OSMNames User Documentation

Release 2.0

Aug 03, 2017

Contents

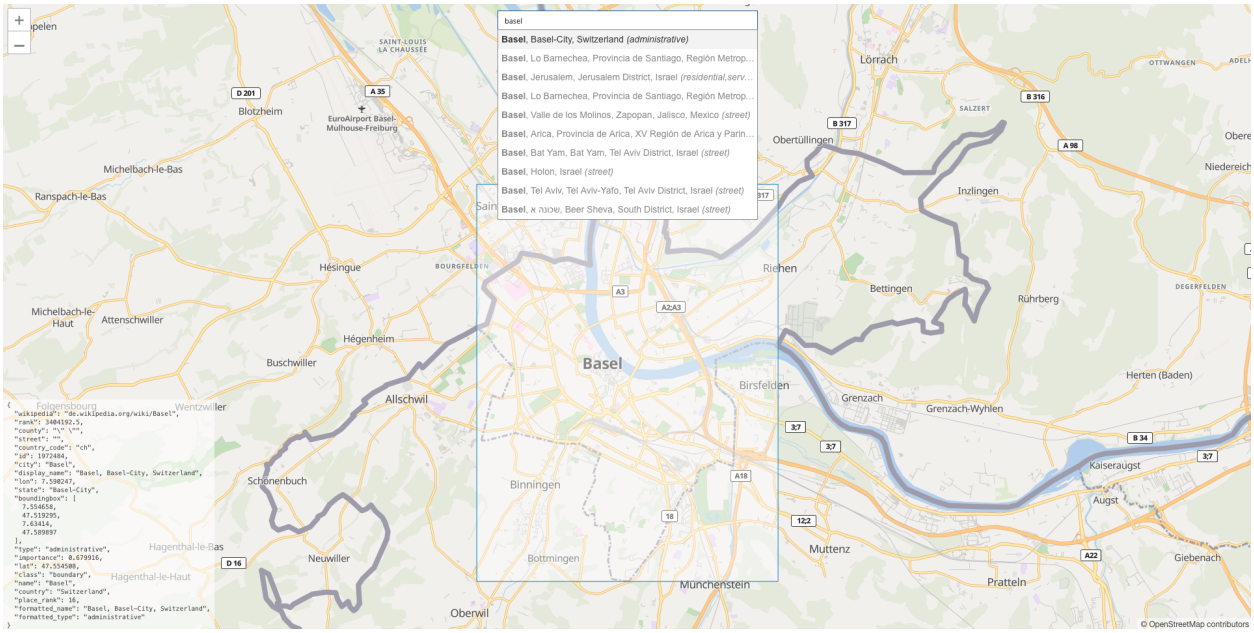
CHAPTER 1

Introduction

OSMNames is an open source tool that allows creating geographical gazetteer data out of OpenStreetMap OSM files.

There is a need for a data set consisting of street names of the world. Such gazetteer data, however, is either not available for every country (openaddresses.io) or is not in a suitable format. Furthermore, if such data exists, it is often not for free. A global data set can be downloaded at https://osmnames.org.

A current implementation on how the data looks like in a geocoder is a available at https://osmnames.org



What can I do with OSMNames?

With OSMNames, you can create your own geocoder data set based on OpenStreetMap. It currently includes all addresses available. For each feature, the hierarchy, as well as a Wikipedia-based importance, is calculated.

Where to Start?

To download the newest set of data go to https://osmnames.org.

To process OpenStreetMap data yourself, check out the Getting Started document.

If you want to have a look at the Source Code or contribute to the project, check out the Development documentation. The source code is available in our GitHub Repository.

Output Format

The exported file *geonames.tsv* contains the following columns:

Column name	Description
name	The name of the feature (default language is en, others available are de, es, fr, ru, zh)
alternative_names	All other available and distinct names separated by commas
osm_type	The OSM type of the feature (node, way, relation)
osm_id	The unique osm_id as identifier for the house numbers in the second file *housenumbers.tsv*
class	The class of the feature e.g. boundary
type	The type of the feature e.g. administrative
lon	The decimal degrees (WGS84) longitude of the centroid of the feature
lat	The decimal degrees (WGS84) latitude of the centroid of the feature
place_rank	Rank from 1-30 ascending, 1 being the highest. Calculated with the type and class of the feature.
importance	Importance of the feature, ranging [0.0-1.0], 1.0 being the most important.
street	The name of the street if the feature is some kind of street
city	The name of the city of the feature, if it has one
county	The name of the county of the feature, if it has one
state	The name of the state of the feature, it it has one
country	The name of the country of the feature
country_code	The ISO-3166 2-letter country code of the feature
display_name	The display name of the feature representing the hierarchy, if available in English
west	The western decimal degrees (WGS84) longitude of the bounding box of the feature
south	The southern decimal degrees (WGS84) latitude of the bounding box of the feature
east	The eastern decimal degrees (WGS84) longitude of the bounding box of the feature
north	The northern decimal degrees (WGS84) latitude of the bounding box of the feature
wikidata	The wikidata associated with the feature
wikipedia	The wikipedia URL associated with the feature
housenumbers	All house numbers, comma separated, associated to this element. Coordinates of the house numbers are part of the second output file *housenumbers.tsv*

Note: All coordinates are rounded to seven digits after the decimal point.

Note: The *housenumbers* column is a redundant information of all house numbers contained in the file *housenumbers.tsv*. The redundancy is accepted due to advantages for the full-text search of geocoders.

The second file *housenumber.tsv* contains the following columns:

Column name	Description
osm_id	The unique osm_id for debug purposes
street_id	The osm_id of the element, the house number is associated to
street	The name of the street it is associated to for debug purposes
housenumber	The actual house number
lon	The decimal degrees (WGS84) longitude of the centroid of the house number
lat	The decimal degrees (WGS84) latitude of the centroid of the house number

CHAPTER 2

Getting Started

System requirements

With the following set of commands one can easily setup OSMNames in a matter of minutes. Prerequisites are a working installation of Docker https://www.docker.com/ along with Docker Compose.

Note: In order to increase the speed of the processing, an SSD disk is recommended. It is also recommended to tweak the database settings to match the specficiations of your system.

Run OSMNames

To run OSMNames, follow these steps:

1. Checkout source from GitHub

```
git clone https://github.com/OSMNames/OSMNames.git
```

2. Specify the URL to the PBF file in the *.env* file

```
PBF_FILE_URL=http://download.geofabrik.de/europe/switzerland-latest.osm.
↪pbf
```

   Alternatively place a custom PBF file in the *data/import* directory and define it in the *.env* file

```
PBF_FILE=Zuerich.osm.pbf
```

   If *PBF_FILE* is defined *PBF_FILE_URL* will be ignored.

3. **Now run OSMNames**

```
docker-compose run --rm osmnames
```

This command will:

1. Initialize the database inside a docker container
2. Download and import a wikipedia dump
3. Download and process the specified PBF file
4. Export the OSMNames data

The export files for example, *switzerland_geonames.tsv.gz* and *switzerland_housenumbers.tsv.gz* can be found in the export directory *data/export*.

A more detailed and technical overview can be found in the documentation about the Implementation.

Note: The execution time is highly dependent from the size of the PBF file and the available hardware. More details about the performance can be found in the corresponding documentation.

Extracting countries

The TSV file from the planet export includes more than 21‘000‘000 entries. The current data export can be downloaded at https://osmnames.org. If one is only interested in a specific country, download the file and extract the information with the following command:

```
awk -F $'\t' 'BEGIN {OFS = FS}{if (NR!=1) {if ($16 =="[country_code]")
↪{print}} else {print}}' planet-latest.tsv > countryExtract.tsv
```

where [country_code] needs to be replaced with the ISO-3166 2-letter country code.

CHAPTER 3

Components

OSMNames consists of the following components:

Docker

OSMNames is built with Docker and is therefore shipped in containers. This allows to have an extra layer of abstraction and avoids overhead of a real virtual machine. Specifically, it is built with docker-compose thus allowing to define a multi-container architecture defined in a single file.

Imposm3

Imposm3 by Omniscale is a data importer for OpenStreetMap data. It reads PBF files and writes the data into the PostgreSQL database. In OSMNames it is used in favor of osm2pgsql mainly because of its superior speed results. It makes heavy use of parallel processing favoring multicore systems. Explicit tag filters are set in order to have only the relevant data imported.

PostgreSQL

PostgreSQL is the open source database powering OMSNames.

OSMNames uses PostgreSQL for the following tasks:

- Storing OSM data read from PBF file.
- OSM data processing
- data export to TSV file

At this moment OSMNames runs PostgreSQL 9.6.x version.

PostGIS

PostGIS is the extension which adds spatial capabilities to PostgreSQL. It allows working with geospatial types or running geospatial functions in PostgreSQL.

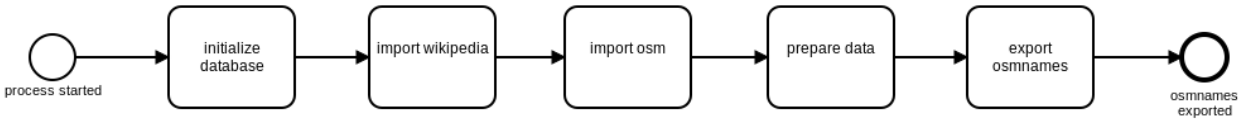
At this moment OSMNames runs PostGIS 2.3 version.

CHAPTER 4

Implementation

This document describes the implementational aspects of OSMNames.

OSMNames is written in Python. Whereas the main entry point is the file run.py. The script calls the necessary tasks in the correct order. The following diagram shows the full process of OSMNames:



Details about the tasks can be found in the particular documents:

Initialize Database

The initialization of the database is skipped, if it is already present.

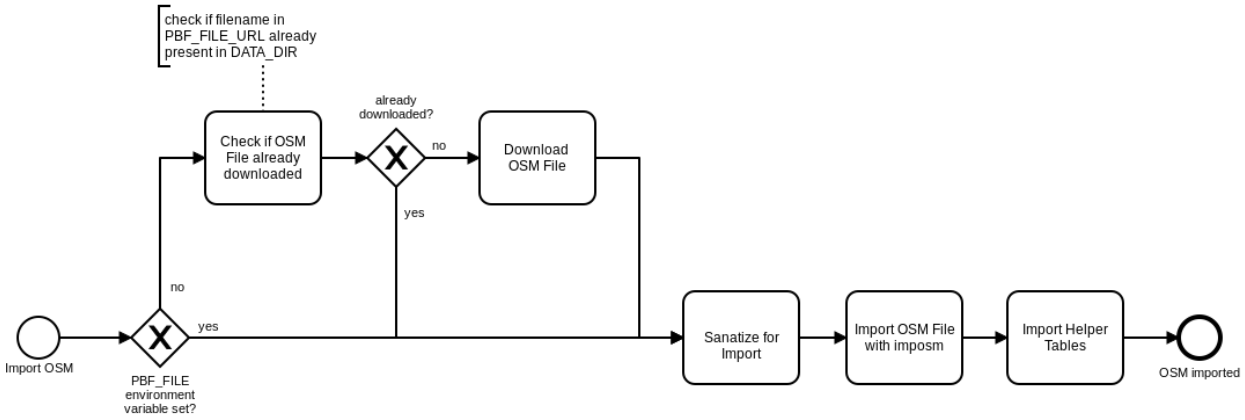
The database is created with the template *template_postgis*. The user and password are set via the environment variables *DB_USER* and *DB_PASSWORD*. The default values are *osm* and *password*.

Import Wikipedia

To have an importance value for each feature, a wikipedia helper table is downloaded from a Nominatim server. This is the same information Nominatim uses to determine the importance. It was decided to take this pre-calculated data instead of calculating it due to longer processing times (up to several days!). Also, the same calculations are applied, to achieve the same results. The initialization of the database is skipped, if it is already present.

The download and import of the wikipedia dump are skipped, if it is already present. Since the dump was created with the username *brian*, a temporary user is created to restore the dump, which is dropped after transferring the ownership to the *osm* user.

Import OSM



The PBF file can be set with the environment variable *PBF_FILE_URL* or *PBF_FILE*. When defining the URL, the file is download, if not already present in the import directory. When the file is defined directly, the download is skipped.

Before importing the PBF file with Imposm, the database is sanitized by dropping all previously imported tables.

To import the PBF file Imposm3 is used, which is an importer for OpenStreetMap data. The corresponding mapping can be found here. After the import, the following tables will be created:

- osm_linestring
- osm_polygon
- osm_point
- osm_housenumber
- osm_relation
- osm_relation_member

More details about the columns of the tables can be found in the mapping of Impsom3. Additionally, will the tables be extended with custom columns when preparing the data.

Import Helper Tables

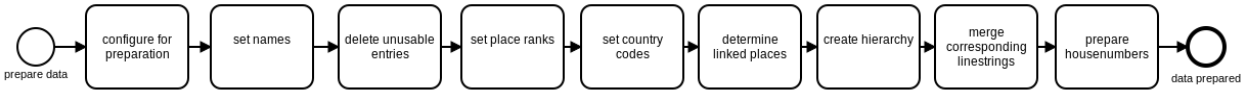
Besides the OpenStreetMap data, are the following tables imported:

Table	Description
country_osm_grid	Contains the country code and geometries for all countries.
country_name	Contains the country code and country names of all countries.

The tables are later used to enrich the imported data. Both are provided by Nominatim.

Prepare Data

The preparation of the imported OpenStreetMap data for the export is the heart of OSMNames. Missing names are completed, a hierarchy is created, unusable entries are removed and more. In this document are all involved steps explained in detail. The following diagram shows the full process of preparing the data:



configure for preparation

This step configures the database for the other steps. This involves:

- Dropping unused indexes for better performance
- Add custom columns, necessary for the preparation, to tables imported in *import_osm*. The added columns can be found here.
- Set tables to unlogged for better performance

set names

The following approaches are used to complete the name and alternative_names attribute on polygons, linestrings and points.

set names from tags

All tags of polygons, linestrings and points imported. On some elements is the name not set with the key *name* but with a different key, e.g. *name:en*. The value of the name attribute is tried to set with following approaches, whereas the order matches the priority:

1. Set the name to the imported name if present.
2. **Set the name to the first present value of these keys, whereas the order matches the priority:**
   (a) *name:en*
   (b) *name:fr*
   (c) *name:de*
   (d) *name:es*
   (e) *name:ru*
   (f) *name:zh*
3. If still no name is found, take the first alternative name.

Additionally is the attribute *alternative_names* set with all available names, except the value of the name attribute. The value of *alternative_names* is a comma separated string.

Note: All available names for the alternative names are determined by the keys of the tags. Keys starting with *name:* and others are considered. Details about the relevant keys can be found in the corresponding query.

Note: Tabs in the name or alternative_names are replaced with spaces, since the final export format is TSV.

Example

A node was imported with following attributes:

Attribute	Value
name	NULL
all_tags	{ “name:de”: “Matterhorn”, “name:fr”: “Cervin”, “name:it”: “Cervino” }

After running set_names_from_tags, the following values are set:

Attribute	Value	Explanation
name	Cervin	The French name from all_tags because the name attribute was empty and French has a higher priority then German
alterna-tive_names	Matterhorn, Cervino	All remaining names from all_tags, except the French, since it was set as name

set linestring names from relations

Sometimes is the name not set on a linestring directly, but on the relation, where the linestring is a member. If so, the name is set to the name of the relation.

Implemented with Issue #106.

delete unusable entries

Elements are unusable and deleted if:

- Name attribute of polygons, points or linestrings is still empty.
- Geometry of polygons is empty.

set place ranks

The place rank indicates how important a element is (lower means more important). A continent for example has a place_rank of 2, which is the lowest place_rank possible. The place_rank is either the double of the admin_level, if the admin_level is set, or a value depending on the type of the element. The mapping can be found here.

set country codes

To determine the country of a element, the country_code must be present on each polygon. It is only necessary for polygons since the country code of all other elements can be determined based on the hierarchically associated polygon.

If present the imported country_code is taken. Otherwise is the country code set based on the country_osm_grid.

determine linked places

In order to determine linked places (points linked with polygons) additional tags about the relations are imported. Specifically, the role values admin_centre and label are of interest.

This information is later on used in the export mainly to rule out point features linked to their polygon features as well as determining city types instead of administrative types.

For example the relation Kreuzberg is linked to the member node Kreuzberg with the role *label*. Since they are linked, only the polygon will be exported.

create hierarchy

The hierarchy of the elements is created based on their geometries. The process is as simple as this:

1. Set the parent id of each element within a polygon, with the place rank 22, to the id of the polygon. Polygons with the place rank 22 have the admin level 11 or the type *neighbourhood* or *residential*.

Note: The parent id of a polygon is only set if the place rank is higher than the place rank of the parent. This prevents a meaningless hierarchy.

---

2. When all polygons with the place rank 22 are processed, the same step is done with all polygons with the place rank 21, 20, 19 and so forth.
3. It ends with the place rank 2, which corresponds to polygons of the type *continent*.

Note: If a element is contained in a polygon, is determined with the PostGIS function st_contains. Since it only returns true if a geometry is fully contained in another geometry, the child elements are determined only with the center of a geometry and not the full geometry. The centers of geometries are set here.

Note: Polygons of the type *water*, *desert*, *bay* and *reservoir* are ignored, since it makes no sense to assign them as parents of other elements.

---

merge corresponding linestrings

Linestrings are merged to one linestring if all of these conditions are met:

- They have the same name
- They have the same polygon as parent
- They are at least 1000 meters near each other

When merging the linestring a new table *osm_merged_linestring* is created, which contains, besides the shared attributes of the sub-linestrings, following attributes:

Attribute	Description
osm_id	Smallest id of the sub-linestring ids.
member_ids	The ids of the sub-linestrings.
type	Types of the sub-linestrings, comma separated.
geometry	Combination of the sub-linestring geometries.

Note: The geometry of the merged linestring is sligthly simplified with the PostGIS function st_simplify, see Issue #90

After creating the table *osm_merged_linestring*, the attribute *merged_into* of the original linestrings in the table *osm_linestring* are updated to the *osm_id* of the linestring they have been merged into.

Examples

For example the linestrings with the OSM IDs 26085954, 289620118, 289620119 are merged to one linestring.

Other examples can be found in the issues #74 and #85.

prepare housenumbers

The goal of preparing the house numbers is, to connect each geometry, which has an house number as attribute, to a corresponding street or place. All geometries with an house number are imported into the *osm_housenumber* table. Some of them have already the *street* attribute set, with the name of a street. Others do only have the *housenumber* attribute and nothing else set. For these house numbers multiple approaches are applied to complete the missing *street* attributes. The steps are shown by the following diagram:

Note: The individual steps are sorted according to their costs. It is for example fast to determine the missing street attribute from a relation, if one exists. But it is slow and costly to find the nearest street depending on the geometry.

set street attributes by street relation members

If a house number is part of a relation, where another member has the role *street* or *associatedStreet*, set the *street_id* and the *street* to the *osm_id* and *name* of this member.

set street names by relation attributes

If a house number is part of a relation with the type *street* or *associatedStreet*, set the *street* to the *street* or *name* attribute of this relation.

normalize street names

To match house numbers with streets by the street name, the attributes *normalized_street* and *normalized_name* of house numbers and linestrings are set to a normalized version of the street and name. The name is normalized by:

- removing all white spaces and dashes
- lower casing the name
- removing accents

Some examples for normalized names and streets:

Name / Street	Normalized Name / Street
Bietinger Weg	bietingerweg
Cité Préville	citepreville
Chemin du Pra-de-Villars	chemindupradevillars
Rue de'Gare	ruedegare

set street ids by street name

It is tried to set the *street_id* of the house numbers to the *osm_id* of a linestring, which has the same *parent_id* and a matching name. These approaches are executed in the given order:

1. Find a linestring with **the same parent_id** and the **exactly** same *name* as the *street* of the house number.
2. Find a **within 1000 meters** and the **exactly** same *name* as the *street* of the house number.
3. Find a linestring with **the same parent_id** and the **most similar** *name*. This approach makes use of the PostgreSQL module pg_trgm.
4. Find a **within 1000 meters** and the **most similar** *name*. This approach makes use of the PostgreSQL module pg_trgm.

Note: The approaches are executed in this order because the more accurate and best performing approaches are executed first. If still no street was found, the restrictions are softened.

Here some examples for the matching street names. Note that in the queries the matching is done with the normalized name.

House number street	Linestring name
Haldenweg	Haldenweg
Bochslenrasse	Bochslenstrasse
Cité Préville 19	Cité Préville

set street attributes by nearest street

Still not all house numbers will have a street assigned at this point. As the last approach will the **nearest** street be assigned to the house number. Note that this is **very slow, expensive and inaccurate** and therefore is only executed if no street was found with the previous approaches.

Export OSMNames

When exporting OSMNames the output files get created. This documents describes the implementation of the export. Details about the output format can be found in the introduction.

create functions

This step creates the SQL functions later used for the export.

Besides the following descriptions of the functions are the unit tests of Python a good entry point to understand how the functions work.

determine_class

Returns a class for a given type. For example, the type *city* leads to the class *place*.

The full mapping can be found in the code.

get_parent_info

This function makes use of the hierarchy and the place rank to return the following information for an element:

- city
- county
- state
- country_code
- display name

Whereas the display name is a concatenation of the name of the element and all other information.

Note: More information about the impelementation of the function can be found in the PR #82

Example

These elements exists:

Type	ID	Name	Parent ID
Linestring	1	Oberseestrasse	2
Polygon	2	Rapperswil-Jona	3
Polygon	3	Wahlkreis See-Gaster	4
Polygon	4	Sankt Gallen	5
Polygon	5	Schweiz	-

When calling the function *get_parent_info* with the parent id and the name of linestring *Oberseestrasse* following information will be returned:

Attribute	Value
city	Rapperswil-Jona
county	Wahlkreis See-Gaster
state	Sankt Gallen
country_code	ch
display name	Oberseestrasse, Rapperswil-Jona, Wahlkreis See-Gaster, Sankt Gallen, Switzerland

Note: The decision which polygon is the city, county or state is based on the corresponding place rank.

get_country_name

Returns the name of a country for a given country code. The name will be returned in the first language present, following the precedence: [English -> native name -> French -> German -> Spanish -> Russian -> Chinese].

The names are read from the helper table *country_name* (see *Import Helper Tables*).

get_importance

This function returns an importance for an element by its URL to a wikipedia article if present or its place rank.

If a feature has a wikipedia URL a matching entry in the wikipedia helper table is taken for calculating the importance with the following formula:

```
importance = log (totalcount) / log( max(totalcount))
```

where totalcount is the number of references to the article from other wikipedia articles. In case there is no wikipedia information or no match was found, the following formula is applied:

```
importance = 0.75 - (place_rank/40)
```

Since every feature has a rank, it is ensured that every feature also has an importance.

get_country_language_code

Returns the default language for a country. The value is read from the helper table *country_name* (see *Import Helper Tables*).

get_housenumbers

Returns a comma separated string of all house numbers, associated to the given *osm_id*.

get_bounding_box

This functions takes a geometry, a country code and an admin_level as attribute and determines a bounding box. It is only used for polygons to handle these special cases:

- Some countries do have colonies where are big bounding box is returned. Since this is inconvenient from a user perspective, a smaller bounding box, only covering the main country is returned. See Issue #57 for more details.
- When a polygons intersects the antimeridian, a unintuitive bounding box is returned. In this case the bounding box is shifted manually. See Issue #94 for more details.

create views

This function creates the views, which are later used to export the geonames and house numbers. The columns of the views equals the output format of OSMNames.

export geonames

This function exports all rows of the polygon, linestring and point view to the file *<import-file-name>_geonames.tsv*. This by making use of the PostgreSQL function COPY.

export housenumbers

This function exports all rows of house number view to the file *<import-file-name>_housenumbers.tsv*. This by making use of the PostgreSQL function COPY.

Note: House numbers unable to associated to a street or place when preparing the data, are not exported.

---

gzip tsv files

This function finally uses gzip to compress the *tsv* files created before.

CHAPTER 5

Development

Contributing / Issues

If you like to contribute feel free to create an issue on the OSMNames GitHub repository. It is optimal if the issue description includes some real examples, like OSM IDs of existing OpenStreetMap elements. Additionally should each new functionality or bugfix be covered by a new test case (see *Testing*).

Keep in mind that the following styleguides should be respected:

- *PEP8 <https://www.python.org/dev/peps/pep-0008/>_* for Python code
- *SQL Style Guide <http://www.sqlstyle.guide/>_* for SQL

Testing

To have a sustainable code base, tests are indispensable. OSMNames uses the Python testing framework pytest for testing.

The tests run inside a docker container and uses the same docker container for the database, as the main process of OSMNames. To run the tests, following command can be executed:

```
docker-compose run --rm osmnames bash run_tests.sh
```

This executes the script *run_tests.sh* inside the docker container.

Alternatively can a path be added as argument to execute a specific test:

```
docker-compose run --rm osmnames bash run_tests.sh tests/prepare_data/test_delete_
↪unusable_entries.py
```

Some important notes about the architecture of the tests:

- The tests can be found in the directory *tests/*

- The name of the Python test files and the name of the functions must have the prefix *test_* to be executed by pytest.
- When including the pytest fixture *session* in a test method, the test database is dropped and recreated before the test. The fixture is defined here.
- A good way to structure a test, is to import a SQL dump with the necessary schema, after the database was recreated by the *session* fixture.
- To create rows in Python code, the helper class Tables can be used.

Example for a Test

The following code tests the functionality of the function *delete_unusable_entries*. For better understanding are some parts of the file *conftest.py* also listed.

tests/conftest.py:

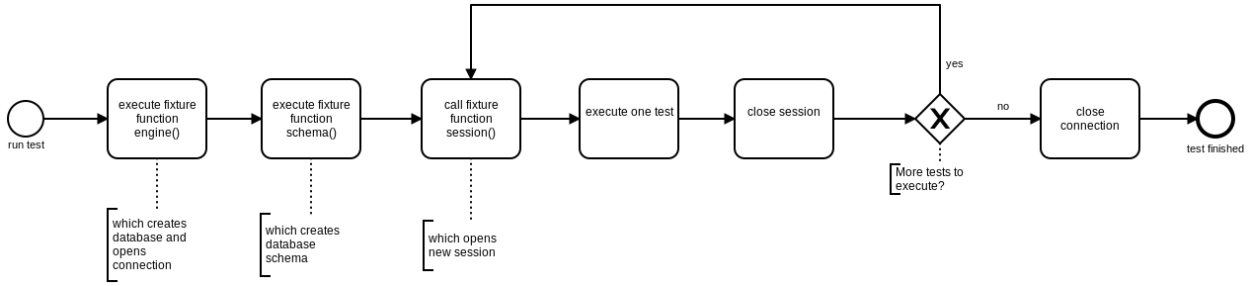
```
# ...

@pytest.fixture(scope="module")
def engine():
    wait_for_database()
    _recreate_database()

    yield connection.engine

    connection.engine.dispose()


@pytest.fixture(scope="function")
def session(engine):
    session = Session(engine)

    yield session

    session.close()


@pytest.fixture(scope="module")
def tables(engine):
    return Tables(engine)

# ...
```

tests/prepare_data/test_delete_unusuable_entries.py:

```
# ...

@pytest.fixture(scope="module")
def schema():
    current_directory = os.path.dirname(os.path.realpath(__file__))
    exec_sql_from_file('fixtures/test_prepare_imported_data.sql.dump', cwd=current_
↪directory)


def test_osm_polygon_with_blank_names_get_deleted(session, schema, tables):
    session.add(tables.osm_polygon(name="gugus"))
    session.add(tables.osm_polygon(name=""))
    session.commit()
```

```
    delete_unusable_entries()

    assert session.query(tables.osm_polygon).count(), 1


def test_osm_polygon_with_null_names_get_deleted(session, schema, tables):
    session.add(tables.osm_polygon(name="gugus"))
    session.add(tables.osm_polygon())
    session.commit()

    delete_unusable_entries()

    assert session.query(tables.osm_polygon).count(), 1

# ...
```

The method *test_osm_polygon_with_blank_names_get_deleted* includes the fixtures session, schema and tables. The fixture *engine* is also included indirectly, since the fixture *session* in *conftest* includes it. The fixture *schema* will be executed after the database was recreated and restores the SQL dump *fixtures/test_prepare_imported_data.sql.dump* which contains relevant database schema for the test. The following diagram visualizes this process:



Note: Since the fixture *engine* and *schema* are in the scope *module* they are only executed once per file and not for each test.

Logging

To analyze the progress of OSMNames multiple ways of logging are available.

Python Logs

To write logging messages from Python code, a logger can be used, which is implemented here. It makes use of the logging facility of Python. It can be defined and called like this:

```
log = logger.setup(__name__)

#...
def some_method():
  log.debug('some method called')
  #...
  log.error('some method failed')
```

The log entries are sent to the default output and to a log file inside the directory *data/logs/*.

Python Profiling

Besides the logger is also the profiling facility of Python used. In the file *run.py* is the profiler started at the beginning and the statistics are written after the whole process. This results in a file with the suffix *.cprofile* in the directory *data/logs*. It contains statistics how often and for how long various parts of the program have been executed.

A simple way to look at these data is the tool RunSnakeRun, which results in a GUI like this:

PostgreSQL Logs

The simplest way to have a look at the log files of PostgreSQL is by using the logging capabilities of docker-compose. The following command follows the log files of PostgreSQL:

```
docker-compose logs -f -t postgres
```

Consistency Checks

Consistency checks do some checking while processing the data to get a feedback how well the preparation is working. For example, after running the function *set_parent_ids*, when creating the hierarchy, a consistency check writes to the log how many elements still have no parent id set. This could be because of a wrong functionality or invalid input from OpenStreetMap (e.g. missing attributes which should be set, invalid geometries, spelling mistakes and so forth).

The consistency checks are defined here and called at the relevant position in the code.

Tips

These tips may help for efficient development:

- Use a small PBF file, for example your hometown, to test the your changes locally by running the full process.
- OSMNames vacuums the Postgres database a lot. This only makes sense when processing a large PBF file. When running a small PBF file the environment variable *SKIP_VACUUM* can be set to *True* in the *.env* file.
- When working with a small file in development, one can forget about the performance influences for large files easily. Some minutes more for small files can lead to a increased runtime of multiple hours for the whole planet.

CHAPTER 6

Others

Performance

The following tips can help to improve the performance for processing large PBF files with OSMNames.

Database Configuration

For better performance, the database needs to be configured according to the resources of the host system, the process runs on. A custom configuration can be added by creating a file */docker-entrypoint-initdb.d/alter_system.sh* inside the postgres container and marking it as executable. The script is executed when restarting the database container.

Here is an example for the content of the script:

```
#!/bin/bash
set -o errexit
set -o pipefail
set -o nounset

function alter_system() {
    echo "Altering System parameters"
    PGUSER="$POSTGRES_USER" psql --dbname="$POSTGRES_DB" <<-EOSQL
    alter system set autovacuum_work_mem = '4GB';
    alter system set checkpoint_completion_target = '0.9';
    alter system set checkpoint_timeout = '20min';
    alter system set datestyle = 'iso, mdy';
    alter system set default_statistics_target = '500';
    alter system set default_text_search_config = 'pg_catalog.english';
    alter system set dynamic_shared_memory_type = 'posix';
    alter system set effective_cache_size = '96GB';
    alter system set fsync = 'off';
    alter system set lc_messages = 'en_US.utf8';
    alter system set lc_monetary = 'en_US.utf8';
    alter system set lc_numeric = 'en_US.utf8';
    alter system set lc_time = 'en_US.utf8';
```

```
    alter system set listen_addresses = '*';
    alter system set log_checkpoints = 'on';
    alter system set log_temp_files = '1MB';
    alter system set log_timezone = 'UTC';
    alter system set maintenance_work_mem = '96GB';
    alter system set max_connections = '20';
    alter system set random_page_cost = '1.1';
    alter system set shared_buffers = '96GB';
    alter system set synchronous_commit = 'off';
    alter system set temp_buffers = '120MB';
    alter system set timezone = 'UTC';
    alter system set track_counts = 'on';
    alter system set wal_buffers = '16MB';
    alter system set max_wal_size = '5GB';
    alter system set work_mem = '6GB';
    alter system set log_statement = 'all';
EOSQL
}

alter_system
```

Determining the best configuration for a host is not easy. A good starting point for that is PgTune.

tmpfs

To improve the performance of OSMNames the database can be hold in the RAM while processing. The easiest way to do this, is by adding following line to the *docker-compose.yml* file:

```
...
postgres:
  ...
  tmpfs: /var/lib/postgresql/data:size=300G
```

This only makes sense if the necessary amount of RAM is available. Additionally keep in mind that the data will be lost when restarting the docker container.