# geograpy3

Aug 12, 2021

# Contents:


**1 geograpy package** **1**
- 1.1 Submodules . . . 1
- 1.2 geograpy.extraction module . . . 1
- 1.3 geograpy.labels module . . . 1
- 1.4 geograpy.locator module . . . 2
- 1.5 geograpy.places module . . . 10
- 1.6 geograpy.prefixtree module . . . 10
- 1.7 geograpy.utils module . . . 10
- 1.8 geograpy.wikidata module . . . 10
- 1.9 Module contents . . . 12

**2 setup module** **13**

**3 tests package** **15**
- 3.1 Submodules . . . 15
- 3.2 tests.test_extractor module . . . 15
- 3.3 tests.test_locator module . . . 16
- 3.4 tests.test_places module . . . 17
- 3.5 tests.test_prefixtree module . . . 18
- 3.6 tests.test_wikidata module . . . 18
- 3.7 Module contents . . . 18

**4 Indices and tables** **19**

**Python Module Index** **21**

**Index** **23**

CHAPTER 1

# geograpy package

## 1.1 Submodules

## 1.2 geograpy.extraction module

**class** geograpy.extraction.**Extractor**(*text=None*, *url=None*, *debug=False*)

Bases: object

Extract geo context for text or from url

**find_entities**(*labels=['GPE', 'GSP', 'PERSON', 'ORGANIZATION']*)

Find entities with the given labels set self.places and returns it :param labels: Labels: The labels to filter

**Returns** List of places

**Return type** list

**find_geoEntities**()

Find geographic entities

**Returns** List of places

**Return type** list

**set_text**()

Setter for text

**split**(*delimiter=', '*)

simpler regular expression splitter with not entity check

hat tip: https://stackoverflow.com/a/1059601/1497139

## 1.3 geograpy.labels module

Created on 2020-09-10

@author: wf

**class** geograpy.labels.**Labels**
Bases: object

NLTK labels

**default = ['GPE', 'GSP', 'PERSON', 'ORGANIZATION']**

**geo = ['GPE', 'GSP']**

## 1.4 geograpy.locator module

The locator module allows to get detailed city information including the region and country of a city from a location string.

Examples for location strings are:

Amsterdam, Netherlands Vienna, Austria Vienna, IL Paris - Texas Paris TX

the locator will lookup the cities and try to disambiguate the result based on the country or region information found.

The results in string representationa are:

Amsterdam (NH(North Holland) - NL(Netherlands)) Vienna (9(Vienna) - AT(Austria)) Vienna (IL(Illinois) - US(United States)) Paris (TX(Texas) - US(United States)) Paris (TX(Texas) - US(United States))

Each city returned has a city.region and city.country attribute with the details of the city.

Created on 2020-09-18

@author: wf

**class** geograpy.locator.**City**(***kwargs*)
Bases: *geograpy.locator.Location*

a single city as an object

**country**

**static fromGeoLite2**(*record*)

**classmethod getSamples**()

**region**

**setValue**(*name*, *record*)
set a field value with the given name to the given record dicts corresponding entry or none

**Parameters**

- **name** (*string*) – the name of the field
- **record** (*dict*) – the dict to get the value from

**class** geograpy.locator.**CityManager**(*name: str = 'CityManager'*, *config: lodstorage.storageconfig.StorageConfig = None*, *debug=False*)
Bases: *geograpy.locator.LocationManager*

a list of cities

**classmethod fromJSONBackup**(*config: lodstorage.storageconfig.StorageConfig = None*)
get city list from json backup (json backup is based on wikidata query results)

**Parameters** **jsonStr** (*str*) – JSON string the CityList should be loaded from. If None json backup is loaded. Default is None

**Returns** CityList based on the json backup

**classmethod fromWikidata**(*fromBackup: bool = True*, *countryIDs: list = None*, *regionIDs: list = None*, *config: lodstorage.storageconfig.StorageConfig = None*)

get city list form wikidata

**Parameters**

- **fromBackup** (*bool*) – If True instead of querying wikidata a backup of the wikidata results is used to create the city list. Otherwise wikidata is queried for the city data. Default is True
- **countryIDs** (*list*) – List of countryWikiDataIDs. Limits the returned cities to the given countries
- **regionIDs** (*list*) – List of regionWikiDataIDs. Limits the returned cities to the given regions

**Returns** CityList based wikidata query results

**classmethod getLocationLodFromJsonBackup**()

**updateCity**(*wikidataid: str*, *cityRecord: dict*)

Updates the city corresponding to the given city with the given data. If the city does not exist a new city object is created and added to this CityList :param wikidataid: wikidata id of the city that should be updated/added :type wikidataid: str :param cityRecord: data of the given city that should be updated/added :type cityRecord: dict

**Returns** Nothing

**class** geograpy.locator.**Country**(*lookupSource='sqlDB'*, ***kwargs*)

Bases: *geograpy.locator.Location*

a country

**static fromGeoLite2**(*record*)

create a country from a geolite2 record

**static fromPyCountry**(*pcountry*)

**Parameters** **pcountry** (*PyCountry*) – a country as gotten from pycountry

**Returns** the country

**Return type** *Country*

**classmethod getSamples**()

**class** geograpy.locator.**CountryManager**(*name: str = 'CountryManager'*, *config: lodstorage.storageconfig.StorageConfig = None*, *debug=False*)

Bases: *geograpy.locator.LocationManager*

a list of countries

**classmethod fromErdem**()

get country list provided by Erdem Ozkol https://github.com/erdem

**classmethod fromJSONBackup**(*config: lodstorage.storageconfig.StorageConfig = None*)

get country list from json backup (json backup is based on wikidata query results)

**Returns** CountryList based on the json backup

**classmethod fromWikidata**()
get country list form wikidata

**classmethod from_sqlDb**(*sqlDB*)

**classmethod getLocationLodFromJsonBackup**()

**class** geograpy.locator.**Earth**
Bases: object

**radius = 6371.0**

**class** geograpy.locator.**Location**(**kwargs*)
Bases: lodstorage.jsonable.JSONAble

Represents a Location

**balltreeQueryResultToLocationManager**(*distances*, *indices*, *lookupListOfLocations*)
convert the given ballTree Query Result to a LocationManager

**Parameters**

- **distances** (*list*) – array of distances
- **indices** (*list*) – array of indices
- **lookupListOfLocations** (*list*) – a list of valid locations to use for lookup

**Returns** a list of result Location/distance tuples

**Return type** list

**distance**(*other*) → float
calculate the distance to another Location

**Parameters** **other** (Location) – the other location

**Returns** the haversine distance in km

**getLocationsWithinRadius**(*lookupLocationManager*, *radiusKm: float*)
Gives the n closest locations to me from the given lookupListOfLocations

**Parameters**

- **lookupLocationManager** (LocationManager) – a LocationManager object to use for lookup
- **radiusKm** (*float*) – the radius in which to check (in km)

**Returns** a list of result Location/distance tuples

**Return type** list

**getNClosestLocations**(*lookupLocationManager*, *n: int*)
Gives a list of up to n locations which have the shortest distance to me as calculated from the given listOfLocations

**Parameters**

- **lookupLocationManager** (LocationManager) – a LocationManager object to use for lookup
- **n** (*int*) – the maximum number of closest locations to return

**Returns** a list of result Location/distance tuples

**Return type** list

**classmethod getSamples**()

**static haversine**(*lon1*, *lat1*, *lon2*, *lat2*)
Calculate the great circle distance between two points on the earth (specified in decimal degrees)

**isKnownAs**(*name*) → bool
Checks if this location is known under the given name

**Parameters** **name** (*str*) – name the location should be checked against

**Returns** True if the given name is either the name of the location or present in the labels of the location

**class** geograpy.locator.**LocationContext**(*countryManager: geograpy.locator.CountryManager*, *regionManager: geograpy.locator.RegionManager*, *cityManager: geograpy.locator.CityManager*)

Bases: object

Holds LocationManagers of all hierarchy levels and provides methods to traverse through the levels

**cities**

**countries**

**classmethod fromCache**(*config: lodstorage.storageconfig.StorageConfig = None*, *forceUpdate: bool = False*)
Inits a LocationContext form Cache if existent otherwise init cache

**classmethod fromJSONBackup**(*config: lodstorage.storageconfig.StorageConfig = None*)
Inits a LocationContext form the JSON backup

**getCities**(*name: str*)
Returns all cities that are known under the given name

**getCountries**(*name: str*)
Returns all countries that are known under the given name

**static getDefaultConfig**() → lodstorage.storageconfig.StorageConfig
Returns default StorageConfig

**getRegions**(*name: str*)
Returns all regions that are known under the given name

**interlinkLocations**()
Interlinks locations by adding the hierarchy references to the locations

**locateLocation**(**locations*)
Get possible locations for the given location names. Current prioritization of the results is city(ordered by population)→region→country ToDo: Extend the ranking of the results e.g. matching of multiple location parts increase ranking :param *locations:

Returns:

**regions**

**class** geograpy.locator.**LocationManager**(*name: str*, *entityName: str*, *entityPluralName: str*, *listName: str = None*, *clazz=None*, *primaryKey: str = None*, *config: lodstorage.storageconfig.StorageConfig = None*, *handleInvalidListTypes=True*, *filterInvalidListTypes=False*, *debug=False*)

Bases: lodstorage.entity.EntityManager

a list of locations

**static downloadBackupFile**(*url: str*, *fileName: str*, *force: bool = False*)
Downloads from the given url the zip-file and extracts the file corresponding to the given fileName.

**Parameters**

- **url** – url linking to a downloadable gzip file
- **fileName** – Name of the file that should be extracted from gzip file
- **force** (*bool*) – True if the download should be forced

**Returns** Name of the extracted file with path to the backup directory

**static getBackupDirectory**()

**getBallTuple**(*cache: bool = True*)
get the BallTuple=BallTree,validList of this location list

**Parameters**

- **cache** (*bool*) – if True calculate and use a cached version otherwise recalculate on
- **call of this function** (*every*) –

**Returns** a sklearn.neighbors.BallTree for the given list of locations, list: the valid list of locations list: valid list of locations

**Return type** BallTree,list

**getByName**(*name: str*)
Get locations matching given name :param name: Name of the location

**Returns** Returns locations that match the given name

**static getFileContent**(*path: str*)

**getLocationByID**(*wikidataID: str*)
Returns the location object that corresponds to the given location

**Parameters** **wikidataID** – wikidataid of the location that should be returned

**Returns** Location object

**static getURLContent**(*url: str*)

**class** geograpy.locator.**Locator**(*db_file=None*, *correctMisspelling=False*, *debug=False*)
Bases: object

location handling

**cities_for_name**(*cityName*)
find cities with the given cityName

**Parameters** **cityName** (*string*) – the potential name of a city

**Returns** a list of city records

**correct_country_misspelling**(*name*)
correct potential misspellings :param name: the name of the country potentially misspelled :type name: string

**Returns** correct name of unchanged

**Return type** string

**createViews**(*sqlDB*)

**db_has_data**()
check whether the database has data / is populated

**Returns** True if the cities table exists and has more than one record

**Return type** boolean

**db_recordCount**(*tableList*, *tableName*)
count the number of records for the given tableName

**Parameters**

- **tableList** (*list*) – the list of table to check
- **tableName** (*str*) – the name of the table to check

**Returns** int: the number of records found for the table

**disambiguate**(*country*, *regions*, *cities*, *byPopulation=True*)
try determining country, regions and city from the potential choices

**Parameters**

- **country** (Country) – a matching country found
- **regions** (*list*) – a list of matching Regions found
- **cities** (*list*) – a list of matching cities found

**Returns** the found city or None

**Return type** *City*

**getAliases**()
get the aliases hashTable

**getCountry**(*name*)
get the country for the given name :param name: the name of the country to lookup :type name: string

**Returns** the country if one was found or None if not

**Return type** country

**getGeolite2Cities**()
get the Geolite2 City-Locations as a list of Dicts

**Returns** a list of Geolite2 City-Locator dicts

**Return type** list

**static getInstance**(*correctMisspelling=False*, *debug=False*)
get the singleton instance of the Locator. If parameters are changed on further calls the initial parameters will still be in effect since the original instance will be returned!

**Parameters**

- **correctMispelling** (*bool*) – if True correct typical misspellings
- **debug** (*bool*) – if True show debug information

**getView**()
get the view to be used

**Returns** the SQL view to be used for CityLookups e.g. GeoLite2CityLookup

**Return type** str

**getWikidataCityPopulation**(*sqlDB*, *endpoint=None*)

**Parameters**

- **sqlDB** (*SQLDB*) – target SQL database
- **endpoint** (*str*) – url of the wikidata endpoint or None if default should be used

**static isISO**(*s*)
check if the given string is an ISO code

**Returns** True if the string is an ISO Code

**Return type** bool

**is_a_country**(*name*)
check if the given string name is a country

**Parameters** **name** (*string*) – the string to check

**Returns** if pycountry thinks the string is a country

**Return type** True

**locateCity**(*places*)
locate a city, region country combination based on the given wordtoken information

**Parameters**

- **places** (*list*) – a list of places derived by splitting a locality e.g. “San Francisco, CA”
- **to "San Francisco", "CA"** (*leads*) –

**Returns** a city with country and region details

**Return type** *City*

**locator = None**

**places_by_name**(*placeName*, *columnName*)
get places by name and column :param placeName: the name of the place :type placeName: string :param columnName: the column to look at :type columnName: string

**populateFromWikidata**(*sqlDB*)
populate countries and regions from Wikidata

**Parameters** **sqlDB** (*SQLDB*) – target SQL database

**populate_Cities**(*sqlDB*)
populate the given sqlDB with the Geolite2 Cities

**Parameters** **sqlDB** (*SQLDB*) – the SQL database to use

**populate_Cities_FromWikidata**(*sqlDB*)
populate the given sqlDB with the Wikidata Cities

**Parameters** **sqlDB** (*SQLDB*) – target SQL database

**populate_Countries**(*sqlDB*)
populate database with countries from wikiData

**Parameters** **sqlDB** (*SQLDB*) – target SQL database

**populate_Regions**(*sqlDB*)
populate database with regions from wikiData

**Parameters** **sqlDB** (*SQLDB*) – target SQL database

**populate_Version**(*sqlDB*)
populate the version table

**Parameters** **sqlDB** (*SQLDB*) – target SQL database

**populate_db**(*force=False*)
populate the cities SQL database which caches the information from the GeoLite2-City-Locations.csv file

**Parameters** **force** (*bool*) – if True force a recreation of the database

**readCSV**(*fileName*)

**recreateDatabase**()
recreate my lookup database

**regions_for_name**(*region_name*)
get the regions for the given region_name (which might be an ISO code)

**Parameters** **region_name** (*string*) – region name

**Returns** the list of cities for this region

**Return type** list

**static resetInstance**()

**class** geograpy.locator.**Region**(***kwargs*)
Bases: *geograpy.locator.Location*

a Region (Subdivision)

**country**

**static fromGeoLite2**(*record*)
create a region from a Geolite2 record

**Parameters** **record** (*dict*) – the records as returned from a Query

**Returns** the corresponding region information

**Return type** *Region*

**static fromWikidata**(*record*)
create a region from a Wikidata record

**Parameters** **record** (*dict*) – the records as returned from a Query

**Returns** the corresponding region information

**Return type** *Region*

**classmethod getSamples**()

**class** geograpy.locator.**RegionManager**(*name: str = 'RegionManager'*, *config: lodstorage.storageconfig.StorageConfig = None*, *debug=False*)
Bases: *geograpy.locator.LocationManager*

a list of regions

**classmethod fromJSONBackup**(*config: lodstorage.storageconfig.StorageConfig = None*)
get region list from json backup (json backup is based on wikidata query results)

**Returns** RegionList based on the json backup

**classmethod fromWikidata**(*config: lodstorage.storageconfig.StorageConfig = None*)
get region list form wikidata

**classmethod from_sqlDb**(*sqlDB*, *config: lodstorage.storageconfig.StorageConfig = None*)

**classmethod getLocationLodFromJsonBackup**()

geograpy.locator.**main**(*argv=None*)
main program.

## 1.5 geograpy.places module

**class** geograpy.places.**PlaceContext**(*place_names*, *setAll=True*)
Bases: *geograpy.locator.Locator*

Adds context information to a place name

**get_region_names**(*country_name*)

**setAll**()
Set all context information

**set_cities**()
set the cities information

**set_countries**()
get the country information from my places

**set_other**()

**set_regions**()

## 1.6 geograpy.prefixtree module

## 1.7 geograpy.utils module

geograpy.utils.**fuzzy_match**(*s1*, *s2*, *max_dist=0.8*)
Fuzzy match the given two strings with the given maximum distance :param s1: string: First string :param s2: string: Second string :param max_dist: float: The distance - default: 0.8

**Returns** jellyfish jaro_winkler_similarity based on https://en.wikipedia.org/wiki/Jaro-Winkler_distance

**Return type** float

geograpy.utils.**remove_non_ascii**(*s*)
Remove non ascii chars from the given string :param s: string: The string to remove chars from

**Returns** The result string with non-ascii chars removed

**Return type** string

Hat tip: http://stackoverflow.com/a/1342373/2367526

## 1.8 geograpy.wikidata module

Created on 2020-09-23

@author: wf

**class** geograpy.wikidata.**Wikidata**(*endpoint='https://query.wikidata.org/sparql'*)
Bases: object

Wikidata access

**getCities**(*region=None*, *country=None*)
get the cities from Wikidata

**Parameters**

- **region** – List of countryWikiDataIDs. Limits the returned cities to the given countries
- **country** – List of regionWikiDataIDs. Limits the returned cities to the given regions

**getCitiesOfRegion**(*regionWikidataId: str*, *limit: int*)
Queries the cities of the given region. If the region is a city state the region is returned as city. The cities are ordered by population and can be limited by the given limit attribute.

**Parameters**

- **regionWikidataId** – wikidata id of the region the cities should be queried for
- **limit** – Limits the amount of returned cities

**Returns** Returns list of cities of the given region ordered by population

**getCityPopulations**(*profile=True*)
get the city populations from Wikidata

**Parameters** **profile** (*bool*) – if True show profiling information

**static getCoordinateComponents**(*coordinate: str) -> (<class 'float'>*, *<class 'float'>*)
Converts the wikidata coordinate representation into its subcomponents longitude and latitude Example: 'Point(-118.25 35.05694444)' results in ('-118.25' '35.05694444')

**Parameters** **coordinate** – coordinate value in the format as returned by wikidata queries

**Returns** Returns the longitude and latitude of the given coordinate as separate values

**getCountries**()
get a list of countries

try query

**getRegions**()
get Regions from Wikidata

try query

**static getValuesClause**(*varName: str*, *values*, *wikidataEntities: bool = True*)
generates the SPARQL value clause for the given variable name containing the given values :param varName: variable name for the ValuesClause :param values: values for the clause :param wikidataEntities: if true the wikidata prefix is added to the values otherwise it is expected taht the given values are proper IRIs :type wikidataEntities: bool

**Returns** str

**static getWikidataId**(*wikidataURL: str*)
Extracts the wikidata id from the given wikidata URL

**Parameters** **wikidataURL** – wikidata URL the id should be extracted from

**Returns** The wikidata id if present in the given wikidata URL otherwise None

## 1.9 Module contents

main geography 3 module

geograpy.**get_geoPlace_context**(*url=None*, *text=None*, *debug=False*)

Get a place context for a given text with information about country, region, city and other based on NLTK Named Entities having the Geographic(GPE) label.

**Parameters**

- **url** (*String*) – the url to read text from (if any)
- **text** (*String*) – the text to analyze
- **debug** (*boolean*) – if True show debug information

**Returns** PlaceContext: the place context

**Return type** places

geograpy.**get_place_context**(*url=None, text=None, labels=['GPE', 'GSP', 'PERSON', 'ORGANIZATION'], debug=False*)

Get a place context for a given text with information about country, region, city and other based on NLTK Named Entities in the label set Geographic(GPE), Person(PERSON) and Organization(ORGANIZATION).

**Parameters**

- **url** (*String*) – the url to read text from (if any)
- **text** (*String*) – the text to analyze
- **debug** (*boolean*) – if True show debug information

**Returns** PlaceContext: the place context

**Return type** pc

geograpy.**locateCity**(*location*, *correctMisspelling=False*, *debug=False*)

locate the given location string :param location: the description of the location :type location: string

**Returns** the location

**Return type** *Locator*

CHAPTER 2

# setup module

CHAPTER 3

# tests package

## 3.1 Submodules

## 3.2 tests.test_extractor module

**class** tests.test_extractor.**TestExtractor**(*methodName='runTest'*)

Bases: unittest.case.TestCase

test Extractor

**check**(*places*, *expectedList*)

check the places for begin non empty and having at least the expected List of elements

**Parameters**

- **places** (*Places*) – the places to check
- **expectedList** (*list*) – the list of elements to check

**setUp**()

Hook method for setting up the test fixture before exercising it.

**tearDown**()

Hook method for deconstructing the test fixture after testing it.

**testExtractorFromText**()

test different texts for getting geo context information

**testExtractorFromUrl**()

test the extractor

**testGeograpyIssue32**()

test https://github.com/ushahidi/geograpy/issues/32

**testGetGeoPlace**()

test geo place handling

**testIssue10**()
test https://github.com/somnathrakshit/geograpy3/issues/10 Add ISO country code

**testIssue7**()
test https://github.com/somnathrakshit/geograpy3/issues/7 disambiguating countries

**testIssue9**()
test https://github.com/somnathrakshit/geograpy3/issues/9 [BUG]AttributeError: 'NoneType' object has no attribute 'name' on "Pristina, Kosovo"

**testStackOverflow54721435**()
see https://stackoverflow.com/questions/54721435/unable-to-extract-city-names-from-a-text-using-geograpypython

**testStackoverflow43322567**()
see https://stackoverflow.com/questions/43322567

**testStackoverflow54077973**()
see https://stackoverflow.com/questions/54077973/geograpy3-library-for-extracting-the-locations-in-the-text-gives-unicode

**testStackoverflow54712198**()
see https://stackoverflow.com/questions/54712198/not-only-extracting-places-from-a-text-but-also-other-names-in-geograp

**testStackoverflow55548116**()
see https://stackoverflow.com/questions/55548116/geograpy3-library-is-not-working-properly-and-give-traceback-error

**testStackoverflow62152428**()
see https://stackoverflow.com/questions/62152428/extracting-country-information-from-description-using-geograpy?noredirect=1#comment112899776_62152428

## 3.3 tests.test_locator module

Created on 2020-09-19

@author: wf

**class** tests.test_locator.**TestLocator**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test the Locator class from the location module

**checkExamples**(*examples*, *countries*, *debug=False*, *check=True*)
check that the given example give results in the given countries :param examples: a list of example location strings :type examples: list :param countries: a list of expected country iso codes :type countries: list

**setUp**()
Hook method for setting up the test fixture before exercising it.

**tearDown**()
Hook method for deconstructing the test fixture after testing it.

**testDelimiters**()
test the delimiter statistics for names

**testExamples**()
test examples

**testGeolite2Cities**()
test the locs.db cache for cities

**testHasData**()
check has data and populate functionality

**testIsoRegexp**()
test regular expression for iso codes

**testIssue15**()
https://github.com/somnathrakshit/geograpy3/issues/15 test Issue 15 Disambiguate via population, gdp data

**testIssue17**()
test issue 17:

https://github.com/somnathrakshit/geograpy3/issues/17

[BUG] San Francisco, USA and Auckland, New Zealand should be locatable #17

**testIssue19**()
test issue 19

**testIssue22**()
https://github.com/somnathrakshit/geograpy3/issues/22

**testIssue41_CountriesFromErdem**()
test getting Country list from Erdem

**testIssue_42_distance**()
test haversine and location

**testPopulation**()
test adding population data from wikidata to GeoLite2 information

**testProceedingsExample**()
test a proceedings title Example

**testStackOverflow64379688**()
compare old and new geograpy interface

**testStackOverflow64418919**()
https://stackoverflow.com/questions/64418919/problem-retrieving-region-in-us-with-geograpy3

**testWordCount**()
test the word count

## 3.4 tests.test_places module

**class** tests.test_places.**TestPlaces**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test Places

**setUp**()
Hook method for setting up the test fixture before exercising it.

**tearDown**()
Hook method for deconstructing the test fixture after testing it.

**testPlaces**()
test places

## 3.5 tests.test_prefixtree module

## 3.6 tests.test_wikidata module

Created on 2020-09-23

@author: wf

**class** tests.test_wikidata.**TestWikidata**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test the wikidata access for cities

**setUp**()
Hook method for setting up the test fixture before exercising it.

**tearDown**()
Hook method for deconstructing the test fixture after testing it.

**testGetCitiesOfRegion**()
Test getting cities based on region wikidata id

**testGetCoordinateComponents**()
test the splitting of coordinate components in WikiData query results

**testGetWikidataId**()
test getting a wikiDataId from a given URL

**testLocatorWithWikiData**()
test Locator

**testWikidataCities**()

test getting city information from wikidata

1372 Singapore 749 Beijing, China 704 Paris, France 649 Barcelona, Spain 625 Rome, Italy 616 Hong Kong 575 Bangkok, Thailand 502 Vienna, Austria 497 Athens, Greece 483 Shanghai, China

**testWikidataCountries**()
test getting country information from wikidata

## 3.7 Module contents

CHAPTER 4

# Indices and tables

- genindex
- modindex
- search

# Python Module Index

## g

geograpy, 12
geograpy.extraction, 1
geograpy.labels, 1
geograpy.locator, 2
geograpy.places, 10
geograpy.utils, 10
geograpy.wikidata, 10

## t

tests, 18
tests.test_extractor, 15
tests.test_locator, 16
tests.test_places, 17
tests.test_wikidata, 18

# Index


## B

balltreeQueryResultToLocationManager() (*geograpy.locator.Location method*), 4

## C

check() (*tests.test_extractor.TestExtractor method*), 15
checkExamples() (*tests.test_locator.TestLocator method*), 16
cities (*geograpy.locator.LocationContext attribute*), 5
cities_for_name() (*geograpy.locator.Locator method*), 6
City (*class in geograpy.locator*), 2
CityManager (*class in geograpy.locator*), 2
correct_country_misspelling() (*geograpy.locator.Locator method*), 6
countries (*geograpy.locator.LocationContext attribute*), 5
Country (*class in geograpy.locator*), 3
country (*geograpy.locator.City attribute*), 2
country (*geograpy.locator.Region attribute*), 9
CountryManager (*class in geograpy.locator*), 3
createViews() (*geograpy.locator.Locator method*), 6

## D

db_has_data() (*geograpy.locator.Locator method*), 6
db_recordCount() (*geograpy.locator.Locator method*), 7
default (*geograpy.labels.Labels attribute*), 2
disambiguate() (*geograpy.locator.Locator method*), 7
distance() (*geograpy.locator.Location method*), 4
downloadBackupFile() (*geograpy.locator.LocationManager static method*), 5

## E

Earth (*class in geograpy.locator*), 4
Extractor (*class in geograpy.extraction*), 1

## F

find_entities() (*geograpy.extraction.Extractor method*), 1
find_geoEntities() (*geograpy.extraction.Extractor method*), 1
from_sqlDb() (*geograpy.locator.CountryManager class method*), 4
from_sqlDb() (*geograpy.locator.RegionManager class method*), 9
fromCache() (*geograpy.locator.LocationContext class method*), 5
fromErdem() (*geograpy.locator.CountryManager class method*), 3
fromGeoLite2() (*geograpy.locator.City static method*), 2
fromGeoLite2() (*geograpy.locator.Country static method*), 3
fromGeoLite2() (*geograpy.locator.Region static method*), 9
fromJSONBackup() (*geograpy.locator.CityManager class method*), 2
fromJSONBackup() (*geograpy.locator.CountryManager class method*), 3
fromJSONBackup() (*geograpy.locator.LocationContext class method*), 5
fromJSONBackup() (*geograpy.locator.RegionManager class method*), 9
fromPyCountry() (*geograpy.locator.Country static method*), 3
fromWikidata() (*geograpy.locator.CityManager class method*), 3
fromWikidata() (*geograpy.locator.CountryManager class method*), 3
fromWikidata() (*geograpy.locator.Region static method*), 9
fromWikidata() (*geograpy.locator.RegionManager*

*class method*), 9
fuzzy_match() (*in module geograpy.utils*), 10

## G

geo (*geograpy.labels.Labels attribute*), 2
geograpy (*module*), 12
geograpy.extraction (*module*), 1
geograpy.labels (*module*), 1
geograpy.locator (*module*), 2
geograpy.places (*module*), 10
geograpy.utils (*module*), 10
geograpy.wikidata (*module*), 10
get_geoPlace_context() (*in module geograpy*), 12
get_place_context() (*in module geograpy*), 12
get_region_names() (*geograpy.places.PlaceContext method*), 10
getAliases() (*geograpy.locator.Locator method*), 7
getBackupDirectory() (*geograpy.locator.LocationManager static method*), 6
getBallTuple() (*geograpy.locator.LocationManager method*), 6
getByName() (*geograpy.locator.LocationManager method*), 6
getCities() (*geograpy.locator.LocationContext method*), 5
getCities() (*geograpy.wikidata.Wikidata method*), 11
getCitiesOfRegion() (*geograpy.wikidata.Wikidata method*), 11
getCityPopulations() (*geograpy.wikidata.Wikidata method*), 11
getCoordinateComponents() (*geograpy.wikidata.Wikidata static method*), 11
getCountries() (*geograpy.locator.LocationContext method*), 5
getCountries() (*geograpy.wikidata.Wikidata method*), 11
getCountry() (*geograpy.locator.Locator method*), 7
getDefaultConfig() (*geograpy.locator.LocationContext static method*), 5
getFileContent() (*geograpy.locator.LocationManager static method*), 6
getGeolite2Cities() (*geograpy.locator.Locator method*), 7
getInstance() (*geograpy.locator.Locator static method*), 7
getLocationByID() (*geograpy.locator.LocationManager method*), 6
getLocationLodFromJsonBackup() (*geograpy.locator.CityManager class method*), 3
getLocationLodFromJsonBackup() (*geograpy.locator.CountryManager class method*), 4
getLocationLodFromJsonBackup() (*geograpy.locator.RegionManager class method*), 10
getLocationsWithinRadius() (*geograpy.locator.Location method*), 4
getNClosestLocations() (*geograpy.locator.Location method*), 4
getRegions() (*geograpy.locator.LocationContext method*), 5
getRegions() (*geograpy.wikidata.Wikidata method*), 11
getSamples() (*geograpy.locator.City class method*), 2
getSamples() (*geograpy.locator.Country class method*), 3
getSamples() (*geograpy.locator.Location class method*), 4
getSamples() (*geograpy.locator.Region class method*), 9
getURLContent() (*geograpy.locator.LocationManager static method*), 6
getValuesClause() (*geograpy.wikidata.Wikidata static method*), 11
getView() (*geograpy.locator.Locator method*), 7
getWikidataCityPopulation() (*geograpy.locator.Locator method*), 7
getWikidataId() (*geograpy.wikidata.Wikidata static method*), 11

## H

haversine() (*geograpy.locator.Location static method*), 5

## I

interlinkLocations() (*geograpy.locator.LocationContext method*), 5
is_a_country() (*geograpy.locator.Locator method*), 8
isISO() (*geograpy.locator.Locator static method*), 8
isKnownAs() (*geograpy.locator.Location method*), 5

## L

Labels (*class in geograpy.labels*), 2
locateCity() (*geograpy.locator.Locator method*), 8
locateCity() (*in module geograpy*), 12

locateLocation() (*geograpy.locator.LocationContext method*), 5
Location (*class in geograpy.locator*), 4
LocationContext (*class in geograpy.locator*), 5
LocationManager (*class in geograpy.locator*), 5
Locator (*class in geograpy.locator*), 6
locator (*geograpy.locator.Locator attribute*), 8

## M

main() (*in module geograpy.locator*), 10

## P

PlaceContext (*class in geograpy.places*), 10
places_by_name() (*geograpy.locator.Locator method*), 8
populate_Cities() (*geograpy.locator.Locator method*), 8
populate_Cities_FromWikidata() (*geograpy.locator.Locator method*), 8
populate_Countries() (*geograpy.locator.Locator method*), 8
populate_db() (*geograpy.locator.Locator method*), 9
populate_Regions() (*geograpy.locator.Locator method*), 8
populate_Version() (*geograpy.locator.Locator method*), 8
populateFromWikidata() (*geograpy.locator.Locator method*), 8

## R

radius (*geograpy.locator.Earth attribute*), 4
readCSV() (*geograpy.locator.Locator method*), 9
recreateDatabase() (*geograpy.locator.Locator method*), 9
Region (*class in geograpy.locator*), 9
region (*geograpy.locator.City attribute*), 2
RegionManager (*class in geograpy.locator*), 9
regions (*geograpy.locator.LocationContext attribute*), 5
regions_for_name() (*geograpy.locator.Locator method*), 9
remove_non_ascii() (*in module geograpy.utils*), 10
resetInstance() (*geograpy.locator.Locator static method*), 9

## S

set_cities() (*geograpy.places.PlaceContext method*), 10
set_countries() (*geograpy.places.PlaceContext method*), 10
set_other() (*geograpy.places.PlaceContext method*), 10
set_regions() (*geograpy.places.PlaceContext method*), 10
set_text() (*geograpy.extraction.Extractor method*), 1
setAll() (*geograpy.places.PlaceContext method*), 10
setUp() (*tests.test_extractor.TestExtractor method*), 15
setUp() (*tests.test_locator.TestLocator method*), 16
setUp() (*tests.test_places.TestPlaces method*), 17
setUp() (*tests.test_wikidata.TestWikidata method*), 18
setValue() (*geograpy.locator.City method*), 2
split() (*geograpy.extraction.Extractor method*), 1

## T

tearDown() (*tests.test_extractor.TestExtractor method*), 15
tearDown() (*tests.test_locator.TestLocator method*), 16
tearDown() (*tests.test_places.TestPlaces method*), 17
tearDown() (*tests.test_wikidata.TestWikidata method*), 18
testDelimiters() (*tests.test_locator.TestLocator method*), 16
testExamples() (*tests.test_locator.TestLocator method*), 16
TestExtractor (*class in tests.test_extractor*), 15
testExtractorFromText() (*tests.test_extractor.TestExtractor method*), 15
testExtractorFromUrl() (*tests.test_extractor.TestExtractor method*), 15
testGeograpyIssue32() (*tests.test_extractor.TestExtractor method*), 15
testGeolite2Cities() (*tests.test_locator.TestLocator method*), 16
testGetCitiesOfRegion() (*tests.test_wikidata.TestWikidata method*), 18
testGetCoordinateComponents() (*tests.test_wikidata.TestWikidata method*), 18
testGetGeoPlace() (*tests.test_extractor.TestExtractor method*), 15
testGetWikidataId() (*tests.test_wikidata.TestWikidata method*), 18
testHasData() (*tests.test_locator.TestLocator method*), 16
testIsoRegexp() (*tests.test_locator.TestLocator method*), 16
testIssue10() (*tests.test_extractor.TestExtractor method*), 15

testIssue15() (*tests.test_locator.TestLocator method*), 17
testIssue17() (*tests.test_locator.TestLocator method*), 17
testIssue19() (*tests.test_locator.TestLocator method*), 17
testIssue22() (*tests.test_locator.TestLocator method*), 17
testIssue41_CountriesFromErdem() (*tests.test_locator.TestLocator method*), 17
testIssue7() (*tests.test_extractor.TestExtractor method*), 16
testIssue9() (*tests.test_extractor.TestExtractor method*), 16
testIssue_42_distance() (*tests.test_locator.TestLocator method*), 17
TestLocator (*class in tests.test_locator*), 16
testLocatorWithWikiData() (*tests.test_wikidata.TestWikidata method*), 18
TestPlaces (*class in tests.test_places*), 17
testPlaces() (*tests.test_places.TestPlaces method*), 17
testPopulation() (*tests.test_locator.TestLocator method*), 17
testProceedingsExample() (*tests.test_locator.TestLocator method*), 17
tests (*module*), 18
tests.test_extractor (*module*), 15
tests.test_locator (*module*), 16
tests.test_places (*module*), 17
tests.test_wikidata (*module*), 18
testStackoverflow43322567() (*tests.test_extractor.TestExtractor method*), 16
testStackoverflow54077973() (*tests.test_extractor.TestExtractor method*), 16
testStackoverflow54712198() (*tests.test_extractor.TestExtractor method*), 16
testStackOverflow54721435() (*tests.test_extractor.TestExtractor method*), 16
testStackoverflow55548116() (*tests.test_extractor.TestExtractor method*), 16
testStackoverflow62152428() (*tests.test_extractor.TestExtractor method*), 16
testStackOverflow64379688() (*tests.test_locator.TestLocator method*), 17
testStackOverflow64418919() (*tests.test_locator.TestLocator method*), 17
TestWikidata (*class in tests.test_wikidata*), 18
testWikidataCities() (*tests.test_wikidata.TestWikidata method*), 18
testWikidataCountries() (*tests.test_wikidata.TestWikidata method*), 18
testWordCount() (*tests.test_locator.TestLocator method*), 17

## U

updateCity() (*geograpy.locator.CityManager method*), 3

## W

Wikidata (*class in geograpy.wikidata*), 10