon bigger than this will get divided up for multiple queries to API (default 50km x 50km). if None, use settings.max_query_area_size

Returns **geometry**

Return type shapely.geometry.Polygon or shapely.geometry.MultiPolygon

osmnx.utils_geo.**_get_polygons_coordinates**(*geometry*)
Extract exterior coordinates from polygon(s) to pass to OSM.

Ignore the interior ("holes") coordinates.

Parameters **geometry** (*shapely.geometry.Polygon or shapely.geometry.MultiPolygon*) – the geometry to extract exterior coordinates from

Returns **polygon_coord_strs**

Return type list

osmnx.utils_geo.**_intersect_index_quadrats**(*geometries*, *polygon*, *quadrat_width=0.05*, *min_num=3*)

Identify geometries that intersect a (multi)polygon.

Uses an r-tree spatial index and cuts polygon up into smaller sub-polygons for r-tree acceleration. Ensure that geometries and polygon are in the same coordinate reference system.

Parameters

- **geometries** (*geopandas.GeoSeries*) – the geometries to intersect with the polygon
- **polygon** (*shapely.geometry.Polygon or shapely.geometry.MultiPolygon*) – the polygon to intersect with the geometries
- **quadrat_width** (*numeric*) – linear length (in polygon's units) of quadrat lines with which to cut up the polygon (default = 0.05 degrees, approx 4km at NYC's latitude)
- **min_num** (*int*) – the minimum number of linear quadrat lines (e.g., min_num=3 would produce a quadrat grid of 4 squares)

Returns **geoms_in_poly** – index labels of geometries that intersected polygon

Return type set

osmnx.utils_geo.**_quadrat_cut_geometry**(*geometry*, *quadrat_width*, *min_num=3*)

Split a Polygon or MultiPolygon up into sub-polygons of a specified size.

Parameters

- **geometry** (*shapely.geometry.Polygon or shapely.geometry.MultiPolygon*) – the geometry to split up into smaller sub-polygons
- **quadrat_width** (*numeric*) – the linear width of the quadrats with which to cut up the geometry (in the units the geometry is in)
- **min_num** (*int*) – the minimum number of linear quadrat lines (e.g., min_num=3 would produce a quadrat grid of 4 squares)

Returns **geometry**

Return type shapely.geometry.MultiPolygon

osmnx.utils_geo.**_round_linestring_coords**(*ls*, *precision*)

Round the coordinates of a shapely LineString to some decimal precision.

Parameters

- **ls** (*shapely.geometry.LineString*) – the LineString to round the coordinates of
- **precision** (*int*) – decimal precision to round coordinates to

Returns

Return type shapely.geometry.LineString

osmnx.utils_geo.**_round_multilinestring_coords**(*mls*, *precision*)

Round the coordinates of a shapely MultiLineString to some decimal precision.

Parameters

- **mls** (*shapely.geometry.MultiLineString*) – the MultiLineString to round the coordinates of
- **precision** (*int*) – decimal precision to round coordinates to

Returns

Return type shapely.geometry.MultiLineString

osmnx.utils_geo.**_round_multipoint_coords**(*mpt*, *precision*)
Round the coordinates of a shapely MultiPoint to some decimal precision.

Parameters

- **mpt** (*shapely.geometry.MultiPoint*) – the MultiPoint to round the coordinates of
- **precision** (*int*) – decimal precision to round coordinates to

Returns

Return type shapely.geometry.MultiPoint

osmnx.utils_geo.**_round_multipolygon_coords**(*mp*, *precision*)
Round the coordinates of a shapely MultiPolygon to some decimal precision.

Parameters

- **mp** (*shapely.geometry.MultiPolygon*) – the MultiPolygon to round the coordinates of
- **precision** (*int*) – decimal precision to round coordinates to

Returns

Return type shapely.geometry.MultiPolygon

osmnx.utils_geo.**_round_point_coords**(*pt*, *precision*)
Round the coordinates of a shapely Point to some decimal precision.

Parameters

- **pt** (*shapely.geometry.Point*) – the Point to round the coordinates of
- **precision** (*int*) – decimal precision to round coordinates to

Returns

Return type shapely.geometry.Point

osmnx.utils_geo.**_round_polygon_coords**(*p*, *precision*)
Round the coordinates of a shapely Polygon to some decimal precision.

Parameters

- **p** (*shapely.geometry.Polygon*) – the polygon to round the coordinates of
- **precision** (*int*) – decimal precision to round coordinates to

Returns **new_poly** – the polygon with rounded coordinates

Return type shapely.geometry.Polygon

osmnx.utils_geo.**bbox_from_point**(*point*, *dist=1000*, *project_utm=False*, *return_crs=False*)
Create a bounding box from a (lat, lng) center point.

Create a bounding box some distance in each direction (north, south, east, and west) from the center point and optionally project it.

Parameters

- **point** (*tuple*) – the (lat, lng) center point to create the bounding box around
- **dist** (*int*) – bounding box distance in meters from the center point
- **project_utm** (*bool*) – if True, return bounding box as UTM-projected coordinates
- **return_crs** (*bool*) – if True, and project_utm=True, return the projected CRS too

Returns (north, south, east, west) or (north, south, east, west, crs_proj)

Return type tuple

osmnx.utils_geo.**bbox_to_poly**(*north*, *south*, *east*, *west*)

Convert bounding box coordinates to shapely Polygon.

Parameters

- **north** (*float*) – northern coordinate
- **south** (*float*) – southern coordinate
- **east** (*float*) – eastern coordinate
- **west** (*float*) – western coordinate

Returns

Return type shapely.geometry.Polygon

osmnx.utils_geo.**redistribute_vertices**(*geom*, *dist*)

Redistribute the vertices on a projected LineString or MultiLineString.

The distance argument is only approximate since the total distance of the linestring may not be a multiple of the preferred distance. This function works on only (Multi)LineString geometry types.

Parameters

- **geom** (*shapely.geometry.LineString or shapely.geometry.MultiLineString*) – a Shapely geometry (should be projected)
- **dist** (*float*) – spacing length along edges. Units are same as the geom: degrees for unprojected geometries and meters for projected geometries. The smaller the dist value, the more points are created.

Returns the redistributed vertices as a list if geom is a LineString or MultiLineString if geom is a MultiLineString

Return type list or shapely.geometry.MultiLineString

osmnx.utils_geo.**round_geometry_coords**(*geom*, *precision*)

Round the coordinates of a shapely geometry to some decimal precision.

Parameters

- **geom** (*shapely.geometry.geometry {Point, MultiPoint, LineString, MultiLineString, Polygon, MultiPolygon}*) – the geometry to round the coordinates of
- **precision** (*int*) – decimal precision to round coordinates to

Returns

Return type shapely.geometry.geometry

3.2.20 osmnx.utils_graph module

Graph utility functions.

osmnx.utils_graph.**_is_duplicate_edge**(*data1*, *data2*)
Check if two graph edge data dicts have the same osmid and geometry.

Parameters

- **data1** (*dict*) – the first edge's data
- **data2** (*dict*) – the second edge's data

Returns **is_dupe**

Return type bool

osmnx.utils_graph.**_is_same_geometry**(*ls1*, *ls2*)
Determine if two LineString geometries are the same (in either direction).

Check both the normal and reversed orders of their constituent points.

Parameters

- **ls1** (*shapely.geometry.LineString*) – the first LineString geometry
- **ls2** (*shapely.geometry.LineString*) – the second LineString geometry

Returns

Return type bool

osmnx.utils_graph.**_update_edge_keys**(*G*)
Increment key of one edge of parallel edges that differ in geometry.

For example, two streets from u to v that bow away from each other as separate streets, rather than opposite direction edges of a single street. Increment one of these edge's keys so that they do not match across u, v, k or v, u, k so we can add both to an undirected MultiGraph.

Parameters **G** (*networkx.MultiDiGraph*) – input graph

Returns **G**

Return type networkx.MultiDiGraph

osmnx.utils_graph.**add_edge_lengths**(*G*, *precision=3*)
Add *length* attribute (in meters) to each edge.

Calculated via great-circle distance between each edge's incident nodes, so ensure graph is in unprojected coordinates. Graph should be unsimplified to get accurate distances. Note: this function is run by all the *graph.graph_from_x* functions automatically to add *length* attributes to all edges.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **precision** (*int*) – decimal precision to round lengths

Returns **G** – graph with edge length attributes

Return type networkx.MultiDiGraph

osmnx.utils_graph.**count_streets_per_node**(*G*, *nodes=None*)
Count how many physical street segments connect to each node in a graph.

This function uses an undirected representation of the graph and special handling of self-loops to accurately count physical streets rather than directed edges. Note: this function is automatically run by all

the *graph.graph_from_x* functions prior to truncating the graph to the requested boundaries, to add accurate *street_count* attributes to each node even if some of its neighbors are outside the requested graph boundaries.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **nodes** (*list*) – which node IDs to get counts for. if None, use all graph nodes, otherwise calculate counts only for these node IDs

Returns **streets_per_node** – counts of how many physical streets connect to each node, with keys = node ids and values = counts

Return type dict

osmnx.utils_graph.**get_digraph**(*G*, *weight='length'*)

Convert MultiDiGraph to DiGraph.

Chooses between parallel edges by minimizing *weight* attribute value. Note: see also *get_undirected* to convert MultiDiGraph to MultiGraph.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **weight** (*string*) – attribute value to minimize when choosing between parallel edges

Returns

Return type networkx.DiGraph

osmnx.utils_graph.**get_largest_component**(*G*, *strongly=False*)

Get subgraph of G's largest weakly/strongly connected component.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **strongly** (*bool*) – if True, return the largest strongly instead of weakly connected component

Returns **G** – the largest connected component subgraph of the original graph

Return type networkx.MultiDiGraph

osmnx.utils_graph.**get_route_edge_attributes**(*G*, *route*, *attribute=None*, *minimize_key='length'*, *retrieve_default=None*)

Get a list of attribute values for each edge in a path.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **route** (*list*) – list of nodes IDs constituting the path
- **attribute** (*string*) – the name of the attribute to get the value of for each edge. If None, the complete data dict is returned for each edge.
- **minimize_key** (*string*) – if there are parallel edges between two nodes, select the one with the lowest value of minimize_key
- **retrieve_default** (*Callable[Tuple[Any, Any], Any]*) – function called with the edge nodes as parameters to retrieve a default value, if the edge does not contain the given attribute (otherwise a *KeyError* is raised)

Returns **attribute_values** – list of edge attribute values

Return type list

osmnx.utils_graph.**get_undirected**(*G*)
Convert MultiDiGraph to undirected MultiGraph.

Maintains parallel edges only if their geometries differ. Note: see also *get_digraph* to convert MultiDiGraph to DiGraph.

Parameters **G** (*networkx.MultiDiGraph*) – input graph

Returns

Return type networkx.MultiGraph

osmnx.utils_graph.**graph_from_gdfs**(*gdf_nodes*, *gdf_edges*, *graph_attrs=None*)
Convert node and edge GeoDataFrames to a MultiDiGraph.

This function is the inverse of *graph_to_gdfs* and is designed to work in conjunction with it. However, you can convert arbitrary node and edge GeoDataFrames as long as gdf_nodes is uniquely indexed by *osmid* and gdf_edges is uniquely multi-indexed by *u*, *v*, *key* (following normal MultiDiGraph structure). This allows you to load any node/edge shapefiles or GeoPackage layers as GeoDataFrames then convert them to a MultiDiGraph for graph analysis.

Parameters

- **gdf_nodes** (*geopandas.GeoDataFrame*) – GeoDataFrame of graph nodes uniquely indexed by osmid
- **gdf_edges** (*geopandas.GeoDataFrame*) – GeoDataFrame of graph edges uniquely multi-indexed by u, v, key
- **graph_attrs** (*dict*) – the new G.graph attribute dict. if None, use crs from gdf_edges as the only graph-level attribute (gdf_edges must have crs attribute set)

Returns **G**

Return type networkx.MultiDiGraph

osmnx.utils_graph.**graph_to_gdfs**(*G*, *nodes=True*, *edges=True*, *node_geometry=True*, *fill_edge_geometry=True*)
Convert a MultiDiGraph to node and/or edge GeoDataFrames.

This function is the inverse of *graph_from_gdfs*.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **nodes** (*bool*) – if True, convert graph nodes to a GeoDataFrame and return it
- **edges** (*bool*) – if True, convert graph edges to a GeoDataFrame and return it
- **node_geometry** (*bool*) – if True, create a geometry column from node x and y data
- **fill_edge_geometry** (*bool*) – if True, fill in missing edge geometry fields using nodes u and v

Returns gdf_nodes or gdf_edges or tuple of (gdf_nodes, gdf_edges)

Return type geopandas.GeoDataFrame or tuple

osmnx.utils_graph.**remove_isolated_nodes**(*G*)
Remove from a graph all nodes that have no incident edges.

Parameters **G** (*networkx.MultiDiGraph*) – graph from which to remove isolated nodes

Returns **G** – graph with all isolated nodes removed

Return type networkx.MultiDiGraph

CHAPTER

FOUR

SUPPORT

If you have a usage question, please ask it on StackOverflow. If you've discovered a bug in OSMnx, please open an issue at the OSMnx GitHub repo.

CHAPTER

FIVE

LICENSE

The project is licensed under the MIT license.

CHAPTER

SIX

INDICES

- genindex
- modindex
- search

PYTHON MODULE INDEX

O

osmnx.bearing, 7
osmnx.distance, 8
osmnx.downloader, 11
osmnx.elevation, 11
osmnx.folium, 12
osmnx.geocoder, 13
osmnx.geometries, 14
osmnx.graph, 17
osmnx.io, 21
osmnx.osm_xml, 23
osmnx.plot, 25
osmnx.projection, 28
osmnx.settings, 29
osmnx.simplification, 29
osmnx.speed, 31
osmnx.stats, 32
osmnx.truncate, 34
osmnx.utils, 36
osmnx.utils_geo, 38
osmnx.utils_graph, 39

INDEX

A

add_edge_bearings() (*in module osmnx.bearing*), 7
add_edge_grades() (*in module osmnx.elevation*), 11
add_edge_lengths() (*in module osmnx.utils_graph*), 39
add_edge_speeds() (*in module osmnx.speed*), 31
add_edge_travel_times() (*in module osmnx.speed*), 31
add_node_elevations() (*in module osmnx.elevation*), 12

B

basic_stats() (*in module osmnx.stats*), 32
bbox_from_point() (*in module osmnx.utils_geo*), 38
bbox_to_poly() (*in module osmnx.utils_geo*), 38

C

citation() (*in module osmnx.utils*), 36
config() (*in module osmnx.utils*), 36
consolidate_intersections() (*in module osmnx.simplification*), 29
count_streets_per_node() (*in module osmnx.utils_graph*), 40

E

euclidean_dist_vec() (*in module osmnx.distance*), 8
extended_stats() (*in module osmnx.stats*), 33

G

geocode() (*in module osmnx.geocoder*), 13
geocode_to_gdf() (*in module osmnx.geocoder*), 13
geometries_from_address() (*in module osmnx.geometries*), 14
geometries_from_bbox() (*in module osmnx.geometries*), 15
geometries_from_place() (*in module osmnx.geometries*), 15
geometries_from_point() (*in module osmnx.geometries*), 16
geometries_from_polygon() (*in module osmnx.geometries*), 16
geometries_from_xml() (*in module osmnx.geometries*), 17
get_bearing() (*in module osmnx.bearing*), 7
get_colors() (*in module osmnx.plot*), 25
get_digraph() (*in module osmnx.utils_graph*), 40
get_edge_colors_by_attr() (*in module osmnx.plot*), 25
get_largest_component() (*in module osmnx.utils_graph*), 40
get_nearest_edge() (*in module osmnx.distance*), 8
get_nearest_edges() (*in module osmnx.distance*), 8
get_nearest_node() (*in module osmnx.distance*), 9
get_nearest_nodes() (*in module osmnx.distance*), 9
get_node_colors_by_attr() (*in module osmnx.plot*), 25
get_route_edge_attributes() (*in module osmnx.utils_graph*), 40
get_undirected() (*in module osmnx.utils_graph*), 41
graph_from_address() (*in module osmnx.graph*), 17
graph_from_bbox() (*in module osmnx.graph*), 18
graph_from_gdfs() (*in module osmnx.utils_graph*), 41
graph_from_place() (*in module osmnx.graph*), 19
graph_from_point() (*in module osmnx.graph*), 20
graph_from_polygon() (*in module osmnx.graph*), 20
graph_from_xml() (*in module osmnx.graph*), 21
graph_to_gdfs() (*in module osmnx.utils_graph*), 41
great_circle_vec() (*in module osmnx.distance*), 10

K

k_shortest_paths() (*in module osmnx.distance*), 10

L

load_graphml() (*in module osmnx.io*), 21
log() (*in module osmnx.utils*), 38

M

module
    osmnx.bearing, 7
    osmnx.distance, 8
    osmnx.downloader, 11
    osmnx.elevation, 11
    osmnx.folium, 12
    osmnx.geocoder, 13
    osmnx.geometries, 14
    osmnx.graph, 17
    osmnx.io, 21
    osmnx.osm_xml, 23
    osmnx.plot, 25
    osmnx.projection, 28
    osmnx.settings, 29
    osmnx.simplification, 29
    osmnx.speed, 31
    osmnx.stats, 32
    osmnx.truncate, 34
    osmnx.utils, 36
    osmnx.utils_geo, 38
    osmnx.utils_graph, 39

N

nominatim_request() (*in module osmnx.downloader*), 11

O

osmnx.bearing
    module, 7
osmnx.distance
    module, 8
osmnx.downloader
    module, 11
osmnx.elevation
    module, 11
osmnx.folium
    module, 12
osmnx.geocoder
    module, 13
osmnx.geometries
    module, 14
osmnx.graph
    module, 17
osmnx.io
    module, 21
osmnx.osm_xml
    module, 23
osmnx.plot
    module, 25
osmnx.projection
    module, 28
osmnx.settings
    module, 29
osmnx.simplification
    module, 29
osmnx.speed
    module, 31
osmnx.stats
    module, 32
osmnx.truncate
    module, 34
osmnx.utils
    module, 36
osmnx.utils_geo
    module, 38
osmnx.utils_graph
    module, 39
overpass_request() (*in module osmnx.downloader*), 11

P

plot_figure_ground() (*in module osmnx.plot*), 26
plot_footprints() (*in module osmnx.plot*), 26
plot_graph() (*in module osmnx.plot*), 27
plot_graph_folium() (*in module osmnx.folium*), 12
plot_graph_route() (*in module osmnx.plot*), 28
plot_graph_routes() (*in module osmnx.plot*), 28
plot_route_folium() (*in module osmnx.folium*), 13
project_gdf() (*in module osmnx.projection*), 28
project_geometry() (*in module osmnx.projection*), 29
project_graph() (*in module osmnx.projection*), 29

R

redistribute_vertices() (*in module osmnx.utils_geo*), 39
remove_isolated_nodes() (*in module osmnx.utils_graph*), 42
round_geometry_coords() (*in module osmnx.utils_geo*), 39

S

save_graph_geopackage() (*in module osmnx.io*), 21
save_graph_shapefile() (*in module osmnx.io*), 22
save_graph_xml() (*in module osmnx.io*), 22
save_graph_xml() (*in module osmnx.osm_xml*), 23
save_graphml() (*in module osmnx.io*), 23
shortest_path() (*in module osmnx.distance*), 10

simplify_graph() (*in module osmnx.simplification*), 30

T

truncate_graph_bbox() (*in module osmnx.truncate*), 34
truncate_graph_dist() (*in module osmnx.truncate*), 34
truncate_graph_polygon() (*in module osmnx.truncate*), 35
ts() (*in module osmnx.utils*), 38