# GeoBases Documentation

***Release 5***

## OpenTravelData

**Oct 09, 2017**

# Contents

# CHAPTER 1

## Indices and tables

- genindex
- modindex
- search

This module defines a class *GeoBase* to manipulate geographical data (or not). It loads static files containing data, then provides tools to play with it.

It relies on four other modules:

- *GeoUtils*: to compute haversine distances between points
- *LevenshteinUtils*: to calculate distances between strings. Indeed, we need a good tool to do it, in order to recognize things like station names in schedule files where we do not have the station id
- *GeoGridModule*: to handle geographical indexation
- *SourcesManagerModule*: to handle data sources

Examples for airports:

```
>>> geo_a = GeoBase(data='airports', verbose=False)
>>> sorted(geo_a.findNearKey('ORY', 50)) # Orly, airports <= 50km
[(0.0, 'ORY'), (18.8..., 'TNF'), (27.8..., 'LBG'), (34.8..., 'CDG')]
>>> geo_a.get('CDG', 'city_code')
'PAR'
>>> geo_a.distance('CDG', 'NCE')
694.516...
```

Examples for stations:

```
>>> geo_t = GeoBase(data='stations', verbose=False)
>>>
>>> # Nice, stations <= 5km
>>> point = (43.70, 7.26)
>>> [geo_t.get(k, 'name') for d, k in sorted(geo_t.findNearPoint(point, 3))]
['Nice-Ville', 'Nice-Riquier', 'Nice-St-Roch']
```

```
>>>
>>> geo_t.get('frpaz', 'name')
'Paris-Austerlitz'
>>> geo_t.distance('frnic', 'frpaz')
683.526...
```

From any point of reference, we have a few duplicates even with `('iata_code', 'location_type')` key:

```
>>> geo = GeoBase(data='optd_por', key_fields=['iata_code', 'location_type'])
In skipped zone, dropping line 1: "iata_code...".
/!\ [lno ...] CRK+C is duplicated #1, first found lno ...: creation of ...
/!\ [lno ...] EAP+C is duplicated #1, first found lno ...: creation of ...
/!\ [lno ...] OSF+C is duplicated #1, first found lno ...: creation of ...
/!\ [lno ...] RDU+C is duplicated #1, first found lno ...: creation of ...
Import successful from ...
Available fields for things: ...
```

**class** GeoBases.GeoBaseModule.**GeoBase**(*data*, ***kwargs*)

Bases: *GeoBases.VisualMixinModule.VisualMixin*

This is the main and only class. After __init__, a file is loaded in memory, and the user may use the instance to get information.

**__init__**(*data*, ***kwargs*)

Initialization

The `kwargs` parameters given when creating the object may be:

- source : `None` by default, file-like to the source
- paths : `None` by default, path or list of paths to the source. This will only be used if source is `None`.
- headers : `[]` by default, list of fields in the data
- key_fields : `None` by default, list of fields defining the key for a line, `None` means line numbers will be used to generate keys
- indices : `[]` by default, an iterable of additional indexed fields
- delimiter : `'^'` by default, delimiter for each field,
- subdelimiters : `{}` by default, a `{ 'field' : 'delimiter' }` dict to define subdelimiters
- join : `[]` by default, list of dict defining join clauses. A join clause is a dict `{ 'fields' : fields, 'with' : [base, fields]}`, for example `{ 'fields' : 'country_code', 'with' : ['countries', 'code']}`
- quotechar : `'"'` by default, this is the string defined for quoting
- limit : `None` by default, put an int if you want to load only the first lines
- skip : `None` by default, put an int if you want to skip the first lines during loading
- discard_dups : `False` by default, boolean to discard key duplicates or handle them
- verbose : `True` by default, toggle verbosity

**Parameters**

- **data** – the type of data, `'airports'`, `'stations'`, and many more available. `'feed'` will create an empty instance.
- **kwargs** – additional parameters

**Raises** `ValueError`, if data parameters is not recognized

**Returns** `None`

```
>>> geo_a = GeoBase(data='airports')
Import successful from ...
Available fields for things: ...
>>> geo_t = GeoBase(data='stations')
Import successful from ...
Available fields for things: ...
>>> geo_f = GeoBase(data='feed')
No source specified, skipping loading...
Available fields for things: ...
No geocode support, skipping grid...
>>> geo_c = GeoBase(data='odd')
Traceback (most recent call last):
ValueError: Wrong data type "odd". Not in ['aircraft', ...]
```

Import some custom data.

```
>>> p = 'DataSources/Airports/GeoNames/airports_geonames_only_clean.csv'
>>> fl = open(op.join(op.realpath(op.dirname(__file__)), p))
>>> GeoBase(data='feed',
...         source=fl,
...         headers=['iata_code', 'name', 'city'],
...         key_fields='iata_code',
...         delimiter='^',
...         verbose=False).get('ORY', 'name')
'Paris-Orly'
>>> fl.close()
>>> GeoBase(data='airports',
...         headers=['iata_code', 'cname', 'city'],
...         join=[],
...         verbose=False).get('ORY', 'cname')
'Paris-Orly'
```

**addGrid** (*radius=50*, *precision=5*, *force=False*, *verbose=True*)

Create the grid for geographical indexation.

This operation is automatically performed an initialization if there is geocode support in headers.

**Parameters**

- **radius** – the grid accuracy, in kilometers the `precision` parameter is used to define grid size
- **precision** – the hash length. This is only used if `radius` is `None`, otherwise this parameter (a hash length) is computed from the radius
- **force** – `False` by default, force grid update if it already exists
- **verbose** – toggle verbosity

**Returns** `None`

```
>>> geo_o.addGrid(radius=50, force=True, verbose=True)
/!\ Grid already built, overriding...
```

**addIndex** (*fields*, *force=False*, *verbose=True*)

Add an index on an iterable of fields.

**Parameters**

- **fields** – the iterable of fields
- **force** – `False` by default, force index update if it already exists
- **verbose** – toggle verbosity

```
>>> geo_o.addIndex('iata_code', force=True, verbose=True)
/!\ Index on ('iata_code',) already built, overriding...
Built index for fields ('iata_code',)
```

Index on multiple fields.

```
>>> geo_o.addIndex(('icao_code', 'location_type'), verbose=True)
Built index for fields ('icao_code', 'location_type')
```

Do not force.

```
>>> geo_o.addIndex('iata_code', force=False, verbose=True)
/!\ Index on ('iata_code',) already built, exiting...
```

**biasFuzzyCache** (*fuzzy_value*, *field*, *max_results=None*, *min_match=0.75*, *from_keys=None*, *biased_result=()*)

If algorithms for fuzzy searches are failing on a single example, it is possible to use a first cache which will block the research and force the result.

**Parameters**

- **fuzzy_value** – the value, like `'Marseille'`
- **field** – the field we look into, like `'name'`
- **max_results** – if `None`, returns all, if an int, only returns the first ones
- **min_match** – filter out matches under this threshold
- **from_keys** – if `None`, it takes all keys into consideration, else takes `from_keys` iterable of keys as search domain
- **biased_result** – the expected result

**Returns** `None`

```
>>> geo_t.fuzzyFindCached('Marseille Saint Ch.', 'name')[0]
(0.8..., 'frmsc')
>>> geo_t.biasFuzzyCache('Marseille Saint Ch.',
...                      field='name',
...                      biased_result=[(1.0, 'Me!')])
>>> geo_t.fuzzyFindCached('Marseille Saint Ch.', 'name')[0]
(1.0, 'Me!')
```

**clearFuzzyBiasCache**()

Clear biasing cache for fuzzy searches.

```
>>> geo_t.clearFuzzyBiasCache()
```

**clearFuzzyCache**()

Clear cache for fuzzy searches.

```
>>> geo_t.clearFuzzyCache()
```

**delete**(*key*, *field=None*)
Method to manually remove a value in the base.

**Parameters** **key** – the key we want to delete

**Returns** None

```
>>> data = geo_t.get('frxrn') # Output all data in one dict
>>> geo_t.delete('frxrn')
>>> geo_t.get('frxrn', 'name')
Traceback (most recent call last):
KeyError: 'Thing not found: frxrn'
```

How to reverse the delete if data has been stored:

```
>>> geo_t.set('frxrn', **data)
>>> geo_t.get('frxrn', 'name')
'Redon'
```

We can delete just a field.

```
>>> geo_t.delete('frxrn', 'lat')
>>> geo_t.get('frxrn', 'lat')
Traceback (most recent call last):
KeyError: "Field 'lat' [for key 'frxrn'] not in ...
>>> geo_t.get('frxrn', 'name')
'Redon'
```

And put it back again.

```
>>> geo_t.set('frxrn', lat='47.65179')
>>> geo_t.get('frxrn', 'lat')
'47.65179'
```

**distance**(*key0*, *key1*)
Compute distance between two elements.

This is just a wrapper between the original haversine function, but it is probably one of the most used feature :)

**Parameters**

- **key0** – the first key
- **key1** – the second key

**Returns** the distance (km)

```
>>> geo_t.distance('frnic', 'frpaz')
683.526...
```

**dropGrid**(*verbose=True*)
Delete grid.

**Returns** None

```
>>> geo_t.dropGrid()
>>> geo_t.hasGrid()
False
```

Attempt to use the grid, failure.

```
>>> sorted(geo_t.findNearKey('frbve', grid=False))[0:3]
[(0.0, 'frbve'), (7.63..., 'fr2698'), (9.07..., 'fr3065')]
>>> sorted(geo_t.findNearKey('frbve'))[0:3]
Traceback (most recent call last):
ValueError: Attempting to use grid, but grid is None
```

Adding the grid again.

```
>>> geo_t.addGrid(radius=50, verbose=True)
>>> sorted(geo_t.findNearKey('frbve'))[0:3]
[(0.0, 'frbve'), (7.63..., 'fr2698'), (9.07..., 'fr3065')]
```

**dropIndex** (*fields=None*, *verbose=True*)

Drop an index on an iterable of fields.

If fields is not given all indices are dropped.

**Parameters** **fields** – the iterable of fields, if `None`, all indices will be dropped

```
>>> geo_o.hasIndex(('icao_code', 'location_type'))
True
>>> geo_o.dropIndex(('icao_code', 'location_type'))
>>> geo_o.hasIndex(('icao_code', 'location_type'))
False
```

**findClosestFromKey** (*key*, *N=1*, *from_keys=None*, *grid=True*, *double_check=True*)

Same as `findClosestFromPoint`, except the point is given not by a `(lat, lng)`, but with its key, like `'ORY'` or `'SFO'`. We just look up in the base to retrieve latitude and longitude, then call `findClosestFromPoint`.

**Parameters**

- **key** – the key of the element (like `'SFO'`)
- **N** – the N closest results wanted
- **from_keys** – if `None`, it takes all keys in consideration, else takes `from_keys` iterable of keys to perform `findClosestFromKey`. This is useful when we have names and have to perform a matching based on name and location (see `fuzzyFindNearPoint`).
- **grid** – boolean, use grid or not
- **double_check** – when using grid, perform an additional check on results distance, this is useful because the grid is approximate, so the results are only as accurate as the grid size

**Returns** an iterable of `(distance, key)` like `[(3.2, 'SFO'), (4.5, 'LAX')]`

```
>>> list(geo_a.findClosestFromKey('ORY')) # Orly
[(0.0, 'ORY')]
>>> list(geo_a.findClosestFromKey('ORY', N=3))
[(0.0, 'ORY'), (18.80..., 'TNF'), (27.80..., 'LBG')]
>>> # Corner case, from_keys empty is not used
>>> list(geo_t.findClosestFromKey('ORY', N=2, from_keys=()))
[]
>>> list(geo_t.findClosestFromKey(None, N=2))
[]
```

No grid.

```
>>> list(geo_o.findClosestFromKey('ORY', grid=False))
[(0.0, 'ORY')]
>>> list(geo_a.findClosestFromKey('ORY', N=3, grid=False))
[(0.0, 'ORY'), (18.80..., 'TNF'), (27.80..., 'LBG')]
>>> list(geo_t.findClosestFromKey('frnic', N=1, grid=False))
[(0.0, 'frnic')]
```

Custom keys as search domain.

```
>>> keys = ('frpaz', 'frply', 'frbve')
>>> list(geo_t.findClosestFromKey('frnic',
...                               N=2,
...                               grid=False,
...                               from_keys=keys))
[(482.79..., 'frbve'), (683.52..., 'frpaz')]
```

**findClosestFromPoint**(*lat_lng*, *N=1*, *from_keys=None*, *grid=True*, *double_check=True*)

Concept close to `findNearPoint`, but here we do not look for the things radius-close to a point, we look for the closest thing from this point, given by latitude/longitude.

**Parameters**

- **lat_lng** – the lat_lng of the point (a tuple `(lat, lng)`)
- **N** – the N closest results wanted
- **from_keys** – if `None`, it takes all keys in consideration, else takes `from_keys` iterable of keys to perform `findClosestFromPoint`. This is useful when we have names and have to perform a matching based on name and location (see `fuzzyFindNearPoint`).
- **grid** – boolean, use grid or not
- **double_check** – when using grid, perform an additional check on results distance, this is useful because the grid is approximate, so the results are only as accurate as the grid size

**Returns** an iterable of `(distance, key)` like `[(3.2, 'SFO'), (4.5, 'LAX')]`

```
>>> point = (43.70, 7.26) # Nice
>>> list(geo_a.findClosestFromPoint(point))
[(5.82..., 'NCE')]
>>> list(geo_a.findClosestFromPoint(point, N=3))
[(5.82..., 'NCE'), (30.28..., 'CEQ'), (79.71..., 'ALL')]
>>> list(geo_t.findClosestFromPoint(point, N=1))
[(0.56..., 'frnic')]
>>> # Corner case, from_keys empty is not used
>>> list(geo_t.findClosestFromPoint(point, N=2, from_keys=()))
[]
>>> list(geo_t.findClosestFromPoint(None, N=2))
[]
```

No grid.

```
>>> list(geo_o.findClosestFromPoint(point, grid=False))
[(0.60..., 'NCE@1')]
>>> list(geo_a.findClosestFromPoint(point, grid=False))
[(5.82..., 'NCE')]
>>> list(geo_a.findClosestFromPoint(point, N=3, grid=False))
[(5.82..., 'NCE'), (30.28..., 'CEQ'), (79.71..., 'ALL')]
```

```
>>> list(geo_t.findClosestFromPoint(point, N=1, grid=False))
[(0.56..., 'frnic')]
```

Custom keys as search domain.

```
>>> keys = ('frpaz', 'frply', 'frbve')
>>> list(geo_t.findClosestFromPoint(point,
...                                 N=2,
...                                 grid=False,
...                                 from_keys=keys))
[(482.84..., 'frbve'), (683.89..., 'frpaz')]
```

**findNearKey**(*key*, *radius=50*, *from_keys=None*, *grid=True*, *double_check=True*)

Same as `findNearPoint`, except the point is given not by a `(lat, lng)`, but with its key, like `'ORY'` or `'SFO'`. We just look up in the base to retrieve latitude and longitude, then call `findNearPoint`.

**Parameters**

- **key** – the key of the element (like `'SFO'`)
- **radius** – the radius of the search (kilometers)
- **from_keys** – if `None`, it takes all keys in consideration, else takes `from_keys` iterable of keys to perform search.
- **grid** – boolean, use grid or not
- **double_check** – when using grid, perform an additional check on results distance, this is useful because the grid is approximate, so the results are only as accurate as the grid size

**Returns** an iterable of `(distance, key)` like `[(3.2, 'SFO'), (4.5, 'LAX')]`

```
>>> sorted(geo_o.findNearKey('ORY', 10)) # Orly, por <= 10km
[(0.0, 'ORY'), (6.94..., 'XJY'), (9.96..., 'QFC')]
>>> sorted(geo_a.findNearKey('ORY', 50)) # Orly, airports <= 50km
[(0.0, 'ORY'), (18.8..., 'TNF'), (27.8..., 'LBG'), (34.8..., 'CDG')]
>>> sorted(geo_t.findNearKey('frnic', 3)) # Nice station, stations <= 3km
[(0.0, 'frnic'), (2.2..., 'fr4342'), (2.3..., 'fr5737')]
```

No grid.

```
>>> # Orly, airports <= 50km
>>> sorted(geo_a.findNearKey('ORY', 50, grid=False))
[(0.0, 'ORY'), (18.8..., 'TNF'), (27.8..., 'LBG'), (34.8..., 'CDG')]
>>>
>>> # Nice station, stations <= 3km
>>> sorted(geo_t.findNearKey('frnic', 3, grid=False))
[(0.0, 'frnic'), (2.2..., 'fr4342'), (2.3..., 'fr5737')]
>>>
>>> keys = ['ORY', 'CDG', 'SFO']
>>> sorted(geo_a.findNearKey('ORY', 50, grid=False, from_keys=keys))
[(0.0, 'ORY'), (34.8..., 'CDG')]
```

**findNearPoint**(*lat_lng*, *radius=50*, *from_keys=None*, *grid=True*, *double_check=True*)

Returns a list of nearby things from a point (given latidude and longitude), and a radius for the search. Note that the haversine function, which compute distance at the surface of a sphere, here returns kilometers, so the radius should be in kms.

**Parameters**

- **lat_lng** – the lat_lng of the point (a tuple `(lat, lng)`)
- **radius** – the radius of the search (kilometers)
- **from_keys** – if `None`, it takes all keys in consideration, else takes `from_keys` iterable of keys to perform search.
- **grid** – boolean, use grid or not
- **double_check** – when using grid, perform an additional check on results distance, this is useful because the grid is approximate, so the results are only as accurate as the grid size

**Returns** an iterable of `(distance, key)` like `[(3.2, 'SFO'), (4.5, 'LAX')]`

```
>>> # Paris, airports <= 20km
>>> [geo_a.get(k, 'name') for d, k in
...  sorted(geo_a.findNearPoint((48.84, 2.367), 20))]
['Paris-Orly', 'Paris-Le Bourget']
>>>
>>> # Nice, stations <= 3km
>>> [geo_t.get(k, 'name') for d, k in
...  sorted(geo_t.findNearPoint((43.70, 7.26), 3))]
['Nice-Ville', 'Nice-Riquier', 'Nice-St-Roch']
>>>
>>> # Wrong geocode
>>> sorted(geo_t.findNearPoint(None, 5))
[]
```

No grid mode.

```
>>> # Paris, airports <= 20km
>>> [geo_a.get(k, 'name') for d, k in
...  sorted(geo_a.findNearPoint((48.84, 2.367), 20, grid=False))]
['Paris-Orly', 'Paris-Le Bourget']
>>>
>>> # Nice, stations <= 3km
>>> [geo_t.get(k, 'name') for d, k in
...  sorted(geo_t.findNearPoint((43.70, 7.26), 3, grid=False))]
['Nice-Ville', 'Nice-Riquier', 'Nice-St-Roch']
>>>
>>> # Paris, airports <= 50km with from_keys input list
>>> sorted(geo_a.findNearPoint((48.84, 2.367), 50,
...                            from_keys=['ORY', 'CDG', 'BVE'],
...                            grid=False))
[(12.76..., 'ORY'), (23.40..., 'CDG')]
```

**findWith**(*conditions*, *from_keys=None*, *reverse=False*, *mode='and'*, *index=True*, *verbose=False*)

Get iterator of all keys with particular field.

For example, if you want to know all airports in Paris.

**Parameters**

- **conditions** – a list of `('field', 'value')` conditions
- **reverse** – we look keys where the field is *not* the particular value. Note that this negation is done at the lower level, before combining conditions. So if you have two conditions with `mode='and'`, expect results matching not condition 1 *and* not condition 2.
- **mode** – either `'or'` or `'and'`, how to handle several conditions

- **from_keys** – if given, we will look for results from this iterable of keys
- **index** – boolean to disable index when searching
- **verbose** – toggle verbosity during search

**Returns** an iterable of `(v, key)` where `v` is the number of matched conditions

```
>>> list(geo_a.findWith([('city_code', 'PAR')]))
[(1, 'ORY'), (1, 'TNF'), (1, 'CDG'), (1, 'BVA')]
>>> len(list(geo_o.findWith([('comment', '')], reverse=True)))
212
>>> len(list(geo_o.findWith([('__dup__', [])])))
6264
>>> # Counting duplicated keys
>>> len(list(geo_o.findWith([('__par__', [])], reverse=True)))
5377
```

Testing indices.

```
>>> list(geo_o.findWith([('iata_code', 'MRS')], mode='and', verbose=True))
["and" mode] Using index for ('iata_code',): value(s) ('MRS',)
[(1, 'MRS'), (1, 'MRS@1')]
>>> geo_o.addIndex('iata_code', force=True)
/!\ Index on ('iata_code',) already built, overriding...
Built index for fields ('iata_code',)
>>> geo_o.addIndex('location_type')
Built index for fields ('location_type',)
```

Now querying with simple indices (dropping multiple index if it exists).

```
>>> geo_o.dropIndex(('iata_code', 'location_type'), verbose=False)
>>> list(geo_o.findWith([('iata_code', 'NCE'), ('location_type', ('A',))],
...                     mode='and',
...                     verbose=True))
["and" mode] Using index for ('iata_code',) and ('location_type',): value(s) (
↪'NCE',); (('A',),)
[(2, 'NCE')]
```

Multiple index.

```
>>> geo_o.addIndex(('iata_code', 'location_type'), verbose=False)
>>> list(geo_o.findWith([('iata_code', 'NCE'), ('location_type', ('A',))],
...                     mode='and',
...                     verbose=True))
["and" mode] Using index for ('iata_code', 'location_type'): value(s) ('NCE',␣
↪('A',))
[(2, 'NCE')]
```

Mode “or” with index.

```
>>> geo_o.addIndex('city_code_list')
Built index for fields ('city_code_list',)
>>> list(geo_o.findWith([('iata_code', 'NCE'), ('city_code_list', ('NCE',))],
...                     mode='or',
...                     verbose=True))
["or" mode] Using index for ('iata_code',) and ('city_code_list',): value(s) (
↪'NCE',); (('NCE',),)
[(2, 'NCE@1'), (2, 'NCE')]
>>> list(geo_o.findWith([('iata_code', 'NCE'), ('city_code_list', ('NCE',))],
```

```
...                       mode='or',
...                       index=False,
...                       verbose=True))
[(2, 'NCE'), (2, 'NCE@1')]
```

Testing several conditions.

```
>>> c_1 = [('city_code_list', ('PAR',))]
>>> c_2 = [('location_type', ('H',))]
>>> len(list(geo_o.findWith(c_1)))
17
>>> len(list(geo_o.findWith(c_2)))
100
>>> len(list(geo_o.findWith(c_1 + c_2, mode='and')))
2
>>> len(list(geo_o.findWith(c_1 + c_2, mode='or')))
111
```

**static `fuzzyClean`**(*value*)

Cleaning from LevenshteinUtils.

```
>>> GeoBase.fuzzyClean('antibes ville 2')
'antibes'
```

**`fuzzyFind`**(*fuzzy_value*, *field*, *max_results=None*, *min_match=0.75*, *from_keys=None*)

Fuzzy searches are retrieving an information on a thing when we do not know the code. We compare the value `fuzzy_value` which is supposed to be a field (e.g. a city or a name), to all things we have in the base, and we output the best match. Matching is performed using Levenshtein module, with a modified version of the Lenvenshtein ratio, adapted to the type of data.

Example: we look up 'Marseille Saint Ch.' in our base and we find the corresponding code by comparing all station names with ''Marseille Saint Ch.''.

**Parameters**

- **`fuzzy_value`** – the value, like `'Marseille'`
- **`field`** – the field we look into, like `'name'`
- **`max_results`** – max number of results, None means all results
- **`min_match`** – filter out matches under this threshold
- **`from_keys`** – if `None`, it takes all keys in consideration, else takes `from_keys` iterable of keys to perform `fuzzyFind`. This is useful when we have geocodes and have to perform a matching based on name and location (see `fuzzyFindNearPoint`).

**Returns** an iterable of `(distance, key)` like `[(0.97, 'SFO'), (0.55, 'LAX')]`

```
>>> geo_t.fuzzyFind('Marseille Charles', 'name')[0]
(0.8..., 'frmsc')
>>> geo_a.fuzzyFind('paris de gaulle', 'name')[0]
(0.78..., 'CDG')
>>> geo_a.fuzzyFind('paris de gaulle',
...                 field='name',
...                 max_results=3,
...                 min_match=0.55)
[(0.78..., 'CDG'), (0.60..., 'HUX'), (0.57..., 'LBG')]
```

Some corner cases.

```
>>> geo_a.fuzzyFind('paris de gaulle', 'name', max_results=None)[0]
(0.78..., 'CDG')
>>> geo_a.fuzzyFind('paris de gaulle', 'name',
...                 max_results=1, from_keys=[])
[]
```

**fuzzyFindCached**(*fuzzy_value*, *field*, *max_results=None*, *min_match=0.75*, *from_keys=None*, *verbose=False*, *d_range=None*)

Same as `fuzzyFind` but with a caching and bias system.

**Parameters**

- **fuzzy_value** – the value, like `'Marseille'`
- **field** – the field we look into, like `'name'`
- **max_results** – max number of results, None means all results
- **min_match** – filter out matches under this threshold
- **from_keys** – if `None`, it takes all keys in consideration, else takes `from_keys` iterable of keys to perform fuzzyFind. This is useful when we have geocodes and have to perform a matching based on name and location (see `fuzzyFindNearPoint`).
- **verbose** – display information on caching for a certain range of similarity
- **d_range** – the range of similarity

**Returns** an iterable of `(distance, key)` like `[(0.97, 'SFO'), (0.55, 'LAX')]`

```
>>> geo_t.fuzzyFindCached('Marseille Saint Ch.', 'name')[0]
(0.8..., 'frmsc')
>>> geo_a.fuzzyFindCached('paris de gaulle',
...                       field='name',
...                       verbose=True,
...                       d_range=(0, 1))[0]
[0.79]           paris+de+gaulle ->   paris+charles+de+gaulle (  CDG)
(0.78..., 'CDG')
>>> geo_a.fuzzyFindCached('paris de gaulle',
...                       field='name',
...                       min_match=0.60,
...                       max_results=2,
...                       verbose=True,
...                       d_range=(0, 1))
[0.79]           paris+de+gaulle ->   paris+charles+de+gaulle (  CDG)
[0.61]           paris+de+gaulle ->        bahias+de+huatulco (  HUX)
[(0.78..., 'CDG'), (0.60..., 'HUX')]
```

Some biasing:

```
>>> geo_a.biasFuzzyCache('paris de gaulle',
...                      field='name',
...                      biased_result=[(0.5, 'Biased result')])
>>> geo_a.fuzzyFindCached('paris de gaulle',
...                       field='name',
...                       max_results=None,
...                       verbose=True,
...                       d_range=(0, 1))
Using bias: ('paris+de+gaulle', 'name', None, 0.75, None)
[(0.5, 'Biased result')]
>>> geo_a.clearFuzzyBiasCache()
```

```
>>> geo_a.fuzzyFindCached('paris de gaulle',
...                       field='name',
...                       max_results=None,
...                       verbose=True,
...                       min_match=0.75)
[(0.78..., 'CDG')]
```

**fuzzyFindNearPoint**(*lat_lng*, *radius*, *fuzzy_value*, *field*, *max_results=None*, *min_match=0.75*, *from_keys=None*, *grid=True*, *double_check=True*)

Same as `fuzzyFind` but with we search only within a radius from a geocode.

**Parameters**

- **lat_lng** – the lat_lng of the point (a tuple `(lat, lng)`)
- **radius** – the radius of the search (kilometers)
- **fuzzy_value** – the value, like `'Marseille'`
- **field** – the field we look into, like `'name'`
- **max_results** – if `None`, returns all, if an int, only returns the first ones
- **min_match** – filter out matches under this threshold
- **from_keys** – if `None`, it takes all keys in consideration, else takes a from_keys iterable of keys to perform search.
- **grid** – boolean, use grid or not
- **double_check** – when using grid, perform an additional check on results distance, this is useful because the grid is approximate, so the results are only as accurate as the grid size

**Returns** an iterable of `(distance, key)` like `[(0.97, 'SFO'), (0.55, 'LAX')]`

```
>>> geo_a.fuzzyFind('Brussels', 'name', min_match=0.60)[0]
(0.61..., 'BQT')
>>> geo_a.get('BQT', 'name')  # Brussels just matched on Brest!!
'Brest'
>>> geo_a.get('BRU', 'name') # We wanted BRU for 'Bruxelles'
'Bruxelles National'
>>>
>>> # Now a request limited to a circle of 20km around BRU gives BRU
>>> point = (50.9013, 4.4844)
>>> geo_a.fuzzyFindNearPoint(point,
...                          radius=20,
...                          fuzzy_value='Brussels',
...                          field='name',
...                          min_match=0.40)[0]
(0.46..., 'BRU')
>>>
>>> # Now a request limited to some input keys
>>> geo_a.fuzzyFindNearPoint(point,
...                          radius=2000,
...                          fuzzy_value='Brussels',
...                          field='name',
...                          max_results=1,
...                          min_match=0.30,
...                          from_keys=['ORY', 'CDG'])
[(0.33..., 'ORY')]
```

**get** (*key*, *field=None*, ***kwargs*)
Simple get on the base.

Get data on `key` for `field` information. For example you can get data on `CDG` for its `city_code_list`. You can use the `None` as `field` value to get all information in a dictionary. You can give an additional keyword argument `default`, to avoid `KeyError` on the `key` parameter.

**Parameters**

- **key** – the key of the element (like `'SFO'`)
- **field** – the field (like `'name'` or `'iata_code'`)
- **kwargs** – other named arguments, use 'default' to avoid `KeyError` on `key` (not `KeyError` on `field`). Use 'ext_field' to field data from join base.

**Raises** `KeyError` if the key is not in the base

**Returns** the needed information

```
>>> geo_a.get('CDG', 'city_code')
'PAR'
>>> geo_t.get('frnic', 'name')
'Nice-Ville'
>>> geo_t.get('frnic')
{'info': 'Desserte Voyageur-Infrastructure', 'code': 'frnic', ...}
```

Cases of unknown key.

```
>>> geo_t.get('frmoron', 'name', default='There')
'There'
>>> geo_t.get('frmoron', 'name')
Traceback (most recent call last):
KeyError: 'Thing not found: frmoron'
>>> geo_t.get('frmoron', 'name', default=None)
>>> geo_t.get('frmoron', default='There')
'There'
```

Cases of unknown field, this is a bug and always fail.

```
>>> geo_t.get('frnic', 'not_a_field', default='There')
Traceback (most recent call last):
KeyError: "Field 'not_a_field' [for key 'frnic'] not in ['__dup__', ...
```

**getFromAllDuplicates** (*key*, *field=None*, ***kwargs*)
Get all duplicates data, parent key included.

**Parameters**

- **key** – the key of the element (like `'SFO'`)
- **field** – the field (like `'name'` or `'iata_code'`)
- **kwargs** – other named arguments, use 'default' to avoid key failure

**Returns** the list of values for the given field iterated on all duplicates for the key, including the key itself

```
>>> for n in geo_o.getFromAllDuplicates('ORY', 'name'):
...     print(n)
Paris Orly Airport
```

```
>>> geo_o.getFromAllDuplicates('THA', 'name')
['Tullahoma Regional Airport/William Northern Field', 'Tullahoma']
```

One parent, one duplicate example.

```
>>> geo_o.get('THA@1', '__par__')
['THA']
>>> geo_o.get('THA', '__dup__')
['THA@1']
```

Use getFromAllDuplicates on master or duplicates gives the same results.

```
>>> geo_o.getFromAllDuplicates('THA', '__key__')
['THA', 'THA@1']
>>> geo_o.getFromAllDuplicates('THA@1', '__key__')
['THA@1', 'THA']
```

Corner cases are handled in the same way as `get` method.

```
>>> geo_o.getFromAllDuplicates('nnnnnnoooo', default='that')
'that'
>>> it = geo_o.getFromAllDuplicates('THA', field=None)
>>> [e['__key__'] for e in it]
['THA', 'THA@1']
```

**getJoinBase** (*fields*, *verbose=True*)

Get joined base from the fields who have join.

**Parameters**

- **fields** – the iterable of fields
- **verbose** – boolean, toggle verbosity

**Returns** a GeoBase object or `None` if fields are not joined

```
>>> geo_o.getJoinBase('iata_code')
Fields "('iata_code',)" do not have join, cannot retrieve external base.
>>> geo_o.getJoinBase('country_code')
<GeoBases.GeoBaseModule.GeoBase object at 0x...>
```

**getLocation** (*key*, ***kwargs*)

Returns geocode as (float, float) or None.

**Parameters**

- **key** – the key of the element (like `'SFO'`)
- **kwargs** – other named arguments, use 'default' to avoid `KeyError` on `key` (not `None` on wrong value).

**Returns** the location, a tuple of floats like `(lat, lng)`, or `None` if any problem happened during execution

```
>>> geo_o.getLocation('AGN')
(57.5..., -134...)
```

Behavior on unkwown key.

```
>>> geo_o.getLocation('UNKNOWN')
Traceback (most recent call last):
KeyError: 'Thing not found: UNKNOWN'
>>> geo_o.getLocation('UNKNOWN', default=(0, 0))
(0, 0)
```

**hasDuplicates** (*key*)

Tell if a key has duplicates.

**Parameters** **key** – the key of the element (like `'SFO'`)

**Returns** the number of duplicates

```
>>> geo_o.hasDuplicates('MRS')
1
>>> geo_o.hasDuplicates('MRS@1')
1
>>> geo_o.hasDuplicates('PAR')
0
```

**hasGeoSupport** (*key=None*)

Check if data type has geocoding support.

If a key parameter is given, check the geocode support of this specific key.

**Parameters** **key** – if key parameter is not `None`, we check the geocode support for this specific key, not for the general data with `fields` attribute

**Returns** boolean for geocoding support

```
>>> geo_t.hasGeoSupport()
True
>>> geo_f.hasGeoSupport()
False
```

For a specific key.

```
>>> geo_o.hasGeoSupport('ORY')
True
>>> geo_o.set('EMPTY')
>>> geo_o.hasGeoSupport('EMPTY')
False
>>> geo_o.delete('EMPTY') # avoid messing other tests
```

**hasGrid**()

Tells if an iterable of fields is indexed.

**Parameters** **fields** – the iterable of fields

**Returns** a boolean

```
>>> geo_t.hasGrid()
True
>>> geo_t.dropGrid()
>>> geo_t.hasGrid()
False
>>> geo_t.addGrid()
```

**hasIndex** (*fields=None*)

Tells if an iterable of fields is indexed.

Default value is `None` for fields, this will test the presence of any index.

**Parameters** **`fields`** – the iterable of fields

**Returns** a boolean

```
>>> geo_o.hasIndex('iata_code')
True
>>> geo_o.hasIndex(('iata_code', 'asciiname'))
False
>>> geo_o.hasIndex()
True
```

**`hasJoin`**(*fields=None*)

Tells if an iterable of fields has join information.

Default value is `None` for fields, this will test the presence of any join information.

**Parameters** **`fields`** – the iterable of fields

**Returns** a boolean

```
>>> geo_o.hasJoin('iata_code')
False
>>> geo_o.hasJoin('tvl_por_list')
True
>>> geo_o.hasJoin()
True
```

**`hasParents`**(*key*)

Tell if a key has parents.

**Parameters** **`key`** – the key of the element (like `'SFO'`)

**Returns** the number of parents

```
>>> geo_o.hasParents('MRS')
0
>>> geo_o.hasParents('MRS@1')
1
>>> geo_o.hasParents('PAR')
0
```

static **`hasTrepSupport`**()

Check if module has OpenTrep support.

**`keys`**()

Returns a list of all keys in the base.

**Returns** the list of all keys

```
>>> geo_a.keys()
['AGN', 'AGM', 'AGJ', 'AGH', ...
```

static **`phonemes`**(*value*, *method='dmetaphone'*)

Compute phonemes for any value.

**Parameters**

- **`value`** – the input value
- **`method`** – change the phonetic method used

**Returns** the phonemes

```
>>> GeoBase.phonemes('sheekago')
['XKK', None]
>>> GeoBase.phonemes('sheekago', 'nysiis')
'SACAG'
```

**phoneticFind**(*value*, *field*, *method='dmetaphone'*, *from_keys=None*, *verbose=False*)

Phonetic search.

**Parameters**

- **value** – the value for which we look for a match
- **field** – the field, like `'name'`
- **method** – change the phonetic method used
- **from_keys** – if `None`, it takes all keys in consideration, else takes `from_keys` iterable of keys to perform search.
- **verbose** – toggle verbosity

**Returns** an iterable of (phonemes, key) matching

```
>>> list(geo_o.get(k, 'name') for _, k in
...      geo_o.phoneticFind(value='chicago',
...                         field='name',
...                         method='dmetaphone',
...                         verbose=True))
Looking for phonemes like ['XKK', None] (for "chicago")
['Chicago']
>>> list(geo_o.get(k, 'name') for _, k in
...      geo_o.phoneticFind('chicago', 'name', 'nysiis'))
['Chicago']
```

Alternate methods.

```
>>> list(geo_o.phoneticFind('chicago', 'name', 'dmetaphone'))
[(['XKK', None], 'CHI')]
>>> list(geo_o.phoneticFind('chicago', 'name', 'metaphone'))
[('XKK', 'CHI')]
>>> list(geo_o.phoneticFind('chicago', 'name', 'nysiis'))
[('CACAG', 'CHI')]
```

**save**(*path=None*, *safe=False*, *headers=None*, *verbose=True*)

Save the data structure in the initial loaded file.

**Parameters**

- **path** – `None` as default. If no argument is given for this parameter, we will try to save to the default path defined in the configuration file. Otherwise we will try to save in the path given.
- **safe** – default is `False`. If `safe` is `False`, the data is dumped in the initial loaded file. If `True`, a `filename.new` will be created to dump the data.
- **headers** – the headers of data which will be dumped. Leave default to use headers defined in configuration. Otherwise, this must be a list of fields.
- **verbose** – toggle verbosity

**Returns** `None`

**set** (*key*, ***kwargs*)
Method to manually change a value in the base.

**Parameters**

- **key** – the key we want to change a value of
- **kwargs** – the keyword arguments containing new data

**Returns** None

Here are a few examples.

```
>>> geo_t.get('frnic', 'name')
'Nice-Ville'
>>> geo_t.set('frnic', name='Nice Gare SNCF')
>>> geo_t.get('frnic', 'name')
'Nice Gare SNCF'
>>> geo_t.set('frnic', name='Nice-Ville') # tearDown
```

We may even add new fields.

```
>>> geo_t.set('frnic', new_field='some_value')
>>> geo_t.get('frnic', 'new_field')
'some_value'
```

We can create just the key.

```
>>> geo_t.set('NEW_KEY_1')
>>> geo_t.get('NEW_KEY_1')
{'__gar__': [], ..., '__lno__': 0, '__key__': 'NEW_KEY_1'}
>>> geo_t.delete('NEW_KEY_1') # tearDown
```

Examples with an empty base.

```
>>> geo_f.keys()
[]
```

Set a new key with a dict, then get the data back.

```
>>> d = {
...     'code' : 'frnic',
...     'name' : 'Nice',
... }
>>> geo_f.set('frnic', **d)
>>> geo_f.keys()
['frnic']
>>> geo_f.get('frnic', 'name')
'Nice'
```

The base fields are *not* automatically updated when setting data.

```
>>> geo_f.fields
[]
```

You can manually update the fields.

```
>>> geo_f.syncFields()
>>> geo_f.fields
['__dup__', '__gar__', '__key__', '__lno__', '__par__', 'code', 'name']
```

**syncFields**(*mode='all'*, *sort=True*)

Iterate through the collection to look for all available fields. Then affect the result to `self.fields`.

If you execute this method, be aware that fields order may change depending on how dictionaries return their keys. To have better consistency, we automatically sort the found fields. You can change this behavior with the `sort` parameter.

**Parameters**

- **mode** – `'all'` or `'any'`, `'all'` will look for fields shared by all keys, `'any'` will look for all fields from all keys
- **sort** – sort the fields found

**Returns** `None`

```
>>> from pprint import pprint
>>> pprint(geo_t.fields)
['__key__',
 '__dup__',
 '__par__',
 '__lno__',
 'code',
 'lines@raw',
 'lines',
 'name',
 'info',
 'lat',
 'lng',
 '__gar__']
```

Fields synchronisation, common fields for all keys.

```
>>> geo_t.set('frnic', new_field='Nice Gare SNCF')
>>> geo_t.syncFields(mode='all')
>>> pprint(geo_t.fields) # did not change, except order
['__dup__',
 '__gar__',
 '__key__',
 '__lno__',
 '__par__',
 'code',
 'info',
 'lat',
 'lines',
 'lines@raw',
 'lng',
 'name']
```

Fields synchronisation, all fields for all keys.

```
>>> geo_t.syncFields(mode='any')
>>> pprint(geo_t.fields) # notice the new field 'new_field'
['__dup__',
 '__gar__',
 '__key__',
 '__lno__',
 '__par__',
 'code',
 'info',
```

```
 'lat',
 'lines',
 'lines@raw',
 'lng',
 'name',
 'new_field']
```

Restore previous state, drop new field and synchronize fields again.

```
>>> geo_t.delete('frnic', 'new_field')
>>> geo_t.syncFields()
>>> pprint(geo_t.fields)
['__dup__',
 '__gar__',
 '__key__',
 '__lno__',
 '__par__',
 'code',
 'info',
 'lat',
 'lines',
 'lines@raw',
 'lng',
 'name']
```

**static** **trepSearch**(*fuzzy_value*, *trep_format='S'*, *from_keys=None*, *verbose=False*)

OpenTrep integration.

If not hasTrepSupport(), main_trep is not defined and trepSearch will raise an exception if called.

**Parameters**

- **fuzzy_value** – the fuzzy value
- **trep_format** – the format given to OpenTrep
- **from_keys** – if `None`, it takes all keys in consideration, else takes `from_keys` iterable of keys to perform search.
- **verbose** – toggle verbosity

**Returns** an iterable of `(distance, key)` like `[(0.97, 'SFO'), (0.55, 'LAX')]`

```
>>> if GeoBase.hasTrepSupport():
...     print geo_t.trepSearch('sna francisco los agneles')
[(31.5192, 'SFO'), (46.284, 'LAX')]
```

```
>>> if GeoBase.hasTrepSupport():
...     print geo_t.trepSearch('sna francisco', verbose=True)
 -> Raw result: SFO/31.5192
 -> Fmt result: ([(31.5192, 'SFO')], '')
[(31.5192, 'SFO')]
```

**updateGrid**(*verbose=True*)

Update the grid for geographical indexation.

**Parameters**

- **radius** – the grid accuracy, in kilometers the `precision` parameter is used to define grid size

- **precision** – the hash length. This is only used if `radius` is `None`, otherwise this parameter (a hash length) is computed from the radius
- **verbose** – toggle verbosity

**Returns** `None`

We use the grid for a query.

```
>>> sorted(geo_t.findNearKey('frbve'))[0:3]
[(0.0, 'frbve'), (7.63..., 'fr2698'), (9.07..., 'fr3065')]
```

Now we add a new key to the data.

```
>>> geo_t.set('NEW_KEY_3', **{
...     'lat' : '45.152',
...     'lng' : '1.528',
... })
```

If we run the query again, the result is wrong when using the grid, because it is not up-to-date.

```
>>> sorted(geo_t.findNearKey('frbve'))[0:3]
[(0.0, 'frbve'), (7.63..., 'fr2698'), (9.07..., 'fr3065')]
>>> sorted(geo_t.findNearKey('frbve', grid=False))[0:3]
[(0.0, 'frbve'), (0.07..., 'NEW_KEY_3'), (7.63..., 'fr2698')]
```

Now we update the grid, then the query works.

```
>>> geo_t.updateGrid()
>>> sorted(geo_t.findNearKey('frbve'))[0:3]
[(0.0, 'frbve'), (0.07..., 'NEW_KEY_3'), (7.63..., 'fr2698')]
>>> geo_t.delete('NEW_KEY_3') # avoid messing other tests
```

Note that `updateGrid` will not create the grid if it does not exist.

```
>>> geo_f.updateGrid()
No grid to update.
```

**updateIndex** (*fields=None*, *verbose=True*)

Update index on fields.

If fields is not given all indices are updated.

**Parameters**

- **fields** – the iterable of fields, if `None`, all indices will be updated
- **verbose** – toggle verbosity

Here is an example, we drop the index then make a query.

```
>>> geo_o.dropIndex('iata_code')
>>> list(geo_o.findWith([('iata_code', 'NCE')])) # not indexed
[(1, 'NCE'), (1, 'NCE@1')]
```

Now we index and make the same query.

```
>>> geo_o.addIndex('iata_code')
Built index for fields ('iata_code',)
>>> list(geo_o.findWith([('iata_code', 'NCE')])) # indexed
[(1, 'NCE'), (1, 'NCE@1')]
```

Now we add a new key to the data.

```
>>> geo_o.set('NEW_KEY_2', **{
...     'iata_code' : 'NCE',
... })
```

If we run the query again, the result is wrong when using the index, because it is not up-to-date.

```
>>> list(geo_o.findWith([('iata_code', 'NCE')])) # indexed
[(1, 'NCE'), (1, 'NCE@1')]
>>> list(geo_o.findWith([('iata_code', 'NCE')], index=False))
[(1, 'NCE'), (1, 'NEW_KEY_2'), (1, 'NCE@1')]
```

Now we update the index, then the query works.

```
>>> geo_o.updateIndex('iata_code')
Built index for fields ('iata_code',)
>>> list(geo_o.findWith([('iata_code', 'NCE')])) # indexed, up to date
[(1, 'NCE'), (1, 'NEW_KEY_2'), (1, 'NCE@1')]
>>> geo_o.delete('NEW_KEY_2') # avoid messing other tests
```

Note that `updateIndex` will not create indices if it does not exist.

```
>>> geo_f.updateIndex('iata_code')
No index to update on "iata_code".
```

This module defines a class *VisualMixin* which will be used by the *GeoBase* as a mixin.

**class** GeoBases.VisualMixinModule.**VisualMixin**

Bases: object

Main class used as mixin for the *GeoBase* class.

**buildDashboardData**(*keep=10*, *dashboard_weight=None*, *from_keys=None*)

Build dashboard data.

**Parameters**

- **keep** – the number of values kept after counting for each field
- **dashboard_weight** – the field used as weight for the graph. Leave `None` if you just want to count the number of keys
- **from_keys** – only use this iterable of keys if not `None`

**Returns** a dictionary of fields counters information

**buildGraphData**(*graph_fields*, *graph_weight=None*, *with_types=False*, *directed=False*, *from_keys=None*)

Build graph data.

**Parameters**

- **graph_fields** – iterable of fields used to define the nodes. Nodes are the values of these fields. Edges represent the data.
- **graph_weight** – field used to define the weight of nodes and edges. If `None`, the weight is `1` for each key.
- **with_types** – boolean to consider values from different fields of the same "type" or not, meaning we will create only one node if the same value is found accross different fields, if there are no types. Otherwise we create different nodes. Default is `False`, meaning untyped graphs.

- **directed** – boolean, if the graph is directed or not, default is `False`.
- **from_keys** – only use this iterable of keys if not `None`

**Returns** the nodes data

```
>>> nodes = g.buildGraphData(
...     graph_fields=['continent_name', 'country_code'],
...     graph_weight='page_rank'
... )
>>> edges = nodes['Antarctica']['edges'].values()
>>> sorted(edges[0].items())
[('from', 'Antarctica'), ('to', 'AQ'), ('weight', 0)]
```

**dashboardVisualize**(*output='example'*, *output_dir=None*, *keep=10*, *dashboard_weight=None*, *from_keys=None*, *verbose=True*)

Dashboard display (aggregated view).

**Parameters**

- **output** – set the name of the rendered files
- **output_dir** – set the directory of the rendered files, will be created if it does not exist
- **keep** – the number of values kept after counting for each field
- **dashboard_weight** – the field used as weight for the graph. Leave `None` if you just want to count the number of keys
- **from_keys** – only display this iterable of keys if not `None`
- **verbose** – toggle verbosity

**Returns** this is the tuple of (names of templates rendered, (list of html templates, list of static files))

**graphVisualize**(*graph_fields*, *graph_weight=None*, *with_types=False*, *from_keys=None*, *output='example'*, *output_dir=None*, *verbose=True*)

Graph display (like force directed graph).

**Parameters**

- **graph_fields** – iterable of fields used to define the nodes. Nodes are the values of these fields. Edges represent the data.
- **graph_weight** – field used to define the weight of nodes and edges. If `None`, the weight is `1` for each key.
- **with_types** – boolean to consider values from different fields of the same "type" or not, meaning we will create only one node if the same value is found accross different fields, if there are no types. Otherwise we create different nodes. Default is `False`, meaning untyped graphs.
- **from_keys** – only display this iterable of keys if not `None`
- **output** – set the name of the rendered files
- **output_dir** – set the directory of the rendered files, will be created if it does not exist
- **verbose** – toggle verbosity

**Returns** this is the tuple of (names of templates rendered, (list of html templates, list of static files))

**visualize**(*output='example'*, *output_dir=None*, *icon_label=None*, *icon_weight=None*, *icon_color=None*, *icon_type='auto'*, *from_keys=None*, *add_lines=None*, *add_anonymous_icons=None*, *add_anonymous_lines=None*, *link_duplicates=True*, *draw_join_fields=True*, *catalog=None*, *line_colors=None*, *use_3D=False*, *verbose=True*, *warnings=False*)

Map and table display.

**Parameters**

- **output** – set the name of the rendered files
- **output_dir** – set the directory of the rendered files, will be created if it does not exist
- **icon_label** – set the field which will appear as map icons title
- **icon_weight** – set the field defining the map icons circle surface
- **icon_color** – set the field defining the map icons colors
- **icon_type** – set the icon size, either `'B'`, `'S'`, `'auto'` or `None` for no-icons mode
- **from_keys** – only display this iterable of keys if not `None`
- **add_lines** – list of `(key1, key2, ..., keyN)` to draw additional lines
- **add_anonymous_icons** – list of geocodes, like `[(lat1, lng1), (lat2, lng2), ..., (latN, lngN)]`, to draw additional icons from geocodes not in the data
- **add_anonymous_icons** – list of list of geocodes, like `[[(lat1, lng1), (lat2, lng2), ..., (latN, lngN)], ...]`, to draw additional lines from geocodes not in the data
- **link_duplicates** – boolean toggling lines between duplicated keys, default `True`
- **draw_join_fields** – boolean toggling drawing of join fields containing geocode information, default `True`
- **catalog** – dictionary of `{'value': 'color'}` to have specific colors for some categories, which is computed with the `icon_color` field
- **line_colors** – tuple of 4 colors to change the default lines color, the three values are for the three line types: those computed with `link_duplicates`, those given with `add_lines`, those given with `add_anonymous_lines`, those computed with `draw_join_fields`
- **use_3D** – toggle 3D visualizations
- **verbose** – toggle verbosity
- **warnings** – toggle warnings, even more verbose

**Returns** this is the tuple of (names of templates rendered, (list of html templates, list of static files))

# Python Module Index

## g

# Index

## Symbols



## A



## B



## C



## D



## F



## G



## H

## K



## P



## S



## T



## U



## V