OSMnx

Release 2.1.1

Geoff Boeing

Nov 10, 2026

CONTENTS

1 Citation 3

2 Getting Started 5

3 Installation 7

4 Support 9

5 License 11

6 User Guides 13

7 Indices 125

Python Module Index 127

Index 129

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 paper:

Boeing, G. (2025). Modeling and Analyzing Urban Networks and Amenities with OSMnx. *Geographical Analysis* 57 (4), 567-577. doi:10.1111/gean.70009

CHAPTER

TWO

GETTING STARTED

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

Then work through the 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

USER GUIDES

6.1 Installation

6.1.1 Conda

The foolproof way to install OSMnx is with conda or mamba:

```
conda create --strict-channel-priority -c conda-forge -n ox 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. See the conda-forge documentation for details.

6.1.2 Docker

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

6.1.3 Pip

You can also install OSMnx with uv or pip (into a virtual environment):

```
pip install osmnx
```

OSMnx is written in pure Python and distributed on PyPI. Its installation alone is thus trivially simple if you have its dependencies installed and tested on your system. However, OSMnx depends on other packages that in turn depend on compiled C/C++ libraries, which may present some challenges depending on your specific system's configuration. If precompiled binaries are not available for your system, you may need to compile and configure those dependencies by following their installation instructions. So, if you're not sure what you're doing, just follow the conda instructions above to avoid installation problems.

6.2 Getting Started

6.2.1 Get Started in 4 Steps

1. Install OSMnx by following the *Installation* guide.
2. Read the *Introducing OSMnx* section 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.

6.2.2 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.

Read more about the *settings* module in the User Reference.

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. Read more about the *geocoder* module in the User Reference.

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 any 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.

Read more about the *features* module in the User Reference.

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.

In short, 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. 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. By default, it returns the largest weakly connected component. It also simplifies the graph topology as discussed below.

Read more about the *graph* module in the User Reference and refer to the official reference paper at the *Further Reading* page for complete modeling details.

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 can be aggregated into a single value.

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. When multiple OpenStreetMap nodes are merged into a single graph node, the nodes' attribute values can be aggregated into a single value.

Read more about the *simplification* module in the User Reference.

Model Attributes

An OSMnx model has some standard required attributes, plus some optional attributes. The latter are sometimes present based on the source OSM data's tagging, the `settings` module configuration, and any processing you may have done to add additional attributes (as noted in various functions' documentation).

As a NetworkX MultiDiGraph object, it has top-level `graph`, `nodes`, and `edges` attributes. The `graph` attribute dictionary must contain a "crs" key defining its coordinate reference system. The `nodes` are identified by OSM ID and each must contain a `data` attribute dictionary that must have "x" and "y" keys defining its coordinates and a "street_count" key defining how many physical streets are incident to it. The `edges` are identified by a 3-tuple of "u" (source node ID), "v" (target node ID), and "key" (to differentiate parallel edges), and each must contain a `data` attribute dictionary that must have an "osmid" key defining its OSM ID and a "length" key defining its length in meters.

The OSMnx `graph` module automatically creates MultiDiGraphs with these required attributes, plus additional optional attributes based on the `settings` module configuration. If you instead manually create your own graph model, make sure it has these required attributes at a minimum.

Convert, Project, Save

OSMnx's `convert` module can convert a MultiDiGraph to a DiGraph if you prefer a directed representation of the network without any parallel edges, or to a MultiGraph if you need an undirected representation for use with functions or algorithms that only accept a MultiGraph object. If you just want a fully bidirectional graph (such as for a walking network), just configure the `settings` module's `bidirectional_network_types` before creating your graph.

The `convert` module 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. The `convert` module exposes validation functions to verify that your MultiDiGraph or GeoDataFrames satisfy OSMnx requirements. Read more about the *convert* module in the User Reference.

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. Read more about the *projection* module in the User Reference.

Using the `io` module, you can save your graph to disk as a GraphML file (to load into other network analysis software), a GeoPackage (to load into other GIS software), or an OSM XML file. Use the GraphML format whenever saving a graph for later work with OSMnx. Read more about the *io* module in the User Reference.

Network Statistics

You can use the `stats` module to calculate a variety of geometric and topological measures as well as street network bearing and 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. Read more about the *stats* module in the User Reference.

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

Working with Elevation

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

Read more about the *elevation* module in the User Reference.

Routing

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). It can also impute missing speeds to the graph edges. This imputation can obviously be imprecise, so 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.

Read more about the *distance* and *routing* modules in the User Reference.

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.

Read more about the *plot* module in the User Reference.

Usage Limits

Be a good neighbor! OSMnx works with Overpass's rate limiting to avoid overwhelming their resources. Don't run multiple/parallel OSMnx instances simultaneously to circumvent their limits. These APIs are free to use, but they're not free to operate. If you will make many queries (e.g., more than 1k/day), you need to host your own local Overpass instance. Refer to the Nominatim Usage Policy and Overpass Commons documentation for API usage limits and restrictions to which you must adhere. If you configure OSMnx to use an alternative API instance, ensure you understand and follow their policies. If you feel you need to exceed these limits, just install your own hosted instance and set OSMnx to use it.

6.2.3 More Info

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

6.2.4 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.

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.

6.3.1 osmnx.bearing module

Calculate graph edge bearings and orientation entropy.

`osmnx.bearing.`**`add_edge_bearings`**`(G)`

Calculate and 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 *bearing* 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 (`MultiDiGraph`) – Unprojected graph.

Returns
G – Graph with *bearing* attributes on the edges.

Return type
networkx.MultiDiGraph

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

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

Overloads

- **lat1** (float), **lon1** (float), **lat2** (float), **lon2** (float) → float
- **lat1** (npt.NDArray[np.float64]), **lon1** (npt.NDArray[np.float64]), **lat2** (npt.NDArray[np.float64]), **lon2** (npt.NDArray[np.float64]) → npt.NDArray[np.float64]

Parameters

- **lat1** (*float | npt.NDArray[np.float64]*)
- **lon1** (*float | npt.NDArray[np.float64]*)
- **lat2** (*float | npt.NDArray[np.float64]*)
- **lon2** (*float | npt.NDArray[np.float64]*)

Return type
float | npt.NDArray[np.float64]

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

Parameters

- **lat1** (float | npt.NDArray[np.float64]) – First point's latitude coordinate(s).
- **lon1** (float | npt.NDArray[np.float64]) – First point's longitude coordinate(s).
- **lat2** (float | npt.NDArray[np.float64]) – Second point's latitude coordinate(s).
- **lon2** (float | npt.NDArray[np.float64]) – Second point's longitude coordinate(s).

Returns
bearing – The bearing(s) in decimal degrees.

Return type
float | npt.NDArray[np.float64]

osmnx.bearing.**orientation_entropy**(*G*, ** (Keyword-only parameters separator (PEP 3102))*, *num_bins=36*, *min_length=0*, *weight=None*)

Calculate graph's orientation entropy.

Orientation entropy is the Shannon entropy of the graphs' edges' bearings across evenly spaced bins. Ignores self-loop edges as their bearings are undefined. If *G* is a MultiGraph, all edge bearings will be bidirectional (ie, two reciprocal bearings per undirected edge). If *G* is a MultiDiGraph, all edge bearings will be directional (ie, one bearing per directed edge).

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

- **G** (nx.MultiGraph | nx.MultiDiGraph) – 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** (str | None) – If None, apply equal weight for each bearing. Otherwise, weight edges' bearings by this (non-null) edge attribute. For example, if "length" is provided, each edge's bearing observation will be weighted by its "length" attribute value.

Returns
entropy – The orientation entropy of *G*.

Return type
float

6.3.2 osmnx.convert module

Convert spatial graphs to/from different data types.

osmnx.convert.**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 network 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** (GeoDataFrame) – GeoDataFrame of graph nodes uniquely indexed by *osmid*.
- **gdf_edges** (GeoDataFrame) – GeoDataFrame of graph edges uniquely multi-indexed by *(u, v, key)*.
- **graph_attrs** (dict[str, Any] | None) – The new *G.graph* attribute dictionary. If None, use *gdf_edges*'s CRS as the only graph-level attribute (*gdf_edges* must have its *crs* attribute set).

Returns
G – The converted MultiDiGraph.

Return type
networkx.MultiDiGraph

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

Convert a MultiGraph or MultiDiGraph to node and/or edge GeoDataFrames.

Overloads

- **G** (nx.MultiGraph | nx.MultiDiGraph), **node_geometry** (bool), **fill_edge_geometry** (bool) → tuple[gpd.GeoDataFrame, gpd.GeoDataFrame]
- **G** (nx.MultiGraph | nx.MultiDiGraph), **nodes** (Literal[True]), **edges** (Literal[True]), **node_geometry** (bool), **fill_edge_geometry** (bool) → tuple[gpd.GeoDataFrame, gpd.GeoDataFrame]

- **G** (nx.MultiGraph | nx.MultiDiGraph), **nodes** (Literal[True]), **edges** (Literal[False]), **node_geometry** (bool), **fill_edge_geometry** (bool) → gpd.GeoDataFrame
- **G** (nx.MultiGraph | nx.MultiDiGraph), **nodes** (Literal[False]), **edges** (Literal[True]), **node_geometry** (bool), **fill_edge_geometry** (bool) → gpd.GeoDataFrame
- **G** (nx.MultiGraph | nx.MultiDiGraph), **edges** (Literal[True]), **node_geometry** (bool), **fill_edge_geometry** (bool) → tuple[gpd.GeoDataFrame, gpd.GeoDataFrame]
- **G** (nx.MultiGraph | nx.MultiDiGraph), **edges** (Literal[False]), **node_geometry** (bool), **fill_edge_geometry** (bool) → gpd.GeoDataFrame
- **G** (nx.MultiGraph | nx.MultiDiGraph), **nodes** (Literal[True]), **edges** (bool), **node_geometry** (bool), **fill_edge_geometry** (bool) → tuple[gpd.GeoDataFrame, gpd.GeoDataFrame]
- **G** (nx.MultiGraph | nx.MultiDiGraph), **nodes** (Literal[False]), **edges** (bool), **node_geometry** (bool), **fill_edge_geometry** (bool) → gpd.GeoDataFrame

Parameters

- **G** (*nx.MultiGraph | nx.MultiDiGraph*)
- **nodes** (*bool*)
- **edges** (*bool*)
- **node_geometry** (*bool*)
- **fill_edge_geometry** (*bool*)

Return type
gpd.GeoDataFrame | tuple[gpd.GeoDataFrame, gpd.GeoDataFrame]

This function is the inverse of *graph_from_gdfs*.

Parameters

- **G** (nx.MultiGraph | nx.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 missing edge geometry fields using endpoint nodes' coordinates to create a LineString.

Returns
gdf_nodes or gdf_edges or (gdf_nodes, gdf_edges) – *gdf_nodes* is indexed by *osmid* and *gdf_edges* is multi-indexed by *(u, v, key)* following normal MultiGraph/MultiDiGraph structure.

Return type
gpd.GeoDataFrame | tuple[gpd.GeoDataFrame, gpd.GeoDataFrame]

`osmnx.convert.to_digraph(G, *, weight='length')`

Convert MultiDiGraph to DiGraph.

Chooses between parallel edges by minimizing *weight* attribute value. See also *to_undirected* to convert MultiDiGraph to MultiGraph.

Parameters

- **G** (`MultiDiGraph`) – Input graph.
- **weight** (`str`) – Attribute value to minimize when choosing between parallel edges.

Returns
D – The converted DiGraph.

Return type
networkx.DiGraph

osmnx.convert.**to_undirected**(*G*)

Convert MultiDiGraph to undirected MultiGraph.

This function has a limited use case: it allows you to create a MultiGraph for use with functions/algorithms that only accept a MultiGraph object. Rather, if you want a fully bidirectional graph (such as for a walking network), configure the *settings* module's *bidirectional_network_types* before creating your graph to generate a fully bidirectional MultiDiGraph.

This function maintains parallel edges only if their geometries differ. See also *to_digraph* to convert MultiDiGraph to DiGraph.

Parameters
G (MultiDiGraph) – Input graph.

Returns
Gu – The converted MultiGraph.

Return type
networkx.MultiGraph

osmnx.convert.**validate_features_gdf**(*gdf*)

Validate that features GeoDataFrame satisfies OSMnx expectations.

Raises a *ValidationError* if validation fails.

Parameters
gdf (GeoDataFrame) – GeoDataFrame of features uniquely multi-indexed by *(element_type, osmid)*.

Return type
None

osmnx.convert.**validate_graph**(*G*, *, *strict=True*)

Validate that a graph object satisfies OSMnx expectations.

Raises *ox._errors.GraphValidationError* if validation fails.

Parameters

- **G** (MultiDiGraph) – The input graph.
- **strict** (bool) – If *True*, enforce optional rules in addition to required rules. These optional rules primarily enforce expected attribute data types.

Return type
None

osmnx.convert.**validate_node_edge_gdfs**(*gdf_nodes*, *gdf_edges*, *, *strict=True*)

Validate that node/edge GeoDataFrames can be converted to a MultiDiGraph.

Raises a *ValidationError* if validation fails.

Parameters

- **gdf_nodes** (GeoDataFrame) – GeoDataFrame of graph nodes uniquely indexed by *osmid*.
- **gdf_edges** (GeoDataFrame) – GeoDataFrame of graph edges uniquely multi-indexed by *(u, v, key)*.

- **strict** (bool) – If *True*, elevate warnings to errors.

Return type
None

6.3.3 osmnx.distance module

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

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

Calculate and add *length* attribute (in meters) to each edge.

Vectorized function to calculate great-circle distance between each edge's incident nodes. Ensure graph is unprojected and unsimplified to calculate 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 (potentially) curvilinear geometry. If you wish to calculate edge lengths later, note that you will be calculating straight-line distances which necessarily ignore the curvilinear geometry. Thus 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** (MultiDiGraph) – Unprojected and unsimplified input graph.
- **edges** (Iterable[tuple[int, int, int]] | None) – The subset of edges to add *length* attributes to, as *(u, v, k)* tuples. If None, add lengths to all edges.

Returns
G – Graph with *length* attributes on the edges.

Return type
networkx.MultiDiGraph

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

Calculate Euclidean distances between pairs of points.

Overloads

- **y1** (float), **x1** (float), **y2** (float), **x2** (float) → float
- **y1** (npt.NDArray[np.float64]), **x1** (npt.NDArray[np.float64]), **y2** (npt.NDArray[np.float64]), **x2** (npt.NDArray[np.float64]) → npt.NDArray[np.float64]

Parameters

- **y1** (*float | npt.NDArray[np.float64]*)
- **x1** (*float | npt.NDArray[np.float64]*)
- **y2** (*float | npt.NDArray[np.float64]*)
- **x2** (*float | npt.NDArray[np.float64]*)

Return type
float | npt.NDArray[np.float64]

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 | npt.NDArray[np.float64]) – First point's y coordinate(s).

- **x1** (float | npt.NDArray[np.float64]) – First point's x coordinate(s).
- **y2** (float | npt.NDArray[np.float64]) – Second point's y coordinate(s).
- **x2** (float | npt.NDArray[np.float64]) – Second point's x coordinate(s).

Returns
dist – Distance from each *(x1, y1)* point to each *(x2, y2)* point in same units as the points' coordinates.

Return type
float | npt.NDArray[np.float64]

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

Calculate great-circle distances between pairs of points.

Overloads

- **lat1** (float), **lon1** (float), **lat2** (float), **lon2** (float) → float
- **lat1** (float), **lon1** (float), **lat2** (float), **lon2** (float), **earth_radius** (float) → float
- **lat1** (npt.NDArray[np.float64]), **lon1** (npt.NDArray[np.float64]), **lat2** (npt.NDArray[np.float64]), **lon2** (npt.NDArray[np.float64]) → npt.NDArray[np.float64]
- **lat1** (npt.NDArray[np.float64]), **lon1** (npt.NDArray[np.float64]), **lat2** (npt.NDArray[np.float64]), **lon2** (npt.NDArray[np.float64]), **earth_radius** (float) → npt.NDArray[np.float64]

Parameters

- **lat1** (*float | npt.NDArray[np.float64]*)
- **lon1** (*float | npt.NDArray[np.float64]*)
- **lat2** (*float | npt.NDArray[np.float64]*)
- **lon2** (*float | npt.NDArray[np.float64]*)
- **earth_radius** (*float*)

Return type
float | npt.NDArray[np.float64]

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 | npt.NDArray[np.float64]) – First point's latitude coordinate(s).
- **lon1** (float | npt.NDArray[np.float64]) – First point's longitude coordinate(s).
- **lat2** (float | npt.NDArray[np.float64]) – Second point's latitude coordinate(s).
- **lon2** (float | npt.NDArray[np.float64]) – Second point's longitude coordinate(s).
- **earth_radius** (float) – Earth's radius in units in which distance will be returned (default represents meters).

Returns
dist – Distance from each *(lat1, lon1)* point to each *(lat2, lon2)* point in units of *earth_radius*.

Return type
float | npt.NDArray[np.float64]

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

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

Overloads

- **G** (nx.MultiDiGraph), **X** (float), **Y** (float) → tuple[int, int, int]
- **G** (nx.MultiDiGraph), **X** (float), **Y** (float), **return_dist** (Literal[False]) → tuple[int, int, int]
- **G** (nx.MultiDiGraph), **X** (float), **Y** (float), **return_dist** (Literal[True]) → tuple[tuple[int, int, int], float]
- **G** (nx.MultiDiGraph), **X** (Iterable[float]), **Y** (Iterable[float]) → npt.NDArray[np.int64]
- **G** (nx.MultiDiGraph), **X** (Iterable[float]), **Y** (Iterable[float]), **return_dist** (Literal[False]) → npt.NDArray[np.int64]
- **G** (nx.MultiDiGraph), **X** (Iterable[float]), **Y** (Iterable[float]), **return_dist** (Literal[True]) → tuple[npt.NDArray[np.int64], npt.NDArray[np.float64]]

Parameters

- **G** (*nx.MultiDiGraph*)
- **X** (*float | Iterable[float]*)
- **Y** (*float | Iterable[float]*)
- **return_dist** (*bool*)

Return type

tuple[int, int, int] | npt.NDArray[np.int64] | tuple[tuple[int, int, int], float] | tuple[npt.NDArray[np.int64], npt.NDArray[np.float64]]

If *X* and *Y* are single coordinate values, this function will return the nearest edge to that point. If *X* and *Y* are iterables of coordinate values, it will return the nearest edge to each point.

This function is vectorized: if you have many points to search for, pass them in one call as numpy arrays (avoid using loops) to maximize runtime speed. It 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 projected points.

Parameters

- **G** (nx.MultiDiGraph) – Graph in which to find nearest edges.
- **X** (float | Iterable[float]) – The points' x (longitude) coordinates, in same CRS/units as graph and containing no nulls.
- **Y** (float | Iterable[float]) – The points' y (latitude) coordinates, in same CRS/units as graph and containing no nulls.
- **return_dist** (bool) – If True, optionally also return the distance(s) between point(s) and nearest edge(s), in same units as graph and points.

Returns

ne or (ne, dist) – Nearest edge ID(s) as *(u, v, k)* tuples, or optionally a tuple of ID(s) and distance(s) between each point and its nearest edge.

Return type

tuple[int, int, int] | npt.NDArray[np.int64] | tuple[tuple[int, int, int], float] | tuple[npt.NDArray[np.int64], npt.NDArray[np.float64]]

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

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

Overloads

- **G** (nx.MultiDiGraph), **X** (float), **Y** (float) → int
- **G** (nx.MultiDiGraph), **X** (float), **Y** (float), **return_dist** (Literal[False]) → int
- **G** (nx.MultiDiGraph), **X** (float), **Y** (float), **return_dist** (Literal[True]) → tuple[npt.NDArray[np.int64], npt.NDArray[np.float64]]
- **G** (nx.MultiDiGraph), **X** (Iterable[float]), **Y** (Iterable[float]) → npt.NDArray[np.int64]
- **G** (nx.MultiDiGraph), **X** (Iterable[float]), **Y** (Iterable[float]), **return_dist** (Literal[False]) → npt.NDArray[np.int64]
- **G** (nx.MultiDiGraph), **X** (Iterable[float]), **Y** (Iterable[float]), **return_dist** (Literal[True]) → tuple[npt.NDArray[np.int64], npt.NDArray[np.float64]]

Parameters

- **G** (*nx.MultiDiGraph*)
- **X** (*float | Iterable[float]*)
- **Y** (*float | Iterable[float]*)
- **return_dist** (*bool*)

Return type

int | npt.NDArray[np.int64] | tuple[int, float] | tuple[npt.NDArray[np.int64], npt.NDArray[np.float64]]

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

This function is vectorized: if you have many points to search for, pass them in one call as numpy arrays (avoid using loops) to maximize runtime speed. If the graph is projected, it uses a k-d tree for Euclidean nearest neighbor search, which requires that scipy is installed as an optional dependency. If the graph is unprojected, it uses a ball tree for haversine nearest neighbor search, which requires that scikit-learn is installed as an optional dependency.

Parameters

- **G** (nx.MultiDiGraph) – Graph in which to find nearest nodes.
- **X** (float | Iterable[float]) – The points' x (longitude) coordinates, in same CRS/units as graph and containing no nulls.
- **Y** (float | Iterable[float]) – The points' y (latitude) coordinates, in same CRS/units as graph and containing no nulls.
- **return_dist** (bool) – If True, optionally also return the distance(s) between point(s) and nearest node(s).

Returns

nn or (nn, dist) – Nearest node ID(s) or optionally a tuple of ID(s) and distance(s) between each point and its nearest node.

Return type

int | npt.NDArray[np.int64] | tuple[int, float] | tuple[npt.NDArray[np.int64], npt.NDArray[np.float64]]

6.3.4 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*)

Calculate and add *grade* attributes to all graph edges.

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

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

Parameters

- **G** (MultiDiGraph) – Graph with *elevation* node attributes.
- **add_absolute** (bool) – If True, also add absolute value of grade as *grade_abs* attribute.

Returns
G – Graph with *grade* (and optionally *grade_abs*) attributes on the edges.

Return type
networkx.MultiDiGraph

osmnx.elevation.**add_node_elevations_google**(*G*, *, *api_key=None*, *batch_size=512*, *pause=0*)

Add *elevation* (meters) attributes to all nodes using a web API.

By default this uses the Google Maps Elevation API, but you could instead use any equivalent API with the same interface and response format (such as the Open Topo Data API or the Open-Elevation API) via the *settings* module's *elevation_url_template*. Adjust the *batch_size* and *pause* arguments as needed for the provider. The Google Maps Elevation API requires an API key but other providers may not. You can find more information about the Google Maps Elevation API interface and format at: https://developers.google.com/maps/documentation/elevation

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

Parameters

- **G** (MultiDiGraph) – Graph to add elevation data to.
- **api_key** (str | None) – 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 request (depends on provider's limits). Google's limit is 512.
- **pause** (float) – How long to pause in seconds between API calls, which can be increased if you get rate limited.

Returns
G – Graph with *elevation* attributes on the nodes.

Return type
networkx.MultiDiGraph

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

Add *elevation* attributes to all nodes from local raster file(s).

If *filepath* is an iterable 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** (MultiDiGraph) – Graph in same CRS as raster.

- **filepath** (str | Path | Iterable[str | Path]) – The path(s) to the raster file(s) to query.
- **band** (int) – Which raster band to query.
- **cpus** (int | None) – How many CPU cores to use if multiprocessing. If None, use all available. If you are multiprocessing, make sure you protect your entry point: see the Python docs for details.

Returns
G – Graph with *elevation* attributes on the nodes.

Return type
networkx.MultiDiGraph

6.3.5 osmnx.features module

Download and create GeoDataFrames from OpenStreetMap geospatial features.

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*)

Download OSM features 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. This function searches for features using tags. For more details, see: https://wiki.openstreetmap.org/wiki/Map_features

Parameters

- **address** (str) – The address to geocode and use as the center point around which to retrieve the features.
- **tags** (dict[str, bool | str | list[str]]) – Tags for finding elements in the selected area. Results are the union, not intersection of the tags and each result matches at least one tag. The keys are OSM tags (e.g. *building*, *landuse*, *highway*, etc) and the values can be either *True* to retrieve all elements matching the tag, or a string to retrieve a single *tag:value* combination, or a list of strings to retrieve multiple values for the tag. For example, *tags = {'building': True}* would return all buildings in the area. Or, *tags = {'amenity':True, 'landuse':['retail','commercial'], 'highway':'bus_stop'}* would return all amenities, any landuse=retail, any landuse=commercial, and any highway=bus_stop.
- **dist** (float) – Distance in meters from *address* to create a bounding box to query.

Returns
gdf – The features, multi-indexed by element type and OSM ID.

Return type
geopandas.GeoDataFrame

osmnx.features.**features_from_bbox**(*bbox*, *tags*)

Download OSM features within a lat-lon 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. This function searches for features using tags. For more details, see: https://wiki.openstreetmap.org/wiki/Map_features

Parameters

- **bbox** (tuple[float, float, float, float]) – Bounding box as *(left, bottom, right, top)*. Coordinates should be in unprojected latitude-longitude degrees (EPSG:4326).
- **tags** (dict[str, bool | str | list[str]]) – Tags for finding elements in the selected area. Results are the union, not intersection of the tags and each result matches at least one tag. The keys are OSM tags (e.g. *building*, *landuse*, *highway*, etc) and the values can be either *True* to retrieve all elements matching the tag, or a string to retrieve a single *tag:value* combination, or a list of strings to retrieve multiple values for the tag. For example, *tags = {'building': True}* would return all buildings in the area. Or, *tags = {'amenity':True, 'landuse':['retail','commercial'], 'highway':'bus_stop'}* would return all amenities, any landuse=retail, any landuse=commercial, and any highway=bus_stop.

Returns
gdf – The features, multi-indexed by element type and OSM ID.

Return type
geopandas.GeoDataFrame

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

Download OSM features 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 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. This function searches for features using tags. For more details, see: https://wiki.openstreetmap.org/wiki/Map_features

Parameters

- **query** (str | dict[str, str] | list[str | dict[str, str]]) – The query or queries to geocode to retrieve place boundary polygon(s).
- **tags** (dict[str, bool | str | list[str]]) – Tags for finding elements in the selected area. Results are the union, not intersection of the tags and each result matches at least one tag. The keys are OSM tags (e.g. *building*, *landuse*, *highway*, etc) and the values can be either *True* to retrieve all elements matching the tag, or a string to retrieve a single *tag:value* combination, or a list of strings to retrieve multiple values for the tag. For example, *tags = {'building': True}* would return all buildings in the area. Or, *tags = {'amenity':True, 'landuse':['retail','commercial'], 'highway':'bus_stop'}* would return all amenities, any landuse=retail, any landuse=commercial, and any highway=bus_stop.
- **which_result** (int | None | list[int | None]) – Which search result to return. If None, auto-select the first (Multi)Polygon or raise an error if OSM doesn't return one.

Returns
gdf – The features, multi-indexed by element type and OSM ID.

Return type
geopandas.GeoDataFrame

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

Download OSM features 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. This function searches for features using tags. For more details, see: https://wiki.openstreetmap.org/wiki/Map_features

Parameters

- **center_point** (tuple[float, float]) – The *(lat, lon)* center point around which to retrieve the features. Coordinates should be in unprojected latitude-longitude degrees (EPSG:4326).
- **tags** (dict[str, bool | str | list[str]]) – Tags for finding elements in the selected area. Results are the union, not intersection of the tags and each result matches at least one tag. The keys are OSM tags (e.g. *building*, *landuse*, *highway*, etc) and the values can be either *True* to retrieve all elements matching the tag, or a string to retrieve a single *tag:value* combination, or a list of strings to retrieve multiple values for the tag. For example, *tags = {'building': True}* would return all buildings in the area. Or, *tags = {'amenity':True, 'landuse':['retail','commercial'], 'highway':'bus_stop'}* would return all amenities, any landuse=retail, any landuse=commercial, and any highway=bus_stop.
- **dist** (float) – Distance in meters from *center_point* to create a bounding box to query.

Returns

gdf – The features, multi-indexed by element type and OSM ID.

Return type

geopandas.GeoDataFrame

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

Download OSM features 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. This function searches for features using tags. For more details, see: https://wiki.openstreetmap.org/wiki/Map_features

Parameters

- **polygon** (Polygon | MultiPolygon) – The geometry within which to retrieve features. Coordinates should be in unprojected latitude-longitude degrees (EPSG:4326).
- **tags** (dict[str, bool | str | list[str]]) – Tags for finding elements in the selected area. Results are the union, not intersection of the tags and each result matches at least one tag. The keys are OSM tags (e.g. *building*, *landuse*, *highway*, etc) and the values can be either *True* to retrieve all elements matching the tag, or a string to retrieve a single *tag:value* combination, or a list of strings to retrieve multiple values for the tag. For example, *tags = {'building': True}* would return all buildings in the area. Or, *tags = {'amenity':True, 'landuse':['retail','commercial'], 'highway':'bus_stop'}* would return all amenities, any landuse=retail, any landuse=commercial, and any highway=bus_stop.

Returns

gdf – The features, multi-indexed by element type and OSM ID.

Return type

gpd.GeoDataFrame

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

Create a GeoDataFrame of OSM features from data in an OSM XML file.

Because this function creates a GeoDataFrame of features from an OSM XML file that has already been downloaded (i.e., no query is made to the Overpass API), the *polygon* and *tags* arguments are optional. If they are None, filtering will be skipped.

Parameters

- **filepath** (str | Path) – Path to file containing OSM XML data.
- **polygon** (Polygon | MultiPolygon | None) – Spatial boundaries to optionally filter the final GeoDataFrame.
- **tags** (dict[str, bool | str | list[str]] | None) – Query tags to optionally filter the final GeoDataFrame.
- **encoding** (str) – The OSM XML file's character encoding.

Returns
gdf – The features, multi-indexed by element type and OSM ID.

Return type
gpd.GeoDataFrame

6.3.6 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 (str) – The query string to geocode.

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

Return type
tuple[float, float]

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

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 an int 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** (str | dict[str, str] | list[str | dict[str, str]]) – The query string(s) or structured dict(s) to geocode.

- **which_result** (int | None | list[int | None]) – 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 (sorted by importance) 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.

Returns
gdf – GeoDataFrame with one row for each query result.

Return type
geopandas.GeoDataFrame

6.3.7 osmnx.graph module

Download and create graphs from OpenStreetMap data.

Refer to the Getting Started guide for usage limitations.

osmnx.graph.**graph_from_address**(*address*, *dist*, ***, *dist_type='bbox'*, *network_type='all'*, *simplify=True*, *retain_all=False*, *truncate_by_edge=False*, *custom_filter=None*)

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

This function uses filters to query the Overpass API: you can either specify a pre-defined *network_type* or provide your own *custom_filter* with Overpass QL.

Use the *settings* module's *useful_tags_node* and *useful_tags_way* settings to configure which OSM node/way tags are added as graph node/edge attributes. If you want a fully bidirectional network, ensure your *network_type* is in *settings.bidirectional_network_types* before creating your graph. You can also 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 customizations.

Parameters

- **address** (str) – The address to geocode and use as the central point around which to construct the graph.
- **dist** (float) – Retain only those nodes within this many meters of *center_point*, measuring distance according to *dist_type*.
- **dist_type** (str) – {"network", "bbox"} If "bbox", retain only those nodes within a bounding box of *dist*. If "network", retain only those nodes within *dist* network distance from the centermost node.
- **network_type** (str) – {"all", "all_public", "bike", "drive", "drive_service", "walk"} What type of street network to retrieve 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. If False, retain only the largest weakly connected component.
- **truncate_by_edge** (bool) – If True, retain nodes the outside bounding box if at least one of the node's neighbors lies within the bounding box.
- **custom_filter** (str | list[str] | None) – A custom ways filter to be used instead of the *network_type* presets, e.g. *'["power"~"line"]' or '["highway"~"motorway|trunk"]'*. If *str*, the intersection of keys/values will be used, e.g., *'[maxspeed=50][lanes=2]'* will return all ways having both maxspeed of 50 and two lanes. If *list*, the union of the *list* items will be used, e.g., *['[maxspeed=50]', '[lanes=2]']* will return all ways having either maximum speed of 50 or two lanes. Also pass in a *network_type* that is in *settings.bidirectional_network_types* if you want the graph to be fully bidirectional.

Returns
G – The resulting MultiDiGraph.

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_bbox**(*bbox*, *, *network_type='all'*, *simplify=True*, *retain_all=False*, *truncate_by_edge=False*, *custom_filter=None*)

Download and create a graph within a lat-lon bounding box.

This function uses filters to query the Overpass API: you can either specify a pre-defined *network_type* or provide your own *custom_filter* with Overpass QL.

Use the *settings* module's *useful_tags_node* and *useful_tags_way* settings to configure which OSM node/way tags are added as graph node/edge attributes. If you want a fully bidirectional network, ensure your *network_type* is in *settings.bidirectional_network_types* before creating your graph. You can also 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 customizations.

Parameters

- **bbox** (tuple[float, float, float, float]) – Bounding box as *(left, bottom, right, top)*. Coordinates should be in unprojected latitude-longitude degrees (EPSG:4326).
- **network_type** (str) – {"all", "all_public", "bike", "drive", "drive_service", "walk"} What type of street network to retrieve if *custom_filter* is None.
- **simplify** (bool) – If True, simplify graph topology via the *simplify_graph* function.
- **retain_all** (bool) – If True, return the entire graph even if it is not connected. If False, retain only the largest weakly connected component.
- **truncate_by_edge** (bool) – If True, retain nodes the outside bounding box if at least one of the node's neighbors lies within the bounding box.
- **custom_filter** (str | list[str] | None) – A custom ways filter to be used instead of the *network_type* presets, e.g. *'["power"~"line"]' or '["highway"~"motorway|trunk"]'*. If *str*, the intersection of keys/values will be used, e.g., *'[maxspeed=50][lanes=2]'* will return all ways having both maxspeed of 50 and two lanes. If *list*, the union of the *list* items will be used, e.g., *['[maxspeed=50]', '[lanes=2]']* will return all ways having either maximum speed of 50 or two lanes. Also pass in a *network_type* that is in *settings.bidirectional_network_types* if you want the graph to be fully bidirectional.

Returns
G – The resulting MultiDiGraph.

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'*, *simplify=True*, *retain_all=False*, *truncate_by_edge=False*, *which_result=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 *features_from_polygon* function.

This function uses filters to query the Overpass API: you can either specify a pre-defined *network_type* or provide your own *custom_filter* with Overpass QL.

Use the *settings* module's *useful_tags_node* and *useful_tags_way* settings to configure which OSM node/way tags are added as graph node/edge attributes. If you want a fully bidirectional network, ensure your *network_type* is in *settings.bidirectional_network_types* before creating your graph. You can also 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 customizations.

Parameters

- **query** (str | dict[str, str] | list[str | dict[str, str]]) – The query or queries to geocode to retrieve place boundary polygon(s).
- **network_type** (str) – {"all", "all_public", "bike", "drive", "drive_service", "walk"} What type of street network to retrieve 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. If False, retain only the largest weakly connected component.
- **truncate_by_edge** (bool) – If True, retain nodes outside the place boundary polygon(s) if at least one of the node's neighbors lies within the polygon(s).
- **which_result** (int | None | list[int | None]) – Which geocoding result to use. if None, auto-select the first (Multi)Polygon or raise an error if OSM doesn't return one.
- **custom_filter** (str | list[str] | None) – A custom ways filter to be used instead of the *network_type* presets, e.g. *'["power"~"line"]' or '["highway"~"motorway|trunk"]'*. If *str*, the intersection of keys/values will be used, e.g., *'[maxspeed=50][lanes=2]'* will return all ways having both maxspeed of 50 and two lanes. If *list*, the union of the *list* items will be used, e.g., *['[maxspeed=50]', '[lanes=2]']* will return all ways having either maximum speed of 50 or two lanes. Also pass in a *network_type* that is in *settings.bidirectional_network_types* if you want the graph to be fully bidirectional.

Returns

G – The resulting MultiDiGraph.

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*, *, *dist_type='bbox'*, *network_type='all'*, *simplify=True*, *retain_all=False*, *truncate_by_edge=False*, *custom_filter=None*)

Download and create a graph within some distance of a lat-lon point.

This function uses filters to query the Overpass API: you can either specify a pre-defined *network_type* or provide your own *custom_filter* with Overpass QL.

Use the *settings* module's *useful_tags_node* and *useful_tags_way* settings to configure which OSM node/way tags are added as graph node/edge attributes. If you want a fully bidirectional network, ensure your *network_type* is in *settings.bidirectional_network_types* before creating your graph. You can also 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 customizations.

Parameters

- **center_point** (tuple[float, float]) – The *(lat, lon)* center point around which to construct the graph. Coordinates should be in unprojected latitude-longitude degrees (EPSG:4326).
- **dist** (float) – Retain only those nodes within this many meters of *center_point*, measuring distance according to *dist_type*.
- **dist_type** (str) – {"bbox", "network"} If "bbox", retain only those nodes within a bounding box of *dist* length/width. If "network", retain only those nodes within *dist* network distance of the nearest node to *center_point*.
- **network_type** (str) – {"all", "all_public", "bike", "drive", "drive_service", "walk"} What type of street network to retrieve 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. If False, retain only the largest weakly connected component.
- **truncate_by_edge** (bool) – If True, retain nodes the outside bounding box if at least one of the node's neighbors lies within the bounding box.
- **custom_filter** (str | list[str] | None) – A custom ways filter to be used instead of the *network_type* presets, e.g. *'["power"~"line"]' or '["highway"~"motorway|trunk"]'*. If *str*, the intersection of keys/values will be used, e.g., *'[maxspeed=50][lanes=2]'* will return all ways having both maxspeed of 50 and two lanes. If *list*, the union of the *list* items will be used, e.g., *['[maxspeed=50]', '[lanes=2]']* will return all ways having either maximum speed of 50 or two lanes. Also pass in a *network_type* that is in *settings.bidirectional_network_types* if you want the graph to be fully bidirectional.

Returns
G – The resulting MultiDiGraph.

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'*, *simplify=True*, *retain_all=False*, *truncate_by_edge=False*, *custom_filter=None*)

Download and create a graph within the boundaries of a (Multi)Polygon.

This function uses filters to query the Overpass API: you can either specify a pre-defined *network_type* or provide your own *custom_filter* with Overpass QL.

Use the *settings* module's *useful_tags_node* and *useful_tags_way* settings to configure which OSM node/way tags are added as graph node/edge attributes. If you want a fully bidirectional network, ensure your *network_type* is in *settings.bidirectional_network_types* before creating your graph. You can also 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 customizations.

Parameters

- **polygon** (Polygon | MultiPolygon) – The geometry within which to construct the graph. Coordinates should be in unprojected latitude-longitude degrees (EPSG:4326).
- **network_type** (str) – {"all", "all_public", "bike", "drive", "drive_service", "walk"} What type of street network to retrieve 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. If False, retain only the largest weakly connected component.
- **truncate_by_edge** (bool) – If True, retain nodes outside *polygon* if at least one of the node's neighbors lies within *polygon*.
- **custom_filter** (str | list[str] | None) – A custom ways filter to be used instead of the *network_type* presets, e.g. *'["power"~"line"]' or '["highway"~"motorway|trunk"]'*. If *str*, the intersection of keys/values will be used, e.g., *'[maxspeed=50][lanes=2]'* will return all ways having both maxspeed of 50 and two lanes. If *list*, the union of the *list* items will be used, e.g., *['[maxspeed=50]', '[lanes=2]']* will return all ways having either maximum speed of 50 or two lanes. Also pass in a *network_type* that is in *settings.bidirectional_network_types* if you want the graph to be fully bidirectional.

Returns
G – The resulting MultiDiGraph.

Return type
nx.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 an OSM XML file.

Do not load an XML file previously 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.

Use the *settings* module's *useful_tags_node* and *useful_tags_way* settings to configure which OSM node/way tags are added as graph node/edge attributes.

Parameters

- **filepath** (str | Path) – Path to file containing OSM XML data.
- **bidirectional** (bool) – If True, create bidirectional 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. If False, retain only the largest weakly connected component.
- **encoding** (str) – The OSM XML file's character encoding.

Returns
G – The resulting MultiDiGraph.

Return type
networkx.MultiDiGraph

6.3.8 osmnx.io module

File I/O functions to save/load 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** (str | Path | None) – Path to the GraphML file.
- **graphml_str** (str | None) – Valid and decoded string representation of a GraphML file's contents.
- **node_dtypes** (dict[str, Any] | None) – Dict of node attribute names:types to convert values' data types. The type can be a type or a custom string converter function.
- **edge_dtypes** (dict[str, Any] | None) – Dict of edge attribute names:types to convert values' data types. The type can be a type or a custom string converter function.
- **graph_dtypes** (dict[str, Any] | None) – Dict of graph-level attribute names:types to convert values' data types. The type can be a type or a custom string converter function.

Returns
G – The loaded MultiDiGraph.

Return type
networkx.MultiDiGraph

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

Save graph nodes and edges to disk as layers in a GeoPackage file.

Parameters

- **G** (MultiDiGraph) – The graph to save.
- **filepath** (str | Path | None) – Path to the GeoPackage file including extension. If None, use default *settings.data_folder/graph.gpkg*.

- **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.
- **encoding** (str) – The character encoding of the saved GeoPackage file.

Return type
None

osmnx.io.**save_graph_xml**(*G*, *filepath=None*, ***, *way_tag_aggs=None*, *encoding='utf-8'*)

Save graph to disk as an OSM XML file.

This function exists only to allow serialization to the OSM XML format for applications that require it, and has constraints to conform to that. As such, it 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 an OSM XML file that you have downloaded or generated elsewhere, use the *graph.graph_from_xml* function.

Use the *settings* module's *useful_tags_node* and *useful_tags_way* settings to configure which tags your graph is created and saved with. This function merges graph edges such that each OSM way has one entry in the XML output, with the way's nodes topologically sorted. *G* must be unsimplified to save as OSM XML: otherwise, one edge could comprise multiple OSM ways, making it impossible to group and sort edges in way. *G* should also have been created with *ox.settings.all_oneway=True* for this function to behave properly.

Parameters

- **G** (MultiDiGraph) – Unsimplified, unprojected graph to save as an OSM XML file.
- **filepath** (str | Path | None) – Path to the saved file including extension. If None, use default *settings.data_folder/graph.osm*.
- **way_tag_aggs** (dict[str, Any] | None) – Keys are OSM way tag keys and values are aggregation functions (anything accepted as an argument by pandas.agg). Allows user to aggregate graph edge attribute values into single OSM way values. If None, or if some tag's key does not exist in the dict, the way attribute will be assigned the value of the first edge of the way.
- **encoding** (str) – The character encoding of the saved OSM XML file.

Return type
None

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

Save graph to disk as GraphML file.

Parameters

- **G** (MultiDiGraph) – The graph to save as.
- **filepath** (str | Path | None) – Path to the GraphML file including extension. If None, use default *settings.data_folder/graph.graphml*.
- **gephi** (bool) – If True, give each edge a unique key/id for compatibility with Gephi's interpretation of the GraphML specification.
- **encoding** (str) – The character encoding of the saved GraphML file.

Return type
None

6.3.9 osmnx.plot module

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

osmnx.plot.**get_colors**(*n*, ***, *cmap='viridis'*, *start=0*, *stop=1*, *alpha=None*)

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

Parameters

- **n** (int) – How many colors to sample.
- **cmap** (str) – Name of the matplotlib colormap from which to sample the colors.
- **start** (float) – Where to start sampling from the colorspace (from 0 to 1).
- **stop** (float) – Where to end sampling from the colorspace (from 0 to 1).
- **alpha** (float | None) – If *None*, return colors as HTML-like hex triplet "#rrggbb" RGB strings. If *float*, return as "#rrggbbaa" RGBa strings.

Returns
color_list – The sampled colors.

Return type
list[str]

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

Return colors based on edges' numerical attribute values.

Parameters

- **G** (MultiDiGraph) – Input graph.
- **attr** (str) – Name of a node attribute with numerical values.
- **num_bins** (int | None) – If None, linearly map a color to each value. Otherwise, assign values to this many bins then assign a color to each bin.
- **cmap** (str) – Name of the matplotlib colormap from which to choose the colors.
- **start** (float) – Where to start in the colorspace (from 0 to 1).
- **stop** (float) – Where to end in the colorspace (from 0 to 1).
- **na_color** (str) – The color to assign to 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
edge_colors – Labels are *(u, v, k)* edge IDs, values are colors as hex strings.

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*)

Return colors based on nodes' numerical attribute values.

Parameters

- **G** (MultiDiGraph) – Input graph.
- **attr** (str) – Name of a node attribute with numerical values.

- **num_bins** (int | None) – If None, linearly map a color to each value. Otherwise, assign values to this many bins then assign a color to each bin.
- **cmap** (str) – Name of the matplotlib colormap from which to choose the colors.
- **start** (float) – Where to start in the colorspace (from 0 to 1).
- **stop** (float) – Where to end in the colorspace (from 0 to 1).
- **na_color** (str) – The color to assign to 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 – Labels are node IDs, values are colors as hex strings.

Return type
pandas.Series

osmnx.plot.**plot_figure_ground**(*G*, ***, *dist=805*, *street_widths=None*, *default_width=4*, *color='w'*, ***pg_kwargs*)

Plot a figure-ground diagram of a street network.

Parameters

- **G** (MultiDiGraph) – An unprojected graph.
- **dist** (float) – How many meters to extend plot's bounding box from the graph's center point. Default corresponds to a square mile bounding box.
- **street_widths** (dict[str, float]|None) – Dict keys are street types (ie, OSM "highway" tags) and values are the widths to plot them, in pixels.
- **default_width** (float) – Fallback width, in pixels, for any street type not in *street_widths*.
- **color** (str) – The color of the streets.
- ****pg_kwargs** (Any) – Keyword arguments to pass to *plot_graph*.

Returns
fig, ax – The resulting matplotlib figure and axes objects.

Return type
tuple[matplotlib.figure.Figure, matplotlib.axes._axes.Axes]

osmnx.plot.**plot_footprints**(*gdf*, ***, *ax=None*, *figsize=(8, 8)*, *color='orange'*, *edge_color='none'*, *edge_linewidth=0*, *alpha=None*, *bgcolor='#111111'*, *bbox=None*, *show=True*, *close=False*, *save=False*, *filepath=None*, *dpi=600*)

Visualize a GeoDataFrame of geospatial features' footprints.

Parameters

- **gdf** (gpd.GeoDataFrame) – GeoDataFrame of footprints (i.e., Polygons and/or MultiPolygons).
- **ax** (Axes | None) – If not None, plot on this pre-existing axes instance.
- **figsize** (tuple[float, float]) – If *ax* is None, create new figure with size *(width, height)*.
- **color** (str) – Color of the footprints.
- **edge_color** (str) – Color of the footprints' edges.
- **edge_linewidth** (float) – Width of the footprints' edges.

- **alpha** (float | None) – Opacity of the footprints' edges.
- **bgcolor** (str) – Background color of the figure.
- **bbox** (tuple[float, float, float, float] | None) – Bounding box as *(left, bottom, right, top)*. If None, calculate it from the spatial extents of the geometries in *gdf*.
- **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** (str | Path | None) – The path to the file if *save* is True. File format is determined from the extension. If None, save at *settings.imgs_folder/image.png*.
- **dpi** (int) – The resolution of saved file if *save* is True.

Returns
fig, ax – The resulting matplotlib figure and axes objects.

Return type
tuple[Figure, Axes]

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*, *bbox=None*, *show=True*, *close=False*, *save=False*, *filepath=None*, *dpi=300*)

Visualize a graph.

Parameters

- **G** (nx.MultiGraph | nx.MultiDiGraph) – Input graph.
- **ax** (Axes | None) – If not None, plot on this pre-existing axes instance.
- **figsize** (tuple[float, float]) – If *ax* is None, create new figure with size *(width, height)*.
- **bgcolor** (str) – Background color of the figure.
- **node_color** (str | Sequence[str]) – Color(s) of the nodes.
- **node_size** (float | Sequence[float]) – Size(s) of the nodes. If 0, then skip plotting the nodes.
- **node_alpha** (float | None) – Opacity of the nodes. If you passed RGBa values to *node_color*, set *node_alpha=None* to use the alpha channel in *node_color*.
- **node_edgecolor** (str | Iterable[str]) – Color(s) of the nodes' markers' borders.
- **node_zorder** (int) – The zorder to plot nodes. Edges are always 1, so set *node_zorder=0* to plot nodes beneath edges.
- **edge_color** (str | Iterable[str]) – Color(s) of the edges' lines.
- **edge_linewidth** (float | Sequence[float]) – Width(s) of the edges' lines. If 0, then skip plotting the edges.
- **edge_alpha** (float | None) – Opacity of the edges. If you passed RGBa values to *edge_color*, set *edge_alpha=None* to use the alpha channel in *edge_color*.
- **bbox** (tuple[float, float, float, float] | None) – Bounding box as *(left, bottom, right, top)*. If None, calculate it from spatial extents of plotted geometries.
- **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** (str | Path | None) – The path to the file if *save* is True. File format is determined from the extension. If None, save at *settings.imgs_folder/image.png*.
- **dpi** (int) – The resolution of saved file if *save* is True.

Returns
fig, ax – The resulting matplotlib figure and axes objects.

Return type
tuple[Figure, Axes]

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 path along a graph.

Parameters

- **G** (nx.MultiDiGraph) – Input graph.
- **route** (list[int]) – A path of node IDs.
- **route_color** (str) – The color of the route.
- **route_linewidth** (float) – Width of the route's line.
- **route_alpha** (float) – Opacity of the route's line.
- **orig_dest_size** (float) – Size of the origin and destination nodes.
- **ax** (Axes | None) – If not None, plot on this pre-existing axes instance.
- ****pg_kwargs** (Any) – Keyword arguments to pass to *plot_graph*.

Returns
fig, ax – The resulting matplotlib figure and axes objects.

Return type
tuple[Figure, Axes]

osmnx.plot.**plot_graph_routes**(*G*, *routes*, *, *route_colors='r'*, *route_linewidths=4*, ***pgr_kwargs*)

Visualize multiple paths along a graph.

Parameters

- **G** (MultiDiGraph) – Input graph.
- **routes** (Iterable[list[int]]) – Paths of node IDs.
- **route_colors** (str | Iterable[str]) – If string, the one color for all routes. Otherwise, the color for each route.
- **route_linewidths** (float | Iterable[float]) – If float, the one linewidth for all routes. Otherwise, the linewidth for each route.
- ****pgr_kwargs** (Any) – Keyword arguments to pass to *plot_graph_route*.

Returns
fig, ax – The resulting matplotlib figure and axes objects.

Return type
tuple[matplotlib.figure.Figure, matplotlib.axes._axes.Axes]

osmnx.plot.**plot_orientation**(*G*, ***, *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 edge bearings.

Ignores self-loop edges as their bearings are undefined. If *G* is a MultiGraph, all edge bearings will be bidirectional (ie, two reciprocal bearings per undirected edge). If *G* is a MultiDiGraph, all edge bearings will be directional (ie, one bearing per directed edge). See also the *bearings* module.

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

- **G** (nx.MultiGraph | nx.MultiDiGraph) – 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" attribute values less than *min_length*.
- **weight** (str | None) – If not None, weight the edges' bearings by this (non-null) edge attribute.
- **ax** (PolarAxes | None) – If not None, plot on this pre-existing axes instance (must have projection=polar).
- **figsize** (tuple[float, float]) – 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** (str) – Color of the histogram bars.
- **edgecolor** (str) – Color of the histogram bar edges.
- **linewidth** (float) – Width of the histogram bar edges.
- **alpha** (float) – Opacity of the histogram bars.
- **title** (str | None) – The figure's title.
- **title_y** (float) – The y position to place *title*.
- **title_font** (dict[str, Any] | None) – The title's *fontdict* to pass to matplotlib.
- **xtick_font** (dict[str, Any] | None) – The xtick labels' *fontdict* to pass to matplotlib.

Returns

fig, ax – The resulting matplotlib figure and polar axes objects.

Return type

tuple[Figure, PolarAxes]

6.3.10 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 (Any) – The identifier of the coordinate reference system. This 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 coordinate reference system 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 (or UPS) zone given *gdf*'s bounds.

Parameters

- **gdf** (GeoDataFrame) – The GeoDataFrame to be projected.
- **to_crs** (Any | None) – 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**(*geom*, *, *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 geometry to the coordinate reference system 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 (or UPS) zone given *geometry*'s bounds.

Parameters

- **geom** (Geometry) – The geometry to be projected.
- **crs** (Any | None) – The initial CRS of *geometry*. If None, it will be set to *settings.default_crs*.
- **to_crs** (Any | None) – 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
geom_proj, crs – The projected geometry and its new CRS.

Return type
tuple[shapely.Geometry, *Any*]

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 graph to the coordinate reference system 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 (or UPS) zone given *geometry*'s bounds.

Parameters

- **G** (MultiDiGraph) – The graph to be projected.
- **to_crs** (Any | None) – 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

6.3.11 osmnx.routing module

Calculate edge speeds, travel times, and weighted shortest paths.

osmnx.routing.**add_edge_speeds**(*G*, *, *hwy_speeds=None*, *fallback=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

If you wish to set all edge speeds to a single constant value (such as for a walking network), use *nx.set_edge_attributes* to set the *speed_kph* attribute value directly, rather than using this function.

Parameters

- **G** (MultiDiGraph) – Input graph.
- **hwy_speeds** (dict[str, float] | None) – Dict keys are OSM highway types and values are 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 pre-existing speed value of all edges of that highway type.
- **fallback** (float | None) – Default speed value (km per hour) to assign to edges whose highway type did not appear in *hwy_speeds* and had no pre-existing speed attribute values on any edge.
- **agg** (Callable[[Any], Any]) – 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.routing.**add_edge_travel_times**(*G*)

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 (MultiDiGraph) – Input graph.

Returns
G – Graph with *travel_time* attributes on all edges.

Return type
networkx.MultiDiGraph

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** (MultiDiGraph) – Input graph.
- **orig** (int) – Origin node ID.
- **dest** (int) – Destination node ID.
- **k** (int) – Number of shortest paths to solve.
- **weight** (str) – Edge attribute to minimize when solving shortest paths.

Yields
path – The node IDs constituting the next-shortest path.

Return type
Iterator[list[int]]

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

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

Parameters

- **G** (MultiDiGraph) – Input graph.
- **route** (list[int]) – Node IDs constituting the path.
- **weight** (str) – Attribute value to minimize when choosing between parallel edges.

Returns
gdf_edges – The ordered edges in the path.

Return type
geopandas.GeoDataFrame

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

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

Overloads

- **G** (nx.MultiDiGraph), **orig** (int), **dest** (int), **weight** (str), **cpus** (int | None) → list[int] | None
- **G** (nx.MultiDiGraph), **orig** (int), **dest** (int), **cpus** (int | None) → list[int] | None
- **G** (nx.MultiDiGraph), **orig** (int), **dest** (int), **weight** (str) → list[int] | None
- **G** (nx.MultiDiGraph), **orig** (int), **dest** (int) → list[int] | None
- **G** (nx.MultiDiGraph), **orig** (Iterable[int]), **dest** (Iterable[int]), **weight** (str), **cpus** (int | None) → list[list[int] | None]
- **G** (nx.MultiDiGraph), **orig** (Iterable[int]), **dest** (Iterable[int]), **cpus** (int | None) → list[list[int] | None]

- **G** (nx.MultiDiGraph), **orig** (Iterable[int]), **dest** (Iterable[int]), **weight** (str) → list[list[int] | None]
- **G** (nx.MultiDiGraph), **orig** (Iterable[int]), **dest** (Iterable[int]) → list[list[int] | None]

Parameters

- **G** (*networkx.MultiDiGraph*)
- **orig** (*int | Iterable[int]*)
- **dest** (*int | Iterable[int]*)
- **weight** (*str*)
- **cpus** (*int | None*)

Return type
list[int] | None | list[list[int] | None]

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** (`MultiDiGraph`) – Input graph.
- **orig** (`int` | `Iterable[int]`) – Origin node ID(s).
- **dest** (`int` | `Iterable[int]`) – Destination node ID(s).
- **weight** (`str`) – Edge attribute to minimize when solving shortest path.
- **cpus** (`int` | `None`) – How many CPU cores to use if multiprocessing. If None, use all available. If you are multiprocessing, make sure you protect your entry point: see the Python docs for details.

Returns
path – The node IDs constituting the shortest path, or, if *orig* and *dest* are both iterable, then a list of such paths.

Return type
list[int] | None | list[list[int] | None]

6.3.12 osmnx.settings module

Global settings that can be configured by the user.

all_oneway
[bool] Only use if subsequently saving graph to an OSM XML file via the *save_graph_xml* function. If True, forces all ways to be added as one-way ways, preserving the original order of the nodes in the OSM way. This also retains the original OSM way's oneway tag's string value as edge attribute values, rather than converting them to True/False bool values. Default is *False*.

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

cache_folder
[str | Path] Path to folder to save/load HTTP response cache files, if the *use_cache* setting is 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 Overpass response 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
[str | Path] Path to folder to save/load graph files by default. Default is *"./data"*.

default_access
[str] Filter for the OSM "access" tag. 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" tag 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
[str] Default coordinate reference system to set when creating graphs. Default is *"epsg:4326"*.

doh_url_template
[str | None] 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
[str] Endpoint of the Google Maps Elevation API (or equivalent), containing up to two parameters, in order: *locations* and *key*. Default is: *"https://maps.googleapis.com/maps/api/elevation/json?locations={locations}&key={key}"*.
As alternative free examples, the Open Topo Data API would be: *"https://api.opentopodata.org/v1/aster30m?locations={locations}"* and the Open-Elevation API would be: *"https://api.open-elevation.com/api/v1/lookup?locations={locations}"*.

http_accept_language
[str] HTTP header accept-language. Default is *"en"*. Note that Nominatim's default language is "en" and it may sort its results' importance scores differently if a different language is specified.

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

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

imgs_folder
[str | 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
[str] 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
[str] Name of the logger. Default is *"OSMnx"*.

logs_folder
[str | Path] Path to folder in which to save log files. Default is *"./logs"*.

max_query_area_size
[float] 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*.

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

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

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

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
[str] 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.

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

requests_kwargs
[dict[str, Any]] 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 *{}*.

requests_timeout
[int] The timeout interval in seconds for HTTP requests, and (when applicable) for Overpass server to use for executing the query. Default is *180*.

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

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

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

6.3.13 osmnx.simplification module

Simplify, correct, and consolidate spatial graph nodes and edges.

osmnx.simplification.**consolidate_intersections**(*G*, *, *tolerance=10*, *rebuild_graph=True*, *max_length=None*, *dead_ends=False*, *reconnect_edges=True*, *node_attr_aggs=None*)

Consolidate intersections comprising clusters of nearby nodes.

This algorithm is described in the journal article: Boeing, G. 2025. "Topological Graph Simplification Solutions to the Street Intersection Miscount Problem." Transactions in GIS, 29 (3), e70037. https://doi.org/10.1111/tgis.70037

Merges nearby nodes and returns either their centroids or a rebuilt graph with consolidated intersections and reconnected edge geometries. The *tolerance* argument can be a single value applied to all nodes or individual per-node values. It should be adjusted to approximately match street design standards in the specific street network, and you should use a projected graph to work in meaningful and consistent units like meters. Note: *tolerance* represents a per-node buffering radius. For example, to consolidate nodes within 10 meters of each other, use *tolerance=5*.

When *rebuild_graph* is False, it uses a purely geometric (and relatively fast) algorithm to identify "geometrically close" nodes, merge them, and return the merged intersections' centroids. When *rebuild_graph* is 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 "osmid" values. Refer to nodes' "osmid_original" attributes for original "osmid" values. If multiple nodes were merged together, the "osmid_original" attribute is a list of merged nodes' "osmid" values.

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** (nx.MultiDiGraph) – A projected graph.
- **tolerance** (float | dict[int, float]) – Nodes are buffered to this distance (in graph's geometry's units) and subsequent overlaps are dissolved into a single node. If scalar, then that single value will be used for all nodes. If dict (mapping node IDs to individual values), then those values will be used per node and any missing node IDs will not be buffered.
- **rebuild_graph** (bool) – If True, consolidate the nodes topologically, rebuild the graph, and return as MultiDiGraph. Otherwise, consolidate the nodes geometrically and return the consolidated node points as GeoSeries.
- **max_length** (float | None) – Ignore edges longer than this when determining whether nearby nodes are topologically connected. If None, consider all edges. This prevents long edges that loop outside an intersection, such as cloverleaf ramps, from merging their nearby endpoint nodes.
- **dead_ends** (bool) – If False, discard dead-end nodes to return only street intersection points.
- **reconnect_edges** (bool) – If True, reconnect edges (and their geometries) to the consolidated nodes in rebuilt graph, and update the edge length attributes. If False, the returned graph has no edges (which is faster if you just need topologically consolidated intersection counts). Ignored if *rebuild_graph* is not True.
- **node_attr_aggs** (dict[str, Any] | None) – Allows user to aggregate node attributes values when merging nodes. Keys are node attribute names and values are aggregation functions (anything accepted as an argument by *pandas.agg*). Node attributes not in *node_attr_aggs* will contain the unique values across the merged nodes. If None, defaults to *{"elevation": numpy.mean}*.

Returns

Gc or gs – If *rebuild_graph=True*, returns MultiDiGraph with consolidated intersections and

(optionally) reconnected edge geometries. If *rebuild_graph=False*, returns GeoSeries of Points representing the centroids of street intersections.

Return type
nx.MultiDiGraph | gpd.GeoSeries

osmnx.simplification.**simplify_graph**(*G*, *, *node_attrs_include=None*, *edge_attrs_differ=None*, *remove_rings=True*, *track_merged=False*, *edge_attr_aggs=None*)

Simplify a graph's topology by removing interstitial nodes.

This algorithm is described in the journal article: Boeing, G. 2025. "Topological Graph Simplification Solutions to the Street Intersection Miscount Problem." Transactions in GIS, 29 (3), e70037. https://doi.org/10.1111/tgis.70037

This simplifies the graph's topology by removing all nodes that are not intersections or dead-ends, by creating an edge directly between the end points that encapsulate them while retaining the full 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 unique 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.

Use the *node_attrs_include* or *edge_attrs_differ* parameters to relax simplification strictness. For example, *edge_attrs_differ=["osmid"]* will retain every node whose incident edges have different OSM IDs. This lets you keep nodes at elbow two-way intersections (but be aware that sometimes individual blocks have multiple OSM IDs within them too). You could also use this parameter to retain nodes where sidewalks or bike lanes begin/end in the middle of a block. Or for example, *node_attrs_include=["highway"]* will retain every node with a "highway" attribute (regardless of its value), even if it does not represent a street junction.

Parameters

- **G** (`MultiDiGraph`) – Input graph.
- **node_attrs_include** (`Iterable[str] | None`) – Node attribute names for relaxing the strictness of endpoint determination. A node is always an endpoint if it possesses one or more of the attributes in *node_attrs_include*.
- **edge_attrs_differ** (`Iterable[str] | None`) – Edge attribute names for relaxing the strictness of endpoint determination. A node is always an endpoint if its incident edges have different values than each other for any attribute in *edge_attrs_differ*.
- **remove_rings** (`bool`) – If True, remove any graph components that consist only of a single chordless cycle (i.e., an isolated self-contained ring).
- **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.
- **edge_attr_aggs** (`dict[str, Any] | None`) – Allows user to aggregate edge segment attributes when simplifying an edge. Keys are edge attribute names and values are aggregation functions to apply to these attributes when they exist for a set of edges being merged. Edge attributes not in *edge_attr_aggs* will contain the unique values across the merged edge segments. If None, defaults to *{"length": sum, "travel_time": sum}*.

Returns
Gs – Topologically simplified graph, with a new *geometry* attribute on each simplified edge.

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. Due to OSMnx's periphery cleaning when the graph was created, 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** (`MultiDiGraph`) – Input graph.
- **area** (`float | None`) – If not None, calculate density measures and use *area* (in square meters) as the denominator.
- **clean_int_tol** (`float | None`) – 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

`dict[str, Any]` – *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[str, *Any*]

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. Returns None if the edge lengths sum to zero.

Parameters
Gu (MultiGraph) – Undirected input graph.

Returns
circuity_avg – The graph's average undirected edge circuity.

Return type
float | None

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** (MultiDiGraph) – Input graph.
- **nodes** (Iterable[int] | None) – 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[int, int]

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

Calculate graph's total edge length.

Parameters
G (MultiGraph) – Input graph.

Returns
length – Total length (meters) of edges in graph.

Return type
float

osmnx.stats.**intersection_count**(*G*, *, *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** (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 (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 (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 (MultiGraph) – Undirected input graph.

Returns
count – Count of street segments in graph.

Return type
int

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

Retrieve nodes' *street_count* attribute values.

See also the *count_streets_per_node* function for the calculation.

Parameters
G (MultiDiGraph) – Input graph.

Returns
spn – Dictionary with node ID keys and street count values.

Return type
dict[int, int]

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

Calculate graph's average count of streets per node.

Parameters
G (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 (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[int, int]

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

Calculate streets-per-node proportions.

Parameters
G (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[int, float]

6.3.15 osmnx.truncate module

Truncate graph by distance, bounding box, or polygon.

osmnx.truncate.**largest_component**(*G*, *, *strongly=False*)

Return *G*'s largest weakly or strongly connected component as a graph.

Parameters

- **G** (MultiDiGraph) – Input graph.
- **strongly** (bool) – If True, return the largest strongly connected component. Otherwise return the largest weakly connected component.

Returns
G – The largest connected component subgraph of the original graph.

Return type
networkx.MultiDiGraph

osmnx.truncate.**truncate_graph_bbox**(*G*, *bbox*, *, *truncate_by_edge=False*)

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

Parameters

- **G** (MultiDiGraph) – Input graph.
- **bbox** (tuple[float, float, float, float]) – Bounding box as *(left, bottom, right, top)*.
- **truncate_by_edge** (bool) – If True, retain nodes outside bounding box if at least one of node's neighbors is within the bounding box.

Returns
G – The truncated graph.

Return type
networkx.MultiDiGraph

osmnx.truncate.**truncate_graph_dist**(*G*, *source_node*, *dist*, *, *weight='length'*)

Remove from a graph every node beyond some network distance from a node.

This function must calculate shortest path distances between *source_node* and every other graph node, which can be slow on large graphs.

Parameters

- **G** (MultiDiGraph) – Input graph.
- **source_node** (int) – Node from which to measure network distances to all other nodes.
- **dist** (float) – Remove every node in the graph that is greater than *dist* distance (in same units as *weight* attribute) along the network from *source_node*.
- **weight** (str) – Graph edge attribute to use to measure distance.

Returns
G – The truncated graph.

Return type
networkx.MultiDiGraph

osmnx.truncate.**truncate_graph_polygon**(*G*, *polygon*, *, *truncate_by_edge=False*)

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

Parameters

- **G** (nx.MultiDiGraph) – Input graph.
- **polygon** (Polygon | MultiPolygon) – Only retain nodes in graph that lie within this geometry.
- **truncate_by_edge** (bool) – If True, retain nodes outside boundary polygon if at least one of node's neighbors is within the polygon.

Returns
G – The truncated graph.

Return type
nx.MultiDiGraph

6.3.16 osmnx.utils module

General utility functions.

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

Print the OSMnx package's citation information.

Boeing, G. (2025). Modeling and Analyzing Urban Networks and Amenities with OSMnx. Geographical Analysis, 57(4), 567-577. doi:10.1111/gean.70009

Parameters
style (str) – {"apa", "bibtex", "ieee"} The citation format, either APA or BibTeX or IEEE.

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** (str) – The message to log.
- **level** (int | None) – One of the Python *logger.level* constants. If None, set to *settings.log_level* value.
- **name** (str | None) – The name of the logger. If None, set to *settings.log_name* value.
- **filename** (str | None) – The name of the log file, without file extension. If None, set to *settings.log_filename* value.

Return type
None

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

Return current local timestamp as a string.

Parameters

- **style** (str) – {"datetime", "iso8601", "date", "time"} Format the timestamp with this built-in style.
- **template** (str | None) – If not None, format the timestamp with this format string instead of one of the built-in styles.

Returns
timestamp – The current timestamp.

Return type
str

6.3.17 osmnx.utils_geo module

Geospatial utility functions.

osmnx.utils_geo.**bbox_from_point**(*point*, *dist*, *, *project_utm=False*, *return_crs=False*)

Create a bounding box around a (lat, lon) point.

Overloads

- **point** (tuple[float, float]), **dist** (float) → tuple[float, float, float, float]
- **point** (tuple[float, float]), **dist** (float), **return_crs** (Literal[True]) → tuple[float, float, float, float]
- **point** (tuple[float, float]), **dist** (float), **return_crs** (Literal[False]) → tuple[float, float, float, float]
- **point** (tuple[float, float]), **dist** (float), **project_utm** (Literal[True]) → tuple[float, float, float, float]

- **point** (tuple[float, float]), **dist** (float), **project_utm** (Literal[True]), **return_crs** (Literal[True]) → tuple[tuple[float, float, float, float], Any]
- **point** (tuple[float, float]), **dist** (float), **project_utm** (Literal[True]), **return_crs** (Literal[False]) → tuple[float, float, float, float]
- **point** (tuple[float, float]), **dist** (float), **project_utm** (Literal[False]) → tuple[float, float, float, float]
- **point** (tuple[float, float]), **dist** (float), **project_utm** (Literal[False]), **return_crs** (Literal[True]) → tuple[float, float, float, float]
- **point** (tuple[float, float]), **dist** (float), **project_utm** (Literal[False]), **return_crs** (Literal[False]) → tuple[float, float, float, float]

Parameters

- **point** (*tuple[float, float]*)
- **dist** (*float*)
- **project_utm** (*bool*)
- **return_crs** (*bool*)

Return type
tuple[float, float, float, float] | tuple[tuple[float, float, float, float], *Any*]

Create a bounding box some distance (in meters) in each direction (top, bottom, right, and left) from the center point and optionally project it.

Parameters

- **point** (tuple[float, float]) – The *(lat, lon)* center point to create the bounding box around.
- **dist** (float) – 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* is True, then return the projected CRS too.

Returns
bbox or bbox, crs – (left, bottom, right, top) or *((left, bottom, right, top), crs)*.

Return type
tuple[float, float, float, float] | tuple[tuple[float, float, float, float], *Any*]

osmnx.utils_geo.**bbox_to_poly**(*bbox*)

Convert bounding box coordinates to Shapely Polygon.

Parameters
bbox (tuple[float, float, float, float]) – Bounding box as *(left, bottom, right, top)*.

Returns
polygon – The resulting bounding box polygon.

Return type
shapely.Polygon

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

Buffer an unprojected Shapely geometry by some distance in meters.

Parameters

- **geom** (Geometry) – The geometry to be buffered. Coordinates should be in unprojected latitude-longitude degrees (EPSG:4326).
- **dist** (float) – The buffer distance in meters.

Returns
geometry_buff – The (also unprojected) buffered geometry.

Return type
shapely.Geometry

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** (LineString | MultiLineString) – A LineString geometry.
- **dist** (float) – Spacing distance between interpolated points, in same units as *geom*. Smaller values accordingly generate more points.

Yields
point – Interpolated point's *(x, y)* coordinates.

Return type
Iterator[tuple[float, float]]

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** (MultiGraph) – Graph from which to sample points. Should be undirected (to avoid oversampling 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 sampled.

Return type
geopandas.GeoSeries

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_v1 module

Expose the old v1 API for backwards compatibility.

This allows common functionality to be accessed directly via the ox.function_name() shortcut by exposing these functions directly in the package's namespace.

6.4.2 osmnx.bearing module

Calculate graph edge bearings and orientation entropy.

osmnx.bearing.**_bearings_distribution**(*G*, *num_bins*, *min_length*, *weight*)

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

- **G** (nx.MultiGraph | nx.MultiDiGraph) – Unprojected graph with *bearing* attributes on each edge.
- **num_bins** (int) – Number of bins for the bearing histogram.
- **min_length** (float) – Ignore edges with *length* attributes less than *min_length*. Useful to ignore the noise of many very short edges.
- **weight** (str | None) – If None, apply equal weight for each bearing. Otherwise, weight edges' bearings by this (non-null) edge attribute. For example, if "length" is provided, each edge's bearing observation will be weighted by its "length" attribute value.

Returns

bin_counts, bin_centers – Counts of bearings per bin and the bins' centers in degrees. Both arrays are of length *num_bins*.

Return type

tuple[npt.NDArray[np.floating], npt.NDArray[np.floating]]

osmnx.bearing.**_extract_edge_bearings**(*G*, *min_length*, *weight*)

Extract graph's edge bearings.

Ignores self-loop edges as their bearings are undefined. If *G* is a MultiGraph, all edge bearings will be bidirectional (ie, two reciprocal bearings per undirected edge). If *G* is a MultiDiGraph, all edge bearings will be directional (ie, one bearing per directed edge). For example, if an undirected edge has a bearing of 90 degrees then we will record bearings of both 90 degrees and 270 degrees for this edge.

Parameters

- **G** (nx.MultiGraph | nx.MultiDiGraph) – 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** (str | None) – If None, apply equal weight for each bearing. Otherwise, weight edges' bearings by this (non-null) edge attribute. For example, if "length" is provided, each edge's bearing observation will be weighted by its "length" attribute value.

Returns

bearings, weights – The edge bearings of *G* and their corresponding weights.

Return type

tuple[npt.NDArray[np.float64], npt.NDArray[np.float64]]

osmnx.bearing.**add_edge_bearings**(*G*)

Calculate and 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 *bearing* 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 (MultiDiGraph) – Unprojected graph.

Returns
G – Graph with *bearing* attributes on the edges.

Return type
networkx.MultiDiGraph

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

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

Overloads

- **lat1** (float), **lon1** (float), **lat2** (float), **lon2** (float) → float
- **lat1** (npt.NDArray[np.float64]), **lon1** (npt.NDArray[np.float64]), **lat2** (npt.NDArray[np.float64]), **lon2** (npt.NDArray[np.float64]) → npt.NDArray[np.float64]

Parameters

- **lat1** (*float | npt.NDArray[np.float64]*)
- **lon1** (*float | npt.NDArray[np.float64]*)
- **lat2** (*float | npt.NDArray[np.float64]*)
- **lon2** (*float | npt.NDArray[np.float64]*)

Return type
float | npt.NDArray[np.float64]

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

Parameters

- **lat1** (float | npt.NDArray[np.float64]) – First point's latitude coordinate(s).
- **lon1** (float | npt.NDArray[np.float64]) – First point's longitude coordinate(s).
- **lat2** (float | npt.NDArray[np.float64]) – Second point's latitude coordinate(s).
- **lon2** (float | npt.NDArray[np.float64]) – Second point's longitude coordinate(s).

Returns
bearing – The bearing(s) in decimal degrees.

Return type
float | npt.NDArray[np.float64]

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

Calculate graph's orientation entropy.

Orientation entropy is the Shannon entropy of the graphs' edges' bearings across evenly spaced bins. Ignores self-loop edges as their bearings are undefined. If *G* is a MultiGraph, all edge bearings will be bidirectional (ie, two reciprocal bearings per undirected edge). If *G* is a MultiDiGraph, all edge bearings will be directional (ie, one bearing per directed edge).

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

- **G** (nx.MultiGraph | nx.MultiDiGraph) – 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** (str | None) – If None, apply equal weight for each bearing. Otherwise, weight edges' bearings by this (non-null) edge attribute. For example, if "length" is provided, each edge's bearing observation will be weighted by its "length" attribute value.

Returns
entropy – The orientation entropy of *G*.

Return type
float

6.4.3 osmnx.convert module

Convert spatial graphs to/from different data types.

osmnx.convert.**_is_duplicate_edge**(*data1*, *data2*)

Check if two graph edge data dicts have the same *osmid* and *geometry*.

Parameters

- **data1** (dict[str, Any]) – The first edge's attribute data.
- **data2** (dict[str, Any]) – The second edge's attribute data.

Returns
is_dupe – True if *osmid* and *geometry* are the same, otherwise False.

Return type
bool

osmnx.convert.**_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** (LineString) – The first LineString geometry.
- **ls2** (LineString) – The second LineString geometry.

Returns
is_same – True if geometries are the same in either direction, otherwise False.

Return type
bool

osmnx.convert.**_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 (MultiDiGraph) – Input graph.

Returns
G – Graph with incremented keys where needed.

Return type
networkx.MultiDiGraph

osmnx.convert.**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 network 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** (GeoDataFrame) – GeoDataFrame of graph nodes uniquely indexed by *osmid*.
- **gdf_edges** (GeoDataFrame) – GeoDataFrame of graph edges uniquely multi-indexed by *(u, v, key)*.
- **graph_attrs** (dict[str, Any] | None) – The new *G.graph* attribute dictionary. If None, use *gdf_edges*'s CRS as the only graph-level attribute (*gdf_edges* must have its *crs* attribute set).

Returns
G – The converted MultiDiGraph.

Return type
networkx.MultiDiGraph

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

Convert a MultiGraph or MultiDiGraph to node and/or edge GeoDataFrames.

Overloads

- **G** (nx.MultiGraph | nx.MultiDiGraph), **node_geometry** (bool), **fill_edge_geometry** (bool) → tuple[gpd.GeoDataFrame, gpd.GeoDataFrame]
- **G** (nx.MultiGraph | nx.MultiDiGraph), **nodes** (Literal[True]), **edges** (Literal[True]), **node_geometry** (bool), **fill_edge_geometry** (bool) → tuple[gpd.GeoDataFrame, gpd.GeoDataFrame]
- **G** (nx.MultiGraph | nx.MultiDiGraph), **nodes** (Literal[True]), **edges** (Literal[False]), **node_geometry** (bool), **fill_edge_geometry** (bool) → gpd.GeoDataFrame
- **G** (nx.MultiGraph | nx.MultiDiGraph), **nodes** (Literal[False]), **edges** (Literal[True]), **node_geometry** (bool), **fill_edge_geometry** (bool) → gpd.GeoDataFrame
- **G** (nx.MultiGraph | nx.MultiDiGraph), **edges** (Literal[True]), **node_geometry** (bool), **fill_edge_geometry** (bool) → tuple[gpd.GeoDataFrame, gpd.GeoDataFrame]
- **G** (nx.MultiGraph | nx.MultiDiGraph), **edges** (Literal[False]), **node_geometry** (bool), **fill_edge_geometry** (bool) → gpd.GeoDataFrame

- **G** (nx.MultiGraph | nx.MultiDiGraph), **nodes** (Literal[True]), **edges** (bool), **node_geometry** (bool), **fill_edge_geometry** (bool) → tuple[gpd.GeoDataFrame, gpd.GeoDataFrame]
- **G** (nx.MultiGraph | nx.MultiDiGraph), **nodes** (Literal[False]), **edges** (bool), **node_geometry** (bool), **fill_edge_geometry** (bool) → gpd.GeoDataFrame

Parameters

- **G** (*nx.MultiGraph | nx.MultiDiGraph*)
- **nodes** (*bool*)
- **edges** (*bool*)
- **node_geometry** (*bool*)
- **fill_edge_geometry** (*bool*)

Return type
gpd.GeoDataFrame | tuple[gpd.GeoDataFrame, gpd.GeoDataFrame]

This function is the inverse of *graph_from_gdfs*.

Parameters

- **G** (nx.MultiGraph | nx.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 missing edge geometry fields using endpoint nodes' coordinates to create a LineString.

Returns
gdf_nodes or gdf_edges or (gdf_nodes, gdf_edges) – *gdf_nodes* is indexed by *osmid* and *gdf_edges* is multi-indexed by *(u, v, key)* following normal MultiGraph/MultiDiGraph structure.

Return type
gpd.GeoDataFrame | tuple[gpd.GeoDataFrame, gpd.GeoDataFrame]

`osmnx.convert.to_digraph`(*G*, *, *weight='length'*)

Convert MultiDiGraph to DiGraph.

Chooses between parallel edges by minimizing *weight* attribute value. See also *to_undirected* to convert MultiDiGraph to MultiGraph.

Parameters

- **G** (`MultiDiGraph`) – Input graph.
- **weight** (`str`) – Attribute value to minimize when choosing between parallel edges.

Returns
D – The converted DiGraph.

Return type
networkx.DiGraph

osmnx.convert.**to_undirected**(*G*)

Convert MultiDiGraph to undirected MultiGraph.

This function has a limited use case: it allows you to create a MultiGraph for use with functions/algorithms that only accept a MultiGraph object. Rather, if you want a fully bidirectional graph (such as for a walking network), configure the *settings* module's *bidirectional_network_types* before creating your graph to generate a fully bidirectional MultiDiGraph.

This function maintains parallel edges only if their geometries differ. See also *to_digraph* to convert MultiDiGraph to DiGraph.

Parameters
G (MultiDiGraph) – Input graph.

Returns
Gu – The converted MultiGraph.

Return type
networkx.MultiGraph

osmnx.convert.**validate_features_gdf**(*gdf*)

Validate that features GeoDataFrame satisfies OSMnx expectations.

Raises a *ValidationError* if validation fails.

Parameters
gdf (GeoDataFrame) – GeoDataFrame of features uniquely multi-indexed by *(element_type, osmid)*.

Return type
None

osmnx.convert.**validate_graph**(*G*, *, *strict=True*)

Validate that a graph object satisfies OSMnx expectations.

Raises *ox._errors.GraphValidationError* if validation fails.

Parameters

- **G** (MultiDiGraph) – The input graph.
- **strict** (bool) – If *True*, enforce optional rules in addition to required rules. These optional rules primarily enforce expected attribute data types.

Return type
None

osmnx.convert.**validate_node_edge_gdfs**(*gdf_nodes*, *gdf_edges*, *, *strict=True*)

Validate that node/edge GeoDataFrames can be converted to a MultiDiGraph.

Raises a *ValidationError* if validation fails.

Parameters

- **gdf_nodes** (GeoDataFrame) – GeoDataFrame of graph nodes uniquely indexed by *osmid*.
- **gdf_edges** (GeoDataFrame) – GeoDataFrame of graph edges uniquely multi-indexed by *(u, v, key)*.
- **strict** (bool) – If *True*, elevate warnings to errors.

Return type
None

6.4.4 osmnx.distance module

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

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

Calculate and add *length* attribute (in meters) to each edge.

Vectorized function to calculate great-circle distance between each edge's incident nodes. Ensure graph is unprojected and unsimplified to calculate 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 (potentially) curvilinear geometry. If you wish to calculate edge lengths later, note that you will be calculating straight-line distances which necessarily ignore the curvilinear geometry. Thus 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** (MultiDiGraph) – Unprojected and unsimplified input graph.
- **edges** (Iterable[tuple[int, int, int]] | None) – The subset of edges to add *length* attributes to, as *(u, v, k)* tuples. If None, add lengths to all edges.

Returns
G – Graph with *length* attributes on the edges.

Return type
networkx.MultiDiGraph

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

Calculate Euclidean distances between pairs of points.

Overloads

- **y1** (float), **x1** (float), **y2** (float), **x2** (float) → float
- **y1** (npt.NDArray[np.float64]), **x1** (npt.NDArray[np.float64]), **y2** (npt.NDArray[np.float64]), **x2** (npt.NDArray[np.float64]) → npt.NDArray[np.float64]

Parameters

- **y1** (*float | npt.NDArray[np.float64]*)
- **x1** (*float | npt.NDArray[np.float64]*)
- **y2** (*float | npt.NDArray[np.float64]*)
- **x2** (*float | npt.NDArray[np.float64]*)

Return type
float | npt.NDArray[np.float64]

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 | npt.NDArray[np.float64]) – First point's y coordinate(s).
- **x1** (float | npt.NDArray[np.float64]) – First point's x coordinate(s).
- **y2** (float | npt.NDArray[np.float64]) – Second point's y coordinate(s).
- **x2** (float | npt.NDArray[np.float64]) – Second point's x coordinate(s).

Returns
dist – Distance from each *(x1, y1)* point to each *(x2, y2)* point in same units as the points' coordinates.

Return type
float | npt.NDArray[np.float64]

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

Calculate great-circle distances between pairs of points.

Overloads

- **lat1** (float), **lon1** (float), **lat2** (float), **lon2** (float) → float
- **lat1** (float), **lon1** (float), **lat2** (float), **lon2** (float), **earth_radius** (float) → float
- **lat1** (npt.NDArray[np.float64]), **lon1** (npt.NDArray[np.float64]), **lat2** (npt.NDArray[np.float64]), **lon2** (npt.NDArray[np.float64]) → npt.NDArray[np.float64]
- **lat1** (npt.NDArray[np.float64]), **lon1** (npt.NDArray[np.float64]), **lat2** (npt.NDArray[np.float64]), **lon2** (npt.NDArray[np.float64]), **earth_radius** (float) → npt.NDArray[np.float64]

Parameters

- **lat1** (*float | npt.NDArray[np.float64]*)
- **lon1** (*float | npt.NDArray[np.float64]*)
- **lat2** (*float | npt.NDArray[np.float64]*)
- **lon2** (*float | npt.NDArray[np.float64]*)
- **earth_radius** (*float*)

Return type
float | npt.NDArray[np.float64]

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 | npt.NDArray[np.float64]) – First point's latitude coordinate(s).
- **lon1** (float | npt.NDArray[np.float64]) – First point's longitude coordinate(s).
- **lat2** (float | npt.NDArray[np.float64]) – Second point's latitude coordinate(s).
- **lon2** (float | npt.NDArray[np.float64]) – Second point's longitude coordinate(s).
- **earth_radius** (float) – Earth's radius in units in which distance will be returned (default represents meters).

Returns
dist – Distance from each *(lat1, lon1)* point to each *(lat2, lon2)* point in units of *earth_radius*.

Return type
float | npt.NDArray[np.float64]

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

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

Overloads

- **G** (nx.MultiDiGraph), **X** (float), **Y** (float) → tuple[int, int, int]

- **G** (nx.MultiDiGraph), **X** (float), **Y** (float), **return_dist** (Literal[False]) → tuple[int, int, int]
- **G** (nx.MultiDiGraph), **X** (float), **Y** (float), **return_dist** (Literal[True]) → tuple[tuple[int, int, int], float]
- **G** (nx.MultiDiGraph), **X** (Iterable[float]), **Y** (Iterable[float]) → npt.NDArray[np.int64]
- **G** (nx.MultiDiGraph), **X** (Iterable[float]), **Y** (Iterable[float]), **return_dist** (Literal[False]) → npt.NDArray[np.int64]
- **G** (nx.MultiDiGraph), **X** (Iterable[float]), **Y** (Iterable[float]), **return_dist** (Literal[True]) → tuple[npt.NDArray[np.int64], npt.NDArray[np.float64]]

Parameters

- **G** (*nx.MultiDiGraph*)
- **X** (*float | Iterable[float]*)
- **Y** (*float | Iterable[float]*)
- **return_dist** (*bool*)

Return type

tuple[int, int, int] | npt.NDArray[np.int64] | tuple[tuple[int, int, int], float] | tuple[npt.NDArray[np.int64], npt.NDArray[np.float64]]

If *X* and *Y* are single coordinate values, this function will return the nearest edge to that point. If *X* and *Y* are iterables of coordinate values, it will return the nearest edge to each point.

This function is vectorized: if you have many points to search for, pass them in one call as numpy arrays (avoid using loops) to maximize runtime speed. It 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 projected points.

Parameters

- **G** (nx.MultiDiGraph) – Graph in which to find nearest edges.
- **X** (float | Iterable[float]) – The points' x (longitude) coordinates, in same CRS/units as graph and containing no nulls.
- **Y** (float | Iterable[float]) – The points' y (latitude) coordinates, in same CRS/units as graph and containing no nulls.
- **return_dist** (bool) – If True, optionally also return the distance(s) between point(s) and nearest edge(s), in same units as graph and points.

Returns

ne or (ne, dist) – Nearest edge ID(s) as *(u, v, k)* tuples, or optionally a tuple of ID(s) and distance(s) between each point and its nearest edge.

Return type

tuple[int, int, int] | npt.NDArray[np.int64] | tuple[tuple[int, int, int], float] | tuple[npt.NDArray[np.int64], npt.NDArray[np.float64]]

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

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

Overloads

- **G** (nx.MultiDiGraph), **X** (float), **Y** (float) → int
- **G** (nx.MultiDiGraph), **X** (float), **Y** (float), **return_dist** (Literal[False]) → int
- **G** (nx.MultiDiGraph), **X** (float), **Y** (float), **return_dist** (Literal[True]) → tuple[npt.NDArray[np.int64], npt.NDArray[np.float64]]

- **G** (nx.MultiDiGraph), **X** (Iterable[float]), **Y** (Iterable[float]) → npt.NDArray[np.int64]
- **G** (nx.MultiDiGraph), **X** (Iterable[float]), **Y** (Iterable[float]), **return_dist** (Literal[False]) → npt.NDArray[np.int64]
- **G** (nx.MultiDiGraph), **X** (Iterable[float]), **Y** (Iterable[float]), **return_dist** (Literal[True]) → tuple[npt.NDArray[np.int64], npt.NDArray[np.float64]]

Parameters

- **G** (*nx.MultiDiGraph*)
- **X** (*float | Iterable[float]*)
- **Y** (*float | Iterable[float]*)
- **return_dist** (*bool*)

Return type
int | npt.NDArray[np.int64] | tuple[int, float] | tuple[npt.NDArray[np.int64], npt.NDArray[np.float64]]

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

This function is vectorized: if you have many points to search for, pass them in one call as numpy arrays (avoid using loops) to maximize runtime speed. If the graph is projected, it uses a k-d tree for Euclidean nearest neighbor search, which requires that scipy is installed as an optional dependency. If the graph is unprojected, it uses a ball tree for haversine nearest neighbor search, which requires that scikit-learn is installed as an optional dependency.

Parameters

- **G** (nx.MultiDiGraph) – Graph in which to find nearest nodes.
- **X** (float | Iterable[float]) – The points' x (longitude) coordinates, in same CRS/units as graph and containing no nulls.
- **Y** (float | Iterable[float]) – The points' y (latitude) coordinates, in same CRS/units as graph and containing no nulls.
- **return_dist** (bool) – If True, optionally also return the distance(s) between point(s) and nearest node(s).

Returns
nn or (nn, dist) – Nearest node ID(s) or optionally a tuple of ID(s) and distance(s) between each point and its nearest node.

Return type
int | npt.NDArray[np.int64] | tuple[int, float] | tuple[npt.NDArray[np.int64], npt.NDArray[np.float64]]

6.4.5 osmnx.elevation module

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

osmnx.elevation.**_build_vrt_file**(*raster_paths*)

Build a virtual raster file compositing multiple individual raster files.

See also https://gdal.org/en/stable/drivers/raster/vrt.html

Parameters
raster_paths (Iterable[str | Path]) – The paths to the raster files.

Returns
vrt_path – The path to the VRT file.

Return type
Path

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

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

Parameters

- **url** (str) – URL of API endpoint, populated with request data.
- **pause** (float) – How long to pause in seconds before request.

Returns
response_json – The elevation API's response.

Return type
dict[str, *Any*]

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

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

Parameters

- **nodes** (DataFrame) – DataFrame indexed by node ID and with two columns representing x and y coordinates.
- **filepath** (str | Path) – Path to the raster file or VRT to query.
- **band** (int) – Which raster band to query.

Returns
nodes_values – Zip of node IDs and corresponding raster values.

Return type
Iterable[tuple[int, *Any*]]

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

Calculate and add *grade* attributes to all graph edges.

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

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

Parameters

- **G** (MultiDiGraph) – Graph with *elevation* node attributes.
- **add_absolute** (bool) – If True, also add absolute value of grade as *grade_abs* attribute.

Returns
G – Graph with *grade* (and optionally *grade_abs*) attributes on the edges.

Return type
networkx.MultiDiGraph

osmnx.elevation.**add_node_elevations_google**(*G*, *, *api_key=None*, *batch_size=512*, *pause=0*)

Add *elevation* (meters) attributes to all nodes using a web API.

By default this uses the Google Maps Elevation API, but you could instead use any equivalent API with the same interface and response format (such as the Open Topo Data API or the Open-Elevation API) via the *settings* module's *elevation_url_template*. Adjust the *batch_size* and *pause* arguments as needed for the provider. The Google

Maps Elevation API requires an API key but other providers may not. You can find more information about the Google Maps Elevation API interface and format at: https://developers.google.com/maps/documentation/elevation

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

Parameters

- **G** (`MultiDiGraph`) – Graph to add elevation data to.
- **api_key** (`str | None`) – 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 request (depends on provider's limits). Google's limit is 512.
- **pause** (`float`) – How long to pause in seconds between API calls, which can be increased if you get rate limited.

Returns
G – Graph with *elevation* attributes on the nodes.

Return type
networkx.MultiDiGraph

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

Add *elevation* attributes to all nodes from local raster file(s).

If *filepath* is an iterable 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** (`MultiDiGraph`) – Graph in same CRS as raster.
- **filepath** (`str | Path | Iterable[str | Path]`) – The path(s) to the raster file(s) to query.
- **band** (`int`) – Which raster band to query.
- **cpus** (`int | None`) – How many CPU cores to use if multiprocessing. If None, use all available. If you are multiprocessing, make sure you protect your entry point: see the Python docs for details.

Returns
G – Graph with *elevation* attributes on the nodes.

Return type
networkx.MultiDiGraph

6.4.6 osmnx._errors module

Define custom errors and exceptions.

exception `osmnx._errors.`**CacheOnlyInterruptError**

Exception for *settings.cache_only_mode=True* interruption.

exception `osmnx._errors.`**GraphSimplificationError**

Exception for a problem with graph simplification.

exception `osmnx._errors.`**InsufficientResponseError**

Exception for empty or too few results in server response.

exception osmnx._errors.**ResponseStatusCodeError**

Exception for an unhandled server response status code.

exception osmnx._errors.**ValidationError**

Exception for failed graph or GeoDataFrame validation.

6.4.7 osmnx.features module

Download and create GeoDataFrames from OpenStreetMap geospatial features.

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.**_build_relation_geometry**(*members*, *way_geoms*)

Build a relation's geometry from its constituent member ways' geometries.

OSM represents simple polygons as closed ways (see *_build_way_geometry*), but it uses relations to represent multipolygons (with or without holes) and polygons with holes. For the former, the relation contains multiple members with role "outer". For the latter, the relation contains at least one member with role "outer" representing the shell(s), and at least one member with role "inner" representing the hole(s). For documentation, see https://wiki.openstreetmap.org/wiki/Relation:multipolygon

Parameters

- **members** (list[dict[str, Any]]) – The members constituting the relation.
- **way_geoms** (dict[int, LineString | Polygon]) – Keyed by OSM way ID with values of their geometries.

Returns

geometry – The relation's geometry.

Return type

Polygon | MultiPolygon

osmnx.features.**_build_way_geometry**(*way_id*, *way_nodes*, *way_tags*, *node_coords*)

Build a way's geometry from its constituent nodes' coordinates.

A way can be a LineString (open or closed way) or a Polygon (closed way) but multi-geometries and polygons with holes are represented as relations. See documentation: https://wiki.openstreetmap.org/wiki/Way#Types_of_ways

Parameters

- **way_id** (int) – The way's OSM ID.
- **way_nodes** (list[int]) – The way's constituent nodes.
- **way_tags** (dict[str, Any]) – The way's tags.
- **node_coords** (dict[int, tuple[float, float]]) – Keyed by OSM node ID with values of *(lat, lon)* coordinate tuples.

Returns

geometry – The way's geometry.

Return type
LineString | Polygon

osmnx.features._create_gdf(*response_jsons*, *polygon*, *tags*)

Convert Overpass API JSON responses to a GeoDataFrame of features.

Parameters

- **response_jsons** (Iterable[dict[str, Any]]) – Iterable of Overpass API JSON responses.
- **polygon** (Polygon | MultiPolygon) – Spatial boundaries to optionally filter the final GeoDataFrame.
- **tags** (dict[str, bool | str | list[str]]) – Query tags to optionally filter the final GeoDataFrame.

Returns
gdf – GeoDataFrame of features with tags and geometry columns.

Return type
gpd.GeoDataFrame

osmnx.features._filter_features(*gdf*, *polygon*, *tags*)

Filter features GeoDataFrame by spatial boundaries and query tags.

If the *polygon* and *tags* arguments are empty objects, the final GeoDataFrame will not be filtered accordingly.

Parameters

- **gdf** (gpd.GeoDataFrame) – Original GeoDataFrame of features.
- **polygon** (Polygon | MultiPolygon) – If not empty, the spatial boundaries to filter the GeoDataFrame.
- **tags** (dict[str, bool | str | list[str]]) – If not empty, the query tags to filter the GeoDataFrame.

Returns
gdf – Filtered GeoDataFrame of features.

Return type
gpd.GeoDataFrame

osmnx.features._process_features(*elements*, *query_tag_keys*)

Convert node/way/relation elements into features with geometries.

Parameters

- **elements** (list[dict[str, Any]]) – The node/way/relation elements retrieved from the server.
- **query_tag_keys** (set[str]) – The keys of the tags used to query for matching features.

Returns
features – The features with geometries.

Return type
list[dict[str, *Any*]]

osmnx.features._remove_polygon_holes(*outer_polygons*, *inner_polygons*)

Subtract inner holes from outer polygons.

This allows possible island polygons within a larger polygon's holes.

Parameters

- **outer_polygons** (list[Polygon]) – Polygons, including possible islands within a larger polygon's holes.

- **inner_polygons** (list[Polygon]) – Inner holes to subtract from the outer polygons that contain them.

Returns
geometry – The geometry minus inner holes.

Return type
Polygon | MultiPolygon

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

Download OSM features 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. This function searches for features using tags. For more details, see: https://wiki.openstreetmap.org/wiki/Map_features

Parameters

- **address** (str) – The address to geocode and use as the center point around which to retrieve the features.
- **tags** (dict[str, bool | str | list[str]]) – Tags for finding elements in the selected area. Results are the union, not intersection of the tags and each result matches at least one tag. The keys are OSM tags (e.g. *building*, *landuse*, *highway*, etc) and the values can be either *True* to retrieve all elements matching the tag, or a string to retrieve a single *tag:value* combination, or a list of strings to retrieve multiple values for the tag. For example, *tags = {'building': True}* would return all buildings in the area. Or, *tags = {'amenity':True, 'landuse':['retail','commercial'], 'highway':'bus_stop'}* would return all amenities, any landuse=retail, any landuse=commercial, and any highway=bus_stop.
- **dist** (float) – Distance in meters from *address* to create a bounding box to query.

Returns
gdf – The features, multi-indexed by element type and OSM ID.

Return type
geopandas.GeoDataFrame

osmnx.features.**features_from_bbox**(*bbox*, *tags*)

Download OSM features within a lat-lon 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. This function searches for features using tags. For more details, see: https://wiki.openstreetmap.org/wiki/Map_features

Parameters

- **bbox** (tuple[float, float, float, float]) – Bounding box as *(left, bottom, right, top)*. Coordinates should be in unprojected latitude-longitude degrees (EPSG:4326).
- **tags** (dict[str, bool | str | list[str]]) – Tags for finding elements in the selected area. Results are the union, not intersection of the tags and each result matches at least one tag. The keys are OSM tags (e.g. *building*, *landuse*, *highway*, etc) and the values can be either *True* to retrieve all elements matching the tag, or a string to retrieve a single *tag:value* combination, or a list of strings to retrieve multiple values for the tag. For example, *tags = {'building': True}* would return all buildings in the area. Or, *tags = {'amenity':True, 'landuse':['retail','commercial'], 'highway':'bus_stop'}* would return all amenities, any landuse=retail, any landuse=commercial, and any highway=bus_stop.

Returns
gdf – The features, multi-indexed by element type and OSM ID.

Return type
geopandas.GeoDataFrame

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

Download OSM features 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 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. This function searches for features using tags. For more details, see: https://wiki.openstreetmap.org/wiki/Map_features

Parameters

- **query** (str | dict[str, str] | list[str | dict[str, str]]) – The query or queries to geocode to retrieve place boundary polygon(s).
- **tags** (dict[str, bool | str | list[str]]) – Tags for finding elements in the selected area. Results are the union, not intersection of the tags and each result matches at least one tag. The keys are OSM tags (e.g. *building*, *landuse*, *highway*, etc) and the values can be either *True* to retrieve all elements matching the tag, or a string to retrieve a single *tag:value* combination, or a list of strings to retrieve multiple values for the tag. For example, *tags = {'building': True}* would return all buildings in the area. Or, *tags = {'amenity':True, 'landuse':['retail','commercial'], 'highway':'bus_stop'}* would return all amenities, any landuse=retail, any landuse=commercial, and any highway=bus_stop.
- **which_result** (int | None | list[int | None]) – Which search result to return. If None, auto-select the first (Multi)Polygon or raise an error if OSM doesn't return one.

Returns
gdf – The features, multi-indexed by element type and OSM ID.

Return type
geopandas.GeoDataFrame

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

Download OSM features 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. This function searches for features using tags. For more details, see: https://wiki.openstreetmap.org/wiki/Map_features

Parameters

- **center_point** (tuple[float, float]) – The *(lat, lon)* center point around which to retrieve the features. Coordinates should be in unprojected latitude-longitude degrees (EPSG:4326).
- **tags** (dict[str, bool | str | list[str]]) – Tags for finding elements in the selected area. Results are the union, not intersection of the tags and each result matches at least one tag. The keys are OSM tags (e.g. *building*, *landuse*, *highway*, etc) and the values can be either *True* to retrieve all elements matching the tag, or a string to retrieve a single *tag:value* combination, or a list of strings to retrieve multiple values for the tag. For example, *tags = {'building': True}* would return all buildings in the area. Or, *tags = {'amenity':True,*

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

- **dist** (float) – Distance in meters from *center_point* to create a bounding box to query.

Returns
gdf – The features, multi-indexed by element type and OSM ID.

Return type
geopandas.GeoDataFrame

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

Download OSM features 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. This function searches for features using tags. For more details, see: https://wiki.openstreetmap.org/wiki/Map_features

Parameters

- **polygon** (Polygon | MultiPolygon) – The geometry within which to retrieve features. Coordinates should be in unprojected latitude-longitude degrees (EPSG:4326).
- **tags** (dict[str, bool | str | list[str]]) – Tags for finding elements in the selected area. Results are the union, not intersection of the tags and each result matches at least one tag. The keys are OSM tags (e.g. *building*, *landuse*, *highway*, etc) and the values can be either *True* to retrieve all elements matching the tag, or a string to retrieve a single *tag:value* combination, or a list of strings to retrieve multiple values for the tag. For example, *tags = {'building': True}* would return all buildings in the area. Or, *tags = {'amenity':True, 'landuse':['retail','commercial'], 'highway':'bus_stop'}* would return all amenities, any landuse=retail, any landuse=commercial, and any highway=bus_stop.

Returns
gdf – The features, multi-indexed by element type and OSM ID.

Return type
gpd.GeoDataFrame

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

Create a GeoDataFrame of OSM features from data in an OSM XML file.

Because this function creates a GeoDataFrame of features from an OSM XML file that has already been downloaded (i.e., no query is made to the Overpass API), the *polygon* and *tags* arguments are optional. If they are None, filtering will be skipped.

Parameters

- **filepath** (str | Path) – Path to file containing OSM XML data.
- **polygon** (Polygon | MultiPolygon | None) – Spatial boundaries to optionally filter the final GeoDataFrame.
- **tags** (dict[str, bool | str | list[str]] | None) – Query tags to optionally filter the final GeoDataFrame.
- **encoding** (str) – The OSM XML file's character encoding.

Returns
gdf – The features, multi-indexed by element type and OSM ID.

Return type
gpd.GeoDataFrame

6.4.8 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** (`str | dict[str, str]`) – Query string or structured dict to geocode.
- **which_result** (`int | None`) – 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.

Returns
gdf – GeoDataFrame with one row containing the geocoding result.

Return type
geopandas.GeoDataFrame

`osmnx.geocoder._get_first_polygon(results)`

Choose first result of geometry type (Multi)Polygon from list of results.

Parameters
results (`list[dict[str, Any]]`) – Results from the Nominatim API.

Returns
result – The chosen result.

Return type
dict[str, *Any*]

`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 (`str`) – The query string to geocode.

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

Return type
tuple[float, float]

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

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 an int 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** (str | dict[str, str] | list[str | dict[str, str]]) – The query string(s) or structured dict(s) to geocode.
- **which_result** (int | None | list[int | None]) – 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 (sorted by importance) 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.

Returns
gdf – GeoDataFrame with one row for each query result.

Return type
geopandas.GeoDataFrame

6.4.9 osmnx.graph module

Download and create graphs from OpenStreetMap data.

Refer to the Getting Started guide for usage limitations.

osmnx.graph.**_add_paths**(*G*, *paths*, *bidirectional*)

Add OSM paths to the graph as edges.

Parameters

- **G** (MultiDiGraph) – The graph to add paths to.
- **paths** (Iterable[dict[str, Any]]) – Iterable of paths' *tag:value* attribute data dicts.
- **bidirectional** (bool) – If True, create bidirectional 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[str, Any]) – OSM element of type "node".

Returns
node – The converted node.

Return type
dict[str, *Any*]

osmnx.graph.**_convert_path**(*element*)

Convert an OSM way element into the format for a NetworkX path.

Parameters
element (dict[str, Any]) – OSM element of type "way".

Returns
path – The converted path.

Return type
dict[str, *Any*]

osmnx.graph.**_create_graph**(*response_jsons*, *bidirectional*)

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[dict[str, Any]]) – Iterable of JSON responses from the Overpass API.
- **bidirectional** (bool) – If True, create bidirectional edges for one-way streets.

Returns
G – The resulting MultiDiGraph.

Return type
networkx.MultiDiGraph

osmnx.graph.**_is_path_one_way**(*attrs*, *bidirectional*, *oneway_values*)

Determine if a path of nodes allows travel in only one direction.

Parameters

- **attrs** (dict[str, Any]) – A path's *tag:value* attribute data.
- **bidirectional** (bool) – Whether this is a bidirectional network type.
- **oneway_values** (set[str]) – The values OSM uses in its "oneway" tag to denote True.

Returns
is_one_way – True if path allows travel in only one direction, otherwise False.

Return type
bool

osmnx.graph.**_is_path_reversed**(*attrs*, *reversed_values*)

Determine if the order of nodes in a path should be reversed.

Parameters

- **attrs** (dict[str, Any]) – A path's *tag:value* attribute data.
- **reversed_values** (set[str]) – The values OSM uses in its 'oneway' tag to denote travel can only occur in the opposite direction of the node order.

Returns
is_reversed – True if nodes' order should be reversed, otherwise False.

Return type
bool

osmnx.graph.**_parse_nodes_paths**(*response_json*)

Construct dicts of nodes and paths from an Overpass response.

Parameters
response_json (dict[str, Any]) – JSON response from the Overpass API.

Returns
nodes, paths – Each dict's keys are OSM IDs and values are dicts of attributes.

Return type
tuple[dict[int, dict[str, *Any*]], dict[int, dict[str, *Any*]]]

osmnx.graph.**graph_from_address**(*address*, *dist*, *, *dist_type='bbox'*, *network_type='all'*, *simplify=True*, *retain_all=False*, *truncate_by_edge=False*, *custom_filter=None*)

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

This function uses filters to query the Overpass API: you can either specify a pre-defined *network_type* or provide your own *custom_filter* with Overpass QL.

Use the *settings* module's *useful_tags_node* and *useful_tags_way* settings to configure which OSM node/way tags are added as graph node/edge attributes. If you want a fully bidirectional network, ensure your *network_type* is in *settings.bidirectional_network_types* before creating your graph. You can also 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 customizations.

Parameters

- **address** (str) – The address to geocode and use as the central point around which to construct the graph.
- **dist** (float) – Retain only those nodes within this many meters of *center_point*, measuring distance according to *dist_type*.
- **dist_type** (str) – {"network", "bbox"} If "bbox", retain only those nodes within a bounding box of *dist*. If "network", retain only those nodes within *dist* network distance from the centermost node.
- **network_type** (str) – {"all", "all_public", "bike", "drive", "drive_service", "walk"} What type of street network to retrieve 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. If False, retain only the largest weakly connected component.
- **truncate_by_edge** (bool) – If True, retain nodes the outside bounding box if at least one of the node's neighbors lies within the bounding box.
- **custom_filter** (str | list[str] | None) – A custom ways filter to be used instead of the *network_type* presets, e.g. *'["power"~"line"]' or '["highway"~"motorway|trunk"]'*. If *str*, the intersection of keys/values will be used, e.g., *'[maxspeed=50][lanes=2]'* will return all ways having both maxspeed of 50 and two lanes. If *list*, the union of the *list* items will be used, e.g., *['[maxspeed=50]', '[lanes=2]']* will return all ways having either maximum speed of 50 or two lanes. Also pass in a *network_type* that is in *settings.bidirectional_network_types* if you want the graph to be fully bidirectional.

Returns
G – The resulting MultiDiGraph.

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_bbox**(*bbox*, *, *network_type='all'*, *simplify=True*, *retain_all=False*, *truncate_by_edge=False*, *custom_filter=None*)

Download and create a graph within a lat-lon bounding box.

This function uses filters to query the Overpass API: you can either specify a pre-defined *network_type* or provide your own *custom_filter* with Overpass QL.

Use the *settings* module's *useful_tags_node* and *useful_tags_way* settings to configure which OSM node/way tags are added as graph node/edge attributes. If you want a fully bidirectional network, ensure your *network_type* is in *settings.bidirectional_network_types* before creating your graph. You can also 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 customizations.

Parameters

- **bbox** (tuple[float, float, float, float]) – Bounding box as *(left, bottom, right, top)*. Coordinates should be in unprojected latitude-longitude degrees (EPSG:4326).
- **network_type** (str) – {"all", "all_public", "bike", "drive", "drive_service", "walk"} What type of street network to retrieve if *custom_filter* is None.
- **simplify** (bool) – If True, simplify graph topology via the *simplify_graph* function.
- **retain_all** (bool) – If True, return the entire graph even if it is not connected. If False, retain only the largest weakly connected component.
- **truncate_by_edge** (bool) – If True, retain nodes the outside bounding box if at least one of the node's neighbors lies within the bounding box.
- **custom_filter** (str | list[str] | None) – A custom ways filter to be used instead of the *network_type* presets, e.g. *'["power"~"line"]' or '["highway"~"motorway|trunk"]'*. If *str*, the intersection of keys/values will be used, e.g., *'[maxspeed=50][lanes=2]'* will return all ways having both maxspeed of 50 and two lanes. If *list*, the union of the *list* items will be used, e.g., *['[maxspeed=50]', '[lanes=2]']* will return all ways having either maximum speed of 50 or two lanes. Also pass in a *network_type* that is in *settings.bidirectional_network_types* if you want the graph to be fully bidirectional.

Returns
G – The resulting MultiDiGraph.

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'*, *simplify=True*, *retain_all=False*, *truncate_by_edge=False*, *which_result=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 *features_from_polygon* function.

This function uses filters to query the Overpass API: you can either specify a pre-defined *network_type* or provide your own *custom_filter* with Overpass QL.

Use the *settings* module's *useful_tags_node* and *useful_tags_way* settings to configure which OSM node/way tags are added as graph node/edge attributes. If you want a fully bidirectional network, ensure your *network_type* is in *settings.bidirectional_network_types* before creating your graph. You can also 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 customizations.

Parameters

- **query** (str | dict[str, str] | list[str | dict[str, str]]) – The query or queries to geocode to retrieve place boundary polygon(s).
- **network_type** (str) – {"all", "all_public", "bike", "drive", "drive_service", "walk"} What type of street network to retrieve 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. If False, retain only the largest weakly connected component.
- **truncate_by_edge** (bool) – If True, retain nodes outside the place boundary polygon(s) if at least one of the node's neighbors lies within the polygon(s).
- **which_result** (int | None | list[int | None]) – Which geocoding result to use. if None, auto-select the first (Multi)Polygon or raise an error if OSM doesn't return one.
- **custom_filter** (str | list[str] | None) – A custom ways filter to be used instead of the *network_type* presets, e.g. *'["power"~"line"]' or '["highway"~"motorway|trunk"]'*. If *str*, the intersection of keys/values will be used, e.g., *'[maxspeed=50][lanes=2]'* will return all ways having both maxspeed of 50 and two lanes. If *list*, the union of the *list* items will be used, e.g., *['[maxspeed=50]', '[lanes=2]']* will return all ways having either maximum speed of 50 or two lanes. Also pass in a *network_type* that is in *settings.bidirectional_network_types* if you want the graph to be fully bidirectional.

Returns
G – The resulting MultiDiGraph.

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*, *, *dist_type='bbox'*, *network_type='all'*, *simplify=True*, *retain_all=False*, *truncate_by_edge=False*, *custom_filter=None*)

Download and create a graph within some distance of a lat-lon point.

This function uses filters to query the Overpass API: you can either specify a pre-defined *network_type* or provide your own *custom_filter* with Overpass QL.

Use the *settings* module's *useful_tags_node* and *useful_tags_way* settings to configure which OSM node/way tags are added as graph node/edge attributes. If you want a fully bidirectional network, ensure your *network_type* is in *settings.bidirectional_network_types* before creating your graph. You can also 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 customizations.

Parameters

- **center_point** (tuple[float, float]) – The *(lat, lon)* center point around which to construct the graph. Coordinates should be in unprojected latitude-longitude degrees (EPSG:4326).
- **dist** (float) – Retain only those nodes within this many meters of *center_point*, measuring distance according to *dist_type*.
- **dist_type** (str) – {"bbox", "network"} If "bbox", retain only those nodes within a bounding box of *dist* length/width. If "network", retain only those nodes within *dist* network distance of the nearest node to *center_point*.
- **network_type** (str) – {"all", "all_public", "bike", "drive", "drive_service", "walk"} What type of street network to retrieve 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. If False, retain only the largest weakly connected component.
- **truncate_by_edge** (bool) – If True, retain nodes the outside bounding box if at least one of the node's neighbors lies within the bounding box.
- **custom_filter** (str | list[str] | None) – A custom ways filter to be used instead of the *network_type* presets, e.g. *'["power"~"line"]' or '["highway"~"motorway|trunk"]'*. If *str*, the intersection of keys/values will be used, e.g., *'[maxspeed=50][lanes=2]'* will return all ways having both maxspeed of 50 and two lanes. If *list*, the union of the *list* items will be used, e.g., *['[maxspeed=50]', '[lanes=2]']* will return all ways having either maximum speed of 50 or two lanes. Also pass in a *network_type* that is in *settings.bidirectional_network_types* if you want the graph to be fully bidirectional.

Returns
G – The resulting MultiDiGraph.

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'*, *simplify=True*, *retain_all=False*, *truncate_by_edge=False*, *custom_filter=None*)

Download and create a graph within the boundaries of a (Multi)Polygon.

This function uses filters to query the Overpass API: you can either specify a pre-defined *network_type* or provide your own *custom_filter* with Overpass QL.

Use the *settings* module's *useful_tags_node* and *useful_tags_way* settings to configure which OSM node/way tags are added as graph node/edge attributes. If you want a fully bidirectional network, ensure your *network_type* is in *settings.bidirectional_network_types* before creating your graph. You can also 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 customizations.

Parameters

- **polygon** (Polygon | MultiPolygon) – The geometry within which to construct the graph. Coordinates should be in unprojected latitude-longitude degrees (EPSG:4326).
- **network_type** (str) – {"all", "all_public", "bike", "drive", "drive_service", "walk"} What type of street network to retrieve 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. If False, retain only the largest weakly connected component.
- **truncate_by_edge** (bool) – If True, retain nodes outside *polygon* if at least one of the node's neighbors lies within *polygon*.
- **custom_filter** (str | list[str] | None) – A custom ways filter to be used instead of the *network_type* presets, e.g. *'["power"~"line"]' or '["highway"~"motorway|trunk"]'*. If *str*, the intersection of keys/values will be used, e.g., *'[maxspeed=50][lanes=2]'* will return all ways having both maxspeed of 50 and two lanes. If *list*, the union of the *list* items will be used, e.g., *['[maxspeed=50]', '[lanes=2]']* will return all ways having either maximum speed of 50 or two lanes. Also pass in a *network_type* that is in *settings.bidirectional_network_types* if you want the graph to be fully bidirectional.

Returns
G – The resulting MultiDiGraph.

Return type
nx.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 an OSM XML file.

Do not load an XML file previously 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.

Use the *settings* module's *useful_tags_node* and *useful_tags_way* settings to configure which OSM node/way tags are added as graph node/edge attributes.

Parameters

- **filepath** (str | Path) – Path to file containing OSM XML data.
- **bidirectional** (bool) – If True, create bidirectional 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. If False, retain only the largest weakly connected component.
- **encoding** (str) – The OSM XML file's character encoding.

Returns
G – The resulting MultiDiGraph.

Return type
networkx.MultiDiGraph

6.4.10 osmnx._http module

Handle HTTP requests to web APIs.

osmnx._http._check_cache(*key*)

Check if a key exists in the cache, and return its cache file path if so.

Parameters
key (str) – The key to look for in the cache.

Returns
cache_filepath – Filepath to cached data for *key* if it exists, otherwise None.

Return type
Path | None

osmnx._http._config_dns(*url*)

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

Resolves URL's hostname 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 (str) – The URL to consistently resolve the IP address of.

Return type
None

osmnx._http._get_http_headers(*, *user_agent=None*, *referer=None*, *accept_language=None*)

Update the default requests HTTP headers with OSMnx information.

Parameters

- **user_agent** (str | None) – The user agent. If None, use *settings.http_user_agent* value.
- **referer** (str | None) – The referer. If None, use *settings.http_referer* value.
- **accept_language** (str | None) – The accept language. If None, use *settings.http_accept_language* value.

Returns
headers – The updated HTTP headers.

Return type
dict[str, str]

osmnx._http._hostname_from_url(*url*)

Extract the hostname (domain) from a URL.

Parameters
url (str) – The url from which to extract the hostname.

Returns
hostname – The extracted hostname (domain).

Return type
str

osmnx._http._parse_response(*response*)

Parse JSON from a requests response and log the details.

Parameters
response (Response) – The response object.

Returns
response_json – Value will be a dict if the response is from the Google or Overpass APIs, and a list if the response is from the Nominatim API.

Return type
dict[str, *Any*] | list[dict[str, *Any*]]

osmnx._http._resolve_cache_filepath(*key*, *extension='json'*)

Determine a cache key's corresponding cache file path.

This uses the configured *settings.cache_folder* and calculates the 160 bit SHA-1 hash digest (40 hexadecimal characters) of *key* to determine a succinct but unique cache filename.

Parameters

- **key** (str) – The key for which to generate a cache file path, for example, a URL.
- **extension** (str) – The desired cache file's extension.

Returns
cache_filepath – Cache file path corresponding to *key*.

Return type
Path

osmnx._http._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 (str) – 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
str

osmnx._http._retrieve_from_cache(*url*)

Retrieve a HTTP response JSON object from the cache if it exists.

A cache hit returns the data. A cache miss returns None.

Parameters
url (str) – The URL of the request.

Returns
response_json – The cached response for *url* if it exists, otherwise None.

Return type
dict[str, *Any*] | list[dict[str, *Any*]] | None

osmnx._http._save_to_cache(*url*, *response_json*, *ok*)

Save a HTTP response JSON object to a file in the cache folder.

If request was sent to server via POST instead of GET, then *url* should be a GET-style representation of the request. Response is only saved to a cache file if *settings.use_cache* is True, *ok* is True, *response_json* is not None, and *response_json* does not contain a server "remark."

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 you will get a cache miss when the URL's parameters appeared in a different order.

Parameters

- **url** (str) – The URL of the request.
- **response_json** (dict[str, Any] | list[dict[str, Any]]) – The JSON HTTP response.
- **ok** (bool) – A *requests.response.ok* value.

Return type
None

6.4.11 osmnx.io module

File I/O functions to save/load 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 (bool | str) – The string to convert to bool.

Returns
bool_value – The boolean equivalent of the string literal.

Return type
bool

osmnx.io._convert_edge_attr_types(*G*, *dtypes*)

Convert graph edges' attributes using a dict of data types.

Parameters

- **G** (MultiDiGraph) – Graph to convert the edge attributes of.
- **dtypes** (dict[str, Any]) – Dict of edge attribute names:types.

Returns
G – The graph with its edges' attributes' types converted.

Return type
networkx.MultiDiGraph

osmnx.io._convert_graph_attr_types(*G*, *dtypes*)

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

Parameters

- **G** (MultiDiGraph) – Graph to convert the graph-level attributes of.
- **dtypes** (dict[str, Any]) – Dict of graph-level attribute names:types.

Returns
G – The graph with its graph-level attributes' types converted.

Return type
networkx.MultiDiGraph

osmnx.io.**_convert_node_attr_types**(*G*, *dtypes*)

Convert graph nodes' attributes using a dict of data types.

Parameters

- **G** (MultiDiGraph) – Graph to convert the node attributes of.
- **dtypes** (dict[str, Any]) – Dict of node attribute names:types.

Returns
G – The graph with its nodes' attributes' types converted.

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 (GeoDataFrame) – GeoDataFrame to stringify non-numeric columns of.

Returns
gdf – GeoDataFrame 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** (str | Path | None) – Path to the GraphML file.
- **graphml_str** (str | None) – Valid and decoded string representation of a GraphML file's contents.

- **node_dtypes** (dict[str, Any] | None) – Dict of node attribute names:types to convert values' data types. The type can be a type or a custom string converter function.
- **edge_dtypes** (dict[str, Any] | None) – Dict of edge attribute names:types to convert values' data types. The type can be a type or a custom string converter function.
- **graph_dtypes** (dict[str, Any] | None) – Dict of graph-level attribute names:types to convert values' data types. The type can be a type or a custom string converter function.

Returns
G – The loaded MultiDiGraph.

Return type
networkx.MultiDiGraph

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

Save graph nodes and edges to disk as layers in a GeoPackage file.

Parameters

- **G** (MultiDiGraph) – The graph to save.
- **filepath** (str | Path | None) – Path to the GeoPackage file including extension. If None, use default *settings.data_folder/graph.gpkg*.
- **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.
- **encoding** (str) – The character encoding of the saved GeoPackage file.

Return type
None

osmnx.io.**save_graph_xml**(*G*, *filepath=None*, ***, *way_tag_aggs=None*, *encoding='utf-8'*)

Save graph to disk as an OSM XML file.

This function exists only to allow serialization to the OSM XML format for applications that require it, and has constraints to conform to that. As such, it 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 an OSM XML file that you have downloaded or generated elsewhere, use the *graph.graph_from_xml* function.

Use the *settings* module's *useful_tags_node* and *useful_tags_way* settings to configure which tags your graph is created and saved with. This function merges graph edges such that each OSM way has one entry in the XML output, with the way's nodes topologically sorted. *G* must be unsimplified to save as OSM XML: otherwise, one edge could comprise multiple OSM ways, making it impossible to group and sort edges in way. *G* should also have been created with *ox.settings.all_oneway=True* for this function to behave properly.

Parameters

- **G** (MultiDiGraph) – Unsimplified, unprojected graph to save as an OSM XML file.
- **filepath** (str | Path | None) – Path to the saved file including extension. If None, use default *settings.data_folder/graph.osm*.
- **way_tag_aggs** (dict[str, Any] | None) – Keys are OSM way tag keys and values are aggregation functions (anything accepted as an argument by pandas.agg). Allows user to aggregate graph edge attribute values into single OSM way values. If None, or if some tag's key does not exist in the dict, the way attribute will be assigned the value of the first edge of the way.
- **encoding** (str) – The character encoding of the saved OSM XML file.

Return type
None

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

Save graph to disk as GraphML file.

Parameters

- **G** (MultiDiGraph) – The graph to save as.
- **filepath** (str | Path | None) – Path to the GraphML file including extension. If None, use default *settings.data_folder/graph.graphml*.
- **gephi** (bool) – If True, give each edge a unique key/id for compatibility with Gephi's interpretation of the GraphML specification.
- **encoding** (str) – The character encoding of the saved GraphML file.

Return type
None

6.4.12 osmnx._nominatim module

Tools to work with the Nominatim API.

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

Retrieve an OSM element from the Nominatim API.

Parameters

- **query** (str | dict[str, str]) – 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** (bool) – Whether to retrieve the place's geometry from the API.

Returns
response_json – The Nominatim API's response.

Return type
list[dict[str, *Any*]]

osmnx._nominatim.**_nominatim_request**(*params*, ***, *request_type='search'*)

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

Parameters

- **params** (OrderedDict[str, int | str]) – Key-value pairs of parameters.
- **request_type** (str) – Which Nominatim API endpoint to query, one of {"search", "reverse", "lookup"}.

Returns
response_json – The Nominatim API's response.

Return type
list[dict[str, *Any*]]

6.4.13 osmnx._osm_xml module

Read/write OSM XML files.

For file format information see https://wiki.openstreetmap.org/wiki/OSM_XML

class osmnx._osm_xml.**_OSMContentHandler**

SAX content handler for OSM XML.

Builds an Overpass-like response JSON object in self.object. For format notes, see https://wiki.openstreetmap.org/wiki/OSM_XML and https://dev.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.

Return type
None

Parameters
name (*str*)

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.

Return type
None

Parameters

- **name** (*str*)
- **attrs** (*AttributesImpl*)

osmnx._osm_xml.**_add_nodes_xml**(*parent*, *gdf_nodes*)

Add graph nodes as subelements of an XML parent element.

Parameters

- **parent** (Element) – The XML parent element.
- **gdf_nodes** (GeoDataFrame) – A GeoDataFrame of graph nodes.

Return type
None

osmnx._osm_xml.**_add_ways_xml**(*parent*, *gdf_edges*, *way_tag_aggs*)

Add graph edges (grouped as ways) as subelements of an XML parent element.

Parameters

- **parent** (Element) – The XML parent element.
- **gdf_edges** (GeoDataFrame) – A GeoDataFrame of graph edges with OSM way "id" column for grouping edges into ways.
- **way_tag_aggs** (dict[str, Any] | None) – Keys are OSM way tag keys and values are aggregation functions (anything accepted as an argument by *pandas.agg*). Allows user to aggregate graph edge attribute values into single OSM way values. If None, or if some tag's

key does not exist in the dict, the way attribute will be assigned the value of the first edge of the way.

Return type
None

osmnx._osm_xml.**_open_file**(*filepath*, *encoding*)

Open a file and return a file object, optionally handling bz2 or gz files.

Uses a wrapper context manager to yield the file object to ensure the file will always get closed when the caller is finished with it.

Parameters

- **filepath** (Path) – Path to file.
- **encoding** (str) – The file's character encoding.

Returns
file – The file handle.

Return type
Iterator[*TextIO*]

osmnx._osm_xml.**_overpass_json_from_xml**(*filepath*, *encoding*)

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

Parameters

- **filepath** (Path) – Path to file containing OSM XML data.
- **encoding** (str) – The XML file's character encoding.

Returns
response_json – A parsed JSON response from the Overpass API.

Return type
dict[str, *Any*]

osmnx._osm_xml.**_save_graph_xml**(*G*, *filepath*, *way_tag_aggs*, *encoding='utf-8'*)

Save graph to disk as an OSM XML file.

Parameters

- **G** (MultiDiGraph) – Unsimplified, unprojected graph to save as an OSM XML file.
- **filepath** (str | Path | None) – Path to the saved file including extension. If None, use default *settings.data_folder/graph.osm*.
- **way_tag_aggs** (dict[str, Any] | None) – Keys are OSM way tag keys and values are aggregation functions (anything accepted as an argument by *pandas.agg*). Allows user to aggregate graph edge attribute values into single OSM way values. If None, or if some tag's key does not exist in the dict, the way attribute will be assigned the value of the first edge of the way.
- **encoding** (str) – The character encoding of the saved OSM XML file.

Return type
None

osmnx._osm_xml.**_sort_nodes**(*G*, *osmid*)

Topologically sort the nodes of an OSM way.

Parameters

- **G** (MultiDiGraph) – The graph representing the OSM way.
- **osmid** (int) – The OSM way ID.

Returns
ordered_nodes – The way's node IDs in topologically sorted order.

Return type
list[int]

6.4.14 osmnx._overpass module

Tools to work with the Overpass API.

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

Create an Overpass features query string based on tags.

Parameters

- **polygon_coord_str** (str) – The lat lon coordinates.
- **tags** (dict[str, bool | str | list[str]]) – Tags used for finding elements in the search area.

Returns
query – The Overpass features query.

Return type
str

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

Retrieve OSM features within some boundary polygon from the Overpass API.

Parameters

- **polygon** (Polygon) – Boundary to retrieve elements within.
- **tags** (dict[str, bool | str | list[str]]) – Tags used for finding elements in the selected area.

Yields
response_json – JSON response from the Overpass server.

Return type
Iterator[dict[str, *Any*]]

osmnx._overpass._download_overpass_network(*polygon*, *network_type*, *custom_filter*)

Retrieve networked ways and nodes within boundary from the Overpass API.

Parameters

- **polygon** (Polygon | MultiPolygon) – The boundary to fetch the network ways/nodes within.
- **network_type** (str) – What type of street network to get if *custom_filter* is None.
- **custom_filter** (str | list[str] | None) – A custom "ways" filter to be used instead of *network_type* presets.

Yields
response_json – JSON response from the Overpass server.

Return type
Iterator[dict[str, Any]]

osmnx._overpass._get_network_filter(*network_type*)

Create a filter to query Overpass for the specified network type.

The filter queries Overpass for every OSM way with a "highway" tag but excludes ways that are incompatible with the requested network type. You can choose from the following types:

"all" retrieves all public and private-access ways currently in use, excluding those that represent areas either explicitly (area=yes) or by convention (rest_area, services).

"all_public" retrieves all public ways currently in use.

"bike" retrieves public bikeable ways and excludes foot ways, motor ways, and anything tagged biking=no.

"drive" retrieves public drivable streets and excludes service roads, anything tagged motor=no, and certain non-service roads tagged as providing certain services (such as alleys or driveways).

"drive_service" retrieves public drivable streets including service roads but excludes certain services (such as parking or emergency access).

"walk" retrieves public walkable ways and excludes cycle ways, motor ways, and anything tagged foot=no. It includes service roads like parking lot aisles and alleys that you can walk on even if they are unpleasant walks.

Parameters
network_type (str) – {"all", "all_public", "bike", "drive", "drive_service", "walk"} What type of street network to retrieve.

Returns
way_filter – The Overpass query filter.

Return type
str

osmnx._overpass._get_overpass_pause(*base_endpoint*, *, *recursion_pause=5*, *default_pause=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** (str) – Base Overpass API URL (without "/status" at the end).
- **recursion_pause** (float) – How long to wait between recursive calls if the server is currently running a query.
- **default_pause** (float) – If a fatal error occurs, fall back on this liberal pause duration.

Returns
pause – The current pause duration specified by the Overpass status endpoint.

Return type
float

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. Ignore interior ("holes") coordinates.

Parameters
polygon (Polygon | MultiPolygon) – The (Multi)Polygon to convert to exterior coordinate strings.

Returns
coord_strs – Exterior coordinates of polygon(s).

Return type
list[str]

osmnx._overpass.**_make_overpass_settings**()

Make settings string to send in Overpass query.

Returns
overpass_settings – The *settings.overpass_settings* string formatted with "timeout" and "maxsize" values.

Return type
str

osmnx._overpass.**_overpass_request**(*data*)

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

Parameters
data (OrderedDict[str, Any]) – Key-value pairs of parameters.

Returns
response_json – The Overpass API's response.

Return type
dict[str, *Any*]

6.4.15 osmnx.plot module

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

osmnx.plot.**_config_ax**(*ax*, *crs*, *bbox*, *padding*)

Configure a matplotlib axes instance for display.

Parameters

- **ax** (Axes) – The axes instance.
- **crs** (Any) – The coordinate reference system of the plotted geometries.
- **bbox** (tuple[float, float, float, float]) – Bounding box as *(left, bottom, right, top)*.
- **padding** (float) – Relative padding to add around *bbox*.

Returns
ax – The configured matplotlib axes object.

Return type
matplotlib.axes._axes.Axes

osmnx.plot.**_get_colors_by_value**(*vals*, *num_bins*, *cmap*, *start*, *stop*, *na_color*, *equal_size*)

Map colors to the values in a Series of node/edge attribute values.

Parameters

- **vals** (Series) – Series labels are node/edge IDs and values are attribute values.
- **num_bins** (int | None) – If None, linearly map a color to each value. Otherwise, assign values to this many bins then assign a color to each bin.
- **cmap** (str) – Name of the matplotlib colormap from which to choose the colors.
- **start** (float) – Where to start in the colorspace (from 0 to 1).
- **stop** (float) – Where to end in the colorspace (from 0 to 1).
- **na_color** (str) – The color to assign to 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
color_series – Labels are node/edge IDs, values are colors as hex strings.

Return type
pandas.Series

osmnx.plot.**_get_fig_ax**(*ax*, *figsize*, *bgcolor*, *polar*)

Generate a matplotlib Figure and (Polar)Axes or return existing ones.

Overloads

- **ax** (Axes | None), **figsize** (tuple[float, float]), **bgcolor** (str | None), **polar** (Literal[False]) → tuple[Figure, Axes]
- **ax** (Axes | None), **figsize** (tuple[float, float]), **bgcolor** (str | None), **polar** (Literal[True]) → tuple[Figure, PolarAxes]

Parameters

- **ax** (Axes | None) – If not None, plot on this pre-existing axes instance.
- **figsize** (tuple[float, float]) – If *ax* is None, create new figure with size *(width, height)*.
- **bgcolor** (str | None) – Background color of figure.
- **polar** (bool) – If True, generate a *PolarAxes* instead of an *Axes* instance.

Returns
fig, ax – The resulting matplotlib figure and axes objects.

Return type
tuple[Figure, Axes | PolarAxes]

osmnx.plot.**_save_and_show**(*fig*, *ax*, *, *show=True*, *close=True*, *save=False*, *filepath=None*, *dpi=300*)

Save a figure to disk and/or show it, as specified by arguments.

Parameters

- **fig** (Figure) – The figure.
- **ax** (Axes) – The axes instance.
- **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** (str | Path | None) – The path to the file if *save* is True. File format is determined from the extension. If None, save at *settings.imgs_folder/image.png*.
- **dpi** (int) – The resolution of saved file if *save* is True.

Returns
fig, ax – The matplotlib figure and axes objects.

Return type
tuple[matplotlib.figure.Figure, matplotlib.axes._axes.Axes]

osmnx.plot.**_verify_mpl**()

Verify that matplotlib is installed and imported.

Return type
None

osmnx.plot.**get_colors**(*n*, *, *cmap='viridis'*, *start=0*, *stop=1*, *alpha=None*)

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

Parameters

- **n** (int) – How many colors to sample.
- **cmap** (str) – Name of the matplotlib colormap from which to sample the colors.
- **start** (float) – Where to start sampling from the colorspace (from 0 to 1).
- **stop** (float) – Where to end sampling from the colorspace (from 0 to 1).
- **alpha** (float | None) – If *None*, return colors as HTML-like hex triplet "#rrggbb" RGB strings. If *float*, return as "#rrggbbaa" RGBa strings.

Returns
color_list – The sampled colors.

Return type
list[str]

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

Return colors based on edges' numerical attribute values.

Parameters

- **G** (MultiDiGraph) – Input graph.
- **attr** (str) – Name of a node attribute with numerical values.
- **num_bins** (int | None) – If None, linearly map a color to each value. Otherwise, assign values to this many bins then assign a color to each bin.
- **cmap** (str) – Name of the matplotlib colormap from which to choose the colors.
- **start** (float) – Where to start in the colorspace (from 0 to 1).
- **stop** (float) – Where to end in the colorspace (from 0 to 1).
- **na_color** (str) – The color to assign to 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
edge_colors – Labels are *(u, v, k)* edge IDs, values are colors as hex strings.

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*)

Return colors based on nodes' numerical attribute values.

Parameters

- **G** (MultiDiGraph) – Input graph.
- **attr** (str) – Name of a node attribute with numerical values.

- **num_bins** (int | None) – If None, linearly map a color to each value. Otherwise, assign values to this many bins then assign a color to each bin.
- **cmap** (str) – Name of the matplotlib colormap from which to choose the colors.
- **start** (float) – Where to start in the colorspace (from 0 to 1).
- **stop** (float) – Where to end in the colorspace (from 0 to 1).
- **na_color** (str) – The color to assign to 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 – Labels are node IDs, values are colors as hex strings.

Return type
pandas.Series

osmnx.plot.**plot_figure_ground**(*G*, ***, *dist=805*, *street_widths=None*, *default_width=4*, *color='w'*, ***pg_kwargs*)

Plot a figure-ground diagram of a street network.

Parameters

- **G** (MultiDiGraph) – An unprojected graph.
- **dist** (float) – How many meters to extend plot's bounding box from the graph's center point. Default corresponds to a square mile bounding box.
- **street_widths** (dict[str, float]|None) – Dict keys are street types (ie, OSM "highway" tags) and values are the widths to plot them, in pixels.
- **default_width** (float) – Fallback width, in pixels, for any street type not in *street_widths*.
- **color** (str) – The color of the streets.
- ****pg_kwargs** (Any) – Keyword arguments to pass to *plot_graph*.

Returns
fig, ax – The resulting matplotlib figure and axes objects.

Return type
tuple[matplotlib.figure.Figure, matplotlib.axes._axes.Axes]

osmnx.plot.**plot_footprints**(*gdf*, ***, *ax=None*, *figsize=(8, 8)*, *color='orange'*, *edge_color='none'*, *edge_linewidth=0*, *alpha=None*, *bgcolor='#111111'*, *bbox=None*, *show=True*, *close=False*, *save=False*, *filepath=None*, *dpi=600*)

Visualize a GeoDataFrame of geospatial features' footprints.

Parameters

- **gdf** (gpd.GeoDataFrame) – GeoDataFrame of footprints (i.e., Polygons and/or MultiPolygons).
- **ax** (Axes | None) – If not None, plot on this pre-existing axes instance.
- **figsize** (tuple[float, float]) – If *ax* is None, create new figure with size *(width, height)*.
- **color** (str) – Color of the footprints.
- **edge_color** (str) – Color of the footprints' edges.
- **edge_linewidth** (float) – Width of the footprints' edges.

- **alpha** (float | None) – Opacity of the footprints' edges.
- **bgcolor** (str) – Background color of the figure.
- **bbox** (tuple[float, float, float, float] | None) – Bounding box as *(left, bottom, right, top)*. If None, calculate it from the spatial extents of the geometries in *gdf*.
- **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** (str | Path | None) – The path to the file if *save* is True. File format is determined from the extension. If None, save at *settings.imgs_folder/image.png*.
- **dpi** (int) – The resolution of saved file if *save* is True.

Returns
fig, ax – The resulting matplotlib figure and axes objects.

Return type
tuple[Figure, Axes]

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*, *bbox=None*, *show=True*, *close=False*, *save=False*, *filepath=None*, *dpi=300*)

Visualize a graph.

Parameters

- **G** (nx.MultiGraph | nx.MultiDiGraph) – Input graph.
- **ax** (Axes | None) – If not None, plot on this pre-existing axes instance.
- **figsize** (tuple[float, float]) – If *ax* is None, create new figure with size *(width, height)*.
- **bgcolor** (str) – Background color of the figure.
- **node_color** (str | Sequence[str]) – Color(s) of the nodes.
- **node_size** (float | Sequence[float]) – Size(s) of the nodes. If 0, then skip plotting the nodes.
- **node_alpha** (float | None) – Opacity of the nodes. If you passed RGBa values to *node_color*, set *node_alpha=None* to use the alpha channel in *node_color*.
- **node_edgecolor** (str | Iterable[str]) – Color(s) of the nodes' markers' borders.
- **node_zorder** (int) – The zorder to plot nodes. Edges are always 1, so set *node_zorder=0* to plot nodes beneath edges.
- **edge_color** (str | Iterable[str]) – Color(s) of the edges' lines.
- **edge_linewidth** (float | Sequence[float]) – Width(s) of the edges' lines. If 0, then skip plotting the edges.
- **edge_alpha** (float | None) – Opacity of the edges. If you passed RGBa values to *edge_color*, set *edge_alpha=None* to use the alpha channel in *edge_color*.
- **bbox** (tuple[float, float, float, float] | None) – Bounding box as *(left, bottom, right, top)*. If None, calculate it from spatial extents of plotted geometries.
- **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** (str | Path | None) – The path to the file if *save* is True. File format is determined from the extension. If None, save at *settings.imgs_folder/image.png*.
- **dpi** (int) – The resolution of saved file if *save* is True.

Returns
fig, ax – The resulting matplotlib figure and axes objects.

Return type
tuple[Figure, Axes]

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 path along a graph.

Parameters

- **G** (nx.MultiDiGraph) – Input graph.
- **route** (list[int]) – A path of node IDs.
- **route_color** (str) – The color of the route.
- **route_linewidth** (float) – Width of the route's line.
- **route_alpha** (float) – Opacity of the route's line.
- **orig_dest_size** (float) – Size of the origin and destination nodes.
- **ax** (Axes | None) – If not None, plot on this pre-existing axes instance.
- ****pg_kwargs** (Any) – Keyword arguments to pass to *plot_graph*.

Returns
fig, ax – The resulting matplotlib figure and axes objects.

Return type
tuple[Figure, Axes]

osmnx.plot.**plot_graph_routes**(*G*, *routes*, *, *route_colors='r'*, *route_linewidths=4*, ***pgr_kwargs*)

Visualize multiple paths along a graph.

Parameters

- **G** (MultiDiGraph) – Input graph.
- **routes** (Iterable[list[int]]) – Paths of node IDs.
- **route_colors** (str | Iterable[str]) – If string, the one color for all routes. Otherwise, the color for each route.
- **route_linewidths** (float | Iterable[float]) – If float, the one linewidth for all routes. Otherwise, the linewidth for each route.
- ****pgr_kwargs** (Any) – Keyword arguments to pass to *plot_graph_route*.

Returns
fig, ax – The resulting matplotlib figure and axes objects.

Return type
tuple[matplotlib.figure.Figure, matplotlib.axes._axes.Axes]

osmnx.plot.**plot_orientation**(*G*, ***, *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 edge bearings.

Ignores self-loop edges as their bearings are undefined. If *G* is a MultiGraph, all edge bearings will be bidirectional (ie, two reciprocal bearings per undirected edge). If *G* is a MultiDiGraph, all edge bearings will be directional (ie, one bearing per directed edge). See also the *bearings* module.

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

- **G** (nx.MultiGraph | nx.MultiDiGraph) – 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" attribute values less than *min_length*.
- **weight** (str | None) – If not None, weight the edges' bearings by this (non-null) edge attribute.
- **ax** (PolarAxes | None) – If not None, plot on this pre-existing axes instance (must have projection=polar).
- **figsize** (tuple[float, float]) – 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** (str) – Color of the histogram bars.
- **edgecolor** (str) – Color of the histogram bar edges.
- **linewidth** (float) – Width of the histogram bar edges.
- **alpha** (float) – Opacity of the histogram bars.
- **title** (str | None) – The figure's title.
- **title_y** (float) – The y position to place *title*.
- **title_font** (dict[str, Any] | None) – The title's *fontdict* to pass to matplotlib.
- **xtick_font** (dict[str, Any] | None) – The xtick labels' *fontdict* to pass to matplotlib.

Returns

fig, ax – The resulting matplotlib figure and polar axes objects.

Return type

tuple[Figure, PolarAxes]

6.4.16 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 (Any) – The identifier of the coordinate reference system. This 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 coordinate reference system 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 (or UPS) zone given *gdf*'s bounds.

Parameters

- **gdf** (GeoDataFrame) – The GeoDataFrame to be projected.
- **to_crs** (Any | None) – 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**(*geom*, *, *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 geometry to the coordinate reference system 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 (or UPS) zone given *geometry*'s bounds.

Parameters

- **geom** (Geometry) – The geometry to be projected.
- **crs** (Any | None) – The initial CRS of *geometry*. If None, it will be set to *settings.default_crs*.
- **to_crs** (Any | None) – 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
geom_proj, crs – The projected geometry and its new CRS.

Return type
tuple[shapely.Geometry, *Any*]

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 graph to the coordinate reference system 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 (or UPS) zone given *geometry*'s bounds.

Parameters

- **G** (MultiDiGraph) – The graph to be projected.
- **to_crs** (Any | None) – 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

6.4.17 osmnx.routing module

Calculate edge speeds, travel times, and weighted shortest paths.

osmnx.routing._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. If given string is not a valid numeric string, tries to look up its value in implicit maxspeed values mapping. Invalid inputs return None. See https://wiki.openstreetmap.org/wiki/Key:maxspeed for details on values and formats.

Parameters

- **maxspeed** (str | float) – An OSM way "maxspeed" attribute value. Null values are expected to be of type float (*numpy.nan*), and non-null values are strings.
- **agg** (Callable[[Any], Any]) – Aggregation function if *maxspeed* contains multiple values (default is *numpy.mean*).
- **convert_mph** (bool) – If True, convert miles per hour to kilometers per hour.

Returns
clean_value – Clean value resulting from *agg* function.

Return type
float | None

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

Collapse a list of maxspeed values to a single value.

Returns None if a ValueError is encountered.

Parameters

- **value** (str | float | list[str | float]) – An OSM way "maxspeed" attribute value. Null values are expected to be of type float (*numpy.nan*), and non-null values are strings.
- **agg** (Callable[[Any], Any]) – The aggregation function to reduce the list to a single value.

Returns
collapsed – If *value* was a string or null, it is just returned directly. Otherwise, the return is a float representation of the aggregated value in the list (converted to kph if original value was in mph).

Return type
float | str | None

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

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

This function uses Dijkstra's algorithm. It is a convenience wrapper around *networkx.shortest_path*, with exception handling for unsolvable paths. If the path is unsolvable, it returns None.

Parameters

- **G** (MultiDiGraph) – Input graph.
- **orig** (int) – Origin node ID.
- **dest** (int) – Destination node ID.
- **weight** (str) – Edge attribute to minimize when solving shortest path.

Returns
path – The node IDs constituting the shortest path.

Return type
list[int] | None

osmnx.routing.**add_edge_speeds**(*G*, *, *hwy_speeds=None*, *fallback=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

If you wish to set all edge speeds to a single constant value (such as for a walking network), use *nx.set_edge_attributes* to set the *speed_kph* attribute value directly, rather than using this function.

Parameters

- **G** (MultiDiGraph) – Input graph.
- **hwy_speeds** (dict[str, float] | None) – Dict keys are OSM highway types and values are 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 pre-existing speed value of all edges of that highway type.
- **fallback** (float | None) – Default speed value (km per hour) to assign to edges whose highway type did not appear in *hwy_speeds* and had no pre-existing speed attribute values on any edge.
- **agg** (Callable[[Any], Any]) – 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.routing.**add_edge_travel_times**(*G*)

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 (MultiDiGraph) – Input graph.

Returns
G – Graph with *travel_time* attributes on all edges.

Return type
networkx.MultiDiGraph

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** (MultiDiGraph) – Input graph.
- **orig** (int) – Origin node ID.
- **dest** (int) – Destination node ID.
- **k** (int) – Number of shortest paths to solve.
- **weight** (str) – Edge attribute to minimize when solving shortest paths.

Yields
path – The node IDs constituting the next-shortest path.

Return type
Iterator[list[int]]

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

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

Parameters

- **G** (MultiDiGraph) – Input graph.
- **route** (list[int]) – Node IDs constituting the path.
- **weight** (str) – Attribute value to minimize when choosing between parallel edges.

Returns
gdf_edges – The ordered edges in the path.

Return type
geopandas.GeoDataFrame

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

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

Overloads

- **G** (nx.MultiDiGraph), **orig** (int), **dest** (int), **weight** (str), **cpus** (int | None) → list[int] | None
- **G** (nx.MultiDiGraph), **orig** (int), **dest** (int), **cpus** (int | None) → list[int] | None
- **G** (nx.MultiDiGraph), **orig** (int), **dest** (int), **weight** (str) → list[int] | None
- **G** (nx.MultiDiGraph), **orig** (int), **dest** (int) → list[int] | None
- **G** (nx.MultiDiGraph), **orig** (Iterable[int]), **dest** (Iterable[int]), **weight** (str), **cpus** (int | None) → list[list[int] | None]
- **G** (nx.MultiDiGraph), **orig** (Iterable[int]), **dest** (Iterable[int]), **cpus** (int | None) → list[list[int] | None]

- **G** (nx.MultiDiGraph), **orig** (Iterable[int]), **dest** (Iterable[int]), **weight** (str) → list[list[int] | None]
- **G** (nx.MultiDiGraph), **orig** (Iterable[int]), **dest** (Iterable[int]) → list[list[int] | None]

Parameters

- **G** (*networkx.MultiDiGraph*)
- **orig** (*int | Iterable[int]*)
- **dest** (*int | Iterable[int]*)
- **weight** (*str*)
- **cpus** (*int | None*)

Return type
list[int] | None | list[list[int] | None]

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** (MultiDiGraph) – Input graph.
- **orig** (int | Iterable[int]) – Origin node ID(s).
- **dest** (int | Iterable[int]) – Destination node ID(s).
- **weight** (str) – Edge attribute to minimize when solving shortest path.
- **cpus** (int | None) – How many CPU cores to use if multiprocessing. If None, use all available. If you are multiprocessing, make sure you protect your entry point: see the Python docs for details.

Returns
path – The node IDs constituting the shortest path, or, if *orig* and *dest* are both iterable, then a list of such paths.

Return type
list[int] | None | list[list[int] | None]

6.4.18 osmnx.settings module

Global settings that can be configured by the user.

all_oneway
[bool] Only use if subsequently saving graph to an OSM XML file via the *save_graph_xml* function. If True, forces all ways to be added as one-way ways, preserving the original order of the nodes in the OSM way. This also retains the original OSM way's oneway tag's string value as edge attribute values, rather than converting them to True/False bool values. Default is *False*.

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

cache_folder
[str | Path] Path to folder to save/load HTTP response cache files, if the *use_cache* setting is 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 Overpass response 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
[str | Path] Path to folder to save/load graph files by default. Default is *"./data"*.

default_access
[str] Filter for the OSM "access" tag. 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" tag 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
[str] Default coordinate reference system to set when creating graphs. Default is *"epsg:4326"*.

doh_url_template
[str | None] 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
[str] Endpoint of the Google Maps Elevation API (or equivalent), containing up to two parameters, in order: *locations* and *key*. Default is: *"https://maps.googleapis.com/maps/api/elevation/json?locations={locations}&key={key}"*.
As alternative free examples, the Open Topo Data API would be: *"https://api.opentopodata.org/v1/aster30m?locations={locations}"* and the Open-Elevation API would be: *"https://api.open-elevation.com/api/v1/lookup?locations={locations}"*.

http_accept_language
[str] HTTP header accept-language. Default is *"en"*. Note that Nominatim's default language is "en" and it may sort its results' importance scores differently if a different language is specified.

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

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

imgs_folder
[str | 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
[str] 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
[str] Name of the logger. Default is *"OSMnx"*.

logs_folder
[str | Path] Path to folder in which to save log files. Default is *"./logs"*.

max_query_area_size
[float] 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*.

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

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

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

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
[str] 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.

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

requests_kwargs
[dict[str, Any]] 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 *{}*.

requests_timeout
[int] The timeout interval in seconds for HTTP requests, and (when applicable) for Overpass server to use for executing the query. Default is *180*.

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

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

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

6.4.19 osmnx.simplification module

Simplify, correct, and consolidate spatial graph nodes and edges.

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

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

Parameters

- **G** (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[int]) – 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[int]

osmnx.simplification.**_consolidate_intersections_rebuild_graph**(*G*, *tolerance*, *max_length*, *reconnect_edges*, *node_attr_aggs*)

Consolidate intersections comprising clusters of nearby nodes.

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

Parameters

- **G** (MultiDiGraph) – A projected graph.
- **tolerance** (float | dict[int, float]) – Nodes are buffered to this distance (in graph's geometry's units) and subsequent overlaps are dissolved into a single node. If scalar, then that single value will be used for all nodes. If dict (mapping node IDs to individual values), then those values will be used per node and any missing node IDs will not be buffered.
- **max_length** (float | None) – Ignore edges longer than this when determining whether nearby nodes are topologically connected. If None, consider all edges.
- **reconnect_edges** (bool) – If True, reconnect edges (and their geometries) to the consolidated nodes in rebuilt graph, and update the edge length attributes. If False, the returned graph has no edges (which is faster if you just need topologically consolidated intersection counts).
- **node_attr_aggs** (dict[str, Any] | None) – Allows user to aggregate node attributes values when merging nodes. Keys are node attribute names and values are aggregation functions (anything accepted as an argument by *pandas.agg*). Node attributes not in *node_attr_aggs* will contain the unique values across the merged nodes. If None, defaults to *{"elevation": "mean"}*.

Returns

Gc – A rebuilt graph with consolidated intersections and (optionally) reconnected edge geometries.

Return type

networkx.MultiDiGraph

osmnx.simplification.**_get_paths_to_simplify**(*G*, *node_attrs_include*, *edge_attrs_differ*)

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** (MultiDiGraph) – Input graph.

- **node_attrs_include** (Iterable[str] | None) – Node attribute names for relaxing the strictness of endpoint determination. A node is always an endpoint if it possesses one or more of the attributes in *node_attrs_include*.
- **edge_attrs_differ** (Iterable[str] | None) – Edge attribute names for relaxing the strictness of endpoint determination. A node is always an endpoint if its incident edges have different values than each other for any attribute in *edge_attrs_differ*.

Yields
path_to_simplify

Return type
Iterator[list[int]]

osmnx.simplification.**_is_endpoint**(*G*, *node*, *node_attrs_include*, *edge_attrs_differ*)

Determine if a node is a true endpoint of an edge.

Return True if the node is a "true" endpoint of an edge in the network, otherwise False. OpenStreetMap data includes many nodes that exist only as geometric vertices to allow ways to curve. *node* is a true edge endpoint if it satisfies at least 1 of the following 5 rules:

1) It is its own neighbor (ie, it self-loops).

2) Or, it has no incoming edges or no outgoing edges (ie, all its incident edges are inbound or all its incident edges are outbound).

3) Or, it does not have exactly two neighbors and degree of 2 or 4.

4) Or, if *node_attrs_include* is not None and it has one or more of the attributes in *node_attrs_include*.

5) Or, if *edge_attrs_differ* is not None and its incident edges have different values than each other for any of the edge attributes in *edge_attrs_differ*.

Parameters

- **G** (MultiDiGraph) – Input graph.
- **node** (int) – The ID of the node to check.
- **node_attrs_include** (Iterable[str] | None) – Node attribute names for relaxing the strictness of endpoint determination. A node is always an endpoint if it possesses one or more of the attributes in *node_attrs_include*.
- **edge_attrs_differ** (Iterable[str] | None) – Edge attribute names for relaxing the strictness of endpoint determination. A node is always an endpoint if its incident edges have different values than each other for any attribute in *edge_attrs_differ*.

Returns
endpoint – True if node is an endpoint, otherwise False.

Return type
bool

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

Geometrically merge nodes within some distance of each other.

Parameters

- **G** (MultiDiGraph) – A projected graph.
- **tolerance** (float | dict[int, float]) – Nodes are buffered to this distance (in graph's geometry's units) and subsequent overlaps are dissolved into a single node. If scalar, then that single value will be used for all nodes. If dict (mapping node IDs to individual values), then those values will be used per node and any missing node IDs will not be buffered.

Returns
merged – The merged overlapping polygons of the buffered nodes.

Return type
geopandas.GeoSeries

osmnx.simplification.**_remove_rings**(*G*, *node_attrs_include*, *edge_attrs_differ*)

Remove all graph components that consist only of a single chordless cycle.

This identifies all connected components in the graph that consist only of a single isolated self-contained ring, and removes them from the graph.

Parameters

- **G** (MultiDiGraph) – Input graph.
- **node_attrs_include** (Iterable[str] | None) – Node attribute names for relaxing the strictness of endpoint determination. A node is always an endpoint if it possesses one or more of the attributes in *node_attrs_include*.
- **edge_attrs_differ** (Iterable[str] | None) – Edge attribute names for relaxing the strictness of endpoint determination. A node is always an endpoint if its incident edges have different values than each other for any attribute in *edge_attrs_differ*.

Returns
G – Graph with all chordless cycle components removed.

Return type
networkx.MultiDiGraph

osmnx.simplification.**consolidate_intersections**(*G*, ***, *tolerance=10*, *rebuild_graph=True*, *max_length=None*, *dead_ends=False*, *reconnect_edges=True*, *node_attr_aggs=None*)

Consolidate intersections comprising clusters of nearby nodes.

This algorithm is described in the journal article: Boeing, G. 2025. "Topological Graph Simplification Solutions to the Street Intersection Miscount Problem." Transactions in GIS, 29 (3), e70037. https://doi.org/10.1111/tgis.70037

Merges nearby nodes and returns either their centroids or a rebuilt graph with consolidated intersections and reconnected edge geometries. The *tolerance* argument can be a single value applied to all nodes or individual per-node values. It should be adjusted to approximately match street design standards in the specific street network, and you should use a projected graph to work in meaningful and consistent units like meters. Note: *tolerance* represents a per-node buffering radius. For example, to consolidate nodes within 10 meters of each other, use *tolerance=5*.

When *rebuild_graph* is False, it uses a purely geometric (and relatively fast) algorithm to identify "geometrically close" nodes, merge them, and return the merged intersections' centroids. When *rebuild_graph* is 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 "osmid" values. Refer to nodes' "osmid_original" attributes for original "osmid" values. If multiple nodes were merged together, the "osmid_original" attribute is a list of merged nodes' "osmid" values.

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** (nx.MultiDiGraph) – A projected graph.
- **tolerance** (float | dict[int, float]) – Nodes are buffered to this distance (in graph's geometry's units) and subsequent overlaps are dissolved into a single node. If scalar, then that single value will be used for all nodes. If dict (mapping node IDs to individual values), then those values will be used per node and any missing node IDs will not be buffered.
- **rebuild_graph** (bool) – If True, consolidate the nodes topologically, rebuild the graph, and return as MultiDiGraph. Otherwise, consolidate the nodes geometrically and return the consolidated node points as GeoSeries.
- **max_length** (float | None) – Ignore edges longer than this when determining whether nearby nodes are topologically connected. If None, consider all edges. This prevents long edges that loop outside an intersection, such as cloverleaf ramps, from merging their nearby endpoint nodes.
- **dead_ends** (bool) – If False, discard dead-end nodes to return only street intersection points.
- **reconnect_edges** (bool) – If True, reconnect edges (and their geometries) to the consolidated nodes in rebuilt graph, and update the edge length attributes. If False, the returned graph has no edges (which is faster if you just need topologically consolidated intersection counts). Ignored if *rebuild_graph* is not True.
- **node_attr_aggs** (dict[str, Any] | None) – Allows user to aggregate node attributes values when merging nodes. Keys are node attribute names and values are aggregation functions (anything accepted as an argument by *pandas.agg*). Node attributes not in *node_attr_aggs* will contain the unique values across the merged nodes. If None, defaults to *{"elevation": numpy.mean}*.

Returns

Gc or gs – If *rebuild_graph=True*, returns MultiDiGraph with consolidated intersections and (optionally) reconnected edge geometries. If *rebuild_graph=False*, returns GeoSeries of Points representing the centroids of street intersections.

Return type

nx.MultiDiGraph | gpd.GeoSeries

osmnx.simplification.**simplify_graph**(*G*, *, *node_attrs_include=None*, *edge_attrs_differ=None*, *remove_rings=True*, *track_merged=False*, *edge_attr_aggs=None*)

Simplify a graph's topology by removing interstitial nodes.

This algorithm is described in the journal article: Boeing, G. 2025. "Topological Graph Simplification Solutions to the Street Intersection Miscount Problem." Transactions in GIS, 29 (3), e70037. https://doi.org/10.1111/tgis.70037

This simplifies the graph's topology by removing all nodes that are not intersections or dead-ends, by creating an edge directly between the end points that encapsulate them while retaining the full 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 unique 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.

Use the *node_attrs_include* or *edge_attrs_differ* parameters to relax simplification strictness. For example, *edge_attrs_differ=["osmid"]* will retain every node whose incident edges have different OSM IDs. This lets you keep nodes at elbow two-way intersections (but be aware that sometimes individual blocks have multiple OSM IDs within them too). You could also use this parameter to retain nodes where sidewalks or bike lanes begin/end in the middle of a block. Or for example, *node_attrs_include=["highway"]* will retain every node with a "highway" attribute (regardless of its value), even if it does not represent a street junction.

Parameters

- **G** (MultiDiGraph) – Input graph.
- **node_attrs_include** (Iterable[str] | None) – Node attribute names for relaxing the strictness of endpoint determination. A node is always an endpoint if it possesses one or more of the attributes in *node_attrs_include*.
- **edge_attrs_differ** (Iterable[str] | None) – Edge attribute names for relaxing the strictness of endpoint determination. A node is always an endpoint if its incident edges have different values than each other for any attribute in *edge_attrs_differ*.
- **remove_rings** (bool) – If True, remove any graph components that consist only of a single chordless cycle (i.e., an isolated self-contained ring).
- **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.
- **edge_attr_aggs** (dict[str, Any] | None) – Allows user to aggregate edge segment attributes when simplifying an edge. Keys are edge attribute names and values are aggregation functions to apply to these attributes when they exist for a set of edges being merged. Edge attributes not in *edge_attr_aggs* will contain the unique values across the merged edge segments. If None, defaults to *{"length": sum, "travel_time": sum}*.

Returns
Gs – Topologically simplified graph, with a new *geometry* attribute on each simplified edge.

Return type
networkx.MultiDiGraph

6.4.20 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. Due to OSMnx's periphery cleaning when the graph was created, 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** (MultiDiGraph) – Input graph.
- **area** (float | None) – If not None, calculate density measures and use *area* (in square meters) as the denominator.
- **clean_int_tol** (float | None) – 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

dict[str, Any] – *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[str, *Any*]

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. Returns None if the edge lengths sum to zero.

Parameters
Gu (MultiGraph) – Undirected input graph.

Returns
circuity_avg – The graph's average undirected edge circuity.

Return type
float | None

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** (MultiDiGraph) – Input graph.
- **nodes** (Iterable[int] | None) – 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[int, int]

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

Calculate graph's total edge length.

Parameters
G (MultiGraph) – Input graph.

Returns
length – Total length (meters) of edges in graph.

Return type
float

osmnx.stats.**intersection_count**(*G*, *, *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** (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 (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 (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 (MultiGraph) – Undirected input graph.

Returns
count – Count of street segments in graph.

Return type
int

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

Retrieve nodes' *street_count* attribute values.

See also the *count_streets_per_node* function for the calculation.

Parameters
G (MultiDiGraph) – Input graph.

Returns
spn – Dictionary with node ID keys and street count values.

Return type
dict[int, int]

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

Calculate graph's average count of streets per node.

Parameters
G (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 (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[int, int]

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

Calculate streets-per-node proportions.

Parameters
G (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[int, float]

6.4.21 osmnx.truncate module

Truncate graph by distance, bounding box, or polygon.

osmnx.truncate.**largest_component**(*G*, *, *strongly=False*)

Return *G*'s largest weakly or strongly connected component as a graph.

Parameters

- **G** (MultiDiGraph) – Input graph.
- **strongly** (bool) – If True, return the largest strongly connected component. Otherwise return the largest weakly connected component.

Returns
G – The largest connected component subgraph of the original graph.

Return type
networkx.MultiDiGraph

osmnx.truncate.**truncate_graph_bbox**(*G*, *bbox*, *, *truncate_by_edge=False*)

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

Parameters

- **G** (MultiDiGraph) – Input graph.
- **bbox** (tuple[float, float, float, float]) – Bounding box as *(left, bottom, right, top)*.
- **truncate_by_edge** (bool) – If True, retain nodes outside bounding box if at least one of node's neighbors is within the bounding box.

Returns
G – The truncated graph.

Return type
networkx.MultiDiGraph

osmnx.truncate.**truncate_graph_dist**(*G*, *source_node*, *dist*, *, *weight='length'*)

Remove from a graph every node beyond some network distance from a node.

This function must calculate shortest path distances between *source_node* and every other graph node, which can be slow on large graphs.

Parameters

- **G** (MultiDiGraph) – Input graph.
- **source_node** (int) – Node from which to measure network distances to all other nodes.
- **dist** (float) – Remove every node in the graph that is greater than *dist* distance (in same units as *weight* attribute) along the network from *source_node*.
- **weight** (str) – Graph edge attribute to use to measure distance.

Returns
G – The truncated graph.

Return type
networkx.MultiDiGraph

osmnx.truncate.**truncate_graph_polygon**(*G*, *polygon*, *, *truncate_by_edge=False*)

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

Parameters

- **G** (nx.MultiDiGraph) – Input graph.
- **polygon** (Polygon | MultiPolygon) – Only retain nodes in graph that lie within this geometry.
- **truncate_by_edge** (bool) – If True, retain nodes outside boundary polygon if at least one of node's neighbors is within the polygon.

Returns
G – The truncated graph.

Return type
nx.MultiDiGraph

6.4.22 osmnx.utils module

General utility functions.

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

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

Parameters

- **name** (str) – Name of the logger.
- **filename** (str) – Name of the log file, without file extension.

Returns
logger – The logger.

Return type
Logger

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

Print the OSMnx package's citation information.

Boeing, G. (2025). Modeling and Analyzing Urban Networks and Amenities with OSMnx. Geographical Analysis, 57(4), 567-577. doi:10.1111/gean.70009

Parameters
style (str) – {"apa", "bibtex", "ieee"} The citation format, either APA or BibTeX or IEEE.

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** (str) – The message to log.
- **level** (int | None) – One of the Python *logger.level* constants. If None, set to *settings.log_level* value.
- **name** (str | None) – The name of the logger. If None, set to *settings.log_name* value.

- **filename** (str | None) – The name of the log file, without file extension. If None, set to *settings.log_filename* value.

Return type
None

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

Return current local timestamp as a string.

Parameters

- **style** (str) – {"datetime", "iso8601", "date", "time"} Format the timestamp with this built-in style.
- **template** (str | None) – If not None, format the timestamp with this format string instead of one of the built-in styles.

Returns
timestamp – The current timestamp.

Return type
str

6.4.23 osmnx.utils_geo module

Geospatial utility functions.

osmnx.utils_geo.**_consolidate_subdivide_geometry**(*geom*)

Consolidate and subdivide some (projected) 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 the *settings* module's *max_query_area_size*. Geometries with areas much larger than *max_query_area_size* may take a long time to process.

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 if necessary.

Parameters
geom (Polygon | MultiPolygon) – The projected (in meter units) geometry to consolidate and subdivide.

Returns
geom – The resulting consolidated and subdivided geometry.

Return type
MultiPolygon

osmnx.utils_geo.**_intersect_index_quadrats**(*geoms*, *polygon*)

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

- **geoms** (gpd.GeoSeries) – The geometries to intersect with the polygon.
- **polygon** (Polygon | MultiPolygon) – The polygon to intersect with the geometries.

Returns
geoms_in_poly – The index labels of the geometries that intersected the polygon.

Return type
set[Any]

osmnx.utils_geo.**_quadrat_cut_geometry**(*geom*, *quadrat_width*)

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

Parameters

- **geom** (Polygon | MultiPolygon) – The geometry to split up into smaller sub-polygons.
- **quadrat_width** (float) – Width (in geometry's units) of quadrat squares with which to split up the geometry.

Returns
geom – The resulting split-up geometry.

Return type
MultiPolygon

osmnx.utils_geo.**bbox_from_point**(*point*, *dist*, *, *project_utm=False*, *return_crs=False*)

Create a bounding box around a (lat, lon) point.

Overloads

- **point** (tuple[float, float]), **dist** (float) → tuple[float, float, float, float]
- **point** (tuple[float, float]), **dist** (float), **return_crs** (Literal[True]) → tuple[float, float, float, float]
- **point** (tuple[float, float]), **dist** (float), **return_crs** (Literal[False]) → tuple[float, float, float, float]
- **point** (tuple[float, float]), **dist** (float), **project_utm** (Literal[True]) → tuple[float, float, float, float]
- **point** (tuple[float, float]), **dist** (float), **project_utm** (Literal[True]), **return_crs** (Literal[True]) → tuple[tuple[float, float, float, float], Any]
- **point** (tuple[float, float]), **dist** (float), **project_utm** (Literal[True]), **return_crs** (Literal[False]) → tuple[float, float, float, float]
- **point** (tuple[float, float]), **dist** (float), **project_utm** (Literal[False]) → tuple[float, float, float, float]
- **point** (tuple[float, float]), **dist** (float), **project_utm** (Literal[False]), **return_crs** (Literal[True]) → tuple[float, float, float, float]
- **point** (tuple[float, float]), **dist** (float), **project_utm** (Literal[False]), **return_crs** (Literal[False]) → tuple[float, float, float, float]

Parameters

- **point** (*tuple[float, float]*)
- **dist** (*float*)
- **project_utm** (*bool*)
- **return_crs** (*bool*)

Return type
tuple[float, float, float, float] | tuple[tuple[float, float, float, float], *Any*]

Create a bounding box some distance (in meters) in each direction (top, bottom, right, and left) from the center point and optionally project it.

Parameters

- **point** (tuple[float, float]) – The *(lat, lon)* center point to create the bounding box around.
- **dist** (float) – 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* is True, then return the projected CRS too.

Returns
bbox or bbox, crs – (left, bottom, right, top) or *((left, bottom, right, top), crs)*.

Return type
tuple[float, float, float, float] | tuple[tuple[float, float, float, float], *Any*]

osmnx.utils_geo.**bbox_to_poly**(*bbox*)

Convert bounding box coordinates to Shapely Polygon.

Parameters
bbox (tuple[float, float, float, float]) – Bounding box as *(left, bottom, right, top)*.

Returns
polygon – The resulting bounding box polygon.

Return type
shapely.Polygon

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

Buffer an unprojected Shapely geometry by some distance in meters.

Parameters

- **geom** (Geometry) – The geometry to be buffered. Coordinates should be in unprojected latitude-longitude degrees (EPSG:4326).
- **dist** (float) – The buffer distance in meters.

Returns
geometry_buff – The (also unprojected) buffered geometry.

Return type
shapely.Geometry

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** (LineString | MultiLineString) – A LineString geometry.
- **dist** (float) – Spacing distance between interpolated points, in same units as *geom*. Smaller values accordingly generate more points.

Yields
point – Interpolated point's *(x, y)* coordinates.

Return type
Iterator[tuple[float, float]]

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** (MultiGraph) – Graph from which to sample points. Should be undirected (to avoid oversampling 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 sampled.

Return type

geopandas.GeoSeries

6.4.24 osmnx._validate module

Validate that graphs and GeoDataFrames satisfy OSMnx expectations.

osmnx._validate.**_report_validation**(*is_valid*, *valid_msg*, *warn_msg*, *err_msg*)

Report validation results by logging, warning, or raising an exception.

Parameters

- **is_valid** (bool) – Whether or not the validation succeeded.
- **valid_msg** (str) – The message to log if validation succeeded.
- **warn_msg** (str) – Any warning messages to log and either issue a warning or include in error message.
- **err_msg** (str) – Any error messages to include when raising exception if validation failed.

Return type

None

osmnx._validate.**_validate_edges**(*G*, *strict*)

Validate that a graph's edges satisfy OSMnx expectations.

Parameters

- **G** (MultiDiGraph) – The input graph.
- **strict** (bool) – If *True*, elevate warnings to errors.

Returns

is_valid, err_msg, warn_msg – Whether validation passed, plus any error or warning messages.

Return type

tuple[bool, str, str]

osmnx._validate.**_validate_features_gdf**(*gdf*)

Validate that features GeoDataFrame satisfies OSMnx expectations.

Raises a *ValidationError* if validation fails.

Parameters
gdf (GeoDataFrame) – GeoDataFrame of features uniquely multi-indexed by *(element_type, osmid)*.

Return type
None

osmnx._validate.**_validate_graph**(*G*, *, *strict=True*)

Validate that a graph object satisfies OSMnx expectations.

Raises *ox._errors.ValidationError* if validation fails.

Parameters

- **G** (MultiDiGraph) – The input graph.
- **strict** (bool) – If *True*, elevate warnings to errors.

Return type
None

osmnx._validate.**_validate_graph_attrs**(*G*)

Validate that a graph's attributes satisfy OSMnx expectations.

Parameters
G (MultiDiGraph) – The input graph.

Returns
is_valid, err_msg, warn_msg – Whether validation passed, plus any error or warning messages.

Return type
tuple[bool, str, str]

osmnx._validate.**_validate_node_edge_gdfs**(*gdf_nodes*, *gdf_edges*, *, *strict=True*)

Validate that node/edge GeoDataFrames can be converted to a MultiDiGraph.

Raises a *ValidationError* if validation fails.

Parameters

- **gdf_nodes** (GeoDataFrame) – GeoDataFrame of graph nodes uniquely indexed by *osmid*.
- **gdf_edges** (GeoDataFrame) – GeoDataFrame of graph edges uniquely multi-indexed by *(u, v, key)*.
- **strict** (bool) – If *True*, elevate warnings to errors.

Return type
None

osmnx._validate.**_validate_nodes**(*G*, *strict*)

Validate that a graph's nodes satisfy OSMnx expectations.

Parameters

- **G** (MultiDiGraph) – The input graph.
- **strict** (bool) – If *True*, elevate warnings to errors.

Returns
is_valid, err_msg, warn_msg – Whether validation passed, plus any error or warning messages.

Return type
tuple[bool, str, str]

osmnx._validate.**_verify_numeric_edge_attribute**(*G*, *attr*, *, *strict=True*)

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

Raises a ValidationError if this attribute contains non-numeric values, and issues a UserWarning if this attribute is missing or null on any edges.

Parameters

- **G** (MultiDiGraph) – Input graph.
- **attr** (str) – Name of the edge attribute to verify.
- **strict** (bool) – If *True*, elevate warnings to errors.

Return type
None

6.5 Further Reading

Boeing, G. (2025). Modeling and Analyzing Urban Networks and Amenities with OSMnx. *Geographical Analysis* 57 (4), 567-577. doi:10.1111/gean.70009

This is the official reference paper and citation for the OSMnx package.

Boeing, G. (2026). Urban Science Beyond Samples: Up-to-Date Street Network Models and Indicators for Every Urban Area in the World. *Environment and Planning B*, published online ahead of print. doi:10.1177/23998083261446991

This study uses OSMnx to model and analyze the street networks of every urban area in the world: over 180 million OpenStreetMap street network nodes and over 360 million edges across 10,351 urban areas in 189 countries.

Boeing, G. (2025). Topological Graph Simplification Solutions to the Street Intersection Miscount Problem. *Transactions in GIS* 29 (3), e70037. doi:10.1111/tgis.70037

This paper describes and validates the algorithms implemented in OSMnx's simplification module and explains why graph simplification is necessary to accurately measure intersection density, street segment length, node degree, etc.

Boeing, G. (2020). Planarity and Street Network Representation in Urban Form Analysis. *Environment and Planning B* 47 (5), 855-869. doi:10.1177/2399808318802941

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

Boeing, G. (2020). The Right Tools for the Job: The Case for Spatial Science Tool-Building. *Transactions in GIS* 24 (5), 1299-1314. doi:10.1111/tgis.12678

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 early development of OSMnx and reviews its use in scientific research over the previous few years.

CHAPTER

SEVEN

INDICES

- genindex
- modindex
- search

PYTHON MODULE INDEX

O

osmnx.bearing, 17
osmnx.convert, 19
osmnx.distance, 22
osmnx.elevation, 26
osmnx.features, 27
osmnx.geocoder, 30
osmnx.graph, 31
osmnx.io, 36
osmnx.plot, 38
osmnx.projection, 42
osmnx.routing, 44
osmnx.settings, 46
osmnx.simplification, 48
osmnx.stats, 51
osmnx.truncate, 54
osmnx.utils, 55
osmnx.utils_geo, 56

INDEX

A

add_edge_bearings() (*in module osmnx.bearing*), 17
add_edge_grades() (*in module osmnx.elevation*), 26
add_edge_lengths() (*in module osmnx.distance*), 22
add_edge_speeds() (*in module osmnx.routing*), 44
add_edge_travel_times() (*in module osmnx.routing*), 44
add_node_elevations_google() (*in module osmnx.elevation*), 26
add_node_elevations_raster() (*in module osmnx.elevation*), 26

B

basic_stats() (*in module osmnx.stats*), 51
bbox_from_point() (*in module osmnx.utils_geo*), 56
bbox_to_poly() (*in module osmnx.utils_geo*), 57
buffer_geometry() (*in module osmnx.utils_geo*), 57

C

calculate_bearing() (*in module osmnx.bearing*), 18
circuity_avg() (*in module osmnx.stats*), 52
citation() (*in module osmnx.utils*), 55
consolidate_intersections() (*in module osmnx.simplification*), 48
count_streets_per_node() (*in module osmnx.stats*), 52

E

edge_length_total() (*in module osmnx.stats*), 52
euclidean() (*in module osmnx.distance*), 22

F

features_from_address() (*in module osmnx.features*), 27
features_from_bbox() (*in module osmnx.features*), 27
features_from_place() (*in module osmnx.features*), 28
features_from_point() (*in module osmnx.features*), 28
features_from_polygon() (*in module osmnx.features*), 29
features_from_xml() (*in module osmnx.features*), 29

G

geocode() (*in module osmnx.geocoder*), 30
geocode_to_gdf() (*in module osmnx.geocoder*), 30
get_colors() (*in module osmnx.plot*), 38
get_edge_colors_by_attr() (*in module osmnx.plot*), 38
get_node_colors_by_attr() (*in module osmnx.plot*), 38
graph_from_address() (*in module osmnx.graph*), 31
graph_from_bbox() (*in module osmnx.graph*), 32
graph_from_gdfs() (*in module osmnx.convert*), 19
graph_from_place() (*in module osmnx.graph*), 32
graph_from_point() (*in module osmnx.graph*), 33
graph_from_polygon() (*in module osmnx.graph*), 34
graph_from_xml() (*in module osmnx.graph*), 35
graph_to_gdfs() (*in module osmnx.convert*), 19
great_circle() (*in module osmnx.distance*), 23

I

interpolate_points() (*in module osmnx.utils_geo*), 58
intersection_count() (*in module osmnx.stats*), 52
is_projected() (*in module osmnx.projection*), 42

K

k_shortest_paths() (*in module osmnx.routing*), 45

L

largest_component() (*in module osmnx.truncate*), 54
load_graphml() (*in module osmnx.io*), 36
log() (*in module osmnx.utils*), 56

M

module
    osmnx.bearing, 17
    osmnx.convert, 19
    osmnx.distance, 22
    osmnx.elevation, 26
    osmnx.features, 27
    osmnx.geocoder, 30
    osmnx.graph, 31

osmnx.io, 36
osmnx.plot, 38
osmnx.projection, 42
osmnx.routing, 44
osmnx.settings, 46
osmnx.simplification, 48
osmnx.stats, 51
osmnx.truncate, 54
osmnx.utils, 55
osmnx.utils_geo, 56

N

nearest_edges() (*in module osmnx.distance*), 23
nearest_nodes() (*in module osmnx.distance*), 24

O

orientation_entropy() (*in module osmnx.bearing*), 18
osmnx.bearing
    module, 17
osmnx.convert
    module, 19
osmnx.distance
    module, 22
osmnx.elevation
    module, 26
osmnx.features
    module, 27
osmnx.geocoder
    module, 30
osmnx.graph
    module, 31
osmnx.io
    module, 36
osmnx.plot
    module, 38
osmnx.projection
    module, 42
osmnx.routing
    module, 44
osmnx.settings
    module, 46
osmnx.simplification
    module, 48
osmnx.stats
    module, 51
osmnx.truncate
    module, 54
osmnx.utils
    module, 55
osmnx.utils_geo
    module, 56

P

plot_figure_ground() (*in module osmnx.plot*), 39
plot_footprints() (*in module osmnx.plot*), 39
plot_graph() (*in module osmnx.plot*), 40
plot_graph_route() (*in module osmnx.plot*), 41
plot_graph_routes() (*in module osmnx.plot*), 41
plot_orientation() (*in module osmnx.plot*), 41
project_gdf() (*in module osmnx.projection*), 43
project_geometry() (*in module osmnx.projection*), 43
project_graph() (*in module osmnx.projection*), 43

R

route_to_gdf() (*in module osmnx.routing*), 45

S

sample_points() (*in module osmnx.utils_geo*), 58
save_graph_geopackage() (*in module osmnx.io*), 36
save_graph_xml() (*in module osmnx.io*), 37
save_graphml() (*in module osmnx.io*), 37
self_loop_proportion() (*in module osmnx.stats*), 53
shortest_path() (*in module osmnx.routing*), 45
simplify_graph() (*in module osmnx.simplification*), 50
street_length_total() (*in module osmnx.stats*), 53
street_segment_count() (*in module osmnx.stats*), 53
streets_per_node() (*in module osmnx.stats*), 53
streets_per_node_avg() (*in module osmnx.stats*), 53
streets_per_node_counts() (*in module osmnx.stats*), 54
streets_per_node_proportions() (*in module osmnx.stats*), 54

T

to_digraph() (*in module osmnx.convert*), 20
to_undirected() (*in module osmnx.convert*), 21
truncate_graph_bbox() (*in module osmnx.truncate*), 54
truncate_graph_dist() (*in module osmnx.truncate*), 55
truncate_graph_polygon() (*in module osmnx.truncate*), 55
ts() (*in module osmnx.utils*), 56

V

validate_features_gdf() (*in module osmnx.convert*), 21
validate_graph() (*in module osmnx.convert*), 21
validate_node_edge_gdfs() (*in module osmnx.convert*), 21