djio Documentation

Release 0.0.5

Pat Daburu

Feb 25, 2018

Contents:

CHAPTER 1

API Documentation

1.1 djio.errors

Did something go wrong? It did? Use something from this module to deal with it.

exception djio.errors.**DjioException**(*message: str*, *inner: Exception = None*)

Bases: Exception

This is a common base class for all custom djio exceptions.

__init__(*message: str*, *inner: Exception = None*)

Parameters

- **message** – the exception message
- **inner** – the exception that caused this exception

inner

Get the inner exception that caused this exception. :return: the inner exception

message

Get the exception message. :return: the exception message

1.2 djio.geometry

Working with geometries? Need help? Here it is!

class djio.geometry.**Envelope**(*min_x: float*, *min_y: float*, *max_x: float*, *max_y: float*, *spatial_reference: djio.geometry.SpatialReference*)

Bases: *djio.geometry.Polygon*

An envelope represents the minimum bounding rectangle (minimum x and y values, along with maximum x and y values) defined by coordinate pairs of a geometry. All coordinates for the geometry fall within the envelope.

__init__(*min_x: float*, *min_y: float*, *max_x: float*, *max_y: float*, *spatial_reference: djio.geometry.SpatialReference*)

Parameters

- **min_x** – the minimum X coordinate
- **min_y** – the minimum Y coordinate
- **max_x** – the maximum X coordinate
- **max_y** – the maximum Y coordinate
- **spatial_reference** – the spatial reference (or spatial reference ID) in which the coordinates are expressed

class djio.geometry.**Geometry**(*shapely_geometry: <MagicMock id='140264541303920'>, spatial_reference: djio.geometry.SpatialReference = None*)

Bases: object

This is the common base class for all of the geometry types.

__init__(*shapely_geometry: <MagicMock id='140264541303920'>, spatial_reference: djio.geometry.SpatialReference = None*)

Parameters

- **shapely_geometry** – a Shapely geometry
- **spatial_reference** – the geometry's spatial reference

dimensions

How many dimensions does this geometry occupy? For example: a point is zero-dimensional (0); a line is one-dimensional (1); and a polygon is two-dimensional (2).

Returns the dimensionality of the geometry

djiohash()

Get this geometry's hash value.

Returns the hash value

envelope

Get the envelope (bounding box) of the geometry.

Returns the geometry's envelope

flip_coordinates() → djio.geometry.Geometry

Create a geometry based on this one, but with the X and Y axis reversed.

Returns a new *Geometry* with reversed ordinals.

static from_ewkt() → djio.geometry.Geometry

Create a geometry from EWKT, a PostGIS-specific format that includes the spatial reference system identifier an up to four (4) ordinate values (XYZM). For example: SRID=4326;POINT(-44.3 60.1) to locate a longitude/latitude coordinate using the WGS 84 reference coordinate system.

Parameters **ewkt** – the extended well-known text (EWKT)

Returns the geometry

static from_geoalchemy2(*spatial_reference: djio.geometry.SpatialReference*) → djio.geometry.Geometry

static from_gml() → djio.geometry.Geometry

static from_ogr(*spatial_reference: djio.geometry.SpatialReference = None*) → djio.geometry.Geometry

Create a djio geometry from an OGR geometry.

Parameters

- **ogr_geom** – the OGR geometry
- **spatial_reference** – the spatial reference

Returns a djio geometry based on the OGR geometry

Raises ***GeometryException*** – if the OGR has no spatial reference and no spatial reference is supplied

static from_shapely(*spatial_reference: djio.geometry.SpatialReference*) → djio.geometry.Geometry

Create a new geometry based on a Shapely `BaseGeometry`.

Parameters

- **shapely_geometry** – the Shapely base geometry
- **spatial_reference** – the spatial reference (or spatial reference ID)

Returns the new geometry

Seealso *Point*

Seealso *Polyline*

Seealso *Polygon*

static from_wkb(*spatial_reference: djio.geometry.SpatialReference*) → djio.geometry.Geometry

static from_wkt(*spatial_reference: djio.geometry.SpatialReference*) → djio.geometry.Geometry

geometry_type

Get this geometry's type.

Returns the geometry's type

get_point_tuples() → Iterable[djio.geometry.PointTuple]

Get an ordered iteration of all the coordinates in the geometry as point tuples. :return: an enumeration of point tuples

iter_coords() → Iterable[Tuple[float, float]]

Retrieve the coordinates that define this geometry as a flattened, ordered iteration.

Returns and ordered iteration of tuples that describe the geometry's coordinates

project(*preferred_spatial_reference: djio.geometry.SpatialReference = None, fallback_spatial_reference: djio.geometry.SpatialReference = 3857*) → djio.geometry.Geometry

Project (or re-project) this geometry.

Parameters

- **preferred_spatial_reference** – the preferred spatial reference
- **fallback_spatial_reference** – a spatial reference that may be used as a "fall-back" if the preferred spatial reference is not provided and a suitable projected spatial reference system isn't available

Returns a new, projected, geometry

See also:

Projector

representative_point

shapely_geometry

Get the Shapely geometry underlying this geometry object.

Returns the Shapely geometry

spatial_reference
Get the geometry's spatial reference.

Returns the geometry's spatial reference

to_gml(*version: int = 3*) → str
Export the geometry to GML.

Parameters **version** – the desired GML version

Returns the GML representation of the geometry

to_ogr
Get the OGR geometry equivalent of this geometry.

Returns the OGR geometry equivalent

transform(*spatial_reference: djio.geometry.SpatialReference*) → djio.geometry.Geometry
Create a new geometry based on this geometry but in another spatial reference.

Parameters **spatial_reference** – the target spatial reference

Returns the new transformed geometry

transform_to_utm() → djio.geometry.Geometry
Transform this geometry to an appropriate UTM coordinate system based on its location.

Returns the new geometry

exception djio.geometry.**GeometryException**(*message: str*, *inner: Exception = None*)
Bases: *djio.errors.DjioException*

Raised when something goes wrong with a geometry.

class djio.geometry.**GeometryType**
Bases: enum.IntFlag

These are the supported geometric data types.

POINT = 1

POLYGON = 4

POLYLINE = 2

UNKNOWN = 0

class djio.geometry.**LatLonTuple**
Bases: tuple

This is a lightweight tuple that represents a specific latitude and longitude

latitude
Alias for field number 0

longitude
Alias for field number 1

class djio.geometry.**LateralSides**
Bases: enum.Enum

This is a simple enumeration that identifies the lateral side of line (left or right).

LEFT = 'left'
the left side of the line

RIGHT = 'right'
the right side of the line

class djio.geometry.**Point**(*shapely_geometry: <MagicMock id='140264541405312'>, spatial_reference: djio.geometry.SpatialReference = None*)
Bases: `djio.geometry.Geometry`

In modern mathematics, a point refers usually to an element of some set called a space. More specifically, in Euclidean geometry, a point is a primitive notion upon which the geometry is built, meaning that a point cannot be defined in terms of previously defined objects. That is, a point is defined only by some properties, called axioms, that it must satisfy. In particular, the geometric points do not have any length, area, volume or any other dimensional attribute. A common interpretation is that the concept of a point is meant to capture the notion of a unique location in Euclidean space.

__init__(*shapely_geometry: <MagicMock id='140264541405312'>, spatial_reference: djio.geometry.SpatialReference = None*)

Parameters

- **shapely_geometry** – a Shapely geometry
- **spatial_reference** – the geometry's spatial reference

dimensions
A point is zero-dimensional (0) :return: zero (0)

flip_coordinates() → djio.geometry.Point
Create a point based on this one, but with the X and Y axis reversed.

Returns a new `Geometry` with reversed ordinals.

static from_coordinates(*y: float, spatial_reference: djio.geometry.SpatialReference, z: Union[float, NoneType] = None*)
Create a point from its coordinates.

Parameters

- **x** – the X coordinate
- **y** – the Y coordinate
- **spatial_reference** – the spatial reference (or spatial reference ID)
- **z** – the Z coordinate

Returns the new `Point`

static from_lat_lon(*longitude: float*) → djio.geometry.Point
Create a geometry from a set of latitude, longitude coordinates.

Parameters

- **latitude** – the latitude
- **longitude** – the longitude

Returns `Point`

static from_latlon_tuple() → djio.geometry.Point
Create a point from a latitude/longitude tuple.

Parameters latlon_tuple – the latitude/longitude tuple

Returns the new point

static from_point_tuple() → djio.geometry.Point
Create a point from a point tuple.

Parameters **point_tuple** – the point tuple

Returns the new point

static from_shapely(*srid: int*) → djio.geometry.Point
Create a new point based on a Shapely point.

Parameters

- **shapely** – the Shapely point
- **srid** – the spatial reference ID

Returns the new geometry

Seealso `Geometry.from_shapely()`

geometry_type
Get the geometry type.

Returns `GeometryType.POINT`

iter_coords() → Iterable[Tuple[float, float]]
Retrieve the coordinates that define this geometry. For a `Point`, the iteration shall contain a single set of coordinates. :return: an iteration containing a single tuple containing this point's coordinates

to_latlon_tuple() → djio.geometry.LatLonTuple
Get a lightweight latitude/longitude tuple representation of this point.

Returns the latitude/longitude tuple representation of this point

to_point_tuple() → djio.geometry.PointTuple
Get a lightweight tuple representation of this point.

Returns the tuple representation of the point

x
Get the X coordinate.

Returns the X coordinate

y
Get the Y coordinate.

Returns the Y coordinate

z
Get the Z coordinate.

Returns the Z coordinate

class djio.geometry.**PointTuple**
Bases: `tuple`

This is a lightweight tuple that represents a point.

srid
Alias for field number 3

x
Alias for field number 0

y
Alias for field number 1

z
Alias for field number 2

class djio.geometry.**Polygon**(*shapely_geometry: <MagicMock id='140264541397344'>, spatial_reference: djio.geometry.SpatialReference = None*)
Bases: `djio.geometry.Geometry`

In elementary geometry, a polygon is a plane figure that is bounded by a finite chain of straight line segments closing in a loop to form a closed polygonal chain or circuit. These segments are called its edges or sides, and the points where two edges meet are the polygon's vertices (singular: vertex) or corners. The interior of the polygon is sometimes called its body.

__init__(*shapely_geometry: <MagicMock id='140264541397344'>, spatial_reference: djio.geometry.SpatialReference = None*)

Parameters

- **shapely_geometry** – a Shapely geometry
- **spatial_reference** – the geometry's spatial reference

dimensions
A polygon is two-dimensional (2). :return: two (2)

geometry_type
Get the geometry type.

Returns `GeometryType.POLYGON`

get_area(*spatial_reference: Union[djio.geometry.SpatialReference, NoneType] = None*) → <MagicMock id='140264541399360'>

iter_coords() → Iterable[Tuple[float, float]]
Retrieve the polygon's coordinates as a flattened enumeration. :return: an iteration containing the polyline's coordinates

class djio.geometry.**Polyline**(*shapely_geometry: <MagicMock id='140264565205312'>, spatial_reference: djio.geometry.SpatialReference = None*)
Bases: `djio.geometry.Geometry`

In geometry, a polygonal chain is a connected series of line segments. More formally, a polygonal chain P is a curve specified by a sequence of points (A1 , A2, . . . , An) called its vertices. The curve itself consists of the line segments connecting the consecutive vertices. A polygonal chain may also be called a polygonal curve, polygonal path, polyline, piecewise linear curve, broken line or, in geographic information systems (that's us), a linestring or linear ring.

__init__(*shapely_geometry: <MagicMock id='140264565205312'>, spatial_reference: djio.geometry.SpatialReference = None*)

Parameters

- **shapely_geometry** – a Shapely geometry
- **spatial_reference** – the geometry's spatial reference

dimensions
A polyline is one-dimensional (1) :return: one (1)

geometry_type
Get the geometry type.

Returns `GeometryType.POLYLINE`

iter_coords() → Iterable[Tuple[float, float]]
Retrieve the coordinates that define this line. :return: an iteration containing the polyline's coordinates

class djio.geometry.**Projector**
Bases: `object`

Use a projector to get a projected version of a geographic geometry, or to re-project a projected geometry.

static get_instance() → djio.geometry.Projector
Get the shared projector instance. :return: the shared projector instance

static project(*preferred_spatial_reference: djio.geometry.SpatialReference = None, fallback_spatial_reference: djio.geometry.SpatialReference = 3857*) → djio.geometry.Geometry
Project a geometry. :param geometry: the original geometry :param preferred_spatial_reference: the preferred spatial reference (If no preferred spatial reference is supplied, the projector will attempt to select an appropriate metric projection.) :param fallback_spatial_reference: the fallback spatial reference (if your preferred spatial reference isn't available) :return: the projected geometry

static set_instance()
Set the shared projector instance. :param projector: the shared projector

class djio.geometry.**ProtoGeometry**(*spatial_reference: djio.geometry.SpatialReference = 4326, projector: djio.geometry.Projector = None*)
Bases: object

Use a proto-geometry build up a new geometry from individual coordinates.

__init__(*spatial_reference: djio.geometry.SpatialReference = 4326, projector: djio.geometry.Projector = None*)
Initialize self. See help(type(self)) for accurate signature.

add(*p: djio.geometry.Point*)
Add a point to the prototype's exterior :param p: the new coordinate you want to add

clear()
Clear the current contents.

to_polygon() → djio.geometry.Polygon
Create a Polygon from the contents of this proto-geometry. :return: the Polygon

to_polyline() → djio.geometry.Polyline
Create a Polyline from the contents of this proto-geometry. :return: the Polyline

class djio.geometry.**SpatialReference**(*srid: int*)
Bases: object

A spatial reference system (SRS) or coordinate reference system (CRS) is a coordinate-based local, regional or global system used to locate geographical entities. A spatial reference system defines a specific map projection, as well as transformations between different spatial reference systems.

See also:

https://en.wikipedia.org/wiki/Spatial_reference_system

__init__(*srid: int*)

Parameters srid – the well-known spatial reference ID

static from_srid() → djio.geometry.SpatialReference

static get_utm_for_zone() → djio.geometry.SpatialReference
Get the UTM (Universal Trans-Mercator) spatial reference for a given zone. :param zone: the UTM zone :return: the UTM spatial reference :raises SpatialReferenceException: if the UTM zone has no supported spatial reference

static get_utm_from_longitude() → djio.geometry.SpatialReference
Get the UTM (Universal Trans-Mercator) spatial reference for a given longitude. :param longitude: the longitude :return: the UTM spatial reference :raises SpatialReferenceException: if the UTM zone has no supported spatial reference

is_geographic
Is this spatial reference geographic?

Returns *true* if this is a geographic spatial reference, otherwise *false*

is_metric
Is this a projected spatial reference system that measures linear units in single meters?

Returns *true* if this is a projected spatial reference system that measures linear units in single meters

is_projected
Is this spatial reference projected?

Returns *true* if this is a projected spatial reference, otherwise *false*

is_same_as (*other: djio.geometry.SpatialReference*) → bool
Test a spatial reference or SRID to see if it represents this spatial reference. :param other: the other spatial reference (or SRID) :return: `True` if the other parameter represents the same spatial reference, otherwise `False`

is_utm

ogr_sr
Get the OGR spatial reference.

Returns the OGR spatial reference

srid
Get the spatial reference's well-known ID (srid).

Returns the well-known spatial reference ID

utm_zone
Get the UTM (Universal Trans-Mercator) zone associated with this spatial reference. :return: the associated UTM zone

exception djio.geometry.**SpatialReferenceException** (*message: str*, *inner: Exception = None*)
Bases: *djio.errors.DjioException*

Raised when something goes wrong with a spatial reference.

1.3 djio.hashing

Need to know at a glance if two geometries are the same?

djio.hashing.**bytes_to_int** (*b: bytes*)
Convert a byte array to an integer.

Parameters **b** – the byte array

Returns the integer

djio.hashing.**djiohash_v1** (*geometry_type_code: int*, *srid: int*, *coordinates: Iterable[Tuple[float, float]]*, *precision: int = 4*) → bytearray
This is the first version of the djio hashing algorithm.

Parameters

- **geometry_type_code** – an integer indicating the type of the geometry
- **srid** – the numeric spatial reference ID

- **coordinates** – a flattened, ordered iteration of coordinates in the geometry expressed as tuples
- **precision** – the maximum precision (points behind decimal places) to consider in the supplied coordinates

Returns a hash value for the geometry

See also:

djio.geometry.GeometryType

djio.hashing.**int_to_bytes**(*i: int*, *width: int*, *unsigned: bool = False*, *as_iterable: bool = False*) → bytearray

Convert an integer to a fixed-width byte array.

Parameters

- **i** – the integer
- **width** – the number of bytes in the array
- **unsigned** – *True* if the sign of the *int* may be disregarded, otherwise *False*
- **as_iterable** – when *True* the function returns a list of byte-sized *int's instead of a 'bytearray*.

Returns the byte array

CHAPTER 2

Python Module Dependencies

The `requirements.txt` file contains this project's module dependencies. You can install these dependencies using `pip`.

```
pip install -r requirements.txt
```

2.1 requirements.txt

```
alabaster>=0.7.10,<1
Babel>=2.5.3,<3
CaseInsensitiveDict>=1.0.0,<2
certifi>=2018.1.18
chardet==3.0.4
docutils==0.14
GDAL>=2.1.0,<3
GeoAlchemy2>=0.4.0,<1
idna==2.6
imagesize>=0.7.1
Jinja2>=2.10,<3
MarkupSafe==1.0
measurement>=1.8.0,<2
mock>=2.0.0,<3
numpy>=1.13.3,<2
Pygments==2.2.0
pyparsing>=2.2.0
pytz>=2018.3
pytest>=3.4.0,<4
pytest-cov>=2.5.1,<3
pytest-pythonpath>=0.7.2,<1
requests==2.18.4
setuptools>=38.4.0
Shapely>=1.6.1,<2
```

(continues on next page)

(continued from previous page)

```
six==1.11.0
snowballstemmer==1.2.1
Sphinx>=1.7.0,<2
sphinx-rtd-theme==0.2.4
sphinxcontrib-websupport==1.0.1
SQLAlchemy>=1.1.14,<2
sympy==0.7.6.1
urllib3==1.22
```

CHAPTER 3

Development

djio is in the early stages of development.

3.1 How To. . .

3.1.1 How To Install GDAL/OGR Packages on Ubuntu

GDAL is a translator library for raster and vector geospatial data formats.

OGR Simple Features Library is a C++ open source library (and commandline tools) providing read (and sometimes write) access to a variety of vector file formats including ESRI Shapefiles, S-57, SDTS, PostGIS, Oracle Spatial, and Mapinfo mid/mif and TAB formats.

OGR is a part of the GDAL library.

GDAL/OGR are used in numerous GIS software projects and, lucky for us, there are bindings for python. In fact, you may want to check out the Python GDAL/OGR Cookbook.

This article describes a process you can follow to install GDAL/OGR on Ubuntu.

Before You Begin: Python 3.6

If you are installing the GDAL/OGR packages into a virtual environment based on Python 3.6, you may need to install the python3.6-dev package.

```
sudo apt-get install python3.6-dev
```

For more information about creating virtual environments on Ubuntu 16.04 LTS, see *A Note About Python 3.6 and Ubuntu 16.04 LTS*.

Install GDAL/OGR

Much of this section is taken from a really helpful blog post by Sara Safavi. Follow these steps to get GDAL/OGR installed.

To get the latest GDAL/OGR version, add the PPA to your sources, then install the gdal-bin package (this should automatically grab any necessary dependencies, including at least the relevant libgdal version).

```
sudo add-apt-repository ppa:ubuntugis/ppa && sudo apt-get update
```

Once you add the repository, go ahead and update your source packages.

```
sudo apt-get update
```

Now you should be able to install the GDAL/OGR package.

```
sudo apt-get install gdal-bin
```

To verify the installation, you can run `ogrinfo --version`.

```
ogrinfo --version
```

Install GDAL for Python

Before installing the GDAL Python libraries, you'll need to install the GDAL development libraries.

```
sudo apt-get install libgdal-dev
```

You'll also need to `export` a couple of environment variables for the compiler.

```
export CPLUS_INCLUDE_PATH=/usr/include/gdal
export C_INCLUDE_PATH=/usr/include/gdal
```

Now you can use `pip` to install the Python GDAL bindings.

```
pip install GDAL
```

Putting It All Together

If you want to run the whole process at once, we've collected all the commands above in the script below.

```
#!/usr/bin/env bash

sudo add-apt-repository ppa:ubuntugis/ppa && sudo apt-get update
sudo apt-get update
sudo apt-get install gdal-bin
sudo apt-get install libgdal-dev
export CPLUS_INCLUDE_PATH=/usr/include/gdal
export C_INCLUDE_PATH=/usr/include/gdal
pip install GDAL
```

Try It Out

Now that GDAL/OGR is installed, and you can program against it in Python, why not try it out? The code block below is a sample from the Python OGR/GDAL Cookbook that gets all the layers in an Esri file geodatabase.

```
# standard imports
import sys

# import OGR
from osgeo import ogr

# use OGR specific exceptions
ogr.UseExceptions()

# get the driver
driver = ogr.GetDriverByName("OpenFileGDB")

# opening the FileGDB
try:
    gdb = driver.Open(gdb_path, 0)
except Exception, e:
    print e
    sys.exit()

# list to store layers'names
featsClassList = []

# parsing layers by index
for featsClass_idx in range(gdb.GetLayerCount()):
    featsClass = gdb.GetLayerByIndex(featsClass_idx)
    featsClassList.append(featsClass.GetName())

# sorting
featsClassList.sort()

# printing
for featsClass in featsClassList:
    print featsClass

# clean close
del gdb
```

Acknowledgements

Thanks to Sara Safavi and Paul Whipp for contributing some of the leg work on this.

3.1.2 How To Set Up a Virtual Python Environment (Linux)

`virtualenv` is a tool to create isolated Python environments. You can read more about it in the Virtualenv documentation. This article provides a quick summary to help you set up and use a virtual environment.

A Note About Python 3.6 and Ubuntu 16.04 LTS

If you're running Ubuntu 16.04 LTS (or and earlier version), Python 3.5 is likely installed by default. *Don't remove it!* To get Python 3.6, follow the instructions in this section.

Add the PPA

Run the following command to add the Python 3.6 PPA.

```
sudo add-apt-repository ppa:jonathonf/python-3.6
```

Check for Updates and Install

Check for updates and install Python 3.6 via the following commands.

```
sudo apt-get update
sudo apt-get install python3.6
```

Now you have three Python version, use `python` to run version 2.7, `python3` for version 3.5, and `python3.6` for version 3.6.

For more information on this subject, check out Ji m's article How to Install Python 3.6.1 in Ubuntu 16.04 LTS.

Create a Virtual Python Environment

`cd` to your project directory and run `virtualenv` to create the new virtual environment.

The following commands will create a new virtual environment under my-project/my-venv.

```
cd my-project
virtualenv --python python3.6 venv
```

Activate the Environment

Now that we have a virtual environment, we need to activate it.

```
source venv/bin/activate
```

After you activate the environment, your command prompt will be modified to reflect the change.

Add Libraries and Create a *requirements.txt* File

After you activate the virtual environment, you can add packages to it using `pip`. You can also create a description of your dependencies using `pip`.

The following command creates a file called `requirements.txt` that enumerates the installed packages.

```
pip freeze > requirements.txt
```

This file can then be used by collaborators to update virtual environments using the following command.

```
pip install -r requirements.txt
```

Deactivate the Environment

To return to normal system settings, use the `deactivate` command.

```
deactivate
```

After you issue this command, you'll notice that the command prompt returns to normal.

Acknowledgments

Much of this article is taken from The Hitchhiker's Guide to Python. Go buy a copy right now.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

d

Index

Symbols

A

B

C

D

E

F

G

I

L

M

O

P

R

S

T

U

X

Y

Z