cartoframes Documentation

Release 0.5.3

CARTO

Mar 06, 2018

Contents

A Python package for integrating CARTO maps, analysis, and data services into data science workflows.

Python data analysis workflows often rely on the de facto standards pandas and Jupyter notebooks. Integrating CARTO into this workflow saves data scientists time and energy by not having to export datasets as files or retain multiple copies of the data. Instead, CARTOframes give the ability to communicate reproducible analysis while providing the ability to gain from CARTO's services like hosted, dynamic or static maps and Data Observatory augmentation.

Features

- Write pandas DataFrames to CARTO tables
- Read CARTO tables and queries into pandas DataFrames
- Create customizable, interactive CARTO maps in a Jupyter notebook
- Interact with CARTO's Data Observatory
- Use CARTO's spatially-enabled database for analysis

More info

- Complete documentation: http://cartoframes.readthedocs.io/en/latest/
- Source code: https://github.com/CartoDB/cartoframes
- bug tracker / feature requests: https://github.com/CartoDB/cartoframes/issues

---

Note: *cartoframes* users must have a CARTO API key for most *cartoframes* functionality. For example, writing DataFrames to an account, reading from private tables, and visualizing data on maps all require an API key. CARTO provides API keys for education and nonprofit uses, among others. Request access at support@carto.com. API key access is also given through GitHub's Student Developer Pack.

CHAPTER 1

Install Instructions

To install *cartoframes* (currently in beta) on your machine, do the following to install the latest pre-release version:

```
$ pip install cartoframes
```

It is recommended to use *cartoframes* in Jupyter Notebooks (*pip install jupyter*). See the example usage section below or notebooks in the examples directory for using *cartoframes* in that environment.

1.1 Virtual Environment

To setup *cartoframes* and *Jupyter* in a virtual environment:

```
$ virtualenv venv
$ source venv/bin/activate
(venv) $ pip install cartoframes
(venv) $ pip install jupyter
(venv) $ jupyter notebook
```

Then create a new notebook and try the example code snippets below with tables that are in your CARTO account.

CHAPTER 2

Example usage

2.1 Data workflow

Get table from CARTO, make changes in pandas, sync updates with CARTO:

```
import cartoframes
# `base_url`s are of the form `http://{username}.carto.com/` for most users
cc = cartoframes.CartoContext(base_url='https://eschbacher.carto.com/',
                              api_key=APIKEY)

# read a table from your CARTO account to a DataFrame
df = cc.read('brooklyn_poverty_census_tracts')

# do fancy pandas operations (add/drop columns, change values, etc.)
df['poverty_per_pop'] = df['poverty_count'] / df['total_population']

# updates CARTO table with all changes from this session
cc.write(df, 'brooklyn_poverty_census_tracts', overwrite=True)
```

Write an existing pandas DataFrame to CARTO.

```
import pandas as pd
import cartoframes
df = pd.read_csv('acadia_biodiversity.csv')
cc = cartoframes.CartoContext(base_url=BASEURL,
                              api_key=APIKEY)
cc.write(df, 'acadia_biodiversity')
```

2.2 Map workflow

The following will embed a CARTO map in a Jupyter notebook, allowing for custom styling of the maps driven by TurboCARTO and CARTOColors. See the CARTOColors wiki for a full list of available color schemes.

```
from cartoframes import Layer, BaseMap, styling
cc = cartoframes.CartoContext(base_url=BASEURL,
                              api_key=APIKEY)
cc.map(layers=[BaseMap('light'),
               Layer('acadia_biodiversity',
                     color={'column': 'simpson_index',
                            'scheme': styling.tealRose(5)}),
               Layer('peregrine_falcon_nest_sites',
                     size='num_eggs',
                     color={'column': 'bird_id',
                            'scheme': styling.vivid(10)})],
       interactive=True)
```

Note: Legends are under active development. See https://github.com/CartoDB/cartoframes/pull/184 for more information. To try out that code, install *cartoframes* as:

pip install git+https://github.com/cartodb/cartoframes.git@add-legends-v1#egg=cartoframes

2.3 Augment from Data Observatory

Interact with CARTO's Data Observatory:

```
import cartoframes
cc = cartoframes.CartoContext(BASEURL, APIKEY)

# total pop, high school diploma (normalized), median income, poverty status␣
↪(normalized)
# See Data Observatory catalog for codes: https://cartodb.github.io/bigmetadata/index.
↪html
data_obs_measures = [{'numer_id': 'us.census.acs.B01003001'},
                     {'numer_id': 'us.census.acs.B15003017',
                      'normalization': 'predenominated'},
                     {'numer_id': 'us.census.acs.B19013001'},
                     {'numer_id': 'us.census.acs.B17001002',
                      'normalization': 'predenominated'},]
df = cc.data('transactions', data_obs_measures)
```

2.4 CARTO Credential Management

Save and update your CARTO credentials for later use.

```
from cartoframes import Credentials, CartoContext
creds = Credentials(username='eschbacher', key='abcdefg')
creds.save()  # save credentials for later use (not dependent on Python session)
```

Once you save your credentials, you can get started in future sessions more quickly:

```
from cartoframes import CartoContext
cc = CartoContext()  # automatically loads credentials if previously saved
```

CHAPTER 3

CARTOframes Functionality

3.1 CartoContext

class context.**CartoContext**(*base_url=None*, *api_key=None*, *creds=None*, *session=None*, *verbose=0*)

CartoContext class for authentication with CARTO and high-level operations such as reading tables from CARTO into dataframes, writing dataframes to CARTO tables, creating custom maps from dataframes and CARTO tables, and augmenting data using CARTO's Data Observatory. Future methods will interact with CARTO's services like routing, geocoding, and isolines, PostGIS backend for spatial processing, and much more.

Manages connections with CARTO for data and map operations. Modeled after SparkContext.

creds

cartoframes.Credentials – `Credentials` instance

Parameters

- **base_url** (*str*) – Base URL of CARTO user account. Cloud-based accounts should use the form `https://{username}.carto.com` (e.g., https://eschbacher.carto.com for user `eschbacher`) whether on a personal or multi-user account. On-premises installation users should ask their admin.
- **api_key** (*str*) – CARTO API key.
- **session** (*requests.Session, optional*) – requests session. See requests documentation for more information.
- **verbose** (*bool, optional*) – Output underlying process states (True), or suppress (False, default)

Returns A CartoContext object that is authenticated against the user's CARTO account.

Return type *CartoContext*

Example

Create a CartoContext object:

```
import cartoframes
cc = cartoframes.CartoContext(BASEURL, APIKEY)
```

write(*df*, *table_name*, *temp_dir=SYSTEM_TMP_PATH*, *overwrite=False*, *lnglat=None*, *encode_geom=False*, *geom_col=None*, ***kwargs*)

Write a DataFrame to a CARTO table.

Example

Write a pandas DataFrame to CARTO.

```
cc.write(df, 'brooklyn_poverty', overwrite=True)
```

Scrape an HTML table from Wikipedia and send to CARTO with content guessing to create a geometry from the country column. This uses a CARTO Import API param *content_guessing* parameter.

```
url = 'https://en.wikipedia.org/wiki/List_of_countries_by_life_expectancy'
# retrieve first HTML table from that page
df = pd.read_html(url, header=0)[0]
# send to carto, let it guess polygons based on the 'country'
#   column. Also set privacy to 'public'
cc.write(df, 'life_expectancy',
         content_guessing=True,
         privacy='public')
cc.map(layers=Layer('life_expectancy',
                    color='both_sexes_life_expectancy'))
```

Parameters

- **df** (*pandas.DataFrame*) – DataFrame to write to `table_name` in user CARTO account
- **table_name** (*str*) – Table to write `df` to in CARTO.
- **temp_dir** (*str, optional*) – Directory for temporary storage of data that is sent to CARTO. Defaults are defined by appdirs.
- **overwrite** (*bool, optional*) – Behavior for overwriting `table_name` if it exits on CARTO. Defaults to `False`.
- **lnglat** (*tuple, optional*) – lng/lat pair that can be used for creating a geometry on CARTO. Defaults to `None`. In some cases, geometry will be created without specifying this. See CARTO's Import API for more information.
- **encode_geom** (*bool, optional*) – Whether to write *geom_col* to CARTO as *the_geom*.
- **geom_col** (*str, optional*) – The name of the column where geometry information is stored. Used in conjunction with *encode_geom*.
- **kwargs** – Keyword arguments to control write operations. Options are:
  - *compression* to set compression for files sent to CARTO. This will cause write speedups depending on the dataset. Options are `None` (no compression, default) or `gzip`.

– Some arguments from CARTO's Import API. See the params listed in the documentation for more information. For example, when using *content_guessing='true'*, a column named 'countries' with country names will be used to generate polygons for each country. Another use is setting the privacy of a dataset. To avoid unintended consequences, avoid *file*, *url*, and other similar arguments.

Returns If *lnglat* flag is set and the DataFrame has more than 100,000 rows, a `BatchJobStatus` instance is returned. Otherwise, None.

Return type `BatchJobStatus` or None

Note: DataFrame indexes are changed to ordinary columns. CARTO creates an index called *cartodb_id* for every table that runs from 1 to the length of the DataFrame.

---

read(*table_name*, *limit=None*, *index='cartodb_id'*, *decode_geom=False*)

Read a table from CARTO into a pandas DataFrames.

Parameters

- **table_name** (*str*) – Name of table in user's CARTO account.
- **limit** (*int, optional*) – Read only *limit* lines from *table_name*. Defaults to `None`, which reads the full table.
- **index** (*str, optional*) – Not currently in use.
- **decode_geom** (*bool, optional*) – Decodes CARTO's geometries into a Shapely object that can be used, for example, in GeoPandas.

Returns DataFrame representation of *table_name* from CARTO.

Return type pandas.DataFrame

Example

```
import cartoframes
cc = cartoframes.CartoContext(BASEURL, APIKEY)
df = cc.read('acadia_biodiversity')
```

delete(*table_name*)

Delete a table in user's CARTO account.

Parameters **table_name** (*str*) – Name of table to delete

Returns None

query(*query*, *table_name=None*, *decode_geom=False*)

Pull the result from an arbitrary SQL query from a CARTO account into a pandas DataFrame. Can also be used to perform database operations (creating/dropping tables, adding columns, updates, etc.).

Parameters

- **query** (*str*) – Query to run against CARTO user database. This data will then be converted into a pandas DataFrame.
- **table_name** (*str, optional*) – If set, this will create a new table in the user's CARTO account that is the result of the query. Defaults to None (no table created).

- **decode_geom** (*bool, optional*) – Decodes CARTO's geometries into a Shapely object that can be used, for example, in GeoPandas.

Returns DataFrame representation of query supplied. Pandas data types are inferred from PostgreSQL data types. In the case of PostgreSQL date types, dates are attempted to be converted, but on failure a data type 'object' is used.

Return type pandas.DataFrame

map (*layers=None*, *interactive=True*, *zoom=None*, *lat=None*, *lng=None*, *size=(800*, *400)*, *ax=None*)
Produce a CARTO map visualizing data layers.

Examples

Create a map with two data layers, and one BaseMap layer:

```
import cartoframes
from cartoframes import Layer, BaseMap, styling
cc = cartoframes.CartoContext(BASEURL, APIKEY)
cc.map(layers=[BaseMap(),
               Layer('acadia_biodiversity',
                     color={'column': 'simpson_index',
                            'scheme': styling.tealRose(7)}),
               Layer('peregrine_falcon_nest_sites',
                     size='num_eggs',
                     color={'column': 'bird_id',
                            'scheme': styling.vivid(10))],
       interactive=True)
```

Create a snapshot of a map at a specific zoom and center:

```
cc.map(layers=Layer('acadia_biodiversity',
                    color='simpson_index'),
       interactive=False,
       zoom=14,
       lng=-68.3823549,
       lat=44.3036906)
```

Parameters

- **layers** (*list, optional*) – List of one or more of the following:
  - Layer: cartoframes Layer object for visualizing data from a CARTO table. See *layer.Layer* for all styling options.
  - BaseMap: Basemap for contextualizng data layers. See *layer.BaseMap* for all styling options.
  - QueryLayer: Layer from an arbitrary query. See *layer.QueryLayer* for all styling options.
- **interactive** (*bool, optional*) – Defaults to `True` to show an interactive slippy map. Setting to `False` creates a static map.
- **zoom** (*int, optional*) – Zoom level of map. Acceptable values are usually in the range 0 to 19. 0 has the entire earth on a single tile (256px square). Zoom 19 is the size of a city block. Must be used in conjunction with `lng` and `lat`. Defaults to a view to have all data layers in view.

- **lat** (*float, optional*) – Latitude value for the center of the map. Must be used in conjunction with `zoom` and `lng`. Defaults to a view to have all data layers in view.
- **lng** (*float, optional*) – Longitude value for the center of the map. Must be used in conjunction with `zoom` and `lat`. Defaults to a view to have all data layers in view.
- **size** (*tuple, optional*) – List of pixel dimensions for the map. Format is `(width, height)`. Defaults to `(800, 400)`.
- **ax** – matplotlib axis on which to draw the image. Only used when `interactive` is `False`.

Returns Interactive maps are rendered as HTML in an *iframe*, while static maps are returned as matplotlib Axes objects or IPython Image.

Return type IPython.display.HTML or matplotlib Axes

data_boundaries(*boundary=None*, *region=None*, *decode_geom=False*, *timespan=None*, *include_nonclipped=False*)

Find all boundaries available for the world or a *region*. If *boundary* is specified, get all available boundary polygons for the region specified (if any). This method is espeically useful for getting boundaries for a region and, with *CartoContext.data* and *CartoContext.data_discovery*, getting tables of geometries and the corresponding raw measures. For example, if you want to analyze how median income has changed in a region (see examples section for more).

Examples

Find all boundaries available for Australia. The columns *geom_name* gives us the name of the boundary and *geom_id* is what we need for the *boundary* argument.

```
import cartoframes
cc = cartoframes.CartoContext('base url', 'api key')
au_boundaries = cc.data_boundaries(region='Australia')
au_boundaries[['geom_name', 'geom_id']]
```

Get the boundaries for Australian Postal Areas and map them.

```
from cartoframes import Layer
au_postal_areas = cc.data_boundaries(boundary='au.geo.POA')
cc.write(au_postal_areas, 'au_postal_areas')
cc.map(Layer('au_postal_areas'))
```

Get census tracts around Idaho Falls, Idaho, USA, and add median income from the US census. Without limiting the metadata, we get median income measures for each census in the Data Observatory.

```
cc = cartoframes.CartoContext('base url', 'api key')
# will return DataFrame with columns `the_geom` and `geom_ref`
tracts = cc.data_boundaries(
    boundary='us.census.tiger.census_tract',
    region=[-112.096642,43.429932,-111.974213,43.553539])
# write geometries to a CARTO table
cc.write(tracts, 'idaho_falls_tracts')
# gather metadata needed to look up median income
median_income_meta = cc.data_discovery(
    'idaho_falls_tracts',
    keywords='median income',
    boundaries='us.census.tiger.census_tract')
# get median income data and original table as new dataframe
```

```
idaho_falls_income = cc.data(
    'idaho_falls_tracts',
    median_income_meta,
    how='geom_refs')
# overwrite existing table with newly-enriched dataframe
cc.write(idaho_falls_income,
         'idaho_falls_tracts',
         overwrite=True)
```

Parameters

- **boundary** (*str, optional*) – Boundary identifier for the boundaries that are of interest. For example, US census tracts have a boundary ID of `us.census.tiger.census_tract`, and Brazilian Municipios have an ID of `br.geo.municipios`. Find IDs by running *CartoContext.data_boundaries* without any arguments, or by looking in the Data Observatory catalog.
- **region** (*str, optional*) – Region where boundary information or, if *boundary* is specified, boundary polygons are of interest. *region* can be one of the following:
  - table name (str): Name of a table in user's CARTO account
  - bounding box (list of float): List of four values (two lng/lat pairs) in the following order: western longitude, southern latitude, eastern longitude, and northern latitude. For example, Switzerland fits in `[5.9559111595,45.8179931641,10.4920501709,47.808380127]`
- **timespan** (*str, optional*) – Specific timespan to get geometries from. Defaults to use the most recent. See the Data Observatory catalog for more information.
- **decode_geom** (*bool, optional*) – Whether to return the geometries as Shapely objects or keep them encoded as EWKB strings. Defaults to False.
- **include_nonclipped** (*bool, optional*) – Optionally include non-shoreline-clipped boundaries. These boundaries are the raw boundaries provided by, for example, US Census Tiger.

Returns If *boundary* is specified, then all available boundaries and accompanying *geom_refs* in *region* (or the world if *region* is `None` or not specified) are returned. If *boundary* is not specified, then a DataFrame of all available boundaries in *region* (or the world if *region* is `None`)

Return type pandas.DataFrame

data_discovery(*region*, *keywords=None*, *regex=None*, *time=None*, *boundaries=None*, *include_quantiles=False*)

Discover Data Observatory measures. This method returns the full Data Observatory metadata model for each measure or measures that match the conditions from the inputs. The full metadata in each row uniquely defines a measure based on the timespan, geographic resolution, and normalization (if any). Read more about the metadata response in Data Observatory documentation.

Internally, this method finds all measures in *region* that match the conditions set in *keywords*, *regex*, *time*, and *boundaries* (if any of them are specified). Then, if *boundaries* is not specified, a geographical resolution for that measure will be chosen subject to the type of region specified:

1. If *region* is a table name, then a geographical resolution that is roughly equal to *region size / number of subunits*.
2. If *region* is a country name or bounding box, then a geographical resolution will be chosen roughly equal to *region size / 500*.

Since different measures are in some geographic resolutions and not others, different geographical resolutions for different measures are oftentimes returned.

Tip: To remove the guesswork in how geographical resolutions are selected, specify one or more boundaries in *boundaries*. See the boundaries section for each region in the Data Observatory catalog.

The metadata returned from this method can then be used to create raw tables or for augmenting an existing table from these measures using *CartoContext.data*. For the full Data Observatory catalog, visit https://cartodb.github.io/bigmetadata/. When working with the metadata DataFrame returned from this method, be careful to only remove rows not columns as *CartoContext.data* generally needs the full metadata.

Note: Narrowing down a discovery query using the *keywords*, *regex*, and *time* filters is important for getting a manageable metadata set. Besides there being a large number of measures in the DO, a metadata response has acceptable combinations of measures with demonimators (normalization and density), and the same measure from other years.

For example, setting the region to be United States counties with no filter values set will result in many thousands of measures.

Examples

Get all European Union measures that mention `freight`.

```
meta = cc.data_discovery('European Union',
                         keywords='freight',
                         time='2010')
print(meta['numer_name'].values)
```

Parameters

- **region** (*str or list of float*) – Information about the region of interest. *region* can be one of three types:
  - region name (str): Name of region of interest. Acceptable values are limited to: 'Australia', 'Brazil', 'Canada', 'European Union', 'France', 'Mexico', 'Spain', 'United Kingdom', 'United States'.
  - table name (str): Name of a table in user's CARTO account with geometries. The region will be the bounding box of the table.

    Note: If a table name is also a valid Data Observatory region name, the Data Observatory name will be chosen over the table.

  - bounding box (list of float): List of four values (two lng/lat pairs) in the following order: western longitude, southern latitude, eastern longitude, and northern latitude. For example, Switzerland fits in `[5.9559111595,45.8179931641,10.4920501709,47.808380127]`

    Note: Geometry levels are generally chosen by subdividing the region into the next smallest administrative unit. To override this behavior, specify the *boundaries*

flag. For example, set *boundaries* to `'us.census.tiger.census_tract'` to choose US census tracts.

- **keywords** (*str or list of str, optional*) – Keyword or list of keywords in measure description or name. Response will be matched on all keywords listed (boolean *or*).
- **regex** (*str, optional*) – A regular expression to search the measure descriptions and names. Note that this relies on PostgreSQL's case insensitive operator `~*`. See PostgreSQL docs for more information.
- **boundaries** (*str or list of str, optional*) – Boundary or list of boundaries that specify the measure resolution. See the boundaries section for each region in the Data Observatory catalog.
- **include_quantiles** (*bool, optional*) – Include quantiles calculations which are a calculation of how a measure compares to all measures in the full dataset. Defaults to `False`. If `True`, quantiles columns will be returned for each column which has it pre-calculated.

Returns A dataframe of the complete metadata model for specific measures based on the search parameters.

Return type pandas.DataFrame

Raises

- `ValueError` – If *region* is a `list` and does not consist of four elements, or if *region* is not an acceptable region
- `CartoException` – If *region* is not a table in user account

data (*table_name*, *metadata*, *persist_as=None*, *how='the_geom'*)

Get an augmented CARTO dataset with Data Observatory measures. Use *CartoContext.data_discovery* to search for available measures, or see the full Data Observatory catalog. Optionally persist the data as a new table.

Example

Get a DataFrame with Data Observatory measures based on the geometries in a CARTO table.

```
cc = cartoframes.CartoContext(BASEURL, APIKEY)
median_income = cc.data_discovery('transaction_events',
                                  regex='.*median income.*',
                                  time='2011 - 2015')
df = cc.data('transaction_events',
             median_income)
```

Pass in cherry-picked measures from the Data Observatory catalog. The rest of the metadata will be filled in, but it's important to specify the geographic level as this will not show up in the column name.

```
median_income = [{'numer_id': 'us.census.acs.B19013001',
                  'geom_id': 'us.census.tiger.block_group',
                  'numer_timespan': '2011 - 2015'}]
df = cc.data('transaction_events', median_income)
```

Parameters

- **table_name** (*str*) – Name of table on CARTO account that Data Observatory measures are to be added to.
- **metadata** (*pandas.DataFrame*) – List of all measures to add to *table_name*. See *CartoContext.data_discovery* outputs for a full list of metadata columns.
- **persist_as** (*str, optional*) – Output the results of augmenting *table_name* to *persist_as* as a persistent table on CARTO. Defaults to `None`, which will not create a table.
- **how** (*str, optional*) – **Not fully implemented**. Column name for identifying the geometry from which to fetch the data. Defaults to *the_geom*, which results in measures that are spatially interpolated (e.g., a neighborhood boundary's population will be calculated from underlying census tracts). Specifying a column that has the geometry identifier (for example, GEOID for US Census boundaries), results in measures directly from the Census for that GEOID but normalized how it is specified in the metadata.

Returns A DataFrame representation of *table_name* which has new columns for each measure in *metadata*.

Return type pandas.DataFrame

Raises

- `NameError` – If the columns in *table_name* are in the `suggested_name` column of *metadata*.
- `ValueError` – If metadata object is invalid or empty, or if the number of requested measures exceeds 50.
- `CartoException` – If user account consumes all of Data Observatory quota

3.2 Analysis

3.2.1 Overview

Analysis in cartoframes takes two forms:

- **Pipelines**: *AnalysisTree* pipelines where multiple analyses can be listed sequentially off a base data source node (e.g., a *Table* object). This tree is lazily evaluated by applying a `.compute()` method after it is created. Besides the class constructor, analyses can be appended to the tree after it has been instantiated. See *AnalysisTree* for more information. This is modeled after Builder analysis workflows, scikit-learn's PipeLine class, PySpark's SQL syntax, and directed acyclic graphs.

  A key feature of this data structure is that most analyses, besides accepting parameters, can also accept other data sources (*Table*, *Query*, etc.) as well as other `AnalysisTrees`.

  Example:

```
from cartoframes import AnalysisTree, analyses as ca
tree = AnalysisTree(
    Table(cc, 'brooklyn_demographics'),
    [
        ('buffer', ca.Buffer(100.0)),
        ('join', ca.Join(
            Table('gps_pings').filter('type=cell')
            on='the_geom',
            type='left',
```

```
                null_replace=0)
            ),
            ('distinct', ca.Distinct(on='user_id')),
            ('agg', ca.Agg(
                by=['geoid', 'the_geom', ],
                [('num_gps_pings', 'count'),
                 ('num_gps_pings', 'avg'),
                 ('median_income', 'min')]),
            ('div', ca.Division([
                ('num_gps_pings', 'total_pop'),
                ('num_gps_pings', 'the_geom')
            ]))
        ]
)
```

- **Method Chaining**: By chaining analysis methods off of a base data source node. A base data source can be a supported *Data Source*. For a full list of analyses, see the methods of *Query* and *Table*.

  Example:

```
from cartoframes import Table
pt_count = Table('brooklyn_demographics')\
    .join(Table('gps_pings'), on='the_geom', type='left')\
    .agg([('num_gps_pings', 'count'), ], by='the_geom')

# calculate results on a subset of the data
pt_count.compute(subset=0.1)
pt_count.map(color='num_gps_pings')
```

Data Sources

The following data sources can be used as nodes:

- *Table* - a table present in user account
- *Query* - a query against user account
- *BuilderAnalysis* - an analysis node already performed
- *LocalData* - use local data (file, dataframe, etc.)
- *AnalysisTree* - previously constructed analysis tree

Analysis Library

These include traditional GIS operations, common database operations like JOINs, and more advanced spatial statistics. Geopandas has some nice functionality for operations like this, as does pandas and pyspark. Here we want to take advantage of CARTO's cloud-based database to perform these methods, while being careful to stay in a Pythonic syntax like you see in pandas. Other reference: OGC standards: http://www.opengeospatial.org/standards/sfs

Functions

- `AddressNormalization` (#377)
- `Agg`
  - **Use Case:** Summary info of any category/group

  - Params
    - agg_values (list of agg/column tuples): If *category* is specified, use this option for carrying over aggregations within categories. Options available are: *min*, *max*, *count*, *avg*, *sum*, *stddev* and other PostgreSQL aggregation operations.

- `Area`
  - Use Case: Add an area column calculated from geometry in square kilometers.
  - Params
    - units (str): One of `sqkm` (default), `sqmeters`, or `sqmiles`.

- `Buffer`
  - **Use case:** Buffer points, lines, polygons by a given distance. Used widely in spatial analysis for finding points within a distance among other uses
  - References
    - camshaft node
  - Params
    - radius (float, required): radius of buffer in meters

- `Centroid`
  - References
    - camshaft node but need the ability to carry over summary information: e.g., avg value of that centroid group, num of items present
    - Weighted camshaft node
  - Params
    - category (str or list of str, optional): Column name(s) to group by.
    - weight (str, optional): weight the centroid based on the weight of a column value
    - agg_values (list of agg/column tuples): If *category* is specified, use this option for carrying over aggregations within categories. Options available are: *min*, *max*, *count*, *avg*, *sum*, *stddev* and other PostgreSQL aggregation operations.

- `Custom`
  - **Use case:** Define a custom analysis using SQL.
  - References
    - Deprecated SQL node in camshaft
  - Params
    - sql (str): Custom analysis defined as a SQL query

- `DataObs`
  - **Use case:** Augment a dataset with data observatory measures
  - References
    - Existing implementation: cartoframes docs
  - Params
    - same as existing implementation

- `Difference`
- `Distinct`
  - **Use case:** De-dupe a dataset on the columns passed
  - Params
    - cols (str or list of str, optional): Column(s) to de-duplicate the records on. Default is de-duplicate across all columns. Read more in PostgreSQL documentations.
  - Example

```
Distinct(cols=('the_geom', 'id_num', 'reading'))
```

- `Div`
  - **Use case:** Divide one column by another
  - Note: Or should we have an inline-custom function that allows you to write select-level math like this. E.g., user can pass `col1 + 2 * col2 / col3`
  - Params
    - pairs (tuple of str): Tuple of numerator, denominator, and optional name
  - Example

```
Div([('num_pings', 'total_pop', 'pings_per_pop'),
     ('num_pings', 'num_unique_foot_traffic'),
     ('num_pings', 1000.0)]
```

- `Envelope`
  - **Use Case:** Group geometries into a convex hull, bounding box, bounding circle, concave hull, or union
  - References
    - PostGIS docs on these: convex hull, concave hull, bounding box, bounding circle, union.
  - Params
    - type (str, optional): One of *bounding_box* (default), *convex_hull*, *concave_hull*, *bounding_circle*, or *union*
    - category (str or list of str, optional): Column name(s) to group by.
    - agg_values (list of agg/column tuples): If *category* is specified, use this option for carrying over aggregations within categories. Options available are: *min*, *max*, *count*, *avg*, *sum*, *stddev* and other PostgreSQL aggregation operations.
  - Example

```
Envelope(type='convex_hull')
```

- `FillNull`
  - **Use case:** Fill in null values with a specific value
  - References
    - This uses PostgreSQL's coalesce
  - Params
    - fill_vals (dict or list of dicts): Entry in the form:

  - Example

```
# option 1
FillNull({'colname': 1})
# option 2
FillNull({'colname': ['other_column', 0]})
# option 3
FillNull(
    [{'colname': 1},
     {'colname2': 10},
     {'colname3': ['colname', 'colname2', 0]}])
```

- `Filter`
  - **Use case:** Filter out records
  - Params
    - filters (str or list of str): filter (e.g., `col1 <= 10`) or a list of filter conditions. PostgreSQL conditions are valid: <https://www.postgresql.org/docs/9.6/static/functions-comparison.html>.
  - Example

```
Filter(['col1 <= 10', 'ST_Area(the_geom::geography) > 10'])
Filter('col1 == col2')
```

- `Geocoding`
  - **Use case:** Turn numerical or text data into geometries. E.g., street addresses to points, country names to boundaries, IP addresses to points, etc.
  - Reference
    - See the files beginning with *georeference* in camshaft node library
- `Imputer`
  - **Use case:** Help clean up messy data by filling in null values with the mean, median, or mode of the column of interest.
  - Reference
    - Idea comes from scikit-learn
    - See also `StandardScaler`
- `Isochrone`
  - **Use case:** Drive time polygons, etc.
- `Join` (spatial or attribute)
  - **Use case:** Combine data from different data sources which have some data in common (e.g., points intersecting polygons, JOIN by common column values)
  - References
    - GeoPandas
    - PostgreSQL full list of JOIN types
    - PySpark
  - Params
    - *on* (str or list of str): column name or list of column names, 'the_geom' for spatial joins

        - *op* (str): one of *intersects*, *within*, *contains*
        - *how* (str, optional, only use with *on='the_geom'*): *inner* (default), *left*, *right*, *outer*, *cross*
        - *expr* (str, optional): Custom expression (e.g., *col1 == col2* a la PySpark). TODO: should we use Python's == syntax or SQL's = and then convert internally
        - *agg* (TBD, optional): implicit group by / aggregation
- `Kmeans`
    - **Use case:** Find clusters in data spatially or in parameter space
    - Reference
        - camshaft spatial
        - crankshaft non-spatial
- `Length`
    - **Use case:** Get the length of a linestring in meters, kilometers, or miles
- `Line`
    - **Use case:** Create lines out of a series of points
    - Reference
        - See all camshaft nodes like line-*
- `Limit`
    - **Use case:** limit to *n_rows* entries
    - Params:
        - *n_rows* (int): Number of rows to return
- `MoranLocal`
    - **Use case:** Classify geometries by whether they are in a cluster of similarly high or low values, or are an outlier compared to their neighbors
    - References
        - camshaft node
        - crankshaft functions
    - Params
        - val (str): Column name
        - denom (str, optional): Column name for denominator
        - weight_type (str, optional): one of `knn` (default) or `queen`, which requires adjacent geometries
        - n_neighbors (int, optional): Choose the number of neighbors, only valid for k-nearest neighbors
- `Nearest` (give back the n-nearest geometries to another geometry)
- `NullIf`
    - **Use case:** Replace values with null values if a condition is met. Useful for replacing quirky null values like empty strings, values like `'(none)'`, and so on.
    - Params

        * vals (tuple or list of tuples): Column name / value pairs. For example, `('colname', '')` to replace empty strings in the column *colname* with null values.

- `Routing`
  - **Use case:** Routing
  - References
    * See camshaft nodes beginning in routing
  - Params
    * destination (tuple of *`Table`* or *`Query`* and colname)
    * geom_type (str): Whether to return a `road` (default) or `straight`
  - Returns
    * new linestring of route replaces previous *the_geom*
    * length_km (float): length of route
    * duration_sec (float): if possible, return the travel time estimate. Will be `null` if not possible.
- `Sample`
  - **Use case:** Sample from a data source
  - References:
    * camshaft node but *TABLESAMPLE* is a better solution
    * TABLESAMPLE in PostgreSQL.
    * PySpark sample
  - Params
    * fraction (float): fraction (0 <= x <= 1) of dataset to return
- `StandardScaler`
  - **Use case:** Transforms data to be centered on its mean and scaled to have unit variance
  - References:
    * Modeled after scikit-learn's StandardScaler
- `StrJoin` - join a list of column names or literals into a new column
  - **Use case:** Useful for constructing text from a combination of other columns and custom values (e.g., a full address from multiple columns).
  - Params
    * vals (list of str): List of columns or literal values to concatenate into a new column
    * new_colname (str): New column name

Uncertain about adding

- `Closest`
  - This is a camshaft node
- `Contour`

  - Not sure if this is ready for primetime? camshaft node
- `Existence`
  - This is a camshaft node related to widgets and connected analysis tables
- `Gravity`
  - We have a better implementation that's an open PR in crankshaft.
    - current camshaft node
    - crankshaft PR

Note:

- Add status updates (e.g., `node 5 / 7 complete`)
- Have a better representation of an analysis? E.g., scikit-learn passes class instances to the Pipeline

```
from sklearn import svm
from sklearn.datasets import samples_generator
from sklearn.feature_selection import SelectKBest, f_regression
from sklearn.pipeline import Pipeline
# build pipeline
anova_filter = SelectKBest(f_regression, k=5)
clf = svm.SVC(kernel='linear')
anova_svm = Pipeline([('anova', anova_filter), ('svc', clf)])
```

  Each 'analysis' in cartoframes could exist as a class and be constructed similarly. Most would be a clone of the camshaft node, and others would be more data-science-specific.
- Method chaining builds up an AnalysisTree by reapeatedly applying the `.append(...)` to `self`
- Chained methods are lazily evaluated as well
- Add AnalysisTree validation steps for column names / existence of data, etc. for each step of the tree
- Instantiating the *`Table`* or *`Query`* classes is clumsy if the `cc` needs to be passed to it everytime – should it be instantiated differently? Maybe like `cc.table('foo')`, which is equivalent to `Table(cc, 'foo')`? One conflict here is that `cc.query` already exists and means something different.
- `Layer` should have a `Query` attribute instead of storing the query as a string?
- Idea: Partial evaluation to get states of the data along the tree? User could create a shorter tree to do this instead.
- Use Python's operator overloading for operations like *Analyses* + *Analysis* does an `AnalysisTree.append` under the hood
- Add method for trashing / invalidating analysis table and starting anew
- What's the standard on column name inheritance from analysis n to n+1? Which columns come over, which don't, and which are added?
- What can be gleaned from http://www.opengeospatial.org/standards/wps ?
- Draw inspiration from Spark: http://spark.apache.org/docs/2.2.0/api/python/pyspark.sql.html And place functions/classes into a *functions* module http://spark.apache.org/docs/2.2.0/api/python/pyspark.sql.html#module-pyspark.sql.functions
- Keep in mind that the chain is actually a tree since data can come in at different nodes. AnalysisTree may be a better name.
- How should the analyses be structured? Similar to scikit-learn's PipeLine? *[A(param), B(param)]* and *A(param).fit(data)* happens once the tree is evaluated?

- Ability to mix in analyses that are run locally
- Standardize aggregation tuples as they appear in many places. Besides required values, we could have *distinct*, nullif, coalesce, etc. handling options

---

class analysis.**AnalysisTree**(*source*, *analyses*)

Build up an analysis tree à la Builder Analysis or scikit learn Pipeline. Once evaluated with `AnalysisTree.compute()`, the results will persist as a table in the user CARTO account and be returned into the `data` attribute.

AnalysisTree allows you to build up a tree of analyses which are applied sequentially to a source (*Query* or *Table*).

The analysis inputs can be passed as a list of analyses, or as a list of tuples, each with a unique identifier *key* so it can be later referenced or removed from the chain programmatically.

AnalysisTree can also be used as a layer in a map. If it has not already been evaluated, then *.compute()* will be applied to it and the map will not render until the result is computed.

Example

Build and evaluate an analysis tree, return the results into a pandas DataFrame, and map the output

```
from cartoframes import AnalysisTree, Table, CartoContext
from cartoframes import analyses as ca
cc = CartoContext()

# base data node
bklyn_demog = Table(cc, 'brooklyn_demographics')

# build analysis tree
tree = AnalysisTree(
    bklyn_demog,
    [
        # buffer by 100 meters
        ca.Buffer(100.0),
        # spatial join
        ca.Join(target=Table(cc, 'gps_pings').filter('type=cell'),
            on='the_geom',
            type='left'),
        ca.Distinct(on='user_id'),
        # aggregate points to polygons
        ca.Agg(by='geoid', ops=[('count', 'num_gps_pings'), ]),
        # add new column to normalize point count
        ca.Div([('num_gps_pings', 'total_pop')])
    ]
)

# evaluate analysis
tree.compute()

# visualize with carto map
tree.map(color='num_gps_pings_per_total_pop')
```

Parameters

- **source** (str, *Table*, or *Query*) – If str, the name of a table in user account. If *Table* or *Query*, the base data for the analysis tree.
- **analyses** (*list*) – A list of analyses to apply to *source*. The following are *available analyses* and their parameters.

- data
pandas.DataFrame – None until the analysis tree is evaluated, and then a dataframe of the results

- state
str – Current state of the *AnalysisTree*:

- 'not evaluated': Chain has not yet been evaluated
- 'running': Analysis is currently running
- 'enqueued': Analysis is queued to be run
- 'complete': Chain successfully run. Results stored in AnalysisTree.data and .results_url.
- 'failed': Failure message if the analysis failed to complete

- results_url
URL where results stored on CARTO. Note: user has to be authenticated to view the table

- Add method for running the analysis off of a subsample of the data.
E.g., .compute(subsample=0.1), .compute_preview(), etc. etc. With the goal that users feel compelled to run the analysis first on a smaller sample to get the results before running the whole enchilada.

append (*analysis*)
Append a new analysis to an existing tree.

Example

```
tree = AnalysisTree(
    Table('transactions'),
    [
        Buffer(100),
        DataObs('median_income')
    ]
)
tree.append(('knn', {'mean': 'median_income'}))
```

Parameters **analysis** (analysis) – An analysis node

compute (*subset=None*)
Trigger the AnalysisTree to run.

Example

```
tree = AnalysisTree(...)
# compute analysis tree
tree.compute()
# show results
df.data.head()
```

Returns promise object, which reports the status of the analysis if not complete. Once the analysis finishes, the results will be stored in the attributes `data`, `results_url`, and `state`. Also returned would be: compute time and other metadata about the job.

results_url
returns the URL where the analysis table exists on CARTO

class analysis.**BuilderAnalysis**(*context*, *node_hash*)
Builder analysis node. Use this option for placing an analysis node from Builder.

buffer(*dist*)
Buffer query

Example

```
q = Query('...')
buffered_q = q.buffer(150).compute()
cc.map(layers=[buffered_q, q])
```

Parameters **dist** (*float*) – Distance in meters to buffer a geometry

columns
return the column names of the table or query

Returns Column names

Return type pandas.Index

custom(*query*)
Define custom query to add to the tree

Can info be gleaned from the spark registerUDF? http://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html#pyspark.sql.SQLContext.registerJavaFunction

describe(*cols=None*)
Gives back basic statistics for a table

Parameters **cols** (*list of str*) – List of column names to get summary statistics

Returns A statistical description of the data

Return type pandas.DataFrame

div(*numerator*, *denominator*)
Divided one column by another column or expression to produce a new column

Example

Divide by a constant to convert from square kilometers to square miles.

```
from cartoframes import CartoContext, Table
cc = CartoContext()
t = Table(cc, 'acadia_biodiversity')
t.div('ospreys_per_sqkm', 1.6**2)
```

Normalize a column by another column.

```
t = Table(cc, 'acadia_biodiversity')
t.div('osprey_count', 'num_observations')
```

Get the density of a value.

```
t = Table(cc, 'acadia_biodiversity')
t.div('osprey_count', 'the_geom')
```

head(*n_rows=5*)
similar to pandas.DataFrame.head

moran_local(*colname*, *denom=None*, *n_neighbors=5*)
Local Moran's I

Parameters

- **colname** (str) – Column name for performing Local Moran's I analysis on
- **denom** (str, optional) – Optional denominator for *colname*.
- **n_neighbors** (int, optional) – Number of neighbors for each geometry. Defaults to 5.

pgtypes
return the dtypes of the columns of the table or query

Returns Data types (in a PostgreSQL database) of columns

Return type pandas.Series

plot()
Plot all the columns in the query.

Example:

```
Query('''
    SELECT simpson_index, species
    FROM acadia_biodiversity
''').plot()
<matplotlib plot>
```

read()
Read the query to a pandas DataFrame

Returns Query represented as a pandas DataFrame

Return type pandas.DataFrame

stop()
stop job from running. Other names: halt, delete, ... This operates on the promise object after running AnalysisTree.compute

tail(*n_rows=5*)
Return last n_rows of self.query

class analysis.**LocalData**(*context*, *data*)
Use a local file, dataframe, etc. as a node in the analysis framework

buffer(*dist*)
Buffer query

Example

```
q = Query('...')
buffered_q = q.buffer(150).compute()
cc.map(layers=[buffered_q, q])
```

Parameters **dist** (*float*) – Distance in meters to buffer a geometry

columns
return the column names of the table or query

Returns Column names

Return type pandas.Index

custom(*query*)
Define custom query to add to the tree

Can info be gleaned from the spark registerUDF? http://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html#pyspark.sql.SQLContext.registerJavaFunction

describe(*cols=None*)
Gives back basic statistics for a table

Parameters **cols** (*list of str*) – List of column names to get summary statistics

Returns A statistical description of the data

Return type pandas.DataFrame

div(*numerator*, *denominator*)
Divided one column by another column or expression to produce a new column

Example

Divide by a constant to convert from square kilometers to square miles.

```
from cartoframes import CartoContext, Table
cc = CartoContext()
t = Table(cc, 'acadia_biodiversity')
t.div('ospreys_per_sqkm', 1.6**2)
```

Normalize a column by another column.

```
t = Table(cc, 'acadia_biodiversity')
t.div('osprey_count', 'num_observations')
```

Get the density of a value.

```
t = Table(cc, 'acadia_biodiversity')
t.div('osprey_count', 'the_geom')
```

head(*n_rows=5*)
similar to pandas.DataFrame.head

moran_local(*colname*, *denom=None*, *n_neighbors=5*)
Local Moran's I

Parameters

- **colname** (str) – Column name for performing Local Moran's I analysis on
- **denom** (str, optional) – Optional denominator for *colname*.
- **n_neighbors** (int, optional) – Number of neighbors for each geometry. Defaults to 5.

pgtypes
return the dtypes of the columns of the table or query

Returns Data types (in a PostgreSQL database) of columns

Return type pandas.Series

plot()
Plot all the columns in the query.

Example:

```
Query('''
    SELECT simpson_index, species
    FROM acadia_biodiversity
''').plot()
<matplotlib plot>
```

read()
Read the query to a pandas DataFrame

Returns Query represented as a pandas DataFrame

Return type pandas.DataFrame

stop()
stop job from running. Other names: halt, delete, ... This operates on the promise object after running AnalysisTree.compute

tail(*n_rows=5*)
Return last n_rows of self.query

class analysis.**Query**(*context*, *query*)
Query gives a representation of a query in a users's CARTO account.

Example

```
from cartoframes import CartoContext, Query
cc = CartoContext()
snapshot = Query('''
    SELECT
        count(*) as num_sightings,
        b.acadia_district_name,
        b.the_geom
    FROM
        bird_sightings as a, acadia_districts as b
    WHERE
        ST_Intersects(b.the_geom, a.the_geom)
    GROUP BY 2, 3
''')
snapshot.local_moran('num_sightings', 5).filter('significance<=0.05')
```

Parameters

- **context** (CartoContext) – CartoContext instance authenticated against the user's CARTO account.
- **query** (*str*) – Valid query against user's CARTO account.

buffer(*dist*)
Buffer query

Example

```
q = Query('...')
buffered_q = q.buffer(150).compute()
cc.map(layers=[buffered_q, q])
```

Parameters **dist** (*float*) – Distance in meters to buffer a geometry

columns
return the column names of the table or query

Returns Column names

Return type pandas.Index

custom(*query*)
Define custom query to add to the tree

Can info be gleaned from the spark registerUDF? http://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html#pyspark.sql.SQLContext.registerJavaFunction

describe(*cols=None*)
Gives back basic statistics for a table

Parameters **cols** (*list of str*) – List of column names to get summary statistics

Returns A statistical description of the data

Return type pandas.DataFrame

div(*numerator*, *denominator*)
Divided one column by another column or expression to produce a new column

Example

Divide by a constant to convert from square kilometers to square miles.

```
from cartoframes import CartoContext, Table
cc = CartoContext()
t = Table(cc, 'acadia_biodiversity')
t.div('ospreys_per_sqkm', 1.6**2)
```

Normalize a column by another column.

```
t = Table(cc, 'acadia_biodiversity')
t.div('osprey_count', 'num_observations')
```

Get the density of a value.

```
t = Table(cc, 'acadia_biodiversity')
t.div('osprey_count', 'the_geom')
```

head(*n_rows=5*)
similar to pandas.DataFrame.head

moran_local(*colname*, *denom=None*, *n_neighbors=5*)
Local Moran's I

Parameters

- **colname** (str) – Column name for performing Local Moran's I analysis on
- **denom** (str, optional) – Optional denominator for *colname*.
- **n_neighbors** (int, optional) – Number of neighbors for each geometry. Defaults to 5.

pgtypes
return the dtypes of the columns of the table or query

Returns Data types (in a PostgreSQL database) of columns

Return type pandas.Series

plot()
Plot all the columns in the query.

Example:

```
Query('''
    SELECT simpson_index, species
    FROM acadia_biodiversity
''').plot()
<matplotlib plot>
```

read()
Read the query to a pandas DataFrame

Returns Query represented as a pandas DataFrame

Return type pandas.DataFrame

stop()
stop job from running. Other names: halt, delete, ... This operates on the promise object after running AnalysisTree.compute

tail(*n_rows=5*)
Return last n_rows of self.query

class analysis.**Table**(*context*, *table_name*)
Table object

buffer(*dist*)
Buffer query

Example

```
q = Query('...')
buffered_q = q.buffer(150).compute()
cc.map(layers=[buffered_q, q])
```

Parameters **dist** (*float*) – Distance in meters to buffer a geometry

columns
return the column names of the table or query

Returns Column names

Return type pandas.Index

custom(*query*)
Define custom query to add to the tree

Can info be gleaned from the spark registerUDF? http://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html#pyspark.sql.SQLContext.registerJavaFunction

describe(*cols=None*)
Gives back basic statistics for a table

Parameters **cols** (*list of str*) – List of column names to get summary statistics

Returns A statistical description of the data

Return type pandas.DataFrame

div(*numerator*, *denominator*)
Divided one column by another column or expression to produce a new column

Example

Divide by a constant to convert from square kilometers to square miles.

```
from cartoframes import CartoContext, Table
cc = CartoContext()
t = Table(cc, 'acadia_biodiversity')
t.div('ospreys_per_sqkm', 1.6**2)
```

Normalize a column by another column.

```
t = Table(cc, 'acadia_biodiversity')
t.div('osprey_count', 'num_observations')
```

Get the density of a value.

```
t = Table(cc, 'acadia_biodiversity')
t.div('osprey_count', 'the_geom')
```

head(*n_rows=5*)
similar to pandas.DataFrame.head

moran_local(*colname*, *denom=None*, *n_neighbors=5*)
Local Moran's I

Parameters

- **colname** (str) – Column name for performing Local Moran's I analysis on
- **denom** (str, optional) – Optional denominator for *colname*.
- **n_neighbors** (int, optional) – Number of neighbors for each geometry. Defaults to 5.

pgtypes
return the dtypes of the columns of the table or query

Returns Data types (in a PostgreSQL database) of columns

Return type pandas.Series

plot()
Plot all the columns in the query.

Example:

```
Query('''
    SELECT simpson_index, species
    FROM acadia_biodiversity
''').plot()
<matplotlib plot>
```

read()
Read the query to a pandas DataFrame

Returns Query represented as a pandas DataFrame

Return type pandas.DataFrame

stop()
stop job from running. Other names: halt, delete, ... This operates on the promise object after running AnalysisTree.compute

tail(*n_rows=5*)
Return last n_rows of self.query

3.3 Map Layer Classes

Layer classes for map creation. See examples in *layer.Layer* and *layer.QueryLayer* for example usage for data layers. See *layer.BaseMap* for basemap layers.

class layer.**BaseMap**(*source='voyager'*, *labels='back'*, *only_labels=False*)
Layer object for adding basemaps to a cartoframes map.

Example

Add a custom basemap to a cartoframes map.

```
import cartoframes
from cartoframes import BaseMap, Layer
cc = cartoframes.CartoContext(BASEURL, APIKEY)
cc.map(layers=[BaseMap(source='light', labels='front'),
               Layer('acadia_biodiversity')])
```

Parameters

- **source** (*str, optional*) – One of `light` or `dark`. Defaults to `voyager`. Basemaps come from https://carto.com/location-data-services/basemaps/
- **labels** (*str, optional*) – One of `back`, `front`, or None. Labels on the front will be above the data layers. Labels on back will be underneath the data layers but on top of the basemap. Setting labels to `None` will only show the basemap.

- **only_labels** (*bool, optional*) – Whether to show labels or not.

is_basic()
Does BaseMap pull from CARTO default basemaps?

Returns *True* if using a CARTO basemap (Dark Matter, Positron or Voyager), *False* otherwise.

Return type bool

class layer.**Layer**(*table_name*, *source=None*, *overwrite=False*, *time=None*, *color=None*, *size=None*, *tooltip=None*, *legend=None*)
A cartoframes Data Layer based on a specific table in user's CARTO database. This layer class is used for visualizing individual datasets with *CartoContext.map()*'s `layers` keyword argument.

Example

```
import cartoframes
from cartoframes import QueryLayer, styling
cc = cartoframes.CartoContext(BASEURL, APIKEY)
cc.map(layers=[Layer('fantastic_sql_table',
                     size=7,
                     color={'column': 'mr_fox_sightings',
                            'scheme': styling.prism(10)})])
```

Parameters

- **table_name** (*str*) – Name of table in CARTO account
- **Styling** – See *QueryLayer* for a full list of all arguments arguments for styling this map data layer.
- **source** (*pandas.DataFrame, optional*) – Not currently implemented
- **overwrite** (*bool, optional*) – Not currently implemented

class layer.**QueryLayer**(*query*, *time=None*, *color=None*, *size=None*, *tooltip=None*, *legend=None*)
cartoframes data layer based on an arbitrary query to the user's CARTO database. This layer class is useful for offloading processing to the cloud to do some of the following:

- Visualizing spatial operations using PostGIS and PostgreSQL, which is the database underlying CARTO
- Performing arbitrary relational database queries (e.g., complex JOINs in SQL instead of in pandas)
- Visualizing a subset of the data (e.g., `SELECT * FROM table LIMIT 1000`)

Used in the *layers* keyword in *CartoContext.map()*.

Example

Underlay a QueryLayer with a complex query below a layer from a table. The QueryLayer is colored by the calculated column `abs_diff`, and points are sized by the column `i_measure`.

```
import cartoframes
from cartoframes import QueryLayer, styling
cc = cartoframes.CartoContext(BASEURL, APIKEY)
cc.map(layers=[QueryLayer('''
                          WITH i_cte As (
```

```
                            SELECT
                                ST_Buffer(the_geom::geography, 500)::geometry As␣
↪the_geom,
                                cartodb_id,
                                measure,
                                date
                              FROM interesting_data
                             WHERE date > '2017-04-19'
                           )
                           SELECT
                              i.cartodb_id, i.the_geom,
                              ST_Transform(i.the_geom, 3857) AS the_geom_
↪webmercator,
                              abs(i.measure - j.measure) AS abs_diff,
                              i.measure AS i_measure
                            FROM i_cte AS i
                            JOIN awesome_data AS j
                              ON i.event_id = j.event_id
                           WHERE j.measure IS NOT NULL
                             AND j.date < '2017-04-29'
                           ''',
                           color={'column': 'abs_diff',
                                  'scheme': styling.sunsetDark(7)},
                           size='i_measure'),
              Layer('fantastic_sql_table')])
```

Parameters

- **query** (*str*) – Query to expose data on a map layer. At a minimum, a query needs to have the columns *cartodb_id*, *the_geom*, and *the_geom_webmercator* for the map to display. Read more about queries in CARTO's docs.
- **time** (*dict or str, optional*) – Time-based style to apply to layer.

  If *time* is a `str`, it must be the name of a column which has a data type of *datetime* or *float*.

  If *time* is a `dict`, the following keys are options:

  - column (*str*, required): Column for animating map, which must be of type *datetime* or *float*.
  - method (*str*, optional): Type of aggregation method for operating on Torque Tile-Cubes. Must be one of `avg`, `sum`, or another PostgreSQL aggregate functions with a numeric output. Defaults to `count`.
  - cumulative (*bool*, optional): Whether to accumulate points over time (`True`) or not (`False`, default)
  - frames (*int*, optional): Number of frames in the animation. Defaults to 256.
  - duration (*int*, optional): Number of seconds in the animation. Defaults to 30.
  - trails (*int*, optional): Number of trails after the incidence of a point. Defaults to 2.
- **color** (*dict or str, optional*) – Color style to apply to map. For example, this can be used to change the color of all geometries in this layer, or to create a graduated color or choropleth map.

  If *color* is a `str`, there are two options:

- A column name to style by to create, for example, a choropleth map if working with polygons. The default classification is *quantiles* for quantitative data and *category* for qualitative data.
- A hex value or web color name.

If *color* is a `dict`, the following keys are options, with values described:

- column (*str*): Column used for the basis of styling
- scheme (*dict*, optional): Scheme such as *styling.sunset(7)* from the *styling module* of cartoframes that exposes CARTOColors. Defaults to *mint* scheme for quantitative data and *bold* for qualitative data. More control is given by using *styling.scheme*.

  If you wish to define a custom scheme outside of CARTOColors, it is recommended to use the *styling.custom* utility function.

- **`size`** (*`dict or int, optional`*) – Size style to apply to point data.

  If *size* is an `int`, all points are sized by this value.

  If *size* is a `str`, this value is interpreted as a column, and the points are sized by the value in this column. The classification method defaults to *quantiles*, with a min size of 5, and a max size of 5. Use the `dict` input to override these values.

  If *size* is a `dict`, the follow keys are options, with values described as:

  - column (*str*): Column to base sizing of points on
  - bin_method (str, optional): Quantification method for dividing data range into bins. Must be one of the methods in `BinMethod` (excluding *category*).
  - bins (*int*, optional): Number of bins to break data into. Defaults to 5.
  - max (*int*, optional): Maximum point width (in pixels). Defaults to 25.
  - min (*int*, optional): Minimum point width (in pixels). Defaults to 5.

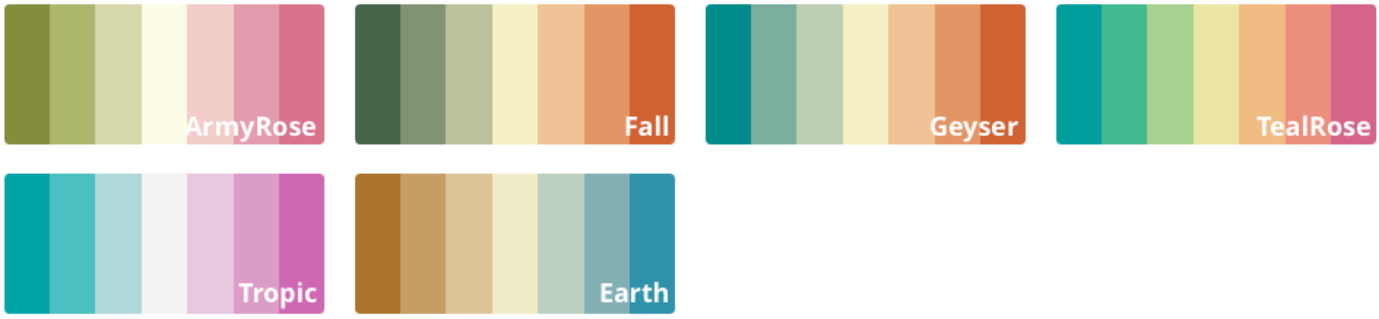
- **`tooltip`** (*`tuple, optional`*) – **Not yet implemented.**
- **`legend`** – **Not yet implemented.**

3.4 Map Styling Functions

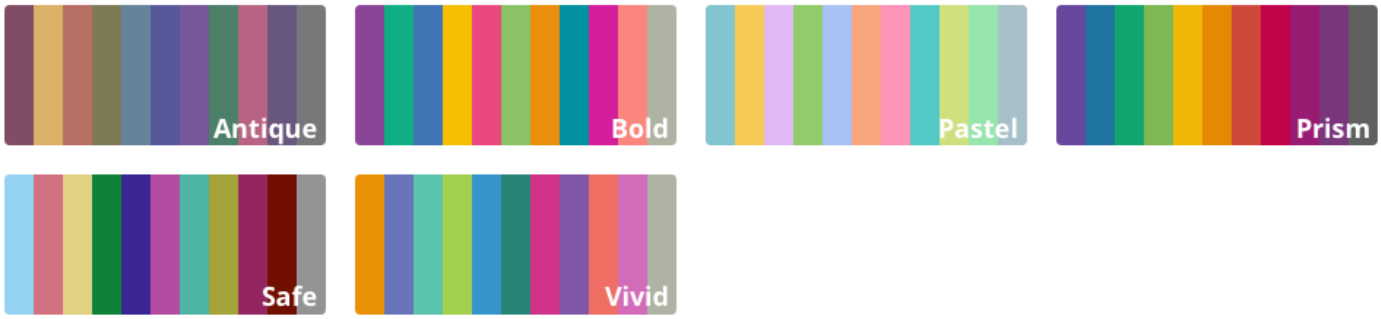
Styling module that exposes CARTOColors schemes. Read more about CARTOColors in its GitHub repository.

Sequential Schemes

Diverging Schemes



Qualitative Schemes



class styling.**BinMethod**
Data classification methods used for the styling of data on maps.

quantiles
str – Quantiles classification for quantitative data

jenks
str – Jenks classification for quantitative data

headtails
str – Head/Tails classification for quantitative data

equal
str – Equal Interval classification for quantitative data

category
str – Category classification for qualitative data

mapping
dict – The TurboCarto mappings

styling.**get_scheme_cartocss**(*column*, *scheme_info*)
Get TurboCARTO CartoCSS based on input parameters

styling.**custom**(*colors*, *bins=None*, *bin_method='quantiles'*)
Create a custom scheme.

Parameters

- **colors** (*list of str*) – List of hex values for styling data
- **bins** (*int, optional*) – Number of bins to style by. If not given, the number of colors will be used.
- **bin_method** (*str, optional*) – Classification method. One of the values in *BinMethod*. Defaults to *quantiles*, which only works with quantitative data.

styling.**scheme**(*name*, *bins*, *bin_method*)
Return a custom scheme based on CARTOColors.

Parameters

- **name** (*str*) – Name of a CARTOColor.
- **bins** (*int or iterable*) – If an *int*, the number of bins for classifying data. CARTOColors have 7 bins max for quantitative data, and 11 max for qualitative data. If *bins* is a *list*, it is the upper range for classifying data. E.g., *bins* can be of the form `(10, 20, 30, 40, 50)`.
- **bin_method** (*str*) – One of methods in *BinMethod*.

Warning: Input types are particularly sensitive in this function, and little feedback is given for errors. `name` and `bin_method` arguments are case-sensitive.

styling.**burg**(*bins*, *bin_method='quantiles'*)
CARTOColors Burg quantitative scheme

styling.**burgYl**(*bins*, *bin_method='quantiles'*)
CARTOColors BurgYl quantitative scheme

styling.**redOr**(*bins*, *bin_method='quantiles'*)
CARTOColors RedOr quantitative scheme

styling.**orYel**(*bins*, *bin_method='quantiles'*)
CARTOColors OrYel quantitative scheme

styling.**peach**(*bins*, *bin_method='quantiles'*)
CARTOColors Peach quantitative scheme

styling.**pinkYl**(*bins*, *bin_method='quantiles'*)
CARTOColors PinkYl quantitative scheme

styling.**mint**(*bins*, *bin_method='quantiles'*)
CARTOColors Mint quantitative scheme

styling.**bluGrn**(*bins*, *bin_method='quantiles'*)
CARTOColors BluGrn quantitative scheme

styling.**darkMint**(*bins*, *bin_method='quantiles'*)
CARTOColors DarkMint quantitative scheme

styling.**emrld**(*bins*, *bin_method='quantiles'*)
CARTOColors Emrld quantitative scheme

styling.**bluYl**(*bins*, *bin_method='quantiles'*)
CARTOColors BluYl quantitative scheme

styling.**teal**(*bins*, *bin_method='quantiles'*)
CARTOColors Teal quantitative scheme

styling.**tealGrn**(*bins*, *bin_method='quantiles'*)
CARTOColors TealGrn quantitative scheme

styling.**purp**(*bins*, *bin_method='quantiles'*)
CARTOColors Purp quantitative scheme

styling.**purpOr**(*bins*, *bin_method='quantiles'*)
CARTOColors PurpOr quantitative scheme

styling.**sunset**(*bins*, *bin_method='quantiles'*)
CARTOColors Sunset quantitative scheme

styling.**magenta**(*bins*, *bin_method='quantiles'*)
CARTOColors Magenta quantitative scheme

styling.**sunsetDark**(*bins*, *bin_method='quantiles'*)
CARTOColors SunsetDark quantitative scheme

styling.**brwnYl**(*bins*, *bin_method='quantiles'*)
CARTOColors BrwnYl quantitative scheme

styling.**armyRose**(*bins*, *bin_method='quantiles'*)
CARTOColors ArmyRose divergent quantitative scheme

styling.**fall**(*bins*, *bin_method='quantiles'*)
CARTOColors Fall divergent quantitative scheme

styling.**geyser**(*bins*, *bin_method='quantiles'*)
CARTOColors Geyser divergent quantitative scheme

styling.**temps**(*bins*, *bin_method='quantiles'*)
CARTOColors Temps divergent quantitative scheme

styling.**tealRose**(*bins*, *bin_method='quantiles'*)
CARTOColors TealRose divergent quantitative scheme

styling.**tropic**(*bins*, *bin_method='quantiles'*)
CARTOColors Tropic divergent quantitative scheme

styling.**earth**(*bins*, *bin_method='quantiles'*)
CARTOColors Earth divergent quantitative scheme

styling.**antique**(*bins*, *bin_method='category'*)
CARTOColors Antique qualitative scheme

styling.**bold**(*bins*, *bin_method='category'*)
CARTOColors Bold qualitative scheme

styling.**pastel**(*bins*, *bin_method='category'*)
CARTOColors Pastel qualitative scheme

styling.**prism**(*bins*, *bin_method='category'*)
CARTOColors Prism qualitative scheme

styling.**safe**(*bins*, *bin_method='category'*)
CARTOColors Safe qualitative scheme

styling.**vivid**(*bins*, *bin_method='category'*)
CARTOColors Vivid qualitative scheme

3.5 BatchJobStatus

class context.**BatchJobStatus**(*carto_context*, *job*)
Status of a write or query operation. Read more at Batch SQL API docs about responses and how to interpret them.

Example

Poll for a job's status if you've caught the *BatchJobStatus* instance.

```
import time
job = cc.write(df, 'new_table',
               lnglat=('lng_col', 'lat_col'))
while True:
    curr_status = job.status()['status']
    if curr_status in ('done', 'failed', 'canceled', 'unknown', ):
        print(curr_status)
        break
    time.sleep(5)
```

Create a *BatchJobStatus* instance if you have a *job_id* output from a *cc.write* operation.

```
>>> from cartoframes import CartoContext, BatchJobStatus
>>> cc = CartoContext(username='...', api_key='...')
>>> cc.write(df, 'new_table', lnglat=('lng', 'lat'))
'BatchJobStatus(job_id='job-id-string', ...)'
>>> batch_job = BatchJobStatus(cc, 'job-id-string')
```

Attrs: job_id (str): Job ID of the Batch SQL API job last_status (str): Status of `job_id` job when last polled created_at (str): Time and date when job was created

Parameters

- **carto_context** (*carto.CartoContext*) – CartoContext instance
- **job** (*dict or str*) – If a dict, job status dict returned after sending a Batch SQL API request. If str, a Batch SQL API job id.

get_status()
return current status of job

status()
Checks the current status of job `job_id`

Returns Status and time it was updated

Return type dict

Warns **UserWarning** – If the job failed, a warning is raised with information about the failure

3.6 Credentials Management

Credentials management for cartoframes usage.

class credentials.**Credentials**(*creds=None*, *key=None*, *username=None*, *base_url=None*, *cred_file=None*)
Credentials class for managing and storing user CARTO credentials. The arguments are listed in order of precedence: *Credentials* instances are first, *key* and *base_url*/*username* are taken next, and *config_file* (if given) is taken last. If no arguments are passed, then there will be an attempt to retrieve credentials from a previously saved session. One of the above scenarios needs to be met to successfully instantiate a *Credentials* object.

Parameters

- **creds** (cartoframes.Credentials, optional) – Credentials instance
- **key** (*str, optional*) – API key of user's CARTO account
- **username** (*str, optional*) – Username of CARTO account
- **base_url** (*str, optional*) – Base URL used for API calls. This is usually of the form *https://eschbacher.carto.com/* for user *eschbacher*. On premises installations (and others) have a different URL pattern.
- **cred_file** (*str, optional*) – Pull credentials from a stored file. If this and all other args are not entered, Credentials will attempt to load a user config credentials file that was previously set with Credentials(...).save().

Raises RuntimeError – If not enough credential information is passed and no stored credentials file is found, this error will be raised.

Example

```
from cartoframes import Credentials, CartoContext
creds = Credentials(key='abcdefg', username='eschbacher')
cc = CartoContext(creds=creds)
```

base_url (*base_url=None*)
Return or set *base_url*.

Parameters **base_url** (*str, optional*) – If set, updates the *base_url*. Otherwise returns current *base_url*.

Note: This does not update the *username* attribute. Separately update the username with `Credentials.username` or update *base_url* and *username* at the same time with `Credentials.set`.

Example

```
>>> from cartoframes import Credentials
# load credentials saved in previous session
>>> creds = Credentials()
# returns current base_url
>>> creds.base_url()
'https://eschbacher.carto.com/'
# updates base_url with new value
>>> creds.base_url('new_base_url')
```

delete (*config_file=None*)
Deletes the credentials file specified in *config_file*. If no file is specified, it deletes the default user credential file.

Parameters **config_file** (*str*) – Path to configuration file. Defaults to delete the user default location if *None*.

Tip: To see if there is a default user credential file stored, do the following:

```
>>> creds = Credentials()
>>> print(creds)
Credentials(username=eschbacher, key=abcdefg,
        base_url=https://eschbacher.carto.com/)
```

---

key (*key=None*)
Return or set API *key*.

Parameters **key** (*str, optional*) – If set, updates the API key, otherwise returns current API key.

Example

```
>>> from cartoframes import Credentials
# load credentials saved in previous session
>>> creds = Credentials()
# returns current API key
>>> creds.key()
'abcdefg'
# updates API key with new value
>>> creds.key('new_api_key')
```

save(*config_loc=None*)
Saves current user credentials to user directory.

Parameters **config_loc** (*str, optional*) – Location where credentials are to be stored. If no argument is provided, it will be send to the default location.

Example

```
from cartoframes import Credentials
creds = Credentials(username='eschbacher', key='abcdefg')
creds.save()  # save to default location
```

set(*key=None*, *username=None*, *base_url=None*)
Update the credentials of a Credentials instance instead with new values.

Parameters

- **key** (*str*) – API key of user account. Defaults to previous value if not specified.
- **username** (*str*) – User name of account. This parameter is optional if *base_url* is not specified, but defaults to the previous value if not set.
- **base_url** (*str*) – Base URL of user account. This parameter is optional if *username* is specified and on CARTO's cloud-based account. Generally of the form `https://your_user_name.carto.com/` for cloud-based accounts. If on-prem or otherwise, contact your admin.

Example

```
from cartoframes import Credentials
# load credentials saved in previous session
creds = Credentials()
# set new API key
creds.set(key='new_api_key')
# save new creds to default user config directory
creds.save()
```

Note: If the *username* is specified but the *base_url* is not, the *base_url* will be updated to `https://<username>.carto.com/`.

username(*username=None*)
Return or set *username*.

Parameters **username** (*str, optional*) – If set, updates the *username*. Otherwise returns current *username*.

Note: This does not update the *base_url* attribute. Use *Credentials.set* to have that updated with *username*.

Example

```
>>> from cartoframes import Credentials
# load credentials saved in previous session
>>> creds = Credentials()
# returns current username
>>> creds.username()
'eschbacher'
# updates username with new value
>>> creds.username('new_username')
```

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Version 0.5.3

Python Module Index

Index