OSMnx

Release 1.8.0

Geoff Boeing

Dec 10, 2023

CONTENTS

OSMnx is a Python package to easily download, model, analyze, and visualize street networks and other geospatial features from OpenStreetMap. You can download and model walking, driving, or biking networks with a single line of code then analyze and visualize them. You can just as easily work with urban amenities/points of interest, building footprints, transit stops, elevation data, street orientations, speed/travel time, and routing.

CHAPTER

ONE

CITATION

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.

CHAPTER

TWO

GETTING STARTED

First read the *Getting Started* guide for an introduction to the package and FAQ.

Then work through the OSMnx Examples gallery for step-by-step tutorials and sample code.

CHAPTER

THREE

INSTALLATION

Follow the *Installation* guide to install OSMnx.

CHAPTER

FOUR

SUPPORT

If you have any trouble, consult the *User Reference*. The OSMnx repository is hosted on GitHub. If you have a "how-to" or usage question, please ask it on StackOverflow, as we reserve the repository's issue tracker for bug tracking and feature development.

CHAPTER

FIVE

LICENSE

OSMnx is open source and licensed under the MIT license. OpenStreetMap's open data license requires that derivative works provide proper attribution. Refer to the *Getting Started* guide for usage limitations.

CHAPTER

SIX

DOCUMENTATION

6.1 Getting Started

6.1.1 Get Started in 4 Steps

1. Install OSMnx by following the *Installation* guide.
2. Read "*Introducing OSMnx*" below on this page.
3. Work through the OSMnx Examples gallery for step-by-step tutorials and sample code.
4. Consult the *User Reference* for complete details on using the package.

Finally, if you're not already familiar with NetworkX and GeoPandas, make sure you read their user guides as OSMnx uses their data structures and functionality.

6.1.2 Introducing OSMnx

This quick introduction explains key concepts and the basic functionality of OSMnx.

Overview

OSMnx is pronounced as the initialism: "oh-ess-em-en-ex". It is built on top of NetworkX and GeoPandas, and interacts with OpenStreetMap APIs to:

- Download and model street networks or other infrastructure anywhere in the world with a single line of code
- Download geospatial features (e.g., political boundaries, building footprints, grocery stores, transit stops) as a GeoDataFrame
- Query by city name, polygon, bounding box, or point/address + distance
- Model driving, walking, biking, and other travel modes
- Attach node elevations from a local raster file or web service and calculate edge grades
- Impute missing speeds and calculate graph edge travel times
- Simplify and correct the network's topology to clean-up nodes and consolidate complex intersections
- Fast map-matching of points, routes, or trajectories to nearest graph edges or nodes
- Save/load network to/from disk as GraphML, GeoPackage, or .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
- Explore street networks and geospatial features as a static map or interactive web map
- Visualize travel distance and travel time with isoline and isochrone maps
- Plot figure-ground diagrams of street networks and building footprints

The OSMnx Examples gallery contains tutorials and demonstrations of all these features, and package usage is detailed in the *User Reference*.

Configuration

You can configure OSMnx using the `settings` module. Here you can adjust logging behavior, caching, server endpoints, and more. You can also configure OSMnx to retrieve historical snapshots of OpenStreetMap data as of a certain date. Refer to the FAQ below for server usage limitations.

Geocoding and Querying

OSMnx geocodes place names and addresses with the OpenStreetMap Nominatim API. You can use the `geocoder` module to geocode place names or addresses to lat-lon coordinates. Or, you can retrieve place boundaries or any other OpenStreetMap elements by name or ID.

Using the `features` and `graph` modules, as described below, you can download data by lat-lon point, address, bounding box, bounding polygon, or place name (e.g., neighborhood, city, county, etc).

Urban Amenities

Using OSMnx's `features` module, you can search for and download geospatial features (such as building footprints, grocery stores, schools, public parks, transit stops, etc) from the OpenStreetMap Overpass API as a GeoPandas GeoDataFrame. This uses OpenStreetMap tags to search for matching elements.

Modeling a Network

Using OSMnx's `graph` module, you can retrieve any spatial network data (such as streets, paths, rail, canals, etc) from the Overpass API and model them as NetworkX MultiDiGraphs.

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. Because these graphs are nonplanar, they correctly model the topology of interchanges, bridges, and tunnels. That is, edge crossings in a two-dimensional plane are not intersections in an OSMnx model unless they represent true junctions in the three-dimensional real world.

The `graph` module uses filters to query the Overpass API: you can either specify a built-in network type or provide your own custom filter with Overpass QL. Refer to the `graph` module's documentation for more details. Under the hood, OSMnx does several things to generate the best possible model. It initially creates a 500m-buffered graph before truncating it to your desired query area, to ensure accurate streets-per-node stats and to attenuate graph perimeter effects. It also simplifies the graph topology as discussed below.

Topology Clean-Up

The `simplification` module automatically processes the network's topology from the original raw OpenStreetMap data, such that nodes represent intersections/dead-ends and edges represent the street segments that link them. This takes two primary forms: graph simplification and intersection consolidation.

Graph simplification cleans up the graph's topology so that nodes represent intersections or dead-ends and edges represent street segments. This is important because in OpenStreetMap raw data, ways comprise sets of straight-line segments between nodes: that is, nodes are vertices for streets' curving line geometries, not just intersections and dead-ends. By default, OSMnx simplifies this topology by discarding non-intersection/dead-end nodes while retaining the complete true edge geometry as an edge attribute. When multiple OpenStreetMap ways are merged into a single graph edge, the ways' attribute values are collapsed into a single edge attribute value if they are all the same, or a list of unique values if any of them differ.

Intersection consolidation is important because many real-world street networks feature complex intersections and traffic circles, resulting in a cluster of graph nodes where there is really just one true intersection as we would think of it in transportation or urban design. Similarly, 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, but these 4 nodes represent a single intersection in the real world. OSMnx can consolidate such complex intersections into a single node and optionally rebuild the graph's edge topology accordingly.

Converting, Projecting, Saving

OSMnx can convert a MultiDiGraph to a MultiGraph if you prefer an undirected representation of the network, or to a DiGraph if you prefer a directed representation without any parallel edges. It can also convert a MultiDiGraph to/from GeoPandas node and edge GeoDataFrames. The nodes GeoDataFrame is indexed by OSM ID and the edges GeoDataFrame is multi-indexed by `u, v, key` just like a NetworkX edge. This allows you to load arbitrary node/edge ShapeFiles or GeoPackage layers as GeoDataFrames then model them as a MultiDiGraph for graph analysis.

You can easily project your graph to different coordinate reference systems using the `projection` module. If you're unsure which CRS you want to project to, OSMnx can automatically determine an appropriate UTM CRS for you.

Using the `io` module, you can save your graph to disk as a GraphML file (to load into other network analysis software) or a GeoPackage (to load into other GIS software). Use the GraphML format whenever saving a graph for later work with OSMnx.

Working with Elevation

The `elevation` module lets you automatically attach elevations to the graph's nodes from a local raster file or a web service like the Google Maps Elevation API. You can also calculate edge grades (i.e., rise-over-run) and analyze the steepness of certain streets or routes.

Network Statistics

You can use the `stats` module to calculate a variety of geometric and topological measures as well as street network bearing/orientation statistics. These measures define streets as the edges in an undirected representation of the graph to prevent double-counting bidirectional edges of a two-way street. You can easily generate common stats in transportation studies, urban design, and network science, including intersection density, circuity, average node degree (connectedness), betweenness centrality, and much more.

You can also use NetworkX directly to calculate additional topological network measures.

Routing

The `speed` module can impute missing speeds to the graph edges. This imputation can obviously be imprecise, and the user can override it by passing in arguments that define local speed limits. It can also calculate free-flow travel times for each edge.

The `distance` module can find the nearest node(s) or edge(s) to coordinates using a fast spatial index. The `routing` module can solve shortest paths for network routing, parallelized with multiprocessing, using different weights (e.g., distance, travel time, elevation change, etc).

Visualization

You can plot graphs, routes, network figure-ground diagrams, building footprints, and street network orientation rose diagrams (aka, polar histograms) with the `plot` module. You can also explore street networks, routes, or geospatial features as interactive Folium web maps.

6.1.3 More Info

All of this functionality is demonstrated step-by-step in the OSMnx Examples gallery, and usage is detailed in the *User Reference*. More feature development details are in the Changelog. Consult the *Further Reading* resources for additional technical details and research.

6.1.4 Frequently Asked Questions

How do I install OSMnx? Follow the *Installation* guide.

How do I use OSMnx? Check out the step-by-step tutorials in the OSMnx Examples gallery.

How does this or that function work? Consult the *User Reference*.

What can I do with OSMnx? Check out recent projects that use OSMnx.

I have a usage question. Please ask it on StackOverflow.

Are there any usage limitations? Yes. Refer to the Nominatim Usage Policy and Overpass Commons documentation for usage limitations and restrictions that you must adhere to at all times. If you use an alternative Nominatim/Overpass instance, ensure you understand and obey their usage policies. If you need to exceed these limitations, consider installing your own hosted instance and setting OSMnx to use it.

6.2 Installation

6.2.1 Conda

The official supported way to install OSMnx is with conda:

```
conda create -n ox -c conda-forge --strict-channel-priority osmnx
```

This creates a new conda environment and installs OSMnx into it, via the conda-forge channel. If you want other packages, such as `jupyterlab`, installed in this environment as well, just add their names after `osmnx` above.

To upgrade OSMnx to a newer release, 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. See the conda and conda-forge documentation for more details.

6.2.2 Docker

You can run OSMnx + JupyterLab directly from the official OSMnx Docker image.

6.2.3 Pip

If you already have all of its dependencies installed and tested on your system, you can install OSMnx with pip. OSMnx is written in pure Python and its installation alone is thus trivially simple. However, it depends on other packages written in C. Installing those dependencies with pip can be challenging depending on your specific system's configuration. Therefore, if you're not sure what you're doing, just follow the conda instructions above to avoid installation problems.

6.3 User Reference

This is the User Reference for the OSMnx package. If you are looking for an introduction to OSMnx, read the *Getting Started* guide.

This guide describes the usage of OSMnx's public API. Every function can be accessed via `ox.module_name.function_name()` and many can also be accessed directly via `ox.function_name()` as a shortcut.

6.3.1 osmnx.bearing module

Calculate graph edge bearings.

`osmnx.bearing.`**`add_edge_bearings`**(*G*, *precision=None*)

Add compass *bearing* attributes to all graph edges.

Vectorized function to calculate (initial) bearing from origin node to destination node for each edge in a directed, unprojected graph then add these bearings as new edge attributes. Bearing represents angle in degrees (clockwise) between north and the geodesic line from the origin node to the destination node. Ignores self-loop edges as their bearings are undefined.

Parameters

- **G** (*networkx.MultiDiGraph*) – unprojected graph
- **precision** (*int*) – deprecated, do not use

Returns
G – graph with edge bearing attributes

Return type
networkx.MultiDiGraph

`osmnx.bearing.`**`calculate_bearing`**(*lat1*, *lon1*, *lat2*, *lon2*)

Calculate the compass bearing(s) between pairs of lat-lon points.

Vectorized function to calculate initial bearings between two points' coordinates or between arrays of points' coordinates. Expects coordinates in decimal degrees. Bearing represents the clockwise angle in degrees between north and the geodesic line from (lat1, lon1) to (lat2, lon2).

Parameters

- **lat1** (*float or numpy.array of float*) – first point's latitude coordinate
- **lon1** (*float or numpy.array of float*) – first point's longitude coordinate
- **lat2** (*float or numpy.array of float*) – second point's latitude coordinate

- **lon2** (*float or numpy.array of float*) – second point's longitude coordinate

Returns
bearing – the bearing(s) in decimal degrees

Return type
float or numpy.array of float

osmnx.bearing.**orientation_entropy**(*Gu*, *num_bins=36*, *min_length=0*, *weight=None*)

Calculate undirected graph's orientation entropy.

Orientation entropy is the entropy of its edges' bidirectional bearings across evenly spaced bins. Ignores self-loop edges as their bearings are undefined.

For more info see: Boeing, G. 2019. "Urban Spatial Order: Street Network Orientation, Configuration, and Entropy." Applied Network Science, 4 (1), 67. https://doi.org/10.1007/s41109-019-0189-1

Parameters

- **Gu** (*networkx.MultiGraph*) – undirected, unprojected graph with *bearing* attributes on each edge
- **num_bins** (*int*) – number of bins; for example, if *num_bins=36* is provided, then each bin will represent 10 degrees around the compass
- **min_length** (*float*) – ignore edges with *length* attributes less than *min_length*; useful to ignore the noise of many very short edges
- **weight** (*string*) – if not None, weight edges' bearings by this (non-null) edge attribute. for example, if "length" is provided, this will return 1 bearing observation per meter per street, which could result in a very large *bearings* array.

Returns
entropy – the graph's orientation entropy

Return type
float

osmnx.bearing.**plot_orientation**(*Gu*, *num_bins=36*, *min_length=0*, *weight=None*, *ax=None*, *figsize=(5, 5)*, *area=True*, *color='#003366'*, *edgecolor='k'*, *linewidth=0.5*, *alpha=0.7*, *title=None*, *title_y=1.05*, *title_font=None*, *xtick_font=None*)

Do not use: deprecated.

The plot_orientation function moved to the plot module. Calling it via the bearing module will raise an error in a future release.

Parameters

- **Gu** (*networkx.MultiGraph*) – deprecated, do not use
- **num_bins** (*int*) – deprecated, do not use
- **min_length** (*float*) – deprecated, do not use
- **weight** (*string*) – deprecated, do not use
- **ax** (*matplotlib.axes.PolarAxesSubplot*) – deprecated, do not use
- **figsize** (*tuple*) – deprecated, do not use
- **area** (*bool*) – deprecated, do not use
- **color** (*string*) – deprecated, do not use
- **edgecolor** (*string*) – deprecated, do not use

- **linewidth** (*float*) – deprecated, do not use
- **alpha** (*float*) – deprecated, do not use
- **title** (*string*) – deprecated, do not use
- **title_y** (*float*) – deprecated, do not use
- **title_font** (*dict*) – deprecated, do not use
- **xtick_font** (*dict*) – deprecated, do not use

Returns
fig, ax – matplotlib figure, axis

Return type
tuple

6.3.2 osmnx.distance module

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

osmnx.distance.**add_edge_lengths**(*G*, *precision=None*, *edges=None*)

Add *length* attribute (in meters) to each edge.

Vectorized function to calculate great-circle distance between each edge's incident nodes. Ensure graph is in unprojected coordinates, and 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. It calculates edge lengths as the great-circle distance from node *u* to node *v*. When OSMnx automatically runs this function upon graph creation, it does it before simplifying the graph: thus it calculates the straight-line lengths of edge segments that are themselves all straight. Only after simplification do edges take on a (potentially) curvilinear geometry. If you wish to calculate edge lengths later, you are calculating straight-line distances which necessarily ignore the curvilinear geometry. You only want to run this function on a graph with all straight edges (such as is the case with an unsimplified graph).

Parameters

- **G** (*networkx.MultiDiGraph*) – unprojected, unsimplified input graph
- **precision** (*int*) – deprecated, do not use
- **edges** (*tuple*) – tuple of (u, v, k) tuples representing subset of edges to add length attributes to. if None, add lengths to all edges.

Returns
G – graph with edge length attributes

Return type
networkx.MultiDiGraph

osmnx.distance.**euclidean**(*y1*, *x1*, *y2*, *x2*)

Calculate Euclidean distances between pairs of points.

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

Parameters

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

- **x2** (*float or numpy.array of float*) – second point's x coordinate

Returns
dist – distance from each (x1, y1) to each (x2, y2) in coordinates' units

Return type
float or numpy.array of float

osmnx.distance.**euclidean_dist_vec**(*y1*, *x1*, *y2*, *x2*)

Do not use, deprecated.

The *euclidean_dist_vec* function has been renamed *euclidean*. Calling *euclidean_dist_vec* will raise an error in a future release.

Parameters

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

Returns
dist – distance from each (x1, y1) to each (x2, y2) in coordinates' units

Return type
float or numpy.array of float

osmnx.distance.**great_circle**(*lat1*, *lon1*, *lat2*, *lon2*, *earth_radius=6371009*)

Calculate great-circle distances between pairs of 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 numpy.array of float*) – first point's latitude coordinate
- **lon1** (*float or numpy.array of float*) – first point's longitude coordinate
- **lat2** (*float or numpy.array of float*) – second point's latitude coordinate
- **lon2** (*float or numpy.array of float*) – second point's longitude coordinate
- **earth_radius** (*float*) – earth's radius in units in which distance will be returned (default is meters)

Returns
dist – distance from each (lat1, lon1) to each (lat2, lon2) in units of earth_radius

Return type
float or numpy.array of float

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

Do not use, deprecated.

The *great_circle_vec* function has been renamed *great_circle*. Calling *great_circle_vec* will raise an error in a future release.

Parameters

- **lat1** (*float or numpy.array of float*) – first point's latitude coordinate
- **lng1** (*float or numpy.array of float*) – first point's longitude coordinate

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

Returns
dist – distance from each (lat1, lng1) to each (lat2, lng2) in units of earth_radius

Return type
float or numpy.array of float

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

Do not use, deprecated.

The *k_shortest_paths* function has moved to the *routing* module. Calling it via the *distance* module will raise an error in a future release.

Parameters

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

Yields
path (*list*) – a generator of *k* shortest paths ordered by total weight. each path is a list of node IDs.

osmnx.distance.**nearest_edges**(*G*, *X*, *Y*, *interpolate=None*, *return_dist=False*)

Find the nearest edge to a point or to each of several points.

If *X* and *Y* are single coordinate values, this will return the nearest edge to that point. If *X* and *Y* are lists of coordinate values, this will return the nearest edge to each point. This function uses an R-tree spatial index and minimizes the euclidean distance from each point to the possible matches. For accurate results, use a projected graph and points.

Parameters

- **G** (*networkx.MultiDiGraph*) – graph in which to find nearest edges
- **X** (*float or list*) – points' x (longitude) coordinates, in same CRS/units as graph and containing no nulls
- **Y** (*float or list*) – points' y (latitude) coordinates, in same CRS/units as graph and containing no nulls
- **interpolate** (*float*) – deprecated, do not use
- **return_dist** (*bool*) – optionally also return distance between points and nearest edges

Returns
ne or (ne, dist) – nearest edges as (u, v, key) or optionally a tuple where *dist* contains distances between the points and their nearest edges

Return type
tuple or list

osmnx.distance.**nearest_nodes**(*G*, *X*, *Y*, *return_dist=False*)

Find the nearest node to a point or to each of several points.

If *X* and *Y* are single coordinate values, this will return the nearest node to that point. If *X* and *Y* are lists of coordinate values, this will return the nearest node to each point.

If the graph is projected, this uses a k-d tree for euclidean nearest neighbor search, which requires that scipy is installed as an optional dependency. If it is unprojected, this uses a ball tree for haversine nearest neighbor search, which requires that scikit-learn is installed as an optional dependency.

Parameters

- **G** (*networkx.MultiDiGraph*) – graph in which to find nearest nodes
- **X** (*float or list*) – points' x (longitude) coordinates, in same CRS/units as graph and containing no nulls
- **Y** (*float or list*) – points' y (latitude) coordinates, in same CRS/units as graph and containing no nulls
- **return_dist** (*bool*) – optionally also return distance between points and nearest nodes

Returns

nn or (nn, dist) – nearest node IDs or optionally a tuple where *dist* contains distances between the points and their nearest nodes

Return type

int/list or tuple

osmnx.distance.**shortest_path**(*G*, *orig*, *dest*, *weight='length'*, *cpus=1*)

Do not use, deprecated.

The *shortest_path* function has moved to the *routing* module. Calling it via the *distance* module will raise an error in a future release.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **orig** (*int or list*) – origin node ID, or a list of origin node IDs
- **dest** (*int or list*) – destination node ID, or a list of destination node IDs
- **weight** (*string*) – edge attribute to minimize when solving shortest path
- **cpus** (*int*) – how many CPU cores to use; if None, use all available

Returns

path – list of node IDs constituting the shortest path, or, if orig and dest are lists, then a list of path lists

Return type

list

6.3.3 osmnx.elevation module

Add node elevations from raster files or web APIs, and calculate edge grades.

osmnx.elevation.**add_edge_grades**(*G*, *add_absolute=True*, *precision=None*)

Add *grade* attribute to each graph edge.

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

See also the *add_node_elevations_raster* and *add_node_elevations_google* functions.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph with *elevation* node attribute
- **add_absolute** (*bool*) – if True, also add absolute value of grade as *grade_abs* attribute
- **precision** (*int*) – deprecated, do not use

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

Return type
networkx.MultiDiGraph

osmnx.elevation.**add_node_elevations_google**(*G*, *api_key=None*, *batch_size=350*, *pause=0*, *max_locations_per_batch=None*, *precision=None*, *url_template=None*)

Add an *elevation* (meters) attribute to each node using a web service.

By default, this uses the Google Maps Elevation API but you can optionally use an equivalent API with the same interface and response format, such as Open Topo Data, via the *settings* module's *elevation_url_template*. The Google Maps Elevation API requires an API key but other providers may not.

For a free local alternative see the *add_node_elevations_raster* function. See also the *add_edge_grades* function.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **api_key** (*string*) – a valid API key, can be None if the API does not require a key
- **batch_size** (*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 allowed)
- **pause** (*float*) – time to pause between API calls, which can be increased if you get rate limited
- **max_locations_per_batch** (*int*) – deprecated, do not use
- **precision** (*int*) – deprecated, do not use
- **url_template** (*string*) – deprecated, do not use

Returns
G – graph with node elevation attributes

Return type
networkx.MultiDiGraph

osmnx.elevation.**add_node_elevations_raster**(*G*, *filepath*, *band=1*, *cpus=None*)

Add *elevation* attribute to each node from local raster file(s).

If *filepath* is a list of paths, this will generate a virtual raster composed of the files at those paths as an intermediate step.

See also the *add_edge_grades* function.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph, in same CRS as raster
- **filepath** (*string or pathlib.Path or list of strings/Paths*) – path (or list of paths) to the raster file(s) to query
- **band** (*int*) – which raster band to query
- **cpus** (*int*) – how many CPU cores to use; if None, use all available

Returns

G – graph with node elevation attributes

Return type

networkx.MultiDiGraph

6.3.4 osmnx.features module

Download OpenStreetMap geospatial features' geometries and attributes.

Retrieve points of interest, building footprints, transit lines/stops, or any other map features from OSM, including their geometries and attribute data, then construct a GeoDataFrame of them. You can use this module to query for nodes, ways, and relations (the latter of type "multipolygon" or "boundary" only) by passing a dictionary of desired OSM tags.

For more details, see https://wiki.openstreetmap.org/wiki/Map_features and https://wiki.openstreetmap.org/wiki/Elements

Refer to the Getting Started guide for usage limitations.

osmnx.features.**features_from_address**(*address*, *tags*, *dist=1000*)

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

You can use the *settings* module to retrieve a snapshot of historical OSM data as of a certain date, or to configure the Overpass server timeout, memory allocation, and other custom settings.

For more details, see: https://wiki.openstreetmap.org/wiki/Map_features

Parameters

- **address** (*string*) – the address to geocode and use as the central point around which to get the features
- **tags** (*dict*) – Dict of tags used for finding elements 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

osmnx.features.**features_from_bbox**(*north*, *south*, *east*, *west*, *tags*)

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

You can use the *settings* module to retrieve a snapshot of historical OSM data as of a certain date, or to configure the Overpass server timeout, memory allocation, and other custom settings.

For more details, see: https://wiki.openstreetmap.org/wiki/Map_features

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 elements 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

osmnx.features.**features_from_place**(*query*, *tags*, *which_result=None*, *buffer_dist=None*)

Create GeoDataFrame of OSM features within boundaries of some 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 features within it using the *features_from_address* function, which geocodes the place name to a point and gets the features 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 *features_from_polygon* function.

You can use the *settings* module to retrieve a snapshot of historical OSM data as of a certain date, or to configure the Overpass server timeout, memory allocation, and other custom settings.

For more details, see: https://wiki.openstreetmap.org/wiki/Map_features

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 elements 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*) – deprecated, do not use

Returns
gdf

Return type
geopandas.GeoDataFrame

osmnx.features.**features_from_point**(*center_point*, *tags*, *dist=1000*)

Create GeoDataFrame of OSM features within some distance N, S, E, W of a point.

You can use the *settings* module to retrieve a snapshot of historical OSM data as of a certain date, or to configure the Overpass server timeout, memory allocation, and other custom settings.

For more details, see: https://wiki.openstreetmap.org/wiki/Map_features

Parameters

- **center_point** (*tuple*) – the (lat, lon) center point around which to get the features
- **tags** (*dict*) – Dict of tags used for finding elements 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

osmnx.features.**features_from_polygon**(*polygon*, *tags*)

Create GeoDataFrame of OSM features within boundaries of a (multi)polygon.

You can use the *settings* module to retrieve a snapshot of historical OSM data as of a certain date, or to configure the Overpass server timeout, memory allocation, and other custom settings.

For more details, see: https://wiki.openstreetmap.org/wiki/Map_features

Parameters

- **polygon** (*shapely.geometry.Polygon or shapely.geometry.MultiPolygon*) – geographic boundaries to fetch features within
- **tags** (*dict*) – Dict of tags used for finding elements 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

`osmnx.features.`**`features_from_xml`**(*filepath*, *polygon=None*, *tags=None*, *encoding='utf-8'*)

Create a GeoDataFrame of OSM features in an OSM-formatted XML file.

Because this function creates a GeoDataFrame of features 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, features_from_xml() will return features for all of the tagged elements in the file. If they are supplied they will be used to filter the final GeoDataFrame.

For more details, see: https://wiki.openstreetmap.org/wiki/Map_features

Parameters

- **filepath** (*string or pathlib.Path*) – path to file containing OSM XML data
- **polygon** (*shapely.geometry.Polygon*) – optional geographic boundary to filter elements
- **tags** (*dict*) – optional dict of tags for filtering elements 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.
- **encoding** (*string*) – the XML file's character encoding

Returns
gdf

Return type
geopandas.GeoDataFrame

6.3.5 osmnx.geocoder module

Geocode place names or addresses or retrieve OSM elements by place name or ID.

This module uses the Nominatim API's "search" and "lookup" endpoints. For more details see https://wiki.openstreetmap.org/wiki/Elements and https://nominatim.org/.

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

Geocode place names or addresses to (lat, lon) with the Nominatim API.

This geocodes the query via the Nominatim "search" endpoint.

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

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

Return type
tuple

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

Retrieve OSM elements by place name or OSM ID with the Nominatim API.

If searching by place name, the *query* argument can be a string or structured dict, or a list of such strings/dicts to send to the geocoder. This uses the Nominatim "search" endpoint to geocode the place name to the best-matching OSM element, then returns that element and its attribute data.

You can instead query by OSM ID by passing *by_osmid=True*. This uses the Nominatim "lookup" endpoint to retrieve the OSM element with that ID. In this case, the function treats the *query* argument as an OSM ID (or list of OSM IDs), which must be prepended with their types: node (N), way (W), or relation (R) in accordance with the Nominatim API format. For example, *query=["R2192363", "N240109189", "W427818536"]*.

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

Parameters

- **query** (*string or dict or list of strings/dicts*) – query string(s) or structured dict(s) to geocode
- **which_result** (*int*) – which search result to return. 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. ignored if by_osmid=True.
- **by_osmid** (*bool*) – if True, treat query as an OSM ID lookup rather than text search
- **buffer_dist** (*float*) – deprecated, do not use

Returns
gdf – a GeoDataFrame with one row for each query

Return type
geopandas.GeoDataFrame

6.3.6 osmnx.graph module

Download and create graphs from OpenStreetMap data.

This module uses filters to query the Overpass API: you can either specify a built-in network type or provide your own custom filter with Overpass QL.

Refer to the Getting Started guide for usage limitations.

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=None*, *custom_filter=None*)

Download and create a graph within some distance of an address.

You can use the *settings* module to retrieve a snapshot of historical OSM data as of a certain date, or to configure the Overpass server timeout, memory allocation, and other custom settings.

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*) – deprecated, do not use
- **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.

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

Notes

Very large query areas use the *utils_geo._consolidate_subdivide_geometry* function to automatically make multiple requests: see that function's documentation for caveats.

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

Download and create a graph within some bounding box.

You can use the *settings* module to retrieve a snapshot of historical OSM data as of a certain date, or to configure the Overpass server timeout, memory allocation, and other custom settings.

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*) – deprecated, do not use
- **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

Very large query areas use the *utils_geo._consolidate_subdivide_geometry* function to automatically make multiple requests: see that function's documentation for caveats.

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=None*, *custom_filter=None*)

Download and create a graph within the boundaries of some 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.

You can use the *settings* module to retrieve a snapshot of historical OSM data as of a certain date, or to configure the Overpass server timeout, memory allocation, and other custom settings.

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*) – deprecated, do not use
- **clean_periphery** (*bool*) – deprecated, do not use

- **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

Very large query areas use the *utils_geo._consolidate_subdivide_geometry* function to automatically make multiple requests: see that function's documentation for caveats.

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=None*, *custom_filter=None*)

Download and create a graph within some distance of a (lat, lon) point.

You can use the *settings* module to retrieve a snapshot of historical OSM data as of a certain date, or to configure the Overpass server timeout, memory allocation, and other custom settings.

Parameters

- **center_point** (*tuple*) – the (lat, lon) 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,*) – deprecated, do not use
- **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

Very large query areas use the *utils_geo._consolidate_subdivide_geometry* function to automatically make multiple requests: see that function's documentation for caveats.

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

Download and create a graph within the boundaries of a (multi)polygon.

You can use the *settings* module to retrieve a snapshot of historical OSM data as of a certain date, or to configure the Overpass server timeout, memory allocation, and other custom settings.

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*) – deprecated, do not use
- **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

Very large query areas use the *utils_geo._consolidate_subdivide_geometry* function to automatically make multiple requests: see that function's documentation for caveats.

osmnx.graph.**graph_from_xml**(*filepath*, *bidirectional=False*, *simplify=True*, *retain_all=False*, *encoding='utf-8'*)

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

Do not load an XML file generated by OSMnx: this use case is not supported and may not behave as expected. To save/load graphs to/from disk for later use in OSMnx, use the *io.save_graphml* and *io.load_graphml* functions instead.

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.
- **encoding** (*string*) – the XML file's character encoding

Returns
G

Return type
networkx.MultiDiGraph

6.3.7 osmnx.io module

Serialize graphs to/from files on disk.

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

Load an OSMnx-saved GraphML file from disk or GraphML string.

This function converts node, edge, and graph-level attributes (serialized as strings) to their appropriate data types. These can be customized as needed by passing in dtypes arguments providing types or custom converter functions. For example, if you want to convert some attribute's values to *bool*, consider using the built-in *ox.io._convert_bool_string* function to properly handle "True"/"False" string literals as True/False booleans: *ox.load_graphml(fp, node_dtypes={my_attr: ox.io._convert_bool_string})*.

If you manually configured the *all_oneway=True* setting, you may need to manually specify here that edge *oneway* attributes should be type *str*.

Note that you must pass one and only one of *filepath* or *graphml_str*. If passing *graphml_str*, you may need to decode the bytes read from your file before converting to string to pass to this function.

Parameters

- **filepath** (*string or pathlib.Path*) – path to the GraphML file
- **graphml_str** (*string*) – a valid and decoded string representation of a GraphML file's contents
- **node_dtypes** (*dict*) – dict of node attribute names:types to convert values' data types. the type can be a python type or a custom string converter function.
- **edge_dtypes** (*dict*) – dict of edge attribute names:types to convert values' data types. the type can be a python type or a custom string converter function.
- **graph_dtypes** (*dict*) – dict of graph-level attribute names:types to convert values' data types. the type can be a python type or a custom string converter function.

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

Return type
None

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

Do not use: deprecated. Use the save_graph_geopackage function instead.

The Shapefile format is proprietary and outdated. Instead, use the superior GeoPackage file format via the save_graph_geopackage function. See http://switchfromshapefile.org/ for more information.

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

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*, *api_version=0.6*, *precision=6*)

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. As such, this function has a limited use case which does not include saving/loading graphs for subsequent OSMnx analysis. To save/load graphs to/from disk for later use in OSMnx, use the *io.save_graphml* and *io.load_graphml* functions instead. To load a graph from a .osm file that you have downloaded or generated elsewhere, use the *graph.graph_from_xml* function.

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))
```

(continues on next page)

(continued from previous page)

```
>>> ox.settings.all_oneway = True
>>> ox.settings.useful_tags_node = utn
>>> ox.settings.useful_tags_way = utw
>>> G = ox.graph_from_place('Piedmont, CA, USA', network_type='drive')
>>> ox.save_graph_xml(G, filepath='./data/graph.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.
- **api_version** (*float*) – OpenStreetMap API version to write to the XML file header
- **precision** (*int*) – number of decimal places to round latitude and longitude values

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

Return type
None

6.3.8 osmnx.plot module

Visualize street networks, routes, orientations, and geospatial features.

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'*, *edge_color='none'*, *edge_linewidth=0*, *alpha=None*, *bgcolor='#111111'*, *bbox=None*, *save=False*, *show=True*, *close=False*, *filepath=None*, *dpi=600*)

Visualize a GeoDataFrame of geospatial features' 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
- **edge_color** (*string*) – color of the edge of the footprints
- **edge_linewidth** (*float*) – width of the edge 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*)

Visualize 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*)

Visualize 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'*, *route_linewidths=4*, ***pgr_kwargs*)

Visualize 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.
- **route_linewidths** (*int or list*) – if int, 1 linewidth for all routes. if list, the linewidth for each route.

- **pgr_kwargs** – keyword arguments to pass to plot_graph_route

Returns
fig, ax – matplotlib figure, axis

Return type
tuple

osmnx.plot.**plot_orientation**(*Gu*, *num_bins=36*, *min_length=0*, *weight=None*, *ax=None*, *figsize=(5, 5)*, *area=True*, *color='#003366'*, *edgecolor='k'*, *linewidth=0.5*, *alpha=0.7*, *title=None*, *title_y=1.05*, *title_font=None*, *xtick_font=None*)

Plot a polar histogram of a spatial network's bidirectional edge bearings.

Ignores self-loop edges as their bearings are undefined.

For more info see: Boeing, G. 2019. "Urban Spatial Order: Street Network Orientation, Configuration, and Entropy." Applied Network Science, 4 (1), 67. https://doi.org/10.1007/s41109-019-0189-1

Parameters

- **Gu** (*networkx.MultiGraph*) – undirected, unprojected graph with *bearing* attributes on each edge
- **num_bins** (*int*) – number of bins; for example, if *num_bins=36* is provided, then each bin will represent 10 degrees around the compass
- **min_length** (*float*) – ignore edges with *length* attributes less than *min_length*
- **weight** (*string*) – if not None, weight edges' bearings by this (non-null) edge attribute
- **ax** (*matplotlib.axes.PolarAxesSubplot*) – if not None, plot on this preexisting axis; must have projection=polar
- **figsize** (*tuple*) – if ax is None, create new figure with size (width, height)
- **area** (*bool*) – if True, set bar length so area is proportional to frequency, otherwise set bar length so height is proportional to frequency
- **color** (*string*) – color of histogram bars
- **edgecolor** (*string*) – color of histogram bar edges
- **linewidth** (*float*) – width of histogram bar edges
- **alpha** (*float*) – opacity of histogram bars
- **title** (*string*) – title for plot
- **title_y** (*float*) – y position to place title
- **title_font** (*dict*) – the title's fontdict to pass to matplotlib
- **xtick_font** (*dict*) – the xtick labels' fontdict to pass to matplotlib

Returns
fig, ax – matplotlib figure, axis

Return type
tuple

6.3.9 osmnx.projection module

Project a graph, GeoDataFrame, or geometry to a different CRS.

osmnx.projection.**is_projected**(*crs*)

Determine if a coordinate reference system is projected or not.

Parameters
crs (*string or pyproj.CRS*) – the identifier of the coordinate reference system, which can be anything accepted by *pyproj.CRS.from_user_input()* such as an authority string or a WKT string

Returns
projected – True if crs is projected, otherwise False

Return type
bool

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

Project a GeoDataFrame from its current CRS to another.

If *to_latlong* is *True*, this projects the GeoDataFrame to the CRS defined by *settings.default_crs*, otherwise it projects it to the CRS defined by *to_crs*. If *to_crs* is *None*, it projects it to the CRS of an appropriate UTM zone given *gdf*'s bounds.

Parameters

- **gdf** (*geopandas.GeoDataFrame*) – the GeoDataFrame to be projected
- **to_crs** (*string or pyproj.CRS*) – if None, project to an appropriate UTM zone, 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_latlong* is *True*, this projects the GeoDataFrame to the CRS defined by *settings.default_crs*, otherwise it projects it to the CRS defined by *to_crs*. If *to_crs* is *None*, it projects it to the CRS of an appropriate UTM zone given *geometry*'s bounds.

Parameters

- **geometry** (*shapely geometry*) – the geometry to be projected
- **crs** (*string or pyproj.CRS*) – the initial CRS of *geometry*. if None, it will be set to *settings.default_crs*
- **to_crs** (*string or pyproj.CRS*) – if None, project to an appropriate UTM zone, 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*, *to_latlong=False*)

Project a graph from its current CRS to another.

If *to_latlong* is *True*, this projects the GeoDataFrame to the CRS defined by *settings.default_crs*, otherwise it projects it to the CRS defined by *to_crs*. If *to_crs* is *None*, it projects it to the CRS of an appropriate UTM zone given *G*'s bounds.

Parameters

- **G** (*networkx.MultiDiGraph*) – the graph to be projected
- **to_crs** (*string or pyproj.CRS*) – if None, project to an appropriate UTM zone, otherwise project to this CRS
- **to_latlong** (*bool*) – if True, project to *settings.default_crs* and ignore *to_crs*

Returns

G_proj – the projected graph

Return type

networkx.MultiDiGraph

6.3.10 osmnx.routing module

Calculate weighted shortest paths between graph nodes.

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

Solve *k* shortest paths from an origin node to a destination node.

Uses Yen's algorithm. See also *shortest_path* to solve 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 solve
- **weight** (*string*) – edge attribute to minimize when solving shortest paths. default is edge length in meters.

Yields

path (*list*) – a generator of *k* shortest paths ordered by total weight. each path is a list of node IDs.

osmnx.routing.**shortest_path**(*G*, *orig*, *dest*, *weight='length'*, *cpus=1*)

Solve shortest path from origin node(s) to destination node(s).

Uses Dijkstra's algorithm. If *orig* and *dest* are single node IDs, this will return a list of the nodes constituting the shortest path between them. If *orig* and *dest* are lists of node IDs, this will return a list of lists of the nodes constituting the shortest path between each origin-destination pair. If a path cannot be solved, this will return None for that path. You can parallelize solving multiple paths with the *cpus* parameter, but be careful to not exceed your available RAM.

See also *k_shortest_paths* to solve multiple shortest paths between a single origin and destination. For additional functionality or different solver algorithms, use NetworkX directly.

Parameters

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

- **orig** (*int or list*) – origin node ID, or a list of origin node IDs
- **dest** (*int or list*) – destination node ID, or a list of destination node IDs
- **weight** (*string*) – edge attribute to minimize when solving shortest path
- **cpus** (*int*) – how many CPU cores to use; if None, use all available

Returns
path – list of node IDs constituting the shortest path, or, if orig and dest are lists, then a list of path lists

Return type
list

6.3.11 osmnx.settings module

Global settings that can be configured by the user.

all_oneway
[bool] Only use if specifically saving to .osm XML file with the *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. This also retains original OSM string values for oneway attribute values, rather than converting them to a True/False bool. Default is *False*.

bidirectional_network_types
[list] Network types for which a fully bidirectional graph will be created. Default is *["walk"]*.

cache_folder
[string or pathlib.Path] Path to folder in which to save/load HTTP response cache, if the *use_cache* setting equals *True*. Default is *"./cache"*.

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 local cache to quickly build many graphs simultaneously with multiprocessing. Default is *False*.

data_folder
[string or pathlib.Path] Path to folder in which to save/load graph files by default. Default is *"./data"*.

default_accept_language
[string] HTTP header accept-language. Default is *"en"*.

default_access
[string] Default filter for OSM "access" key. Default is *'["access"!~"private"]'*. Note that also filtering out "access=no" ways prevents including transit-only bridges (e.g., Tilikum Crossing) from appearing in drivable road network (e.g., *'["access"!~"private|no"]'*). However, some drivable tollroads have "access=no" plus a "access:conditional" key to clarify when it is accessible, so we can't filter out all "access=no" ways by default. Best to be permissive here then remove complicated combinations of tags programatically after the full graph is downloaded and constructed.

default_crs
[string] Default coordinate reference system to set when creating graphs. Default is *"epsg:4326"*.

default_referer
[string] HTTP header referer. Default is *"OSMnx Python package (https://github.com/gboeing/osmnx)"*.

default_user_agent
[string] HTTP header user-agent. Default is *"OSMnx Python package (https://github.com/gboeing/osmnx)"*.

doh_url_template
[string] Endpoint to resolve DNS-over-HTTPS if local DNS resolution fails. Set to None to disable DoH, but see *downloader._config_dns* documentation for caveats. Default is: *"https://8.8.8.8/resolve?name={hostname}"*

elevation_url_template
[string] Endpoint of the Google Maps Elevation API (or equivalent), containing exactly two parameters: *locations* and *key*. Default is: *"https://maps.googleapis.com/maps/api/elevation/json?locations={locations}&key={key}"*
One example of an alternative equivalent would be Open Topo Data: *"https://api.opentopodata.org/v1/aster30m?locations={locations}&key={key}"*

imgs_folder
[string or pathlib.Path] Path to folder in which to save plotted images by default. Default is *"./images"*.

log_file
[bool] If True, save log output to a file in logs_folder. Default is *False*.

log_filename
[string] Name of the log file, without file extension. Default is *"osmnx"*.

log_console
[bool] If True, print log output to the console (terminal window). Default is *False*.

log_level
[int] One of Python's logger.level constants. Default is *logging.INFO*.

log_name
[string] Name of the logger. Default is *"OSMnx"*.

logs_folder
[string or pathlib.Path] Path to folder in which to save log files. Default is *"./logs"*.

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 the API. Default is *2500000000*.

memory
[int] Overpass server memory allocation size for the query, in bytes. If None, server will use its default allocation size. Use with caution. Default is *None*.

nominatim_endpoint
[string] The base API url to use for Nominatim queries. Default is *"https://nominatim.openstreetmap.org/"*.

nominatim_key
[string] Your Nominatim API key, if you are using an API instance that requires one. Default is *None*.

osm_xml_node_attrs
[list] Node attributes for saving .osm XML files with *save_graph_xml* function. Default is *["id", "timestamp", "uid", "user", "version", "changeset", "lat", "lon"]*.

osm_xml_node_tags
[list] Node tags for saving .osm XML files with *save_graph_xml* function. Default is *["highway"]*.

osm_xml_way_attrs
[list] Edge attributes for saving .osm XML files with *save_graph_xml* function. Default is *["id", "timestamp", "uid", "user", "version", "changeset"]*.

osm_xml_way_tags
[list] Edge tags for for saving .osm XML files with *save_graph_xml* function. Default is *["highway", "lanes", "maxspeed", "name", "oneway"]*.

overpass_endpoint
[string] The base API url to use for Overpass queries. Default is *"https://overpass-api.de/api"*.

overpass_rate_limit
[bool] If True, check the Overpass server status endpoint for how long to pause before making request. Necessary if server uses slot management, but can be set to False if you are running your own overpass instance without rate limiting. Default is *True*.

overpass_settings
[string] Settings string for Overpass queries. Default is *"[out:json][timeout:{timeout}]{maxsize}"*. By default, the {timeout} and {maxsize} values are set dynamically by OSMnx when used. To query, for example, historical OSM data as of a certain date: *'[out:json][timeout:90][date:"2019-10-28T19:20:00Z"]'*. Use with caution.

requests_kwargs
[dict] Optional keyword args to pass to the requests package when connecting to APIs, for example to configure authentication or provide a path to a local certificate file. More info on options such as auth, cert, verify, and proxies can be found in the requests package advanced docs. Default is *{}*.

timeout
[int] The timeout interval in seconds for HTTP requests, and (when applicable) for API to use while running the query. Default is *180*.

use_cache
[bool] If True, cache HTTP responses locally instead of calling API repeatedly for the same request. Default is *True*.

useful_tags_node
[list] OSM "node" tags to add as graph node attributes, when present in the data retrieved from OSM. Default is *["ref", "highway"]*.

useful_tags_way
[list] OSM "way" tags to add as graph edge attributes, when present in the data retrieved from OSM. Default is *["bridge", "tunnel", "oneway", "lanes", "ref", "name", "highway", "maxspeed", "service", "access", "area", "landuse", "width", "est_width", "junction"]*.

6.3.12 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. Note the tolerance represents a per-node buffering radius: for example, to consolidate nodes within 10 meters of each other, use tolerance=5.

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*, *track_merged=False*)

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. Optionally, the simplified edges can receive a *merged_edges* attribute that contains a list of all the (u, v) node pairs that were merged together.

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
- **track_merged** (*bool*) – if True, add *merged_edges* attribute on simplified edges, containing a list of all the (u, v) node pairs that were merged together

Returns

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

Return type

networkx.MultiDiGraph

6.3.13 osmnx.speed module

Calculate graph edge speeds and travel times.

osmnx.speed.**add_edge_speeds**(*G*, *hwy_speeds=None*, *fallback=None*, *precision=None*, *agg=numpy.mean*)

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

By default, this imputes free-flow travel speeds for all edges via the mean *maxspeed* value of the edges of each highway type. For highway types in the graph that have no *maxspeed* value on any edge, it assigns the mean of all *maxspeed* values in graph.

This default mean-imputation can obviously be imprecise, and the user can override it by passing in *hwy_speeds* and/or *fallback* arguments that correspond to local speed limit standards. The user can also specify a different aggregation function (such as the median) to impute missing values from the observed values.

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*) – deprecated, do not use
- **agg** (*function*) – aggregation function to impute missing values from observed values. the default is numpy.mean, but you might also consider for example numpy.median, numpy.nanmedian, or your own custom function

Returns
G – graph with speed_kph attributes on all edges

Return type
networkx.MultiDiGraph

osmnx.speed.**add_edge_travel_times**(*G*, *precision=None*)

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*) – deprecated, do not use

Returns
G – graph with travel_time attributes on all edges

Return type
networkx.MultiDiGraph

6.3.14 osmnx.stats module

Calculate geometric and topological network measures.

This module defines streets as the edges in an undirected representation of the graph. Using undirected graph edges prevents double-counting bidirectional edges of a two-way street, but may double-count a divided road's separate centerlines with different end point nodes. If *clean_periphery=True* when the graph was created (which is the default parameterization), then you will get accurate node degrees (and in turn streets-per-node counts) even at the periphery of the graph.

You can use NetworkX directly for additional topological network measures.

osmnx.stats.**basic_stats**(*G*, *area=None*, *clean_int_tol=None*)

Calculate basic descriptive geometric and topological measures of a graph.

Density measures are only calculated if *area* is provided and clean intersection measures are only calculated if *clean_int_tol* is provided.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **area** (*float*) – if not None, calculate density measures and use this value (in square meters) as the denominator
- **clean_int_tol** (*float*) – if not None, calculate consolidated intersections count (and density, if *area* is also provided) and use this tolerance value; refer to the *simplification.consolidate_intersections* function documentation for details

Returns

stats –

dictionary containing the following keys

- *circuity_avg* - see *circuity_avg* function documentation
- *clean_intersection_count* - see *clean_intersection_count* function documentation
- *clean_intersection_density_km* - *clean_intersection_count* per sq km
- *edge_density_km* - *edge_length_total* per sq km
- *edge_length_avg* - *edge_length_total / m*
- *edge_length_total* - see *edge_length_total* function documentation
- *intersection_count* - see *intersection_count* function documentation
- *intersection_density_km* - *intersection_count* per sq km
- *k_avg* - graph's average node degree (in-degree and out-degree)
- *m* - count of edges in graph
- *n* - count of nodes in graph
- *node_density_km* - *n* per sq km
- *self_loop_proportion* - see *self_loop_proportion* function documentation
- *street_density_km* - *street_length_total* per sq km
- *street_length_avg* - *street_length_total / street_segment_count*
- *street_length_total* - see *street_length_total* function documentation
- *street_segment_count* - see *street_segment_count* function documentation

- *streets_per_node_avg* - see *streets_per_node_avg* function documentation
- *streets_per_node_counts* - see *streets_per_node_counts* function documentation
- *streets_per_node_proportions* - see *streets_per_node_proportions* function documentation

Return type
dict

osmnx.stats.**circuity_avg**(*Gu*)

Calculate average street circuity using edges of undirected graph.

Circuity is the sum of edge lengths divided by the sum of straight-line distances between edge endpoints. Calculates straight-line distance as euclidean distance if projected or great-circle distance if unprojected.

Parameters
Gu (*networkx.MultiGraph*) – undirected input graph

Returns
circuity_avg – the graph's average undirected edge circuity

Return type
float

osmnx.stats.**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.stats.**edge_length_total**(*G*)

Calculate graph's total edge length.

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

Returns
length – total length (meters) of edges in graph

Return type
float

osmnx.stats.**intersection_count**(*G=None*, *min_streets=2*)

Count the intersections in a graph.

Intersections are defined as nodes with at least *min_streets* number of streets incident on them.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **min_streets** (*int*) – a node must have at least *min_streets* incident on them to count as an intersection

Returns
count – count of intersections in graph

Return type
int

osmnx.stats.**self_loop_proportion**(*Gu*)

Calculate percent of edges that are self-loops in a graph.

A self-loop is defined as an edge from node *u* to node *v* where *u*==*v*.

Parameters
Gu (*networkx.MultiGraph*) – undirected input graph

Returns
proportion – proportion of graph edges that are self-loops

Return type
float

osmnx.stats.**street_length_total**(*Gu*)

Calculate graph's total street segment length.

Parameters
Gu (*networkx.MultiGraph*) – undirected input graph

Returns
length – total length (meters) of streets in graph

Return type
float

osmnx.stats.**street_segment_count**(*Gu*)

Count the street segments in a graph.

Parameters
Gu (*networkx.MultiGraph*) – undirected input graph

Returns
count – count of street segments in graph

Return type
int

osmnx.stats.**streets_per_node**(*G*)

Count streets (undirected edges) incident on each node.

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

Returns
spn – dictionary with node ID keys and street count values

Return type
dict

osmnx.stats.**streets_per_node_avg**(*G*)

Calculate graph's average count of streets per node.

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

Returns
spna – average count of streets per node

Return type
float

osmnx.stats.**streets_per_node_counts**(*G*)

Calculate streets-per-node counts.

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

Returns
spnc – dictionary keyed by count of streets incident on each node, and with values of how many nodes in the graph have this count

Return type
dict

osmnx.stats.**streets_per_node_proportions**(*G*)

Calculate streets-per-node proportions.

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

Returns
spnp – dictionary keyed by count of streets incident on each node, and with values of what proportion of nodes in the graph have this count

Return type
dict

6.3.15 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

6.3.16 osmnx.utils module

General utility functions.

osmnx.utils.**citation**(*style='bibtex'*)

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

Parameters
style (*string {"apa", "bibtex", "ieee"}*) – citation format, either APA or BibTeX or IEEE

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)'*, *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='https://overpass-api.de/api'*, *overpass_rate_limit=True*, *overpass_settings='[out:json][timeout:{timeout}]{maxsize}'*, *requests_kwargs={}*, *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']*)

Do not use: deprecated. Use the settings module directly.

Parameters

- **all_oneway** (*bool*) – deprecated
- **bidirectional_network_types** (*list*) – deprecated
- **cache_folder** (*string or pathlib.Path*) – deprecated
- **data_folder** (*string or pathlib.Path*) – deprecated
- **cache_only_mode** (*bool*) – deprecated
- **default_accept_language** (*string*) – deprecated
- **default_access** (*string*) – deprecated

- **default_crs** (*string*) – deprecated
- **default_referer** (*string*) – deprecated
- **default_user_agent** (*string*) – deprecated
- **imgs_folder** (*string or pathlib.Path*) – deprecated
- **log_file** (*bool*) – deprecated
- **log_filename** (*string*) – deprecated
- **log_console** (*bool*) – deprecated
- **log_level** (*int*) – deprecated
- **log_name** (*string*) – deprecated
- **logs_folder** (*string or pathlib.Path*) – deprecated
- **max_query_area_size** (*int*) – deprecated
- **memory** (*int*) – deprecated
- **nominatim_endpoint** (*string*) – deprecated
- **nominatim_key** (*string*) – deprecated
- **osm_xml_node_attrs** (*list*) – deprecated
- **osm_xml_node_tags** (*list*) – deprecated
- **osm_xml_way_attrs** (*list*) – deprecated
- **osm_xml_way_tags** (*list*) – deprecated
- **overpass_endpoint** (*string*) – deprecated
- **overpass_rate_limit** (*bool*) – deprecated
- **overpass_settings** (*string*) – deprecated
- **requests_kwargs** (*dict*) – deprecated
- **timeout** (*int*) – deprecated
- **use_cache** (*bool*) – deprecated
- **useful_tags_node** (*list*) – deprecated
- **useful_tags_way** (*list*) – deprecated

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

Return type
None

osmnx.utils.**ts**(*style='datetime'*, *template=None*)

Return current local timestamp as a string.

Parameters

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

Returns
ts – local timestamp string

Return type
string

6.3.17 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, lon) 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, lon) 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

Return type
shapely.geometry.Polygon

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

Interpolate evenly spaced points along a LineString.

The spacing is approximate because the LineString's length may not be evenly divisible by it.

Parameters

- **geom** (*shapely.geometry.LineString*) – a LineString geometry
- **dist** (*float*) – spacing distance between interpolated points, in same units as *geom*. smaller values generate more points.

Yields

point (*tuple of floats*) – a generator of (x, y) tuples of the interpolated points' coordinates

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

Do not use: deprecated.

Parameters

- **geom** (*shapely.geometry.geometry {Point, MultiPoint, LineString, MultiLineString, Polygon, MultiPolygon}*) – deprecated, do not use
- **precision** (*int*) – deprecated, do not use

Return type

shapely.geometry.geometry

osmnx.utils_geo.**sample_points**(*G*, *n*)

Randomly sample points constrained to a spatial graph.

This generates a graph-constrained uniform random sample of points. Unlike typical spatially uniform random sampling, this method accounts for the graph's geometry. And unlike equal-length edge segmenting, this method guarantees uniform randomness.

Parameters

- **G** (*networkx.MultiGraph*) – graph to sample points from; should be undirected (to not oversample bidirectional edges) and projected (for accurate point interpolation)
- **n** (*int*) – how many points to sample

Returns

points – the sampled points, multi-indexed by (u, v, key) of the edge from which each point was drawn

Return type

geopandas.GeoSeries

6.3.18 osmnx.utils_graph module

Graph utility functions.

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

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*)

Do not use: deprecated.

Use the *route_to_gdf* function instead.

Parameters

- **G** (*networkx.MultiDiGraph*) – deprecated
- **route** (*list*) – deprecated
- **attribute** (*string*) – deprecated
- **minimize_key** (*string*) – deprecated
- **retrieve_default** (*Callable[Tuple[Any, Any], Any]*) – deprecated

Returns
attribute_values – deprecated

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

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 1) *gdf_nodes* is uniquely indexed by *osmid*, 2) *gdf_nodes* contains *x* and *y* coordinate columns representing node geometries, 3) *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. Note that any *geometry* attribute on *gdf_nodes* is discarded since *x* and *y* provide the necessary node geometry information instead.

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 attributes
- **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). gdf_nodes is indexed by osmid and gdf_edges is multi-indexed by u, v, key following normal MultiDiGraph structure.

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

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

Return a GeoDataFrame of the edges in a path, in order.

Parameters

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

- **route** (*list*) – list of node IDs constituting the path
- **weight** (*string*) – if there are parallel edges between two nodes, choose lowest weight

Returns
gdf_edges – GeoDataFrame of the edges

Return type
geopandas.GeoDataFrame

6.4 Internals Reference

This is the complete OSMnx internals reference for developers, including private internal modules and functions. If you are instead looking for a user guide to OSMnx's public API, see the *User Reference*.

6.4.1 osmnx._api module

Expose most common parts of public API directly in package namespace.

6.4.2 osmnx.bearing module

Calculate graph edge bearings.

osmnx.bearing.**_bearings_distribution**(*Gu*, *num_bins*, *min_length=0*, *weight=None*)

Compute distribution of bearings across evenly spaced bins.

Prevents bin-edge effects around common values like 0 degrees and 90 degrees by initially creating twice as many bins as desired, then merging them in pairs. For example, if *num_bins=36* is provided, then each bin will represent 10 degrees around the compass, with the first bin representing 355 degrees to 5 degrees.

Parameters

- **Gu** (*networkx.MultiGraph*) – undirected, unprojected graph with *bearing* attributes on each edge
- **num_bins** (*int*) – number of bins for the bearings histogram
- **min_length** (*float*) – ignore edges with *length* attributes less than *min_length*; useful to ignore the noise of many very short edges
- **weight** (*string*) – if not None, weight edges' bearings by this (non-null) edge attribute. for example, if "length" is provided, this will return 1 bearing observation per meter per street, which could result in a very large *bearings* array.

Returns
bin_counts, bin_edges – counts of bearings per bin and the bins edges

Return type
tuple of numpy.array

osmnx.bearing.**_extract_edge_bearings**(*Gu*, *min_length=0*, *weight=None*)

Extract undirected graph's bidirectional edge bearings.

For example, if an edge has a bearing of 90 degrees then we will record bearings of both 90 degrees and 270 degrees for this edge.

Parameters

- **Gu** (*networkx.MultiGraph*) – undirected, unprojected graph with *bearing* attributes on each edge
- **min_length** (*float*) – ignore edges with *length* attributes less than *min_length*; useful to ignore the noise of many very short edges
- **weight** (*string*) – if not None, weight edges' bearings by this (non-null) edge attribute. for example, if "length" is provided, this will return 1 bearing observation per meter per street, which could result in a very large *bearings* array.

Returns
bearings – the graph's bidirectional edge bearings

Return type
numpy.array

osmnx.bearing.**add_edge_bearings**(*G*, *precision=None*)

Add compass *bearing* attributes to all graph edges.

Vectorized function to calculate (initial) bearing from origin node to destination node for each edge in a directed, unprojected graph then add these bearings as new edge attributes. Bearing represents angle in degrees (clockwise) between north and the geodesic line from the origin node to the destination node. Ignores self-loop edges as their bearings are undefined.

Parameters

- **G** (*networkx.MultiDiGraph*) – unprojected graph
- **precision** (*int*) – deprecated, do not use

Returns
G – graph with edge bearing attributes

Return type
networkx.MultiDiGraph

osmnx.bearing.**calculate_bearing**(*lat1*, *lon1*, *lat2*, *lon2*)

Calculate the compass bearing(s) between pairs of lat-lon points.

Vectorized function to calculate initial bearings between two points' coordinates or between arrays of points' coordinates. Expects coordinates in decimal degrees. Bearing represents the clockwise angle in degrees between north and the geodesic line from (lat1, lon1) to (lat2, lon2).

Parameters

- **lat1** (*float or numpy.array of float*) – first point's latitude coordinate
- **lon1** (*float or numpy.array of float*) – first point's longitude coordinate
- **lat2** (*float or numpy.array of float*) – second point's latitude coordinate
- **lon2** (*float or numpy.array of float*) – second point's longitude coordinate

Returns
bearing – the bearing(s) in decimal degrees

Return type
float or numpy.array of float

osmnx.bearing.**orientation_entropy**(*Gu*, *num_bins=36*, *min_length=0*, *weight=None*)

Calculate undirected graph's orientation entropy.

Orientation entropy is the entropy of its edges' bidirectional bearings across evenly spaced bins. Ignores self-loop edges as their bearings are undefined.

For more info see: Boeing, G. 2019. "Urban Spatial Order: Street Network Orientation, Configuration, and Entropy." Applied Network Science, 4 (1), 67. https://doi.org/10.1007/s41109-019-0189-1

Parameters

- **Gu** (*networkx.MultiGraph*) – undirected, unprojected graph with *bearing* attributes on each edge
- **num_bins** (*int*) – number of bins; for example, if *num_bins=36* is provided, then each bin will represent 10 degrees around the compass
- **min_length** (*float*) – ignore edges with *length* attributes less than *min_length*; useful to ignore the noise of many very short edges
- **weight** (*string*) – if not None, weight edges' bearings by this (non-null) edge attribute. for example, if "length" is provided, this will return 1 bearing observation per meter per street, which could result in a very large *bearings* array.

Returns

entropy – the graph's orientation entropy

Return type

float

osmnx.bearing.**plot_orientation**(*Gu*, *num_bins=36*, *min_length=0*, *weight=None*, *ax=None*, *figsize=(5, 5)*, *area=True*, *color='#003366'*, *edgecolor='k'*, *linewidth=0.5*, *alpha=0.7*, *title=None*, *title_y=1.05*, *title_font=None*, *xtick_font=None*)

Do not use: deprecated.

The plot_orientation function moved to the plot module. Calling it via the bearing module will raise an error in a future release.

Parameters

- **Gu** (*networkx.MultiGraph*) – deprecated, do not use
- **num_bins** (*int*) – deprecated, do not use
- **min_length** (*float*) – deprecated, do not use
- **weight** (*string*) – deprecated, do not use
- **ax** (*matplotlib.axes.PolarAxesSubplot*) – deprecated, do not use
- **figsize** (*tuple*) – deprecated, do not use
- **area** (*bool*) – deprecated, do not use
- **color** (*string*) – deprecated, do not use
- **edgecolor** (*string*) – deprecated, do not use
- **linewidth** (*float*) – deprecated, do not use
- **alpha** (*float*) – deprecated, do not use
- **title** (*string*) – deprecated, do not use
- **title_y** (*float*) – deprecated, do not use
- **title_font** (*dict*) – deprecated, do not use
- **xtick_font** (*dict*) – deprecated, do not use

Returns

fig, ax – matplotlib figure, axis

Return type
tuple

6.4.3 osmnx.distance module

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

osmnx.distance.**add_edge_lengths**(*G*, *precision=None*, *edges=None*)

Add *length* attribute (in meters) to each edge.

Vectorized function to calculate great-circle distance between each edge's incident nodes. Ensure graph is in unprojected coordinates, and 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. It calculates edge lengths as the great-circle distance from node *u* to node *v*. When OSMnx automatically runs this function upon graph creation, it does it before simplifying the graph: thus it calculates the straight-line lengths of edge segments that are themselves all straight. Only after simplification do edges take on a (potentially) curvilinear geometry. If you wish to calculate edge lengths later, you are calculating straight-line distances which necessarily ignore the curvilinear geometry. You only want to run this function on a graph with all straight edges (such as is the case with an unsimplified graph).

Parameters

- **G** (*networkx.MultiDiGraph*) – unprojected, unsimplified input graph
- **precision** (*int*) – deprecated, do not use
- **edges** (*tuple*) – tuple of (u, v, k) tuples representing subset of edges to add length attributes to. if None, add lengths to all edges.

Returns
G – graph with edge length attributes

Return type
networkx.MultiDiGraph

osmnx.distance.**euclidean**(*y1*, *x1*, *y2*, *x2*)

Calculate Euclidean distances between pairs of points.

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

Parameters

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

Returns
dist – distance from each (x1, y1) to each (x2, y2) in coordinates' units

Return type
float or numpy.array of float

osmnx.distance.**euclidean_dist_vec**(*y1*, *x1*, *y2*, *x2*)

Do not use, deprecated.

The *euclidean_dist_vec* function has been renamed *euclidean*. Calling *euclidean_dist_vec* will raise an error in a future release.

Parameters

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

Returns
dist – distance from each (x1, y1) to each (x2, y2) in coordinates' units

Return type
float or numpy.array of float

osmnx.distance.**great_circle**(*lat1*, *lon1*, *lat2*, *lon2*, *earth_radius=6371009*)

Calculate great-circle distances between pairs of 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 numpy.array of float*) – first point's latitude coordinate
- **lon1** (*float or numpy.array of float*) – first point's longitude coordinate
- **lat2** (*float or numpy.array of float*) – second point's latitude coordinate
- **lon2** (*float or numpy.array of float*) – second point's longitude coordinate
- **earth_radius** (*float*) – earth's radius in units in which distance will be returned (default is meters)

Returns
dist – distance from each (lat1, lon1) to each (lat2, lon2) in units of earth_radius

Return type
float or numpy.array of float

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

Do not use, deprecated.

The *great_circle_vec* function has been renamed *great_circle*. Calling *great_circle_vec* will raise an error in a future release.

Parameters

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

Returns
dist – distance from each (lat1, lng1) to each (lat2, lng2) in units of earth_radius

Return type
float or numpy.array of float

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

Do not use, deprecated.

The *k_shortest_paths* function has moved to the *routing* module. Calling it via the *distance* module will raise an error in a future release.

Parameters

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

Yields

path (*list*) – a generator of *k* shortest paths ordered by total weight. each path is a list of node IDs.

osmnx.distance.**nearest_edges**(*G*, *X*, *Y*, *interpolate=None*, *return_dist=False*)

Find the nearest edge to a point or to each of several points.

If *X* and *Y* are single coordinate values, this will return the nearest edge to that point. If *X* and *Y* are lists of coordinate values, this will return the nearest edge to each point. This function uses an R-tree spatial index and minimizes the euclidean distance from each point to the possible matches. For accurate results, use a projected graph and points.

Parameters

- **G** (*networkx.MultiDiGraph*) – graph in which to find nearest edges
- **X** (*float or list*) – points' x (longitude) coordinates, in same CRS/units as graph and containing no nulls
- **Y** (*float or list*) – points' y (latitude) coordinates, in same CRS/units as graph and containing no nulls
- **interpolate** (*float*) – deprecated, do not use
- **return_dist** (*bool*) – optionally also return distance between points and nearest edges

Returns

ne or (ne, dist) – nearest edges as (u, v, key) or optionally a tuple where *dist* contains distances between the points and their nearest edges

Return type

tuple or list

osmnx.distance.**nearest_nodes**(*G*, *X*, *Y*, *return_dist=False*)

Find the nearest node to a point or to each of several points.

If *X* and *Y* are single coordinate values, this will return the nearest node to that point. If *X* and *Y* are lists of coordinate values, this will return the nearest node to each point.

If the graph is projected, this uses a k-d tree for euclidean nearest neighbor search, which requires that scipy is installed as an optional dependency. If it is unprojected, this uses a ball tree for haversine nearest neighbor search, which requires that scikit-learn is installed as an optional dependency.

Parameters

- **G** (*networkx.MultiDiGraph*) – graph in which to find nearest nodes

- **X** (*float or list*) – points' x (longitude) coordinates, in same CRS/units as graph and containing no nulls
- **Y** (*float or list*) – points' y (latitude) coordinates, in same CRS/units as graph and containing no nulls
- **return_dist** (*bool*) – optionally also return distance between points and nearest nodes

Returns
nn or (nn, dist) – nearest node IDs or optionally a tuple where *dist* contains distances between the points and their nearest nodes

Return type
int/list or tuple

osmnx.distance.**shortest_path**(*G*, *orig*, *dest*, *weight='length'*, *cpus=1*)

Do not use, deprecated.

The *shortest_path* function has moved to the *routing* module. Calling it via the *distance* module will raise an error in a future release.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **orig** (*int or list*) – origin node ID, or a list of origin node IDs
- **dest** (*int or list*) – destination node ID, or a list of destination node IDs
- **weight** (*string*) – edge attribute to minimize when solving shortest path
- **cpus** (*int*) – how many CPU cores to use; if None, use all available

Returns
path – list of node IDs constituting the shortest path, or, if orig and dest are lists, then a list of path lists

Return type
list

6.4.4 osmnx._downloader module

Handle HTTP requests to web APIs.

osmnx._downloader.**_config_dns**(*url*)

Force socket.getaddrinfo to use IP address instead of hostname.

Resolves the URL's domain to an IP address so that we use the same server for both 1) checking the necessary pause duration and 2) sending the query itself even if there is round-robin redirecting among multiple server machines on the server-side. Mutates the getaddrinfo function so it uses the same IP address everytime it finds the hostname in the URL.

For example, the server overpass-api.de just redirects to one of the other servers (currently gall.openstreetmap.de and lambert.openstreetmap.de). So if we check the status endpoint of overpass-api.de, we may see results for server gall, but when we submit the query itself it gets redirected to server lambert. This could result in violating server lambert's slot management timing.

Parameters
url (*string*) – the URL to consistently resolve the IP address of

Return type
None

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._hostname_from_url(*url*)

Extract the hostname (domain) from a URL.

Parameters
url (*string*) – the url from which to extract the hostname

Returns
hostname – the extracted hostname (domain)

Return type
string

osmnx._downloader._parse_response(*response*)

Parse JSON from a requests response and log the details.

Parameters
response (*requests.response*) – the response object

Returns
response_json

Return type
dict

osmnx._downloader._resolve_host_via_doh(*hostname*)

Resolve hostname to IP address via Google's public DNS-over-HTTPS API.

Necessary fallback as socket.gethostbyname will not always work when using a proxy. See https://developers.google.com/speed/public-dns/docs/doh/json If the user has set *settings.doh_url_template=None* or if resolution fails (e.g., due to local network blocking DNS-over-HTTPS) the hostname itself will be returned instead. Note that this means that server slot management may be violated: see *_config_dns* documentation for details.

Parameters
hostname (*string*) – the hostname to consistently resolve the IP address of

Returns
ip_address – resolved IP address of host, or hostname itself if resolution failed

Return type
string

osmnx._downloader._retrieve_from_cache(*url*, *check_remark=True*)

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*, *ok*)

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 ok is True.

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
- **ok** (*bool*) – requests response.ok value

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

6.4.5 osmnx.elevation module

Add node elevations from raster files or web APIs, and calculate edge grades.

osmnx.elevation._elevation_request(*url*, *pause*)

Send a HTTP GET request to Google Maps-style Elevation API.

Parameters

- **url** (*string*) – URL for API endpoint populated with request data
- **pause** (*float*) – how long to pause in seconds before request

Returns
response_json

Return type
dict

osmnx.elevation.**_query_raster**(*nodes*, *filepath*, *band*)

Query a raster for values at coordinates in a DataFrame's x/y columns.

Parameters

- **nodes** (*pandas.DataFrame*) – DataFrame indexed by node ID and with two columns: x and y
- **filepath** (*string or pathlib.Path*) – path to the raster file or VRT to query
- **band** (*int*) – which raster band to query

Returns
nodes_values – zipped node IDs and corresponding raster values

Return type
zip

osmnx.elevation.**add_edge_grades**(*G*, *add_absolute=True*, *precision=None*)

Add *grade* attribute to each graph edge.

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

See also the *add_node_elevations_raster* and *add_node_elevations_google* functions.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph with *elevation* node attribute
- **add_absolute** (*bool*) – if True, also add absolute value of grade as *grade_abs* attribute
- **precision** (*int*) – deprecated, do not use

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

Return type
networkx.MultiDiGraph

osmnx.elevation.**add_node_elevations_google**(*G*, *api_key=None*, *batch_size=350*, *pause=0*, *max_locations_per_batch=None*, *precision=None*, *url_template=None*)

Add an *elevation* (meters) attribute to each node using a web service.

By default, this uses the Google Maps Elevation API but you can optionally use an equivalent API with the same interface and response format, such as Open Topo Data, via the *settings* module's *elevation_url_template*. The Google Maps Elevation API requires an API key but other providers may not.

For a free local alternative see the *add_node_elevations_raster* function. See also the *add_edge_grades* function.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **api_key** (*string*) – a valid API key, can be None if the API does not require a key
- **batch_size** (*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 allowed)
- **pause** (*float*) – time to pause between API calls, which can be increased if you get rate limited

- **max_locations_per_batch** (*int*) – deprecated, do not use
- **precision** (*int*) – deprecated, do not use
- **url_template** (*string*) – deprecated, do not use

Returns
G – graph with node elevation attributes

Return type
networkx.MultiDiGraph

osmnx.elevation.**add_node_elevations_raster**(*G*, *filepath*, *band=1*, *cpus=None*)

Add *elevation* attribute to each node from local raster file(s).

If *filepath* is a list of paths, this will generate a virtual raster composed of the files at those paths as an intermediate step.

See also the *add_edge_grades* function.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph, in same CRS as raster
- **filepath** (*string or pathlib.Path or list of strings/Paths*) – path (or list of paths) to the raster file(s) to query
- **band** (*int*) – which raster band to query
- **cpus** (*int*) – how many CPU cores to use; if None, use all available

Returns
G – graph with node elevation attributes

Return type
networkx.MultiDiGraph

6.4.6 osmnx._errors module

Define custom errors and exceptions.

exception osmnx._errors.**CacheOnlyInterruptError**(**args*, ***kwargs*)

Exception for settings.cache_only_mode=True interruption.

exception osmnx._errors.**GraphSimplificationError**(**args*, ***kwargs*)

Exception for a problem with graph simplification.

exception osmnx._errors.**InsufficientResponseError**(**args*, ***kwargs*)

Exception for empty or too few results in server response.

exception osmnx._errors.**ResponseStatusCodeError**(**args*, ***kwargs*)

Exception for an unhandled server response status code.

6.4.7 osmnx.features module

Download OpenStreetMap geospatial features' geometries and attributes.

Retrieve points of interest, building footprints, transit lines/stops, or any other map features from OSM, including their geometries and attribute data, then construct a GeoDataFrame of them. You can use this module to query for nodes, ways, and relations (the latter of type "multipolygon" or "boundary" only) by passing a dictionary of desired OSM tags.

For more details, see https://wiki.openstreetmap.org/wiki/Map_features and https://wiki.openstreetmap.org/wiki/Elements

Refer to the Getting Started guide for usage limitations.

`osmnx.features._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.features._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.features._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 features 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 features and their associated tags

Return type
geopandas.GeoDataFrame

osmnx.features.**_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.features.**_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 blocklist/passlist

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

Return type
bool

osmnx.features.**_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.features.**_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.features.**_parse_relation_to_multipolygon**(*element*, *geometries*)

Parse multipolygon from OSM relation (type:MultiPolygon).

See more information about relations from OSM documentation: https://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.features.**_parse_way_to_linestring_or_polygon**(*element*, *coords*)

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

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

Return type
dict

osmnx.features.**_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.features.**features_from_address**(*address*, *tags*, *dist=1000*)

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

You can use the *settings* module to retrieve a snapshot of historical OSM data as of a certain date, or to configure the Overpass server timeout, memory allocation, and other custom settings.

For more details, see: https://wiki.openstreetmap.org/wiki/Map_features

Parameters

- **address** (*string*) – the address to geocode and use as the central point around which to get the features
- **tags** (*dict*) – Dict of tags used for finding elements 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

osmnx.features.**features_from_bbox**(*north*, *south*, *east*, *west*, *tags*)

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

You can use the *settings* module to retrieve a snapshot of historical OSM data as of a certain date, or to configure the Overpass server timeout, memory allocation, and other custom settings.

For more details, see: https://wiki.openstreetmap.org/wiki/Map_features

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 elements 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

osmnx.features.**features_from_place**(*query*, *tags*, *which_result=None*, *buffer_dist=None*)

Create GeoDataFrame of OSM features within boundaries of some 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 features within it using the *features_from_address* function, which geocodes the place name to a point and gets the features 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 *features_from_polygon* function.

You can use the *settings* module to retrieve a snapshot of historical OSM data as of a certain date, or to configure the Overpass server timeout, memory allocation, and other custom settings.

For more details, see: https://wiki.openstreetmap.org/wiki/Map_features

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 elements 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*) – deprecated, do not use

Returns
gdf

Return type
geopandas.GeoDataFrame

osmnx.features.**features_from_point**(*center_point*, *tags*, *dist=1000*)

Create GeoDataFrame of OSM features within some distance N, S, E, W of a point.

You can use the *settings* module to retrieve a snapshot of historical OSM data as of a certain date, or to configure the Overpass server timeout, memory allocation, and other custom settings.

For more details, see: https://wiki.openstreetmap.org/wiki/Map_features

Parameters

- **center_point** (*tuple*) – the (lat, lon) center point around which to get the features
- **tags** (*dict*) – Dict of tags used for finding elements 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

osmnx.features.**features_from_polygon**(*polygon*, *tags*)

Create GeoDataFrame of OSM features within boundaries of a (multi)polygon.

You can use the *settings* module to retrieve a snapshot of historical OSM data as of a certain date, or to configure the Overpass server timeout, memory allocation, and other custom settings.

For more details, see: https://wiki.openstreetmap.org/wiki/Map_features

Parameters

- **polygon** (*shapely.geometry.Polygon or shapely.geometry.MultiPolygon*) – geographic boundaries to fetch features within
- **tags** (*dict*) – Dict of tags used for finding elements 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

osmnx.features.**features_from_xml**(*filepath*, *polygon=None*, *tags=None*, *encoding='utf-8'*)

Create a GeoDataFrame of OSM features in an OSM-formatted XML file.

Because this function creates a GeoDataFrame of features 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, features_from_xml() will return features for all of the tagged elements in the file. If they are supplied they will be used to filter the final GeoDataFrame.

For more details, see: https://wiki.openstreetmap.org/wiki/Map_features

Parameters

- **filepath** (*string or pathlib.Path*) – path to file containing OSM XML data
- **polygon** (*shapely.geometry.Polygon*) – optional geographic boundary to filter elements
- **tags** (*dict*) – optional dict of tags for filtering elements 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.
- **encoding** (*string*) – the XML file's character encoding

Returns
gdf

Return type
geopandas.GeoDataFrame

6.4.8 osmnx.folium module

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

This module is deprecated. Do not use. It will be removed in a future release. You can generate and explore interactive web maps of graph nodes, edges, and/or routes automatically using GeoPandas.GeoDataFrame.explore instead, for example like: *ox.graph_to_gdfs(G, nodes=False).explore()*. See the OSMnx examples gallery for complete details and demonstrations.

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*, ***kwargs*)

Do not use: deprecated.

You can generate and explore interactive web maps of graph nodes, edges, and/or routes automatically using GeoPandas.GeoDataFrame.explore instead, for example like: *ox.graph_to_gdfs(G, nodes=False).explore()*. See the OSMnx examples gallery for complete details and demonstrations.

Parameters

- **G** (*networkx.MultiDiGraph*) – deprecated
- **graph_map** (*folium.folium.Map*) – deprecated
- **popup_attribute** (*string*) – deprecated
- **tiles** (*string*) – deprecated
- **zoom** (*int*) – deprecated
- **fit_bounds** (*bool*) – deprecated
- **kwargs** – deprecated

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*, ***kwargs*)

Do not use: deprecated.

You can generate and explore interactive web maps of graph nodes, edges, and/or routes automatically using GeoPandas.GeoDataFrame.explore instead, for example like: *ox.graph_to_gdfs(G, nodes=False).explore()*. See the OSMnx examples gallery for complete details and demonstrations.

Parameters

- **G** (*networkx.MultiDiGraph*) – deprecated
- **route** (*list*) – deprecated
- **route_map** (*folium.folium.Map*) – deprecated
- **popup_attribute** (*string*) – deprecated
- **tiles** (*string*) – deprecated
- **zoom** (*int*) – deprecated
- **fit_bounds** (*bool*) – deprecated
- **kwargs** – deprecated

Return type
folium.folium.Map

6.4.9 osmnx.geocoder module

Geocode place names or addresses or retrieve OSM elements by place name or ID.

This module uses the Nominatim API's "search" and "lookup" endpoints. For more details see https://wiki.openstreetmap.org/wiki/Elements and https://nominatim.org/.

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. ignored if by_osmid=True.
- **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 place names or addresses to (lat, lon) with the Nominatim API.

This geocodes the query via the Nominatim "search" endpoint.

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

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

Return type
tuple

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

Retrieve OSM elements by place name or OSM ID with the Nominatim API.

If searching by place name, the *query* argument can be a string or structured dict, or a list of such strings/dicts to send to the geocoder. This uses the Nominatim "search" endpoint to geocode the place name to the best-matching OSM element, then returns that element and its attribute data.

You can instead query by OSM ID by passing *by_osmid=True*. This uses the Nominatim "lookup" endpoint to retrieve the OSM element with that ID. In this case, the function treats the *query* argument as an OSM ID (or list of OSM IDs), which must be prepended with their types: node (N), way (W), or relation (R) in accordance with the Nominatim API format. For example, *query=["R2192363", "N240109189", "W427818536"]*.

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

Parameters

- **query** (*string or dict or list of strings/dicts*) – query string(s) or structured dict(s) to geocode
- **which_result** (*int*) – which search result to return. 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. ignored if by_osmid=True.
- **by_osmid** (*bool*) – if True, treat query as an OSM ID lookup rather than text search
- **buffer_dist** (*float*) – deprecated, do not use

Returns
gdf – a GeoDataFrame with one row for each query

Return type
geopandas.GeoDataFrame

6.4.10 osmnx.geometries module

Do not use: deprecated.

The *geometries* module has been renamed the *features* module. The *geometries* module is deprecated and will be removed in a future release.

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

Do not use: deprecated.

The *geometries* module and *geometries_from_X* functions have been renamed the *features* module and *features_from_X* functions. Use these instead. The *geometries* module and functions are deprecated and will be removed in a future release.

Parameters

- **address** (*string*) – Do not use: deprecated.

- **tags** (*dict*) – Do not use: deprecated.
- **dist** (*numeric*) – Do not use: deprecated.

Returns
gdf

Return type
geopandas.GeoDataFrame

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

Do not use: deprecated.

The *geometries* module and *geometries_from_X* functions have been renamed the *features* module and *features_from_X* functions. Use these instead. The *geometries* module and functions are deprecated and will be removed in a future release.

Parameters

- **north** (*float*) – Do not use: deprecated.
- **south** (*float*) – Do not use: deprecated.
- **east** (*float*) – Do not use: deprecated.
- **west** (*float*) – Do not use: deprecated.
- **tags** (*dict*) – Do not use: deprecated.

Returns
gdf

Return type
geopandas.GeoDataFrame

osmnx.geometries.**geometries_from_place**(*query*, *tags*, *which_result=None*, *buffer_dist=None*)

Do not use: deprecated.

The *geometries* module and *geometries_from_X* functions have been renamed the *features* module and *features_from_X* functions. Use these instead. The *geometries* module and functions are deprecated and will be removed in a future release.

Parameters

- **query** (*string or dict or list*) – Do not use: deprecated.
- **tags** (*dict*) – Do not use: deprecated.
- **which_result** (*int*) – Do not use: deprecated.
- **buffer_dist** (*float*) – Do not use: deprecated.

Returns
gdf

Return type
geopandas.GeoDataFrame

osmnx.geometries.**geometries_from_point**(*center_point*, *tags*, *dist=1000*)

Do not use: deprecated.

The *geometries* module and *geometries_from_X* functions have been renamed the *features* module and *features_from_X* functions. Use these instead. The *geometries* module and functions are deprecated and will be removed in a future release.

Parameters

- **center_point** (*tuple*) – Do not use: deprecated.
- **tags** (*dict*) – Do not use: deprecated.
- **dist** (*numeric*) – Do not use: deprecated.

Returns
gdf

Return type
geopandas.GeoDataFrame

osmnx.geometries.**geometries_from_polygon**(*polygon*, *tags*)

Do not use: deprecated.

The *geometries* module and *geometries_from_X* functions have been renamed the *features* module and *features_from_X* functions. Use these instead. The *geometries* module and functions are deprecated and will be removed in a future release.

Parameters

- **polygon** (*shapely.geometry.Polygon or shapely.geometry.MultiPolygon*) – Do not use: deprecated.
- **tags** (*dict*) – Do not use: deprecated.

Returns
gdf

Return type
geopandas.GeoDataFrame

osmnx.geometries.**geometries_from_xml**(*filepath*, *polygon=None*, *tags=None*)

Do not use: deprecated.

The *geometries* module and *geometries_from_X* functions have been renamed the *features* module and *features_from_X* functions. Use these instead. The *geometries* module and functions are deprecated and will be removed in a future release.

Parameters

- **filepath** (*string or pathlib.Path*) – Do not use: deprecated.
- **polygon** (*shapely.geometry.Polygon*) – Do not use: deprecated.
- **tags** (*dict*) – Do not use: deprecated.

Returns
gdf

Return type
geopandas.GeoDataFrame

6.4.11 osmnx.graph module

Download and create graphs from OpenStreetMap data.

This module uses filters to query the Overpass API: you can either specify a built-in network type or provide your own custom filter with Overpass QL.

Refer to the Getting Started guide for usage limitations.

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

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 *distance.add_edge_lengths* function.

Parameters

- **response_jsons** (*iterable*) – iterable 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

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

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=None*, *custom_filter=None*)

Download and create a graph within some distance of an address.

You can use the *settings* module to retrieve a snapshot of historical OSM data as of a certain date, or to configure the Overpass server timeout, memory allocation, and other custom settings.

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*) – deprecated, do not use
- **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.

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

Notes

Very large query areas use the *utils_geo._consolidate_subdivide_geometry* function to automatically make multiple requests: see that function's documentation for caveats.

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

Download and create a graph within some bounding box.

You can use the *settings* module to retrieve a snapshot of historical OSM data as of a certain date, or to configure the Overpass server timeout, memory allocation, and other custom settings.

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*) – deprecated, do not use
- **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

Very large query areas use the *utils_geo._consolidate_subdivide_geometry* function to automatically make multiple requests: see that function's documentation for caveats.

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=None*, *custom_filter=None*)

Download and create a graph within the boundaries of some 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.

You can use the *settings* module to retrieve a snapshot of historical OSM data as of a certain date, or to configure the Overpass server timeout, memory allocation, and other custom settings.

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*) – deprecated, do not use
- **clean_periphery** (*bool*) – deprecated, do not use
- **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

Very large query areas use the *utils_geo._consolidate_subdivide_geometry* function to automatically make multiple requests: see that function's documentation for caveats.

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=None*, *custom_filter=None*)

Download and create a graph within some distance of a (lat, lon) point.

You can use the *settings* module to retrieve a snapshot of historical OSM data as of a certain date, or to configure the Overpass server timeout, memory allocation, and other custom settings.

Parameters

- **center_point** (*tuple*) – the (lat, lon) 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,*) – deprecated, do not use
- **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

Very large query areas use the *utils_geo._consolidate_subdivide_geometry* function to automatically make multiple requests: see that function's documentation for caveats.

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

Download and create a graph within the boundaries of a (multi)polygon.

You can use the *settings* module to retrieve a snapshot of historical OSM data as of a certain date, or to configure the Overpass server timeout, memory allocation, and other custom settings.

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*) – deprecated, do not use
- **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

Very large query areas use the *utils_geo._consolidate_subdivide_geometry* function to automatically make multiple requests: see that function's documentation for caveats.

osmnx.graph.**graph_from_xml**(*filepath*, *bidirectional=False*, *simplify=True*, *retain_all=False*, *encoding='utf-8'*)

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

Do not load an XML file generated by OSMnx: this use case is not supported and may not behave as expected. To save/load graphs to/from disk for later use in OSMnx, use the *io.save_graphml* and *io.load_graphml* functions instead.

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.
- **encoding** (*string*) – the XML file's character encoding

Returns
G

Return type
networkx.MultiDiGraph

6.4.12 osmnx.io module

Serialize graphs to/from files on disk.

osmnx.io.**_convert_bool_string**(*value*)

Convert a "True" or "False" string literal to corresponding boolean type.

This is necessary because Python will otherwise parse the string "False" to the boolean value True, that is, *bool("False") == True*. This function raises a ValueError if a value other than "True" or "False" is passed.

If the value is already a boolean, this function just returns it, to accommodate usage when the value was originally inside a stringified list.

Parameters
value (*string {"True", "False"}*) – the value to convert

Return type
bool

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_graph_attr_types**(*G*, *dtypes=None*)

Convert graph-level attributes using a dict of data types.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **dtypes** (*dict*) – dict of graph-level 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=None*, *graphml_str=None*, *node_dtypes=None*, *edge_dtypes=None*, *graph_dtypes=None*)

Load an OSMnx-saved GraphML file from disk or GraphML string.

This function converts node, edge, and graph-level attributes (serialized as strings) to their appropriate data types. These can be customized as needed by passing in dtypes arguments providing types or custom converter functions. For example, if you want to convert some attribute's values to *bool*, consider using the built-in *ox.io._convert_bool_string* function to properly handle "True"/"False" string literals as True/False booleans: *ox.load_graphml(fp, node_dtypes={my_attr: ox.io._convert_bool_string})*.

If you manually configured the *all_oneway=True* setting, you may need to manually specify here that edge *oneway* attributes should be type *str*.

Note that you must pass one and only one of *filepath* or *graphml_str*. If passing *graphml_str*, you may need to decode the bytes read from your file before converting to string to pass to this function.

Parameters

- **filepath** (*string or pathlib.Path*) – path to the GraphML file
- **graphml_str** (*string*) – a valid and decoded string representation of a GraphML file's contents
- **node_dtypes** (*dict*) – dict of node attribute names:types to convert values' data types. the type can be a python type or a custom string converter function.
- **edge_dtypes** (*dict*) – dict of edge attribute names:types to convert values' data types. the type can be a python type or a custom string converter function.
- **graph_dtypes** (*dict*) – dict of graph-level attribute names:types to convert values' data types. the type can be a python type or a custom string converter function.

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

Return type
None

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

Do not use: deprecated. Use the save_graph_geopackage function instead.

The Shapefile format is proprietary and outdated. Instead, use the superior GeoPackage file format via the save_graph_geopackage function. See http://switchfromshapefile.org/ for more information.

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

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*, *api_version=0.6*, *precision=6*)

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. As such, this function has a limited use case which does not include saving/loading graphs for subsequent OSMnx analysis. To save/load graphs to/from disk for later use in OSMnx, use the *io.save_graphml* and *io.load_graphml* functions instead. To load a graph from a .osm file that you have downloaded or generated elsewhere, use the *graph.graph_from_xml* function.

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.settings.all_oneway = True
>>> ox.settings.useful_tags_node = utn
>>> ox.settings.useful_tags_way = utw
>>> G = ox.graph_from_place('Piedmont, CA, USA', network_type='drive')
>>> ox.save_graph_xml(G, filepath='./data/graph.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.
- **api_version** (*float*) – OpenStreetMap API version to write to the XML file header
- **precision** (*int*) – number of decimal places to round latitude and longitude values

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

Return type
None

6.4.13 osmnx._nominatim module

Tools to work with the Nominatim API.

osmnx._nominatim.**_download_nominatim_element**(*query*, *by_osmid=False*, *limit=1*, *polygon_geojson=1*)

Retrieve an OSM element from the Nominatim API.

Parameters

- **query** (*string or dict*) – query string or structured query dict
- **by_osmid** (*bool*) – if True, treat query as an OSM ID 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._nominatim.**_nominatim_request**(*params*, *request_type='search'*, *pause=1*, *error_pause=60*)

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

Parameters

- **params** (*OrderedDict*) – key-value pairs of parameters
- **request_type** (*string {"search", "reverse", "lookup"}*) – which Nominatim API endpoint to query
- **pause** (*float*) – 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** (*float*) – how long to pause in seconds before re-trying request if error

Returns
response_json

Return type
dict

6.4.14 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 https://wiki.openstreetmap.org/wiki/OSM_XML and https://overpass-api.de

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_merged_edge_attrs**(*xml_edge*, *sample_edge*, *all_edges_df*, *edge_tags*, *edge_tag_aggs*)

Extract edge attributes and append to XML edge.

Parameters

- **xml_edge** (*ElementTree.SubElement*) – XML representation of an output graph edge
- **sample_edge** (*pandas.Series*) – sample row from the the dataframe of way edges

- **all_edges_df** (*pandas.DataFrame*) – a dataframe with one row for each edge in an OSM way
- **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')]* 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.

osmnx.osm_xml.**_append_nodes_as_edge_attrs**(*xml_edge*, *sample_edge*, *all_edges_df*)

Extract list of ordered nodes and append as attributes of XML edge.

Parameters

- **xml_edge** (*ElementTree.SubElement*) – XML representation of an output graph edge
- **sample_edge** (*pandas.Series*) – sample row from the the dataframe of way edges
- **all_edges_df** (*pandas.DataFrame*) – a dataframe with one row for each edge in an OSM way

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.**_create_way_for_each_edge**(*root*, *gdf_edges*, *edge_attrs*, *edge_tags*)

Append a new way to an empty XML tree graph for each edge in way.

This will generate separate OSM ways for each network edge, even if the edges are all part of the same original OSM way. As such, each way will be composed of two nodes, and there will be many ways with the same OSM ID. This does not conform to the OSM XML schema standard, but the data will still comprise a valid network and will be readable by most OSM tools.

Parameters

- **root** (*ElementTree.Element*) – an empty 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

osmnx.osm_xml.**_get_unique_nodes_ordered_from_way**(*df_way_edges*)

Recover original node order from edges associated with a 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. 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 ensures 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*, *encoding*)

Read OSM XML from file and return Overpass-like JSON.

Parameters

- **filepath** (*string or pathlib.Path*) – path to file containing OSM XML data
- **encoding** (*string*) – the XML file's character encoding

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*, *api_version=0.6*, *precision=6*)

Save graph to disk as an OSM-formatted UTF-8 encoded XML .osm file.

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.

- **api_version** (*float*) – OpenStreetMap API version to write to the XML file header
- **precision** (*int*) – number of decimal places to round latitude and longitude values

Return type
None

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*, *api_version=0.6*, *precision=6*)

Do not use: deprecated.

The *save_graph_xml* has moved from the *osm_xml* module to the *io* module. *osm_xml.save_graph_xml* has been deprecated and will be removed in a future release. Access the function via the *io* module instead.

Parameters

- **data** (*networkx.multidigraph*) – do not use, deprecated
- **filepath** (*string or pathlib.Path*) – do not use, deprecated
- **node_tags** (*list*) – do not use, deprecated
- **node_attrs** (*list*) – do not use, deprecated
- **edge_tags** (*list*) – do not use, deprecated
- **edge_attrs** (*list*) – do not use, deprecated
- **oneway** (*bool*) – do not use, deprecated
- **merge_edges** (*bool*) – do not use, deprecated
- **edge_tag_aggs** (*list of length-2 string tuples*) – do not use, deprecated
- **api_version** (*float*) – do not use, deprecated
- **precision** (*int*) – do not use, deprecated

Return type
None

6.4.15 osmnx._overpass module

Tools to work with the Overpass API.

osmnx._overpass.**_create_overpass_query**(*polygon_coord_str*, *tags*)

Create an Overpass features query string based on passed tags.

Parameters

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

Returns
query

Return type
string

osmnx._overpass.**_download_overpass_features**(*polygon*, *tags*)

Retrieve OSM features 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

Yields
response_json (*dict*) – a generator of JSON responses from the Overpass server

osmnx._overpass.**_download_overpass_network**(*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

Yields
response_json (*dict*) – a generator of JSON responses from the Overpass server

osmnx._overpass.**_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

Return type
string

osmnx._overpass.**_get_overpass_pause**(*base_endpoint*, *recursive_delay=5*, *default_duration=60*)

Retrieve a pause duration from the Overpass API status endpoint.

Check the Overpass API status endpoint to determine how long to wait until the next slot is available. You can disable this via the *settings* module's *overpass_rate_limit* setting.

Parameters

- **base_endpoint** (*string*) – base Overpass API url (without "/status" at the end)
- **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._overpass._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-lon, then get a list of polygon(s) exterior coordinates

Parameters
polygon (*shapely.geometry.Polygon or shapely.geometry.MultiPolygon*) – polygon to convert to exterior coordinate strings

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

Return type
list

osmnx._overpass._make_overpass_settings()

Make settings string to send in Overpass query.

Return type
string

osmnx._overpass._overpass_request(*data*, *pause=None*, *error_pause=60*)

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

Parameters

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

Returns
response_json

Return type
dict

6.4.16 osmnx.plot module

Visualize street networks, routes, orientations, and geospatial features.

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.**_verify_mpl**()

Verify that matplotlib is installed and successfully imported.

Return type

None

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'*, *edge_color='none'*, *edge_linewidth=0*, *alpha=None*, *bgcolor='#111111'*, *bbox=None*, *save=False*, *show=True*, *close=False*, *filepath=None*, *dpi=600*)

Visualize a GeoDataFrame of geospatial features' 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
- **edge_color** (*string*) – color of the edge of the footprints
- **edge_linewidth** (*float*) – width of the edge 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*)

Visualize 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*)

Visualize 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'*, *route_linewidths=4*, ***pgr_kwargs*)

Visualize 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.
- **route_linewidths** (*int or list*) – if int, 1 linewidth for all routes. if list, the linewidth for each route.
- **pgr_kwargs** – keyword arguments to pass to plot_graph_route

Returns
fig, ax – matplotlib figure, axis

Return type
tuple

osmnx.plot.**plot_orientation**(*Gu*, *num_bins=36*, *min_length=0*, *weight=None*, *ax=None*, *figsize=(5, 5)*, *area=True*, *color='#003366'*, *edgecolor='k'*, *linewidth=0.5*, *alpha=0.7*, *title=None*, *title_y=1.05*, *title_font=None*, *xtick_font=None*)

Plot a polar histogram of a spatial network's bidirectional edge bearings.

Ignores self-loop edges as their bearings are undefined.

For more info see: Boeing, G. 2019. "Urban Spatial Order: Street Network Orientation, Configuration, and Entropy." Applied Network Science, 4 (1), 67. https://doi.org/10.1007/s41109-019-0189-1

Parameters

- **Gu** (*networkx.MultiGraph*) – undirected, unprojected graph with *bearing* attributes on each edge
- **num_bins** (*int*) – number of bins; for example, if *num_bins=36* is provided, then each bin will represent 10 degrees around the compass
- **min_length** (*float*) – ignore edges with *length* attributes less than *min_length*
- **weight** (*string*) – if not None, weight edges' bearings by this (non-null) edge attribute
- **ax** (*matplotlib.axes.PolarAxesSubplot*) – if not None, plot on this preexisting axis; must have projection=polar
- **figsize** (*tuple*) – if ax is None, create new figure with size (width, height)
- **area** (*bool*) – if True, set bar length so area is proportional to frequency, otherwise set bar length so height is proportional to frequency
- **color** (*string*) – color of histogram bars
- **edgecolor** (*string*) – color of histogram bar edges
- **linewidth** (*float*) – width of histogram bar edges
- **alpha** (*float*) – opacity of histogram bars
- **title** (*string*) – title for plot
- **title_y** (*float*) – y position to place title
- **title_font** (*dict*) – the title's fontdict to pass to matplotlib
- **xtick_font** (*dict*) – the xtick labels' fontdict to pass to matplotlib

Returns
fig, ax – matplotlib figure, axis

Return type
tuple

6.4.17 osmnx.projection module

Project a graph, GeoDataFrame, or geometry to a different CRS.

osmnx.projection.**is_projected**(*crs*)

Determine if a coordinate reference system is projected or not.

Parameters
crs (*string or pyproj.CRS*) – the identifier of the coordinate reference system, which can be anything accepted by *pyproj.CRS.from_user_input()* such as an authority string or a WKT string

Returns
projected – True if crs is projected, otherwise False

Return type
bool

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

Project a GeoDataFrame from its current CRS to another.

If *to_latlong* is *True*, this projects the GeoDataFrame to the CRS defined by *settings.default_crs*, otherwise it projects it to the CRS defined by *to_crs*. If *to_crs* is *None*, it projects it to the CRS of an appropriate UTM zone given *gdf*'s bounds.

Parameters

- **gdf** (*geopandas.GeoDataFrame*) – the GeoDataFrame to be projected
- **to_crs** (*string or pyproj.CRS*) – if None, project to an appropriate UTM zone, 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_latlong* is *True*, this projects the GeoDataFrame to the CRS defined by *settings.default_crs*, otherwise it projects it to the CRS defined by *to_crs*. If *to_crs* is *None*, it projects it to the CRS of an appropriate UTM zone given *geometry*'s bounds.

Parameters

- **geometry** (*shapely geometry*) – the geometry to be projected
- **crs** (*string or pyproj.CRS*) – the initial CRS of *geometry*. if None, it will be set to *settings.default_crs*
- **to_crs** (*string or pyproj.CRS*) – if None, project to an appropriate UTM zone, 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*, *to_latlong=False*)

Project a graph from its current CRS to another.

If *to_latlong* is *True*, this projects the GeoDataFrame to the CRS defined by *settings.default_crs*, otherwise it projects it to the CRS defined by *to_crs*. If *to_crs* is *None*, it projects it to the CRS of an appropriate UTM zone given *G*'s bounds.

Parameters

- **G** (*networkx.MultiDiGraph*) – the graph to be projected
- **to_crs** (*string or pyproj.CRS*) – if None, project to an appropriate UTM zone, otherwise project to this CRS
- **to_latlong** (*bool*) – if True, project to *settings.default_crs* and ignore *to_crs*

Returns
G_proj – the projected graph

Return type
networkx.MultiDiGraph

6.4.18 osmnx.routing module

Calculate weighted shortest paths between graph nodes.

osmnx.routing.**_single_shortest_path**(*G*, *orig*, *dest*, *weight*)

Solve the shortest path from an origin node to a destination node.

This function is a convenience wrapper around networkx.shortest_path, with exception handling for unsolvable paths. It uses Dijkstra's algorithm.

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

Returns
path – list of node IDs constituting the shortest path

Return type
list

osmnx.routing.**_verify_edge_attribute**(*G*, *attr*)

Verify attribute values are numeric and non-null across graph edges.

Raises a *ValueError* if attribute contains non-numeric values and raises a warning if attribute is missing or null on any edges.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **attr** (*string*) – edge attribute to verify

Return type
None

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

Solve *k* shortest paths from an origin node to a destination node.

Uses Yen's algorithm. See also *shortest_path* to solve 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 solve
- **weight** (*string*) – edge attribute to minimize when solving shortest paths. default is edge length in meters.

Yields
path (*list*) – a generator of *k* shortest paths ordered by total weight. each path is a list of node IDs.

osmnx.routing.**shortest_path**(*G*, *orig*, *dest*, *weight='length'*, *cpus=1*)

Solve shortest path from origin node(s) to destination node(s).

Uses Dijkstra's algorithm. If *orig* and *dest* are single node IDs, this will return a list of the nodes constituting the shortest path between them. If *orig* and *dest* are lists of node IDs, this will return a list of lists of the nodes constituting the shortest path between each origin-destination pair. If a path cannot be solved, this will return None for that path. You can parallelize solving multiple paths with the *cpus* parameter, but be careful to not exceed your available RAM.

See also *k_shortest_paths* to solve multiple shortest paths between a single origin and destination. For additional functionality or different solver algorithms, use NetworkX directly.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **orig** (*int or list*) – origin node ID, or a list of origin node IDs
- **dest** (*int or list*) – destination node ID, or a list of destination node IDs
- **weight** (*string*) – edge attribute to minimize when solving shortest path
- **cpus** (*int*) – how many CPU cores to use; if None, use all available

Returns
path – list of node IDs constituting the shortest path, or, if orig and dest are lists, then a list of path lists

Return type
list

6.4.19 osmnx.settings module

Global settings that can be configured by the user.

all_oneway
[bool] Only use if specifically saving to .osm XML file with the *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. This also retains original OSM string values for oneway attribute values, rather than converting them to a True/False bool. Default is *False*.

bidirectional_network_types
[list] Network types for which a fully bidirectional graph will be created. Default is *["walk"]*.

cache_folder
[string or pathlib.Path] Path to folder in which to save/load HTTP response cache, if the *use_cache* setting equals *True*. Default is *"./cache"*.

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 local cache to quickly build many graphs simultaneously with multiprocessing. Default is *False*.

data_folder
[string or pathlib.Path] Path to folder in which to save/load graph files by default. Default is *"./data"*.

default_accept_language
[string] HTTP header accept-language. Default is *"en"*.

default_access
[string] Default filter for OSM "access" key. Default is *'["access"!~"private"]'*. Note that also filtering out "access=no" ways prevents including transit-only bridges (e.g., Tilikum Crossing) from appearing in drivable road network (e.g., *'["access"!~"private|no"]'*). However, some drivable tollroads have "access=no" plus a "access:conditional" key to clarify when it is accessible, so we can't filter out all "access=no" ways by default. Best to be permissive here then remove complicated combinations of tags programatically after the full graph is downloaded and constructed.

default_crs
[string] Default coordinate reference system to set when creating graphs. Default is *"epsg:4326"*.

default_referer
[string] HTTP header referer. Default is *"OSMnx Python package (https://github.com/gboeing/osmnx)"*.

default_user_agent
[string] HTTP header user-agent. Default is *"OSMnx Python package (https://github.com/gboeing/osmnx)"*.

doh_url_template
[string] Endpoint to resolve DNS-over-HTTPS if local DNS resolution fails. Set to None to disable DoH, but see *downloader._config_dns* documentation for caveats. Default is: *"https://8.8.8.8/resolve?name={hostname}"*

elevation_url_template
[string] Endpoint of the Google Maps Elevation API (or equivalent), containing exactly two parameters: *locations* and *key*. Default is: *"https://maps.googleapis.com/maps/api/elevation/json?locations={locations}&key={key}"*
One example of an alternative equivalent would be Open Topo Data: *"https://api.opentopodata.org/v1/aster30m?locations={locations}&key={key}"*

imgs_folder
[string or pathlib.Path] Path to folder in which to save plotted images by default. Default is *"./images"*.

log_file
[bool] If True, save log output to a file in logs_folder. Default is *False*.

log_filename
[string] Name of the log file, without file extension. Default is *"osmnx"*.

log_console
[bool] If True, print log output to the console (terminal window). Default is *False*.

log_level
[int] One of Python's logger.level constants. Default is *logging.INFO*.

log_name
[string] Name of the logger. Default is *"OSMnx"*.

logs_folder
[string or pathlib.Path] Path to folder in which to save log files. Default is *"./logs"*.

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 the API. Default is *2500000000*.

memory
[int] Overpass server memory allocation size for the query, in bytes. If None, server will use its default allocation size. Use with caution. Default is *None*.

nominatim_endpoint
[string] The base API url to use for Nominatim queries. Default is *"https://nominatim.openstreetmap.org/"*.

nominatim_key
[string] Your Nominatim API key, if you are using an API instance that requires one. Default is *None*.

osm_xml_node_attrs
[list] Node attributes for saving .osm XML files with *save_graph_xml* function. Default is *["id", "timestamp", "uid", "user", "version", "changeset", "lat", "lon"]*.

osm_xml_node_tags
[list] Node tags for saving .osm XML files with *save_graph_xml* function. Default is *["highway"]*.

osm_xml_way_attrs
[list] Edge attributes for saving .osm XML files with *save_graph_xml* function. Default is *["id", "timestamp", "uid", "user", "version", "changeset"]*.

osm_xml_way_tags
[list] Edge tags for for saving .osm XML files with *save_graph_xml* function. Default is *["highway", "lanes", "maxspeed", "name", "oneway"]*.

overpass_endpoint
[string] The base API url to use for Overpass queries. Default is *"https://overpass-api.de/api"*.

overpass_rate_limit
[bool] If True, check the Overpass server status endpoint for how long to pause before making request. Necessary if server uses slot management, but can be set to False if you are running your own overpass instance without rate limiting. Default is *True*.

overpass_settings
[string] Settings string for Overpass queries. Default is *"[out:json][timeout:{timeout}]{maxsize}"*. By default, the {timeout} and {maxsize} values are set dynamically by OSMnx when used. To query, for example, historical OSM data as of a certain date: *'[out:json][timeout:90][date:"2019-10-28T19:20:00Z"]'*. Use with caution.

requests_kwargs
[dict] Optional keyword args to pass to the requests package when connecting to APIs, for example to configure authentication or provide a path to a local certificate file. More info on options such as auth, cert, verify, and proxies can be found in the requests package advanced docs. Default is *{}*.

timeout
[int] The timeout interval in seconds for HTTP requests, and (when applicable) for API to use while running the query. Default is *180*.

use_cache
[bool] If True, cache HTTP responses locally instead of calling API repeatedly for the same request. Default is *True*.

useful_tags_node
[list] OSM "node" tags to add as graph node attributes, when present in the data retrieved from OSM. Default is *["ref", "highway"]*.

useful_tags_way
[list] OSM "way" tags to add as graph edge attributes, when present in the data retrieved from OSM. Default is *["bridge", "tunnel", "oneway", "lanes", "ref", "name", "highway", "maxspeed", "service", "access", "area", "landuse", "width", "est_width", "junction"]*.

6.4.20 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*) – a generator of paths to simplify

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

Determine if a node is 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

Return type
bool

osmnx.simplification.**_merge_nodes_geometric**(*G*, *tolerance*)

Geometrically merge nodes within some distance of each other.

Parameters

- **G** (*networkx.MultiDiGraph*) – a projected graph
- **tolerance** (*float*) – buffer nodes to this distance (in graph's geometry's units) then merge overlapping polygons into a single polygon via a unary union operation

Returns
merged – the merged overlapping polygons of the buffered nodes

Return type
GeoSeries

osmnx.simplification.**_remove_rings**(*G*)

Remove all self-contained rings from a graph.

This identifies any connected components that form a self-contained ring without any endpoints, and removes them from the graph.

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

Returns
G – graph with self-contained rings removed

Return type
networkx.MultiDiGraph

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. Note the tolerance represents a per-node buffering radius: for example, to consolidate nodes within 10 meters of each other, use tolerance=5.

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*, *track_merged=False*)

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. Optionally, the simplified edges can receive a *merged_edges* attribute that contains a list of all the (u, v) node pairs that were merged together.

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
- **track_merged** (*bool*) – if True, add *merged_edges* attribute on simplified edges, containing a list of all the (u, v) node pairs that were merged together

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

Return type
networkx.MultiDiGraph

6.4.21 osmnx.speed module

Calculate graph edge speeds and travel times.

osmnx.speed.**_clean_maxspeed**(*maxspeed*, *agg=numpy.mean*, *convert_mph=True*)

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

If present, splits maxspeed on "|" (which denotes that the value contains different speeds per lane) then aggregates the resulting values. Invalid inputs return None. See https://wiki.openstreetmap.org/wiki/Key:maxspeed for details on values and formats.

Parameters

- **maxspeed** (*string*) – a valid OpenStreetMap way maxspeed value
- **agg** (*function*) – aggregation function if maxspeed contains multiple values (default is numpy.mean)
- **convert_mph** (*bool*) – if True, convert miles per hour to km per hour

Returns
clean_value

Return type
string

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

Collapse a list of maxspeed values to a single value.

Parameters

- **value** (*list or string*) – an OSM way maxspeed value, or a list of them
- **agg** (*function*) – the aggregation function to reduce the list to a single value

Returns
agg_value – an integer representation of the aggregated 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=None*, *agg=numpy.mean*)

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

By default, this imputes free-flow travel speeds for all edges via the mean *maxspeed* value of the edges of each highway type. For highway types in the graph that have no *maxspeed* value on any edge, it assigns the mean of all *maxspeed* values in graph.

This default mean-imputation can obviously be imprecise, and the user can override it by passing in *hwy_speeds* and/or *fallback* arguments that correspond to local speed limit standards. The user can also specify a different aggregation function (such as the median) to impute missing values from the observed values.

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*) – deprecated, do not use
- **agg** (*function*) – aggregation function to impute missing values from observed values. the default is numpy.mean, but you might also consider for example numpy.median, numpy.nanmedian, or your own custom function

Returns

G – graph with speed_kph attributes on all edges

Return type

networkx.MultiDiGraph

osmnx.speed.**add_edge_travel_times**(*G*, *precision=None*)

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*) – deprecated, do not use

Returns

G – graph with travel_time attributes on all edges

Return type

networkx.MultiDiGraph

6.4.22 osmnx.stats module

Calculate geometric and topological network measures.

This module defines streets as the edges in an undirected representation of the graph. Using undirected graph edges prevents double-counting bidirectional edges of a two-way street, but may double-count a divided road's separate centerlines with different end point nodes. If *clean_periphery=True* when the graph was created (which is the default parameterization), then you will get accurate node degrees (and in turn streets-per-node counts) even at the periphery of the graph.

You can use NetworkX directly for additional topological network measures.

osmnx.stats.**basic_stats**(*G*, *area=None*, *clean_int_tol=None*)

Calculate basic descriptive geometric and topological measures of a graph.

Density measures are only calculated if *area* is provided and clean intersection measures are only calculated if *clean_int_tol* is provided.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **area** (*float*) – if not None, calculate density measures and use this value (in square meters) as the denominator
- **clean_int_tol** (*float*) – if not None, calculate consolidated intersections count (and density, if *area* is also provided) and use this tolerance value; refer to the *simplification.consolidate_intersections* function documentation for details

Returns

stats –

dictionary containing the following keys

- *circuity_avg* - see *circuity_avg* function documentation
- *clean_intersection_count* - see *clean_intersection_count* function documentation
- *clean_intersection_density_km* - *clean_intersection_count* per sq km
- *edge_density_km* - *edge_length_total* per sq km
- *edge_length_avg* - *edge_length_total / m*
- *edge_length_total* - see *edge_length_total* function documentation
- *intersection_count* - see *intersection_count* function documentation
- *intersection_density_km* - *intersection_count* per sq km
- *k_avg* - graph's average node degree (in-degree and out-degree)
- *m* - count of edges in graph
- *n* - count of nodes in graph
- *node_density_km* - *n* per sq km
- *self_loop_proportion* - see *self_loop_proportion* function documentation
- *street_density_km* - *street_length_total* per sq km
- *street_length_avg* - *street_length_total / street_segment_count*
- *street_length_total* - see *street_length_total* function documentation
- *street_segment_count* - see *street_segment_count* function documentation

- *streets_per_node_avg* - see *streets_per_node_avg* function documentation
- *streets_per_node_counts* - see *streets_per_node_counts* function documentation
- *streets_per_node_proportions* - see *streets_per_node_proportions* function documentation

Return type
dict

osmnx.stats.**circuity_avg**(*Gu*)

Calculate average street circuity using edges of undirected graph.

Circuity is the sum of edge lengths divided by the sum of straight-line distances between edge endpoints. Calculates straight-line distance as euclidean distance if projected or great-circle distance if unprojected.

Parameters
Gu (*networkx.MultiGraph*) – undirected input graph

Returns
circuity_avg – the graph's average undirected edge circuity

Return type
float

osmnx.stats.**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.stats.**edge_length_total**(*G*)

Calculate graph's total edge length.

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

Returns
length – total length (meters) of edges in graph

Return type
float

osmnx.stats.**intersection_count**(*G=None*, *min_streets=2*)

Count the intersections in a graph.

Intersections are defined as nodes with at least *min_streets* number of streets incident on them.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **min_streets** (*int*) – a node must have at least *min_streets* incident on them to count as an intersection

Returns
count – count of intersections in graph

Return type
int

osmnx.stats.**self_loop_proportion**(*Gu*)

Calculate percent of edges that are self-loops in a graph.

A self-loop is defined as an edge from node *u* to node *v* where *u*==*v*.

Parameters
Gu (*networkx.MultiGraph*) – undirected input graph

Returns
proportion – proportion of graph edges that are self-loops

Return type
float

osmnx.stats.**street_length_total**(*Gu*)

Calculate graph's total street segment length.

Parameters
Gu (*networkx.MultiGraph*) – undirected input graph

Returns
length – total length (meters) of streets in graph

Return type
float

osmnx.stats.**street_segment_count**(*Gu*)

Count the street segments in a graph.

Parameters
Gu (*networkx.MultiGraph*) – undirected input graph

Returns
count – count of street segments in graph

Return type
int

osmnx.stats.**streets_per_node**(*G*)

Count streets (undirected edges) incident on each node.

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

Returns
spn – dictionary with node ID keys and street count values

Return type
dict

osmnx.stats.**streets_per_node_avg**(*G*)

Calculate graph's average count of streets per node.

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

Returns
spna – average count of streets per node

Return type
float

osmnx.stats.**streets_per_node_counts**(*G*)

Calculate streets-per-node counts.

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

Returns
spnc – dictionary keyed by count of streets incident on each node, and with values of how many nodes in the graph have this count

Return type
dict

osmnx.stats.**streets_per_node_proportions**(*G*)

Calculate streets-per-node proportions.

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

Returns
spnp – dictionary keyed by count of streets incident on each node, and with values of what proportion of nodes in the graph have this count

Return type
dict

6.4.23 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

6.4.24 osmnx.utils module

General utility functions.

osmnx.utils.**_get_logger**(*level*, *name*, *filename*)

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**(*style='bibtex'*)

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

Parameters
style (*string {"apa", "bibtex", "ieee"}*) – citation format, either APA or BibTeX or IEEE

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)'*, *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='https://overpass-api.de/api'*, *overpass_rate_limit=True*, *overpass_settings='[out:json][timeout:{timeout}]{maxsize}'*, *requests_kwargs={}*, *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']*)

Do not use: deprecated. Use the settings module directly.

Parameters

- **all_oneway** (*bool*) – deprecated
- **bidirectional_network_types** (*list*) – deprecated
- **cache_folder** (*string or pathlib.Path*) – deprecated
- **data_folder** (*string or pathlib.Path*) – deprecated
- **cache_only_mode** (*bool*) – deprecated
- **default_accept_language** (*string*) – deprecated
- **default_access** (*string*) – deprecated
- **default_crs** (*string*) – deprecated
- **default_referer** (*string*) – deprecated
- **default_user_agent** (*string*) – deprecated
- **imgs_folder** (*string or pathlib.Path*) – deprecated
- **log_file** (*bool*) – deprecated
- **log_filename** (*string*) – deprecated
- **log_console** (*bool*) – deprecated
- **log_level** (*int*) – deprecated
- **log_name** (*string*) – deprecated
- **logs_folder** (*string or pathlib.Path*) – deprecated
- **max_query_area_size** (*int*) – deprecated
- **memory** (*int*) – deprecated
- **nominatim_endpoint** (*string*) – deprecated
- **nominatim_key** (*string*) – deprecated
- **osm_xml_node_attrs** (*list*) – deprecated
- **osm_xml_node_tags** (*list*) – deprecated
- **osm_xml_way_attrs** (*list*) – deprecated
- **osm_xml_way_tags** (*list*) – deprecated
- **overpass_endpoint** (*string*) – deprecated
- **overpass_rate_limit** (*bool*) – deprecated
- **overpass_settings** (*string*) – deprecated
- **requests_kwargs** (*dict*) – deprecated
- **timeout** (*int*) – deprecated
- **use_cache** (*bool*) – deprecated
- **useful_tags_node** (*list*) – deprecated
- **useful_tags_way** (*list*) – deprecated

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

Return type
None

osmnx.utils.**ts**(*style='datetime'*, *template=None*)

Return current local timestamp as a string.

Parameters

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

Returns
ts – local timestamp string

Return type
string

6.4.25 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.

When the geometry has a very large area relative to its vertex count, the resulting MultiPolygon's boundary may differ somewhat from the input, due to the way long straight lines are projected. You can interpolate additional vertices along your input geometry's exterior to mitigate this.

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.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

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

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

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

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

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

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, lon) 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, lon) 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

Return type
shapely.geometry.Polygon

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

Interpolate evenly spaced points along a LineString.

The spacing is approximate because the LineString's length may not be evenly divisible by it.

Parameters

- **geom** (*shapely.geometry.LineString*) – a LineString geometry
- **dist** (*float*) – spacing distance between interpolated points, in same units as *geom*. smaller values generate more points.

Yields
point (*tuple of floats*) – a generator of (x, y) tuples of the interpolated points' coordinates

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

Do not use: deprecated.

Parameters

- **geom** (*shapely.geometry.geometry {Point, MultiPoint, LineString, MultiLineString, Polygon, MultiPolygon}*) – deprecated, do not use
- **precision** (*int*) – deprecated, do not use

Return type
shapely.geometry.geometry

osmnx.utils_geo.**sample_points**(*G*, *n*)

Randomly sample points constrained to a spatial graph.

This generates a graph-constrained uniform random sample of points. Unlike typical spatially uniform random sampling, this method accounts for the graph's geometry. And unlike equal-length edge segmenting, this method guarantees uniform randomness.

Parameters

- **G** (*networkx.MultiGraph*) – graph to sample points from; should be undirected (to not oversample bidirectional edges) and projected (for accurate point interpolation)
- **n** (*int*) – how many points to sample

Returns
points – the sampled points, multi-indexed by (u, v, key) of the edge from which each point was drawn

Return type
geopandas.GeoSeries

6.4.26 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

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.**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

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*)

Do not use: deprecated.

Use the *route_to_gdf* function instead.

Parameters

- **G** (*networkx.MultiDiGraph*) – deprecated
- **route** (*list*) – deprecated
- **attribute** (*string*) – deprecated
- **minimize_key** (*string*) – deprecated

- **retrieve_default** (*Callable[Tuple[Any, Any], Any]*) – deprecated

Returns
attribute_values – deprecated

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

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 1) *gdf_nodes* is uniquely indexed by *osmid*, 2) *gdf_nodes* contains *x* and *y* coordinate columns representing node geometries, 3) *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. Note that any *geometry* attribute on *gdf_nodes* is discarded since *x* and *y* provide the necessary node geometry information instead.

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 attributes

- **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). gdf_nodes is indexed by osmid and gdf_edges is multi-indexed by u, v, key following normal MultiDiGraph structure.

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

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

Return a GeoDataFrame of the edges in a path, in order.

Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **route** (*list*) – list of node IDs constituting the path
- **weight** (*string*) – if there are parallel edges between two nodes, choose lowest weight

Returns
gdf_edges – GeoDataFrame of the edges

Return type
geopandas.GeoDataFrame

6.4.27 osmnx._version module

OSMnx package version information.

6.5 Further Reading

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

This is the original paper introducing OSMnx and is the official citation for the project.

Boeing, G. 2020. The Right Tools for the Job: The Case for Spatial Science Tool-Building. *Transactions in GIS* 24 (5), 1299-1314.

This paper was presented as the 8th annual Transactions in GIS plenary address at the American Association of Geographers annual meeting in Washington, DC. It describes the development of OSMnx and reviews its use in scientific research over the previous few years.

Boeing, G. 2020. Planarity and Street Network Representation in Urban Form Analysis. *Environment and Planning B: Urban Analytics and City Science* 47 (5), 855-869.

This paper discusses the importance of using nonplanar graphs when modeling urban street networks, which was one of the original motivations for developing OSMnx.

Boeing, G. 2021. Street Network Models and Indicators for Every Urban Area in the World. *Geographical Analysis* 54 (3), 519-535.

This study uses OSMnx to model and analyze the street networks of every urban area in the world: over 160 million OpenStreetMap street network nodes and over 320 million edges across 8,914 urban areas in 178 countries.

Boeing, G., C. Higgs, S. Liu, B. Giles-Corti, J.F. Sallis, E. Cerin, et al. 2022. Using Open Data and Open-Source Software to Develop Spatial Indicators of Urban Design and Transport Features for Achieving Healthy and Sustainable Cities. *The Lancet Global Health* 10 (6), 907-918.

This study by an international consortium of public health and urban planning researchers uses OSMnx to develop and demonstrate a computational framework to benchmark and monitor urban accessibility around the world.

CHAPTER

SEVEN

INDICES

- genindex
- modindex
- search

PYTHON MODULE INDEX

O

INDEX

M

N

O

P

R

S

T