

---

# OSMnx Documentation

*Release 0.16.2*

**Geoff Boeing**

**Nov 30, 2020**



## REFERENCE

<b>1</b>	<b>osmnx package</b>	<b>1</b>
1.1	osmnx.bearing module . . . . .	1
1.2	osmnx.distance module . . . . .	2
1.3	osmnx.downloader module . . . . .	5
1.4	osmnx.elevation module . . . . .	6
1.5	osmnx.folium module . . . . .	6
1.6	osmnx.footprints module . . . . .	8
1.7	osmnx.geocoder module . . . . .	10
1.8	osmnx.geometries module . . . . .	10
1.9	osmnx.graph module . . . . .	13
1.10	osmnx.io module . . . . .	17
1.11	osmnx.plot module . . . . .	19
1.12	osmnx.pois module . . . . .	23
1.13	osmnx.projection module . . . . .	25
1.14	osmnx.settings module . . . . .	26
1.15	osmnx.simplification module . . . . .	26
1.16	osmnx.speed module . . . . .	28
1.17	osmnx.stats module . . . . .	29
1.18	osmnx.truncate module . . . . .	31
1.19	osmnx.utils module . . . . .	33
1.20	osmnx.utils_geo module . . . . .	35
1.21	osmnx.utils_graph module . . . . .	36
<b>2</b>	<b>Citation info</b>	<b>41</b>
<b>3</b>	<b>Features</b>	<b>43</b>
<b>4</b>	<b>Installation</b>	<b>45</b>
<b>5</b>	<b>Usage examples</b>	<b>47</b>
<b>6</b>	<b>Support</b>	<b>49</b>
<b>7</b>	<b>License</b>	<b>51</b>
<b>8</b>	<b>Indices and tables</b>	<b>53</b>
	<b>Python Module Index</b>	<b>55</b>
	<b>Index</b>	<b>57</b>



## OSMNX PACKAGE

Users' reference for the OSMnx API.

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

## 1.1 osmnx.bearing module

Calculate graph edge bearings.

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

Add *bearing* attributes to all graph edges.

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

**Parameters**

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

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

**Return type** `networkx.MultiDiGraph`

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

Calculate the bearing between two lat-lng points.

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

**Parameters**

- **origin\_point** (*tuple*) – (lat, lng)
- **destination\_point** (*tuple*) – (lat, lng)

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

**Return type** `float`

## 1.2 osmnx.distance module

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

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

Calculate Euclidean distances between points.

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

### Parameters

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

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

**Return type** float or np.array of float

`osmnx.distance.get_nearest_edge(G, point, return_geom=False, return_dist=False)`

Find the nearest edge to a point by minimum Euclidean distance.

### Parameters

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

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

**Return type** tuple

`osmnx.distance.get_nearest_edges(G, X, Y, method=None, dist=0.0001)`

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

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

### Parameters

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

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

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

**Return type** `np.array`

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

Find the nearest node to a point.

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

#### Parameters

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

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

**Return type** `int` or `tuple of (int, float)`

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

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

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

#### Parameters

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

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

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

**Return type** `np.array`

`osmnx.distance.great_circle_vec` (*lat1*, *lng1*, *lat2*, *lng2*, *earth\_radius=6371009*)

Calculate great-circle distances between points.

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

#### Parameters

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

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

**Return type** `float` or `np.array`

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

Get *k* shortest paths from origin node to destination node.

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

#### Parameters

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

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

**Return type** `generator`

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

Get shortest path from origin node to destination node.

See also `k_shortest_paths` to get multiple shortest paths.

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

#### Parameters

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



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

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

**Return type** list

## 1.3 osmnx.downloader module

Interact with the OSM APIs.

`osmnx.downloader.nominatim_request` (*params*, *request\_type='search'*, *pause=1*, *error\_pause=60*)

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

### Parameters

- **params** (*OrderedDict*) – key-value pairs of parameters
- **request\_type** (*string*) – Type of Nominatim query. One of: search, reverse, or lookup
- **pause** (*int*) – how long to pause before request, in seconds. per the nominatim usage policy: “an absolute maximum of 1 request per second” is allowed
- **error\_pause** (*int*) – how long to pause in seconds before re-trying request if error

**Returns** **response\_json**

**Return type** dict

`osmnx.downloader.overpass_request` (*data*, *pause=None*, *error\_pause=60*)

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

### Parameters

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

**Returns** **response\_json**

**Return type** dict

## 1.4 osmnx.elevation module

Get node elevations and calculate edge grades.

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

Add *grade* attribute to each graph edge.

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

See also the *add\_node\_elevations* function.

### Parameters

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

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

**Return type** *networkx.MultiDiGraph*

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

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

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

See also the *add\_edge\_grades* function.

### Parameters

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

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

**Return type** *networkx.MultiDiGraph*

## 1.5 osmnx.folium module

Create leaflet web maps via folium.

```
osmnx.folium.plot_graph_folium(G, graph_map=None, popup_attribute=None,
                                tiles='cartodbpositron', zoom=1, fit_bounds=True,
                                edge_color='#333333', edge_width=5, edge_opacity=1,
                                **kwargs)
```

Plot a graph on an interactive folium web map.

Note that anything larger than a small city can take a long time to plot and create a large web map file that is very slow to load as JavaScript.

**Parameters**

- **G** (*networkx.MultiDiGraph*) – input graph
- **graph\_map** (*folium.folium.Map* or *folium.FeatureGroup*) – if not None, plot the graph on this preexisting folium map object
- **popup\_attribute** (*string*) – edge attribute to display in a pop-up when an edge is clicked
- **tiles** (*string*) – name of a folium tileset
- **zoom** (*int*) – initial zoom level for the map
- **fit\_bounds** (*bool*) – if True, fit the map to the boundaries of the route's edges
- **edge\_color** (*string*) – color of the edge lines
- **edge\_width** (*numeric*) – width of the edge lines
- **edge\_opacity** (*numeric*) – opacity of the edge lines
- **kwargs** (*dict*) – Extra keyword arguments passed through to folium

**Returns graph\_map**

**Return type** folium.folium.Map

```
osmnx.folium.plot_route_folium(G, route, route_map=None, popup_attribute=None,
                               tiles='cartodbpositron', zoom=1, fit_bounds=True,
                               route_color='#cc0000', route_width=5, route_opacity=1,
                               **kwargs)
```

Plot a route on an interactive folium web map.

**Parameters**

- **G** (*networkx.MultiDiGraph*) – input graph
- **route** (*list*) – the route as a list of nodes
- **route\_map** (*folium.folium.Map*) – if not None, plot the route on this preexisting folium map object
- **popup\_attribute** (*string*) – edge attribute to display in a pop-up when an edge is clicked
- **tiles** (*string*) – name of a folium tileset
- **zoom** (*int*) – initial zoom level for the map
- **fit\_bounds** (*bool*) – if True, fit the map to the boundaries of the route's edges
- **route\_color** (*string*) – color of the route's line
- **route\_width** (*numeric*) – width of the route's line
- **route\_opacity** (*numeric*) – opacity of the route lines
- **kwargs** (*dict*) – Extra parameters passed through to folium

**Returns route\_map**

**Return type** folium.folium.Map

## 1.6 osmnx.footprints module

Download footprints from OpenStreetMap.

Deprecated: use the new *geometries* module instead.

`osmnx.footprints.footprints_from_address` (*address*, *dist=1000*, *footprint\_type='building'*, *retain\_invalid=False*)

Get footprints within some distance N, S, E, W of an address.

Deprecated: use *geometries* module instead.

### Parameters

- **address** (*string*) – the address to geocode to a lat-lng point
- **dist** (*numeric*) – distance in meters
- **footprint\_type** (*string*) – type of footprint to be downloaded. OSM tag key e.g. ‘building’, ‘landuse’, ‘place’, etc.
- **retain\_invalid** (*bool*) – deprecated, is ignored

### Returns

**Return type** `geopandas.GeoDataFrame`

### Notes

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

`osmnx.footprints.footprints_from_place` (*place*, *footprint\_type='building'*, *retain\_invalid=False*, *which\_result=None*)

Get footprints within the boundaries of some place.

Deprecated: use *geometries* module instead.

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 footprints using the `footprints_from_address` function, which geocodes the place name to a point and gets the footprints within some distance of that point.

### Parameters

- **place** (*string*) – the query to geocode to get place boundary polygon
- **footprint\_type** (*string*) – type of footprint to be downloaded. OSM tag key e.g. ‘building’, ‘landuse’, ‘place’, etc.
- **retain\_invalid** (*bool*) – deprecated, is ignored
- **which\_result** (*int*) – which geocoding result to use. if `None`, auto-select the first multi/polygon or raise an error if OSM doesn’t return one.

### Returns

**Return type** `geopandas.GeoDataFrame`

## Notes

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

`osmnx.footprints.footprints_from_point` (*point*, *dist=1000*, *footprint\_type='building'*, *retain\_invalid=False*)

Get footprints within some distance N, S, E, W of a lat-lng point.

Deprecated: use geometries module instead.

### Parameters

- **point** (*tuple*) – a lat-lng point
- **dist** (*numeric*) – distance in meters
- **footprint\_type** (*string*) – type of footprint to be downloaded. OSM tag key e.g. 'building', 'landuse', 'place', etc.
- **retain\_invalid** (*bool*) – deprecated, is ignored

### Returns

**Return type** `geopandas.GeoDataFrame`

## Notes

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

`osmnx.footprints.footprints_from_polygon` (*polygon*, *footprint\_type='building'*, *retain\_invalid=False*)

Get footprints within some polygon.

Deprecated: use geometries module instead.

### Parameters

- **polygon** (*shapely.geometry.Polygon or shapely.geometry.MultiPolygon*) – the shape to get data within. coordinates should be in units of latitude-longitude degrees.
- **footprint\_type** (*string*) – type of footprint to be downloaded. OSM tag key e.g. 'building', 'landuse', 'place', etc.
- **retain\_invalid** (*bool*) – deprecated, is ignored

### Returns

**Return type** `geopandas.GeoDataFrame`

## Notes

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

## 1.7 osmnx.geocoder module

Geocode queries and create GeoDataFrames of place boundaries.

`osmnx.geocoder.geocode(query)`

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

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

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

**Return type** tuple

`osmnx.geocoder.geocode_to_gdf(query, which_result=None, buffer_dist=None)`

Geocode a query or queries to a GeoDataFrame with the Nominatim geocoder.

Geometry column contains place boundaries if they exist in OpenStreetMap. Query can be a string or dict, or a list of strings/dicts to send to the geocoder. If query is a list, then which\_result should be either a single value or a list of the same length as query.

**Parameters**

- **query** (*string or dict or list*) – query string(s) or structured dict(s) to geocode
- **which\_result** (*int*) – which geocoding result to use. if None, auto-select the first multi/polygon or raise an error if OSM doesn't return one.
- **buffer\_dist** (*float*) – distance to buffer around the place geometry, in meters

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

**Return type** `geopandas.GeoDataFrame`

## 1.8 osmnx.geometries module

Download geospatial entities' geometries and attributes from OpenStreetMap.

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

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

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

**Parameters**

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

**Returns** `gdf`

**Return type** `geopandas.GeoDataFrame`

## Notes

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

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

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

### Parameters

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

### Returns gdf

**Return type** `geopandas.GeoDataFrame`

## Notes

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

`osmnx.geometries.geometries_from_place` (*query, tags, which\_result=None, buffer\_dist=None*)

Create a GeoDataFrame of OSM entities within the boundaries of a place.

### Parameters

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

### Returns gdf

**Return type** `geopandas.GeoDataFrame`

## Notes

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

`osmnx.geometries.geometries_from_point` (*center\_point*, *tags*, *dist=1000*)

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

### Parameters

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

### Returns `gdf`

**Return type** `geopandas.GeoDataFrame`

## Notes

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

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

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

### Parameters

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

### Returns `gdf`

**Return type** `geopandas.GeoDataFrame`



## Notes

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

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

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

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

### Parameters

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

### Returns gdf

**Return type** `geopandas.GeoDataFrame`

## 1.9 osmnx.graph module

Graph creation functions.

`osmnx.graph.graph_from_address` (*address*, *dist=1000*, *dist\_type='bbox'*, *network\_type='all\_private'*, *simplify=True*, *retain\_all=False*, *truncate\_by\_edge=False*, *return\_coords=False*, *clean\_periphery=True*, *custom\_filter=None*)

Create a graph from OSM within some distance of some address.

### Parameters

- **address** (*string*) – the address to geocode and use as the central point around which to construct the graph
- **dist** (*int*) – retain only those nodes within this many meters of the center of the graph
- **dist\_type** (*string*) – {'network', 'bbox'} if 'bbox', retain only those nodes within a bounding box of the distance parameter. if 'network', retain only those nodes within some network distance from the center-most node.
- **network\_type** (*string*) – what type of street network to get if *custom\_filter* is None. One of 'walk', 'bike', 'drive', 'drive\_service', 'all', or 'all\_private'.
- **simplify** (*bool*) – if True, simplify graph topology with the *simplify\_graph* function
- **retain\_all** (*bool*) – if True, return the entire graph even if it is not connected. otherwise, retain only the largest weakly connected component.

- **truncate\_by\_edge** (*bool*) – if True, retain nodes outside bounding box if at least one of node’s neighbors is within the bounding box
- **return\_coords** (*bool*) – optionally also return the geocoded coordinates of the address
- **clean\_periphery** (*bool*,) – if True, buffer 500m to get a graph larger than requested, then simplify, then truncate it to requested spatial boundaries
- **custom\_filter** (*string*) – a custom network filter to be used instead of the `network_type` presets, e.g., `[“power”~”line”]` or `[“highway”~”motorway/trunk”]`. Also pass in a `network_type` that is in `settings.bidirectional_network_types` if you want graph to be fully bi-directional.

### Returns

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

### Notes

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

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

Create a graph from OSM within some bounding box.

### Parameters

- **north** (*float*) – northern latitude of bounding box
- **south** (*float*) – southern latitude of bounding box
- **east** (*float*) – eastern longitude of bounding box
- **west** (*float*) – western longitude of bounding box
- **network\_type** (*string*) – what type of street network to get if `custom_filter` is None. One of `‘walk’`, `‘bike’`, `‘drive’`, `‘drive_service’`, `‘all’`, or `‘all_private’`.
- **simplify** (*bool*) – if True, simplify graph topology with the `simplify_graph` function
- **retain\_all** (*bool*) – if True, return the entire graph even if it is not connected. otherwise, retain only the largest weakly connected component.
- **truncate\_by\_edge** (*bool*) – if True, retain nodes outside bounding box if at least one of node’s neighbors is within the bounding box
- **clean\_periphery** (*bool*) – if True, buffer 500m to get a graph larger than requested, then simplify, then truncate it to requested spatial boundaries
- **custom\_filter** (*string*) – a custom network filter to be used instead of the `network_type` presets, e.g., `[“power”~”line”]` or `[“highway”~”motorway/trunk”]`. Also pass in a `network_type` that is in `settings.bidirectional_network_types` if you want graph to be fully bi-directional.

### Returns G

**Return type** `networkx.MultiDiGraph`

## Notes

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

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

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

The query must be geocodable and OSM must have polygon boundaries for the geocode result. If OSM does not have a polygon for this place, you can instead get its street network using the `graph_from_address` function, which geocodes the place name to a point and gets the network within some distance of that point. Alternatively, you might try to vary the `which_result` parameter to use a different geocode result. For example, the first geocode result (ie, the default) might resolve to a point geometry, but the second geocode result for this query might resolve to a polygon, in which case you can use `graph_from_place` with `which_result=2`. `which_result=None` will auto-select the first multi/polygon among the geocoding results.

### Parameters

- **query** (*string or dict or list*) – the query or queries to geocode to get place boundary polygon(s)
- **network\_type** (*string*) – what type of street network to get if `custom_filter` is `None`. One of 'walk', 'bike', 'drive', 'drive\_service', 'all', or 'all\_private'.
- **simplify** (*bool*) – if `True`, simplify graph topology with the `simplify_graph` function
- **retain\_all** (*bool*) – if `True`, return the entire graph even if it is not connected. otherwise, retain only the largest weakly connected component.
- **truncate\_by\_edge** (*bool*) – if `True`, retain nodes outside boundary polygon if at least one of node's neighbors is within the polygon
- **which\_result** (*int*) – which geocoding result to use. if `None`, auto-select the first multi/polygon or raise an error if OSM doesn't return one.
- **buffer\_dist** (*float*) – distance to buffer around the place geometry, in meters
- **clean\_periphery** (*bool*) – if `True`, buffer 500m to get a graph larger than requested, then simplify, then truncate it to requested spatial boundaries
- **custom\_filter** (*string*) – a custom network filter to be used instead of the `network_type` presets, e.g., `["power"~"line"]` or `["highway"~"motorway|trunk"]`. Also pass in a `network_type` that is in `settings.bidirectional_network_types` if you want graph to be fully bi-directional.

### Returns G

**Return type** `networkx.MultiDiGraph`

## Notes

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

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

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

### Parameters

- **center\_point** (*tuple*) – the (lat, lng) center point around which to construct the graph

- **dist** (*int*) – retain only those nodes within this many meters of the center of the graph, with distance determined according to *dist\_type* argument
- **dist\_type** (*string*) – {‘network’, ‘bbox’} if ‘bbox’, retain only those nodes within a bounding box of the distance parameter. if ‘network’, retain only those nodes within some network distance from the center-most node.
- **network\_type** (*string*) – what type of street network to get if *custom\_filter* is None. One of ‘walk’, ‘bike’, ‘drive’, ‘drive\_service’, ‘all’, or ‘all\_private’.
- **simplify** (*bool*) – if True, simplify graph topology with the *simplify\_graph* function
- **retain\_all** (*bool*) – if True, return the entire graph even if it is not connected. otherwise, retain only the largest weakly connected component.
- **truncate\_by\_edge** (*bool*) – if True, retain nodes outside bounding box if at least one of node’s neighbors is within the bounding box
- **clean\_periphery** (*bool*,) – if True, buffer 500m to get a graph larger than requested, then simplify, then truncate it to requested spatial boundaries
- **custom\_filter** (*string*) – a custom network filter to be used instead of the *network\_type* presets, e.g., [“power”~”line”] or [“highway”~”motorway/trunk”]. Also pass in a *network\_type* that is in *settings.bidirectional\_network\_types* if you want graph to be fully bi-directional.

#### Returns G

**Return type** `networkx.MultiDiGraph`

#### Notes

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

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

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

#### Parameters

- **polygon** (*shapely.geometry.Polygon or shapely.geometry.MultiPolygon*) – the shape to get network data within. coordinates should be in units of latitude-longitude degrees.
- **network\_type** (*string*) – what type of street network to get if *custom\_filter* is None. One of ‘walk’, ‘bike’, ‘drive’, ‘drive\_service’, ‘all’, or ‘all\_private’.
- **simplify** (*bool*) – if True, simplify graph topology with the *simplify\_graph* function
- **retain\_all** (*bool*) – if True, return the entire graph even if it is not connected. otherwise, retain only the largest weakly connected component.
- **truncate\_by\_edge** (*bool*) – if True, retain nodes outside boundary polygon if at least one of node’s neighbors is within the polygon
- **clean\_periphery** (*bool*) – if True, buffer 500m to get a graph larger than requested, then simplify, then truncate it to requested spatial boundaries
- **custom\_filter** (*string*) – a custom network filter to be used instead of the *network\_type* presets, e.g., [“power”~”line”] or [“highway”~”motorway/trunk”]. Also pass

in a `network_type` that is in `settings.bidirectional_network_types` if you want graph to be fully bi-directional.

**Returns** `G`

**Return type** `networkx.MultiDiGraph`

## Notes

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

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

Create a graph from data in an OSM-formatted XML file.

### Parameters

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

**Returns** `G`

**Return type** `networkx.MultiDiGraph`

## 1.10 osmnx.io module

Serialize graphs to/from files on disk.

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

Load an OSMnx-saved GraphML file from disk.

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

### Parameters

- **filepath** (*string*) – path to the GraphML file
- **node\_type** (*None*) – deprecated, do not use; use `node_dtypes` instead
- **node\_dtypes** (*dict*) – dict of node attribute names:types to convert values' data types
- **edge\_dtypes** (*dict*) – dict of edge attribute names:types to convert values' data types

**Returns** `G`

**Return type** `networkx.MultiDiGraph`

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

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

### Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **filepath** (*string*) – path to the GeoPackage file including extension. if None, use default data folder + `graph.gpkg`
- **encoding** (*string*) – the character encoding for the saved file

**Returns****Return type** None`osmnx.io.save_graph_shapefile(G, filepath=None, encoding='utf-8')`

Save graph nodes and edges to disk as ESRI shapefiles.

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

**Parameters**

- **G** (*networkx.MultiDiGraph*) – input graph
- **filepath** (*string*) – path to the shapefiles folder (no file extension). if None, use default data folder + `graph_shapefile`
- **encoding** (*string*) – the character encoding for the saved files

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

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

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

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

**Example**

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

**Parameters**

- **data** (*networkx multi(di)graph OR a length 2 iterable of nodes/edges*) – geopandas GeoDataFrames
- **filepath** (*string*) – path to the .osm file including extension. if None, use default data folder + `graph.osm`
- **node\_tags** (*list*) – osm node tags to include in output OSM XML

- **node\_attrs** (*list*) – osm node attributes to include in output OSM XML
- **edge\_tags** (*list*) – osm way tags to include in output OSM XML
- **edge\_attrs** (*list*) – osm way attributes to include in output OSM XML
- **oneway** (*bool*) – the default oneway value used to fill this tag where missing
- **merge\_edges** (*bool*) – if True merges graph edges such that each OSM way has one entry and one entry only in the OSM XML. Otherwise, every OSM way will have a separate entry for each node pair it contains.
- **edge\_tag\_aggs** (*list of length-2 string tuples*) – useful only if `merge_edges` is True, this argument allows the user to specify edge attributes to aggregate such that the merged OSM way entry tags accurately represent the sum total of their component edge attributes. For example, if the user wants the OSM way to have a “length” attribute, the user must specify `edge_tag_aggs=[('length', 'sum')]` in order to tell this method to aggregate the lengths of the individual component edges. Otherwise, the length attribute will simply reflect the length of the first edge associated with the way.

**Returns****Return type** None

`osmnx.io.save_graphml(G, filepath=None, gephi=False, encoding='utf-8')`  
 Save graph to disk as GraphML file.

**Parameters**

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

**Returns****Return type** None

## 1.11 osmnx.plot module

Plot spatial geometries, street networks, and routes.

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

**Parameters**

- **n** (*int*) – number of colors
- **cmap** (*string*) – name of a matplotlib colormap
- **start** (*float*) – where to start in the colorspace
- **stop** (*float*) – where to end in the colorspace
- **alpha** (*float*) – opacity, the alpha channel for the RGBa colors

- **return\_hex** (*bool*) – if True, convert RGBA colors to HTML-like hexadecimal RGB strings. if False, return colors as (R, G, B, alpha) tuples.

**Returns** `color_list`

**Return type** `list`

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

Get colors based on edge attribute values.

**Parameters**

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

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

**Return type** `pandas.Series`

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

Get colors based on node attribute values.

**Parameters**

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

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

**Return type** `pandas.Series`

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

Plot a figure-ground diagram of a street network.



**Parameters**

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

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

**Return type** tuple

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

Plot a GeoDataFrame of geospatial entities' footprints.

**Parameters**

- **gdf** (*geopandas.GeoDataFrame*) – GeoDataFrame of footprints (shapely Polygons and MultiPolygons)
- **ax** (*axis*) – if not None, plot on this preexisting axis
- **figsize** (*tuple*) – if ax is None, create new figure with size (width, height)
- **color** (*string*) – color of the footprints
- **bgcolor** (*string*) – background color of the plot
- **bbox** (*tuple*) – bounding box as (north, south, east, west). if None, will calculate from the spatial extents of the geometries in gdf
- **save** (*bool*) – if True, save the figure to disk at filepath
- **show** (*bool*) – if True, call pyplot.show() to show the figure
- **close** (*bool*) – if True, call pyplot.close() to close the figure
- **filepath** (*string*) – if save is True, the path to the file. file format determined from extension. if None, use settings.imgs\_folder/image.png
- **dpi** (*int*) – if save is True, the resolution of saved file

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

**Return type** tuple

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

Plot a graph.

#### Parameters

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

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

**Return type** *tuple*

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

Plot a route along a graph.

#### Parameters

- **G** (*networkx.MultiDiGraph*) – input graph
- **route** (*list*) – route as a list of node IDs
- **route\_color** (*string*) – color of the route
- **route\_linewidth** (*int*) – width of the route line
- **route\_alpha** (*float*) – opacity of the route line

- **orig\_dest\_size** (*int*) – size of the origin and destination nodes
- **ax** (*matplotlib axis*) – if not None, plot route on this preexisting axis instead of creating a new fig, ax and drawing the underlying graph
- **pg\_kwargs** – keyword arguments to pass to `plot_graph`

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

**Return type** tuple

`osmnx.plot.plot_graph_routes` (*G*, *routes*, *route\_colors='r'*, *\*\*pgr\_kwargs*)  
Plot several routes along a graph.

#### Parameters

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

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

**Return type** tuple

## 1.12 osmnx.pois module

Download points of interests (POIs) from OpenStreetMap.

Deprecated: use the new *geometries* module instead.

`osmnx.pois.pois_from_address` (*address*, *tags*, *dist=1000*)  
Get point of interests (POIs) within some distance N, S, E, W of address.

Deprecated: use *geometries* module instead.

#### Parameters

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

**Returns** **gdf**

**Return type** `geopandas.GeoDataFrame`

## Notes

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

`osmnx.pois.pois_from_place` (*place*, *tags*, *which\_result=None*)

Get points of interest (POIs) within the boundaries of some place.

Deprecated: use geometries module instead.

### Parameters

- **place** (*string*) – the query to geocode to get place boundary polygon
- **tags** (*dict*) – Dict of tags used for finding POIs from the selected area. Results returned are the union, not intersection of each individual tag. Each result matches at least one tag given. The dict keys should be OSM tags, (e.g., *amenity*, *landuse*, *highway*, etc) and the dict values should be either *True* to retrieve all items with the given tag, or a string to get a single tag-value combination, or a list of strings to get multiple values for the given tag. For example, `tags = {'amenity':True, 'landuse':['retail','commercial'], 'highway':'bus_stop'}` would return all amenities, landuse=retail, landuse=commercial, and highway=bus\_stop.
- **which\_result** (*int*) – which geocoding result to use. if *None*, auto-select the first multi/polygon or raise an error if OSM doesn't return one.

### Returns gdf

**Return type** `geopandas.GeoDataFrame`

## Notes

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

`osmnx.pois.pois_from_point` (*point*, *tags*, *dist=1000*)

Get point of interests (POIs) within some distance N, S, E, W of a point.

Deprecated: use geometries module instead.

### Parameters

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

### Returns gdf

**Return type** `geopandas.GeoDataFrame`

## Notes

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

`osmnx.pois.pois_from_polygon(polygon, tags)`

Get point of interests (POIs) within some polygon.

Deprecated: use geometries module instead.

### Parameters

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

### Returns gdf

**Return type** `geopandas.GeoDataFrame`

## Notes

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

## 1.13 osmnx.projection module

Project spatial geometries and street networks.

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

Project a GeoDataFrame from its current CRS to another.

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

### Parameters

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

**Returns gdf\_proj** – the projected GeoDataFrame

**Return type** `geopandas.GeoDataFrame`

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

Project a shapely geometry from its current CRS to another.

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

**Parameters**

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

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

**Return type** tuple

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

Project graph from its current CRS to another.

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

**Parameters**

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

**Returns** **G\_proj** – the projected graph

**Return type** networkx.MultiDiGraph

## 1.14 osmnx.settings module

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

## 1.15 osmnx.simplification module

Simplify, correct, and consolidate network topology.

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

Consolidate intersections comprising clusters of nearby nodes.

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

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

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

#### Parameters

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

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

**Return type** *networkx.MultiDiGraph* or *geopandas.GeoSeries*

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

Simplify a graph’s topology by removing interstitial nodes.

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

#### Parameters

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

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

**Return type** *networkx.MultiDiGraph*

## 1.16 osmnx.speed module

Calculate graph edge speeds and travel times.

`osmnx.speed.add_edge_speeds` (*G*, *hwy\_speeds*=None, *fallback*=None, *precision*=1)

Add edge speeds (km per hour) to graph as new *speed\_kph* edge attributes.

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

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

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

### Parameters

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

**Returns** *G* – graph with *speed\_kph* attributes on all edges

**Return type** *networkx.MultiDiGraph*

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

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

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

### Parameters

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

**Returns** *G* – graph with *travel\_time* attributes on all edges

**Return type** *networkx.MultiDiGraph*



## 1.17 osmnx.stats module

Calculate graph-theoretic network measures.

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

Calculate basic descriptive metric and topological stats for a graph.

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

### Parameters

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

### Returns

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

- **n** = number of nodes in the graph
- **m** = number of edges in the graph
- **k\_avg** = average node degree of the graph
- **intersection\_count** = number of intersections in graph, that is, nodes with >1 street emanating from them
- **streets\_per\_node\_avg** = how many streets (edges in the undirected representation of the graph) emanate from each node (ie, intersection or dead-end) on average (mean)
- **streets\_per\_node\_counts** = dict, with keys of number of streets emanating from the node, and values of number of nodes with this count
- **streets\_per\_node\_proportion** = dict, same as previous, but as a proportion of the total, rather than counts
- **edge\_length\_total** = sum of all edge lengths in the graph, in meters
- **edge\_length\_avg** = mean edge length in the graph, in meters
- **street\_length\_total** = sum of all edges in the undirected representation of the graph
- **street\_length\_avg** = mean edge length in the undirected representation of the graph, in meters
- **street\_segments\_count** = number of edges in the undirected representation of the graph
- **node\_density\_km** = n divided by area in square kilometers
- **intersection\_density\_km** = intersection\_count divided by area in square kilometers
- **edge\_density\_km** = edge\_length\_total divided by area in square kilometers

- **street\_density\_km** = **street\_length\_total** divided by **area in square** kilometers
- **circuitry\_avg** = **edge\_length\_total** divided by the **sum of the great circle** distances between the nodes of each edge
- **self\_loop\_proportion** = **proportion of edges that have a single node** as its two endpoints (ie, the edge links nodes *u* and *v*, and *u==v*)
- **clean\_intersection\_count** = **number of intersections in street** network, merging complex ones into single points
- **clean\_intersection\_density\_km** = **clean\_intersection\_count** divided by **area in square** kilometers

**Return type** dict

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

Calculate extended topological stats and metrics for a graph.

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

#### Parameters

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

#### Returns

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

- **avg\_neighbor\_degree**
- **avg\_neighbor\_degree\_avg**
- **avg\_weighted\_neighbor\_degree**
- **avg\_weighted\_neighbor\_degree\_avg**
- **degree\_centrality**
- **degree\_centrality\_avg**
- **clustering\_coefficient**
- **clustering\_coefficient\_avg**
- **clustering\_coefficient\_weighted**
- **clustering\_coefficient\_weighted\_avg**
- **pagerank**
- **pagerank\_max\_node**
- **pagerank\_max**

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

**Return type** dict

## 1.18 osmnx.truncate module

Truncate graph by distance, bounding box, or polygon.

`osmnx.truncate.truncate_graph_bbox` (*G*, *north*, *south*, *east*, *west*, *truncate\_by\_edge=False*, *retain\_all=False*, *quadrat\_width=0.05*, *min\_num=3*)

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

### Parameters

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

**Returns** **G** – the truncated graph

**Return type** `networkx.MultiDiGraph`

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

Remove every node farther than some network distance from `source_node`.

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

#### Parameters

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

**Returns** **G** – the truncated graph

**Return type** `networkx.MultiDiGraph`

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

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

#### Parameters

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

**Returns** **G** – the truncated graph

**Return type** `networkx.MultiDiGraph`

## 1.19 osmnx.utils module

General utility functions.

`osmnx.utils.citation()`

Print the OSMnx package’s citation information.

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

**Returns**

**Return type** None

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

Configure OSMnx by setting the default global settings’ values.

Any parameters not passed by the caller are set to their original default values.

**Parameters**

- **use\_cache** (*bool*) – if True, cache HTTP responses locally instead of calling API repeatedly for the same request
- **cache\_folder** (*string*) – path to folder in which to save/load HTTP response cache
- **data\_folder** (*string*) – path to folder in which to save/load graph files by default
- **imgs\_folder** (*string*) – path to folder in which to save plot images by default
- **logs\_folder** (*string*) – path to folder in which to save log files
- **log\_file** (*bool*) – if True, save log output to a file in logs\_folder
- **log\_console** (*bool*) – if True, print log output to the console (terminal window)
- **log\_level** (*int*) – one of Python’s logger.level constants
- **log\_name** (*string*) – name of the logger
- **log\_filename** (*string*) – name of the log file, without file extension
- **useful\_tags\_node** (*list*) – OSM “node” tags to add as graph node attributes, when present

- **useful\_tags\_way** (*list*) – OSM “way” tags to add as graph edge attributes, when present
- **bidirectional\_network\_types** (*list*) – network types for which a fully bidirectional graph will be created
- **osm\_xml\_node\_attrs** (*list*) – node attributes for saving .osm XML files with `save_graph_xml` function
- **osm\_xml\_node\_tags** (*list*) – node tags for saving .osm XML files with `save_graph_xml` function
- **osm\_xml\_way\_attrs** (*list*) – edge attributes for saving .osm XML files with `save_graph_xml` function
- **osm\_xml\_way\_tags** (*list*) – edge tags for for saving .osm XML files with `save_graph_xml` function
- **all\_oneway** (*bool*) – if True, forces all ways to be loaded as oneway ways, preserving the original order of nodes stored in the OSM way XML. Only use if specifically saving to .osm XML file with `save_graph_xml` function.
- **overpass\_settings** (*string*) – Settings string for overpass queries. For example, to query historical OSM data as of a certain date: `'[out:json][timeout:90][date:"2019-10-28T19:20:00Z"]'`. Use with caution.
- **timeout** (*int*) – the timeout interval for the HTTP request and for API to use while running the query
- **memory** (*int*) – Overpass server memory allocation size for the query, in bytes. If None, server will use its default allocation size. Use with caution.
- **max\_query\_area\_size** (*int*) – maximum area for any part of the geometry in meters: any polygon bigger than this will get divided up for multiple queries to API (default 50km x 50km)
- **default\_access** (*string*) – default filter for OSM “access” key
- **default\_crs** (*string*) – default coordinate reference system to set when creating graphs
- **default\_user\_agent** (*string*) – HTTP header user-agent
- **default\_referer** (*string*) – HTTP header referer
- **default\_accept\_language** (*string*) – HTTP header accept-language
- **nominatim\_endpoint** (*string*) – the API endpoint to use for nominatim queries
- **nominatim\_key** (*string*) – your API key, if you are using an endpoint that requires one
- **overpass\_endpoint** (*string*) – the API endpoint to use for overpass queries
- **elevation\_provider** (*string*) – the API provider to use for adding node elevations, can be either “google” or “airmap”

## Returns

**Return type** None

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

This logs to file and/or prints to the console (terminal), depending on the current configuration of settings.log\_file and settings.log\_console.

#### Parameters

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

#### Returns

**Return type** None

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

Get current timestamp as string.

#### Parameters

- **style** (*string*) – format the timestamp with this built-in template. must be one of { ‘date-time’, ‘date’, ‘time’ }
- **template** (*string*) – if not None, format the timestamp with this template instead of one of the built-in styles

**Returns** `ts` – the string timestamp

**Return type** string

## 1.20 osmnx.utils\_geo module

Geospatial utility functions.

`osmnx.utils_geo.bbox_from_point (point, dist=1000, project_utm=False, return_crs=False)`

Create a bounding box from a (lat, lng) center point.

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

#### Parameters

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

**Returns** (north, south, east, west) or (north, south, east, west, crs\_proj)

**Return type** tuple

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

Convert bounding box coordinates to shapely Polygon.

#### Parameters

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

- **west** (*float*) – western coordinate

**Returns**

**Return type** shapely.geometry.Polygon

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

Redistribute the vertices on a projected LineString or MultiLineString.

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

**Parameters**

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

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

**Return type** list or shapely.geometry.MultiLineString

`osmnx.utils_geo.round_geometry_coords` (*shape, precision*)

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

**Parameters**

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

**Returns**

**Return type** shapely.geometry.geometry

## 1.21 osmnx.utils\_graph module

Graph utility functions.

`osmnx.utils_graph.add_edge_lengths` (*G, precision=3*)

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

Calculated via great-circle distance between each edge's incident nodes, so ensure graph is in unprojected coordinates.

**Parameters**

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

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

**Return type** networkx.MultiDiGraph

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

Count how many street segments emanate from each node in this graph.

If *nodes* is passed, then only count the nodes in the graph with those IDs.



**Parameters**

- **G** (*networkx.MultiDiGraph*) – input graph
- **nodes** (*iterable*) – the set of node IDs to get counts for

**Returns** **streets\_per\_node** – counts of how many streets emanate from each node with keys=node id and values=count

**Return type** dict

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

Convert MultiDiGraph to DiGraph.

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

**Parameters**

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

**Returns**

**Return type** `networkx.DiGraph`

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

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

**Parameters**

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

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

**Return type** `networkx.MultiDiGraph`

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

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

**Parameters**

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

**Returns** **attribute\_values** – list of edge attribute values

**Return type** list

`osmnx.utils_graph.get_undirected(G)`

Convert MultiDiGraph to MultiGraph.

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

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

**Returns**

**Return type** *networkx.MultiGraph*

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

Convert node and edge GeoDataFrames to a MultiDiGraph.

This function is the inverse of *graph\_to\_gdfs*.

**Parameters**

- **gdf\_nodes** (*geopandas.GeoDataFrame*) – GeoDataFrame of graph nodes
- **gdf\_edges** (*geopandas.GeoDataFrame*) – GeoDataFrame of graph edges, must have crs attribute set
- **graph\_attrs** (*dict*) – the new *G*.graph attribute dict; if None, add crs as the only graph-level attribute

**Returns** *G*

**Return type** *networkx.MultiDiGraph*

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

Convert a graph to node and/or edge GeoDataFrames.

This function is the inverse of *graph\_from\_gdfs*.

**Parameters**

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

**Returns** *gdf\_nodes* or *gdf\_edges* or tuple of (*gdf\_nodes*, *gdf\_edges*)

**Return type** *geopandas.GeoDataFrame* or tuple

`osmnx.utils_graph.induce_subgraph(G, node_subset)`

Induce a subgraph of *G*: deprecated, do not use.

**Parameters**

- **G** (*networkx.MultiDiGraph*) – input graph
- **node\_subset** (*list-like*) – the subset of nodes to induce a subgraph of *G*

**Returns** the subgraph of *G* induced by *node\_subset*

**Return type** *networkx.MultiDiGraph*

`osmnx.utils_graph.remove_isolated_nodes(G)`

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

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

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

**Return type** *networkx.MultiDiGraph*

**OSMnx:** retrieve, model, analyze, and visualize OpenStreetMap street networks and other spatial data.

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

The OSMnx project is hosted on [GitHub](#).



## CITATION INFO

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

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



## FEATURES

OSMnx is built on top of `geopandas`, `networkx`, and `matplotlib` and interacts with OpenStreetMap's APIs to:

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

Usage examples and demonstrations of these features are in the [examples](#) GitHub repo. More feature development details are in the [change log](#).





## INSTALLATION

You can install OSMnx with conda:

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

Alternatively, you can run OSMnx + Jupyter directly from the official [docker container](#), or you can install OSMnx via [pip](#) if you already have all of its dependencies installed and tested on your system. Note: pip installation is nontrivial. If you don't know *exactly* what you're doing, just use conda as described above.

To upgrade OSMnx to a newer release, just remove the conda environment you created and then create a new one again following the steps above. Don't just run `conda update` or you could get package conflicts.



## USAGE EXAMPLES

For sample code and usage examples/demos, see the [examples](#) GitHub repo.



## SUPPORT

If you've discovered a bug in OSMnx, please open an [issue](#) at the [OSMnx GitHub repo](#) documenting what is broken in the package. Alternatively, if you have a usage question, please ask it on [StackOverflow](#).



## LICENSE

The project is licensed under the MIT license.





## INDICES AND TABLES

- [genindex](#)
- [modindex](#)
- [search](#)
- [Internals Reference](#)



## PYTHON MODULE INDEX

### O

- `osmnx.bearing`, 1
- `osmnx.distance`, 2
- `osmnx.downloader`, 5
- `osmnx.elevation`, 6
- `osmnx.folium`, 6
- `osmnx.footprints`, 8
- `osmnx.geocoder`, 10
- `osmnx.geometries`, 10
- `osmnx.graph`, 13
- `osmnx.io`, 17
- `osmnx.plot`, 19
- `osmnx.pois`, 23
- `osmnx.projection`, 25
- `osmnx.settings`, 26
- `osmnx.simplification`, 26
- `osmnx.speed`, 28
- `osmnx.stats`, 29
- `osmnx.truncate`, 31
- `osmnx.utils`, 33
- `osmnx.utils_geo`, 35
- `osmnx.utils_graph`, 36



## A

`add_edge_bearings()` (in module *osmnx.bearing*), 1  
`add_edge_grades()` (in module *osmnx.elevation*), 6  
`add_edge_lengths()` (in module *osmnx.utils\_graph*), 36  
`add_edge_speeds()` (in module *osmnx.speed*), 28  
`add_edge_travel_times()` (in module *osmnx.speed*), 28  
`add_node_elevations()` (in module *osmnx.elevation*), 6

## B

`basic_stats()` (in module *osmnx.stats*), 29  
`bbox_from_point()` (in module *osmnx.utils\_geo*), 35  
`bbox_to_poly()` (in module *osmnx.utils\_geo*), 35

## C

`citation()` (in module *osmnx.utils*), 33  
`config()` (in module *osmnx.utils*), 33  
`consolidate_intersections()` (in module *osmnx.simplification*), 26  
`count_streets_per_node()` (in module *osmnx.utils\_graph*), 36

## E

`euclidean_dist_vec()` (in module *osmnx.distance*), 2  
`extended_stats()` (in module *osmnx.stats*), 30

## F

`footprints_from_address()` (in module *osmnx.footprints*), 8  
`footprints_from_place()` (in module *osmnx.footprints*), 8  
`footprints_from_point()` (in module *osmnx.footprints*), 9  
`footprints_from_polygon()` (in module *osmnx.footprints*), 9

## G

`geocode()` (in module *osmnx.geocoder*), 10  
`geocode_to_gdf()` (in module *osmnx.geocoder*), 10  
`geometries_from_address()` (in module *osmnx.geometries*), 10  
`geometries_from_bbox()` (in module *osmnx.geometries*), 11  
`geometries_from_place()` (in module *osmnx.geometries*), 11  
`geometries_from_point()` (in module *osmnx.geometries*), 12  
`geometries_from_polygon()` (in module *osmnx.geometries*), 12  
`geometries_from_xml()` (in module *osmnx.geometries*), 13  
`get_bearing()` (in module *osmnx.bearing*), 1  
`get_colors()` (in module *osmnx.plot*), 19  
`get_digraph()` (in module *osmnx.utils\_graph*), 37  
`get_edge_colors_by_attr()` (in module *osmnx.plot*), 20  
`get_largest_component()` (in module *osmnx.utils\_graph*), 37  
`get_nearest_edge()` (in module *osmnx.distance*), 2  
`get_nearest_edges()` (in module *osmnx.distance*), 2  
`get_nearest_node()` (in module *osmnx.distance*), 3  
`get_nearest_nodes()` (in module *osmnx.distance*), 3  
`get_node_colors_by_attr()` (in module *osmnx.plot*), 20  
`get_route_edge_attributes()` (in module *osmnx.utils\_graph*), 37  
`get_undirected()` (in module *osmnx.utils\_graph*), 37  
`graph_from_address()` (in module *osmnx.graph*), 13  
`graph_from_bbox()` (in module *osmnx.graph*), 14  
`graph_from_gdfs()` (in module *osmnx.utils\_graph*), 38  
`graph_from_place()` (in module *osmnx.graph*), 15  
`graph_from_point()` (in module *osmnx.graph*), 15  
`graph_from_polygon()` (in module *osmnx.graph*),

16  
graph\_from\_xml() (in module *osmnx.graph*), 17  
graph\_to\_gdfs() (in module *osmnx.utils\_graph*), 38  
great\_circle\_vec() (in module *osmnx.distance*), 4

## I

induce\_subgraph() (in module *osmnx.utils\_graph*), 38

## K

k\_shortest\_paths() (in module *osmnx.distance*), 4

## L

load\_graphml() (in module *osmnx.io*), 17  
log() (in module *osmnx.utils*), 34

## M

module  
    osmnx.bearing, 1  
    osmnx.distance, 2  
    osmnx.downloader, 5  
    osmnx.elevation, 6  
    osmnx.folium, 6  
    osmnx.footprints, 8  
    osmnx.geocoder, 10  
    osmnx.geometries, 10  
    osmnx.graph, 13  
    osmnx.io, 17  
    osmnx.plot, 19  
    osmnx.pois, 23  
    osmnx.projection, 25  
    osmnx.settings, 26  
    osmnx.simplification, 26  
    osmnx.speed, 28  
    osmnx.stats, 29  
    osmnx.truncate, 31  
    osmnx.utils, 33  
    osmnx.utils\_geo, 35  
    osmnx.utils\_graph, 36

## N

nominatim\_request() (in module *osmnx.downloader*), 5

## O

osmnx.bearing  
    module, 1  
osmnx.distance  
    module, 2  
osmnx.downloader  
    module, 5  
osmnx.elevation  
    module, 6

osmnx.folium  
    module, 6  
osmnx.footprints  
    module, 8  
osmnx.geocoder  
    module, 10  
osmnx.geometries  
    module, 10  
osmnx.graph  
    module, 13  
osmnx.io  
    module, 17  
osmnx.plot  
    module, 19  
osmnx.pois  
    module, 23  
osmnx.projection  
    module, 25  
osmnx.settings  
    module, 26  
osmnx.simplification  
    module, 26  
osmnx.speed  
    module, 28  
osmnx.stats  
    module, 29  
osmnx.truncate  
    module, 31  
osmnx.utils  
    module, 33  
osmnx.utils\_geo  
    module, 35  
osmnx.utils\_graph  
    module, 36  
overpass\_request() (in module *osmnx.downloader*), 5

## P

plot\_figure\_ground() (in module *osmnx.plot*), 20  
plot\_footprints() (in module *osmnx.plot*), 21  
plot\_graph() (in module *osmnx.plot*), 21  
plot\_graph\_folium() (in module *osmnx.folium*), 6  
plot\_graph\_route() (in module *osmnx.plot*), 22  
plot\_graph\_routes() (in module *osmnx.plot*), 23  
plot\_route\_folium() (in module *osmnx.folium*), 7  
pois\_from\_address() (in module *osmnx.pois*), 23  
pois\_from\_place() (in module *osmnx.pois*), 24  
pois\_from\_point() (in module *osmnx.pois*), 24  
pois\_from\_polygon() (in module *osmnx.pois*), 25  
project\_gdf() (in module *osmnx.projection*), 25  
project\_geometry() (in module *osmnx.projection*), 25  
project\_graph() (in module *osmnx.projection*), 26

## R

`redistribute_vertices()` (in module *osmnx.utils\_geo*), 36  
`remove_isolated_nodes()` (in module *osmnx.utils\_graph*), 38  
`round_geometry_coords()` (in module *osmnx.utils\_geo*), 36

## S

`save_graph_geopackage()` (in module *osmnx.io*), 17  
`save_graph_shapefile()` (in module *osmnx.io*), 18  
`save_graph_xml()` (in module *osmnx.io*), 18  
`save_graphml()` (in module *osmnx.io*), 19  
`shortest_path()` (in module *osmnx.distance*), 4  
`simplify_graph()` (in module *osmnx.simplification*), 27

## T

`truncate_graph_bbox()` (in module *osmnx.truncate*), 31  
`truncate_graph_dist()` (in module *osmnx.truncate*), 31  
`truncate_graph_polygon()` (in module *osmnx.truncate*), 32  
`ts()` (in module *osmnx.utils*), 35