postgis-helpers

Release 0.2.2

Jan 13, 2021

Contents:

CHAPTER 1

postgis_helpers

1.1 postgis_helpers package

1.1.1 Submodules

postgis_helpers.PgSQL module

Wrap up PostgreSQL and PostGIS into a convenient class.

Examples

Create a database and import a shapefile:

```
>>> import postgis_helpers as pGIS
>>> db = pGIS.PostgreSQL("my_database_name")
>>> db.create()
>>> db.import_geodata("bike_lanes", "http://url.to.shapefile")
>>> bike_gdf = db.query_as_geo_df("select * from bike_lanes")
```

class postgis_helpers.PgSQL.**PostgreSQL**(*working_db: str*, *un: str = 'postgres'*, *pw: str = 'password1'*, *host: str = 'localhost'*, *port: int = 5432*, *sslmode: str = None*, *super_db: str = 'postgres'*, *super_un: str = 'postgres'*, *super_pw: str = 'password2'*, *active_schema: str = 'public'*, *verbosity: str = 'full'*, *data_inbox: pathlib.Path = PosixPath('/home/docs/sql_data_io/inbox')*, *data_outbox: pathlib.Path = PosixPath('/home/docs/sql_data_io/outbox')*)

Bases: object

This class encapsulates interactions with a PostgreSQL database. It leverages psycopg2, sqlalchemy, and geoalchemy2 as needed. It stores connection information that includes:

- database name
- username & password
- host & port
- superusername & password
- the SQL cluster's master database
- `verbosity` level, which controls how much gets printed out

all_databases_on_cluster_as_list() → list
Get a list of all databases on this SQL cluster.

Returns List of all databases on the cluster

Return type list

all_spatial_tables_as_dict(*schema: str = None*) → dict
Get a dictionary of all spatial tables in the database. Return value is formatted as: `{table_name: epsg}`

Returns Dictionary with spatial table names as keys and EPSG codes as values.

Return type dict

all_tables_as_list(*schema: str = None*) → list
Get a list of all tables in the database. Optionally filter to a schema

Parameters **schema** (*str*) – name of the schema to filter by

Returns List of tables in the database

Return type list

connection_details() → dict
Return a dictionary that can be used to instantiate other database connections on the same SQL cluster.

Returns Dictionary with all of the SQL cluster connection info

Return type dict

db_create() → None
Create this database if it doesn't exist yet

db_delete() → None
Delete this database (if it exists)

db_export_pgdump_file(**args*, ***kwargs*)

db_load_pgdump_file(**args*, ***kwargs*)

execute(*query: str*, *autocommit: bool = False*)
Execute a query for a persistent result in the database. Use `autocommit=True` when creating and deleting databases.

Parameters

- **query** (*str*) – any valid SQL query string
- **autocommit** (*bool, optional*) – flag that will execute against the super db/user, defaults to False

exists() → bool
Does this database exist yet? Returns True or False

Returns True or False if the database exists on the cluster

Return type bool

export_all_shapefiles(*output_folder: pathlib.Path*) → None
Save all spatial tables in the database to shapefile.

Parameters **output_folder** (*Path*) – Folder path to write to

export_shapefile(**args*, ***kwargs*)

import_csv(**args*, ***kwargs*)

import_dataframe(*dataframe: pandas.core.frame.DataFrame*, *table_name: str*, *if_exists: str = 'fail'*, *schema: str = None*) → None
Import an in-memory `pandas.DataFrame` to the SQL database.

Enforce clean column names (without spaces, caps, or weird symbols).

Parameters

- **dataframe** (*pd.DataFrame*) – dataframe with data you want to save
- **table_name** (*str*) – name of the table that will get created
- **if_exists** (*str, optional*) – pandas argument to handle overwriting data, defaults to "fail"

import_geodata(*table_name: str*, *data_path: pathlib.Path*, *src_epsg: Union[int*, *bool] = False*, *if_exists: str = 'fail'*, *schema: str = None*)
Load geographic data into a geodataframe, then save to SQL.

Parameters

- **table_name** (*str*) – Name of the table you want to create
- **data_path** (*Path*) – Path to the data. Anything accepted by Geopandas works here.
- **src_epsg** (*Union[int, bool], optional*) – Manually declare the source EPSG if needed, defaults to False
- **if_exists** (*str, optional*) – pandas argument to handle overwriting data, defaults to "replace"

import_geodataframe(*gdf: geopandas.geodataframe.GeoDataFrame*, *table_name: str*, *src_epsg: Union[int*, *bool] = False*, *if_exists: str = 'replace'*, *schema: str = None*, *uid_col: str = 'uid'*)
Import an in-memory `geopandas.GeoDataFrame` to the SQL database.

Parameters

- **gdf** (*gpd.GeoDataFrame*) – geodataframe with data you want to save
- **table_name** (*str*) – name of the table that will get created
- **src_epsg** (*Union[int, bool], optional*) – The source EPSG code can be passed as an integer. By default this function will try to read the EPSG code directly, but some spatial data is funky and requires that you explicitly declare its projection. Defaults to False
- **if_exists** (*str, optional*) – pandas argument to handle overwriting data, defaults to "replace"

make_geotable_from_query(*query: str*, *new_table_name: str*, *geom_type: str*, *epsg: int*, *schema: str = None*, *uid_col: str = 'uid'*) → None
TODO: docstring

make_hexagon_overlay(*new_table_name: str*, *table_to_cover: str*, *desired_epsg: int*, *hexagon_size: float*, *schema: str = None*) → None

Create a new spatial hexagon grid covering another spatial table. EPSG must be specified for the hexagons, as well as the size in square KM.

Parameters

- **new_table_name** (*str*) – Name of the new table to create
- **table_to_cover** (*str*) – Name of the existing table you want to cover
- **desired_epsg** (*int*) – integer for EPSG you want the hexagons to be in
- **hexagon_size** (*float*) – Size of the hexagons, 1 = 1 square KM

pgsql2shp(*table_name: str*, *output_folder: pathlib.Path = None*, *extra_args: list = None*) → pathlib.Path

Use the command-line `pgsql2shp` utility.

TODO: check if pgsql2shp exists and exit early if not TODO: check if schema is supported

`extra_args` is a list of tuples, passed in as `[(flag1, val1), (flag2, val2)]`

For example: `extra_args = [("-g", "custom_geom_column"), ("-b", "")]`

For more info, see http://www.postgis.net/docs/manual-1.3/ch04.html#id436110

Parameters

- **table_name** (*str*) – name of the spatial table to dump
- **output_folder** (*Path, optional*) – output folder, defaults to DATA_OUTBOX
- **extra_args** (*list, optional*) – [description], defaults to None

Returns path to the newly created shapefile

Return type Path

query_as_df(*query: str*, *super_uri: bool = False*) → pandas.core.frame.DataFrame

Query the database and get the result as a `pandas.DataFrame`

Parameters

- **query** (*str*) – any valid SQL query string
- **super_uri** (*bool, optional*) – flag that will execute against the super db/user, defaults to False

Returns dataframe with the query result

Return type pd.DataFrame

query_as_geo_df(*query: str*, *geom_col: str = 'geom'*) → geopandas.geodataframe.GeoDataFrame

Query the database and get the result as a `geopandas.GeoDataFrame`

Parameters

- **query** (*str*) – any valid SQL query string
- **geom_col** (*str*) – name of the column that holds the geometry, defaults to 'geom'

Returns geodataframe with the query result

Return type gpd.GeoDataFrame

query_as_list(*query: str*, *super_uri: bool = False*) → list

Query the database and get the result as a `list`

Parameters

- **query** (*str*) – any valid SQL query string
- **super_uri** (*bool, optional*) – flag that will execute against the super db/user, defaults to False

Returns list with each item being a row from the query result

Return type list

query_as_single_item(*query: str*, *super_uri: bool = False*)

Query the database and get the result as a SINGLETON. For when you want to transform `[(True,)]` into `True`

Parameters

- **query** (*str*) – any valid SQL query string
- **super_uri** (*bool, optional*) – flag that will execute against the super db/user, defaults to False

Returns result from the query

Return type singleton

shp2pgsql(*table_name: str*, *src_shapefile: pathlib.Path*, *new_epsg: int = None*) → str

TODO: Docstring TODO: add schema option

Parameters

- **table_name** (*str*) – [description]
- **src_shapefile** (*Path*) – [description]
- **new_epsg** (*int, optional*) – [description], defaults to None

Returns [description]

Return type str

table_add_or_nullify_column(*table_name: str*, *column_name: str*, *column_type: str*, *schema: str = None*) → None

Add a new column to a table. Overwrite to `NULL` if it already exists.

Parameters

- **table_name** (*str*) – Name of the table
- **column_name** (*str*) – Name of the new column
- **column_type** (*str*) – Data type of the column. Must be valid in PgSQL

table_add_spatial_index(*table_name: str*, *schema: str = None*) → None

Add a spatial index to the 'geom' column in the table.

Parameters **table_name** (*str*) – Name of the table to make the index on

table_add_uid_column(*table_name: str*, *schema: str = None*, *uid_col: str = 'uid'*) → None

Add a serial primary key column named 'uid' to the table.

Parameters **table_name** (*str*) – Name of the table to add a uid column to

table_columns_as_list(*table_name: str*, *schema: str = None*) → list

Get a list of all columns in a table.

Parameters **table_name** (*str*) – Name of the table

Returns List of all columns in a table

Return type list

table_delete(*table_name: str*, *schema: str = None*) → None
Delete the table, cascade.

Parameters **table_name** (*str*) – Name of the table you want to delete.

table_reproject_spatial_data(*table_name: str, old_epsg: Union[int, str], new_epsg: Union[int, str], geom_type: str, schema: str = None*) → None
Transform spatial data from one EPSG into another EPSG.

This can also be used with the same old and new EPSG. This is useful when making a new geotable, as this SQL code will update the table's entry in the `geometry_columns` table.

Parameters

- **table_name** (*str*) – name of the table
- **old_epsg** (*Union[int, str]*) – Current EPSG of the data
- **new_epsg** (*Union[int, str]*) – Desired new EPSG for the data
- **geom_type** (*str*) – PostGIS-valid name of the geometry you're transforming

table_spatialize_points(*src_table: str*, *x_lon_col: str*, *y_lat_col: str*, *epsg: int*, *if_exists: str = 'replace'*, *new_table: str = None*, *schema: str = None*) → geopandas.geodataframe.GeoDataFrame

timer()
Decorator function that will record & report on how long it takes for another function to execute.

Parameters **func** (*function*) – the function to be timed

transfer_data_to_another_db(*table_name: str*, *other_postgresql_db*, *schema: str = None*) → None
Copy data from one SQL database to another.

Parameters

- **table_name** (*str*) – Name of the table to copy
- **other_postgresql_db** (PostgreSQL) – `PostgreSQL()` object for target database

uri(*super_uri: bool = False*) → str
Create a connection string URI for this database.

Parameters **super_uri** (*bool, optional*) – Flag that will provide access to cluster root if True, defaults to False

Returns Connection string URI for PostgreSQL

Return type str

postgis_helpers.PgSQL.**connect_via_uri**(*uri: str*, *verbosity: str = 'full'*, *super_db: str = 'postgres'*, *super_user: str = None*, *super_pw: str = None*)
Create a `PostgreSQL` object from a URI. Note that this process must make assumptions about the super-user of the database. Proceed with caution.

Parameters

- **uri** (*str*) – database connection string
- **verbosity** (*str, optional*) – level of printout desired, defaults to "full"
- **super_db** (*str, optional*) – name of the SQL cluster master DB, defaults to "postgres"

Returns `PostgreSQL()` object

Return type *PostgreSQL*

postgis_helpers.config_helpers module

Summary of `config_helpers.py`

You may have lots of SQL database connections. Or maybe you just want to keep your database connection information out of your analysis code. This module provides a method for storing and retrieving connections to database clusters by named profiles inside `[square_brackets]`.

TODO: update this with new features

`postgis_helpers.config_helpers.`**`configurations`**(*filepath: pathlib.Path = PosixPath('/home/docs/sql_data_io/database_connections.cfg')*) → dict

`postgis_helpers.config_helpers.`**`make_config_file`**(*filepath: Union[pathlib.Path, str] = PosixPath('/home/docs/sql_data_io/database_connections.cfg')*, *overwrite: bool = False*) → bool

`postgis_helpers.config_helpers.`**`read_config_file`**(*filepath: pathlib.Path = PosixPath('/home/docs/sql_data_io/database_connections.cfg')*) → dict

postgis_helpers.sql_helpers module

This function was found and copied from the web link below: https://github.com/minus34/postgis-scripts/blob/master/hex-grid/create-hex-grid-function.sql

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

R

S

T

U