
OSMnx Documentation

Release 0.12.1

Geoff Boeing

May 05, 2020

CONTENTS:

1	osmnx package	1
1.1	Submodules	1
1.2	osmnx.core module	1
1.3	osmnx.downloader module	11
1.4	osmnx.elevation module	13
1.5	osmnx.errors module	13
1.6	osmnx.footprints module	14
1.7	osmnx.geo_utils module	18
1.8	osmnx.osm_content_handler module	23
1.9	osmnx.plot module	24
1.10	osmnx.pois module	31
1.11	osmnx.projection module	34
1.12	osmnx.save_load module	35
1.13	osmnx.settings module	39
1.14	osmnx.simplify module	39
1.15	osmnx.stats module	40
1.16	osmnx.utils module	43
1.17	Module contents	46
2	Citation info	47
3	Features	49
4	Installation	51
5	Examples	53
6	Support	55
7	License	57
8	Indices and tables	59
	Python Module Index	61
	Index	63

OSMNX PACKAGE

1.1 Submodules

1.2 osmnx.core module

`osmnx.core.add_edge_lengths(G)`

Add length (meters) attribute to each edge by great circle distance between nodes u and v.

Parameters *G* (*networkx multidigraph*) –

Returns *G*

Return type *networkx multidigraph*

`osmnx.core.add_path(G, data, one_way)`

Add a path to the graph.

Parameters

- *G* (*networkx multidigraph*) –
- **data** (*dict*) – the attributes of the path
- **one_way** (*bool*) – if this path is one-way or if it is bi-directional

Returns

Return type *None*

`osmnx.core.add_paths(G, paths, bidirectional=False)`

Add a collection of paths to the graph.

Parameters

- *G* (*networkx multidigraph*) –
- **paths** (*dict*) – the paths from OSM
- **bidirectional** (*bool*) – if True, create bidirectional edges for one-way streets

Returns

Return type *None*

`osmnx.core.bbox_from_point(point, distance=1000, project_utm=False, return_crs=False)`

Create a bounding box some distance in each direction (north, south, east, and west) from some (lat, lng) point.

Parameters

- **point** (*tuple*) – the (lat, lon) point to create the bounding box around

- **distance** (*int*) – how many meters the north, south, east, and west sides of the box should each be from the point
- **project_utm** (*bool*) – if True return bbox as UTM coordinates
- **return_crs** (*bool*) – if True and project_utm=True, return the projected CRS

Returns

- **north, south, east, west** (*tuple*, if *return_crs=False*)
- **north, south, east, west, crs_proj** (*tuple*, if *return_crs=True*)

`osmnx.core.consolidate_subdivide_geometry` (*geometry*, *max_query_area_size*)

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

Parameters

- **geometry** (*shapely Polygon or MultiPolygon*) – the geometry to consolidate and subdivide
- **max_query_area_size** (*float*) – max area for any part of the geometry, in the units the geometry is in. any polygon bigger will get divided up for multiple queries to API (default is 50,000 * 50,000 units (ie, 50km x 50km in area, if units are meters))

Returns geometry

Return type Polygon or MultiPolygon

`osmnx.core.create_graph` (*response_jsons*, *name='unnamed'*, *retain_all=False*, *bidirectional=False*)

Create a networkx graph from Overpass API HTTP response objects.

Parameters

- **response_jsons** (*list*) – list of dicts of JSON responses from from the Overpass API
- **name** (*string*) – the name of the graph
- **retain_all** (*bool*) – if True, return the entire graph even if it is not connected
- **bidirectional** (*bool*) – if True, create bidirectional edges for one-way streets

Returns

Return type networkx multidigraph

`osmnx.core.gdf_from_place` (*query*, *gdf_name=None*, *which_result=1*, *buffer_dist=None*)

Create a GeoDataFrame from a single place name query.

Parameters

- **query** (*string or dict*) – query string or structured query dict to geocode/download
- **gdf_name** (*string*) – name attribute metadata for GeoDataFrame (this is used to save shapefile later)
- **which_result** (*int*) – max number of results to return and which to process upon receipt
- **buffer_dist** (*float*) – distance to buffer around the place geometry, in meters

Returns

Return type GeoDataFrame

`osmnx.core.gdf_from_places` (*queries*, *gdf_name*='unnamed', *buffer_dist*=None, *which_results*=None)

Create a GeoDataFrame from a list of place names to query.

Parameters

- **queries** (*list*) – list of query strings or structured query dicts to geocode/download, one at a time
- **gdf_name** (*string*) – name attribute metadata for GeoDataFrame (this is used to save shapefile later)
- **buffer_dist** (*float*) – distance to buffer around the place geometry, in meters
- **which_results** (*list*) – if not None, a list of max number of results to return and which to process upon receipt, for each query in queries

Returns

Return type GeoDataFrame

`osmnx.core.get_node` (*element*)

Convert an OSM node element into the format for a networkx node.

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

Returns

Return type dict

`osmnx.core.get_path` (*element*)

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

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

Returns

Return type dict

`osmnx.core.get_polygons_coordinates` (*geometry*)

Extract exterior coordinates from polygon(s) to pass to OSM in a query by polygon. Ignore the interior (“holes”) coordinates.

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

Returns *polygon_coord_strs*

Return type list

`osmnx.core.graph_from_address` (*address*, *distance*=1000, *distance_type*='bbox', *network_type*='all_private', *simplify*=True, *retain_all*=False, *truncate_by_edge*=False, *return_coords*=False, *name*='unnamed', *timeout*=180, *memory*=None, *max_query_area_size*=2500000000, *clean_periphery*=True, *infrastructure*='way["highway"]', *custom_filter*=None, *custom_settings*=None)

Create a networkx graph from OSM data 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
- **distance** (*int*) – retain only those nodes within this many meters of the center of the graph

- **distance_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
- **simplify** (*bool*) – if true, simplify the graph topology
- **retain_all** (*bool*) – if True, return the entire graph even if it is not connected
- **truncate_by_edge** (*bool*) – if True retain node if it's outside bbox but at least one of node's neighbors are within bbox
- **return_coords** (*bool*) – optionally also return the geocoded coordinates of the address
- **name** (*string*) – the name of the graph
- **timeout** (*int*) – the timeout interval for requests and to pass to API
- **memory** (*int*) – server memory allocation size for the query, in bytes. If none, server will use its default allocation size
- **max_query_area_size** – float, max size for any part of the geometry, in square degrees: any polygon bigger will get divided up for multiple queries to API
- **clean_periphery** (*bool*,) – if True (and simplify=True), buffer 0.5km to get a graph larger than requested, then simplify, then truncate it to requested spatial extent
- **infrastructure** (*string*) – download infrastructure of given type (default is streets (ie, 'way["highway"]') but other infrastructures may be selected like power grids (ie, 'way["power"~"line"]'))
- **custom_filter** (*string*) – a custom network filter to be used instead of the network_type presets
- **custom_settings** (*string*) – custom settings to be used in the overpass query instead of the default ones

Returns multidigraph or optionally (multidigraph, tuple)

Return type networkx multidigraph or tuple

```
osmnx.core.graph_from_bbox(north, south, east, west, network_type='all_private', simplify=True,
                           retain_all=False, truncate_by_edge=False, name='unnamed', time-
                           out=180, memory=None, max_query_area_size=2500000000,
                           clean_periphery=True, infrastructure='way["highway"]', cus-
                           tom_filter=None, custom_settings=None)
```

Create a networkx graph from OSM data 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
- **simplify** (*bool*) – if true, simplify the graph topology
- **retain_all** (*bool*) – if True, return the entire graph even if it is not connected
- **truncate_by_edge** (*bool*) – if True retain node if it's outside bbox but at least one of node's neighbors are within bbox

- **name** (*string*) – the name of the graph
- **timeout** (*int*) – the timeout interval for requests and to pass to API
- **memory** (*int*) – server memory allocation size for the query, in bytes. If none, server will use its default allocation size
- **max_query_area_size** (*float*) – max size for any part of the geometry, in square degrees: any polygon bigger will get divided up for multiple queries to API
- **clean_periphery** (*bool*) – if True (and simplify=True), buffer 0.5km to get a graph larger than requested, then simplify, then truncate it to requested spatial extent
- **infrastructure** (*string*) – download infrastructure of given type (default is streets (ie, 'way["highway"]')) but other infrastructures may be selected like power grids (ie, 'way["power"]~"line"]'))
- **custom_filter** (*string*) – a custom network filter to be used instead of the network_type presets
- **custom_settings** (*string*) – custom settings to be used in the overpass query instead of the default ones

Returns

Return type networkx multidigraph

```
osmnx.core.graph_from_file(filename, bidirectional=False, simplify=True, retain_all=False,
                           name='unnamed')
```

Create a networkx graph from OSM data in an XML file.

Parameters

- **filename** (*string*) – the name of a file containing OSM XML data
- **bidirectional** (*bool*) – if True, create bidirectional edges for one-way streets
- **simplify** (*bool*) – if True, simplify the graph topology
- **retain_all** (*bool*) – if True, return the entire graph even if it is not connected
- **name** (*string*) – the name of the graph

Returns

Return type networkx multidigraph

```
osmnx.core.graph_from_place(query, network_type='all_private', simplify=True, retain_all=False,
                           truncate_by_edge=False, name='unnamed',
                           which_result=1, buffer_dist=None, timeout=180, memory=None,
                           max_query_area_size=2500000000, clean_periphery=True,
                           infrastructure='way["highway"]', custom_filter=None,
                           custom_settings=None)
```

Create a networkx graph from OSM data within the spatial 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`.

Parameters

- **query** (*string or dict or list*) – the place(s) to geocode/download data for

- **network_type** (*string*) – what type of street network to get
- **simplify** (*bool*) – if true, simplify the graph topology
- **retain_all** (*bool*) – if True, return the entire graph even if it is not connected
- **truncate_by_edge** (*bool*) – if True retain node if it's outside bbox but at least one of node's neighbors are within bbox
- **name** (*string*) – the name of the graph
- **which_result** (*int*) – max number of results to return and which to process upon receipt
- **buffer_dist** (*float*) – distance to buffer around the place geometry, in meters
- **timeout** (*int*) – the timeout interval for requests and to pass to API
- **memory** (*int*) – server memory allocation size for the query, in bytes. If none, server will use its default allocation size
- **max_query_area_size** (*float*) – max size for any part of the geometry, in square degrees: any polygon bigger will get divided up for multiple queries to API
- **clean_periphery** (*bool*) – if True (and simplify=True), buffer 0.5km to get a graph larger than requested, then simplify, then truncate it to requested spatial extent
- **infrastructure** (*string*) – download infrastructure of given type (default is streets (ie, 'way["highway"]') but other infrastructures may be selected like power grids (ie, 'way["power"~"line"]'))
- **custom_filter** (*string*) – a custom network filter to be used instead of the network_type presets
- **custom_settings** (*string*) – custom settings to be used in the overpass query instead of the default ones

Returns

Return type networkx multidigraph

```
osmnx.core.graph_from_point(center_point, distance=1000, distance_type='bbox', network_type='all_private', simplify=True, retain_all=False, truncate_by_edge=False, name='unnamed', timeout=180, memory=None, max_query_area_size=2500000000, clean_periphery=True, infrastructure='way["highway"]', custom_filter=None, custom_settings=None)
```

Create a networkx graph from OSM data within some distance of some (lat, lon) center point.

Parameters

- **center_point** (*tuple*) – the (lat, lon) central point around which to construct the graph
- **distance** (*int*) – retain only those nodes within this many meters of the center of the graph, with distance determined according to distance_type argument
- **distance_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
- **simplify** (*bool*) – if true, simplify the graph topology
- **retain_all** (*bool*) – if True, return the entire graph even if it is not connected

- **truncate_by_edge** (*bool*) – if True retain node if it’s outside bbox but at least one of node’s neighbors are within bbox
- **name** (*string*) – the name of the graph
- **timeout** (*int*) – the timeout interval for requests and to pass to API
- **memory** (*int*) – server memory allocation size for the query, in bytes. If none, server will use its default allocation size
- **max_query_area_size** (*float*) – max size for any part of the geometry, in square degrees: any polygon bigger will get divided up for multiple queries to API
- **clean_periphery** (*bool*,) – if True (and simplify=True), buffer 0.5km to get a graph larger than requested, then simplify, then truncate it to requested spatial extent
- **infrastructure** (*string*) – download infrastructure of given type (default is streets (ie, ‘way[“highway”]’) but other infrastructures may be selected like power grids (ie, ‘way[“power”~“line”]’))
- **custom_filter** (*string*) – a custom network filter to be used instead of the network_type presets
- **custom_settings** (*string*) – custom settings to be used in the overpass query instead of the default ones

Returns

Return type networkx multidigraph

```
osmnx.core.graph_from_polygon(polygon, network_type='all_private', simplify=True, retain_all=False, truncate_by_edge=False, name='unnamed', timeout=180, memory=None, max_query_area_size=2500000000, clean_periphery=True, infrastructure='way[“highway”]', custom_filter=None, custom_settings=None)
```

Create a networkx graph from OSM data within the spatial boundaries of the passed-in shapely polygon.

Parameters

- **polygon** (*shapely Polygon or 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
- **simplify** (*bool*) – if true, simplify the graph topology
- **retain_all** (*bool*) – if True, return the entire graph even if it is not connected
- **truncate_by_edge** (*bool*) – if True retain node if it’s outside bbox but at least one of node’s neighbors are within bbox
- **name** (*string*) – the name of the graph
- **timeout** (*int*) – the timeout interval for requests and to pass to API
- **memory** (*int*) – server memory allocation size for the query, in bytes. If none, server will use its default allocation size
- **max_query_area_size** (*float*) – max size for any part of the geometry, in square degrees: any polygon bigger will get divided up for multiple queries to API
- **clean_periphery** (*bool*) – if True (and simplify=True), buffer 0.5km to get a graph larger than requested, then simplify, then truncate it to requested spatial extent

- **infrastructure** (*string*) – download infrastructure of given type (default is streets (ie, 'way["highway"]') but other infrastructures may be selected like power grids (ie, 'way["power"~"line"]'))
- **custom_filter** (*string*) – a custom network filter to be used instead of the network_type presets
- **custom_settings** (*string*) – custom settings to be used in the overpass query instead of the default ones

Returns

Return type networkx multidigraph

```
osmnx.core.intersect_index_quadrats(gdf, geometry, quadrat_width=0.05, min_num=3,  
                                   buffer_amount=1e-09)
```

Intersect points with a polygon, using an r-tree spatial index and cutting the polygon up into smaller sub-polygons for r-tree acceleration.

Parameters

- **gdf** (*GeoDataFrame*) – the set of points to intersect
- **geometry** (*shapely Polygon or MultiPolygon*) – the geometry to intersect with the points
- **quadrat_width** (*numeric*) – 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*) – the minimum number of linear quadrat lines (e.g., min_num=3 would produce a quadrat grid of 4 squares)
- **buffer_amount** (*numeric*) – buffer the quadrat grid lines by quadrat_width times buffer_amount

Returns

Return type GeoDataFrame

```
osmnx.core.osm_net_download(polygon=None, north=None, south=None, east=None,  
                           west=None, network_type='all_private', timeout=180,  
                           memory=None, max_query_area_size=2500000000, in-  
                           frastructure='way["highway"]', custom_filter=None, cus-  
                           tom_settings=None)
```

Download OSM ways and nodes within some bounding box from the Overpass API.

Parameters

- **polygon** (*shapely Polygon or MultiPolygon*) – geographic shape to fetch the street network within
- **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*) – { 'walk', 'bike', 'drive', 'drive_service', 'all', 'all_private' } what type of street network to get
- **timeout** (*int*) – the timeout interval for requests and to pass to API
- **memory** (*int*) – server memory allocation size for the query, in bytes. If none, server will use its default allocation size

- **max_query_area_size** (*float*) – max area for any part of the geometry, in the units the geometry is in: any polygon bigger will get divided up for multiple queries to API (default is 50,000 * 50,000 units [ie, 50km x 50km in area, if units are meters])
- **infrastructure** (*string*) – download infrastructure of given type. default is streets, ie, 'way["highway"]' but other infrastructures may be selected like power grids, ie, 'way["power"~"line"]'
- **custom_filter** (*string*) – a custom network filter to be used instead of the network_type presets
- **custom_settings** (*string*) – custom settings to be used in the overpass query instead of the default ones

Returns response_jsons

Return type list

`osmnx.core.parse_osm_nodes_paths(osm_data)`

Construct dicts of nodes and paths with key=osmid and value=dict of attributes.

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

Returns nodes, paths

Return type tuple

`osmnx.core.quadrat_cut_geometry(geometry, quadrat_width, min_num=3, buffer_amount=1e-09)`

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

Parameters

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

Returns

Return type shapely MultiPolygon

`osmnx.core.remove_isolated_nodes(G)`

Remove from a graph all the nodes that have no incident edges (ie, node degree = 0).

Parameters `G` (*networkx multidigraph*) – the graph from which to remove nodes

Returns

Return type networkx multidigraph

`osmnx.core.truncate_graph_bbox(G, north, south, east, west, truncate_by_edge=False, retain_all=False)`

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

Needed because overpass returns entire ways that also include nodes outside the bbox if the way (that is, a way with a single OSM ID) has a node inside the bbox at some point.

Parameters

- `G` (*networkx multidigraph*) –

- **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 node if it's outside bbox but at least one of node's neighbors are within bbox
- **retain_all** (*bool*) – if True, return the entire graph even if it is not connected

Returns

Return type `networkx multidigraph`

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

Remove everything further than some network distance from a specified node in graph.

Parameters

- **G** (*networkx multidigraph*) –
- **source_node** (*int*) – the node in the graph from which to measure network distances to other nodes
- **max_distance** (*int*) – remove every node in the graph greater than this distance from the source_node
- **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

Returns

Return type `networkx multidigraph`

```
osmnx.core.truncate_graph_polygon(G, polygon, retain_all=False, truncate_by_edge=False, quadrat_width=0.05, min_num=3, buffer_amount=1e-09)
```

Remove every node in graph that falls outside some shapely Polygon or MultiPolygon.

Parameters

- **G** (*networkx multidigraph*) –
- **polygon** (*Polygon or 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
- **truncate_by_edge** (*bool*) – if True retain node if it's outside polygon but at least one of node's neighbors are within 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)
- **buffer_amount** (*numeric*) – passed on to `intersect_index_quadrats`: buffer the quadrat grid lines by `quadrat_width` times `buffer_amount`

Returns

Return type networkx multidigraph

1.3 osmnx.downloader module

`osmnx.downloader.get_from_cache(url)`

Retrieve a HTTP response json object from the cache.

Parameters `url` (*string*) – the url of the request

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

Return type dict

`osmnx.downloader.get_http_headers(user_agent=None, referer=None, accept_language=None)`

Update the default requests HTTP headers with OSMnx info.

Parameters

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

Returns headers

Return type dict

`osmnx.downloader.get_osm_filter(network_type)`

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

Parameters `network_type` (*string*) – {'walk', 'bike', 'drive', 'drive_service', 'all', 'all_private', 'none'} what type of street or other network to get

Returns

Return type string

`osmnx.downloader.get_pause_duration(recursive_delay=5, default_duration=10)`

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

Parameters

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

Returns

Return type int

`osmnx.downloader.nominatim_request(params, type='search', pause_duration=1, timeout=30, error_pause_duration=180)`

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

Parameters

- **params** (*dict or OrderedDict*) – key-value pairs of parameters
- **type** (*string*) – Type of Nominatim query. One of the following: search, reverse or lookup
- **pause_duration** (*int*) – how long to pause before requests, in seconds

- **timeout** (*int*) – the timeout interval for the requests library
- **error_pause_duration** (*int*) – how long to pause in seconds before re-trying requests if error

Returns `response_json`

Return type `dict`

`osmnx.downloader.osm_polygon_download(query, limit=1, polygon_geojson=1)`

Geocode a place and download its boundary geometry from OSM's Nominatim API.

Parameters

- **query** (*string or dict*) – query string or structured query dict to geocode/download
- **limit** (*int*) – max number of results to return
- **polygon_geojson** (*int*) – request the boundary geometry polygon from the API, 0=no, 1=yes

Returns

Return type `dict`

`osmnx.downloader.overpass_request(data, pause_duration=None, timeout=180, error_pause_duration=None)`

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

Parameters

- **data** (*dict or OrderedDict*) – key-value pairs of parameters to post to the API
- **pause_duration** (*int*) – how long to pause in seconds before requests, if None, will query API status endpoint to find when next slot is available
- **timeout** (*int*) – the timeout interval for the requests library
- **error_pause_duration** (*int*) – how long to pause in seconds before re-trying requests if error

Returns

Return type `dict`

`osmnx.downloader.save_to_cache(url, response_json)`

Save an HTTP response json object to the cache.

If the request was sent to server via POST instead of GET, then URL should be a GET-style representation of request. Users should always pass OrderedDicts instead of dicts of parameters into request functions, so that the parameters stay in the same order each time, producing the same URL string, and thus the same hash. Otherwise the cache will eventually contain multiple saved responses for the same request because the URL's parameters appeared in a different order each time.

Parameters

- **url** (*string*) – the url of the request
- **response_json** (*dict*) – the json response

Returns

Return type `None`

`osmnx.downloader.url_in_cache(url)`

Determine if a URL's response exists in the cache.

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

Returns `filepath` – path to cached response for url if it exists in the cache, otherwise None

Return type `string`

1.4 osmnx.elevation module

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

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

Parameters

- **G** (*networkx multidigraph*) –
- **add_absolute** (*bool*) – if True, also add the absolute value of the grade as an edge attribute

Returns

Return type `networkx multidigraph`

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

Get the elevation (meters) of each node in the network and add it to the node as an attribute.

Parameters

- **G** (*networkx multidigraph*) –
- **api_key** (*string*) – your google maps elevation API key
- **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

Returns

Return type `networkx multidigraph`

1.5 osmnx.errors module

exception `osmnx.errors.EmptyOverpassResponse` (**args*, ***kwargs*)

Bases: `ValueError`

exception `osmnx.errors.InsufficientNetworkQueryArguments` (**args*, ***kwargs*)

Bases: `ValueError`

exception `osmnx.errors.InvalidDistanceType` (**args*, ***kwargs*)

Bases: `ValueError`

exception `osmnx.errors.UnknownNetworkType` (**args*, ***kwargs*)

Bases: `ValueError`

1.6 osmnx.footprints module

`osmnx.footprints.create_footprint_geometry` (*footprint_key, footprint_val, vertices*)

Create Shapely geometry for open or closed ways in the initial footprints dictionary.

Closed ways are converted directly to Shapely Polygons, open ways (fragments that will form the outer and inner rings of relations) are converted to LineStrings.

Parameters

- **footprint_key** (*int*) – the id of the way/footprint to process
- **footprint_val** (*dict*) – the nodes and tags of the footprint
- **vertices** (*dict*) – the dictionary of OSM nodes with their coordinates

Returns

Return type Shapely Polygon or LineString

`osmnx.footprints.create_footprints_gdf` (*polygon=None, north=None, south=None, east=None, west=None, footprint_type='building', retain_invalid=False, responses=None, custom_settings=None*)

Get footprint (polygon) data from OSM and convert it into a GeoDataFrame.

Parameters

- **polygon** (*shapely Polygon or MultiPolygon*) – geographic shape to fetch the footprints within
- **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
- **footprint_type** (*string*) – type of footprint to be downloaded. OSM tag key e.g. 'building', 'landuse', 'place', etc.
- **retain_invalid** (*bool*) – if False discard any footprints with an invalid geometry
- **responses** (*list*) – list of response jsons
- **custom_settings** (*string*) – custom settings to be used in the overpass query instead of the default ones

Returns

Return type GeoDataFrame

`osmnx.footprints.create_relation_geometry` (*relation_key, relation_val, footprints*)

Create Shapely geometry for relations - Polygons with holes or MultiPolygons

OSM relations are used to define complex polygons - polygons with holes or multi-polygons. The polygons' outer and inner rings may be made up of chains of LineStrings. <https://wiki.openstreetmap.org/wiki/Relation:multipolygon> requires that multipolygon rings have an outer or inner 'role'.

OSM's data model allows a polygon type tag e.g. 'building' to be added to any OSM element. This can include non-polygon relations e.g. bus routes. Relations that do not have at least one closed ring with an outer role are filtered out.

Inner rings that are tagged with the footprint type in their own right e.g. `landuse=meadow` as an inner ring of `landuse=forest` will have been included in the footprints dictionary as part of the original parsing and are not dealt with here.

Parameters

- **relation_key** (*int*) – the id of the relation to process
- **relation_val** (*dict*) – members and tags of the relation
- **footprints** (*dictionary*) – dictionary of all footprints (including open and closed ways)

Returns

Return type Shapely Polygon or MultiPolygon

`osmnx.footprints.footprints_from_address` (*address*, *distance*, *footprint_type*='building', *retain_invalid*=False, *custom_settings*=None)

Get footprints within some distance north, south, east, and west of an address.

Parameters

- **address** (*string*) – the address to geocode to a lat-long point
- **distance** (*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*) – if False discard any footprints with an invalid geometry
- **custom_settings** (*string*) – custom settings to be used in the overpass query instead of the default ones

Returns

Return type GeoDataFrame

`osmnx.footprints.footprints_from_place` (*place*, *footprint_type*='building', *retain_invalid*=False, *which_result*=1, *custom_settings*=None)

Get footprints within the boundaries of some place.

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 geojson boundary polygon
- **footprint_type** (*string*) – type of footprint to be downloaded. OSM tag key e.g. 'building', 'landuse', 'place', etc.
- **retain_invalid** (*bool*) – if False discard any footprints with an invalid geometry
- **which_result** (*int*) – max number of results to return and which to process upon receipt
- **custom_settings** (*string*) – custom settings to be used in the overpass query instead of the default ones

Returns

Return type GeoDataFrame

```
osmnx.footprints.footprints_from_point(point, distance, footprint_type='building', retain_invalid=False, custom_settings=None)
```

Get footprints within some distance north, south, east, and west of a lat-long point.

Parameters

- **point** (*tuple*) – a lat-long point
- **distance** (*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*) – if False discard any footprints with an invalid geometry
- **custom_settings** (*string*) – custom settings to be used in the overpass query instead of the default ones

Returns

Return type GeoDataFrame

```
osmnx.footprints.footprints_from_polygon(polygon, footprint_type='building', retain_invalid=False, custom_settings=None)
```

Get footprints within some polygon.

Parameters

- **polygon** (*shapely Polygon or 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*) – if False discard any footprints with an invalid geometry
- **custom_settings** (*string*) – custom settings to be used in the overpass query instead of the default ones

Returns

Return type GeoDataFrame

```
osmnx.footprints.osm_footprints_download(polygon=None, north=None, south=None, east=None, west=None, footprint_type='building', timeout=180, memory=None, max_query_area_size=2500000000, custom_settings=None)
```

Download OpenStreetMap footprint data as a list of json responses.

Parameters

- **polygon** (*shapely Polygon or MultiPolygon*) – geographic shape to fetch the footprints within
- **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
- **footprint_type** (*string*) – type of footprint to be downloaded. OSM tag key e.g. 'building', 'landuse', 'place', etc.
- **timeout** (*int*) – the timeout interval for requests and to pass to API

- **memory** (*int*) – server memory allocation size for the query, in bytes. If none, server will use its default allocation size
- **max_query_area_size** (*float*) – max area for any part of the geometry, in the units the geometry is in: any polygon bigger will get divided up for multiple queries to API (default is 50,000 * 50,000 units (ie, 50km x 50km in area, if units are meters))
- **custom_settings** (*string*) – custom settings to be used in the overpass query instead of the default ones

Returns list of response_json dicts

Return type list

```
osmnx.footprints.plot_footprints(gdf, fig=None, ax=None, figsize=None, color='#333333',
                                bgcolor='w', set_bounds=True, bbox=None,
                                save=False, show=True, close=False, filename='image',
                                file_format='png', dpi=600)
```

Plot a GeoDataFrame of footprints.

Parameters

- **gdf** (*GeoDataFrame*) – footprints
- **fig** (*figure*) –
- **ax** (*axis*) –
- **figsize** (*tuple*) –
- **color** (*string*) – the color of the footprints
- **bgcolor** (*string*) – the background color of the plot
- **set_bounds** (*bool*) – if True, set bounds from either passed-in bbox or the spatial extent of the gdf
- **bbox** (*tuple*) – if True and if set_bounds is True, set the display bounds to this bbox
- **save** (*bool*) – whether to save the figure to disk or not
- **show** (*bool*) – whether to display the figure or not
- **close** (*bool*) – close the figure (only if show equals False) to prevent display
- **filename** (*string*) – the name of the file to save
- **file_format** (*string*) – the format of the file to save (e.g., 'jpg', 'png', 'svg')
- **dpi** (*int*) – the resolution of the image file if saving

Returns fig, ax

Return type tuple

```
osmnx.footprints.responses_to_dicts(responses, footprint_type)
```

Parse a list of json responses into dictionaries of vertices, footprints, and relations.

Note: OSM's data model and the Overpass API will return open ways (lines) as part of a 'polygon' query. These may be fragments of the inner and outer rings of relations or they may be open ways mistakenly tagged with 'polygon' type tags.

Ways not directly tagged with the footprint type are added to the `untagged_ways` set for removal from the footprints dictionary at the end of the process.

Some inner ways of relations may be tagged with the footprint type in their own right e.g. `landuse=meadow` as an inner way in a `landuse=forest` relation and need to be kept. These are created here.

Parameters

- **responses** (*list*) – list of json responses
- **footprint_type** (*string*) – type of footprint downloaded. OSM tag key e.g. ‘building’, ‘landuse’, ‘place’, etc.

Returns

- *vertices* – dictionary of OSM nodes including their lat, lon coordinates
- *footprints* – dictionary of OSM ways including their nodes and tags
- *relations* – dictionary of OSM relations including member ids and tags
- *untagged_footprints* – set of ids for ways or relations not directly tagged with footprint_type

1.7 osmnx.geo_utils module

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

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.

Parameters *G* (*networkx multidigraph*) –

Returns *G*

Return type *networkx multidigraph*

`osmnx.geo_utils.bbox_to_poly` (*north, south, east, west*)

Convenience function to parse bbox -> poly

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

Count how many street segments emanate from each node (i.e., intersections and dead-ends) in this graph.

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

Parameters

- *G* (*networkx multidigraph*) –
- **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.geo_utils.geocode` (*query*)

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

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

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

Return type *tuple*

`osmnx.geo_utils.get_bearing` (*origin_point, destination_point*)

Calculate the bearing between two lat-long points. Each tuple should represent (lat, lng) as decimal degrees.

Parameters

- **origin_point** (*tuple*) –
- **destination_point** (*tuple*) –

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

Return type float

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

Return a subgraph of the largest weakly or strongly connected component from a directed graph.

Parameters

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

Returns G – the largest connected component subgraph from the original graph

Return type networkx multidigraph

`osmnx.geo_utils.get_nearest_edge(G, point)`

Return the nearest edge to a pair of coordinates. Pass in a graph and a tuple with the coordinates. We first get all the edges in the graph. Secondly we compute the euclidean distance from the coordinates to the segments determined by each edge. The last step is to sort the edge segments in ascending order based on the distance from the coordinates to the edge. In the end, the first element in the list of edges will be the closest edge that we will return as a tuple containing the shapely geometry, the u, v nodes, and the edge key.

Parameters

- **G** (*networkx multidigraph*) –
- **point** (*tuple*) – The (lat, lng) or (y, x) point for which we will find the nearest edge in the graph

Returns closest_edge_to_point – A geometry object representing the segment and the coordinates of the two nodes that determine the edge section, u and v, the OSM ids of the nodes, and the key for the edge.

Return type tuple (shapely.geometry, u, v, key)

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

Return the graph edges nearest to a list of points. Pass in points as separate vectors 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.

Parameters

- **G** (*networkx multidigraph*) –
- **X** (*list-like*) – The vector of longitudes or x’s 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 vector of latitudes or y’s for which we will find the nearest edge in the graph. For projected graphs, use the projected coordinates, usually in meters.
- **method** (*str {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 `osmnx.utils.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 geom; Degrees for unprojected geometries and meters for projected geometries. The smaller the value, the more points are created.

Returns

- **ne** (*ndarray*) – array of nearest edges represented by their startpoint and endpoint ids, u and v, the OSM ids of the nodes, and the edge key.
- *Info*
- *—*
- *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 the exact perpendicular point*
- *along the edge, but the smaller the *dist parameter, the closer the solution**
- *will be.*
- *Code is adapted from an answer by JHuw from this original question*
- **https** ([//gis.stackexchange.com/questions/222315/geopandas-find-nearest-point](https://gis.stackexchange.com/questions/222315/geopandas-find-nearest-point))
- *-in-other-dataframe*

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

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

Parameters

- **G** (*networkx multidigraph*) –
- **point** (*tuple*) – The (lat, lng) or (y, x) point for which we will find the nearest node in the graph
- **method** (*str {'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.geo_utils.get_nearest_nodes(G, X, Y, method=None)`

Return the graph nodes nearest to a list of points. Pass in points as separate vectors 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*) –

- **X** (*list-like*) – The vector of longitudes or x’s for which we will find the nearest node in the graph
- **Y** (*list-like*) – The vector of latitudes or y’s for which we will find the nearest node in the graph
- **method** (*str {None, 'kdtree', 'balltree'}*) – Which method to use for finding nearest node to each point. If None, we manually find each node one at a time using `osmnx.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** – list of nearest node IDs

Return type array

`osmnx.geo_utils.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*) –
- **route** (*list*) – list of nodes in the path
- **attribute** (*string*) – the name of the attribute to get the value of for each edge. If not specified, 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. Per default, a `KeyError` is raised

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

Return type list

`osmnx.geo_utils.induce_subgraph` (*G*, *node_subset*)

Induce a subgraph of G.

Parameters

- **G** (*networkx multidigraph*) –
- **node_subset** (*list-like*) – the subset of nodes to induce a subgraph of G

Returns **G2** – the subgraph of G induced by `node_subset`

Return type networkx multidigraph

`osmnx.geo_utils.overpass_json_from_file` (*filename*)

Read OSM XML from input filename and return Overpass-like JSON.

Parameters **filename** (*string*) – name of file containing OSM XML data

Returns

Return type OSMContentHandler object

`osmnx.geo_utils.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.

This code is adapted from an answer by Mike T from this original question: <https://stackoverflow.com/questions/34906124/interpolating-every-x-distance-along-multiline-in-shapely>

Parameters

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

Returns list of Point geometries

Return type list

`osmnx.geo_utils.round_linestring_coords(ls, precision)`

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

Parameters

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

Returns

Return type LineString

`osmnx.geo_utils.round_multilinestring_coords(mls, precision)`

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

Parameters

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

Returns

Return type MultiLineString

`osmnx.geo_utils.round_multipoint_coords(mpt, precision)`

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

Parameters

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

Returns

Return type MultiPoint

`osmnx.geo_utils.round_multipolygon_coords(mp, precision)`

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

Parameters

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

Returns

Return type MultiPolygon

`osmnx.geo_utils.round_point_coords(pt, precision)`

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

Parameters

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

Returns**Return type** Point`osmnx.geo_utils.round_polygon_coords(p, precision)`

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

Parameters

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

Returns **new_poly** – the polygon with rounded coordinates**Return type** shapely Polygon`osmnx.geo_utils.round_shape_coords(shape, precision)`

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

Parameters

- **shape** (*shapely geometry, one of Point, MultiPoint, LineString,*) – MultiLineString, Polygon, or MultiPolygon the geometry to round the coordinates of
- **precision** (*int*) – decimal precision to round coordinates to

Returns**Return type** shapely geometry

1.8 osmnx.osm_content_handler module

class `osmnx.osm_content_handler.OSMContentHandler`Bases: `xml.sax.handler.ContentHandler`

SAX content handler for OSM XML.

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

endElement (*name*)

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

The name parameter contains the name of the element type, just as with the `startElement` event.**startElement** (*name, attrs*)

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

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

1.9 osmnx.plot module

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

Return n-length list of RGBA colors from the passed colormap name and alpha.

Parameters

- **n** (*int*) – number of colors
- **cmap** (*string*) – name of a 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 a hexadecimal string

Returns colors

Return type list

`osmnx.plot.get_edge_colors_by_attr(G, attr, num_bins=5, cmap='viridis', start=0, stop=1, na_color='none')`

Get a list of edge colors by binning some continuous-variable attribute into quantiles.

Parameters

- **G** (*networkx multidigraph*) –
- **attr** (*string*) – the name of the continuous-variable attribute
- **num_bins** (*int*) – how many quantiles
- **cmap** (*string*) – name of a 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 null attribute values

Returns

Return type list

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

Get a list of node colors by binning some continuous-variable attribute into quantiles.

Parameters

- **G** (*networkx multidigraph*) –
- **attr** (*string*) – the name of the attribute
- **num_bins** (*int*) – how many quantiles (default None assigns each node to its own bin)
- **cmap** (*string*) – name of a 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 null attribute values

Returns

Return type list

```
osmnx.plot.make_folium_polyline(edge, edge_color, edge_width, edge_opacity,
                                popup_attribute=None)
```

Turn a row from the gdf_edges GeoDataFrame into a folium PolyLine with attributes.

Parameters

- **edge** (*GeoSeries*) – a row from the gdf_edges GeoDataFrame
- **edge_color** (*string*) – color of the edge lines
- **edge_width** (*numeric*) – width of the edge lines
- **edge_opacity** (*numeric*) – opacity of the edge lines
- **popup_attribute** (*string*) – edge attribute to display in a pop-up when an edge is clicked, if None, no popup

Returns pl

Return type folium.PolyLine

```
osmnx.plot.node_list_to_coordinate_lines(G, node_list, use_geom=True)
```

Given a list of nodes, return a list of lines that together follow the path defined by the list of nodes.

Parameters

- **G** (*networkx multidigraph*) –
- **route** (*list*) – the route as a list of nodes
- **use_geom** (*bool*) – if True, use the spatial geometry attribute of the edges to draw geographically accurate edges, rather than just lines straight from node to node

Returns lines

Return type list of lines given as pairs ((x_start, y_start), (x_stop, y_stop))

```
osmnx.plot.plot_figure_ground(G=None, address=None, point=None, dist=805, network_type='drive_service',
                              street_widths=None, default_width=4, fig_length=8, edge_color='w', bg_color='#333333',
                              smooth_joints=True, filename=None, file_format='png', show=False, save=True, close=True,
                              dpi=300)
```

Plot a figure-ground diagram of a street network, defaulting to one square mile.

Parameters

- **G** (*networkx multidigraph. must be unprojected.*) –
- **address** (*string*) – the address to geocode as the center point if G is not passed in
- **point** (*tuple*) – the center point if address and G are not passed in
- **dist** (*numeric*) – how many meters to extend north, south, east, and west from the center point
- **network_type** (*string*) – what type of network to get
- **street_widths** (*dict*) – where keys are street types and values are widths to plot in pixels
- **default_width** (*numeric*) – the default street width in pixels for any street type not found in street_widths dict
- **fig_length** (*numeric*) – the height and width of this square diagram

- **edge_color** (*string*) – the color of the streets
- **bgcolor** (*string*) – the color of the background
- **smooth_joints** (*bool*) – if True, plot nodes same width as streets to smooth line joints and prevent cracks between them from showing
- **filename** (*string*) – filename to save the image as
- **file_format** (*string*) – the format of the file to save (e.g., 'jpg', 'png', 'svg')
- **show** (*bool*) – if True, show the figure
- **save** (*bool*) – if True, save the figure as an image file to disk
- **close** (*bool*) – close the figure (only if show equals False) to prevent display
- **dpi** (*int*) – the resolution of the image file if saving

Returns *fig, ax*

Return type *tuple*

```
osmnx.plot.plot_graph(G, bbox=None, fig_height=6, fig_width=None, margin=0.02, axis_off=True,
                      equal_aspect=False, bgcolor='w', show=True, save=False, close=True,
                      file_format='png', filename='temp', dpi=300, annotate=False,
                      node_color='#66ccff', node_size=15, node_alpha=1, node_edgecolor='none',
                      node_zorder=1, edge_color='#999999', edge_linewidth=1, edge_alpha=1,
                      use_geom=True)
```

Plot a networkx spatial graph.

Parameters

- **G** (*networkx multidigraph*) –
- **bbox** (*tuple*) – bounding box as north,south,east,west - if None will calculate from spatial extents of data. if passing a bbox, you probably also want to pass margin=0 to constrain it.
- **fig_height** (*int*) – matplotlib figure height in inches
- **fig_width** (*int*) – matplotlib figure width in inches
- **margin** (*float*) – relative margin around the figure
- **axis_off** (*bool*) – if True turn off the matplotlib axis
- **equal_aspect** (*bool*) – if True set the axis aspect ratio equal
- **bgcolor** (*string*) – the background color of the figure and axis
- **show** (*bool*) – if True, show the figure
- **save** (*bool*) – if True, save the figure as an image file to disk
- **close** (*bool*) – close the figure (only if show equals False) to prevent display
- **file_format** (*string*) – the format of the file to save (e.g., 'jpg', 'png', 'svg')
- **filename** (*string*) – the name of the file if saving
- **dpi** (*int*) – the resolution of the image file if saving
- **annotate** (*bool*) – if True, annotate the nodes in the figure
- **node_color** (*string*) – the color of the nodes. color is passed to matplotlib
- **node_size** (*int*) – the size of the nodes

- **node_alpha** (*float*) – the opacity of the nodes. if you passed RGBA values to `node_color`, then set this to `None` to use the alpha channel in `node_color`
- **node_edgecolor** (*string*) – the color of the node’s marker’s border
- **node_zorder** (*int*) – zorder to plot nodes, edges are always 2, so make `node_zorder` 1 to plot nodes beneath them or 3 to plot nodes atop them
- **edge_color** (*string*) – the color of the edges’ lines. color is passed to matplotlib.
- **edge_linewidth** (*float*) – the width of the edges’ lines
- **edge_alpha** (*float*) – the opacity of the edges’ lines. if you passed RGBA values to `edge_color`, then set this to `None` to use the alpha channel in `edge_color`
- **use_geom** (*bool*) – if `True`, use the spatial geometry attribute of the edges to draw geographically accurate edges, rather than just lines straight from node to node

Returns `fig, ax`

Return type `tuple`

```
osmnx.plot.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)
```

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

Returns `graph_map`

Return type `folium.folium.Map`

```
osmnx.plot.plot_graph_route(G, route, bbox=None, fig_height=6, fig_width=None, margin=0.02,
                             bgcolor='w', axis_off=True, show=True, save=False, close=True,
                             file_format='png', filename='temp', dpi=300, annotate=False,
                             node_color='#999999', node_size=15, node_alpha=1, node_edgecolor='none',
                             node_zorder=1, edge_color='#999999', edge_linewidth=1,
                             edge_alpha=1, use_geom=True, origin_point=None,
                             destination_point=None, route_color='r', route_linewidth=4,
                             route_alpha=0.5, orig_dest_node_alpha=0.5, orig_dest_node_size=100,
                             orig_dest_node_color='r', orig_dest_point_color='b')
```

Plot a route along a networkx spatial graph.

Parameters

- **G** (*networkx multidigraph*) –
- **route** (*list*) – the route as a list of nodes
- **bbox** (*tuple*) – bounding box as north,south,east,west - if None will calculate from spatial extents of data. if passing a bbox, you probably also want to pass margin=0 to constrain it.
- **fig_height** (*int*) – matplotlib figure height in inches
- **fig_width** (*int*) – matplotlib figure width in inches
- **margin** (*float*) – relative margin around the figure
- **axis_off** (*bool*) – if True turn off the matplotlib axis
- **bgcolor** (*string*) – the background color of the figure and axis
- **show** (*bool*) – if True, show the figure
- **save** (*bool*) – if True, save the figure as an image file to disk
- **close** (*bool*) – close the figure (only if show equals False) to prevent display
- **file_format** (*string*) – the format of the file to save (e.g., 'jpg', 'png', 'svg')
- **filename** (*string*) – the name of the file if saving
- **dpi** (*int*) – the resolution of the image file if saving
- **annotate** (*bool*) – if True, annotate the nodes in the figure
- **node_color** (*string*) – the color of the nodes
- **node_size** (*int*) – the size of the nodes
- **node_alpha** (*float*) – the opacity of the nodes
- **node_edgecolor** (*string*) – the color of the node's marker's border
- **node_zorder** (*int*) – zorder to plot nodes, edges are always 2, so make node_zorder 1 to plot nodes beneath them or 3 to plot nodes atop them
- **edge_color** (*string*) – the color of the edges' lines
- **edge_linewidth** (*float*) – the width of the edges' lines
- **edge_alpha** (*float*) – the opacity of the edges' lines
- **use_geom** (*bool*) – if True, use the spatial geometry attribute of the edges to draw geographically accurate edges, rather than just lines straight from node to node
- **origin_point** (*tuple*) – optional, an origin (lat, lon) point to plot instead of the origin node
- **destination_point** (*tuple*) – optional, a destination (lat, lon) point to plot instead of the destination node
- **route_color** (*string*) – the color of the route
- **route_linewidth** (*int*) – the width of the route line
- **route_alpha** (*float*) – the opacity of the route line
- **orig_dest_node_alpha** (*float*) – the opacity of the origin and destination nodes
- **orig_dest_node_size** (*int*) – the size of the origin and destination nodes

- **orig_dest_node_color** (*string*) – the color of the origin and destination nodes
- **orig_dest_point_color** (*string*) – the color of the origin and destination points if being plotted instead of nodes

Returns *fig, ax*

Return type *tuple*

```
osmnx.plot.plot_graph_routes(G, routes, bbox=None, fig_height=6, fig_width=None, margin=0.02, bgcolor='w', axis_off=True, show=True, save=False, close=True, file_format='png', filename='temp', dpi=300, annotate=False, node_color='#999999', node_size=15, node_alpha=1, node_edgecolor='none', node_zorder=1, edge_color='#999999', edge_linewidth=1, edge_alpha=1, use_geom=True, orig_dest_points=None, route_color='r', route_linewidth=4, route_alpha=0.5, orig_dest_node_alpha=0.5, orig_dest_node_size=100, orig_dest_node_color='r', orig_dest_point_color='b')
```

Plot several routes along a networkx spatial graph.

Parameters

- **G** (*networkx multidigraph*) –
- **routes** (*list*) – the routes as a list of lists of nodes
- **bbox** (*tuple*) – bounding box as north,south,east,west - if None will calculate from spatial extents of data. if passing a bbox, you probably also want to pass margin=0 to constrain it.
- **fig_height** (*int*) – matplotlib figure height in inches
- **fig_width** (*int*) – matplotlib figure width in inches
- **margin** (*float*) – relative margin around the figure
- **axis_off** (*bool*) – if True turn off the matplotlib axis
- **bgcolor** (*string*) – the background color of the figure and axis
- **show** (*bool*) – if True, show the figure
- **save** (*bool*) – if True, save the figure as an image file to disk
- **close** (*bool*) – close the figure (only if show equals False) to prevent display
- **file_format** (*string*) – the format of the file to save (e.g., 'jpg', 'png', 'svg')
- **filename** (*string*) – the name of the file if saving
- **dpi** (*int*) – the resolution of the image file if saving
- **annotate** (*bool*) – if True, annotate the nodes in the figure
- **node_color** (*string*) – the color of the nodes
- **node_size** (*int*) – the size of the nodes
- **node_alpha** (*float*) – the opacity of the nodes
- **node_edgecolor** (*string*) – the color of the node's marker's border
- **node_zorder** (*int*) – zorder to plot nodes, edges are always 2, so make node_zorder 1 to plot nodes beneath them or 3 to plot nodes atop them
- **edge_color** (*string*) – the color of the edges' lines
- **edge_linewidth** (*float*) – the width of the edges' lines

- **edge_alpha** (*float*) – the opacity of the edges’ lines
- **use_geom** (*bool*) – if True, use the spatial geometry attribute of the edges to draw geographically accurate edges, rather than just lines straight from node to node
- **orig_dest_points** (*list of tuples*) – optional, a group of (lat, lon) points to plot instead of the origins and destinations of each route nodes
- **route_color** (*string*) – the color of the route
- **route_linewidth** (*int*) – the width of the route line
- **route_alpha** (*float*) – the opacity of the route line
- **orig_dest_node_alpha** (*float*) – the opacity of the origin and destination nodes
- **orig_dest_node_size** (*int*) – the size of the origin and destination nodes
- **orig_dest_node_color** (*string*) – the color of the origin and destination nodes
- **orig_dest_point_color** (*string*) – the color of the origin and destination points if being plotted instead of nodes

Returns fig, ax

Return type tuple

```
osmnx.plot.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)
```

Plot a route on an interactive folium web map.

Parameters

- **G** (*networkx multidigraph*) –
- **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

Returns route_map

Return type folium.folium.Map

```
osmnx.plot.plot_shape(gdf, fc='#cbe0f0', ec='#999999', linewidth=1, alpha=1, figsize=(6, 6), margin=0.02, axis_off=True)
```

Plot a GeoDataFrame of place boundary geometries.

Parameters

- **gdf** (*GeoDataFrame*) – the gdf containing the geometries to plot
- **fc** (*string or list*) – the facecolor (or list of facecolors) for the polygons

- **ec** (*string or list*) – the edgecolor (or list of edgecolors) for the polygons
- **linewidth** (*numeric*) – the width of the polygon edge lines
- **alpha** (*numeric*) – the opacity
- **figsize** (*tuple*) – the size of the plotting figure
- **margin** (*numeric*) – the size of the figure margins
- **axis_off** (*bool*) – if True, disable the matplotlib axes display

Returns `fig, ax`

Return type `tuple`

`osmnx.plot.rgb_color_list_to_hex` (*color_list*)

Convert a list of RGBA colors to a list of hexadecimal color codes.

Parameters `color_list` (*list*) – the list of RGBA colors

Returns `color_list_hex`

Return type `list`

`osmnx.plot.save_and_show` (*fig, ax, save, show, close, filename, file_format, dpi, axis_off*)

Save a figure to disk and show it, as specified.

Parameters

- **fig** (*figure*) –
- **ax** (*axis*) –
- **save** (*bool*) – whether to save the figure to disk or not
- **show** (*bool*) – whether to display the figure or not
- **close** (*bool*) – close the figure (only if show equals False) to prevent display
- **filename** (*string*) – the name of the file to save
- **file_format** (*string*) – the format of the file to save (e.g., 'jpg', 'png', 'svg')
- **dpi** (*int*) – the resolution of the image file if saving
- **axis_off** (*bool*) – if True matplotlib axis was turned off by `plot_graph` so constrain the saved figure's extent to the interior of the axis

Returns `fig, ax`

Return type `tuple`

1.10 osmnx.pois module

`osmnx.pois.create_poi_gdf` (*polygon=None, amenities=None, north=None, south=None, east=None, west=None, custom_settings=None*)

Parse GeoDataFrames from POI json that was returned by Overpass API.

Parameters

- **polygon** (*shapely Polygon or MultiPolygon*) – geographic shape to fetch the POIs within
- **amenities** (*list*) – List of amenities that will be used for finding the POIs from the selected area. See available amenities from: <http://wiki.openstreetmap.org/wiki/Key:amenity>

- **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
- **custom_settings** (*string*) – custom settings to be used in the overpass query instead of the default ones

Returns

Return type Geopandas GeoDataFrame with POIs and the associated attributes.

`osmnx.pois.invalid_multipoly_handler(gdf, relation, way_ids)`

Handles invalid multipolygon geometries when there exists e.g. a feature without geometry (geometry == NaN)

Parameters

- **gdf** (*gpd.GeoDataFrame*) – GeoDataFrame with Polygon geometries that should be converted into a MultiPolygon object.
- **relation** (*dict*) – OSM ‘relation’ dictionary
- **way_ids** (*list*) – A list of ‘way’ ids that should be converted into a MultiPolygon object.

`osmnx.pois.osm_poi_download(polygon=None, amenities=None, north=None, south=None, east=None, west=None, timeout=180, max_query_area_size=2500000000, custom_settings=None)`

Get points of interests (POIs) from OpenStreetMap based on selected amenity types. Note that if a polygon is passed-in, the query will be limited to its bounding box rather than to the shape of the polygon itself.

Parameters

- **poly** (*shapely.geometry.Polygon*) – Polygon that will be used to limit the POI search.
- **amenities** (*list*) – List of amenities that will be used for finding the POIs from the selected area.
- **custom_settings** (*string*) – custom settings to be used in the overpass query instead of the default ones

Returns **gdf** – Points of interest and the tags associated with them as geopandas GeoDataFrame.

Return type geopandas.GeoDataFrame

`osmnx.pois.parse_nodes_coords(osm_response)`

Parse node coordinates from OSM response. Some nodes are standalone points of interest, others are vertices in polygonal (areal) POIs.

Parameters **osm_response** (*string*) – OSM response JSON string

Returns **coords** – dict of node IDs and their lat, lon coordinates

Return type dict

`osmnx.pois.parse_osm_node(response)`

Parse points from OSM nodes.

Parameters **response** (*JSON*) – Nodes from OSM response.

Returns

Return type Dict of vertex IDs and their lat, lon coordinates.

`osmnx.pois.parse_osm_relations` (*relations*, *osm_way_df*)

Parses the osm relations (multipolygons) from osm ways and nodes. See more information about relations from OSM documentation: <http://wiki.openstreetmap.org/wiki/Relation>

Parameters

- **relations** (*list*) – OSM ‘relation’ items (dictionaries) in a list.
- **osm_way_df** (*gpd.GeoDataFrame*) – OSM ‘way’ features as a GeoDataFrame that contains all the ‘way’ features that will constitute the multipolygon relations.

Returns A GeoDataFrame with MultiPolygon representations of the relations and the attributes associated with them.

Return type `gpd.GeoDataFrame`

`osmnx.pois.parse_poi_query` (*north*, *south*, *east*, *west*, *amenities=None*, *timeout=180*, *maxsize=""*, *custom_settings=None*)

Parse the Overpass QL query based on the list of amenities.

Parameters

- **north** (*float*) – Northernmost coordinate from bounding box of the search area.
- **south** (*float*) – Southernmost coordinate from bounding box of the search area.
- **east** (*float*) – Easternmost coordinate from bounding box of the search area.
- **west** (*float*) – Westernmost coordinate of the bounding box of the search area.
- **amenities** (*list*) – List of amenities that will be used for finding the POIs from the selected area.
- **timeout** (*int*) – Timeout for the API request.
- **custom_settings** (*string*) – custom settings to be used in the overpass query instead of the default ones

`osmnx.pois.parse_polygonal_poi` (*coords*, *response*)

Parse areal POI way polygons from OSM node coords.

Parameters **coords** (*dict*) – dict of node IDs and their lat, lon coordinates

Returns

Return type dict of POIs containing each’s nodes, polygon geometry, and osmid

`osmnx.pois.pois_from_address` (*address*, *distance*, *amenities=None*, *custom_settings=None*)

Get OSM points of Interests within some distance north, south, east, and west of an address.

Parameters

- **address** (*string*) – the address to geocode to a lat-long point
- **distance** (*numeric*) – distance in meters
- **amenities** (*list*) – List of amenities that will be used for finding the POIs from the selected area. See available amenities from: <http://wiki.openstreetmap.org/wiki/Key:amenity>
- **custom_settings** (*string*) – custom settings to be used in the overpass query instead of the default ones

Returns

Return type `GeoDataFrame`

`osmnx.pois.pois_from_place` (*place*, *amenities=None*, *which_result=1*, *custom_settings=None*)

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

Parameters

- **place** (*string*) – the query to geocode to get geojson boundary polygon.
- **amenities** (*list*) – List of amenities that will be used for finding the POIs from the selected area. See available amenities from: <http://wiki.openstreetmap.org/wiki/Key:amenity>
- **which_result** (*int*) – max number of place geocoding results to return and which to process upon receipt
- **custom_settings** (*string*) – custom settings to be used in the overpass query instead of the default ones

Returns**Return type** GeoDataFrame

`osmnx.pois.pois_from_point` (*point*, *distance=None*, *amenities=None*, *custom_settings=None*)
Get point of interests (POIs) within some distance north, south, east, and west of a lat-long point.

Parameters

- **point** (*tuple*) – a lat-long point
- **distance** (*numeric*) – distance in meters
- **amenities** (*list*) – List of amenities that will be used for finding the POIs from the selected area. See available amenities from: <http://wiki.openstreetmap.org/wiki/Key:amenity>
- **custom_settings** (*string*) – custom settings to be used in the overpass query instead of the default ones

Returns**Return type** GeoDataFrame

`osmnx.pois.pois_from_polygon` (*polygon*, *amenities=None*, *custom_settings=None*)
Get OSM points of interest within some polygon.

Parameters

- **polygon** (*Polygon*) – Polygon where the POIs are search from.
- **amenities** (*list*) – List of amenities that will be used for finding the POIs from the selected area. See available amenities from: <http://wiki.openstreetmap.org/wiki/Key:amenity>
- **custom_settings** (*string*) – custom settings to be used in the overpass query instead of the default ones

Returns**Return type** GeoDataFrame

1.11 osmnx.projection module

`osmnx.projection.is_crs_utm` (*crs*)
Determine if a CRS is a UTM CRS

Parameters *crs* (*dict or string or pyproj.CRS*) – a coordinate reference system**Returns** True if crs is UTM, False otherwise**Return type** bool

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

Project a GeoDataFrame to the UTM zone appropriate for its geometries' centroid.

The simple calculation in this function works well for most latitudes, but won't work for some far northern locations like Svalbard and parts of far northern Norway.

Parameters

- **gdf** (*GeoDataFrame*) – the gdf to be projected
- **to_crs** (*dict or string or pyproj.CRS*) – if not None, just project to this CRS instead of to UTM
- **to_latlong** (*bool*) – if True, projects to latlong instead of to UTM

Returns

Return type *GeoDataFrame*

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

Project a shapely Polygon or MultiPolygon from lat-long to UTM, or vice-versa

Parameters

- **geometry** (*shapely Polygon or MultiPolygon*) – the geometry to project
- **crs** (*dict or string or pyproj.CRS*) – the starting coordinate reference system of the passed-in geometry, default value (None) will set settings.default_crs as the CRS
- **to_crs** (*dict or string or pyproj.CRS*) – if not None, just project to this CRS instead of to UTM
- **to_latlong** (*bool*) – if True, project from crs to lat-long, if False, project from crs to local UTM zone

Returns (*geometry_proj, crs*), the projected shapely geometry and the crs of the projected geometry

Return type *tuple*

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

Project a graph from lat-long to the UTM zone appropriate for its geographic location.

Parameters

- **G** (*networkx multidigraph*) – the networkx graph to be projected
- **to_crs** (*dict or string or pyproj.CRS*) – if not None, just project to this CRS instead of to UTM

Returns

Return type *networkx multidigraph*

1.12 osmnx.save_load module

`osmnx.save_load.gdfs_to_graph(gdf_nodes, gdf_edges)`

Convert node and edge GeoDataFrames into a graph

Parameters

- **gdf_nodes** (*GeoDataFrame*) –
- **gdf_edges** (*GeoDataFrame*) –

Returns

Return type networkx multidigraph

`osmnx.save_load.get_undirected(G)`

Convert a directed graph to an undirected graph that maintains parallel edges if geometries differ.

Parameters *G* (*networkx multidigraph*) –

Returns

Return type networkx multigraph

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

Convert a graph into node and/or edge GeoDataFrames

Parameters

- *G* (*networkx multidigraph*) –
- **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 origin and destination nodes

Returns *gdf_nodes* or *gdf_edges* or both as a tuple

Return type GeoDataFrame or tuple

`osmnx.save_load.is_duplicate_edge(data, data_other)`

Check if two edge data dictionaries are the same based on OSM ID and geometry.

Parameters

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

Returns *is_dupe*

Return type bool

`osmnx.save_load.is_same_geometry(ls1, ls2)`

Check if LineString geometries in two edges are the same, in normal or reversed order of points.

Parameters

- **ls1** (*LineString*) – the first edge's geometry
- **ls2** (*LineString*) – the second edge's geometry

Returns

Return type bool

`osmnx.save_load.load_graphml(filename, folder=None, node_type=<class 'int'>)`

Load a GraphML file from disk and convert the node/edge attributes to correct data types.

Parameters

- **filename** (*string*) – the name of the graphml file (including file extension)
- **folder** (*string*) – the folder containing the file, if None, use default data folder
- **node_type** (*type*) – (Python type (default: int)) - Convert node ids to this type

Returns

Return type networkx multidigraph

`osmnx.save_load.make_shp_filename(place_name)`

Create a filename string in a consistent format from a place name string.

Parameters `place_name` (*string*) – place name to convert into a filename

Returns

Return type string

`osmnx.save_load.save_as_osm(data, 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, filename='graph.osm', folder=None)`

Save a graph as an OSM XML formatted file. NOTE: for very large networks this method can take upwards of 30+ minutes to finish.

Parameters

- **data** (*networkx multi(di)graph OR a length 2 iterable of nodes/edges*) – geopandas.GeoDataFrames
- **filename** (*string*) – the name of the osm file (including file extension)
- **folder** (*string*) – the folder to contain the file, if None, use default data folder
- **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.save_load.save_gdf_shapefile(gdf, filename=None, folder=None)`

Save a GeoDataFrame of place shapes or footprints as an ESRI shapefile.

Parameters

- **gdf** (*GeoDataFrame*) – the gdf to be saved
- **filename** (*string*) – the name of the shapefiles (not including file extensions)

- **folder** (*string*) – where to save the shapefile, if none, then default folder

Returns

Return type None

```
osmnx.save_load.save_graph_geopackage (G, filename='graph.gpkg', folder=None, encoding='utf-8')
```

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

Parameters

- **G** (*networkx multidigraph*) –
- **filename** (*string*) – the filename of the GeoPackage including file extension
- **folder** (*string*) – the path to the folder to contain the GeoPackage, if None, use default data folder
- **encoding** (*string*) – the character encoding for the saved files

Returns

Return type None

```
osmnx.save_load.save_graph_shapefile (G, filename='graph', folder=None, encoding='utf-8')
```

Save graph nodes and edges as ESRI shapefiles to disk.

Parameters

- **G** (*networkx multidigraph*) –
- **filename** (*string*) – the name of the shapefiles (not including file extensions)
- **folder** (*string*) – the folder to contain the shapefiles, if None, use default data folder
- **encoding** (*string*) – the character encoding for the saved shapefiles

Returns

Return type None

```
osmnx.save_load.save_graphml (G, filename='graph.graphml', folder=None, gephi=False)
```

Save graph as GraphML file to disk.

Parameters

- **G** (*networkx multidigraph*) –
- **filename** (*string*) – the name of the graphml file (including file extension)
- **folder** (*string*) – the folder to contain the file, if None, use default data folder
- **gephi** (*bool*) – if True, give each edge a unique key to work around Gephi's restrictive interpretation of the GraphML specification

Returns

Return type None

```
osmnx.save_load.update_edge_keys (G)
```

Update the keys of edges that share a u, v with another edge but differ in geometry. For example, two one-way streets from u to v that bow away from each other as separate streets, rather than opposite direction edges of a single street.

Parameters **G** (*networkx multidigraph*) –

Returns

Return type networkx multigraph

1.13 osmnx.settings module

1.14 osmnx.simplify module

`osmnx.simplify.build_path(G, node, endpoints, path)`

Recursively build a path of nodes until you hit an endpoint node.

Parameters

- **G** (*networkx multidigraph*) –
- **node** (*int*) – the current node to start from
- **endpoints** (*set*) – the set of all nodes in the graph that are endpoints
- **path** (*list*) – the list of nodes in order in the path so far

Returns `paths_to_simplify`

Return type list

`osmnx.simplify.clean_intersections(G, tolerance=15, dead_ends=False)`

Clean-up intersections comprising clusters of nodes by merging them and returning their centroids.

Divided roads are 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. This function cleans them up by buffering their points to an arbitrary distance, merging overlapping buffers, and taking their centroid. For best results, the tolerance argument should be adjusted to approximately match street design standards in the specific street network.

Parameters

- **G** (*networkx multidigraph*) –
- **tolerance** (*float*) – nodes within this distance (in graph's geometry's units) will be dissolved into a single intersection
- **dead_ends** (*bool*) – if False, discard dead-end nodes to return only street-intersection points

Returns `intersection_centroids` – a GeoSeries of shapely Points representing the centroids of street intersections

Return type geopandas.GeoSeries

`osmnx.simplify.get_paths_to_simplify(G, strict=True)`

Create a list of all the paths to be simplified between endpoint nodes.

The path is ordered from the first endpoint, through the interstitial nodes, to the second endpoint. If your street network is in a rural area with many interstitial nodes between true edge endpoints, you may want to increase your system's recursion limit to avoid recursion errors.

Parameters

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

Returns `paths_to_simplify`

Return type list

`osmnx.simplify.is_endpoint(G, node, strict=True)`

Return True if the node is a “real” endpoint of an edge in the network, otherwise False. OSM data includes lots of nodes that exist only as points to help streets bend around curves. An end point is a node that either: 1) is its own neighbor, ie, it self-loops. 2) or, has no incoming edges or no outgoing edges, ie, all its incident edges point inward or all its incident edges point outward. 3) or, it does not have exactly two neighbors and degree of 2 or 4. 4) or, if strict mode is false, if its edges have different OSM IDs. :param G: :type G: networkx multidigraph :param node: the node to examine :type node: int :param strict: if False, allow nodes to be end points even if they fail all other rules but have edges with different OSM IDs :type strict: bool

Returns

Return type bool

`osmnx.simplify.is_simplified(G)`

Determine if a graph has already had its topology simplified.

If any of its edges have a geometry attribute, we know that it has previously been simplified.

Parameters *G* (networkx multidigraph) –

Returns

Return type bool

`osmnx.simplify.simplify_graph(G, strict=True)`

Simplify a graph’s 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 attribute in new edge.

Parameters

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

Returns

Return type networkx multidigraph

1.15 osmnx.stats module

`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) –
- **area** (numeric) – the area covered by the street network, in square meters (typically land area); if none, will skip all density-based metrics
- **clean_intersects** (bool) – if True, calculate clean intersections count (and density, if area is provided)

- **tolerance** (*numeric*) – tolerance value passed along if `clean_intersects=True`, see `clean_intersections()` function documentation for details and usage
- **circuitry_dist** (*str*) – ‘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*) –
- **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

1.16 osmnx.utils module

`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>

```
osmnx.utils.config(data_folder='data', logs_folder='logs', imgs_folder='images',
                  cache_folder='cache', use_cache=False, log_file=False, log_console=False,
                  log_level=20, log_name='osmnx', log_filename='osmnx', useful_tags_node=['ref',
                  'highway'], useful_tags_path=['bridge', 'tunnel', 'oneway', 'lanes', 'ref',
                  'name', 'highway', 'maxspeed', 'service', 'access', 'area', 'landuse', 'width',
                  'est_width', 'junction'], osm_xml_node_attrs=['id', 'timestamp', 'uid',
                  'user', 'version', 'changeset', 'lat', 'lon'], osm_xml_node_tags=['highway'],
                  osm_xml_way_attrs=['id', 'timestamp', 'uid', 'user', 'version', 'changeset'],
                  osm_xml_way_tags=['highway', 'lanes', 'maxspeed', 'name', 'oneway'], de-
                  fault_access='["access"!~"private"]', default_crs='+proj=longlat +ellps=WGS84
                  +datum=WGS84 +no_defs', default_user_agent='Python OSMnx package
                  (https://github.com/gboeing/osmnx)', default_referer='Python OSMnx package
                  (https://github.com/gboeing/osmnx)', default_accept_language='en', nomina-
                  tim_endpoint='https://nominatim.openstreetmap.org/', nominatim_key=None,
                  overpass_endpoint='http://overpass-api.de/api', all_oneway=False)
```

Configure osmnx by setting the default global vars to desired values.

Parameters

- **data_folder** (*string*) – where to save and load data files
- **logs_folder** (*string*) – where to write the log files
- **imgs_folder** (*string*) – where to save figures
- **cache_folder** (*string*) – where to save the http response cache
- **use_cache** (*bool*) – if True, use a local cache to save/retrieve http responses instead of calling API repetitively for the same request URL
- **log_file** (*bool*) – if true, save log output to a log file in logs_folder
- **log_console** (*bool*) – if true, print log output to the console
- **log_level** (*int*) – one of the logger.level constants
- **log_name** (*string*) – name of the logger
- **useful_tags_node** (*list*) – a list of useful OSM tags to attempt to save from node elements

- **useful_tags_path** (*list*) – a list of useful OSM tags to attempt to save from path elements
- **default_access** (*string*) – default filter for OSM “access” key
- **default_crs** (*string*) – default CRS 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*) – which 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*) – which API endpoint to use for overpass queries
- **all_oneway** (*boolean*) – if True, forces all paths to be loaded as oneway ways, preserving the original order of nodes stored in the OSM way XML.

Returns

Return type None

`osmnx.utils.euclidean_dist_vec` (*y1, x1, y2, x2*)

Vectorized function to calculate the euclidean distance between two points or between vectors of points.

Parameters

- **y1** (*float or array of float*) –
- **x1** (*float or array of float*) –
- **y2** (*float or array of float*) –
- **x2** (*float or array of float*) –

Returns **distance** – distance or vector of distances from (x1, y1) to (x2, y2) in graph units

Return type float or array of float

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

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

Parameters

- **level** (*int*) – one of the logger.level constants
- **name** (*string*) – name of the logger
- **filename** (*string*) – name of the log file

Returns

Return type logger.logger

`osmnx.utils.get_unique_nodes_ordered_from_way` (*way_edges_df*)

Function to recover the original order of nodes from a dataframe of edges associated with a single OSM way.

Parameters **way_edges_df** (*pandas.DataFrame()*) – Dataframe containing columns ‘u’ and ‘v’ corresponding to origin/desitination nodes.

Returns

- **unique_ordered_nodes** (*list*) – An ordered list of unique node IDs

- **NOTE** *(If the edges do not all connect (e.g. [(1, 2), (2,3), (10, 11), (11, 12), (12, 13)]), then this method will return only those nodes associated with the largest component of connected edges, even if subsequent connected chunks are contain more total nodes. This is done to ensure a proper topological representation of nodes in the XML way records because if there are unconnected components, the sorting algorithm cannot recover their original order. I don't believe that we would ever encounter this kind of disconnected structure of nodes within a given way, but as best I could tell it is not explicitly forbidden in the OSM XML design schema. I'm using a print statement right now to tell the user whether or not any nodes have been dropped and how many.*

`osmnx.utils.great_circle_vec(lat1, lng1, lat2, lng2, earth_radius=6371009)`

Vectorized function to calculate the great-circle distance between two points or between vectors of points, using haversine.

Parameters

- **lat1** (*float or array of float*) –
- **lng1** (*float or array of float*) –
- **lat2** (*float or array of float*) –
- **lng2** (*float or array of float*) –
- **earth_radius** (*numeric*) – radius of earth in units in which distance will be returned (default is meters)

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

Return type float or vector of floats

`osmnx.utils.log(message, level=None, name=None, filename=None)`

Write a message to the log file and/or print to the the console.

Parameters

- **message** (*string*) – the content of the message to log
- **level** (*int*) – one of the logger.level constants
- **name** (*string*) – name of the logger
- **filename** (*string*) – name of the log file

Returns

Return type None

`osmnx.utils.make_str(value)`

Convert a passed-in value to unicode if Python 2, or string if Python 3.

Parameters **value** (*any*) – the value to convert to unicode/string

Returns**Return type** unicode or string`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 { 'datetime', 'date', 'time' }
- **template** (*string*) – if not None, format the timestamp with this template

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

1.17 Module contents

OSMnx: retrieve, model, analyze, and visualize street networks from OpenStreetMap. OSMnx is a Python package that lets you download spatial geometries and model, project, visualize, and analyze street networks from OpenStreetMap's APIs. Users can download and model walkable, drivable, or bikable 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, and network routing.

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 works with OpenStreetMap's APIs to:

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

Examples and demonstrations of these features are in the GitHub repo (see below). 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 its official [docker container](#), or you can install OSMnx via pip if you already have all of its dependencies installed on your system.

EXAMPLES

For examples and 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`

PYTHON MODULE INDEX

O

- `osmnx`, [46](#)
- `osmnx.core`, [1](#)
- `osmnx.downloader`, [11](#)
- `osmnx.elevation`, [13](#)
- `osmnx.errors`, [13](#)
- `osmnx.footprints`, [14](#)
- `osmnx.geo_utils`, [18](#)
- `osmnx.osm_content_handler`, [23](#)
- `osmnx.plot`, [24](#)
- `osmnx.pois`, [31](#)
- `osmnx.projection`, [34](#)
- `osmnx.save_load`, [35](#)
- `osmnx.settings`, [39](#)
- `osmnx.simplify`, [39](#)
- `osmnx.stats`, [40](#)
- `osmnx.utils`, [43](#)

A

`add_edge_bearings()` (in module *osmnx.geo_utils*), 18
`add_edge_grades()` (in module *osmnx.elevation*), 13
`add_edge_lengths()` (in module *osmnx.core*), 1
`add_node_elevations()` (in module *osmnx.elevation*), 13
`add_path()` (in module *osmnx.core*), 1
`add_paths()` (in module *osmnx.core*), 1

B

`basic_stats()` (in module *osmnx.stats*), 40
`bbox_from_point()` (in module *osmnx.core*), 1
`bbox_to_poly()` (in module *osmnx.geo_utils*), 18
`build_path()` (in module *osmnx.simplify*), 39

C

`citation()` (in module *osmnx.utils*), 43
`clean_intersections()` (in module *osmnx.simplify*), 39
`config()` (in module *osmnx.utils*), 43
`consolidate_subdivide_geometry()` (in module *osmnx.core*), 2
`count_streets_per_node()` (in module *osmnx.geo_utils*), 18
`create_footprint_geometry()` (in module *osmnx.footprints*), 14
`create_footprints_gdf()` (in module *osmnx.footprints*), 14
`create_graph()` (in module *osmnx.core*), 2
`create_poi_gdf()` (in module *osmnx.pois*), 31
`create_relation_geometry()` (in module *osmnx.footprints*), 14

E

`EmptyOverpassResponse`, 13
`endElement()` (*osmnx.osm_content_handler.OSMContentHandler* method), 23
`euclidean_dist_vec()` (in module *osmnx.utils*), 44
`extended_stats()` (in module *osmnx.stats*), 41

F

`footprints_from_address()` (in module *osmnx.footprints*), 15
`footprints_from_place()` (in module *osmnx.footprints*), 15
`footprints_from_point()` (in module *osmnx.footprints*), 15
`footprints_from_polygon()` (in module *osmnx.footprints*), 16

G

`gdf_from_place()` (in module *osmnx.core*), 2
`gdf_from_places()` (in module *osmnx.core*), 2
`gdfs_to_graph()` (in module *osmnx.save_load*), 35
`geocode()` (in module *osmnx.geo_utils*), 18
`get_bearing()` (in module *osmnx.geo_utils*), 18
`get_colors()` (in module *osmnx.plot*), 24
`get_edge_colors_by_attr()` (in module *osmnx.plot*), 24
`get_from_cache()` (in module *osmnx.downloader*), 11
`get_http_headers()` (in module *osmnx.downloader*), 11
`get_largest_component()` (in module *osmnx.geo_utils*), 19
`get_logger()` (in module *osmnx.utils*), 44
`get_nearest_edge()` (in module *osmnx.geo_utils*), 19
`get_nearest_edges()` (in module *osmnx.geo_utils*), 19
`get_nearest_node()` (in module *osmnx.geo_utils*), 20
`get_nearest_nodes()` (in module *osmnx.geo_utils*), 20
`get_node()` (in module *osmnx.core*), 3
`get_node_colors_by_attr()` (in module *osmnx.plot*), 24
`get_osm_filter()` (in module *osmnx.downloader*), 11
`get_path()` (in module *osmnx.core*), 3
`get_paths_to_simplify()` (in module *osmnx.simplify*), 39

`get_pause_duration()` (in module *osmnx.downloader*), 11
`get_polygons_coordinates()` (in module *osmnx.core*), 3
`get_route_edge_attributes()` (in module *osmnx.geo_utils*), 21
`get_undirected()` (in module *osmnx.save_load*), 36
`get_unique_nodes_ordered_from_way()` (in module *osmnx.utils*), 44
`graph_from_address()` (in module *osmnx.core*), 3
`graph_from_bbox()` (in module *osmnx.core*), 4
`graph_from_file()` (in module *osmnx.core*), 5
`graph_from_place()` (in module *osmnx.core*), 5
`graph_from_point()` (in module *osmnx.core*), 6
`graph_from_polygon()` (in module *osmnx.core*), 7
`graph_to_gdfs()` (in module *osmnx.save_load*), 36
`great_circle_vec()` (in module *osmnx.utils*), 45

I

`induce_subgraph()` (in module *osmnx.geo_utils*), 21
`InsufficientNetworkQueryArguments`, 13
`intersect_index_quadrats()` (in module *osmnx.core*), 8
`invalid_multipoly_handler()` (in module *osmnx.pois*), 32
`InvalidDistanceType`, 13
`is_crs_utm()` (in module *osmnx.projection*), 34
`is_duplicate_edge()` (in module *osmnx.save_load*), 36
`is_endpoint()` (in module *osmnx.simplify*), 40
`is_same_geometry()` (in module *osmnx.save_load*), 36
`is_simplified()` (in module *osmnx.simplify*), 40

L

`load_graphml()` (in module *osmnx.save_load*), 36
`log()` (in module *osmnx.utils*), 45

M

`make_folium_polyline()` (in module *osmnx.plot*), 25
`make_shp_filename()` (in module *osmnx.save_load*), 37
`make_str()` (in module *osmnx.utils*), 45

N

`node_list_to_coordinate_lines()` (in module *osmnx.plot*), 25
`nominatim_request()` (in module *osmnx.downloader*), 11

O

`osm_footprints_download()` (in module *osmnx.footprints*), 16
`osm_net_download()` (in module *osmnx.core*), 8
`osm_poi_download()` (in module *osmnx.pois*), 32
`osm_polygon_download()` (in module *osmnx.downloader*), 12
`OSMContentHandler` (class in *osmnx.osm_content_handler*), 23
osmnx (module), 46
osmnx.core (module), 1
osmnx.downloader (module), 11
osmnx.elevation (module), 13
osmnx.errors (module), 13
osmnx.footprints (module), 14
osmnx.geo_utils (module), 18
osmnx.osm_content_handler (module), 23
osmnx.plot (module), 24
osmnx.pois (module), 31
osmnx.projection (module), 34
osmnx.save_load (module), 35
osmnx.settings (module), 39
osmnx.simplify (module), 39
osmnx.stats (module), 40
osmnx.utils (module), 43
`overpass_json_from_file()` (in module *osmnx.geo_utils*), 21
`overpass_request()` (in module *osmnx.downloader*), 12

P

`parse_nodes_coords()` (in module *osmnx.pois*), 32
`parse_osm_node()` (in module *osmnx.pois*), 32
`parse_osm_nodes_paths()` (in module *osmnx.core*), 9
`parse_osm_relations()` (in module *osmnx.pois*), 32
`parse_poi_query()` (in module *osmnx.pois*), 33
`parse_polygonal_poi()` (in module *osmnx.pois*), 33
`plot_figure_ground()` (in module *osmnx.plot*), 25
`plot_footprints()` (in module *osmnx.footprints*), 17
`plot_graph()` (in module *osmnx.plot*), 26
`plot_graph_folium()` (in module *osmnx.plot*), 27
`plot_graph_route()` (in module *osmnx.plot*), 27
`plot_graph_routes()` (in module *osmnx.plot*), 29
`plot_route_folium()` (in module *osmnx.plot*), 30
`plot_shape()` (in module *osmnx.plot*), 30
`pois_from_address()` (in module *osmnx.pois*), 33
`pois_from_place()` (in module *osmnx.pois*), 33
`pois_from_point()` (in module *osmnx.pois*), 34
`pois_from_polygon()` (in module *osmnx.pois*), 34
`project_gdf()` (in module *osmnx.projection*), 34

`project_geometry()` (in module *osmnx.projection*), 35
`project_graph()` (in module *osmnx.projection*), 35
`truncate_graph_polygon()` (in module *osmnx.core*), 10
`ts()` (in module *osmnx.utils*), 46

Q

`quadrat_cut_geometry()` (in module *osmnx.core*), 9

R

`redistribute_vertices()` (in module *osmnx.geo_utils*), 21
`remove_isolated_nodes()` (in module *osmnx.core*), 9
`responses_to_dicts()` (in module *osmnx.footprints*), 17
`rgb_color_list_to_hex()` (in module *osmnx.plot*), 31
`round_linestring_coords()` (in module *osmnx.geo_utils*), 22
`round_multilinestring_coords()` (in module *osmnx.geo_utils*), 22
`round_multipoint_coords()` (in module *osmnx.geo_utils*), 22
`round_multipolygon_coords()` (in module *osmnx.geo_utils*), 22
`round_point_coords()` (in module *osmnx.geo_utils*), 22
`round_polygon_coords()` (in module *osmnx.geo_utils*), 23
`round_shape_coords()` (in module *osmnx.geo_utils*), 23

S

`save_and_show()` (in module *osmnx.plot*), 31
`save_as_osm()` (in module *osmnx.save_load*), 37
`save_gdf_shapefile()` (in module *osmnx.save_load*), 37
`save_graph_geopackage()` (in module *osmnx.save_load*), 38
`save_graph_shapefile()` (in module *osmnx.save_load*), 38
`save_graphml()` (in module *osmnx.save_load*), 38
`save_to_cache()` (in module *osmnx.downloader*), 12
`simplify_graph()` (in module *osmnx.simplify*), 40
`startElement()` (*osmnx.osm_content_handler.OSMContentHandler* method), 23

T

`truncate_graph_bbox()` (in module *osmnx.core*), 9
`truncate_graph_dist()` (in module *osmnx.core*), 10

U

`UnknownNetworkType`, 13
`update_edge_keys()` (in module *osmnx.save_load*), 38
`url_in_cache()` (in module *osmnx.downloader*), 12