GeoBases Documentation

Release 4

OpenTravelData

April 01, 2013

CONTENTS

CHAPTER

ONE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

CHAPTER

TWO

GEOBASEMODULE MODULE

This module is a general class *GeoBase* to manipulate geographical data. It loads static csv files containing data about airports or train stations, and then provides tools to browse it.

It relies on three 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

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')
694.5162...
```

Examples for stations:

```
>>> geo_t = GeoBase(data='stations', verbose=False)
>>>
>>> # Nice, stations <= 5km
>>> [geo_t.get(k, 'name') for d, k in sorted(geo_t.findNearPoint((43.70, 7.26), 5))]
['Nice-Ville', 'Nice-Riquier', 'Nice-St-Roch', 'Villefranche-sur-Mer', 'Nice-St-Augustin']
>>>
>>> geo_t.get('frpaz', 'name')
'Paris-Austerlitz'
>>> geo_t.distance('frnic', 'frpaz')
683.526...
```

From any point of reference:

```
>>> geo = GeoBase(data='ori_por_multi') # we have a few duplicates even with (iata, loc_type) key
/!\ [lno ...] CRK+C is duplicated #1, first found lno ...: creation of CRK+C@1...
/!\ [lno ...] DOV+C is duplicated #1, first found lno ...: creation of DOV+C@1...
/!\ [lno ...] EAP+C is duplicated #1, first found lno ...: creation of EAP+C@1...
/!\ [lno ...] LIH+C is duplicated #1, first found lno ...: creation of LIH+C@1...
/!\ [lno ...] OSF+C is duplicated #1, first found lno ...: creation of OSF+C@1...
/!\ [lno ...] RDU+C is duplicated #1, first found lno ...: creation of RDU+C@1...
/!\ [lno ...] STX+C is duplicated #1, first found lno ...: creation of STX+C@1...
/!\ [lno ...] VAF+C is duplicated #1, first found lno ...: creation of VAF+C@1...
```

```
Import successful from ...
Available fields for things: ...
```

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

Bases: object

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:

- local : True by default, is the source local or not
- source : None by default, file-like to the source
- headers : [] by default, list of fields in the data
- indexes : None by default, list of fields defining the key for a line
- delimiter : '^' by default, delimiter for each field,
- subdelimiters : {} by default, a { 'field' : 'delimiter' } dict to define subdelimiters
- 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
- discard_dups : False by default, boolean to discard key duplicates of handle them
- verbose : True by default, toggle verbosity

Parameters

- **data** – the type of data wanted, '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')
Source was None, skipping loading...
>>> geo_c = GeoBase(data='odd')
Traceback (most recent call last):
ValueError: Wrong data type. Not in ['airlines', ...]
>>>
>>> fl = open(relative('DataSources/Airports/GeoNames/airports_geonames_only_clean.csv'))
>>> GeoBase(data='feed',
...         source=fl,
...         headers=['iata_code', 'name', 'city'],
...         indexes='iata_code',
...         delimiter='^',
...         verbose=False).get('ORY')
```

```
{'city': 'PAR', 'name': 'Paris-Orly', 'iata_code': 'ORY', '__gar__': 'FR^France^48.7252780^2
>>> fl.close()
>>> GeoBase(data='airports',
...         headers=['iata_code', 'name', 'city'],
...         verbose=False).get('ORY')
{'city': 'PAR', 'name': 'Paris-Orly', 'iata_code': 'ORY', '__gar__': 'FR^France^48.7252780^2
```

biasFuzzyCache (*fuzzy_value*, *field*, *max_results*, *min_match*, *from_keys*, *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

clearBiasCache()

Clear biasing cache for fuzzy searches.

clearCache()

Clear cache for fuzzy searches.

createGrid()

Create the grid for geographical indexation after loading the data.

delete(*key*)

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

distance(*key0*, *key1*)

Compute distance between two elements.

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

Parameters

- **key0** – the first key

- **key1** – the second key

Returns the distance (km)

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

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 lat/lng, and call findClosestFromPoint.

Parameters

- **key** – the key of the thing (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 findClosestFromPoint. This is useful when we have names and have to perform a matching based on name and location (see fuzzyGetAroundLatLng).
- **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))
[]
>>> #from datetime import datetime
>>> #before = datetime.now()
>>> #for _ in range(100): s = geo_a.findClosestFromKey('NCE', N=3)
>>> #print(datetime.now() - before)
```

No grid.

```
>>> list(geo_o.findClosestFromKey('ORY', grid=False)) # Nice
[(0.0, 'ORY')]
>>> list(geo_a.findClosestFromKey('ORY', N=3, grid=False)) # Nice
[(0.0, 'ORY'), (18.80..., 'TNF'), (27.80..., 'LBG')]
>>> list(geo_t.findClosestFromKey('frnic', N=1, grid=False)) # Nice
[(0.0, 'frnic')]
>>> list(geo_t.findClosestFromKey('frnic', N=2, grid=False, from_keys=('frpaz', 'frply', 'fr
[(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 of (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 fuzzyGetAroundLatLng).
- **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.findClosestFromPoint((43.70, 7.26))) # Nice
[(5.82..., 'NCE')]
>>> list(geo_a.findClosestFromPoint((43.70, 7.26), N=3)) # Nice
[(5.82..., 'NCE'), (30.28..., 'CEQ'), (79.71..., 'ALL')]
>>> list(geo_t.findClosestFromPoint((43.70, 7.26), N=1)) # Nice
[(0.56..., 'frnic')]
>>> # Corner case, from_keys empty is not used
>>> list(geo_t.findClosestFromPoint((43.70, 7.26), N=2, from_keys=()))
[]
>>> list(geo_t.findClosestFromPoint(None, N=2))
[]
>>> #from datetime import datetime
>>> #before = datetime.now()
>>> #for _ in range(100): s = geo_a.findClosestFromPoint((43.70, 7.26), N=3)
>>> #print(datetime.now() - before)
```

No grid.

```
>>> list(geo_o.findClosestFromPoint((43.70, 7.26), grid=False)) # Nice
[(0.60..., 'NCE@1')]
>>> list(geo_a.findClosestFromPoint((43.70, 7.26), grid=False)) # Nice
[(5.82..., 'NCE')]
>>> list(geo_a.findClosestFromPoint((43.70, 7.26), N=3, grid=False)) # Nice
[(5.82..., 'NCE'), (30.28..., 'CEQ'), (79.71..., 'ALL')]
>>> list(geo_t.findClosestFromPoint((43.70, 7.26), N=1, grid=False)) # Nice
[(0.56..., 'frnic')]
>>> list(geo_t.findClosestFromPoint((43.70, 7.26), N=2, grid=False, from_keys=('frpaz', 'frp
[(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 lat/lng, and call findNearPoint.

Parameters

- **key** – the key of the thing (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'), (1.82..., 'JDP'), (8.06..., 'XJY'), (9.95..., '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', 5)) # Nice station, stations <= 5km
[(0.0, 'frnic'), (2.2..., 'fr4342'), (2.3..., 'fr5737'), (4.1..., 'fr4708'), (4.5..., 'fr601
```

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 <= 5km
>>> sorted(geo_t.findNearKey('frnic', 5, grid=False))
[(0.0, 'frnic'), (2.2..., 'fr4342'), (2.3..., 'fr5737'), (4.1..., 'fr4708'), (4.5..., 'fr601
>>>
>>> sorted(geo_a.findNearKey('ORY', 50, grid=False, from_keys=['ORY', 'CDG', 'SFO']))
[(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 of (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 <= 50km
>>> [geo_a.get(k, 'name') for d, k in sorted(geo_a.findNearPoint((48.84, 2.367), 50))]
['Paris-Orly', 'Paris-Le Bourget', 'Toussus-le-Noble', 'Paris - Charles-de-Gaulle']
>>>
>>> # Nice, stations <= 5km
>>> [geo_t.get(k, 'name') for d, k in sorted(geo_t.findNearPoint((43.70, 7.26), 5))]
['Nice-Ville', 'Nice-Riquier', 'Nice-St-Roch', 'Villefranche-sur-Mer', 'Nice-St-Augustin']
>>>
>>> # Wrong geocode
>>> sorted(geo_t.findNearPoint(None, 5))
[]
```

No grid mode.

```
>>> # Paris, airports <= 50km
>>> [geo_a.get(k, 'name') for d, k in sorted(geo_a.findNearPoint((48.84, 2.367), 50, grid=Fa
['Paris-Orly', 'Paris-Le Bourget', 'Toussus-le-Noble', 'Paris - Charles-de-Gaulle']
>>>
>>> # Nice, stations <= 5km
>>> [geo_t.get(k, 'name') for d, k in sorted(geo_t.findNearPoint((43.70, 7.26), 5, grid=Fals
['Nice-Ville', 'Nice-Riquier', 'Nice-St-Roch', 'Villefranche-sur-Mer', 'Nice-St-Augustin']
>>>
```

```
>>> # Paris, airports <= 50km with from_keys input list
>>> sorted(geo_a.findNearPoint((48.84, 2.367), 50, from_keys=['ORY', 'CDG', 'BVE'], grid=Fal
[(12.76..., 'ORY'), (23.40..., 'CDG')]
```

fuzzyGet (*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 fuzzyGet. This is useful when we have geocodes and have to perform a matching based on name and location (see fuzzyGetAroundLatLng).

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

```
>>> geo_t.fuzzyGet('Marseille Charles', 'name')[0]
(0.8..., 'frmsc')
>>> geo_a.fuzzyGet('paris de gaulle', 'name')[0]
(0.78..., 'CDG')
>>> geo_a.fuzzyGet('paris de gaulle', 'name', max_results=3, min_match=0.55)
[(0.78..., 'CDG'), (0.60..., 'HUX'), (0.57..., 'LBG')]
>>> geo_a.fuzzyGet('paris de gaulle', 'name', max_results=3, min_match=0.75)
[(0.78..., 'CDG')]
```

Some corner cases.

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

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

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

Parameters

- **lat_lng** – the lat_lng of the point (a tuple of (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 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.fuzzyGet('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
>>> geo_a.fuzzyGetAroundLatLng((50.9013890, 4.4844440), 20, 'Brussels', 'name', min_match=0.
(0.46..., 'BRU')
>>>
>>> # Now a request limited to some input keys
>>> geo_a.fuzzyGetAroundLatLng((50.9013890, 4.4844440), 2000, 'Brussels', 'name', max_result
[(0.33..., 'ORY')]
```

fuzzyGetCached(*fuzzy_value*, *field*, *max_results=None*, *min_match=0.75*, *from_keys=None*, *verbose=True*, *show_bad=(1, 1)*)

Same as fuzzyGet but with a caching and bias system.

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
- **verbose** – display information on a certain range of similarity
- **show_bad** – the range of similarity

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

```
>>> geo_t.fuzzyGetCached('Marseille Saint Ch.', 'name')[0]
(0.8..., 'frmsc')
>>> geo_a.fuzzyGetCached('paris de gaulle', 'name', show_bad=(0, 1))[0]
[0.79]           paris+de+gaulle ->   paris+charles+de+gaulle (  CDG)
(0.78..., 'CDG')
>>> geo_a.fuzzyGetCached('paris de gaulle', 'name', min_match=0.60, max_results=2, show_bad=
[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', 'name', None, 0.75, None, [(0.5, 'Biased result'
>>> geo_a.fuzzyGetCached('paris de gaulle', 'name', max_results=None, show_bad=(0, 1))[0] #
(0.78..., 'CDG')
>>> geo_a.clearCache()
```

```
>>> geo_a.fuzzyGetCached('paris de gaulle', 'name', max_results=None, min_match=0.75)
Using bias: ('paris+de+gaulle', 'name', None, 0.75, None)
[(0.5, 'Biased result')]
```

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

This get function raises an exception when input is not correct.

Parameters

- **key** – the key of the thing (like 'SFO')
- **field** – the field (like 'name' or 'iata_code')
- **default** – if key is missing, returns default if given

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 ['info', 'code', 'name', 'lines@raw'
```

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

Parameters

- **key** – the key of the thing (like 'SFO')
- **field** – the field (like 'name' or 'iata_code')

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

```
>>> geo_o.getAllDuplicates('ORY', 'name')
['Paris-Orly']
>>> geo_o.getAllDuplicates('THA', 'name')
['Tullahoma Regional Airport/William Northern Field', 'Tullahoma']
>>> geo_o.getAllDuplicates('THA', '__key__')
['THA', 'THA@1']
```

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

getKeysWhere(*conditions*, *from_keys=None*, *reverse=False*, *force_str=False*, *mode='and'*)

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.
- **force_str** – for the str() method before every test
- **mode** – either 'or' or 'and', how to handle several conditions
- **from_keys** – if given, we will look for results from this iterable of keys

Returns an iterable of (v, key) where v is the number of matched condition

```
>>> list(geo_a.getKeysWhere([('city_code', 'PAR')]))
[(1, 'ORY'), (1, 'TNF'), (1, 'CDG'), (1, 'BVA')]
>>> list(geo_o.getKeysWhere([('comment', '')], reverse=True))
[]
>>> list(geo_o.getKeysWhere([('__dup__', '[]')]))
[]
>>> len(list(geo_o.getKeysWhere([('__dup__', [])]))) # 7013 exactly
69...
>>> len(list(geo_o.getKeysWhere([('__dup__', '[]')], force_str=True)))
69...
>>> len(list(geo_o.getKeysWhere([('__par__', [])], reverse=True))) # Counting duplicated key
45...
```

Testing several conditions.

```
>>> c_1 = [('city_code'    , 'PAR')]
>>> c_2 = [('location_type', 'H'  )]
>>> len(list(geo_o.getKeysWhere(c_1)))
18
>>> len(list(geo_o.getKeysWhere(c_2)))
100
>>> len(list(geo_o.getKeysWhere(c_1 + c_2, mode='and')))
2
>>> len(list(geo_o.getKeysWhere(c_1 + c_2, mode='or')))
116
```

This works too o/.

```
>>> len(list(geo_o.getKeysWhere([('city_code', 'PAR'), ('city_code', 'BVE')], mode='and')))
0
>>> len(list(geo_o.getKeysWhere([('city_code', 'PAR'), ('city_code', 'BVE')], mode='or')))
20
```

getLocation(*key*)

Returns geocode as (float, float) or None.

Parameters **key** – the key of the thing (like 'SFO')

Returns the location, a tuple of floats (lat, lng), or None

```
>>> geo_a.getLocation('AGN')
(57.50..., -134.585...)
```

hasDuplicates(*key*)
Tell if a key has duplicates.

Parameters **key** – the key of the thing (like 'SFO')

Returns the number of duplicates

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

hasGeoSupport()
Check if data type has geocoding support.

Returns boolean for geocoding support

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

hasParents(*key*)
Tell if a key has parents.

Parameters **key** – the key of the thing (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', ...
```

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

Parameters

- **key** – the key we want to change a value of
- **field** – the concerned field, like 'name'
- **value** – the new value

Returns None

```
>>> 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') # Not to mess with other tests :)
```

We may even add new fields.

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

setWithDict (*key*, *dictionary*)

Same as set method, except we perform the input with a whole dictionary.

Parameters

- **key** – the key we want to change a value of
- **dictionary** – the dict containing the new data

Returns None

```
>>> geo_f.keys()
[]
>>> geo_f.setWithDict('frnic', {'code' : 'frnic', 'name': 'Nice'})
>>> geo_f.keys()
['frnic']
```

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

OpenTrep integration.

If not hasTrepSupport(), main_trep is not defined and trepGet 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 geo_t.hasTrepSupport():
...     print geo_t.trepGet('sna francisco los agneles')
[(31.5192, 'SFO'), (46.284, 'LAX')]

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

static update (*force=False*)

Launch update script on data files.

visualize(*output='example'*, *label='__key__'*, *point_size=None*, *point_color=None*, *icon_type='auto'*, *from_keys=None*, *catalog=None*, *add_lines=None*, *link_duplicates=True*, *verbose=True*)

Creates map and other visualizations.

Parameters

- **output** – set the name of the rendered files
- **label** – set the field which will appear as map icons title
- **point_size** – set the field defining the map icons circle size
- **point_color** – set the field defining the map icons colors
- **icon_type** – set the global icon size, either 'B', 'S' or 'auto'
- **from_keys** – only display this iterable of keys if not None
- **catalog** – optional color catalog to have specific colors for certain field values
- **add_lines** – optional list of (key1, key2, ..., keyN) to draw additional lines
- **link_duplicates** – boolean toggling lines between duplicated keys feature
- **verbose** – toggle verbosity

Returns (list of templates successfully rendered, total number of templates available).

CHAPTER

THREE

GEOGRIDMODULE MODULE

This module is grid implementation, in order to provide geographical indexation features.

```
>>> a = GeoGrid(radius=20)
Setting grid precision to 4, avg radius to 20km
>>> a.add('ORY', (48.72, 2.359))
>>> a.add('CDG', (48.75, 2.361))
>>> list(a._findInAdjacentCases(encode(48.72, 2.359, a._precision), N=2))
['ORY', 'CDG']
>>> a._keys['ORY']
{'case': 'u09t', 'lat_lng': (48.7..., 2.359)}
>>> neighbors('t0db')
['t0d8', 't0e0', 't06z', 't06x', 't07p', 't0dc', 't0d9', 't0e1']
>>> list(a._recursiveFrontier('t0dbr', N=2))
[set(['t0dbr']), set(['t0e08', 't0e00', 't0dbn', 't0e02', 't0dbq', 't0dbp', 't0dbw', 't0dbx'])]
>>> list(a._recursiveFrontier('t0dbr', N=1))
[set(['t0dbr'])]
>>> sum(len(f) for f in a._recursiveFrontier('t0dbr', N=2))
9
>>> sum(len(f) for f in a._recursiveFrontier('t0dbr', N=3))
25
>>> sum(len(f) for f in a._recursiveFrontier('t0dbr', N=4))
49
>>> sum(len(f) for f in a._recursiveFrontier('t0dbr', N=5))
81
>>> list(a.findNearKey('ORY', 20))
[(0, 'ORY'), (0, 'CDG')]
>>> list(a.findNearKey('ORY', 20, double_check=True))
[(0.0, 'ORY'), (3.33..., 'CDG')]
>>> list(a.findClosestFromPoint((48.75, 2.361), N=2, double_check=True))
[(0.0, 'CDG'), (3.33..., 'ORY')]
```

class GeoBases.GeoGridModule.**GeoGrid**(*precision=5*, *radius=None*, *verbose=True*)

Bases: object

This is the main and only class.

__init__(*precision=5*, *radius=None*, *verbose=True*)

Creates grid.

Parameters

- **radius** – the grid accuracy, in kilometers
- **precision** – the hash length, if radius is given, this length is computed from the radius
- **verbose** – toggle verbosity

Returns None

add(*key*, *lat_lng*, *verbose=True*)
Add a point to the grid.

Parameters

- **key** – the key to be added
- **lat_lng** – the lat_lng of the point (a tuple of (lat, lng))
- **verbose** – toggle verbosity

Returns None

findClosestFromKey(*key*, *N=1*, *double_check=False*, *from_keys=None*)
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.

Note that a similar implementation is done in the LocalHelper, to find efficiently N closest point in a graph, from a point (using heaps).

Parameters

- **key** – the key
- **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 fuzzyGetAroundLatLng).
- **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')]

findClosestFromPoint(*lat_lng*, *N=1*, *double_check=False*, *from_keys=None*)
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.

Note that a similar implementation is done in the LocalHelper, to find efficiently N closest point in a graph, from a point (using heaps).

Parameters

- **lat_lng** – the lat_lng of the point (a tuple of (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 fuzzyGetAroundLatLng).
- **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')]

findNearKey(*key*, *radius=20*, *double_check=False*)
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 lat/lng, and call findNearPoint.

Parameters

- **key** – the key

- **radius** – the radius of the search (kilometers)
- **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')]

findNearPoint(*lat_lng*, *radius=20*, *double_check=False*)

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 of (lat, lng))
- **radius** – the radius of the search (kilometers)
- **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')]

CHAPTER

FOUR

GEOUTILS MODULE

This module is composed of several functions useful for performing calculations on a sphere, such as distance or projections. A part of them has been adapted from: http://www.movable-type.co.uk/scripts/latlong.html

Functions frequently used by other modules:

- *haversine*: a function to compute the shortest path distance between two points on a sphere
- *prog_point*: a function to compute the localization of a point traveling on the shortest path between two points on a sphere, given a progression ratio (like 0%, 50% or 100%). This one uses a dichotomy, because so far I have an exact formula only for 50% (implemented in function *mid_point*)

Simple examples:

```
>>> haversine((48.84, 2.367), (43.70, 7.26)) # Paris -> Nice
683.85...
>>> prog_point(48.84, 2.367, 35.5522, 139.7796, 0.001, accuracy=0.0001)
(48.91..., 2.43...)
>>> prog_point(48.84, 2.367, 35.5522, 139.7796, 1.0)
(35.552..., 139.779...)
```

GeoBases.GeoUtils.**haversine**(*L0*, *L1*)

As a matter of fact, it is easier for other libraries to just use two parameters. Exposing this function is more compliant with geohash signatures and getLocations() geobase function.

Parameters

- **L0** – the LatLng tuple of the first point
- **L1** – the LatLng tuple of the second point

Returns the distance in kilometers

```
>>> haversine((48.84, 2.367), (43.70, 7.26)) # Paris -> Nice
683.85...
```

Case of unknown location.

```
>>> haversine(None, (43.70, 7.26)) # returns None
```

GeoBases.GeoUtils.**haversine_precise**(*lat0*, *lng0*, *lat1*, *lng1*)

A function to compute the shortest path distance between two points on a sphere, using Haversine formula.

Parameters

- **lat0** – the latitude of the first point
- **lng0** – the longitude of the first point

- **lat1** – the latitude of the second point
- **lng1** – the longitude of the second point

Returns the distance in kilometers

```
>>> haversine_precise(48.84, 2.367, 43.70, 7.26) # Paris -> Nice
683.85...
>>> haversine_precise(48.84, 2.367, 35.5522, 139.7796) # Paris -> Tokyo
9730.22...
```

GeoBases.GeoUtils.**haversine_simple**(*lat0*, *lng0*, *lat1*, *lng1*)

Another implementation of Haversine formula, but this one works well only for small amplitudes.

Parameters

- **lat0** – the latitude of the first point
- **lng0** – the longitude of the first point
- **lat1** – the latitude of the second point
- **lng1** – the longitude of the second point

Returns the distance in kilometers

```
>>> haversine_simple(48.84, 2.367, 43.70, 7.26) # Paris -> Nice
683.85...
>>> haversine_simple(48.84, 2.367, 35.5522, 139.7796) # Paris -> Tokyo
9730.22...
```

GeoBases.GeoUtils.**mercator**(*lat*, *lng*)

Returns Mercator projection

Parameters

- **lat** – the latitude of the point
- **lng** – the longitude of the point

Returns the projection

```
>>> mercator(48.84, 2.367)
(0.85..., 0.04...)
```

GeoBases.GeoUtils.**mid_point**(*lat0*, *lng0*, *lat1*, *lng1*)

A function to compute the localization exactly in the middle of the shortest path between two points on a sphere.

The given example provides the point between Paris and Tokyo. You can see the result at: http://maps.google.com/maps?f=q&hl=fr&ie=UTF8&ll=67.461,86.233&t=k&z=13 It it somewhere in the North os Russia.

Parameters

- **lat0** – the latitude of the first point
- **lng0** – the longitude of the first point
- **lat1** – the latitude of the second point
- **lng1** – the longitude of the second point

Returns the position of the point in the middle

```
>>> mid_point(48.84, 2.367, 35.5522, 139.7796) # Paris -> Tokyo
(67.461..., 86.233...)
```

GeoBases.GeoUtils.**prog_point**(*lat0*, *lng0*, *lat1*, *lng1*, *progression=0.5*, *accuracy=0.005*, *verbose=False*)

A function to compute the localization of a point traveling on the shortest path between two points on a sphere, given a progression ratio (like 0%, 50% or 100%). This one uses a dichotomy on mid_point.

Parameters

- **lat0** – the latitude of the first point
- **lng0** – the longitude of the first point
- **lat1** – the latitude of the second point
- **lng1** – the longitude of the second point
- **progression** – the progression of the traveler of the shortest path
- **accuracy** – the accuracy of the dichotomy
- **verbose** – display or not informations about the dichotomy

Raises ValueError, if progression not in [0, 1]

Returns the position of the progressing point

```
>>> prog_point(48.84, 2.367, 35.5522, 139.7796, 0)
(48.84..., 2.367...)
>>> prog_point(48.84, 2.367, 35.5522, 139.7796,
...            progression=0.001,
...            accuracy=0.0001,
...            verbose=True)
0.0000 < 0.0010 < 0.0020 in 10 steps
(48.91..., 2.43...)
>>> prog_point(48.84, 2.367, 35.5522, 139.7796, 0.5)
(67.461..., 86.233...)
>>> prog_point(48.84, 2.367, 35.5522, 139.7796, 1.0)
(35.552..., 139.779...)
```

GeoBases.GeoUtils.**radian**(*a*)

Degree to radian conversion.

Parameters **a** – the input in degree

Returns the output in radian

```
>>> radian(0)
0.0
>>> radian(180)
3.14...
```

GeoBases.GeoUtils.**unradian**(*a*)

Radian to degree conversion.

Parameters **a** – the input in radian

Returns the output in degree

```
>>> unradian(0)
0.0
>>> unradian(3.1415)
179.9...
```

CHAPTER

FIVE

LEVENSHTEINUTILS MODULE

This module is composed of several functions useful for performing string comparisons. It is oriented for data like names of cities, airports, train stations, because the string comparisons will not count standard words like 'ville' or 'sncf'.

Functions frequently used by other modules:

- *mod_leven*: a function to compute the distance between two strings. It is based on the Levenshtein ratio.
- *clean*: a function to clean string before comparisons

This module strongly relies on one other module:

- *Levenshtein*: this module implements some standard algorithms to compare strings, such as the Levenshtein distance

Simple examples:

```
>>> clean('St-Etienne" "    ')
['saint', 'etienne']
>>> clean('antibes sncf 2 (centre)')
['antibes', 'centre']
>>> mod_leven('antibes', 'antibs')
0.92...
>>> mod_leven('Aéroport CDG  2', 'aeroport-cdg')
1.0
```

GeoBases.LevenshteinUtils.**clean**(*string*)

Global cleaning function which put all previous ones together.

This function cleans the string to have a better comparison. Different steps:

•lower and strip (remove leading and trailing spaces/tabulations)

•manage accentuated characters, parenthesis

•properly split the string

•handle common aliases, irrelevant words, numbers and spaces

Parameters **string** – the string to be processed

Returns the clean string

```
>>> clean('Paris')
['paris']
>>> clean('Paris ville')
['paris']
```

```
>>> clean('St-Etienne')
['saint', 'etienne']
>>> clean('Aix-Les   Bains')
['aix']
>>> clean('antibes sncf 2 (centre)')
['antibes', 'centre']
```

GeoBases.LevenshteinUtils.**handle_accents**(*string*)

Remove accentuated characters in a word, and replace them with non-accentuated ones.

Parameters **string** – the string to be processed

Returns the unaccentuad string

```
>>> handle_accents('être')
'etre'
>>> handle_accents('St-Etienne SNCF (Châteaucreux)')
'St-Etienne SNCF (Chateaucreux)'
```

GeoBases.LevenshteinUtils.**handle_alias**(*strings*)

Some common words have different ways to be used. This function normalize those, to have a better comparison tool later. For example, we can replace 'st' by 'saint'.

Parameters **strings** – the list of words to be processed

Returns the list of words after normalization

```
>>> handle_alias(['st', 'etienne', 'SNCF', ''])
['saint', 'etienne', 'SNCF', '']
```

GeoBases.LevenshteinUtils.**handle_numbers_spaces**(*strings*)

Some words contains numbers irrelevant to string comparison. This function remove those, to have a better comparison tool later. It also removes blanks which could have been left during earlier removals.

Parameters **strings** – the list of words to be processed

Returns the list of words number-free

```
>>> handle_numbers_spaces(['saint', 'etienne', '2', ''])
['saint', 'etienne']
```

GeoBases.LevenshteinUtils.**handle_parenthesis_info**(*string*, *parts=None*)

When a word contains parenthesis, this function picks only the part *before* the parenthesis.

Parameters

- **string** – the string to be processed
- **parts** – whichi part to keep, either 'before', 'in', or 'after'

Returns the parenthesis-free string

```
>>> handle_parenthesis_info('Lyon Part-Dieu (TGV)')
'Lyon Part-Dieu TGV'
>>> handle_parenthesis_info('Lyon Part-Dieu (TGV)', parts=['before'])
'Lyon Part-Dieu'
>>> handle_parenthesis_info('(Sncf) City')
'Sncf City'
>>> handle_parenthesis_info('Lyon (Sncf) City', parts=['in', 'after'])
'Sncf City'
>>> handle_parenthesis_info('St-Etienne SNCF (Chateaucreux)')
'St-Etienne SNCF Chateaucreux'
```

GeoBases.LevenshteinUtils.**handle_transparent**(*strings*)

Some words are often parts irrelevant to string comparison. This function remove those, to have a better comparison tool later. For example, we can remove 'ville' or 'sncf'.

Parameters **strings** – the list of words to be processed

Returns the list of words after normalization

```
>>> handle_transparent(['saint', 'etienne', 'sncf', ''])
['saint', 'etienne', '']
>>> handle_transparent(['aix', 'ville'])
['aix']
```

GeoBases.LevenshteinUtils.**is_sublist**(*subL*, *L*)

This function tests the inclusion of a list in another one.

Parameters

- **subL** – the tested sub-list
- **L** – the tested list

Returns a boolean

```
>>> is_sublist([2], [2,3])
True
>>> is_sublist([2,3], [2,3])
True
>>> is_sublist([], [2,3])  # [] is a sub-list of everyone
True
>>> is_sublist([2,3], [])
False
>>> is_sublist([4], [2,3])
False
>>> is_sublist([2,3], [3,2]) # Order matter
False
>>> is_sublist([2,3,4], [2,3])
False
>>> is_sublist([2,3], [2,3,4])
True
```

GeoBases.LevenshteinUtils.**mod_leven**(*str1*, *str2*, *heuristic_inclusion=True*, *heuristic_inclusion_value=0.99*)

The main comparison function. In fact, the real work has already been done previously, with the cleaning function.

This function uses Levenshtein ratio to evaluate the distance between the two strings. It is up to the user to define which distance is acceptable for classic mispelling, but from my point of view, 90% is fairly acceptable.

When we have a inclusion of one string in the other (list inclusion, not possible to include partial words such as toul for toulon), we put the ratio of similarity to 99%, this is heuristic. Why not 100%? Because, if another entry in the base really match 100%, this will probably be an even better match, such as: orleans+gervais matches orleans with inclusion heuristic (so 99%), but we also have the real orleans+gervais station in the base, and this one is a 100% match, so this will take over as the best match.

Sometimes rare cases of high ratio are not relevant. For example, Toul match Toulon with 80%, but this is wrong, and may be handled with a cache to manage exceptional failing by the graph module, or by upping the acceptance limit.

Parameters

- **str1** – the first string to compare

- **str2** – the second string to compare
- **heuristic_inclusion** – boolean to toggle the heuristic inclusion
- **heuristic_inclusion_value** – for heuristic inclusion, the value considered

Returns the distance, which is a ratio (0% to 100%)

```
>>> mod_leven('antibes', 'antibs')
0.92...
>>> mod_leven('toul', 'toulon')
0.8...
>>> mod_leven('Aéroport CDG  2 TGV', 'aeroport-cdg') # Inclusion
0.9...
>>> mod_leven('Bains les bains', 'Tulle')
0.0
```

Tweaking behavior.

```
>>> mod_leven('Aéroport CDG  2 TGV', 'aeroport-cdg', False) # No inclusion
0.85...
```

GeoBases.LevenshteinUtils.**split_separators**(*string*)

When a word contains different separators, this function split the word using all separators.

Parameters **string** – the string to be processed

Returns the list of words after splitting

```
>>> split_separators('Lyon Part-Dieu')
['Lyon', 'Part', 'Dieu']
>>> split_separators('St-Etienne SNCF ')
['St', 'Etienne', 'SNCF', '']
```

GeoBases.LevenshteinUtils.**str_lowercase**(*string*)

Lower case adapted for str type.

```
>>> print('Étaples'.lower()) # Fail!
Étaples
>>> print(str_lowercase('Étaples')) # Win!
étaples
```

PYTHON MODULE INDEX

g